Universitat de Girona

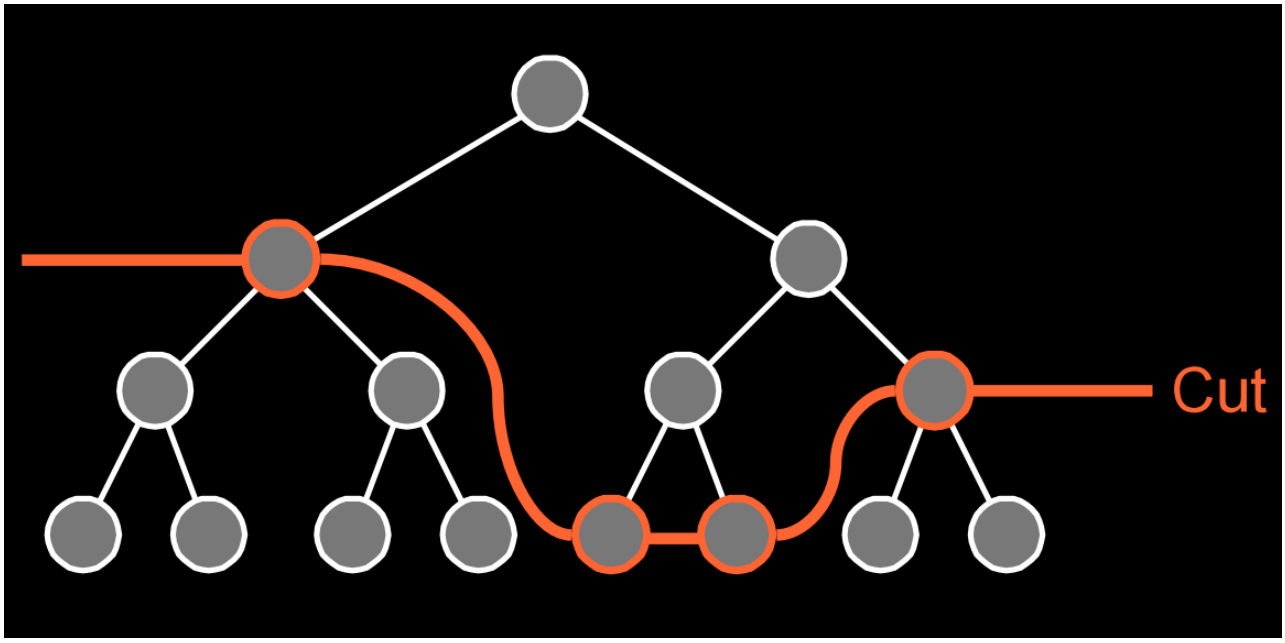
Escola Politècnica Superior

Universitat de Girona

Implementació de la tècnica Lightcuts a l'Unreal Engine 4

Projecte/Treball Fi de Grau

Grau en Enginyeria Informàtica. Pla 2011



Document: Memòria

Autor: Fabià Figueras Palomeras

Director: Gustavo Patow

Departament: Informàtica, Matemàtica Aplicada i Estadística

Àrea: LSI

Convocatoria (mes/any): 09/2015

Índex

Capítol 1

Introducció, motivacions, propòsit i objectius del projecte

La indústria dels videojocs és una de les més importants en el sector de l'entreteniment avui dia gràcies al gran creixement que ha tingut en els darrers anys. Arribats a un punt on els videojocs obtenen més beneficis a Espanya que el cinema i la música junts, cal tenir en compte aquest grup i per tant donar-li la importància que té. Estan aconseguint, cada vegada més, moure masses i atraure més i més gent cada dia. La millora i expansió dels E-Sports també és un fet a tenir en compte, tenint finals de competicions de videojocs amb més espectadors que alguns dels esdeveniments esportius més importants.

De la mateixa manera, els greuges i atacs contra els videojocs també han augmentat. No s'ha demostrat mai empíricament que els vidoejocs augmentin la violència, però sempre s'ha intentat fer creure. Si és cert, que no tot és tan simple i que s'han donat casos de violència en que els vidoejocs hi han estat involucrats. Tot això només serveix per mostrar-nos el gran món que engloba aquest sector, sector que s'ha de tenir en compte pel nombre de seguidors que aconsegueix, pel gran creixement que té en aquest moment i pels continguts de gran qualitat que estem obtenint els consumidors.

La part més important de tot això, és la feina de les persones que hi ha a darrera, la visió de les històries que ens presenten de forma interactiva. Videojocs que ens expliquen històries commocionadores i altres que simplement ens permeten entretenir-nos una estona sense pensar gaire. Sigui quin sigui l'estil, la feina que hi ha a darrera és impressionant, grans equips de persones treballant junts per aconseguir un objectiu. Finalment, crec que fa falta conèixer aquest món més amagat, com es fa la creació i quin és el procés, ja que no es pot crear un bon videojoc sense les persones i eines adequades.

1.1 Motivacions

Les motivacions per a fer un treball normalment acostumen a ser personals i/o professionals. En aquest cas es tenen en compte les dues raons ja que els videojocs han format part de la meva vida des de fa molt temps, donant un gran nombre d'hores de diversió i entreteniment. Així mateix, sempre m'ha agradat saber com es creen aquests continguts, saber el funcionament i la feina que hi ha a darrera d'una gran producció. Per això mateix m'interessava endinsar-me en un treball on pogués obtenir experiència de cares a un possible futur professional.

Entendre com es crea un videojoc és important, però saber com funcionen les eines que hi ha darrera la creació pot ser-ho més. Saber els diferents processos que intervenen en el fun-

cionament del joc pot ser crucial si s'intenta aconseguir la màxima eficiència amb els recursos que es tenen. Saber com millorar una part per aconseguir millors resultats o un nombre més elevat d'imatges per segon (a partir d'ara FPS) pot portar-nos a tenir un producte final molt més pulit de cara al públic.

Així mateix, sempre he pensat que una escena, per molt ben estructurada i dissenyada que estigui, si té una mala il·luminació no destacarà i el resultat serà més pobre. Tenir en compte el nivell d'il·luminació, ombres i colors pot canviar completament la percepció d'un nivell. Tal com es pot observar a la Figura 1.1a, l'escena sembla molt plana, mentre que a la Figura 1.1b s'observa una imatge més viva, més adient en temes d'il·luminació i color al context general de l'escena.

(a) Escena base

(b) Il·luminació i colors corregits

Figura 1.1: Diferència al usar il·luminació i colors adients

1.2 Propòsits

El propòsit principal d'aquest treball és el d'obtenir una millor visió sobre com funciona un motor de videojocs. En especial ens fixarem en l'apartat de rendering i il·luminació, ja que són els més importants per la feina que es vol dur a terme. Així doncs, es tracta d'un treball de recerca en el qual s'estudiarà la possibilitat d'implementar la tècnica d'il·luminació anomenada Lightcuts a l'estructura d'un motor comercial, en el nostre cas, l'Unreal Engine 4.

1.3 Objectius

L'objectiu principal del projecte és implementar, de manera funcional, la tècnica Lightctus, havent de modificar així el codi font del motor Unreal Engine 4. Això implica una primera tasca molt important de recerca que ocuparà gran part del treball i que es durà a terme durant quasi tot el projecte. Això és a causa de la falta de documentació més específica sobre l'estructura interna del motor. Finalment, caldrà estudiar la portabilitat i eficiència d'aquest algoritme en un entorn en temps real.

Així doncs, el desenvolupament principal del projecte es basa en els següents punts:

- Estudi de l'estructura interna del motor.
- Definició del funcionament de la tècnica.
- Disseny i implementació de l'algoritme Lightcuts.
- Definició d'una escena simple de prova per demostrar la correctesa de la tècnica.

- Verificació i proves de funcionament.
- Estudi de l'eficiència de la tècnica en un entorn en temps real.
- Acabament i estructuració de la documentació del projecte.

1.4 Estructura del document

La memòria està estructurada en els següents apartats:

- **Capítol 1. Introducció, motivacions, propòsit i objectius del projecte.** S'expliquen el perquè del desenvolupament d'aquest projecte, quins són els objectius proposats i com s'ha organitzat el desenvolupament.
- **Capítol 2. Estudi de viabilitat.** Es justifiquen els paràmetres que fan possible el desenvolupament del projecte en termes de recursos econòmics, humans i tecnològics.
- **Capítol 3. Metodologia.** S'explica la metodologia seguida per a desenvolupar el projecte justificant les raons de la decisió.
- **Capítol 4. Planificació.** Defineix l'estratègia seguida per arribar als objectius plantejats i la temporització del projecte que s'ha desenvolupat.
- **Capítol 5. Marc de treball i conceptes previs.** Descriu els diversos aspectes relacionats amb el desenvolupament general del projecte, les principals accions que s'han dut a terme als inicis del projecte i conceptes teòrics necessaris per a la millor comprensió de la memòria.
- **Capítol 6. Requisits del sistema.** Es descriuen els requisits funcionals i no funcionals del sistema necessaris per desenvolupar el projecte.
- **Capítol 7. Estudis i decisions.** S'expliquen les eines que han estat utilitzades, les seves característiques i el paper que han tingut dins el projecte.
- **Capítol 8. Anàlisis i disseny del sistema.** S'hi compren la investigació del problema a resoldre tals com les especificacions, esquemes d'implementació, classes i mètodes del projecte.
- **Capítol 9. Implementació i proves.** Es descriu el procés que s'ha dut a terme per implementar la tècnica, explicant els detalls més importants, i com s'han solucionat els problemes sorgits.
- **Capítol 10. Implantació i resultats.** Mostra les proves d'execució de la tècnica a través d'exemples i imatges del motor.
- **Capítol 11. Conclusions.** S'exposen les conclusions obtingudes un cop finalitzat el projecte, així com una crítica als resultats obtinguts.
- **Capítol 12. Treball futur.** Es descriuen possibles ampliacions o millores que es podrien realitzar en un futur a partir de l'estat actual del treball.
- **Capítol 13. Bibliografia.** Conté les referències utilitzades per a desenvolupar el projecte.
- **Capítol 14. Annexos.** Inclou explicacions i/o documents que s'han utilitzat durant el projecte que no són indispensables per a la comprensió global del projecte.

Capítol 2

Estudi de viabilitat

Dins d'aquest capítol es discutirà la viabilitat del projecte tenint en compte els recursos tècnics i/o humans, requisits tecnològics o el cost econòmic ja que tot influeix en la manera d'enfocar el treball.

2.1 Recursos tècnics

El treball sempre s'ha dut a terme sobre la mateixa màquina, la qual té com a sistema operatiu un Windows 8.1 de 64 bits. Tenint en compte la naturalesa del projecte i que el motor Unreal Engine 4 (a partir d'ara UE4) està pensat per a funcionar en tots els sistemes (Windows, Mac OS i Linux), hauria de funcionar perfectament. Tot i això no és recomanable a causa a que s'ha treballat amb el codi font original, i la forma d'instal·lar l'editor a cada sistema operatiu varia en gran mesura.

La màquina que s'ha utilitzat pel projecte té les següents característiques:

- **Sistema operatiu:** Windows 8.1 Pro 64 bits
- **Processador:** Intel i5-4670k 3.4GHz
- **Memòria RAM:** 2x8GB 1600MHz
- **Targeta gràfica:** NVIDIA GTX 780 3GB

2.2 Recursos humans

Qualsevol projecte que comença a ser gran en volum de feina necessita una bona organització humana de cares a fer un bon treball. Per aquest motiu, un cop analitzade les diferents tasques a fer, es va decidir escollir els següents perfils professionals:

- **Cap del projecte:** Qualsevol projecte de gran envergadura necessita un líder que faci un anàlisis dels avenços que es porten a terme així com assegurar-se que tot es faci seguint la metodologia escollida.
- **Analista:** És l'encarregat d'estudiar el projecte i definir com s'estructurarà l'aplicació. Això implica fer un estudi inicial dels objectius i dissenyar adequadament l'estructura interna tenint en compte les funcionalitats que es necessiten. Així mateix també haurà de proporcionar les eines necessàries per dur a terme aquestes tasques.

– **Programador:** Ha de crear l'aplicació a través de les instruccions que l'analista li hagi passat, seguint el disseny creat inicialment.

A causa de la naturalesa d'aquest projecte, les tasques d'analista i programador han recaigut en la mateixa persona. Finalment el rol de cap de projecte ha recaigut sobre el tutor, que ha mantingut un bon ritme de treball en tot moment.

2.3 Recursos tecnològics

Tenint en ment la màquina anteriorment comentada a l'apartat 2.1, s'han definit una sèrie d'eines que s'utilitzaran durant el transcurs d'aquest projecte. Per programar s'ha utilitzat el llenguatge C++ (Secció 7.3), ja que és el llenguatge en el qual està escrit el codi font de l'UE4 i per la qual cosa no es pot canviar. Per aquest motiu necessitarem un IDE per a programar i compilar. En aquest cas l'escollit ha estat el Visual Studio 2013 (Secció 7.2). Per a programar els shaders s'ha utilitzat el llenguatge HLSL (Secció 5.3) per la mateixa raó que el C++, és el que utilitza l'UE4.

Els recursos tecnològics mencionats fins ara són els més importants que s'han utilitzat en el projecte. Tot i això s'ha usat més programari el qual està descrit en més detall al capítol 7.

2.4 Cost econòmic

Cal tenir en compte el pressupost en un projecte ja que sempre és una part molt important. Per aquesta raó, en aquest treball s'han utilitzat eines gratuïtes o que ofereixen llicències d'aprenentatge o versions més simples del producte de pagament. El codi font de l'UE4 és totalment obert a tothom i es pot obtenir obrint un compte d'usuari a la pàgina web www.unrealengine.com. D'altra banda, Visual Studio ofereix llicències gratuïtes per la qual cosa no suposa cap cost extra dins el pressupost.

Primer de tot cal tenir en compte les hores de feina que el personal es passa treballant. En aquest cas, tot i fer la feina una sola persona, optem per calcular el que costaria si contractéssim un analista i un programador segons els seus sous:

– **Analista:** 20€/h.
– **Programador:** 10€/h.

Seguidament cal pensar en les tasques que s'han de desenvolupar i qui les desenvolupa. En el nostre cas són les següents:

– Estudi de l'estructura del motor
– Definició del funcionament de la tècnica
– Disseny de l'algoritme de Lightcuts
– Implementació
– Proves
– Documentació

Al usar un ordinador personal, caldrà comptar en el pressupost el valor de l'amortització de la màquina. Aquest valor ve determinat per la següent fòrmula:

$$\text{Amortització} = \frac{\text{Preu recurs} \cdot \text{Mesos de feina}}{36} = \frac{1306.01 \cdot 8}{36} = 290.23$$

Per tant el cost en funció del perfil de les tasques a realitzar és el següent:

Tasques	**Perfil**	**Hores**	**Euros/Hora**	**Cost total**
Estudi de l'estructura del motor	Analista	200	20	4000
Definició del funcionament de la tècnica	Analista	10	20	200
Disseny de l'algoritme de Lightcuts	Analista	100	20	2000
Implementació	Programador	250	10	2500
Proves	Programador	100	10	1000
Documentació	Analista	100	20	2000
Total Analista		410	20	8200
Total Programador		350	10	3500
Total Amortització maquinària				290.23
TOTAL				**11990.23**

2.5 Conclusions de l'estudi de viabilitat

Gràcies al fet d'utilitzar recursos tècnics a l'abast de tothom i de comptar amb una màquina personal comprada anteriorment, provoca que per l'àmbit tècnic el treball sigui totalment viable. D'altra banda, si posem l'atenció sobre les tasques a fer i les hores de cada tasca, veiem com el preu s'eleva ràpidament. Això passa a causa que inicialment hi ha hagut moltes hores d'investigació per entendre el funcionament del motor. Suposem que si es contractés a gent experta en el camp, els costos totals es reduirien molt respecte el cost actual.

Tot i així, tenint en compte que és un projecte dut a terme per un sol estudiant i que els recursos tècnics i tecnològics no intervenen en el cost final, podem concloure que aquest treball és viable.

Capítol 3

Metodologia

Avui dia trobem moltes metodologies de desenvolupament eficients i diferenciables, des de les més antigues com la metodologia Waterfall fins a les més modernes tècniques Agile. Tot i així, en aquest projecte s'ha decidit no seguir cap metodologia d'un tipus concret, ja sigui Spiral, Scrum o Iterative, així com les esmentades anteriorment. S'ha decidit definir un tipus de metodologia que funcionés bé amb les característiques del projecte, tal com es mostra en el diagrama de flux de la Figura 3.1.

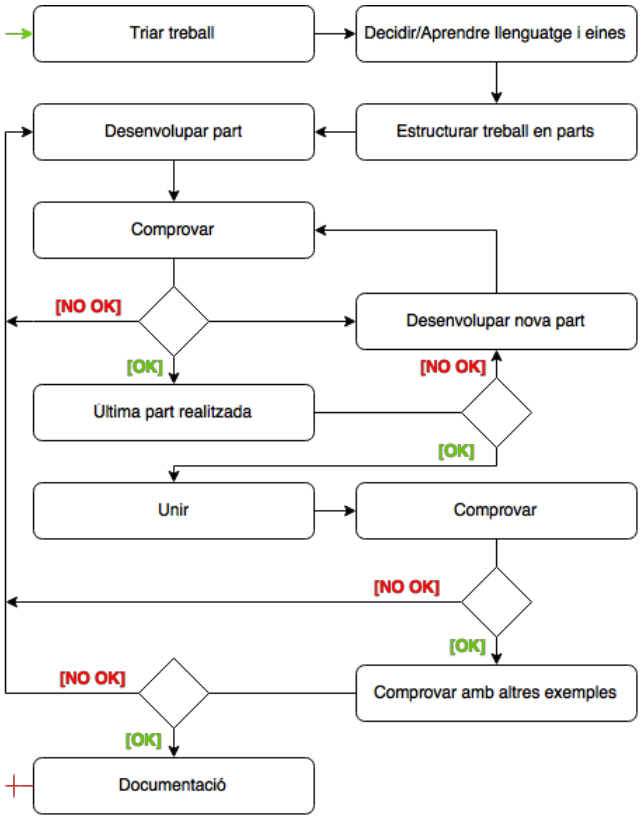


Figura 3.1: Diagrama de flux de la metodologia utilitzada

Els passos que trobem en aquesta metodologia són els següents:

1. Triar el treball a desenvolupar
2. Decidir el llenguatge de programació i les eines a utilitzar
3. Aprendre el llenguatge de programació i el funcionament de les eines escollides.
4. Estructurar el treball en parts segons les funcions que s'han de realitzar.

5. Desenvolupar la part corresponent seguint l'ordre de l'estructura del treball.
6. Fer comprovacions per tal de confirmar que el funcionament és correcte al finalitzar la part.
   6.1. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns a la última part desenvolupada o en les anteriors, si és convenient.
   6.2. Si al fer les comprovacions el resultat és l'esperat, es desenvolupa la part següent tornant al punt 5. Si s'han finalitzat les parts amb les seves respectives comprovacions s'inicia el punt 7.
7. Unir totes les parts desenvolupades i comprovar que el funcionament és correcte.
   7.1. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns a l'última part desenvolupada o en les anteriors, si és convenient.
   7.2. Si al fer les comprovacions el resultat és l'esperat s'inicia el punt 8.
8. Generar diferents models d'exemple per a comprovar que el funcionament és el correcte.
   8.1. Si al fer les comprovacions el resultat no és l'esperat, es torna al punt 5 per a realitzar els canvis oportuns a l'última part desenvolupada o en les anteriors, si és convenient.
   8.2. Si al fer les comprovacions el resultat és l'esperat s'inicia el punt 9.
9. Acabar la documentació.

Tal com es pot veure, consisteix en dividir el projecte en mòduls i organitzar en el temps el desenvolupament, segons el temps de verificació i de correcció. Tant durant el temps de desenvolupament com de verificació, es fa un seguiment mitjançant tutories setmanals o bisetmanals depenent de l'etapa, ja que en els inicis són més lents que no pas en les etapes finals, i no sempre és necessari fer tutories setmanalment.

Quan s'acaba el mòdul, sempre que es pugui, es finalitza totalment de manera que no es torna a tocar. D'aquesta manera es garanteix que els errors que puguin sorgir en la resta de mòduls són únicament d'aquests i no de cap dels anteriors. I si es dóna el cas que ho és, al menys s'han minimitzat al màxim els efectes que s'hagin pogut propagar.

Pel procés de disseny s'utilitzarà el llenguatge de modelat estàndard dins el camp de l'enginyeria de programari, l'UML (*Unified Modeling Language*). L'UML s'utilitza per definir un sistema, per detallar els seus elements, per documentar i construir. Per aconseguir això, l'UML disposa de nombrosos tipus de diagrames que mostren diversos aspectes dels elements representats.

Capítol 4

Planificació

4.1 Planificació inicial

Els inicis del projecte es troben a meitats de Desembre del 2014. Inicialment es va contactar amb el professor per tal de buscar un projecte el qual s'adeqüés a les motivacions personals. Així mateix, es buscava un treball en el qual es tractés el tema d'il·luminació. Finalment, es va decidir escollir el projecte d'implementar la tècnica Lighcuts, normalment utilitzada en rendering offline, dins l'arquitectura del motor de videojocs Unreal Engine 4.

Un cop escollit el tema del projecte es van definir una sèrie de tasques que, seguin la metodologia escollida anteriorment (Capítol 3), s'havien de seguir per tal de complir els objectius dins els terminis establerts i amb resultats satisfactoris. Les tasques que es va creure convenient crear inicialment són les següents:

1. Introducció inicial al motor UE4. Cal tenir els conceptes previs de com funciona el motor de cares a la utilització per part d'usuaris. No podem modificar el codi font sense saber el funcionament com a usuaris.

2. Estudi de l'estructura interna del motor. Començar a estudiar a través del codi font i la documentació com s'estructura el motor per tal de tenir una visió més complerta del funcionament.

3. Dissenyar l'algoritme de la tècnica Lightctus i decidir en quin punt del codi s'introduïrà.

4. Implementar la tècnica per tal que funcioni correctament.

5. Construir un petit nivell de prova per mostrar els resultats

6. Documentar tot el projecte durant el desenvolulpament.

Així doncs, a la Figura 4.1 podem veure les tasques repartides durant el període de temps que s'ha treballat.

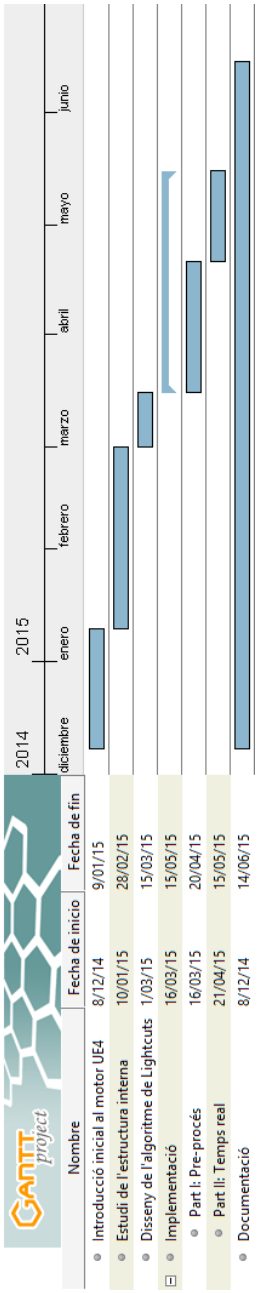


Figura 4.1: Diagrama de Gantt de la planificació prevista

4.2 Planificació final

Tal com es pot observar, la planificació prevista inicialment tenia com a termini d'acabament del projecte el Juny de 2015. Finalment no va ser possible tenir llest el projecte per aquesta data a causa de problemes originats amb el treball.

El principal problema amb el qual ens vam trobar va ser la falta de documentació del codi, ja que per la part d'usuari hom pot trobar molta informació, però en el moment en que es vol modificar el codi font, l'ajuda disminueix dràsticament. Aquesta falta d'informació va provocar que el procés d'estudi intern del motor s'allargués durant les tasques d'implementació, és a dir, es van anar fent les dues a la vegada. Per tant, cada avanç implicava haver de cercar les classes i punts exactes del codi a on insertar la nostra tècnica de forma correcta, per la qual cosa el desenvolupament va agafar una dinàmica més lenta.

Per aquest motiu, la planificació final que s'ha obtingut del desenvolupament del projecte és la que es pot observar a la Figura 4.2. Com pot observar-se, la distribució de tasques va agafar una estructura més caòtica a causa que es va haver d'anar fent investigació al mateix temps en que s'implementaven funcionalitats.

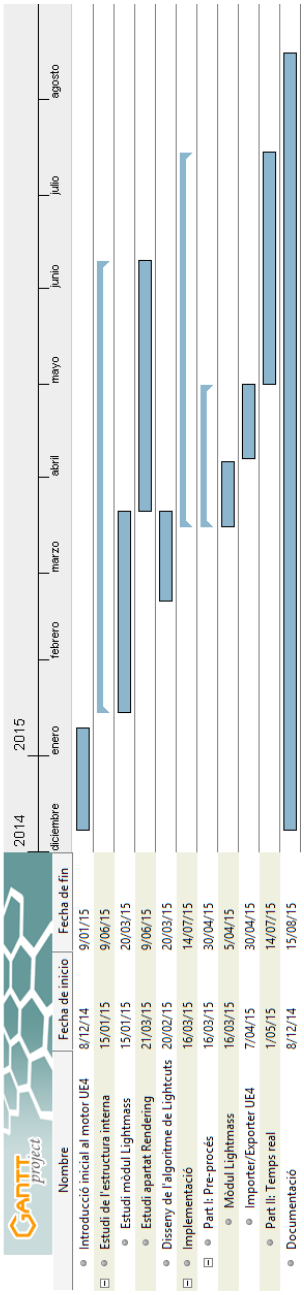


Figura 4.2: Diagrama de Gantt de la planificació final

Capítol 5

Marc de treball i conceptes previs

Abans de continuar amb l'explicació del treball és necessària una explicació sobre varis conceptes que anirem veient al llarg del document per tal de fer més simple i entenedora la comprensió.

5.1 Motors de videojocs

Per a desenvolupar un videojoc sempre s'opta per utilitzar un motor de videojoc (Game Engine), el qual s'adapti a les necessitats dels desenvolupadors. Això permet que no s'hagin d'enfrontar directament amb moltes llibreries de processament gràfic, sistemes de físiques, so, xarxa, etc. Gràcies als motors, el desenvolupament és molt més simple ja que permet que els desenvolupadors es centrin en crear el seu joc i no s'hagin de preocupar que passa en les capes més baixes del sistema.

Qualsevol game engine que vulgui ser utilitzat ha d'aconseguir oferir als usuaris una bona gestió dels nivells més baixos, tal com poden ser models o textures, i que al mateix temps ofereixi tot un conjunt d'eines que permetin programar a un nivell més alt per tal de facilitar la feina a l'usuari.

Escollir un bon motor a la hora de desenvolupar un joc és crucial, ja que hi ha molts motors diferents i, tot i que tots fan la mateixa feina, alguns ofereixen eines que els fan sobresortir. Alguns dels motius que porten a escollir un motor poden ser els següents:

- Facilitar i simplificar el desenvolupament de les aplicacions.
- Obtenir l'abstracció de la plataforma. Si un motor pot utilitzar-se en diverses plataformes, els jocs també ho podran fer.
- Grups de treball en paral·lel. Degut a la separació entre motor i continguts, es pot treballar en diversos grups de treball alhora.
- Beneficis a tercers. Donada la naturalesa del motor, tots aquells avenços que s'hi facin també seran beneficiosos per als usuaris tercers, els quals es beneficien en el desenvolupament d'aquestes millores.
- Elimina la necessitat de tenir grans coneixements de rendering a baix nivell. Gràcies a que la capa de rendering queda amagada pel game engine, el desenvolupador no s'ha de preocupar per això i pot ocupar el temps en altres parts del videojoc. Cal tenir en compte que molts motors permeten modificar forces paràmetres dels sistemes de rendering, la qual cosa beneficia a tothom.

Finalment, ja hem vist que el game engine permet amagar les capes de programació a baix nivell que hi ha darrera seu, per la qual cosa és molt més simple per l'usuari. Així doncs, l'estructura interna d'un motor de videojocs ve definida tal com es pot observar a la Figura

5.1. Per a un usuari, les capes més importants seran les dues primeres. La primera capa fa referència a tots aquells apartats específics del videojoc. La segona ja forma part del motor en qüestió i és la que varia entre els diferents motors (eines posades a disposició de l'usuari).

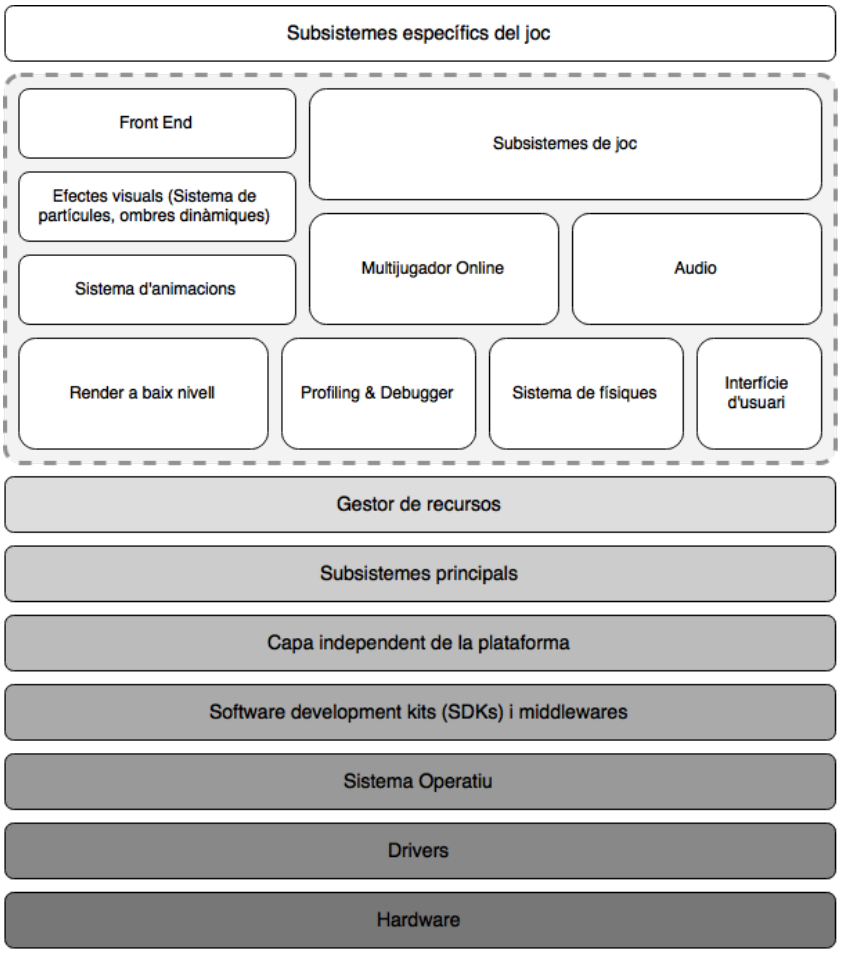


Figura 5.1: Esquema de l'estructuració d'un motor de videojocs

5.1.1 Unreal Engine

L'Unreal Engine és un motor desenvolupat per Epic Games, el qual va ser utilitzat per primera vegada en el joc Unreal l'any 1998. El motor s'ha convertit en un dels més importants dins d'aquest món degut a la gran quantitat d'eines disponibles que té, moltes de les quals fan el desenvolupament d'un videojoc molt més simple.

El motor va guanyar molta popularitat en la seva 3ª versió ja que va ser un dels més utilitzats per desenvolupar sobre PS3 i Xbox 360. L'última versió (UE4) ha portat moltes novetats, com el fet de poder crear tot un joc de principi a fi sense haver de programar ni una sola línia de codi, utilitzant els anomenats Blueprints. L'eina de Blueprints treballa sobre un entorn de programació gràfic, utilitzant diagrames d'estats i transicions. Això ha permés que molta gent sense gaire coneixement en programació pugui entrar més fàcilment a desenvolupar videojocs i donar visió a les seves idees més fàcilment.

Unreal Engine 4 es pot aconseguir des de principis de 2015 gratuïtament. La llicència permet llençar videojocs creats i monetitzar-los sense cap mena de cost adicional mentre no sobrepassin els $3000 per quatrimestre de benefici. Un cop es doni el cas, Epic rebrà un 5% dels beneficis

en conceptes de royalties. També es dona total accés al codi font del motor per a que tots els usuaris li puguin donar un cop d'ull, ja que això pot ajudar a entendre millor el motor o debuggar les interaccions entre el codi del motor i el dels jocs que es creen.

Dins dels videojocs més importants que s'han creat amb aquest motor podem trobar sagues tan característiques com Gears of War (Figura 5.2), Unreal Tournament i Bioshock, entre d'altres.

Figura 5.2: Gears Of War 3, Epic Games 2011

5.1.2 Unity

Unity és un motor multiplataforma desenvolupat per Unity Technologies. Tot i ser anunciat exclusivament per OS X el 2005, finalment s'ha extès a més de 15 plataformes, dins les quals trobem les més importants com PC, PS4 i Xbox One.

Unity s'ha definit com el gran motor per a companyies independents o nous desenvolupadors, gràcies a comptar amb una llicència gratuïta, que ofereix moltes de les prestacions del motor, i una llicència de pagament, però a un preu assequible, que ofereix moltes més característiques que poden ajudar a fer més senzill el desenvolupament alhora que s'aconsegueixen resultats més complexos. Un dels aspectes més importants i característics d'Unity és la gran facilitat per canviar la plataforma objectiu, arribant a tal punt que en alguns casos de jocs més senzills, modificant un paràmetre del joc es pot passar d'Android a iOS molt fàcilment.

Dins la gran quantitat de jocs que s'han desenvolupat per Unity, els més importants o coneguts són jocs com Kerbal Space Program, Cities Skyline (Figura 5.3) i The escapists.

5.1.3 CryEngine

Motor de videojocs desenvolupat per l'empresa alemana Crytek. Inicialment es va construir com un motor de demostració per Nvidia. Vist el potencial que tenia el motor, es va decidir implementar-se per primera vegada al videojoc Far Cry, desenvolupat per la mateixa Crytek.

El motor ofereix unes característiques gràfiques molt elevades, ja que és el millor alicient i qualitat a la hora de convèncer els desenvolupadors. Això queda demostrat en la gran qualitat gràfica que s'ha aconseguit així com el sistema de física. Tot i sobresortir en alguns aspectes, falla en altres com pot ser la interfície d'usuari i la facilitat d'utilització amb l'usuari, per la

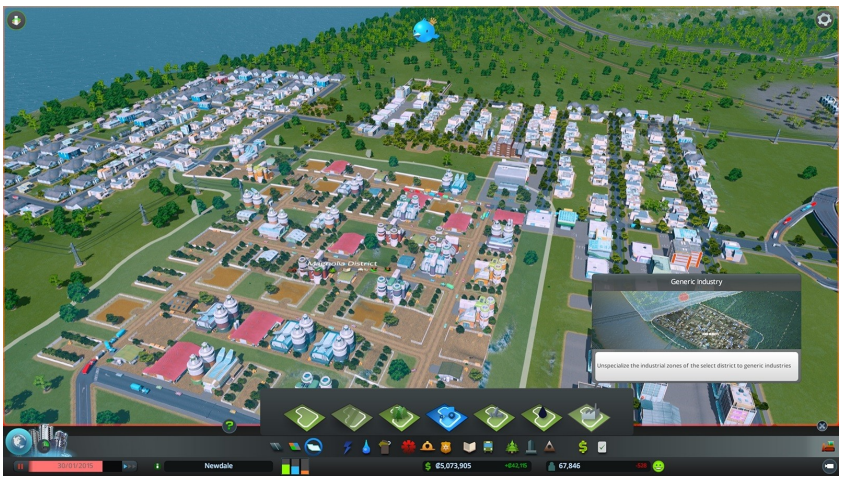


Figura 5.3: Cities Skyline, Colossal Order 2015

qual cosa, altres motors tenen preferència per ser escollits.

El joc més famós i que demostra més clarament les qualitats del motor és el Crysis (Figura 5.4), que tot i ser un joc, a vegades és més vist com una demostració tècnica del potencial del CryEngine. Els primers jocs de la saga Far Cry també estan inclosos dins la família CryEngine, així com Ryse: Son of Rome i Star Citizen, un dels màxims exponents gràfics que s'estan creant avui dia.

Figura 5.4: Crysis 3, Crytek 2013

5.1.4 Elecció de l'Unreal Engine 4

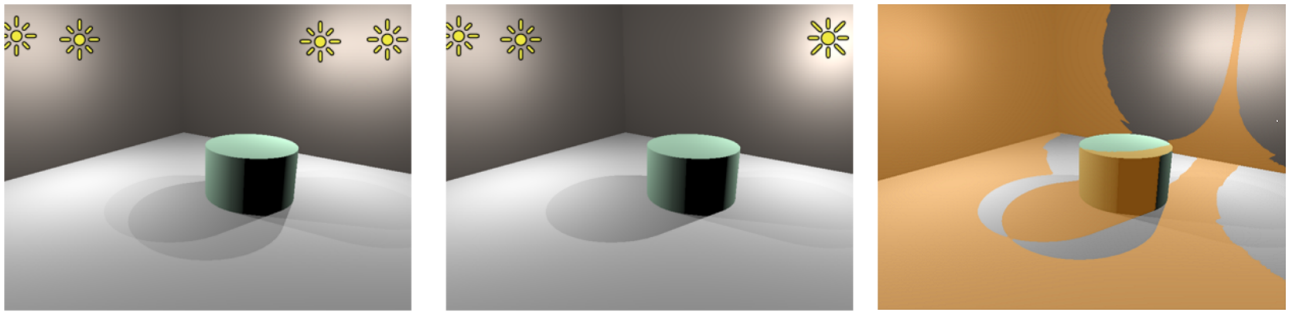
El motor escollit al final va ser l'Unreal Engine 4. La decisió va venir donada per les necessitats d'aquest projecte. El fet de poder accedir al codi font del motor, d'una forma molt simple i amb indicacions clares de com compilar i provar l'entorn van decantar la balança cap aquest motor. A més, la documentació existent sobre el codi, tot i ser insuficient en alguns apartats, especialment pel tema d'il·luminació, indica algunes classes del codi que poden ser una bona introducció a la implementació. Cal remarcar també el fòrum de dubtes que Epic posa a disposició dels desenvolupadors, els quals seran ajudats per la resta de la comunitat o per empleats d'Epic especialitzats en diferents parts del motor.

5.2 Lightcuts

5.2.1 Introducció a la tècnica

Lightcuts és una tècnica que permet aproximar la il·luminació total d'un punt de l'escena sense utilitzar totes les llums que interactuen amb aquell punt. Això provoca que els càlculs d'il·luminació siguin més ràpids però es continuï obtenint una bona il·luminació de l'escena. Per aconseguir aquesta aproximació, utilitza un arbre binari de llums que servirà per definir els talls de llums que indiquen quines llums són relevants per a la il·luminació d'un punt en concret. La tècnica va ser desenvolupada pel Departament de Ciències Computacionals de la Universitat de Cornell i va ser presentat per primera vegada al Siggraph 2005 [1] (Annex 14.1).

A la Figura 10.2 podem apreciar més clarament com funciona la tècnica amb un exemple senzill, mostrant les zones aproximades i aquelles zones que no han variat.



(a) (b) (c)

Figura 5.5: Escena simple amb 4 llums. (a) Solució exacte. (b) Solució aproximada unint les dues llums de la dreta. (c) La regió taronja representa parts a on la solució exacta i l'aproximada no varien. Els errors són més grans prop de les llums i a on la visibilitat varia.

5.2.2 Construcció de l'arbre de llums

El treball es troba dividit en dues parts, la primera de les quals té un paper molt important en la segona. Aquesta primera part és tot el pre-procés que s'ha de fer per tal d'obtenir l'arbre de llums que servirà per aproximar el càlcul d'il·luminació, en la segona etapa del procés.

La clau d'aquesta part de l'algoritme és intentar trobar aquell conjunt de parelles que facin la distància entre totes mínima. És a dir, aconseguir ajuntar les llums de tal manera que l'error per separació que es pot observar a la Figura 5.5c sigui el més petit possible. Hi ha moltes maneres d'aconseguir aquest objectiu. A l'article original del lightcuts [1](Annex 14.1) s'explica que s'utilitza una tècnica basada en proximitat i distàncies de les llums. D'aquesta forma, es creen una sèrie de clusters que aproximen les parelles anteriorment comentades. Això pot portar a aconseguir algunes parelles no del tot òptimes, però jugant una mica amb els paràmetres es poden aconseguir resultats realment bons. Tot i això, en aquest projecte s'ha decidit seguir l'estratègia de trobar les parelles més òptimes, per la qual cosa l'error serà el mínim possible. S'ha seguit aquest camí a causa que aquest pre-procés pot ser molt lent, ja que el càlcul de trobar les parelles pot ser molt costós pel seu cost exponencial, però també ho és seguint la tècnica dels clusters. Per tant, si el càlcul havia de ser igual de lent, s'ha decidit optimitzar les millors parelles.

5.2.3 Càlcul dels cuts i rendering

Un cop tenim l'arbre de llums creat, la part més dura del càlcul ja està feta. Quan toca fer el render en temps real, només cal analitzar l'arbre començant des de l'arrel i llavors anar baixant tenint en compte els criteris que explicarem a continuació.

La selecció de quines llums utilitzar (cut) funciona de la següent manera:

1. Iniciar aproximar la llum utilitzant l'arrel.
2. Si no passa el criteri d'aproximació, seleccionar el fill esquerre i el fill dret, i aproximar de nou amb les llums seleccionades (Repetir aquest pas amb cada fill fins a superar el criteri d'aproximació).

A la Figura 5.6 podem veure més clarament com funciona l'algoritme i com es decideix el cut final que indica les llums que s'utilitzaran per fer el càlcul final.

En el nostre cas, el criteri d'aproximació pel qual escollir una llum, o els fills d'aquesta que trobem a l'arbre, és simplement si el punt que volem pintar de l'escena es troba dins el radi d'acció de la llum. Començarem sempre comprovant si l'arrel de l'arbre aproxima correctament la il·luminació. Si no fos així, baixarem un nivell i comprovarem els dos fills. Si un d'aquests fills no aproxima bé, baixarem un altre nivell i comprovarem els seus fills. De forma contraria, si un node ja aproxima bé, no baixarem cap nivell més. Aquestes operacions s'aniran repetint fins aconseguir un cut que ens aproximi de la millor forma la il·luminació. S'utilitzarà un criteri de distància al punt d'avaluació per a definir si s'utilitza un node o, al contrari, es fa la comprovació amb els seus fills. Aquest mètode ens permet seleccionar aquelles llums que millor aproximen l'escena, per la qual cosa es pot aconseguir un resultat similar al render original sense haver d'utilitzar tantes llums.

5.3 HLSL

HLSL és l'acrònim de High-Level Shading Language (Llenguatge de shaders d'alt nivell) i és el llenguatge per shaders desenvolupat per Microsoft per la pipeline gràfica de Direct3D. Es pot comparar en gran mesura amb llenguatge GLSL (OpenGL Shading Language) i permet crear shaders que tenen una estructura semblant als programes fets amb C.

Els programes que podem crear amb HLSL poden tenir quatre formes diferents:

- **Pixel shader:** També conegut com a Fragment shader. Té la funció de determinar el color, entre altres elements, de cada pixel. S'hi calcula el color però també s'hi poden fer operacions d'il·luminació i ombrejat, entre altres. Aquest shader opera sobre un sol pixel, per la qual cosa no té coneixement de la geometria de l'escena i no pot dur a terme operacions més complexes.

- **Vertex shader:** S'executa per cada vèrtex entrat per l'aplicació. És responsable de transformar el vèrtex de l'espai d'objecte a l'espai de càmera, genera les coordenades de textura i calcula coeficients d'il·luminació tals com la tangent del vèrtex i vectors normals.

- **Geometry shader:** Concepte de shader força nou, el qual pot produïr noves primitives com punt, liínies i triangles a partir de les primitives passades al principi de la pipeline gràfica.

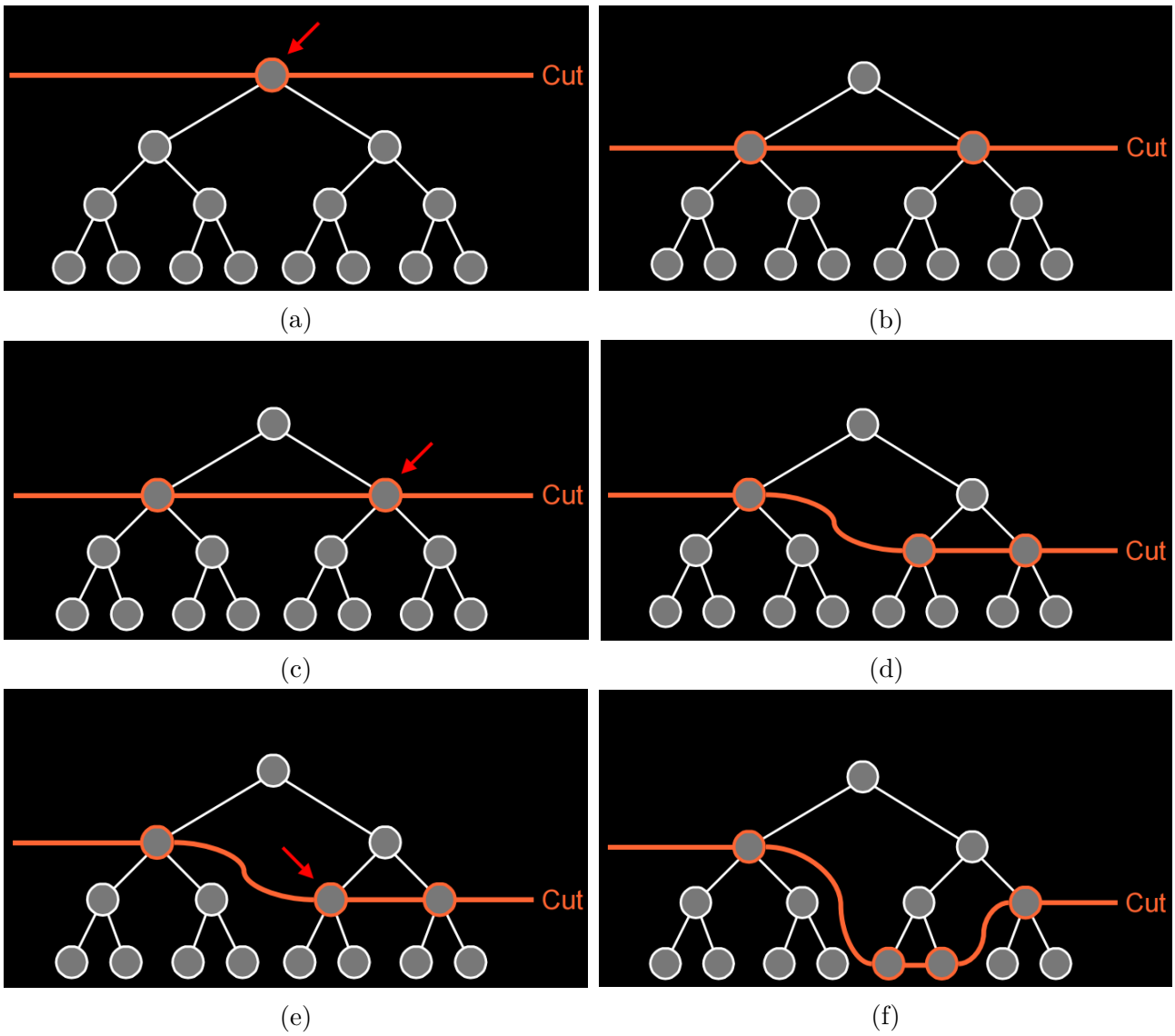


Figura 5.6: Esquema de funcionament de Lightcuts explicat pas a pas. (a) Seleccionem l'arrel i comprovem si aproxima. (b) L'arrel no aproxima bé per tant passem als fills. (c) El fill esquerre aproxima bé i comprovem el dret. (d) No aproxima bé per la qual cosa passem als seus fills. (e) Comprovem el fil esquerra. (f) Repetir les operacions anteriors fins aconseguir una bona aproximació, segons el nostre criteri d'avaluació.

– **Tesselation shader:** Shader molt nou el qual permet subdividir objectes 3D simples en objectes més petits en temps d'execució utilitzant una funció matemàtica. Això permet que els objectes més propera a la càmera tinguin més detall mentre que els més llunyans no tinguin tanta geometria però provocar que la qualitat sigui semblant.

En el nostre cas, el shader més important que utilitzarem serà un **Pixel Shader** ja que són aquells en els quals calculem la il·luminació per pixel. Això vol dir que el procés de càlcul de quines llums utilitzar de l'arbre de Lightcuts es farà dins aquest tipus de shader, ja que ens permetrà obtenir la millor il·luminació possible al fer-ho sobre cada pixel i no sobre cada vèrtex.

Capítol 6

Requisits del sistema

En aquest capítol s'explicarà quin és el problema que s'ha estudiat en aquest projecte, així com la solució que s'ha presentat per tal de solucionar-lo. A més, es donarà informació sobre els requisits funcionals i no funcionals que ha de complir el sistema.

6.1 Plantejament de la problemàtica

El problema a solucionar en aquest projecte, tal com s'ha vist anteriorment, és implementar i estudiar la tècnica Lightcuts en un entorn en temps real, en aquest cas s'ha escollit el motor UE4. L'objectiu que es busca aconseguir és que, un cop implementada la tècnica, aquesta es pugui fer servir de forma eficient i que produeixi els resultats esperats.

Així doncs, caldrà implementar la tècnica de tal manera que l'usuari del motor pugui fer el càlcul de l'arbre i, un cop acabat aquest, veure els resultats en temps real dins l'editor mateix del motor.

Per tal d'aconseguir aquests objectius, sorgeixen un seguit de problemes els quals es poden resoldre en els següents punts:

- Modificar el codi font sense perdre funcionalitats.
- Escollir el punt de modificació del codi per tal d'afegir funcionalitats sense que aquestes estiguin fora de lloc.
- Mostrar els resultats del càlcul en temps real, tant en l'editor com en el visualitzador del resultat final, així com les simulacions.

6.2 Plantejament de la solució

La solució presentada en aquest projecte es troba dividida en dues parts fonamentals, que segueixen l'estructura de la tècnica Lightcuts anteriorment comentada a l'apartat 5.2. Així doncs, tindrem la primera part la qual s'ocuparà de fer el càlcul de l'arbre òptim de llums; i la segona part que s'encarregarà de mostrar els resultats en temps real.

6.2.1 Part I: Càlcul de l'arbre

S'ha optat per incloure el càlcul de l'arbre dins el mòdul de computació anomenat **Lightmass** que fa servir l'UE4 per a fer tots els càlculs d'il·luminació indirecta així com els LightMaps de

totes les llums estàtiques que es troben en un nivell del joc. Hem decidit incloure el càlcul dins d'aquest mòdul tenint en compte el fet que tots els càlculs que fa estan relacionats amb la il·luminació i són tots pre-processos, per la qual cosa la inclusió d'aquesta nova part encaixa naturalment al codi i ens permet tenir una implementació més neta i organitzada.

6.2.2 Part II: Render en temps real

Per a la segona part, s'ha decidit implementar un nou shader el qual tracta només les llums que a nosaltres ens interessa. En aquest cas, les llums objectius són les de tipus Point Light. Aquest tipus de llums funcionen com una bombeta, emeten llum en totes direccions (Figura 6.1). Per a simplificar, les llums emeten la mateixa quantitat de llum en totes direccions de forma equitativa.

Per tant, en temps real s'agafarà l'arbre de llums calculat anteriorment i es farà el càlcul necessari dins el shader per saber quines llums s'han de fer servir per aproximar la il·luminació d'un punt de l'escena.

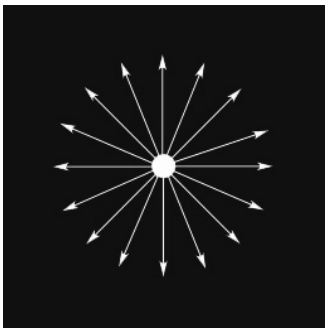


Figura 6.1: Funcionament d'una llum puntual en un espai 2D

6.3 Requisits funcionals

Els requeriments funcionals expliquen què ha de fer l'aplicació, és a dir, totes les funcionalitats que tindrà sense explicar com es farà. Aquestes són les funcionalitats d'aquest projecte:

- Afegir i eliminar llums de l'escena indicant indicant si es tindran en compte dins l'algoritme de Lightcuts.
- Mantenir una interfície gràfica simple i còmode cap a l'usuari.
- Permetre utilitzar llums de Lightcuts alhora que s'utilitzen les llums pròpies del motor.
- Poder iniciar el pre-procés de l'arbre de llums de forma simple i intuïtiva per a l'usuari i sempre que aquest vulgui.
- Mostrar els resultats del pre-procés en temps real, ja sigui des de l'editor o una simulació del joc.

6.4 Requisits no funcionals

En aquest apartat es defineix com ha de ser l'aplicació, i no què és el que ha de fer. Aquests requisits fan referència al que s'ha de tenir en compte a l'hora de dissenyar el sistema, com per

exemple els recursos necessaris.

Com que el programa només pot ser usat d'una manera (mode usuari), no es necessiten mesures de seguretat adicionals per controlar l'accés al programa. Així mateix tampoc es guardaran dades confidencials o d'alt risc, per tant no comptarem amb un sistema de protecció de dades.

Els requisits no funcionals són els següents:

- Es recomana treballar sobre un sistema operatiu Windows a causa que no s'ha provat en altres sistemes.
- Els requisits de hardware mínim seran els que demana l'UE4 degut a que es treballarà sobre una versió modificada d'aquest. En aquest cas són: Sistema operatiu Windows 7 de 64 bits, 8 GB de memòria RAM, tarjeta gràfica compatible amb DirectX 11 i un processador de quatre nuclis.

Capítol 7

Estudis i decisions

En aquest capítol es donarà una descripció de totes aquelles eines que s'han utilitzat durant el desenvolupament del projecte, tot justificant la seva elecció.

7.1 Unreal Engine 4

Ja hem explicat anteriorment que és l'UE4 a la secció 5.1.1, per tant en aquest apartat explicarem el funcionament bàsic i les funcionalitats que més es faran servir durant el projecte. UE4 disposa d'una interfície d'usuari simple i atractiva, creada de tal manera que la feina sigui simple i les eines sempre estiguin disponibles per a l'usuari de la forma més ràpida i eficient possible. Explicarem en detall l'editor de nivells, ja que ha estat el més utilitzat durant el desenvolupament d'aquest projecte, però també es donarà una petita descripció sobre l'editor de materials i l'editor de blueprints. Això és a causa que, durant el desenvolupament d'un videojoc, aquestes dues eines seran, juntament amb l'editor de nivells, les eines més utilitzades sense cap mena de dubtes.

7.1.1 Editor de nivells

Primer de tot ens trobem la pantalla inicial al obrir qualsevol nivell (sempre i quan no s'hagi canviat la interfície a les opcions) que dòna accés a totes les eines principals que utilitza qualsevol usuari. L'editor de nivells serà el més utilitzat al llarg del projecte, ja que ens permet configurar l'escena, afegir i eliminar objectes, etc. A la Figura 7.1 podem apreciar com està configurada la interfície del motor.

Es poden apreciar clarament les 7 seccions en que es troba dividida la interfície, les quals duen a terme les següents funcions:

1. **Mode:** Permet escollir el mode de treball. Entre els modes de treball disponibles trobem Col·locar (Permet posar objectes bàsics com llums, càmeres, disparadors d'objectius així

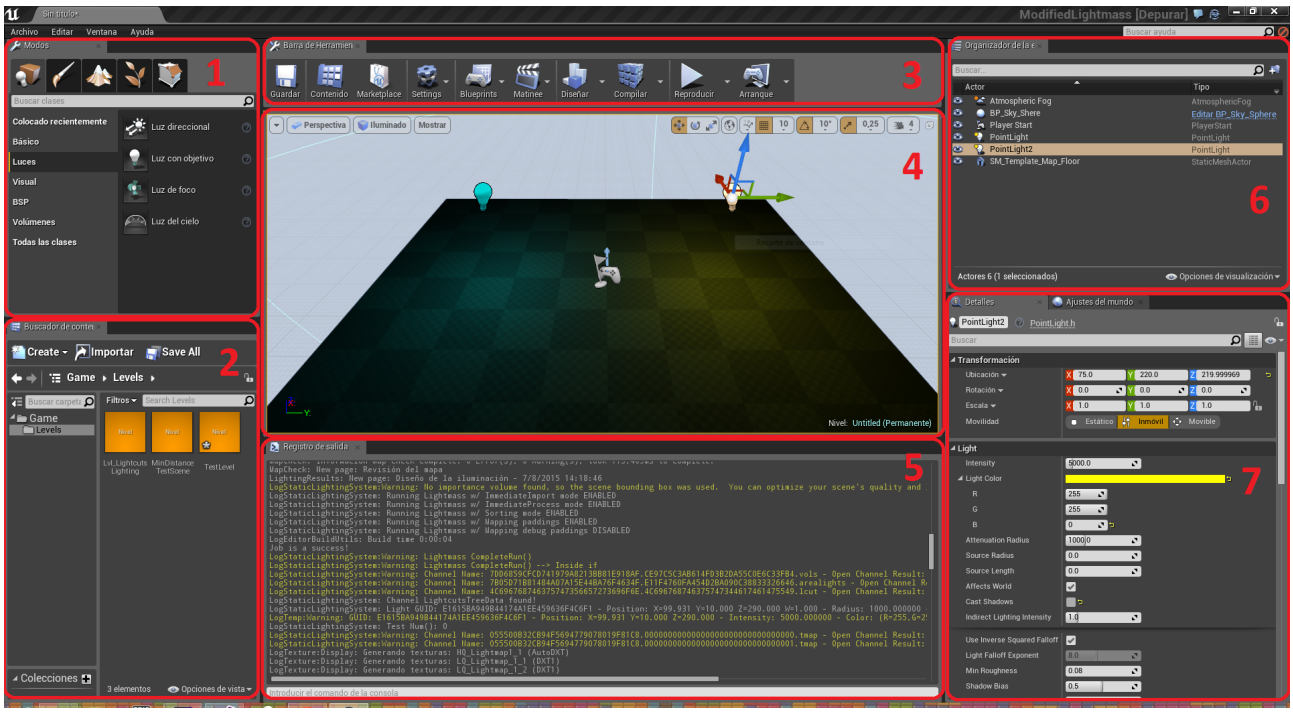


Figura 7.1: Editor de nivells

com geometria bàsica), Pintura (Pintar vèrtexs), Terreny (Editar el terreny del nivell), Fullatge (Pintar fulles d'arbres instanciades) i Editor de geometria (Modificar de forma bàsica alguns elements).

2. **Cercador de contingut:** Dòna accès a tota l'estructura del joc, a través dels directoris de nivells, materials i personatges, entre d'altres.
3. **Barra d'eines:** La barra d'eines, igual que la majoria d'aplicacions, és un grup de comandes que dònen un accés ràpid a eines i operacions usades normalment. En aquest projecte s'ha fet servir constantment l'eina Dissenyar, i en especial per dissenyar la il·luminació, ja que és el punt d'inici pel qual es crida l'algoritme de Lightcuts, iniciant-se el mòdul Lightmass.
4. **Finestra gràfica:** Punt d'entrada als móns que creem a l'UnrealEd. Aquesta finestra pot contenir varies finestres més petites que ens poden mostrar el nivell en una gran varietat de perspectives, donant control a l'usuari sobre què vol veure i com ho vol veure.
5. **Registre de sortida / Consola d'instruccions:** Mostra tot un seguit d'informació que pot ser rellevant a l'usuari. També podem introduïr algunes ordres que poden donar informació extra o permeten activar certs processos a través de la consola.
6. **Organitzador de l'escena:** Mostra tots els objectes que es troben a l'escena utilitzant una vista jeràrquica. Es poden modificar els objectes a través del mateix organitzador i podem mostrar informació extra com el nivell al qual corresponen, l'identificador o el tipus d'objecte.
7. **Detalls:** Conté informació, utilitats i funcions específiques de la selecció actual de la finestra gràfica. Hi trobem funcions d'edició per moure, rotar o escalar objectes, mostra totes les propietats editables i dòna un accés ràpid a propietats que són característiques del tipus d'objecte seleccionat.

Tal com s'ha comentat anteriorment, la part més usada de l'editor ha estat l'opció Dissenyar, més específicament Dissenyar > "Dissenyar només la il·luminació", ja que és el punt en el qual s'activa el mòdul Lightmass i, per tant, comença a funcionar l'algoritme de Lightcuts que s'ha implementat. A la Figura 7.2 podem observar les opcions que dòna el panell Dissenyar i la que nosaltres hem fet servir de punt d'entrada.

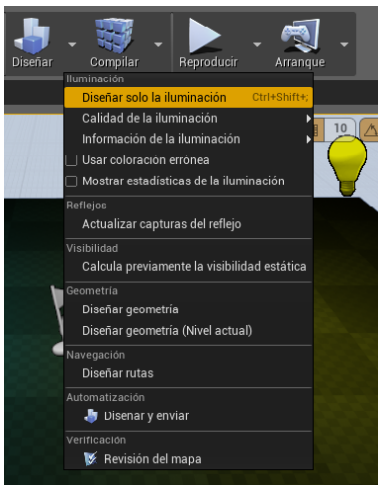


Figura 7.2: Opcions del panell Dissenyar

7.1.2 Editor de materials

L'editor de materials és una de les eines més potents, juntament amb els blueprints, que l'UE4 posa a disposició dels usuaris. Això és a causa de la senzilla i clara interfície gràfica i a la gran quantitat d'opcions i eines que es posen a disposició dels usuaris perquè aquests puguin modificar de la forma que vulguin els materials. A la Figura 7.3 podem observar com està estructurada la interfície.

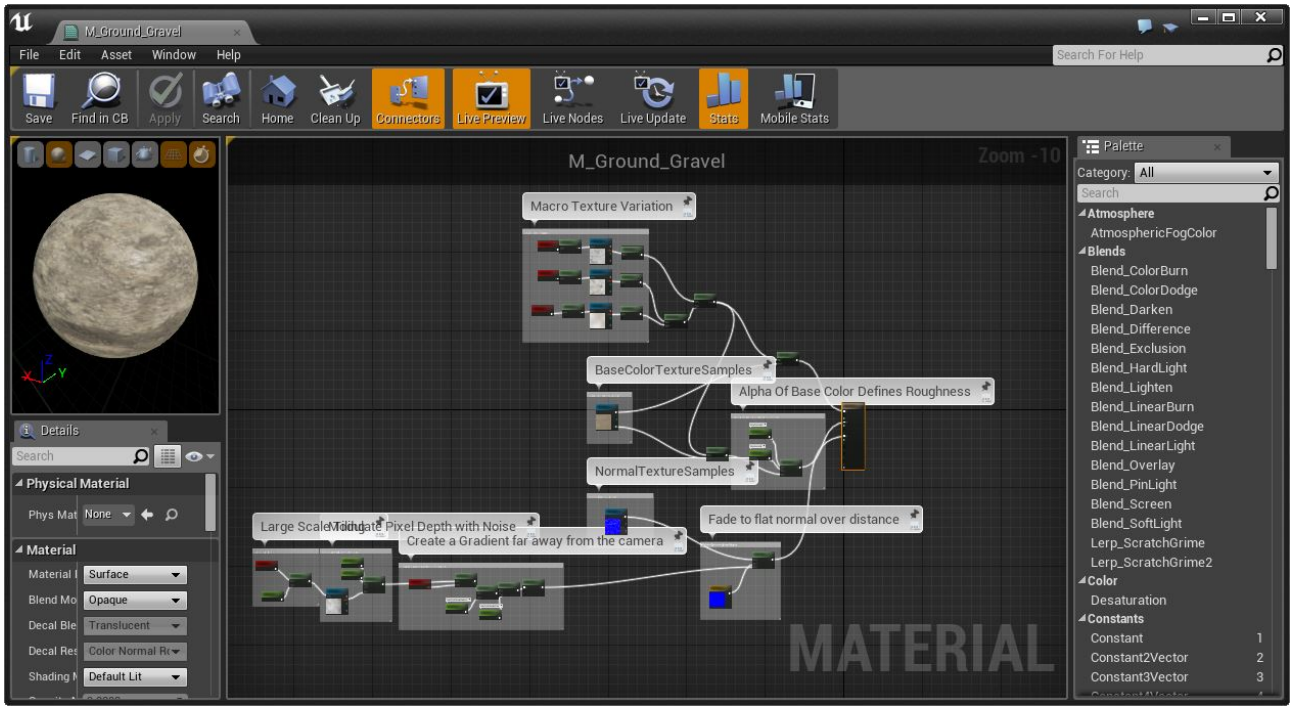


Figura 7.3: Editor de materials

El fet de treballar amb una interfície gràfica que utilitza un diagrama permet al dissenyador un gran control sobre les característiques que es volen pel material final. A més, la pre-visualitzacio ajuda molt a l'hora de veure quins efectes tenen els canvis que es van ocasionant.

Gràcies a aquesta facilitat de creació i modificació, és molt simple crear nous materials o experimentar amb els ja creats per tal d'obtenir els resultats que els usuaris busquen. També es proporciona la opció de mostrar la previsualització en diversos objectes 3D com poden ser

cubs, cons o plans per tal de veure més clarament com quedarà el material sobre la superfície que interessi al dissenyador. No és el mateix crear un material per un terreny i pre-visualitzar-lo sobre una esfera, ja que no quedarà clar quin serà el resultat final.

7.1.3 Editor de Blueprints

Els Blueprints són la principal novetat de l'Unreal Engine 4 i introdueixen tot un nou concepte de programació gràfica dins el motor mai vist anteriorment. Es tracta d'un entorn de programació que utilitza bàsicament grafs de nodes. Cada node té una funció diferent i es poden arribar a programar operacions molt complexes, així com iniciar events quan passi un fet en especial o, fins i tot, programar la intel·ligència artificial dels enemics.

A la Figura 7.4 podem apreciar com està estructurat un blueprint que controla la llum del sostre de l'escena actual. Veiem com es poden afegir objectes (que existeixen a la nostra escena) i variables dins els Blueprints. Això proporciona una forma de poder comprovar si els valors són correctes o fer proves més ràpidament un cop s'estigui fora de l'editor de Blueprints. Les opcions que es donen a l'usuari per a poder crear Blueprints són molt grans, ja que es podria arribar a crear un videojoc sense haver d'escriure ni una línia en C++. Simplement utilitzant els Blueprints en faríem prou.

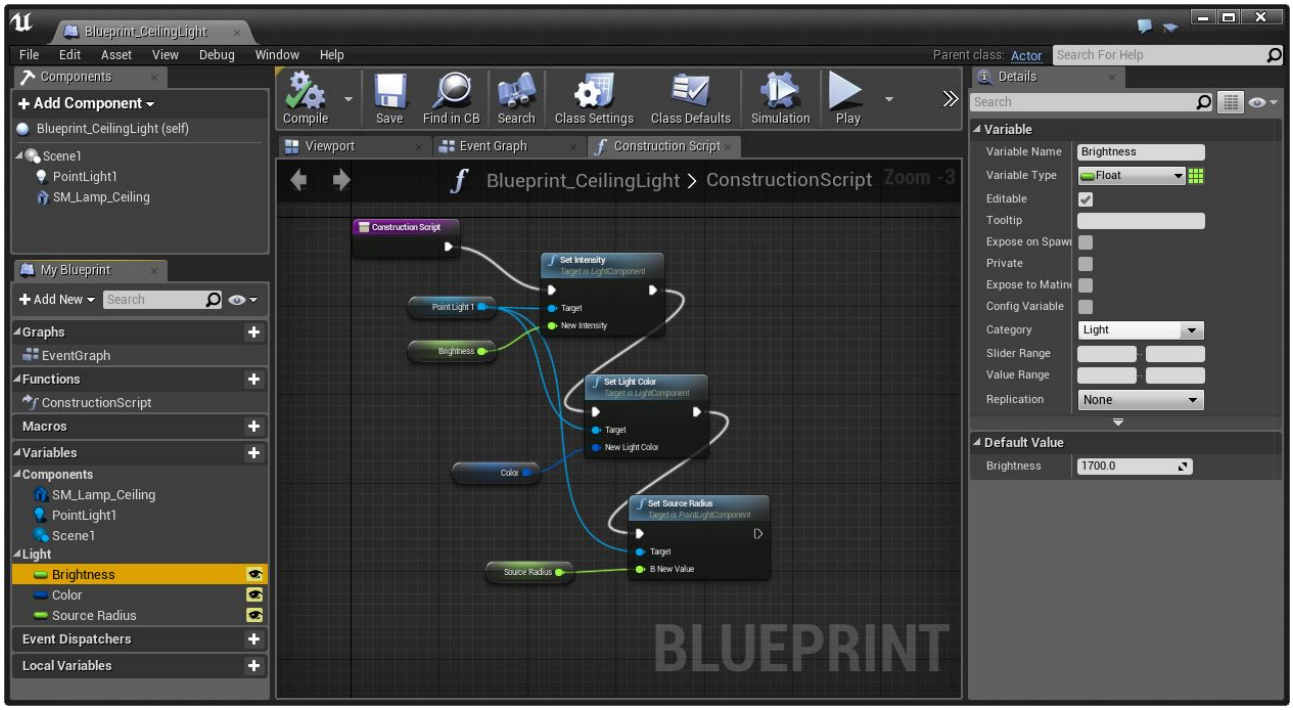


Figura 7.4: Editor de Blueprints

7.1.4 Motius de l'elecció

A la secció 5.1.4 hi trobem una explicació sobre les condicions que s'han donat per elegir aquest motor.

7.2 Visual Studio 2013

Microsoft Visual Studio (VS) és un entorn de desenvolupament integrat (IDE) de Microsoft. Aquest s'utilitza per desenvolupar programes d'ordinador per a la família de sistemes operatius Microsoft Windows. També permet desenvolupar pàgines web, aplicacions web i serveis web. Visual Studio utilitza diverses plataformes de desenvolupament de software com Windows API, Windows Forms, Windows Presentation Foundation, Windows Store i Microsoft Silverlight. Aquest permet produir tant codi natiu com codi interpretat.

Visual Studio inclou un editor de codi amb suport de IntelliSense i code refactoring. El debugger que porta incorporat permet treballar tant a alt nivell com a nivell de màquina. Podem trobar altres eines integrades com formularis de disseny per aplicacions amb interfície gràfica, dissenyador web, dissenyador de classes i un dissenyador d'esquema de bases de dades. VS accepta plug-ins per incorporar funcionalitats a quasi tots els nivells, com per exemple incorporar eines per control de versions, per treballar amb nous llenguatges, per gestionar el cicle de desenvolupament del software entre d'altres.

VS suporta diferents llenguatges de programació, per tant dòna suport en forma d'editors de codi i debuggers per gran part d'aquests llenguatges. Els llenguatges que tenen total suport per defecte són C, C++, VB.NET, C# i F#. Té suport també per llenguatges com SML/XSLT, HTML/XHTML, JavaScript i CSS. Es pot afegir suport per altres llenguatges, com ja s'ha comentat anteriorment, a través d'instal·lacions externes en forma de plug-in.

Hi ha disponible una versió "Express" de VS sense cap cost, així com també una sèrie de llicències comercials gratuïtes per a estudiants mitjançant el programa DreamSpark de Microsoft per a universitats.

7.2.1 Editor de codi

Com qualsevol altre IDE, VS incorpora un editor de codi (Figura 7.5) amb suport per ressaltat de sintaxi i auto completat de codi mitjançant IntelliSense tant per a variables, funcions, mètodes, bucles i queries. Els suggeriments d'auto-completat es mostren en un llistat desplegable que comença a la posició actual del cursor.

L'editor de codi també permet afegir marcadors dins el codi per a una navegació més ràpida i fluïda. Altres ajudes a la navegació que trobem són la capacitat de poder col·lapsar blocs de codi i una cerca incremental, a part de la cerca normal de text i la cerca segons expressions regulars. L'editor també és compatible amb la refactorització de codi, incloent paràmetres de reordenament, renombrament de variables i mètodes, extracció d'interfícies i encapsulació de classes dins de propietats, entre d'altres. Com cada vegada més IDEs fan, VS també incorpora la compilació en background. Mentre anem escrivint el codi, VS compila el codi a la mateixa vegada amb la finalitat de trobar errors sintàctics i errors de compilació. Tot i això, aquest mètode de compilació no genera cap codi executable, per la qual cosa s'haurà de fer servir un altre compilador per obtenir l'executable final.

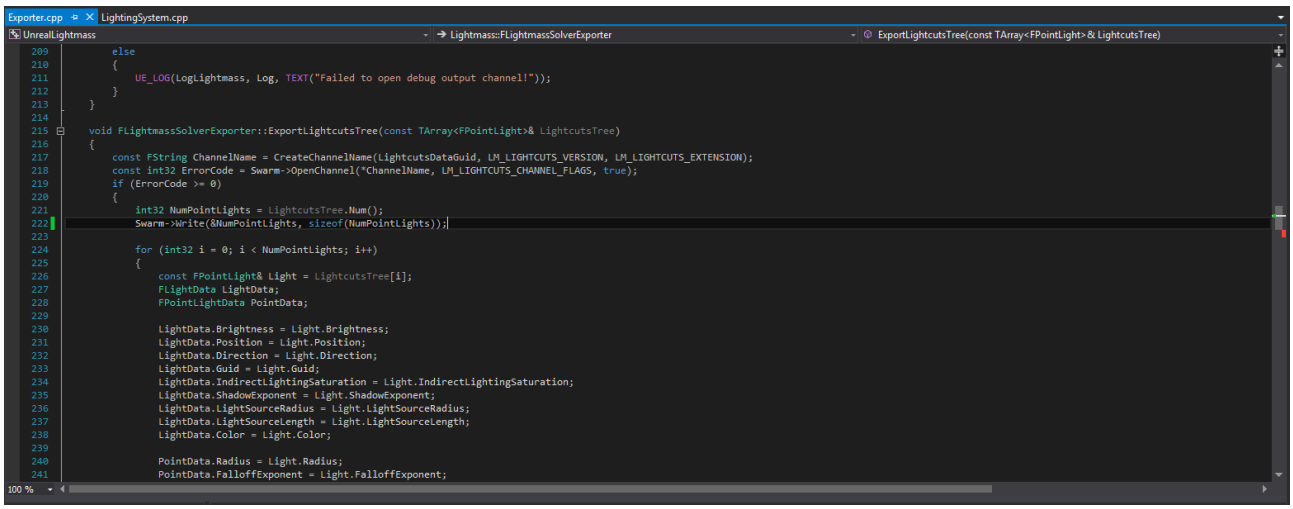


Figura 7.5: Visual Studio 2013. Editor de codi

7.2.2 Debugger

El debugger (Figura 7.6) que trobem inclòs dins el VS pot treballar a alt nivell així com a nivell màquina. Com ja hem comentat funciona tant amb codi interpretat com amb codi natiu i per tant pot ser utilitzat per debuggar qualsevol aplicació que estigui escrita sota un llenguatge suportat per VS. Si es té accès al codi del procés podem mostrar quina línia de codi s'està executant en cada moment i anar seguint la traça. Una funcionalitat afegida és que podem configurar el debugger perque s'activi quan una aplicació fora de l'entorn de VS falli.

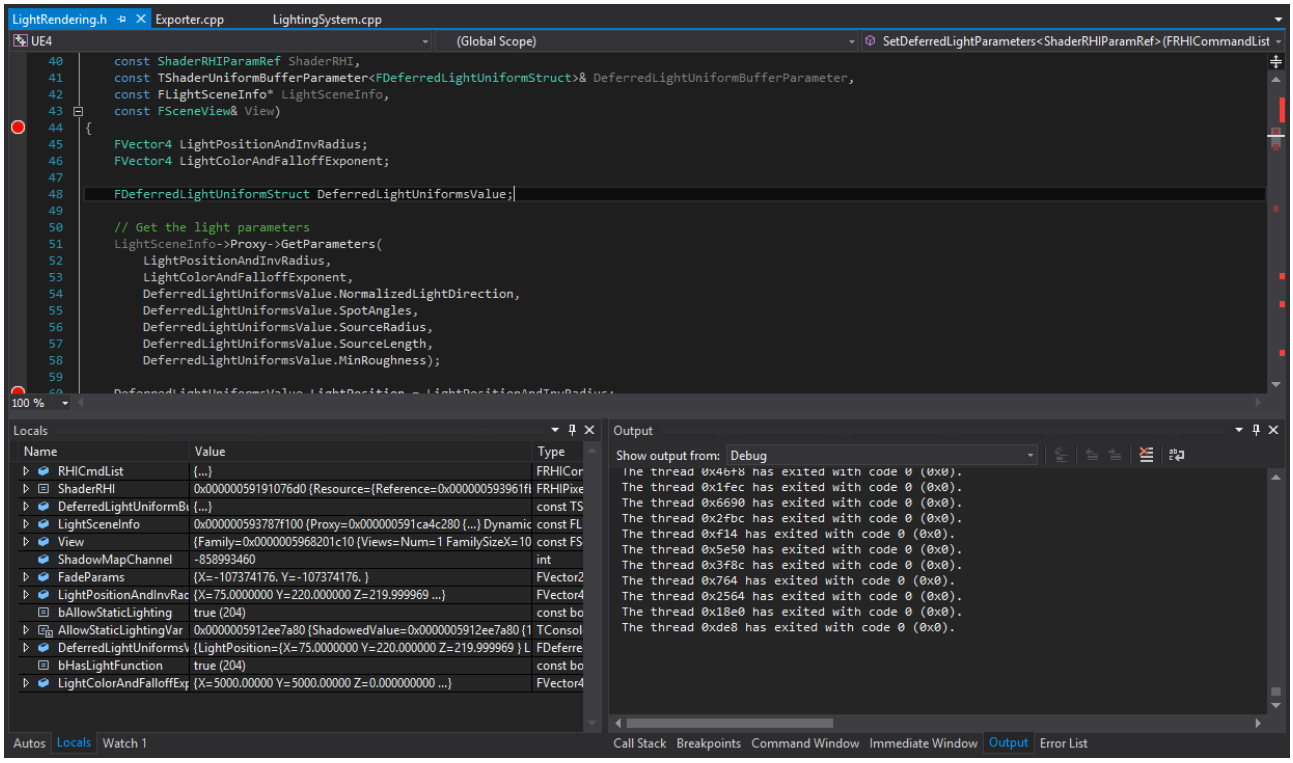


Figura 7.6: Visual Studio 2013. Editor de codi + Debugger

El debugger de VS, com molts d'altres, permet afegir punts de control els quals permetran parar l'execució temporalment quan el codi arribi a aquell punt. Això també permet observar les variables i els valors mentre dura l'execució. Podem configurar els punts de control per tal

que siguin condicionats, és a dir, que només s'activin si es compleix una certa condició. Així mateix, mentre s'està executant el codi línia a línia, podem decidir si entrar dins el codi de les funcions que s'executen o passar per sobre. Aquesta eina és molt poderosa ja que ens permet seguir amb exactitud la traça del programa i possiblement detectar més fàcilment els errors. Finalment, podem seguir els valors de les variables, com ja hem dit, però també les podem posar sota inspecció, la qual cosa permetrà veure sempre en una altra finestra els valors d'aquella variable i com aquests van canviant. També tenim la opció de seguir totes les variables locals de la funció que s'està executant en aquell moment.

7.2.3 Altres eines

Visual Studio disposa de moltes altres eines que fan molt simple i fàcil la utilització. En aquest apartat en descriurem un parell que s'han utilitzat durant aquest projecte i que han ajudat a que la feina fos més directa i ràpida. Aquestes eines són les següents:

- **Dissenyador de pestanyes obertes:** Podem tenir un nombre molt elevat de pestanyes obertes en tot moment. Això ha facilitat molt la feina durant el projecte a causa que s'han modificat i implementat molts mètodes en moltes classes diferents, per la qual cosa poder accedir de forma ràpida i simple a aquests arxius ha simplificat molta feina i cerques innecessàries.
- **Explorador de la solució:** Una solució és un conjunt de fitxers de codi i altres recursos que s'utilitzen per crear una aplicació. Els fitxers d'una solució estan estructurats jeràrquicament. L'explorador permet gestionar i navegar els fitxers de la solució gràficament. A causa de la quantitat d'arxius del codi font de l'UE, l'explorador ha permès gestionar i tenir en tot moment a l'abast aquelles classes que s'havien de modificar. Així mateix, si en qualsevol moment no es troba un arxiu, el cercador que porta incorporat permet trobar tots aquells arxius que contenen la paraula cercada al nom d'una forma ràpida.

7.2.4 Motius de l'elecció

S'ha decidit escollir Visual Studio degut a varis motius. Un dels principals d'aquests és el fet que els desenvolupadors de l'UE l'han fet servir per a desenvolupar el motor, per la qual cosa la preparació del codi font en una màquina a on està instal·lat VS és molt simple. A part d'això, també hi ha el fet que Microsoft ofereix les llicències gratuïtes per a estudiants.

Tenint en compte els motius anteriors i sumant que VS és un IDE amb molts anys de desenvolupament a darrera, així com una comunitat molt gran que permet solucionar problemes amb la IDE de forma ràpida han fet fàcil l'elecció de Visual Studio 2013 com a IDE per a desenvolupar aquest projecte.

7.3 C++

C++ és un llenguatge de programació que va ser creat, com el seu predecessor C, als laboratoris Bell de AT& T. L'autor principal va ser Bjarne Stroustrup. L'any 1980 es van afegir noves característiques al llenguatge C, entre les principals la integració de les classes, idea que va ser presa de Simula67 (per molts considerat el primer llenguatge orientat a objectes). A partir d'aquí va anar evolucionant fins que l'any 1985 va ser consolidat ocm un llenguatge orientat a objectes i anomenat C++.

Actualment existeix un estàndard (ISO C++) al qual s'han adherit la majora dels fabricants de compiladors més moderns. Existeixen també alguns intèrprets, com ara ROOT.

Una particularitat del C++ és la possibilitat de redefinir els operadors (sobrecàrrega d'operadors), i de poder crear nous tipus que es comportin com tipus fonamentals.

El nom C++ va ser proposat per Rick Mascitti l'any 1983, quan el llenguatge va ser utilitzat per primera vegada fora d'un laboratori científic. Abans s'havia fet servir el nom "C amb classes". En C++, l'expressió "C++" significa "increment de C" i es refereix al fet que C++ és una extensió de C.

Dins de C++ trobem els següents tipus fonamentals:

- **Caràcters:** *char* (també és un enter), *wchar_t*
- **Enters:** *short int*, *int*, *long int*, *long long int*
- **Nombres en coma flotant:***float*, *double*, *long double*
- **Booleans:** *bool* (cert o fals)
- **Buit:** *void*

7.3.1 Motius de l'elecció

Els motius per escollir C++ per escriure el codi del projecte és simple, UE4 està escrit en la gran majoria sobre aquest llenguatge. Tot i tenir algunes parts escrites en C, C# o Phyton, C++ és el llenguatge principal. A més, les classes que haurem de modificar estan totes escrites en C++, per la qual cosa l'el·lecció d'aquest llenguatge es troba decidida des del punt en que es va escollir dur a terme aquest projecte.

7.4 Notepad++

Notepad++ és una eina de programari lliure que surt sota la llicència GPL. La seva fortalesa és que permet crear i editar text de forma molt més complexa que el Notepad de Windows, el qual, des d'un inici, intenta superar i millorar les funcionalitats d'aquest.

Està basat en l'editor de text Scintilla, està escrit en C++ i utilitza únicament l'api Win32 i STL, la qual cosa asegura una velocitat d'execució més alta així com una mida del programa menor.

Dins les característiques més importants trobem:

- Ressaltat de sintaxi segons el llenguatge de programació escollit.
- Sistema de búsqueda PCRE (Perl Compatible Regular Expression) que permet utilitzar expressions regulars.
- Interfície gràfica minimalista i totalment personalitzable.
- Multi-document
- Gran varietat de llenguatges.

7.4.1 Motius de l'elecció

Hem escollit Notepad++ per crear i editar els diferents shaders que s'utilitzen durant el projecte ja que permet la possiblitat de configurar el ressaltat de sintaxi segons HLSL. Això ha facilitat la feina amb els shaders així com la comprensió visual i més ràpida dels mateixos.

7.5 StarUML

StarUML és un projecte de codi obert per al desenvolupament fleixble i ràpid de diagrames UML/MDA per a Windows (i altres plataformes). L'objectiu d'aquesta aplicació és construir una eina de modelatge de software lliure com alternativa al software comercial d'UML, tal com Visual Paradigm, Rational Rose, Together, etc.

7.5.1 Motius de l'elecció

S'ha escollit aquesta aplicació per davant d'altres opcions pel fet que és de programari lliure i ens permet desenvolupar tots els diagrames necessaris pel projecte, tals com: diagrames de classe, de cas d'ús, d'activitat, etc.

7.6 Gantt Project

Gantt Project és una aplicació de programari lliure, sota llicència GPL, que permet fer la organització de projectes. Els diagrames de Gantt tenen com a objectiu veure fàcilment la planificació d'un projecte mostrant en un gràfic la dedicació a cadascuna de les tasques.

Aquests diagrames no indiquen les relacions i connexions entre tasques, sinò que les posicionen en el temps, per la qual cosa aquestes relacions i dependències es poden identificar igualment.

7.6.1 Motius de l'elecció

S'ha decidit escollir aquesta eina pel fet que és programari lliure i accessible a tothom. A més el funcionament és senzill i fàcil per a organitzar la planificació del treball durant els mesos de recerca i desenvolupament.

Capítol 8

Anàlisis i disseny del sistema

Aquest capítol tracta sobre com s'ha analitzat i dissenyat el sistema partint del problema proposat tot seguint la metodologia exposada anteriorment. Per a fer-ho s'utilitzaran diagrames de cas d'ús general amb les fitxes corresponents, així com també el diagrama de classes del sistema de forma simplificada, ja que només es tindran en compte aquelles classes que s'han utilitzat i creat durant el projecte, i no totes les que formen part de l'Unreal Engine 4.

La idea principal del projecte és implementar la tècnica Lightcuts dins l'estructura interna del motor UE4. Per a fer-ho s'ha treballat sobre el renderer a més alt nivell del motor així com amb el mòdul Lightmass el qual fa el càlcul de la il·luminació indirecta dels nivells si hi ha llums estàtiques.

8.1 Diagrama i fitxes de cas d'ús

Primer de tot cal definir els diferents actors que interectuen amb el nostre sistema. En aquest cas és una decisió molt fàcil, ja que només existeix un tipus d'actor: l'usuari. Nosaltres hem modificat el codi font del motor, però aquesta part no ha d'estar disponible pels usuaris finals, per la qual cosa només definim un actor.

Tot i això, l'usuari pot arribar a fer una quantitat immensa d'operacions dins el motor, per a la qual cosa, un diagrama de casos d'ús de l'aplicació completa seria molt complicat. Per aquest motiu, a la Figura 8.1 on es mostra el diagrama observem només aquelles operacions indispensables que l'usuari ha de dur a terme per a fer funcionar correctament la nova tècnica implementada.

Podem observar les funcionalitats bàsiques que s'ha utilitzat durant la realització d'aquest projecte, així com els dos mòduls de l'Unreal Engine 4 usats. El principal és l'editor, ja que és la base per a qualsevol usuari del motor. Des de l'editor es criden els diferents mòduls que s'utilitzen. Tot i això, en el nostre cas hem utilitzat assíduament el mòdul Lightmass ja que serveix per iniciar el càlcul d'il·luminació indirecta i iniciar el pre-procés del Lightcuts.

L'editor també ens permet afegir i eliminar objectes a l'escena. En aquest cas serà molt utilitzada la funcionalitat d'afegir llums a l'escena, ja que seran les llums a partir de les quals es calcularà l'arbre de Lightcuts. Cal tenir en compte però, que l'editor de l'UE4 permet dur a terme moltes operacions. Tot i això, no apareixen al diagrama de casos d'ús ja que no són rellevants per a la comprensió de la feina que s'ha dut a terme.

Així doncs, el cas més important inicialment és "Afegir llum". Aquest procediment ens per-

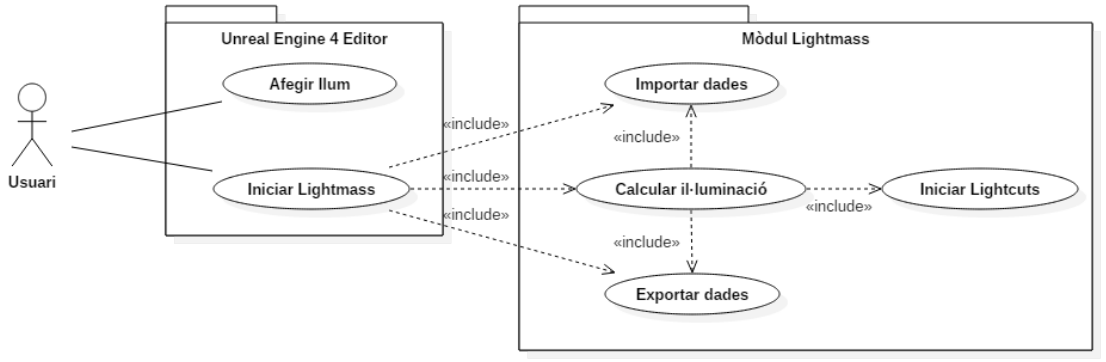


Figura 8.1: Diagrama de casos d'ús de les funcionalitats principals del sistema

metrà afegir a l'escena les llums segons les nostres preferències per tal d'obtenir el resultat d'il·luminació esperat.

Fitxa de cas d'ús	**Afegir llum**
Descripció	L'usuari vol afegir una llum al nivell
Actor	Usuari
Pre-condició	—
Flux principal	1. Seleccionar tipus de llum 2. Indicar posició 3. Introduïr paràmetres de la llum
Flux alternatiu	—
Post-condició	S'ha afegit la llum al nivell, segons la posició i paràmetres indicats, i s'ha afegit a l'estructura d'objectes del nivell

Fitxa de cas d'ús	**Iniciar Lightmass**
Descripció	L'usuari vol iniciar el procés de càlcul d'il·luminació estàtica i indirecta
Actor	Usuari
Pre-condició	—
Flux principal	1. Seleccionar el tipus de disseny (desplegable) 1.1. Seleccionar Disseny de la il·luminació
Flux alternatiu	—
Post-condició	S'ha iniciat el procés de càlcul d'il·luminació estàtica i indirecta, iniciant així conseqüentment el procés de Lightmass

Per a les següents fitxes de cas d'ús cal aclarir que, tot i que només han estat posicionades dins el Mòdul de Lightmass, l'editor de l'UE4 també du a terme una tasca d'exportar i importar dades al iniciar el procés de Lightmass, així com al rebre les dades resultants. Tot i això, s'ha decidit no posar els casos d'ús en aquesta explicació a causa que el procés és exactament

igual en els dos casos, per la qual cosa es pot entendre que aquests dos casos es duuen a terme pels dos sistemes que intervenen en la operació.

Fitxa de cas d'ús	**Importar dades**
Descripció	Procés en el qual s'importen les dades que provenen de l'editor per a fer el càlcul d'il·luminació
Actor	Unreal Engine 4 Editor / Mòdul Lightmass
Pre-condició	S'han creat els canals per a la transmissió de les dades
Flux principal	1. Crear el canal de la transmissió a partir d'un ID 2. Obrir el canal i obtenir el codi d'error corresponent 3. Si CodiError >= 0 3.1. Llegir i guardar el tamany de les dades rebudes 3.2. Llegir les dades rebudes 3.3. Tancar el canal de transmissió 4. Altrament 4.1. Indicar error per la pantalla
Flux alternatiu	—
Post-condició	S'han transmès les dades correctament des de l'editor de l'UE4 fins al mòdul de càlcul d'il·luminació Lightmass.

Fitxa de cas d'ús	**Exportar dades**
Descripció	Procés en el qual s'exporten les dades resultants del càlcul d'il·luminació cap a l'editor de l'UE4
Actor	Unreal Engine 4 Editor / Mòdul Lightmass
Pre-condició	El càlcul s'ha acabat correctament
Flux principal	1. Crear el canal de la transmissió a partir d'un ID 2. Obrir el canal i obtenir el codi d'error corresponent 3. Si CodiError >= 0 3.1. Escriure al canal el tamany de les dades a passar 3.2. Escriure les dades resultants del càlcul 3.3. Tancar el canal de transmissió 4. Altrament 4.1. Indicar error per la pantalla
Flux alternatiu	—
Post-condició	S'han transmès les dades correctament des del mòdul de càlcul d'il·luminació Lightmass a l'editor de l'UE4.

Fitxa de cas d'ús	**Calcular il·luminació**
Descripció	Procés que fa el càlcul d'il·luminació estàtica, indirecta així com el càlcul de l'arbre de Lightcuts
Actor	Mòdul Lightmass
Pre-condició	—
Flux principal	1. Si Nombre Llums > 0 1.1. Calcular lightmaps per a llums estàtiques 1.2. Calcular punts mostrals per a il·luminació indirecta 1.3. Calcular l'arbre de Lightcuts 2. Altrament 2.1. Indicar l'absència de llums per pantalla
Flux alternatiu	—
Post-condició	S'ha fet el càlcul dels lightmaps i l'arbre de Lightcuts correctament

Fitxa de cas d'ús	**Iniciar Lightcuts**
Descripció	Procés que du a terme el càlcul de l'arbre de Lightcuts
Actor	Mòdul Lightmass
Pre-condició	—
Flux principal	1. Si Nombre Llums > 0 1.1. Mentre Arbre.NoComplert == True 1.1.1. Generar següent nivell de l'arbre 1.2. Transformar l'arbre en una estructura d'array 2. Altrament 2.1. Indicar l'absència de llums per pantalla
Flux alternatiu	—
Post-condició	S'ha càlculat l'arbre de Lightcuts més òptim possible

8.2 Diagrames de classes

Un diagrama de classes es defineix com un tipus de diagrama d'estructura estàtica que descriu la organització d'un sistema mostrant les classes, els seus atributs i les relacions entre elles.

8.2.1 Mòduls funcionals

En el nostre cas, el sistema es troba dividit en dues parts ben diferenciades tal com es pot apreciar al diagrama modular de la Figura 8.2.

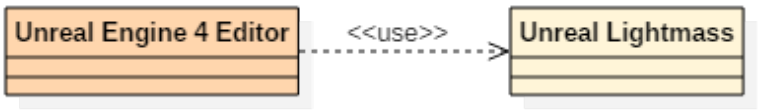


Figura 8.2: Diagrama modular del sistema

Podem apreciar clarament com el mòdul principal és l'Unreal Engine 4 Editor i com l'Unreal Lightmass és utilitzat pel primer. En aquest cas, tot i estar relacionats entre ells, els dos mòduls no comparteixen classes entre ells ja que es tracta de dos programes independents. Això provoca que els dos comparteixin classes o estructures amb el mateix nom, però no passa res pel fet que en cap moment aquestes classes s'entrellacen.

Així doncs, primerament començarem per explicar el diagrama de classes de l'Unreal Engine 4 Editor ja que comporta una part molt més gran de feina respecte l'Unreal Lightmass. Tot i això, la part de l'editor també es troba diferenciada en dues parts ben clares: Rendering i Lightmass (per la crida al procés i la recol·lecció de les dades resultants). Així doncs, també es separen aquestes dues parts per a fer més clara i entenedora l'explicació, ja que les dues queden molt diferenciades dins el codi.

Durant els següents apartats s'explicaran en detall aquelles classes que són les més importants o que són més rellevants per al treball. Els següents punts apunten a donar una visió clara i específica sobre cada classe, la seva estructura, el funcionament i la funció que tenen dins el projecte.

8.2.2 Unreal Engine 4 Editor - Rendering

La part de rendering de l'Unreal Engine és la més complicada pel fet que tot està relacionat entre si. Això provoca que tot estigui connectat en forma de bucle, fet que es posa encara més de manifest des del punt en que el motor està construït tenint molt en compte els punters proporcionats per C++. Aquest mateix fet ja el podem observar al diagrama de classes de la Figura 8.3.

Tot i això, utilitzant les descripcions aquí proporcionades juntament amb els exemples del Capítol 9 es buscarà donar a entendre la feina feta dins aquest apartat per tal de mostrar clarament el funcionament de l'aplicació.

8.2.2.1 FSceneRenderer

Aquesta classe és l'encarregada de guardar totes aquelles dades que es faran servir per a la renderització d'un frame. Això vol dir que es crea cada vegada que s'ha de renderitzar el frame,

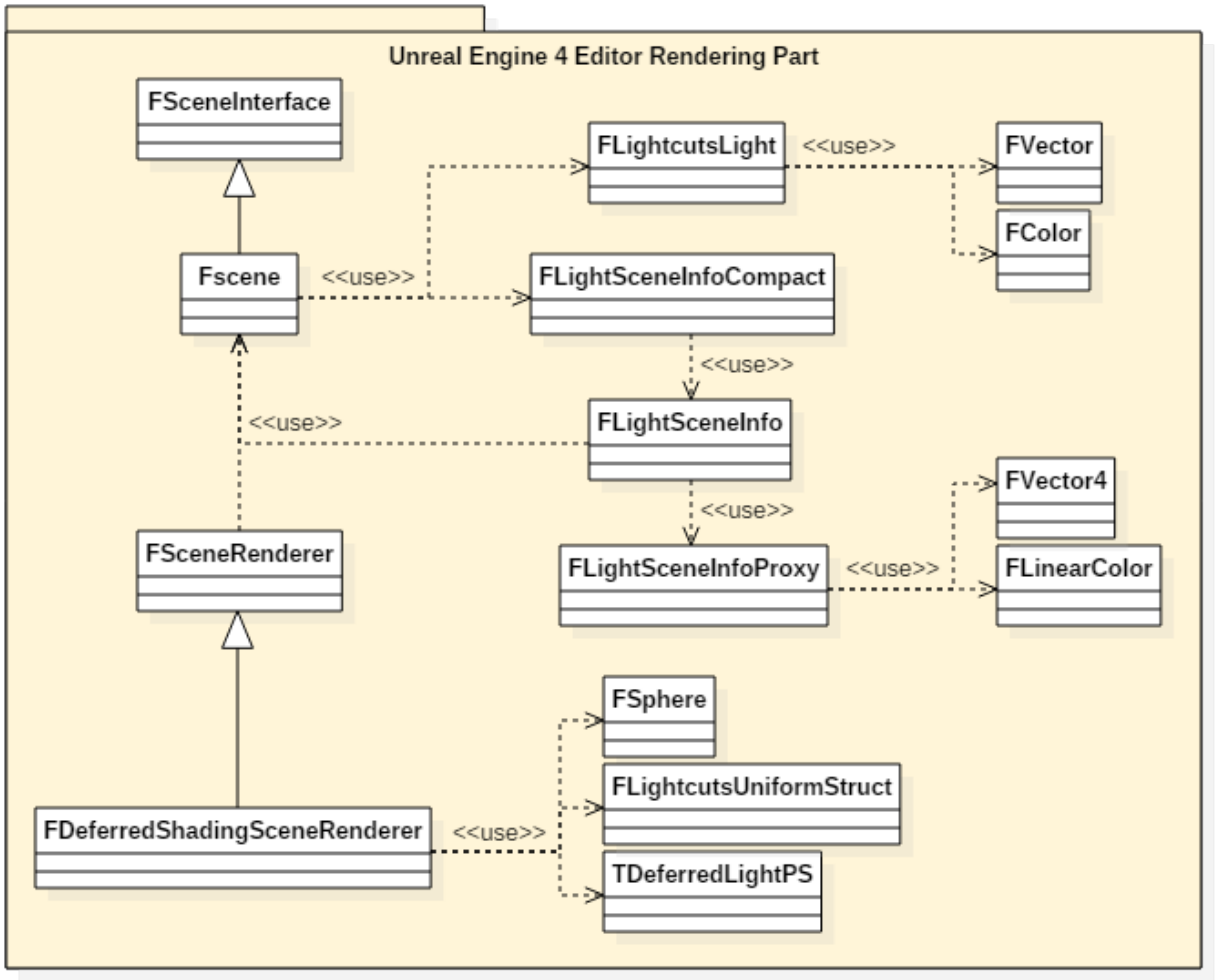


Figura 8.3: Diagrama de classes de l'UE4 Editor - Rendering Part

i un cop acabat el render s'elimina la instància de la classe. Cal tenir en compte que la classe fa la funció d'interfíce, de la qual llavors hereten les classes FDeferredShadingSceneRenderer i FForwardShadingSceneRenderer, que fan la funció dels dos tipus de render disponibles a l'UE4.

El funcionament és el següent: És creada pel Game Thread un cop iniciada la funció BeginRender. Llavors es passa al Rendering Thread que crida la funció Render(), que destrueix a l'objecte un cop acabat el render

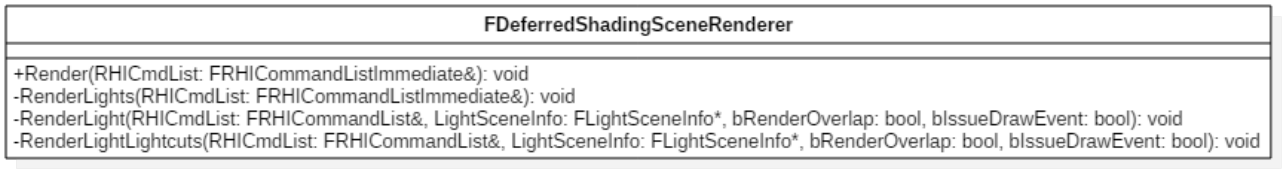
Els **atributs** més importants que trobem són:

- **FScene* FScene:** Punter a l'escena que s'està renderitzant. Permet accedir a tot el contingut de l'escena, ja siguin llums, objectes, càmeres...
- **TArray<FViewInfo> Views:** Les vistes a renderitzar. Com més càmeres hi hagi a l'escena, més vistes tindrem. Normalment només en tindrem una, però si es fa joc a pantalla partida n'hi pot haver més.

Per la part de **mètodes**, trobem els següents:

- **virtual void Render(FRHICommandListImmediate& RHICmdList):** Mètode el qual implementaran els fills i que servirà per a renderitzar el frame actual del joc.

8.2.2.2 FSceneRenderer



Classe que hereta de FSceneRenderer (Apartat 8.2.2.1). En aquest cas implementa un tipus de renderitzat anomenat Deferred Shading. Aquest renderitzat funciona de forma que renderitza pixel per pixel, a diferència del Forward Rendering. Quan s'utilitza Forward Rendering, es passen tots els objectes d'una escena a la GPU. Cada un d'aquests objectes passen pel Vertex Shader, pel Geometry Shader i finalment pel Fragment Shader, o Pixel Shader, a on s'hi fan els últims tractaments del rendering abans de ser mostrats per pantalla. D'altra banda, el Deferred Rendering funciona igual fins que arribem al Fragment Shader. En aquest cas no s'hi fa el càlcul d'il·luminació, sinò que es fan tres renders diferents (Depth, Normals i Color) als quals se'ls hi aplica la il·lumiació un cop fets per tal d'obtenir el render final. Això provoca que es millori el rendiment ja que amb Forward Rendering es farà el càlcul d'il·luminació per a tots els fragments de geometria que s'hagin obtingut, la qual cosa pot ser molt car en temps de càlcul i fer operacions innecessàries ja que molts d'aquests fragments quedaran eliminats quan es faci el Depth test. Així doncs, amb el Deferred Rendering guanyem en eficiència a la hora de calcular la il·luminació tot i que es perdi la opció de calcular anti-aliasing.

Aquesta classe no disposa d'atributs propis ja que tots els que necessita els hereta de FSCeneRenderer. per tant, els seus **mètodes** més importants són:

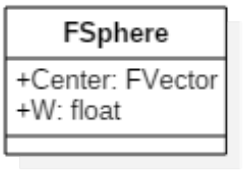
- **virtual void Render(FRHICommandListImmediate& RHICmdList) override:** Renderitza l'escena per a totes les càmeres que es troben a dins.

- **void RenderLights(FRHICommandListImmediate& RHICmdList):** Renderitza la il·luminació de l'escena.

- **void RenderLight(FRHICommandList& RHICmdList, const FLightSceneInfo* LightSceneInfo, bool bRenderOverlap, bool bIssueDrawEvent):** Utilitzat per RenderLights() per a renderitzar una llum al buffer de color de l'escena actual.

- **void RenderLightLightcuts(FRHICommandList& RHICmdList, const FLightSceneInfo* LightSceneInfo, bool bRenderOverlap, bool bIssueDrawEvent):** Utilitzat per RenderLights() per a renderitzar el color del càlcul resultant que es du a terme amb l'arbre de Lightcuts. Això vol dir que renderitza totes les llums de cop d'una sola passada a diferència de RenderLight().

8.2.2.3 FSphere

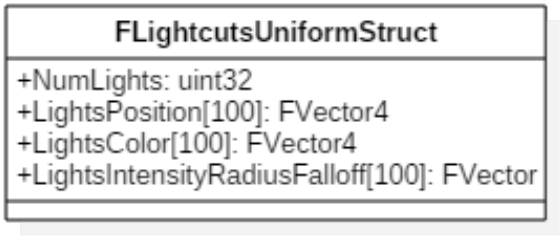


Aquesta classe implementa una esfera bàsica. Simplement guarda els atributs necessàris per a representar-la i alguns mètodes per a comprovar si un punt es troba dins l'esfera. Tot i això, és important ja que té un gran pes a l'hora de renderitzar les llums segons el mètode de Lightcuts, ja que s'ha de fer el render segons una esfera que englobi totes les llums que trobem a l'arbre de Lightcuts.

Dins d'aquesta classe només s'han utilitzat els seus **atributs**, que són els següents:

- **FVector Center:** Guarda en un vector de 3 posicions la posició 3D a on es troba el centre de l'esfera.
- **float W:** Indica el radi de l'esfera.

8.2.2.4 FLightcutsUniformStruct



En aquest cas no estem parlant exactamet d'una classe, ja que es tracta més d'una tupla, creada a partir de mètodes proporcionats per l'UE4 per tal de passar dades fàcilment als diferents shaders. Tot i així, pensem que és important incloure-la aquí, ja que ens ha permés passar totes les propietats de les llums de l'arbre de Lightcuts fàcilment cap als shaders per a poder fer el càlcul final del color. Així doncs, com veurem més endavant al Capítol 9, tot i no ser una classe, té molta importància en el desenvolupament del projecte.

Veiem que s'hi guarden dades en taules ja que, com s'ha comentat anteriorment, l'arbre de Lightcuts es guarda en una estructura de tipus Array per a fer més fàcil aquest tractament de dades que estem veient en aquesta tupla. Cal detallar també el fet que les taules tenen una grandària de 100 posicions a causa de les característiques del codi. A causa de com està construïda l'estructura de la tupla, no permet passar taules dinàmiques, ja que es fa servir la taula per compilar els shaders. Per tant, es va optar per posar una mida elevada que pogués deixar lloc a totes les possibles proves que es s'han dut a terme. En cas de necessitar-se més espai, et pot codificar en forma de textures.

Aquesta classe no utilitza mètodes, per la qual cosa els seus **atributs** són els següents:

- **uint32 NumLights:** Indica el nombre de llums que formen l'arbre de Lightcuts. uint32 és typedef d'unsigned int, el qual serveix per a qualsevol plataforma.
- **FVector4 LightsPosition[100]:** Taula a on es guarda la posició de les llums de l'arbre.
- **FVector3 LightsColor[100]:** Taula a on es guarda el color de les llums de l'arbre.
- **FVector LightsIntensityRadiusFalloff[100]:** Taula a on es guarden les dades referents a la intensitat, el radi i l'exponent de Falloff de cada llum. Entenem Falloff com el ritme al qual va decreixent la intensitat segons la distància a la posició de la llum.

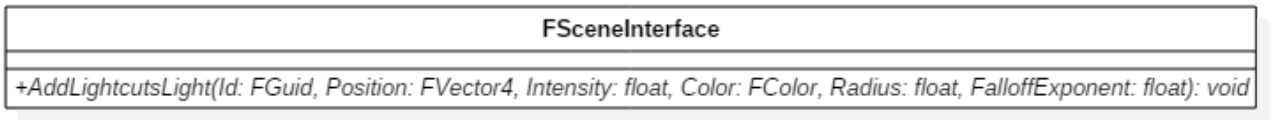
8.2.2.5 TDeferredLightPS

Classe que implementa les funcionalitats necessàries per inicialitzar i passar les dades que s'utilitzaran per compilar els Pixel Shader. En aquest cas s'hi han afegit funcionalitats per tal que es pugui utilitzar tant amb el render propi de les llums de l'UE4, com amb el render segons l'arbre de Lightcuts per les llums puntuals.

D'aquesta classe hem utilitzat els **mètodes** següents:

- **void SetParameters(FRHICommandList& RHICmdList, FSceneView& View, FLightSceneInfo* LightSceneInfo):** Aplica els paràmetres que rebrà el shader segons el que es veu des de la càmera que mostra **View** i segons les característiques de l'escena donades per **LightSceneInfo**. **RHICmdList** permetrà afegir comandes a la *Rendering Hardware Interface*, que s'encarrega de compilar els shaders.
- **void SetLightcutsParameters(FRHICommandList& RHICmdList, FSceneView& View, FLightSceneInfo* LightSceneInfo):** Igual que l'anterior, aplica els paràmetres que rebrà el shader. Tot i així en aquest cas es varia l'estructura que es passarà al shader per tal de fer possible que el shader rebi tot l'arbre de Lightcuts.

8.2.2.6 FSceneInterface



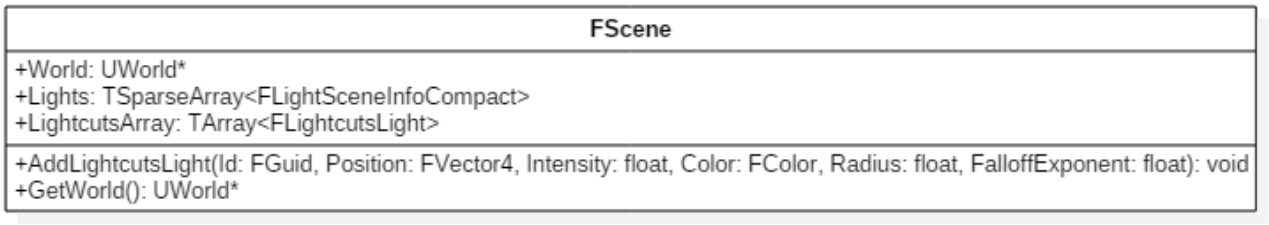
Més que una classe és una interfície tal com indica el nom. En aquest cas serveix d'interfície per a les classes que implementen un gestor de l'escena. D'aquesta forma es proporcionen els mètodes bàsics que es necessiten per gestionar totes les dades, ja siguin per afegir o eliminar

llums, objectes, càmeres, etc.

En aquesta interfície s'ha afegit un mètode que ens ha permès gestionar les llums de l'arbre de Lightcuts de l'escena. Aquest **mètode** és el següent:

– **virtual void AddLightcutsLight(FGuid Id, FVector4 Position, float Intensity, FColor Color, float Radius, float FalloffExponent):** Afegeix una llum a l'estructura de l'arbre de Lightcuts. FGuid indica l'identificador global de la llum.

8.2.2.7 FScene



Implementa la interfície de FSceneInterface per a ser utilitzada durant el rendering. La funció és guardar totes les dades que es troben a l'escena actual i que es necessiten per a fer el render correctament. Tot i això, també té atributs i mètodes propis que proporcionen una eina per accedir a totes les dades referents de l'escena.

Així doncs, els **atributs** més importants que s'han utilitzat són:

– **UWorld* World:** Atribut opcional però que en general sempre apunta al món en el qual es troba l'escena. Entenem com a món l'objecte que quedaria a dalt de tot de la jerarquia, tenint els objectes de l'escena per sota. Més endavant s'explica la classe UWorld (Apartat 8.2.3.4).

– **TSparseArray<FLightSceneInfoCompact> Lights:** Guarda totes les llums que aparèixen a l'escena, siguin del tipus que siguin.

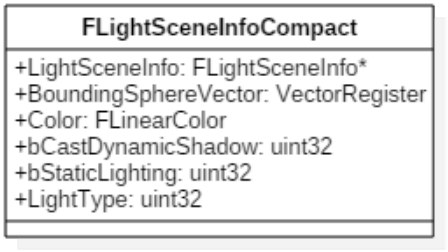
– **TArray<FLightcutsLight> LightcutsArray:** Guarda l'estructura de l'arbre de Lightcuts i ens permet accedir a totes les llums que formen l'arbre.

D'altra banda, dins els **mètodes** més rellevants trobem:

– **void AddLightcutsLight(FGuid Id, FVector4 Position, float Intensity, FColor Color, float Radius, float FalloffExponent):** Afegeix una llum a l'estructura de l'arbre de Lightcuts.

– **UWordl* GetWorld():** Retorna un punter a l'objecte món del qual forma part l'escena actual.

8.2.2.8 FLightSceneInfoCompact

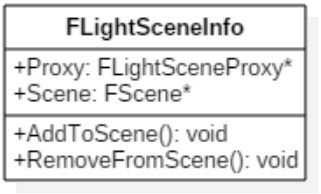


Classe que guarda la informació necessària per a dur a terme una interacció entre una llum i els objectes 3D de l'escena.

En aquest cas només s'han fet servir els **atributs** següents:

- **FLightSceneInfo* LightSceneInfo:** Guarda la informació necessària per a renderitzar una llum.
- **VectorRegister BoundingSphereVector:** Indica la posició i el radi de la llum per tal de poder calcular l'esfera delimitadora d'acció de la llum.
- **FLinearColor Color:** Guarda el color de la llum en format lineal.
- **uint32 bCastDynamicShadow:** Indica si la llum provoca ombres dinàmiques.
- **bStaticLighting:** Indica si la llum és estàtica o dinàmica. En el nostre cas, estaran marcades com a estàtiques però el funcionament serà com una llum dinàmica. Això està fet així per a poder fer servirs el mòdul de Lightmass per al càlcul de l'arbre, ja que només accepta llums estàtiques.
- **uint32 LightType:** Indica el tipus de llum amb el qual estem tractant. Pot ser LightType_Directional, LightType_Point o LightType_Spot.

8.2.2.9 FLightSceneInfo



Guarda la informació necessària per a renderitzar una llum. Fa la funció de classe mirall dins el rendering thread de la classe ULightComponent que es troba dins el game thread.

Els **atributs** que s'han fet servir són els següents:

- **FLightSceneProxy* Proxy:** El proxy de l'escena a on pertany la llum. Dades necessàries per al render.

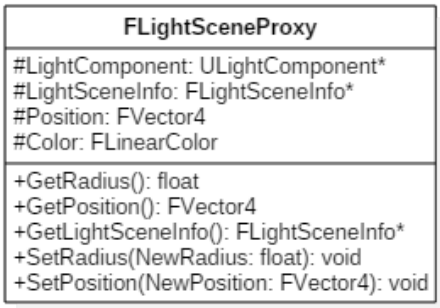
– **FScene* Scene:** Punter de l'escena a on podem trobar la llum.

D'altra banda, els **mètodes** més característics d'aquesta classe són:

– **void AddToScene():** Afegeix la llum actual a l'escena.

– **void RemoveFromScene():** Elimina la llum actual de l'escena.

8.2.2.10 FLightSceneProxy



Encapsula totes les dades que s'utilitzen per a renderitzar una llum de forma paral·lela al game thread.

Els **atributs** més utilitzats o importants són els següents:

– **ULightComponent* LightComponent:** Referencia a l'objecte que es troba al game thread i que representa la llum.

– **FLightSceneInfo* LightSceneInfo:** Guarda una referència a la informació de l'escena on es troba inclosa la llum.

– **FVector4 Position:** Indica la posició de la llum dins el nivell del joc.

– **FLinearColor:** Color de la llum.

D'altra banda, alguns dels **mètodes** que trobem dins la classe són:

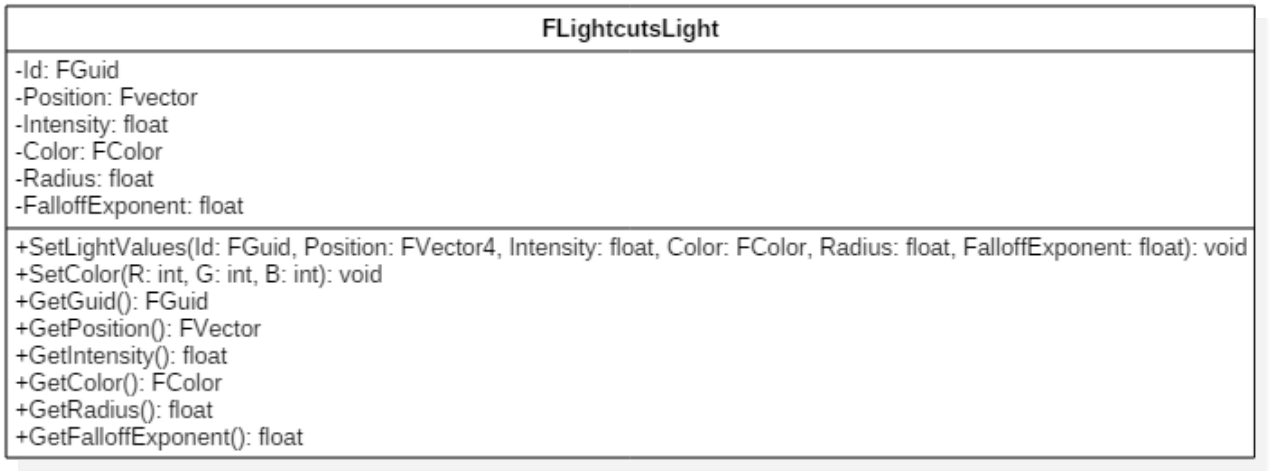
– **float GetRadius():** Retorna el radi d'acció de la llum.

– **FVector4 GetPosition():** Retorna la posició de la llum.

– **void SetRadius(float NewRadius):** Actualitza el valor del radi.

– **void SetPosition(FVector4 NewPosition):** Actualitza el valor de la posició.

8.2.2.11 FLightcutsLight



Aquesta classe permet guardar fàcilment totes aquelles dades que necessitem per a fer funcionar la tècnica de Lightcuts. Així doncs, guardarà la informació d'aquelles llums que formen part de l'arbre de Lightcuts que provenen del càlcul fet al mòdul de Lightmass.

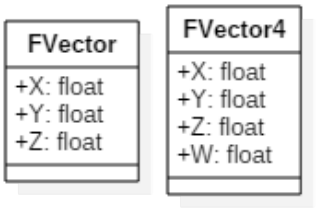
Els **atributs** són els següents:

- **FGuid Id:** Identificador global únic de la llum.
- **FVector Position:** Posició dins el món de la llum.
- **float Intensity:** Intensitat de la llum.
- **FColor Color:** Color de la llum.
- **float Radius:** Radi d'acció de la llum.
- **FalloffExponent:** Valor de decreixement de la intensitat.

D'altra banda, els **mètodes** que trobem dins la classe són:

- **void SetLightValues(FGuid Id, FVector4 Position, float Intensity, FColor Color, float Radius, float FalloffExponent):** Actualitza els valors de la llum.
- **void SetColor(int R, int G, int B):** Actualitza el color.
- **FGuid GetGuid():** Retorna l'identificador global de la llum.
- **FVector GetPosition():** Retorna la posició de la llum.
- **float GetIntensity():** Retorna la intensitat de la llum.
- **FColor GetColor():** Retorna el color de la llum.
- **float GetRadius():** Retorna el radi d'acció de la llum.
- **float GetFalloffExponent():** Retorna el valor de decreixement de la intensitat de la llum.

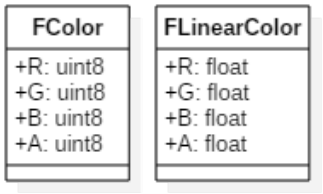
8.2.2.12 FVector i FVector4



Aquestes classes donen la funcionalitat d'un vector de 3D i 4D respectivament. La utilització durant el projecte ha estat de gran importància per guardar dades referents a posicions, color i altres dades importants de les llums.

Com a atributs només tenen els components dels vectors i no hi ha cap mètode d'especial rellevància per la nostra investigació.

8.2.2.13 FColor i FLinearColor



Igual que amb els vectors, aquestes dues classes apunten, en la part més bàsica, a tenir la mateixa funcionalitat: Guardar de forma fàcil i accessible el color en format RGBA de les llums de l'escena. Tot i així, la diferència entre les dues és que FLinearColor s'utilitza més a l'apartat de rendering un cop s'han fet operacions per convertir el color original codificat en 8 bits per passar-lo a una codificació de 32 bits que trobem a FLinearColor, ja que els nombres augmenten bastant de magnitud. Això ho veurem més clarament al Capítol 9 quan s'expliqui la part referent a passar dades als shaders.

Les dues classes utilitzen els atributs bàsics per guardar el color en format RGBA. Per la part de FColor trobem 4 variables del tipus uint8, mentre que per la part de FLinearColor tenim 4 variables de tipus float.

8.2.3 Unreal Engine 4 Editor - Lightmass

En aquest apartat ens centrarem en descriure les classes que trobem a la part de Lightmass dins l'editor de l'UE4. La funció més important d'aquesta secció és passar les dades correctament al mòdul de Lightmass i llavors fer les operacions necessàries amb els resultats obtinguts per tal que l'escena tingui en tot moment disponible l'arbre de Lightcuts.

Tal com es pot apreciar a la Figura 8.4, l'estructura de classes és molt més simple que a la part de Rendering. Tot i això, no implica que aquesta part sigui menys important, ja que es porten a terme operacions importants per tal d'aconseguir el bon funcionament de la tècnica Lightcuts.

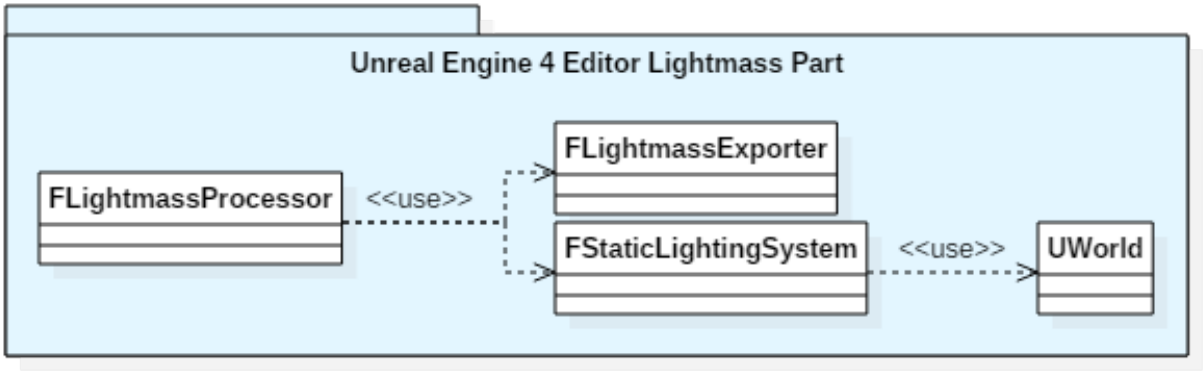
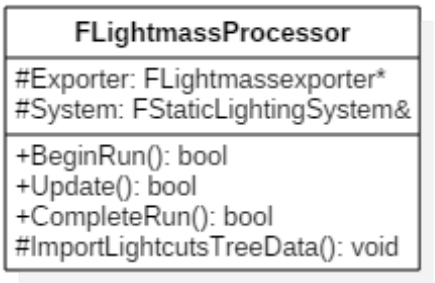


Figura 8.4: Diagrama de classes de l'UE4 Editor - Lightmass Part

8.2.3.1 FLightmassProcessor



Aquesta classe és l'encarregada de dur a terme tots els processos necessàries per a iniciar el mòdul de Lightmass. S'encarrega de cridar el procés i gestionar les dades obtingudes un cop acabat el càlcul. Així doncs, té una gran importància dins el projecte, ja que tot el tractament de dades necessari per a passa les dades del mòdul de Lightmass a l'escena de l'editor es fa aquí.

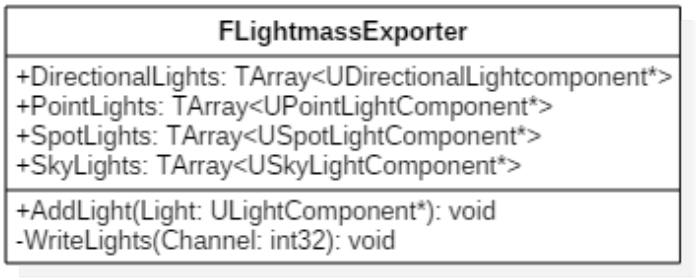
Per la part dels **atributs** trobem que s'han utilitzat els següents:

- **FLightmassExporter* Exporter:** Classe que ens permet exportar les dades necessàries per al càlcul d'il·luminació.
- **FStaticLightingSystem& System:** Sistema d'il·luminació estàtica el qual ens donarà accés a l'escena, a la qual afegirem l'arbre de Lightcuts un cop acabat el càlcul gràcies a aquest objecte.

D'altra banda, les **funcions** més importants que té són:

- **bool BeginRun():** Inicia el procés de càlcul de Lightmass.
- **bool Update():** Comprova l'estat actual del procés de Lightmass.
- **bool CompleteRun():** Finalitza el procés de Lightmass i executa totes les importacions de dades necessàries.
- **void ImportLightcutsTreeData():** Importa les dades resultants del càlcul d'il·luminació referents a l'arbre de Lightcuts i les afegeix a l'escena actual.

8.2.3.2 FLightmassProcessor



La funció principal d'aquesta classe és la de gestionar com és fa l'exportació de dades de l'editor cap al mòdul de Lightmass. És un procés important ja que s'han de passar totes les dades necesssàries per a poder fer el càlcul de la il·luminació estàtica correctament. Podem veure com s'hi troben totes les llums, separades per tipus, que tenim a l'escena.

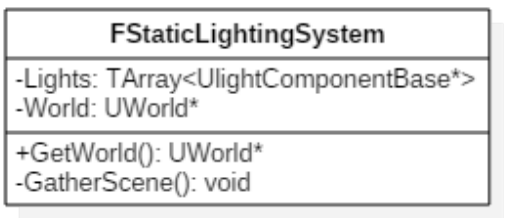
Els **atributs** més importants que es troben dins aquesta classe són els següents:

- **TArray<UDirectionalLightComponent*> DirectionalLights:** Llums direccionals que es troben a l'escena.
- **TArray<UPointLightComponent*> PointLights:** Llums puntuals que es troben a l'escena.
- **TArray<USpotLightComponent*> SpotLights:** Llums focals que es troben a l'escena.
- **TArray<USkyLightComponent*> SkyLights:** Llums usades per a representar la llum provinent del cel.

D'altra banda, els **mètodes** més usats són:

- **void AddLight(ULightComponent* Light):** Afegeix una llum a la llista de llums que toqui segons el seu tipus.
- **void WriteLights(int32 Channel):** Escriu les llums en un canal el qual serà llegit posteriorment pel mòdul de Lightmass per a importar les llums i fer els càlculs que calgui.

8.2.3.3 FLightmassProcessor



Aquesta classe guarda l'estat actual del sistema d'il·luminació estàtica. Això vol dir que té accés a totes les llums, objectes, materials i tot allò que afecta a la il·luminació de l'escena.

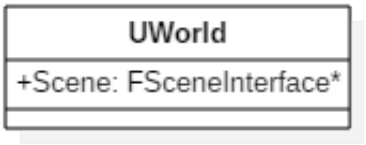
Els **atributs** rellevants pel projecte que hi trobem són:

- **TArray<ULightComponentBase*> Lights:** Llistat de totes les llums que es poden trobar a l'escena, independentment del tipus que siguin.
- **UWorld* World:** El món en el qual aquesta llum va ser creada.

D'altra banda, els **mètodes** més interessants pel nostre treball són:

- **UWorld* GetWorld():** Retorna un punter cap al món a on s'ha creat aquesta llum.
- **void GatherScene():** Obté l'escena que s'enviarà cap a l'exporter.

8.2.3.4 UWorld



El món és l'objecte de més alt nivell dins la jerarquia de l'UE4 el qual representa un mapa del joc en el qual s'hi col·locaran tots els objectes i derivats. Un món pot ser un sol nivell persistent amb una llista opcional de nivells més petits que es carregaran i descarregaran en streaming utilitzant Blueprints. També pot ser una col·lecció de nivells que estigui organitzada dins una composició de món global. En general, en un joc només existirà un sol món.

En aquest cas no ens interessen els mètodes disponibles, ja que l'**atribut** que necessitem és públic:

- **FSceneInterface* Scene:** Escena de la qual s'han extret les llums per a fer el càlcul i a la qual s'insertarà l'arbre de Lightcuts un cop obtingut.

8.2.4 Unreal Lightmass

En aquesta secció descriurem el diagrama de classes que podem trobar a la Figura 8.5, en el qual s'hi troben representades les classes més importants que s'han utilitzat per modificar i afegir contingut al mòdul d'il·luminació estàtica Lightmass.

Es pot observar a simple vista que algunes de les classes comparteixen nom amb les que ja hem vist anteriorment. Això és a causa que l'editor i aquest mòdul són dos programes independents, per la qual cosa no importa si tenen el mateix nom. Aquest fet també indica que una de les parts més importants en aquest mòdul és traslladar les dades de forma correcta cap a l'editor un cop obtinguts els resultats. D'altra banda, la semblança amb el nom tampoc és coincidència, ja que les classes amb el mateix nom tenen una funcionalitat quasi exacta amb la de l'editor.

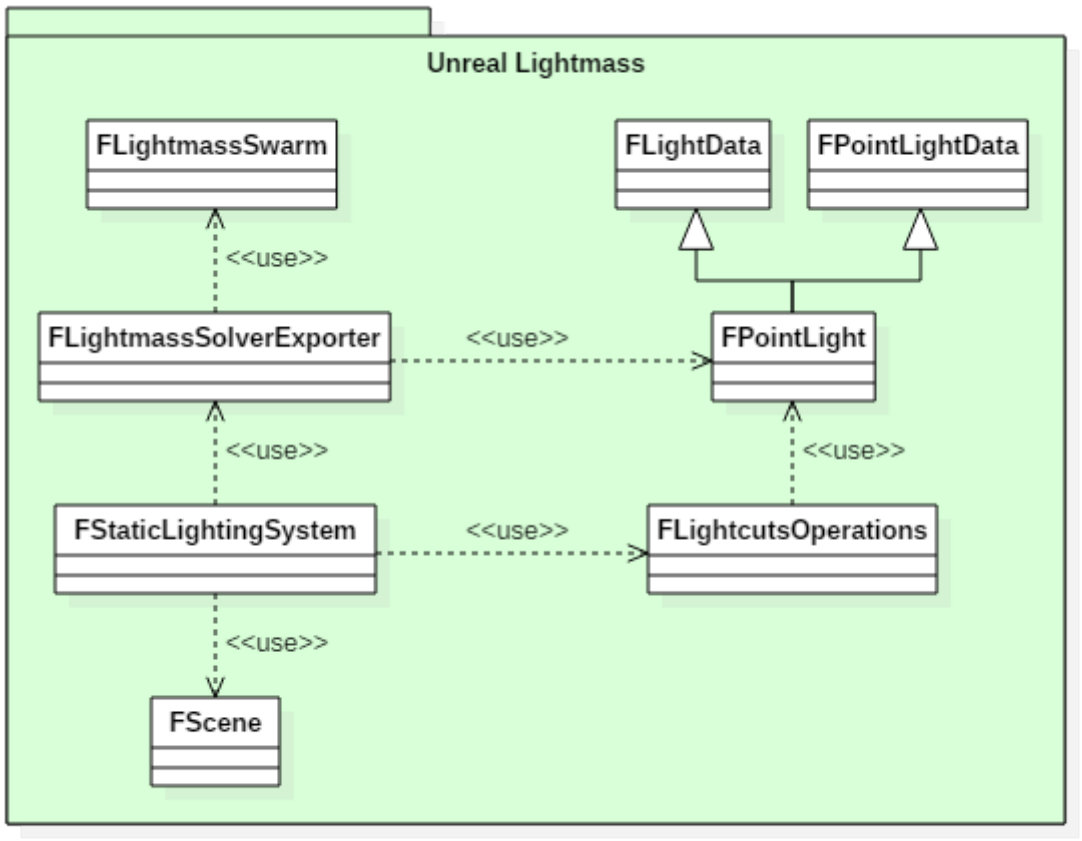


Figura 8.5: Diagrama de classes del mòdul de càlcul d'il·luminació estàtica Lightmass

8.2.4.1 FStaticLightingSystem

FStaticLightingSystem
-Lights: TArray<FLight*> -Scene: FScene& -Exporter: FLightmassSolverExporter& -LightcutsOp: FLightcutsOperations
+FStaticLightingSystem(InOptions: FLightingBuildOptions, InScene: FScene&, InExporter: FLightmassSolverExporter&, InNumThreads: int32) +GetExporter(): FLightmassSolverExporter& -ExportNonMappingTasks(): void

Aquesta classe és la més important pel nostre projecte dins el mòdul Lightmass. És l'encarregada d'iniciar el procés de càlcul de l'arbre de Lightcuts així com de totes les altres operacions necessàries per a calcular la il·luminació estàtica de l'escena.

Com ja s'ha comentat, aquesta classe comparteix nom amb una classe de l'editor. Podem comprovar el perquè d'aquest fet observant els primers atributs que hi trobem. A més la funcionalitat ve a ser la mateixa, donat que també guarda l'estat actual de la il·luminació de l'escena.

Si observem els **atributs**, observem que tenim els següents:

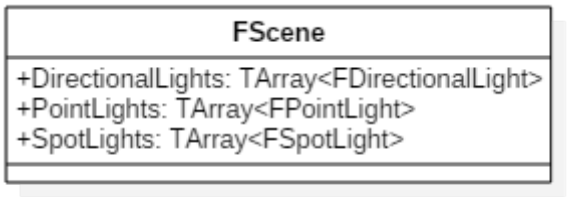
– **TArray<FLight*> Lights:** Les llums que es troben en el món del qual estem calculant la il·luminació.

- **FScene& Scene:** Escena d'entrada que descriu la geometria, materials i les llums disponibles.
- **FLightmassSolverExporter& Exporter:** Exporter que ens servirà per enviar les dades de nou cap a l'editor de l'Unreal.
- **FLightcutsOperations LightcutsOp:** Permet utilitzar els mètodes creats per a construir l'arbre de Lightcuts.

D'altra banda, per la part dels **mètodes** més rellevants, trobem:

- **FStaticLightingSystem(const FLightingBuildOptions& InOptions, class FScene& InScene, class FLightmassSolverExporter& InExporter, int32 InNumThreads):** Constructor de la classe. Col·locat aquí ja que inicialitza el sistema i construeix la il·luminació estàtica basant-se en les opcions proporcionades. **InOptions** indica les opcions, **InScene** conté totes les llums i els objectes, **InExporter** proporciona l'exporter que s'utilitzarà per retornar les dades a l'editor i **InNumThreads** indica el nombre de threads concurrents que es faran servir per a construir la il·luminació estàtica.
- **FLightmassSolverExporter& GetExporter():** Retorna l'exporter que enviarà les dades cap a l'editor.
- **void ExportNonMappingTasks():** Exporta totes aquelles tasques que no siguin mapes (Lightmaps, shadowmaps...). En el nostre cas s'utilitzarà per enviar l'arbre de Lightcuts resultant del càlcul.

8.2.4.2 FScene

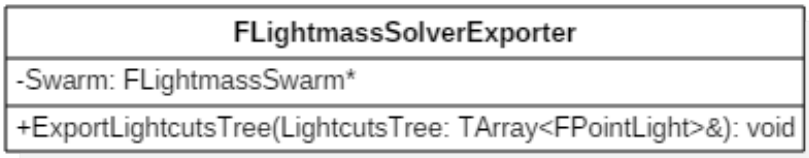


Classe mirall a la que trobem a l'Unreal Editor. En aquest cas però, no té tantes funcionalitats i té una funció de contenidor. És a dir, guarda totes les dades de l'escena necessàries per al càlcul de la il·luminació estàtica. Això vol dir que el més important que hi podrem trobar seran objectes 3D, materials i llums.

D'entre tots els **atributs** que trobem a la classe, els més rellevans per nosaltres en el tema d'il·luminació seran:

- **TArray<FDirectionalLight> DirectionalLights:** Contenidor de les llums direccionals de l'escena.
- **TArray<FPointLight> PointLights:** Contenidor de les llums puntals de l'escena.
- **TArray<FSpotLight> SpotLights:** Contenidor de les llums focals de l'escena.

8.2.4.3 FLightmassSolverExporter



Aquesta classe conté aquells mètodes necessàris per a fer l'exportació de dades resultants dels càlculs cap a l'editor de l'Unreal Engine 4.

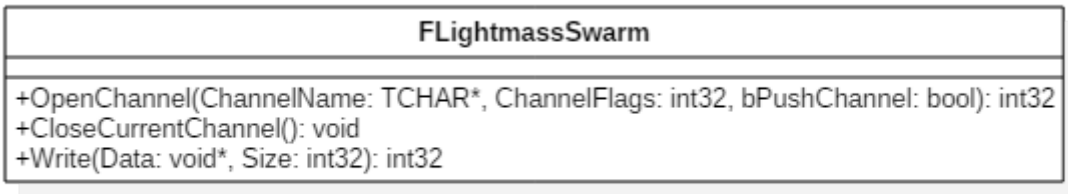
L'**atribut** més important d'aquesta classe és:

- **FlightmassSwarm* Swarm:** Permet exportar els resultats cap a l'editor.

El **mètode** que ens interessa d'aquesta classe és el següent:

- **void ExportLightcutsTree(TArray<FPointLight>& LightcutsTree):** Exporta el contingut de l'arbre de Lightcuts en forma d'array cap a l'editor.

8.2.4.4 FLightmassSwarm



Swarm (versió curta de UnrealSwarm) és una aplicació independent de l'Unreal Engine que distribueix unitats de feina a diferents màquines que es trobin dins la nostra xarxa. És utilitzat juntament amb Lightmass per a fer els càlcus d'il·luminació estàtica. En el nostre cas, només ho farem servir com a pont entre el mòdul Lightmass i l'editor. Utilitzant les eines que proporciona aquesta classe podrem exportar els resultats cap a l'editor utilitzant els canals de transmissió.

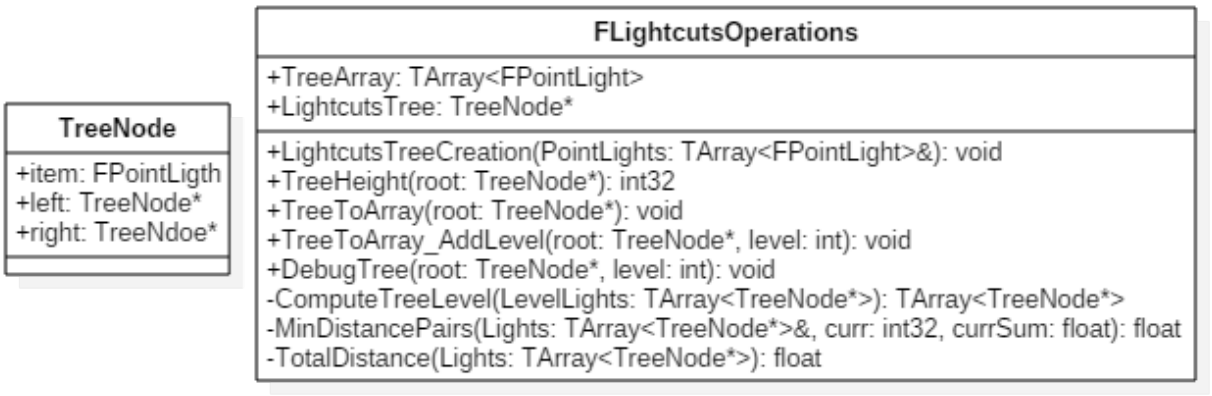
No s'han utilitzat atributs en aquesta classe. Així doncs els **mètodes** utilitzats són:

- **int32 OpenChannel(TCHAR* ChannelName, int32 ChannelFlags, bool bPushChannel):** Obre un nou canal i opcionalment el col·loca a sobre la pila de canals. **ChannelName** indica el nom del canal, **ChannelFlags** diu si ha de ser d'escriptura, lectura, etc; i si **bPushChannel** és cert posiciona el canal a dalt la pila de canals.

- **void CloseCurrentChannel():** Tanca el canal que es troba al top de la pila de canals.

- **int32 Write(void* Data, int32 Size):** Escriu les dades, que se li han passat, al canal tot indicant la mida d'aquestes dades.

8.2.4.5 FLightcutsOperations



Classe creada específicament per aquest projecte i que dòna accés a una sèrie d'atributs i funcions que permeten dur a terme la creació de l'arbre de Lightcuts.

Primer de tot cal introduïr la tupla TreeNode, ja que en serveix per crear i gestionar l'arbre que crearem. És una tupla simple en la qual dins del node trobem l'item, que és una llum de tipus FPointLight, i els dos punters als fills que són punters del tipus TreeNode.

Els **atributs** utilitzats són els següents:

- **TArray<FPointLight> TreeArray:** Inicialment buit, guarda l'arbre en forma d'array un cop aquest ja ha estat calculat.
- **TreeNode* LightcutsTree:** Conté l'arbre de Lightcuts en forma d'arbre binari.

D'altra banda, els **mètodes**, juntament amb el pseudocodi, que permeten dur a terme la construcció de l'arbres són:

- **void LightcutsTreeCreation(TArray<FPointLight>& PointLights):** Calcula l'arbre de Lightcuts a partir de les llums puntuals de l'escena. Utilitza una llista d'ítems TreeNode que s'utilitzen per crear l'arbre. Quan només queda un ítem a la llista indica que s'ha finalitzat la creació de l'arbre.

```
LightcutsTreeCreation(Llums)
{
   Inicialitzar llums
   Mentre (Arbre.Nivell() > 1)
   {
      Calcular parelles optimes
      Crear nou nivell de l'arbre
   }
   Retornar arbre
}
```

- **int32 TreeHeight(TreeNode* root):** Retorna l'altura de l'arbre.
- **void TreeToArray(TreeNode* root):** Converteix l'arbre binari a array i el desa a TreeArray.

– **void TreeToArray_AddLevel(TreeNode* root, int level):** Afegeix un nivell de l'arbre binari a l'estructura d'array de TreeArray. És utilitzat per TreeToArray per a fer més fàcil la conversió.

– **void DebugTree(TreeNode* root, int level):** Escriu en un fitxer de text el contingut de l'arbre.

– **TArray<TreeNode*> ComputeTreeLevel(TArray<TreeNode*> LevelLights):** Calcula un nivell de l'arbre, tenint en compte les distàncies entre llums, a partir de les llums que trobem a l'array LevelLights.

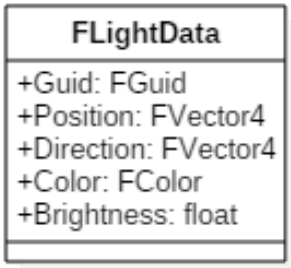
```
ComputeTreeLevel(Llums)
{
   i = 0
   Mentre (i < Llums.Num() - 1)
   {
      Crear nova llum
      Assignar valors a la nova llum utilitzant la parella {i,i+1}
      Crear nou node amb la nova llum com a item i la parella {i,i+1} com a fills
      Afegir node al nou nivell de l'arbre
   }
   Si (Llums.Num() == Senar)
   {
      Afegir ultima llum de la llista Llums al nou nivell
   }
}
```

– **float MinDistancePairs(TArray<TreeNode*>& Lights, int32 curr, float currSum):** Busca les parelles que provoquen que la distància entre aquestes sigui mínima. La variable **curr** indica quin nivell de l'arbre s'està calculant i **currSum** indica la suma mínima obtinguda fins al moment.

```
MinDistancePairs(Llums, Index, SumaActual)
{
   Si (Index == LLums.Num() - 1)
   {
      Calculem la distancia que produeixen les parelles actuals
   }
   Altrament
   {
      Per (i = Index fins Llums.Num() - 1, pas 1)
      {
         Llums = IntercanviarLlums(Index, i)
         NovaDistancia = MinDistancePairs(Llums, Index+1, SumaActual)
         Si (NovaDistancia < SumaActual)
         {
            SumaActual = NovaDistancia
         }
         Llums = IntercanviarLlums(Index, i)
      }
   }
}
```

– **float TotalDistance(TArray<TreeNode*> Lights):** Calcula la suma de distàncies de les parelles de llums que trobem a la llista d'input **Lights**.

8.2.4.6 Struct FLightData

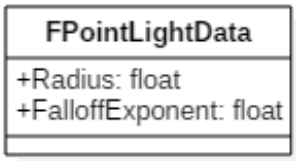


Tupla que encapsula totes les dades pròpies d'una llum genèrica. Això vol dir que guardarà dades com la posició, intensitat, color...

Els **atributs** més rellevants pel nostre treball d'aquesta tupla són:

- **FGuid Guid:** Identificador global de la llum. L'utilitzarem per a diferenciar les llums que es creïn durant el procés de Lightcuts.
- **FVector4 Position:** Posició de la llum dins el món.
- **FVector4 Direction:** Si és llum direccional o focal indica la direcció.
- **FColor Color:** Color en format RGB de la llum.
- **float Brightness:** Intensitat de la llum en unitats de l'UE4.

8.2.4.7 Struct FPointLightData

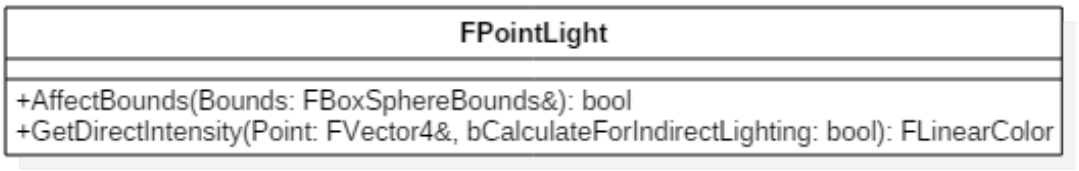


Tupla que afegeix un parell d'atributs per a donar les característiques necessàries per tal de guardar les dades d'una llum puntual.

Els **atributs** que afegeix són:

- **float Radius:** Radi d'influència de la llum.
- **float FalloffExponent:** Valor que indica a quina velocitat decreix la intensitat de la llum dins el radi d'influència corresponent.

8.2.4.8 FPointLight



Classe que implementa les funcionalitats d'una llum puntual utilitzant les dues tuples anteriors. Disposa d'un conjunt de mètodes que permeten fer els càlculs d'il·luminació estàtica.

No té cap **atribut** ni **funció** propi que sigui rellevant. Tot i així, descriurem dos mètodes que són importants pel càlcul d'il·luminació:

- **bool AffectBounds(FBoxSphereBounds& Bounds):** Comprova si la llum afecta al volum delimitador que s'ha donat.
- **FLinearColor GetDirectIntensity(FVector4& Point, bool bCalculateForIndirectLighting):** Fa el càlcul d'intensitat de la il·luminació directa d'aquesta llum respecte d'un punt en específic.

Capítol 9

Implementació i proves

En aquest capítol s'explicaran els passos que s'han seguit per a desenvolupar l'aplicació, des de la modificació del mòdul de Lightmass fins a la modificació del Renderer de l'UE4. La implementació quedarà dividida en 3 apartat ben diferenciats. Primerament tindrem el pas inicial a on es fa la crida al mòdul de Lightmass per a calcular la il·luminació estàtica. Aquest pas serà el més curt a causa del volum de feina, però tot i això és necessari per saber exactament en quin punt es fa la crida dins l'estrucutra el motor. Seguidament hi haurà una explicació del mòdul Lightmass i com s'ha fet la creació de l'arbre de Lightcuts. Finalment trobarem una explicació detallada sobre el Renderer i, especialment, com es fa el render de les llums d'una escena, en la qual explicarem com s'ha modificat per tal d'afegir el render segons la tècnica Lightcuts.

Durant les explicacions hi haurà mostres de proves que s'han fet durant el desenvolupament per tal de comprovar si les modificacions funcionaven correctament. Així doncs, comencem veient com s'ha dut a terme la implementació del procés previ a iniciar el procés de Lightmass.

9.1 Iniciar el mòdul Lightmass

Com ja s'ha comentat anteriorment, el càlcul de l'arbre de Lightcuts s'ha introduït dins el sistema de càlcul d'il·luminació estàtica anomenat Lightmass. Per això mateix, cal definir en quin punt es fa la crida a aquest sistema i així entendre millor com funciona internament el motor. Això indica que en aquesta part no es va dur a terme un volum de feina gran, però la informació obtinguda sobre l'estructura del motor i el seu funcionament van ser claus per la resta del projecte.

Per iniciar Lightmass cal fer-ho des de la barra de tasques que proporciona l'editor de la forma en que ja s'ha detallat a l'apartat 7.1.1. L'editor de l'UE4 funciona a través d'una funció anomenada **Tick(...)**, de la qual només interessa una funció que podem observar al següent tros de codi:

```
void UUnrealEdEngine::Tick(float DeltaSeconds, bool bIdleMode)
{
   // Resta de codi

   // Update lightmass
   UpdateBuildLighting();

   // Resta de codi
}
```

La funció **UpdateBuildLighting()** sempre es crida, sigui o no necessari actualitzar la il·luminació estàtica. Això vol dir que el control sobre si es necessita aquesta actualització es fa

més endavant. El que fa aquesta funció és cridar al Gestor d'il·luminació estàtica i indicar-li que cal fer una actualització, tal com veiem al següent fragment de codi.

```
void UEditorEngine::UpdateBuildLighting()
{
   // Indiquem al gestor cal actualitzar
   FStaticLightingManager::Get()->UpdateBuildLighting();
}
```

La classe **FStaticLightingManager** sempre està activa i només tindrà una instància creada gràcies a ser un singleton. Llavors, un cop dins la funció **UpdateBuildLighting()** d'aquesta classe, observem com només actualitzarà la llum si tenim un sistema d'il·luminació estàtica creat, per la qual cosa, si no s'ha fet la petició, no s'actualitzarà i no es durà a terme la operació. Això també indica que en el moment en que es clica la opció de dissenyar la il·luminació, es crea la instància de la classe **FStaticLightingSystem** per tal que es dugui a terme el procés.

```
void FStaticLightingManager::UpdateBuildLighting()
{
   // Nomes actualitza si existeix el sistema
   if (StaticLightingSystem != NULL)
   {
      StaticLightingSystem->UpdateLightingBuild();
   }
}
```

Aquest mètode fa totes les operacions necessàries per iniciar, actualitzar l'estat i completar el procés de Lightmass. En el nostre cas només ens interessen dos estats, quan s'inicia el procés i quan s'acaba per tal de recuperar les dades obtingudes del càlcul. Així doncs, la primera part en que es crida es fa a partir de la operació **KickoffSwarm()**, la qual simplement indica al LightmassProcessor que s'ha d'iniciar el procés.

```
void FStaticLightingSystem::UpdateLightingBuild()
{
   // Resta de codi
   else if (CurrentBuildStage == FStaticLightingSystem::SwarmKickoff)
   {
      FText Text = LOCTEXT("LightKickoffSwarmMessage", "Kicking off Swarm");
      FStaticLightingManager::Get()->SetNotificationText( Text );
      KickoffSwarm();
   }
   // Resta de codi
}

void FStaticLightingSystem::KickoffSwarm()
{
   bool bSuccessful = LightmassProcessor->BeginRun();

   if (bSuccessful)
   {
      CurrentBuildStage = FStaticLightingSystem::AsynchronousBuilding;
   }
   else
   {
      FStaticLightingManager::Get()->FailLightingBuild(LOCTEXT("SwarmKickoffFailedMessage",
          "Lighting build failed. Swarm failed to kick off."));
   }
}
```

En aquest cas no explicarem la funció **LightmassProcessor->BeginRun()** ja que és molt llarga i no aporta res rellevant a la nostra presentació. La seva funció és controlar en quin

sistema s'està iniciant el procés de Lightmass i llavors dur a terme les comandes necessàries per tal que aquest no falli.

Amb aquestes operacions, el procés de Lightmass ja ha estat cridat i està llest per a fer el càlcul d'il·luminació estàtica, així com calcular l'arbre de Lightcuts que farem servir més tard. El següent pas es veure com duu a terme tot aquest càlcul i de quina forma s'ha afrontat el problema de buscar les parelles òptimes per aconseguir una bona aproximació d'il·luminació.

9.2 Càlcul de l'arbre de Lightcuts

En aquest apartat explicarem com es fa el càlcul de l'arbre de Lightcuts veient com està estructurat el codi del mòdul de Lightmass.

Primer de tot, cal observar el Main del mòdul, ja que des d'aquí es crida la funció que construeix tota la il·luminació estàtica.

```
int LightmassMain(int argc, ANSICHAR* argv[])
{
   // Resta de codi

   // Start the static lighting processing
   UE_LOG(LogLightmass, Display, TEXT("Processing scene GUID: %08X%08X%08X%08X with %d
       threads"), SceneGuid.A, SceneGuid.B, SceneGuid.C, SceneGuid.D, NumThreads );
   BuildStaticLighting(SceneGuid, NumThreads, bDumpTextures);

   // Resta de codi
}
```

Observem que a la funció se li passen nombrosos paràmetres. Els dos primers paràmetres són importants i ajuden a entendre més bé el funcionament del mòdul de Lightmass:

- **SceneGuid:** Identificador global únic de l'escena que s'ha de processar. S'ha fet d'aquesta manera ja que l'Unreal Lightmass es pot iniciar independentment des de fora l'editor, tot indicant-li l'escena a processar.
- **NumThreads:** Nombre de threads que s'utilitzaran per a la construcció de la il·luminació.

Així doncs, un cop entrem a la funció **BuildStaticLighting(...)**, és quan comença la part interessant. Aquest és el mètode mestre de tot el càlcul d'il·luminació, ja que des d'aquí es fa la importació de dades de l'editor i s'inicia el procés que calcula la il·luminació segons el nombre de threads indicat anteriorment. Així mateix, també podem obesrvar les comprovacions que es duen a terme i la informació que es guarda per mostrar més endavant a l'usuari com ha anat el càlcul.

```
void BuildStaticLighting(const FGuid& SceneGuid, int32 NumThreads, bool bDumpTextures)
{
   // [1]
   // We create the importer in order to obtain the data from the editor
   FLightmassImporter Importer( GSwarm );
   FScene Scene;
   if( !Importer.ImportScene( Scene, SceneGuid ) )
   {
      UE_LOG(LogLightmass, Log, TEXT("Failed to import scene file"));
      exit( 1 );
   }
   GStatistics.ImportTimeEnd = FPlatformTime::Seconds();

   // [2]
   // Setup the desired lighting options
   FLightingBuildOptions LightingOptions;

   // [3]
   // We create the exporter to be able to send the results back to the editor
   FLightmassSolverExporter Exporter( GSwarm, Scene, bDumpTextures );

   double LightTimeStart = FPlatformTime::Seconds();

   // [4]
   // Create the global lighting system to kick off the processing
   FStaticLightingSystem LightingSystem(LightingOptions, Scene, Exporter, NumThreads );

   GStatistics.TotalTimeEnd = FPlatformTime::Seconds();

   // [5]
   // Report back statistics over Swarm.
   ReportStatistics();

   double EndTime = FPlatformTime::Seconds();

   UE_LOG(LogLightmass, Log, TEXT("Lighting complete [Startup = %s, Lighting = %s]"), *
       FPlatformTime::PrettyTime(LightTimeStart - SetupTimeStart), *FPlatformTime::
       PrettyTime(EndTime - LightTimeStart));
}
```

Podem observar l'ordre d'operacions i com arribem a calcular finalment la il·luminació:

1. Primer de tot podem observar com es fa la importació de dades de l'editor utilitzant l'identificador global únic de l'escena que s'ha donat anteriorment. Totes les dades que s'importen es guarden a la variable **Scene**.
2. Seguidament es creen les opcions que dictaran com es calcularà la il·luminació. En aquest cas es deixen les opcions per defecte ja que no necessitem modificar res.
3. El següent pas és crear l'exportador que farem servir al final del càlcul per a traslladar les dades un altre cop cap a l'editor.
4. Creem el sistema d'il·luminació estàtica. El constructor d'aquesta classe ja s'encarrega d'iniciar tots els processos necessaris per al càlcul, tal com veurem més endavant.
5. Finalment reportem totes les estadístiques que s'han anat obtinguent i guardant durant el procés per comprovar si hi ha hagut algun error i poder saber fàcilment què ha passat.

Veiem que també hi ha un objecte anomenat GSwarm. Aquest és del tipus FLightmassSwarm (Secció 8.2.4.4) i permetrà enviar les dades cap a l'editor un cop finalitzat el càlcul.

A partir d'ara comença l'explicació de com s'ha aconseguit l'arbre de Lightcuts. El següent

fragment de codi es troba dins el constructor de **FStaticLightingSystem**, que com ja hem comentat, fa tots els passos previs necessaris a iniciar les operacions de càlcul i seguidament inicia el procés de Lightmass que obté les dades d'il·luminació estàtica.

```
FStaticLightingSystem::FStaticLightingSystem(const FLightingBuildOptions& InOptions, FScene&
     InScene, FLightmassSolverExporter& InExporter, int32 InNumThreads)
{
   // Editor's data Import
   // Variable initialization

   // If there's no PointLights in the scene, the Lightcuts creation doesn't starts
   if (InScene.PointLights.Num() > 0) {
      LightcutsOp.LightcutsTreeCreation(InScene.PointLights);
      LightcutsOp.DebugTree(LightcutsOp.LightcutsTree, 0);
      LightcutsOp.TreeToArray(LightcutsOp.LightcutsTree);
   }

   // Spread out the work over multiple parallel threads
   MultithreadProcess();
}
```

Observem que si no hi ha llums puntuals a l'escena, no s'inicia la creació de l'arbre de Lightcuts. Això és a causa que el sistema ha estat creat només utilitzant llums puntuals, ja que va ser la decisió a l'hora de buscar una aproximació inicial a la tècnica.

La variable **LightcutsOp** és del tipus FLightcutsOperations (Secció 8.2.4.5). Veiem que es criden tres mètodes, dels quals el més important i que explicarem amb detall és l'anomenat **LightcutsTreeCreation(...)**. Aquest mètode és el que fa tot el càlcul i creació de l'arbre de Lightcuts, tal com indica el nom. Les altres dues operacions no són rellevants a causa que la última simplement fa la conversió de l'arbre en forma d'arbre binari cap a array, ja que va més bé per a fer l'exportació cap a l'editor, i l'altre serveix per a debuggar l'arbre obtingut.

Tot i això, cal centrar-se un moment en el debug de l'arbre, ja que aquí va sorgir un dels problemes del projecte. Al ser el mòdul de Lightmass un procés iniciat per l'Unreal Editor, vam perdre la possibilitat de debuggar línia a línia el codi, ja que no és el programa principal. Això va portar problemes a l'hora de crear l'arbre, ja que no hi havia una manera d'anar comprovant pas a pas les operacions. Degut a això, es va optar per buscar una forma de debug en forma de fitxers de text. Tot això va provocar que el procés de crear l'arbre fos més lent, a causa que les comprovacions no eren directes i es van haver d'anar fent varies proves per tal d'assegurar-nos que l'arbre es creava correctament.

A la Figura 9.1 podem observar quina estructura tenen aquests fitxers de debug, així com comprovar com queda un arbre de Lightcuts creat a partir de 5 llums puntuals. Veiem que inicialment tenim l'arrel de l'arbre i llavors van apareixent els fills als nivells inferiors. A més, si mirem els valors del radi, color i intensitat, es veu com els pares van agafant la suma dels valors dels fills. Així doncs, també es mostra més clarament el funcionament de la tècnica, que es suma a l'explicació de l'apartat

Tornant a l'explicació de com s'ha creat l'arbre, tenim el mètode **LightcutsTreeCreation(...)**. Aquesta funció, construeix l'arbre binari de Lightcuts que més tard s'utilitzarà per a fer les aproximacions d'il·luminació en temps real a la hora de fer el render de l'escena. 5.2.2.

```
GUID E0050D054264414F31A31CA471CA3BA9 - Position X=-300.000 Y=0.000 Z=150.000 W=1.000 - Radius 700 - Color (R=255,G=255,B=255,A=255) - Brightness = 27500
GUID 5111A67B4FB2E3600975FFB5BC454DEE - Position X=-300.000 Y=0.000 Z=150.000 W=1.000 - Radius 400 - Color (R=255,G=255,B=255,A=255) - Brightness = 17500
GUID F32541E24693C25D57B2BCABC650353E - Position X=-300.000 Y=0.000 Z=150.000 W=1.000 - Radius 337.464 - Color (R=255,G=255,B=255,A=0) - Brightness = 12500
GUID C2C14DFC4DDFAFF43D562F8AE934C28C - Position X=-300.000 Y=0.000 Z=150.000 W=1.000 - Radius 250 - Color (R=255,G=76,B=216,A=0) - Brightness = 5000
GUID E356FBF44AFB193168CEDA98FB01019F - Position X=-225.000 Y=-45.000 Z=150.000 W=1.000 - Radius 250 - Color (R=44,G=255,B=88,A=0) - Brightness = 7500
GUID 97D29E3844BB950A7C882BAA8269BD84 - Position X=-300.000 Y=150.000 Z=150.000 W=1.000 - Radius 250 - Color (R=125,G=125,B=20,A=255) - Brightness = 5000
GUID DDC5E39742F301EF8D05A3A9CF46F053 - Position X=0.000 Y=0.000 Z=150.000 W=1.000 - Radius 400 - Color (R=255,G=230,B=255,A=0) - Brightness = 10000
GUID EAC0D8644993E58FEA6258BABC2A6F1C - Position X=0.000 Y=0.000 Z=150.000 W=1.000 - Radius 250 - Color (R=85,G=59,B=255,A=0) - Brightness = 5000
GUID 06BBF64D490C83FDA8775B910C4E7E39 - Position X=150.000 Y=0.000 Z=150.000 W=1.000 - Radius 250 - Color (R=255,G=171,B=123,A=0) - Brightness = 5000
```

Figura 9.1: Exemple de fitxer de debug

Al següent fragment de codi podem veure com s'ha gestionat i organitzat la construcció de l'arbre:

```
void FLightcutsOperations::LightcutsTreeCreation(TArray<FPointLight>& PointLights)
{
   // Transform the input PointLights into TreeNode. This will help in order to create the
       tree structure
   TArray<TreeNode*> TreeLevelAux;
   for (FPointLight l : PointLights)
   {
      TreeNode *t = new TreeNode;
      t->item = l;
      t->left = NULL;
      t->right = NULL;
      TreeLevelAux.Add(t);
   }

   // The tree creation will be completed once we only have one node into the array
   while (TreeLevelAux.Num() > 1)
   {
      // Search for the TreeNode pairs that achieve the minimal distance sum
      MinDistancePairs(TreeLevelAux, 0, MAX_FLT);
      // Creates a new level of the tree using the previously obtained pairs
      TreeLevelAux = ComputeTreeLevel(TreeLevelAux);
   }

   // We save the obtained tree in order not to lose the work achieved in this method
   LightcutsTree = new TreeNode;
   LightcutsTree = TreeLevelAux[0];
}
```

Inicialment, cal fer la transformació de les llums puntuals, que tenim d'entrada a la funció, a objectes de tipus TreeNode. Això es va dissenyar d'aquesta manera ja que, com veurem properament, a l'hora de fer la construcció de l'arbre, permet modificar l'array de tal forma que aquest va variant dinàmicament i l'arbre va agafant forma a mesura que es calculen els diferents nivells. Tots els nodes inicials només tenen el valor de l'ìtem i els dos fills apunten a NULL. Així doncs, un cop completada la transformació, tenim una array de TreeNode la qual farem servir per a construir l'arbre.

Això si, cal controlar que el nombre d'items que hi ha a l'array sigui major a 1. Això permet controlar dues coses a la vegada:

1. Si l'escena no té llums puntuals, no es pot calcular cap arbre ja que només funciona amb llums puntuals.
2. A mesura que es vagin calculant els diferents nivells de l'arbre, l'array TreeLevelAux anirà disminuïnt el nombre d'items que guarda. En el moment en que TreeLevelAux.Num() sigui 1, sabrem que s'ha calculat tot l'arbre i que per tant a la posició zero hi ha quedat l'arrel.

Seguint amb el codi trobem la funció **MinDistancePairs(TreeLevelAux, 0, MAX_FLT)**. Aquesta funció és l'encarregada de calcular la combinació de parelles que aconsegueix la suma mínima de distàncies entre parelles de llums. Això vol dir que estarem agrupant les llums més properes entre elles i això assegura que els errors d'aproximació un cop es faci el render seran mínims.

Si ens fixem en l'implementació, veiem que té la següent estructura:

```
float FLightcutsOperations::MinDistancePairs(TArray<TreeNode*>& LevelLights, int32 curr,
    float currSum)
{
   // Copy the array so we don't modify the original values
   TArray<TreeNode*> aux = LevelLights;
   float newDist = 0.0f;

   if (curr >= LevelLights.Num() - 1)
   {
      return TotalDistance(aux);
   }
   else
   {
      // We swap the lights to achieve all the possible permutations
      for (int32 i = curr; i < aux.Num(); i++)
      {
         aux.Swap(curr, i);

         // We pass the current level + 1, not to get stuck with infinite permutations
         newDist = MinDistancePairs(aux, curr + 1, currSum);

         // If the new combination of pairs achieves a lower distance sum, we update the
             LevelLights array with the new combination.
         if (newDist < currSum)
         {
            LevelLights = aux;
            currSum = newDist;
         }

         aux.Swap(curr, i);
      }
   }

   return currSum;
}
```

El funcionament del mètode és ben simple: provar totes les combinacions possibles de parelles per tal d'obtenir la millor combinació possible i evitar els errors d'aproximació. Per aconseguir-ho el que vam decidir va ser fer un sistema que provés totes les permutacions possibles.

Així doncs, un cop hem fet els intercanvis suficients per arribar a la última llum, només caldrà calcular la distància total que sumen les seves parelles. Aquesta suma es retornarà i es comprovarà si és menor que l'anterior obtinguda. Aquí entra en joc el fet que, inicialment, al mètode se li passa **MAX_FLT** com a valor de suma actual. D'aquesta forma la primera suma que obtinguem sempre serà menor que aquest valor i ja haurem obtingut una combinació de parelles vàlida.

Si obtenim un nou valor menor que l'anterior, també s'haurà d'actualitzar la llista de llums amb la nova combinació. Això és fàcil a causa que tenim una referència de la llista, per la qual cosa qualsevol canvi que hi hem serà visible fora el mètode. D'aquesta manera, gràcies a la llista auxiliar que anem permutant, podem fer el canvi ràpid i de forma senzilla.

S'observa fàcilment com el mètode no és difícil, simplement és costós en temps. Això no és un problema ja que és un pre-procés i per tant no importa el temps que tardi a calcular les possibles permutacions.

El mètode **TotalDistance(...)** calcula la distància total que sumen les diferents parelles de la llista. La seva única particularitat és que cal controlar si la llista té un nombre parells d'ítems o senar. Si és senar, voldrà dir que l'últim ítem no caldrà tenir-lo en compte ja que no fa parell amb cap altre llum. Així doncs, al següent fragment de codi trobem la implementació del mètode aquí descrit.

```
float FLightcutsOperations::TotalDistance(const TArray<TreeNode*> Lights)
{
   int32 index = 0;
   float sum = 0.0f;

   // We go to Num()-1 adding 2 at a time so if the number of items is odd, the last item
       won't be counted
   while (index < Lights.Num() - 1)
   {
      FPointLight a = Lights[index]->item;
      FVector av = FVector(a.Position.X, a.Position.Y, a.Position.Z);
      FPointLight b = Lights[index + 1]->item;
      FVector bv = FVector(b.Position.X, b.Position.Y, b.Position.Z);

      // Thanks to the class FVector, the distance calculation between two vectors is done
          with a simple call
      sum += FVector::Dist(av, bv);

      index += 2;
   }

   return sum;
}
```

Explicades aquestes funcions, toca anar enrera i veure com es calcula el nou nivell de l'arbre, que després tornarà a passar pel càlcul de parelles òptimes. Això vol dir que toca veure com funciona el mètode **ComputeTreeLevel(TreeLevelAux)**.

La descripció bàsica del mètode és que, a partir del nivell existent (entenem com a nivell l'array de TreeNode que té d'input) crea un nou nivell i el retorna. Tot i això, és més complex per la forma en que es tracten les dades i com s'aconsegueix anar creant l'arbre. Al fragment de codi següent podem veure el funcionament i com hem fet per anar creant l'arbre de Lightcuts des de les fulles fins a l'arrel, obtenint un arbre binari i balancejat.

```
TArray<FLightcutsOperations::TreeNode*> FLightcutsOperations::ComputeTreeLevel(TArray<
    TreeNode*> LevelLights)
{
   bool odd = LevelLights.Num() % 2 != 0;

   TArray<TreeNode*> newLevel;

   int32 i = 0;
   while (i < LevelLights.Num() - 1)
   {
      FPointLight newLight;

      // We assign a new GUID to the light. We chose to give the new light the position of
          the first light in the pair. The new intensity and color will be the sum of the
          intensity and color of the two lights respectively.
      newLight.Guid = newLight.Guid.NewGuid();
      newLight.Position = LevelLights[i]->item.Position;
      newLight.Brightness = LevelLights[i]->item.Brightness + LevelLights[i + 1]->item.
          Brightness;
      newLight.Color = LevelLights[i]->item.Color;
      newLight.Color += LevelLights[i + 1]->item.Color;

      // Calculation of the new radius. It must
      float DistLights = FVector::Dist(LevelLights[i]->item.Position, LevelLights[i + 1]->
          item.Position);

      float MaxDistFromOrigen = DistLights + LevelLights[i + 1]->item.Radius;
      if (LevelLights[i]->item.Radius <= MaxDistFromOrigen)
      {
         newLight.Radius = MaxDistFromOrigen;
      }
      else
      {
         newLight.Radius = LevelLights[i]->item.Radius;
      }

      TreeNode *newNode = new TreeNode;
      newNode->item = newLight;
      newNode->left = LevelLights[i];
      newNode->right = LevelLights[i + 1];

      newLevel.Add(newNode);

      i += 2;
   }
   if (odd)
   {
       // If odd, we add the last light of the list to the new level as it won't be paired
           with any light
       newLevel.Add(LevelLights[LevelLights.Num() - 1]);
   }

   return newLevel;
}
```

Els passos que es segueixen per aconseguir crear un nou nivell de l'arbre són els següents:

1. **newLevel** guardarà el nou nivell que es crearà a partir del nivell actual el qual està guardat a **LevelLights**.
2. Mentre l'índex sigui menor que LevelLights.Num()-1 fem:
   2.1. Creem una nova llum que serà el para de les llums que es troben a la posició de l'índex i la posició de l'índex+1.
   2.2. Assignem un nou identificador global únic a la llum.
   2.3. Com a posició agafa la posició de la llum que trobem a LevelLights[índex].
   2.4. La intensitat i el color de la nova llum serà la suma de la intensitat i el color dels dos fills respectivament.
   2.5. El radi vindrà determinat per unes certes condicions. Si la llum de la posició índex engloba a la segona, el nou radi serà el de la primera llum. Altrament, el nou radi serà la distància entre les llums més el radi de la segona llum.
   2.6. Crear un nou TreeNode que com a ítem té la nova llum creada i com a fills les dues llums utilitzades. Finalment afegir-lo al nou nivell.
3. Si la llista és senar, afegim la última llum al nou nivell sense modificar-la
4. Retornar el nou nivell que hem creat amb el bucle anterior.

Gràcies a aquest mètode i l'estructura TreeNode, podem construir l'arbre de Lightcuts de tal manera que obtenim un arbre binari i balancejat. Això ens va perfecte ja que a l'hora de transformar aquesta estructura en forma d'array, serà un procés molt simple. Tot això facilitarà la feina d'exportació i el tractament de les dades en temps real durant el render.

9.3 Exportació dels resultats cap a l'editor

Aquest apartat tractarà sobre com es fa el traspàs de les dades resultants cap a l'editor. Això implica veure com es creen els diferents canals per exportar les dades i com passem l'arbre a una estructura d'array per a treballar de millor forma.

Per a veure més clarament com funciona, hem de tornar al constructor de la classe FStaticLightingSystem, exactament al mètode **MultithreadProcess()**, tal com podem recordar amb fragment de codi següent:

```
FStaticLightingSystem::FStaticLightingSystem(const FLightingBuildOptions& InOptions, FScene&
     InScene, FLightmassSolverExporter& InExporter, int32 InNumThreads)
{
   // Editor's data Import
   // Variable initialization

   // If there's no PointLights in the scene, the Lightcuts creation doesn't starts
   if (InScene.PointLights.Num() > 0) {
      // Lightcuts Tree creation
   }

   // Spread out the work over multiple parallel threads
   MultithreadProcess();
}
```

Aquesta funció s'encarrega de fer tot el càlcul de la il·luminació estàtica, repartint la feina entre els diferents threads que se li han indicat anteriorment. Tot i això, també s'encarrega de fer les exportacions de les dades un cop aquestes han estat calculades. Aquestes operacions es duen a terme un cop s'han acabat els càlculs, per la qual cosa hi ha comprovacions cada cert temps per saber quines operacions han acabat, i per tant es pot començar a fer l'exportació de resultats per tal de no perdre temps. Tot i això, si es donés el cas que totes les operacions s'acaben al mateix temps, es torna a fer una comprovació al final per assegurar-se que s'exporten tots els resultats correctament.

En el nostre cas, això no importa, ja que tots els càlculs s'han fet abans que s'iniciï aquest procés, per la qual cosa ja podem fer l'exportació directament. Al següent bloc de codi podem veure un fragment del mètode **MultithreadProcess()** en el qual s'observa la funció clau que ens permet fer l'exportació i que explicarem a continuació.

```
void FStaticLightingSystem::MultithreadProcess()
{
   // Create photon maps for further operations
   // Determine the tasks to do
   // Spawn the static lighting threads
   // Wait for all the threads to end while trying to export the results of finished tasks

   // Exports tasks that are not mappings
   ExportNonMappingTasks();
}
```

S'han indicat les tasques més importants en forma de comentaris ja que no són rellevants pel projecte. D'altra banda el mètode **ExportNonMappingTasks()** és important a causa que permet fer l'exportació dels resultats. Al següent fragment de codi podem veure com funciona el mètode en qüestió.

```
void FStaticLightingSystem::ExportNonMappingTasks()
{
   // Export the Lightcuts tree to the editor
   Exporter.ExportLightcutsTree(LightcutsOp.TreeArray);

   // Export the results if task completed
   if (bVolumeLightingSamplesComplete)
   {
      // Export Volume Lighting Samples if task completed
   }
   if (bShouldExportVolumeDistanceField)
   {
      // Export Volume Distance Field if task completed
   }
   // Rest of operations
}
```

El que més importa d'aquesta funció, és veure que aquí es fa l'exportació dels resultats utilitzant l'exporter anteriorment inicialitzat al constructor. D'altra banda, també podem observar el fet que, mentre funcionen els threads, es van fent exportacions igualment. Per la qual cosa, es va comprovant per cada operació si s'ha acabat la tasca. Si es dóna el cas, s'exporten els resultats i s'aconsegueix un millor aprofitament del temps.

Si observem el mètode per exportar, podem observar l'estructura al fragment de codi següent:

```
void FLightmassSolverExporter::ExportLightcutsTree(const TArray<FPointLight>& LightcutsTree)
{
   // Create Channel name using the correct FGUIDs and open it
   const FString ChannelName = CreateChannelName(LightcutsDataGuid, LM_LIGHTCUTS_VERSION,
       LM_LIGHTCUTS_EXTENSION);
   const int32 ErrorCode = Swarm->OpenChannel(*ChannelName, LM_LIGHTCUTS_CHANNEL_FLAGS, true
       );

   // If there's an error don't export anything
   if (ErrorCode >= 0)
   {
      // Write the number of items to pass so the editor knows when to stop reading
      int32 NumPointLights = LightcutsTree.Num();
      Swarm->Write(&NumPointLights, sizeof(NumPointLights));

      // Export the Lightcuts tree
      for (int32 i = 0; i < NumPointLights; i++)
      {
         const FPointLight& Light = LightcutsTree[i];
         FLightData LightData;
         FPointLightData PointData;

         // LightData stores the generic light information
         LightData.Brightness = Light.Brightness;
         LightData.Position = Light.Position;
         LightData.Direction = Light.Direction;
         LightData.Guid = Light.Guid;
         LightData.IndirectLightingSaturation = Light.IndirectLightingSaturation;
         LightData.ShadowExponent = Light.ShadowExponent;
         LightData.LightSourceRadius = Light.LightSourceRadius;
         LightData.LightSourceLength = Light.LightSourceLength;
         LightData.Color = Light.Color;

         // PointData stores the specific point light information
         PointData.Radius = Light.Radius;
         PointData.FalloffExponent = Light.FalloffExponent;

         Swarm->Write(&LightData, sizeof(LightData));
         Swarm->Write(&PointData, sizeof(PointData));
      }

      // Close the exporting channel
      Swarm->CloseCurrentChannel();

      UE_LOG(ModifiedLightmass_TFG, Log, TEXT("Write Channel LightcutsTree - Finished"));
   }
   else
   {
      UE_LOG(ModifiedLightmass_TFG, Log, TEXT("Failed to open dominant lightcuts channel!"))
          ;
   }
}
```

Tal com es pot apreciar, dins d'aquest mètode hi ha moltes parts a comentar. La primera part és la creació del canal que permet passar les dades del mòdul Lightmass cap a l'editor. En aquest cas, per crear un canal es necessita donar-li un identificador FGUID, una versió de les dades que es passen i una extensió que s'incorporarà al canal per a saber les dades a exportar.

En aquest cas, tots aquests identificadors globals s'han hagut d'introduïr al codi tenint en

compte que han de ser diferent dels ja existents per als altres tipus de dades. Així doncs, es va decidir posar uns identificadors globals que indiquin exactament sobre què tracten. Aquests identificadors es van triar segons una sèrie de paraules que identifiquen la funció que fa cada FGUID. Això es va dur a terme agafant aquestes paraules i passar-les a hexadecimal, tenint en compte que necessitem exactament 16 caràcters.

Els identificadors escollits, juntament amb l'explicació, són els següents:

- **Channel Guid - LightcutsDataGuid:** FGuid(0x4c696768, 0x74637574, 0x73446174, 0x61475549). Traducció: LightcutsDataGUI.
- **Version: LM_LIGHTCUTS_VERSION:** FGuid(0x4c696768, 0x74637574, 0x73566572, 0x73696f6e). Traducció: LightcutsVersion.
- **Extension: LM_LIGHTCUTS_EXTENSION:** En aquest cas és un String de valor "lcut".
- **Channel Flags: LM_LIGHTCUTS_CHANNEL_FLAGS:** A la hora d'obrir el canal només li donem permisos per escriure.

Podem veure com també s'ha afegit el valor de Channel Flags a la llista anterior. En aquest cas només interessa crear un canal solament d'escriptura, ja que no s'ha de modificar ni llegir res. Això ens porta a la operació d'obertura del canal, en la qual s'obre el canal mencionat i es comprova el codi d'error retornat. Si aquest codi és superior a zero, és que s'ha obert correctament el canal i podem passar a exportar totes les dades.

A la hora d'exportar, el primer pas és indicar el nombre d'ítems que es passaran, ja que l'editor necessita saber-ho per tal de llegir el nombre correcte de llums del canal. Un cop indicat aquest nombre, passem a exportar cada llum individualment.

Per a fer l'exportació correctament, necessitem uns tipus de dades que existeixi en els dos sistemes. En aquest cas utilitzarem les tuples FLightData (Secció 8.2.4.6) i FPointLightData (Secció 8.2.4.7). La primera serveix per a passar totes aquelles dades genèriques que comparteixen tots els tipus de llums, mentre que la segona ens permet passar els atributs específics que tenen les llums puntuals. Un cop s'han passat les dades de els llums a les dues tuples, aquestes s'escriuen a sobre el canal utilitzant sempre el mateix ordre, que serà utilitzat igualment per l'editor.

Finalment, un cop s'han passat totes les llums es tanca el canal, la qual cosa deixa pas a la resta d'exportacions que s'han de dur a terme. Així doncs, tota la feina que havia de fer el mòdul de Lightmass s'ha dut a terme i la resta que falta és feina de l'editor i del renderer de l'UE4.

9.4 Importació dels resultats cap al renderer

Un cop ja hem fet els càlculs i s'han exportat totes les dades necessàries, l'única operació restant és fer la importació des de l'editor. Això s'aconsegueix amb el mètode **FLightmassProcessor::CompleteRun()** el qual es crida un cop s'han finalitzat les tasques del mòdul Lightmass. Tal com podem observar al següent fragment de codi, les tasques que s'hi duen a terme són simples, importar les dades resultants del càlcul d'il·luminació.

```
bool FLightmassProcessor::CompleteRun()
{
   bRunningLightmass = false;

   double ImportStartTime = FPlatformTime::Seconds();
   double OriginalApplyTime = Statistics.ApplyTimeInProcessing;

   if ( !bProcessingFailed && !GEditor->GetMapBuildCancelled() )
   {
      ImportVolumeSamples();
      ImportPrecomputedVisibility();
      ImportMeshAreaLightData();
      ImportVolumeDistanceFieldData();
      ImportLightcutsTreeData();

      // Extra code
   }

   // Extra code
}
```

Però la part que més ens interessa és quan fem la importació de l'arbre de Lightcuts, la qual es fa dins el mètode **ImportLightcutsTreeData()**. El funcionament del mètode és semblant a com s'han exportat les dades des del mòdul Lightmass.

El primer pas és crear el nom del canal i obrir-lo. Per a dur a terme aquesta operació cal definir els identificadors tal com s'ha fet en el moment de l'exportació. En aquest cas, tots aquests identificadors tenen el mateix FGuid que al mòdul Lightmass ja que, tal com s'ha comentat anteriorment, han de ser iguals per tal de poder accedir al canal. Tot i això, varien els flags d'obertura ja que en aquest cas només ens interessa llegir les dades del canal.

Si la obertura del canal es fa correctament sense errors, passarem a fer la importació de les dades. Tal com ja s'ha fet amb l'exportació, primer de tot cal llegir el nombre d'ítems que formen part de l'arbre.

Un cop tenim el nombre d'ítems, ja només cal fer les iteracions necessàries per a llegir tot l'arbre de Lightcuts. Tenint en compte que l'arbre s'ha exportat ordenadament, podem anar agafant els valors i simplement afegir-los a l'array de forma que ja queda l'arbre ordenat. A més, tenint en compte que tenim les dues tuples amb les quals s'han exportat les dades, a la hora d'afegir les llums és tan simple com agafar els valors de les dues tuples i crear un ítem del tipus **FLightcutsLight** que els guardi i que serà introduït a l'arbre de Lightcuts de l'escena. Aquest últim pas és el que fa la funció **AddLightcutsLight(...)** de la classe FScene.

Tot i això, cal tenir en compte que podem tenir un arbre de Lightcuts anterior. Per aquest motiu, abans de llegir cap dada del canal, netejem la llista que trobem a l'escena i la buidem. D'aquesta manera, assegurem que sempre que es calcula un nou arbre de Lightcuts s'actualitza el valor de l'escena i no apareixen errors inesperats.

Finalment, quan ja s'hagin llegit totes les dades, tanquem el canal de transmissió i ens dirigim al renderer per a fer les modificacions necessàries per tal de renderitzar les llums puntuals segons l'arbre de Lightcuts en comptes de la forma original en que aquestes són tracten a l'UE4.

```
void FLightmassProcessor::ImportLightcutsTreeData()
{
   // Create the channel name and open it
   FString ChannelName = Lightmass::CreateChannelName(Lightmass::LightcutsDataGuid,
       Lightmass::LM_LIGHTCUTS_VERSION, Lightmass::LM_LIGHTCUTS_EXTENSION);
   int32 Channel = Swarm.OpenChannel(*ChannelName, LM_LIGHTCUTS_CHANNEL_FLAGS);

   UE_LOG(LogStaticLightingSystem, Warning, TEXT("Channel Name: %s - Open Channel Result: %d
       "), *ChannelName, Channel);

   if ( Channel >= 0 )
   {
      // Read the number of items that make up the tree
      int32 NumLights = 0;
      Swarm.ReadChannel(Channel, &NumLights, sizeof(NumLights));

      // Resets the Lightcuts Array from the scene
      System.GetWorld()->Scene->UpdateLightcutsValues(0);

      for (int32 i = 0; i < NumLights; i++)
      {
         // Read the data from the channel
         Lightmass::FLightData LightData;
         Lightmass::FPointLightData PointData;
         Swarm.ReadChannel(Channel, &LightData, sizeof(LightData));
         Swarm.ReadChannel(Channel, &PointData, sizeof(PointData));

         // Add the light to the Lightcuts array in the scene
         System.GetWorld()->Scene->AddLightcutsLight(LightData.Guid, LightData.Position,
             LightData.Brightness, LightData.Color, PointData.Radius, PointData.
             FalloffExponent);
      }

      Swarm.CloseChannel(Channel);
   }
   else
   {
      // Error checking
   }
}
```

9.5 Modificació del renderer

En aquest apartat descriurem i explicarem com s'ha modificat el renderer de l'UE4 per tal de fer possible el càlcul d'il·luminació directa utilitzant l'arbre de Lightcuts. Per a dur a terme aquesta operació cal modificar com es renderitzen les llums de l'escena. Per a fer això cal mirar el funcionament del renderer. Una vista ràpida d'aquest ja ens mostra clarament a on es fa aquesta operació, tal com podem veure al fragment de codi de la funció **Render()** de l'UE4. Al següent fragment de codi podem apreciar la part rellevant d'aquesta funció en la qual es fa la crida necessària per a renderitzar les llums. Podem observar també com abans i després de fer el render de la il·luminació, s'avisa sobre què s'està renderitzant en aquell precís instant. Al següent fragment de codi podem veure el que hem comentat:

```
void FDeferredShadingSceneRenderer::Render(FRHICommandListImmediate& RHICmdList)
{
   // Extra code

   RHICmdList.SetCurrentStat(GET_STATID(STAT_CLM_Lighting));
   RenderLights(RHICmdList);
   RHICmdList.SetCurrentStat(GET_STATID(STAT_CLM_AfterLighting));

   // Extra code
}
```

Dins el renderitzat de les llums de l'escena, trobem que hi ha moltes operacions que es duen a terme. Tot i això, a nosaltres només ens interessa la part en que es renderitzen les llums que afecten a la il·luminació indirecta i, en especial, a les que no provoquen ombres dins l'escena. El fet de no provocar ombres ve perquè el motor té un límit de shadow maps que es sobreposin, per la qual cosa si tenim moltes llums juntes, obtindriem aquest problema. Per aquest motiu es va optar per utilitzar llums que no provoquin ombres. Però això permet tenir l'arbre de Lightcuts, el qual les llums no provocaran ombres, i les llums normals de l'UE4, les quals si estan configurades per tal que provoquin ombres, seran renderitzades amb l'algoritme original del motor.

Al següent fragment de codi podem veure com s'ha afrontat la implementació del render segons l'arbre de Lightcuts.

```
void FDeferredShadingSceneRenderer::RenderLights(FRHICommandListImmediate& RHICmdList)
{
   // Extra code
   // Render only if direct lighting is enabled
   if(ViewFamily.EngineShowFlags.DirectLighting)
   {
      // Extra code

      // Draw non-shadowed, non-light function, lights without changing render targets
          between them
      // No rendering will be done in this case. All the non-shadowed lights will be
          rendered using the Lightcuts Tree

      // Render only if a Lightcuts Tree exists
      if (LightcutsArray.Num() > 0)
      {
         // Create a new light that has the position and radius of the Lightcuts Tree root
         FSortedLightSceneInfo& SortedLightInfo = FSortedLightSceneInfo(SortedLights[0]);
         FLightSceneInfoCompact& LightSceneInfoCompact = SortedLightInfo.SceneInfo;
         FLightSceneInfo* LightcutsSceneInfo = LightSceneInfoCompact.LightSceneInfo;

         LightcutsSceneInfo->Proxy->SetPosition(FVector4(LightcutsArray[0].GetPosition(), 1.0
             f));

         LightcutsSceneInfo->Proxy->SetRadius(LightcutsArray[0].GetRadius());

         // Render the lighting using the Lightcuts Tree
         RenderLightLightcuts(RHICmdList, LightcutsSceneInfo, false, false);
      }
   }
   // Extra code
}
```

Observem que el primer pas és comprovar si s'està renderitzant amb il·luminació indirecta o no, ja que es podria triar la opció de només utilitzar il·luminació estàtica prèviament calculada en forma de Lightmaps. En aquest cas però, si que ens interessa ja que, tot i haver estat calculat l'arbre en un pre-procés, es tracten els llums de l'arbre com a llums d'il·luminació directa.

Acte seguit tindríem la part en que es renderitzen les llums que no provoquen ombres. Tal com s'ha comentat però, nosaltres no en farem el render, ja que aquesta configuració de llum serà tractada amb l'arbre de Lightcuts. Per això fem la comprovació de si existeix un arbre de Lightcuts (encara que sigui només d'una llum).

Pel següent pas s'ha utilitzat l'estructura de rendering que utilitza l'UE4 originalment i s'-ha adaptat per tal d'aconseguir uns resultats molt semblants que quan es renderitza les llums amb l'algoritme original. Per aconseguir això farem una còpia d'una de les llums puntuals existents i modificarem els valors de radi i posició. D'aquesta manera, aconseguim indicar-li al renderer quina és la zona que s'ha de renderitzar, que en aquest cas serà l'esfera que té com a posició la posició de l'arrel de l'arbre i com a radi d'acció el radi de la mateixa arrel. Aquesta informació de l'escena i de la llum, que guardem a **LightcutsSceneInfo** serà passada al mètode **RenderLightLightcuts(...)** el qual s'encarregarà de passar al shader les dades necessàries.

Al següent fragment de codi podem observar l'estructura d'aquesta última funció mencionada.

```
void FDeferredShadingSceneRenderer::RenderLightLightcuts(FRHICommandList& RHICmdList, const
    FLightSceneInfo* LightSceneInfo, bool bRenderOverlap, bool bIssueDrawEvent)
{
   // We set the new light bounds (The bounds of the root in the lightcuts array)
   FSphere RootLightcutsBounds = FSphere(LightcutsArray[0].GetPosition(), LightcutsArray[0].
       GetRadius());

   // Do a render for every view we have (ex: split screen, editor views...)
   for (int32 ViewIndex = 0; ViewIndex < Views.Num(); ViewIndex++)
   {
      FViewInfo& View = Views[ViewIndex];

      // Set the device viewport for the view.
      RHICmdList.SetViewport(View.ViewRect.Min.X, View.ViewRect.Min.Y, 0.0f, View.ViewRect.
          Max.X, View.ViewRect.Max.Y, 1.0f);

      TShaderMapRef<TDeferredLightVS<true> > VertexShader(View.ShaderMap);
      SetBoundingGeometryRasterizerAndDepthState(RHICmdList, View, RootLightcutsBounds);

      // Set parameters for the vertex shader and pixel shader
      SetShaderTemplLightingLightcuts<false, true, true>(RHICmdList, View, *VertexShader,
          LightSceneInfo);
      VertexShader->SetParameters(RHICmdList, View, LightSceneInfo);

      StencilingGeometry::DrawSphere(RHICmdList);
   }

   if (bStencilDirty)
   {
      // Clear the stencil buffer to 0.
      RHICmdList.Clear(false, FColor(0, 0, 0), false, 0, true, 0, FIntRect());
   }
}
```

Inicialment podem observar com es crea l'esfera delimitadora que indica quina zona s'haurà de renderitzar. En el nostre cas, com ja hem comentat, com a màxim haurem d'actuar sobre la zona d'acció de la llum situada a l'arrel de l'arbre. Acte seguit s'inicia el render segons el nombre de vistes que tinguem a la pantalla. En aquest cas podem tenir moltes vistes, com podríem trobar a l'editor en que s'observa l'escena des de diferents punts, o podem tenir un joc a pantalla dividida per a varis jugadors en el qual s'haurà de renderitzar l'escena per totes les vistes. Normalment però només disposarem d'una vista a la vegada.

Seguidament es fan els càlculs de la finestra a on es renderitzarà l'escena. Inicialitzem el vertex shader utilitzant el shader que proporciona **View** i indiquem al renderer quines zones de l'escena s'hauran de renderitzar utilitzant l'esfera delimitadora anteriorment calculada.

La operació **SetShaderTempLightingLightcuts(...)** és el punt més important d'aquest mètode ja que aquest és el punt a on s'inicialitzarà el píxel shader, el qual serà el nostre punt d'atenció a la hora de fer els càlculs amb l'arbre de Lightcuts. Podem veure com utilitzem **LightSceneInfo** durant tot el procés per a passar les dades referents a la geometria, materials i altres aspectes de l'escena. Finalment s'indica al renderer que es faci un reset del buffer per tal de no afectar a les següents operacions.

Passem ara a estudiar la funció **SetShaderTempLightingLightcuts(...)**. Aquesta funció inicialitza el Pixel Shader i permet passar-li les dades necessàries pel render.

```
static void SetShaderTemplLightingLightcuts(FRHICommandList& RHICmdList, const FViewInfo&
    View, FShader* VertexShader, const FLightSceneInfo* LightSceneInfo)
{
   TShaderMapRef<TDeferredLightPS<false, false, true, false, true> > PixelShader(View.
       ShaderMap);
   SetGlobalBoundShaderState(RHICmdList, View.GetFeatureLevel(), PixelShader->
       GetBoundShaderState(), GetDeferredLightingVertexDeclaration<bRadialAttenuation>(),
       VertexShader, *PixelShader);
   PixelShader->SetLightcutsParameters(RHICmdList, View, LightSceneInfo);
}
```

La primera operació que observem és la creació del shader. Podem veure que s'utilitza una classe template a causa dels paràmetres que li passem abans de definir el tipus. En aquest cas, es va afegir un últim paràmetre per tal de poder activar fàcilment el render de Lightcuts ja que aquests paràmetres definiran quin tipus d'operació es fa dins el shader. Els paràmetres d'aquest template són els següents:

```
template<bool bUseIESProfile, bool bRadialAttenuation, bool bInverseSquaredFalloff, bool
    bVisualizeLightCulling, bool bLightcuts>
class TDeferredLightPS : public FGlobalShader
{
   // Code
}
```

S'ha afegit el paràmetre **bLightcuts** per tal d'activar el render segons Lightcuts. D'aquesta forma, quan el motor indica el tipus de shader que s'ha d'utilitzar, també li indica la funció que es farà servir dins el render per a calcular el color de l'escena. En aquest cas i amb els paràmetres que s'han passat, s'activa la implementació del shader següent:

```
IMPLEMENT_DEFERREDLIGHT_PIXELSHADER_TYPE(false, false, true, false, true, TEXT("
    LightcutsPixelMain"));
```

Això està indicant que s'implementarà un pixel shader que utilitzarà la funció LightcutsPixelMain, implementada per a la ocasió dins el shader i que serà utilitzada per a calcular el color

de l'escena segons el que indiqui l'arbre de Lightcuts.

Tornant a la funció **SetShaderTempLightingLightcuts(...)** veiem que es fa una crida al mètode **SetLightcutsParameters(...)** del Pixel Shader. D'aquesta funció la part més important és al final, a on un altre cop cridem a un mètode per afegir més paràmetres al Pixel Shader però definim l'estructura de dades amb la qual passarem l'arbre de Lightcuts. En aquest cas l'estructura és la ja definida **FLightcutsUniformStruct** (Secció 8.2.2.4). Podem veure el mètode al fragment de codi següent:

```
void SetLightcutsParameters(FRHICommandList& RHICmdList, const FSceneView& View, const
    FLightSceneInfo* LightSceneInfo)
{
   const FPixelShaderRHIParamRef ShaderRHI = GetPixelShader();
   SetParametersBase(RHICmdList, ShaderRHI, View, LightSceneInfo->Proxy->
       GetIESTextureResource());
   SetDeferredLightcutsParameters(RHICmdList, ShaderRHI, GetUniformBufferParameter<
       FLightcutsUniformStruct>(), LightSceneInfo, View);
}
```

D'aquest mètode, passem a **SetDeferredLightcutsParameters(...)** el qual ja és el punt final de les modificacions que s'han fet al sistema per adaptar-lo a les nostres necessitats. Aquí podem veure l'implementació:

```
void SetDeferredLightcutsParameters(
FRHICommandList& RHICmdList,
const ShaderRHIParamRef ShaderRHI,
const TShaderUniformBufferParameter<FLightcutsUniformStruct>&
    LightcutsUniformBufferParameter,
const FLightSceneInfo* LightSceneInfo,
const FSceneView& View)
{
   // Create the Struct to pass the Lightcuts data to the shader
   FLightcutsUniformStruct LightcutsUniformsValue;

   TArray<FLightcutsLight> LightcutsArray = LightSceneInfo->Scene->LightcutsArray;
   LightcutsUniformsValue.NumLights = LightcutsArray.Num();

   for (int i = 0; i < LightcutsArray.Num(); i++)
   {
      FLightcutsLight Light = LightcutsArray[i];

      // Transform the Color into LinealColor
      FVector4 NewColor = FVector4(Light.GetColor().R, Light.GetColor().G, Light.GetColor().
          B, 1.0f);
      NewColor = FVector4(FMath::Pow(NewColor.X, 2.2), FMath::Pow(NewColor.Y, 2.2), FMath::
          Pow(NewColor.Z, 2.2), NewColor.W);
      NewColor *= 0.025429833;
      NewColor *= 16.0f;

      // Pass the needed values to the shader struct
      LightcutsUniformsValue.LightsPosition[i] = Light.GetPosition();
      LightcutsUniformsValue.LightsColor[i] = NewColor;
      LightcutsUniformsValue.LightsIntensityRadiusFalloff[i] = FVector(Light.GetIntensity(),
           Light.GetRadius(), Light.GetFalloffExponent());
   }

   SetUniformBufferParameterImmediate(RHICmdList, ShaderRHI, LightcutsUniformBufferParameter
       , LightcutsUniformsValue);
}
```

El funcionament d'aquest últim mètode és simple, tot i que té alguns apartats que cal discutir a causa del funcionament del render de l'UE4. Inicialment podem veure com es crea la tupla del tipus **FLightcutsUniformStruct** que s'ha indicat anteriorment. Aquesta permetrà passar l'arbre de Lightcuts al shader fàcilment.

Seguidament es fa un bucle per passar totes les llums que formen l'arbre a la tupla del shader. El pas és simple i només cal assignar els valors. Tot i això, cal fer algunes modificacions amb el color, ja que nosaltres l'estem guardan com a RGB dins el rang 0-255. Cal modificar-lo ja que dins el shader aprofitarem per utilitzar el càlcul d'il·luminació original del motor, però només amb les llums que indiqui l'algoritme de Lightcuts. Això vol dir que hem de modificar el color per passar-lo a sRGB. Aquest canvi, fet a través de la correcció gamma, es porta a terme per aconseguir una varietat de color menys fosca i la qual dona una major profunditat a les escenes.

Aquí va venir un dels problemes del projecte a causa que no sabíem els valors de la funció matemàtica per a transformar el color original al nou espai. Sabíem que la correcció gamma es fa seguint la funció següent:

$$C_{out} = AC_{in}{}^{\gamma}$$

Tot i això, es va fer una bona feina de debug per estudiar les transformacions del color utilitzant llums puntuals normals (no formen part de l'arbre de Lightcuts). D'aquesta forma vam poder observar el valor inicial i el valor transformat. Llavors, les úniques variables que faltaven eren el valor de codificació gamma (γ) i la constant A. Tenint en compte que és una funció potencial, podem transformar-la per obtenir la següent igualtat:

$$\log C_{out} = \log A + \gamma \log C_{in}$$

Si fem una regressió lineal amb els valors originals del color i el corresponent valor final, tal com podem veure a la Figura 9.2, podem obtenir fàcilment els valors de (γ) i la constant A.

In Color	255	216	171	123	85	76	59
Out Color	5000	3470	2076	1005	445	349	200

Figura 9.2: Taula de valors de la regressió

Així doncs, podem obtenir fàcilment el valor de γ obtenint el pendent de la regressió. D'altra banda, la constant A, s'obté trobant el punt d'intersecció amb l'eix d'absisces y, i fent la conversió del logaritme. D'aquesta forma obtenim els seus valors:

$$\gamma = 2.2$$

$$\log A = -1,594656487 \rightarrow A = 10^{\log A} = 0,02542983329$$

Un cop s'han obtingut aquests valors, ja podem fer les transformacions que necessitem per a poder utilitzar parts del renderer de l'UE4, tal com veurem a la part dels shaders (Secció 9.6). Així doncs, ara entenem el perquè de les operacions que es duen a terme sobre el color al mètode **SetDeferredLightcutsParameters(...)**. Finalment el valor del color és multiplica per 16 per fer una correcció d'unitats lúmen per tal de fer funcionar el càlcul del shader correctament.

9.6 Implementacio dels shaders

Amb els anteriors passos ja hem fet totes les operacions necessàries i tenim les dades de l'arbre de Lightcuts al shader. L'únic pas que falta és fer el càlcul de quines llums s'utilitzaran per a renderitzar el color de cada punt de l'escena. Així doncs, la última part que falta descriure són els shaders i com s'ha enfocat la implementació. Totes les modificacions i implementacions dels shaders s'han dut a terme a l'arxiu **DeferredLightPixelShaders.usf**, que és en el qual s'hi troben totes les implementacions dels diferents Pixel Shaders.

A continuació descriurem l'estructura dividint el programa main en blocs segons la seva feina, i per fer més fàcil la seva entesa. Primer de tot tenim la inicialització del shader i de les variables necessàries.

```
// [INITIALITZATION]
void LightcutsPixelMain(float4 InScreenPosition : TEXCOORD0, float4 SVPos : SV_POSITION, out
     float4 OutColor : SV_Target0)
{
   // Initialize the OutColor and get info about the pixel of the screen to be painted
   OutColor = 0;
   float2 ScreenUV = InScreenPosition.xy / InScreenPosition.w * View.ScreenPositionScaleBias
       .xy + View.ScreenPositionScaleBias.wz;
   FScreenSpaceData ScreenSpaceData = GetScreenSpaceData(ScreenUV);

   // Calculate the scene depth, the world position and the camera vector
   float SceneDepth = CalcSceneDepth(ScreenUV);
   float4 HomogeneousWorldPosition = mul(float4(InScreenPosition.xy / InScreenPosition.w *
       SceneDepth, SceneDepth, 1), View.ScreenToWorld);
   float3 WorldPosition = HomogeneousWorldPosition.xyz / HomogeneousWorldPosition.w;
   float3 CameraVector = normalize(WorldPosition - View.ViewOrigin.xyz);

   // Initialize the array that will determine the lights to be used for the render
   // 0 = No used, 1 = Used
   int UsedLights[100];
   int FinalLight = 0;
   for( int i = 0; i < LightcutsUniforms.NumLights; i++ )
   {
      UsedLights[i] = 0;
   }
   UsedLights[0] = 1;

   // [LIGHTCUTS TREE ITERATION]
}
```

El funcionament d'aquesta primera part del shader és simple, ja que cal fer la inicialització de les variables que s'utilitzaran més endavant per a calcular el color del píxel. El primer pas és inicialitzar el color i trobar la informació que necessitem segons el píxel de la pantalla que s'hagi de pintar.

Acte seguit, es buscarà la fondària del punt que s'ha de renderitzar així com les coordenades d'aquest mateix dins el món. Aquestes coordenades del món serviran també per obtenir el vector de direcció de la càmera, utilitzant també la informació que proporciona **View**, la qual prové del codi C++ en el qual hem anat passant la informació de l'escena i les vistes.

Finalment, cal inicialitzar l'array **UsedLights** la qual permetrà saber quines llums s'utilitzaran per a calcular el color. Inicialment només s'utilitza l'arrel de l'arbre de Lightcuts per la qual cosa a part del primer ítem, totes les altres llums estaran desactivades. Observem que la

llista té una capacitat fixa. Això és a causa que els shaders es compilen al iniciar l'aplicació, per la qual cosa no podem passar arrays dinàmics. Això provoca que, si mai es vol utilitzar un arbre de Lightcuts amb un nombre d'ítems superior, s'hagi de modificar el codi font del motor. La variable **FinalLight** indica la posició de la última llum que comprovarem per a calcular el color. Durant la iteració de l'arbre s'anirà modificant el valor d'aquesta si cal comprovar més llums.

Al següent fragment de codi podem veure la part en que es calcula quines llums seran utilitzades per aconseguir el color final.

```
// [LIGHTCUTS TREE ITERATION]
{
   // [INITIALIZATION]

   // Iterate until i > FinalLight
   for( int i = 0; i <= FinalLight; i++ )
   {
      // If the actual light is used, check the evaluation criteria. If good render the
          color, if not chech this light as unused and mark the child lights (if they exist)
           as used
      if (UsedLights[i] == 1)
      {
         // Get the distance between the point and the actual light
         float3 LightPosition = LightcutsUniforms.LightsPosition[i].xyz;
         float PointToLightDist = sqrt(Square(WorldPosition.x - LightPosition.x) + Square(
             WorldPosition.y - LightPosition.y) + Square(WorldPosition.z - LightPosition.z));

         // Evaluate the evaluation criteria and update the used lights if necessary
         if (PointToLightDist < LightcutsUniforms.LightsIntensityRadiusFalloff[i].y)
         {
            int LChild = 2*i+1;
            int RChild = 2*i+2;
            if (LChild < LightcutsUniforms.NumLights)
            {
               UsedLights[LChild] = 1;
               FinalLight = LChild;

               if (RChild < LightcutsUniforms.NumLights)
               {
                  UsedLights[RChild] = 1;
                  FinalLight = RChild;
               }

               UsedLights[i] = 0;
            }
            else
            {
               // [CALCULATE OUTCOLOR]
            }
         }
         else
         {
            // [CALCULATE OUTCOLOR]
         }
      }
   }
}
```

La primera observació que fem és que el bucle arriba fins a **FinalLight** per la raó que ja s'ha explicat anteriorment. Seguidament es comprova si la llum sobre la qual estem iterant està activa o no. Si està activa, vol dir que forma part del cut actual, per la qual cosa comprovem si aproxima la llum adequadament. Per a fer aquesta comprovació utilitzarem el criteri d'avaluació explicat a la Secció 5.2.3. Així doncs, si el punt actual compleix el criteri de distància respecte el punt, deixarem d'utilitzar aquesta llum i comprovarem els seus fills. Això es fa així, a causa que sabem que la distància és prou petita com per a que els fills puguin donar una il·luminació. Per aquest motiu, desactivarem la llum i mirarem d'activar els fills de la llum actual, si aquests existeixen. D'altra banda, si ja hem arribat al punt a on veiem que utilitzar més llums no millorarà el resultat final, obtenim el color que aquesta llum proporciona al punt de l'escena. El color s'anirà sumant sempre ja que en una iteració es poden utilitzar moltes llums diferents. Aquest càlcul de color el podem veure molt més clarament al següent fragment de codi.

```
// [CALCULATE OUTCOLOR]
{
   // Initialize the necessary data from the Lightcuts Tree
   FDeferredLightData LightData = SetupLightDataForLightcutsDeferred(i);

   // Calculate the OutColor using UE4 methods
   float4 NewOutColor = GetDynamicLighting(WorldPosition, CameraVector, ScreenUV,
       ScreenSpaceData, ScreenSpaceData.GBuffer.ShadingModelID, LightData,
       GetPerPixelLightAttenuation(ScreenUV));
   NewOutColor *= ComputeLightProfileMultiplier(WorldPosition, LightcutsUniforms.
       LightsPosition[i].xyz, float3(-1.0, -0.0, -0.0));

   // Add the calculated color to the existing color
   OutColor += NewOutColor;
}
```

La tupla **FDeferredLightData** que trobem aquí al shader permet guardar totes aquelles dades necessàries per a calcular la il·luminació utilitzant els mètodes originals dels shaders. Això vol dir que aconseguirem una il·luminació molt bona només utilitzant les llums que indiqui l'arbre de Lightcuts. Així doncs, primer de tot haurem de passar-li les dades a aquesta nova variable. Això ho farem amb la funció **SetupLightDataForLightcutsDeferred(i)**. Aquesta funcio agafa els valors de la tupla **LightcutsUniform**, la qual prové del codi C++ anteriorment explicat (Secció 9.5), i els passa a la nova tupla.

Llavors, es calcula el color de sortida que dòna aquesta llum utilitzant els mètodes ja existens que proporcionen els shaders de l'UE4. Aquest mètode de càlcul és igual a com s'aconsegueix el color de sortida d'una sola llum, per la qual cosa els resultants seran molt semblants. Només tindrem variacions a causa del criteri d'aproximació utilitzat així com alguns valors per defecte que es donen a la variable **LightData** quan s'inicialitza.

Dònem valors per defecte a alguns paràmetres d'aquesta tupla a causa que no disposem de totes les dades necessàries per a utilitzar-la. Tot i així, els valors que es posen per defecte no són rellevants per al càlcul ja que normalment no es varien.

Com podrem veure, els paràmetres bàsics de posició, radi, color, intensitat i l'exponent falloff els introduïm utilitzant els valors obtinguts amb el càlcul de l'arbre de Lightcuts. D'altra banda, la resta de valors seran iguals a totes les llums. Tot i això, podem observar com la majoria d'aquests valors no els utilitzarem, ja que la direcció de la llum no importa en una llum puntual, la llum no produeix ombres i sempre treballarem tenint en compte que la llum surt d'un punt (no té forma allargada com un fluorescent).

Al següent fragment de codi podem observar com s'inicialitza la variable i quins valors es posen per defecte en els paràmetres no rellevants.

```
FDeferredLightData SetupLightDataForLightcutsDeferred(int i)
{
   // Values obtained from Lightcuts Tree
   FDeferredLightData LightData;
   LightData.LightPositionAndInvRadius = float4(LightcutsUniforms.LightsPosition[i].xyz, 1 /
        LightcutsUniforms.LightsIntensityRadiusFalloff[i].y);
   LightData.LightColorAndFalloffExponent = float4(LightcutsUniforms.LightsColor[i].xyz,
       8.0);

   // Default values (mostly not used on Point Lights)
   LightData.LightDirection = float3(-1.0, -0.0, -0.0);
   LightData.SpotAnglesAndSourceRadius = float4(-2.0, 1.0, 0.0, 0.0);
   LightData.MinRoughness = 0.08;
   LightData.DistanceFadeMAD = float2(0.0, 0.0);
   LightData.ShadowMapChannelMask = float4(0.0, 0.0, 0.0, 0.0);

   // We use inverse squared calculation and a Point Light is a type of radial Light. Not a
       Spot Light and the light does not produce shadows
   LightData.bInverseSquared = true;
   LightData.bRadialLight = true;
   LightData.bSpotLight = false;
   LightData.bShadowed = false;

   return LightData;
}
```

9.7 Proves de funcionament

Finalment, es van dur a terme un parell de proves per a comprovar que el sistema d'elecció dels cuts de l'arbre funciona correctament. Per aconseguir això, es va modificar la sortida per tal que es mostrés un color diferent per a cada cut de llums que s'escollia. D'aquesta manera, aconseguim veure quins són els punts en que s'utilitzen les llums més properes i quins punts tenen una aproximació menor.

9.7.1 Test 1

Així doncs, per a la primera prova es va muntar un sistema de tres llums col·locades tal com es pot observar a la Figura 9.3. En aquesta prova la llum verda i la vermella estan més a prop entre elles que la blava. D'aquesta forma, la primera parella de llums que es farà al càlcul serà entre aquestes dues llums. Llavors tindrem una nova parella entre la nova llum i la blava.

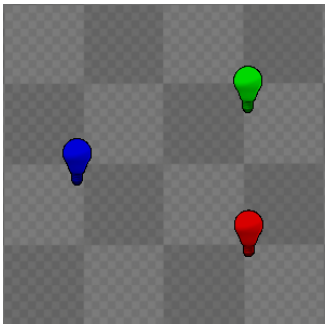


Figura 9.3: Posició de les llums al Test 1

Així doncs, amb aquesta estructura de llums obtenim el següent arbre de Lightcuts (Figura 9.4), el qual mostra clarament el procés que hem explicat anteriorment. Veiem que s'han marcat els diferents cuts possibles, cada un amb un color diferent. En aquest cas, al ser un arbre molt simple, només tenim tres talls possibles.

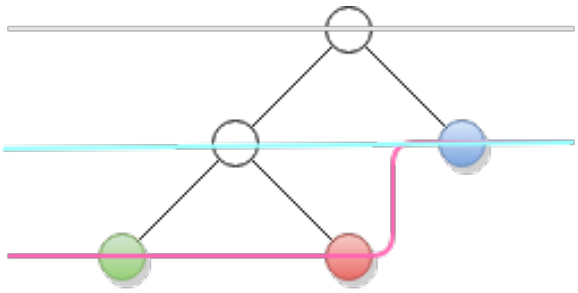


Figura 9.4: Arbre de Lightcuts del Test 1

Finalment, podem veure quin seria el resultat si pintem l'escena per cuts, és a dir, cada punt agafarà el color del cut que l'està pintant. Això vol dir que tindrem una zona magenta, una de cian i una altra de blanca. A la Figura 9.5 podem observar aquest resultat final. Cal comentar que la zona blanca que surt no s'hauria de pintar, ja que queda fora de l'abast de les llums.

Observem com els punts que queden més aprop de les llums verda i vermella es renderitzen amb el millor cut (totes les llums arrel), mentre que els més distants, s'utilitzen llums superiors de l'arbre i per tant s'està aproximant el valor.

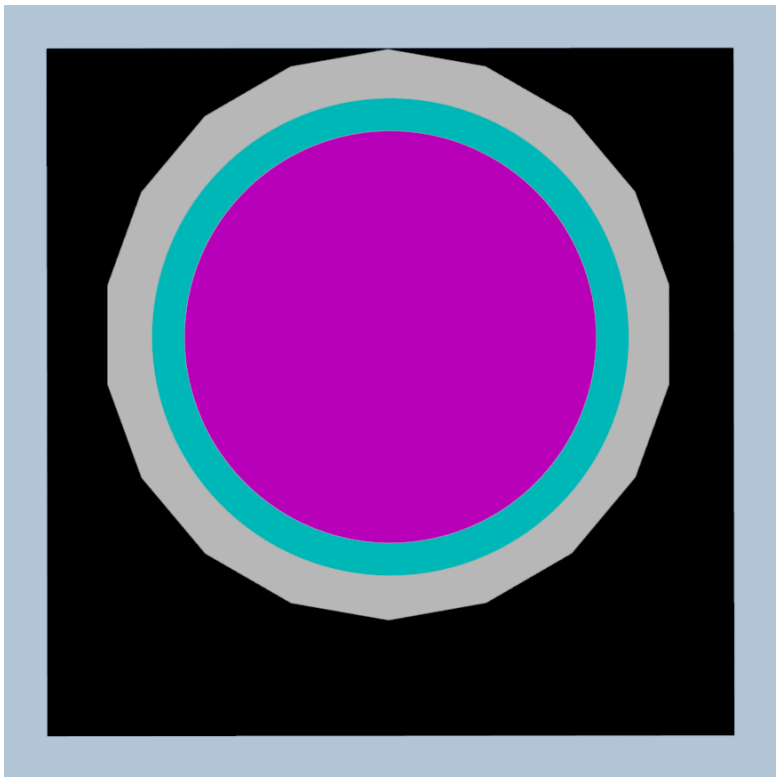


Figura 9.5: Prova de render del Test 1

Finalment, podem observar a la Figura 9.6 el resultat que obtenim si fem el renderitzat de la llum en temps real utilitzant l'arbre de Lightcuts calculat. En aquest cas podem observar que a causa de les poques llums les diferències no són apreciables amb el resultat que s'aconseguiria fent un render normal.

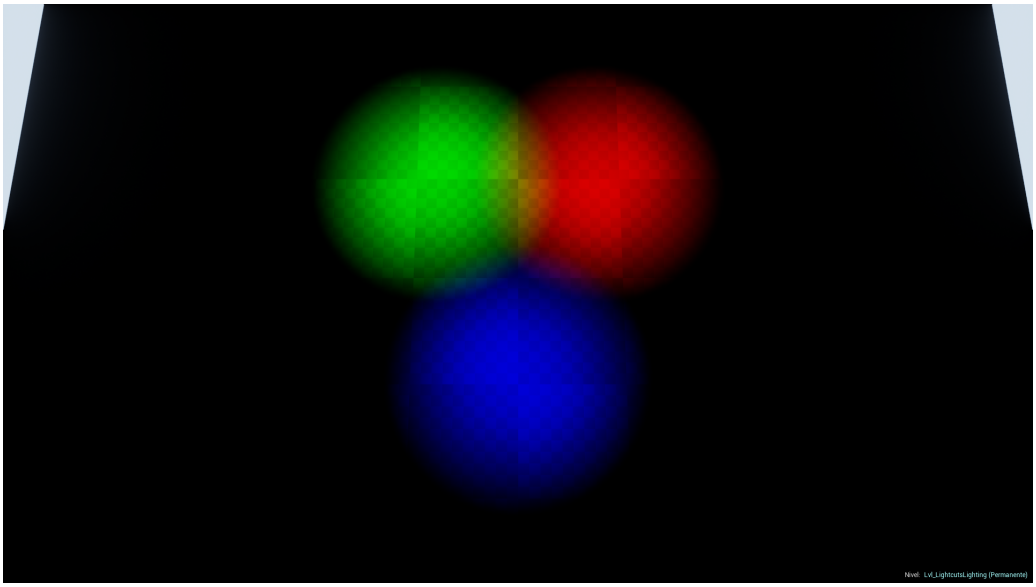


Figura 9.6: Render en temps real del Test 1 utilitzant l'arbre de Lightcuts

9.7.2 Test 2

Es va dur a terme una segona prova en la qual hi trobem un total de 5 llums, la qual cosa permet crear un arbre de Lightcuts més complex. En aquest cas, l'estructuració de les llums és la que trobem a la Figura 9.7. Observem també que es pot definir a simple vista quines seran les parelles que es faran, ja que simplifica la feina d'anàlisis i permet veure fàcilment quins cuts afecten a cada punt.

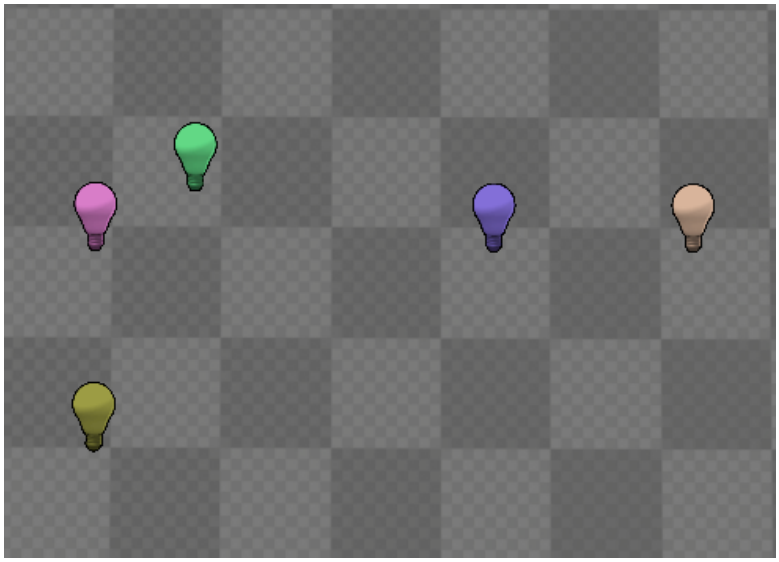


Figura 9.7: Posició de les llums al Test 2

Així doncs, a la Figura 9.8 podem observar l'estructura de l'arbre que obtenim si fem els càlculs. En aquest cas no s'han mostrat els diferents talls, ja que en surten molts i no es mostra clarament en el diagrama com queden.

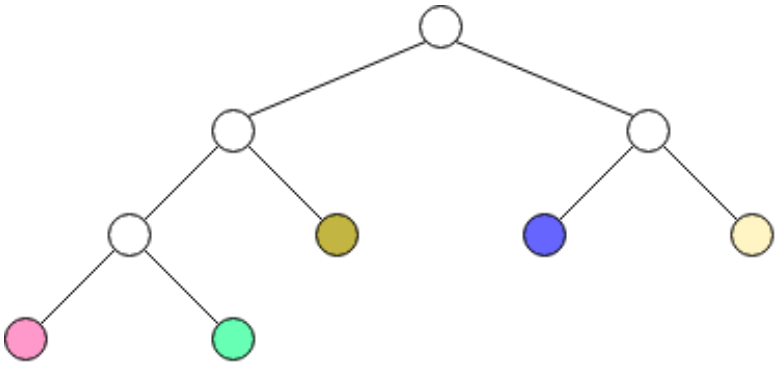


Figura 9.8: Arbre de Lightcuts del Test 2

Tot i així, quan observem el resultat final a la Figura 9.9, veiem com s'utilitzen els diferents cuts. També es pot veure clarament que, com més aprop es troba un punt de les llums, més avall de l'arbre arribarem, ja que rep molta il·luminació de moltes llums i per tant s'haurà d'aproximar més bé el valor. D'altra banda, si el punt es troba més lluny, no caldrà utilitzar tantes llums per a calcular el color, ja que no rep tanta energia de les llums i per tant podem obtenir un resultat semblant sense haver d'utilitzar totes les llums de l'escena.

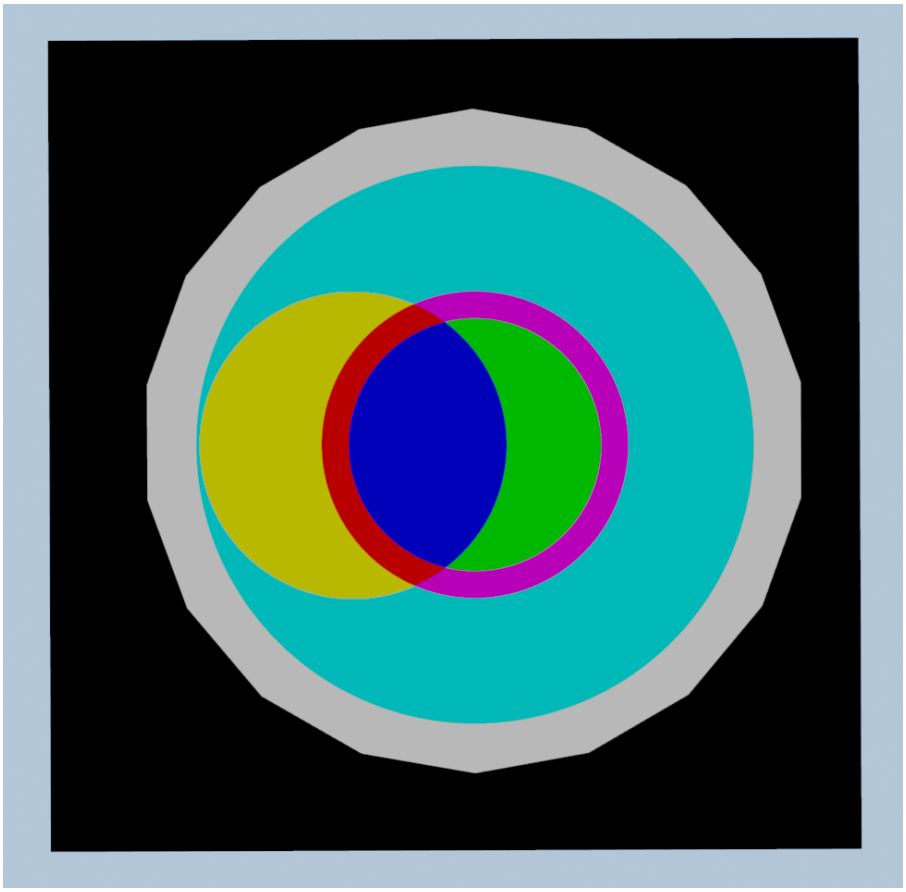


Figura 9.9: Prova de render del Test 2

De la mateixa manera que el primer test, a la Figura 9.10 podem observar el resultat final obtingut utilitzant l'arbre de Lightcuts. Igual que el cas anterior, a causa de la falta d'objectes a l'escena i la proximitat de les llums, no podem apreciar un resultat espectacular. Tot i això, podem veure l'aplicació dels cuts que s'han observat a la Figura 9.9. Al Capítol 10 podrem veure resultats més clars sobre la diferència entre el render normal i un render segons Lightcuts.

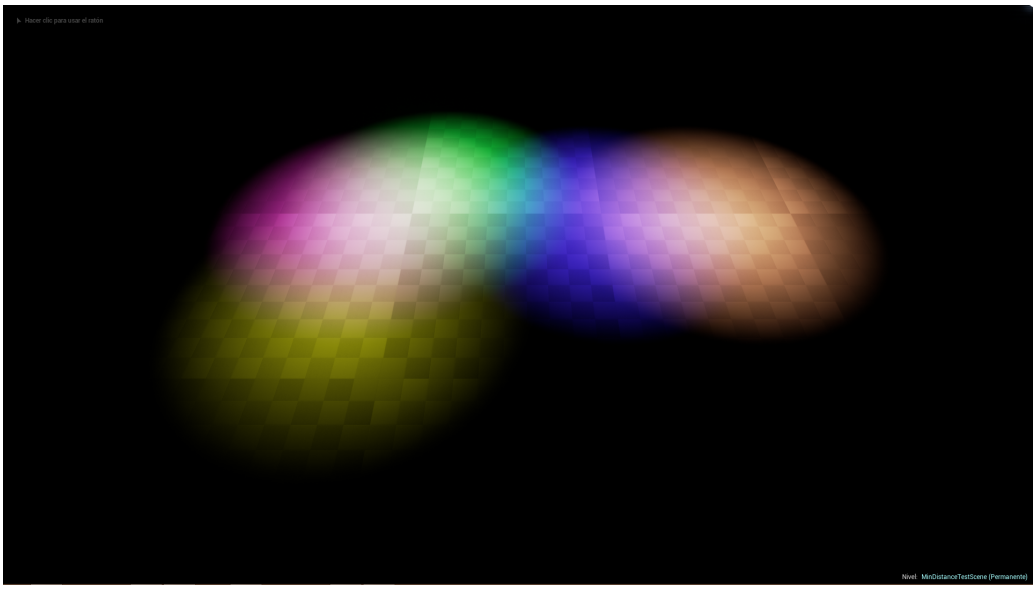


Figura 9.10: Render en temps real del Test 2 utilitzant l'arbre de Lightcuts

Capítol 10

Implantació i resultats

En aquest capítol mostrarem els resultats que s'han obtingut. Es compararan els resultats obtinguts renderitzant l'escena utilitzant la tècnica de Lightcuts amb els resultats que dòna el render original de l'UE4.

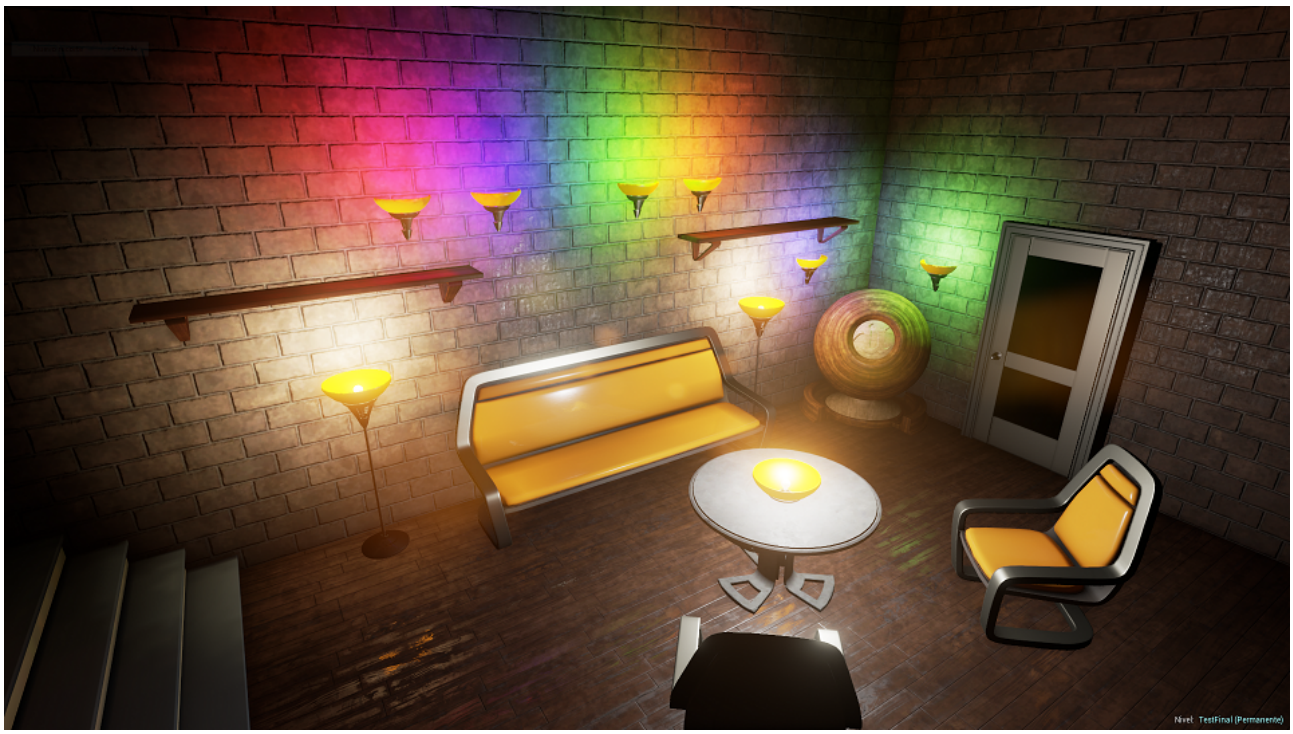
10.1 Escena final

Per a fer les comprovacions i mostrar el resultat final obtingut, es va muntar una escena a l'editor de l'UE4, la qual semblés una habitació que hom es pugui trobar en un videojoc. Així doncs, l'escena que s'ha implementat té les següents característiques:

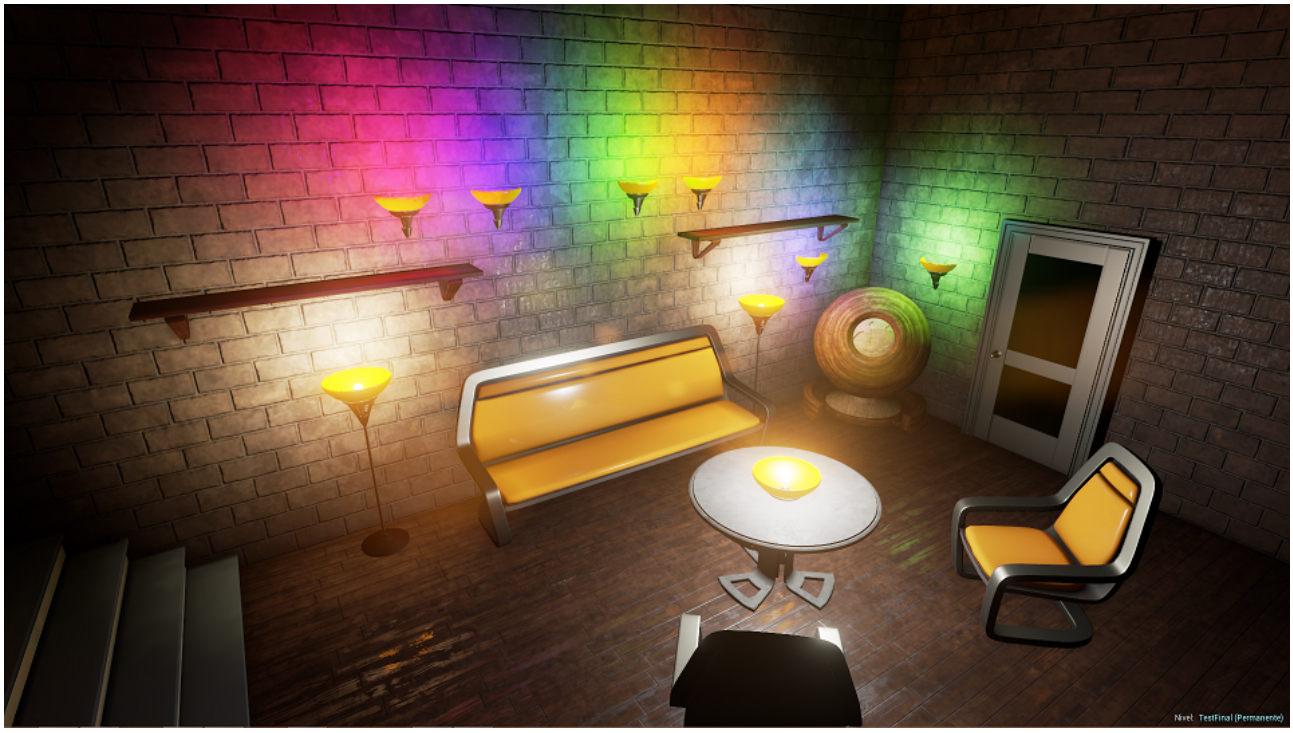
- Habitació de tamany mitjà.
- 1 sofà
- 2 cadires
- 1 tauleta de cafè
- 2 estanteries
- Porta d'entrada
- Escales al pis superior
- 10 llums de característiques diferents

Així doncs, aquesta escena servirà per a demostrar la feina feta durant el projecte i per comprovar si s'han complert els objectius, així com respondre algunes preguntes inicials, tals com si és possible implementar la tècnica en un entorn en temps real.

Per tant, a la Figura 10.1 podem observar un render normal de l'UE4 juntament amb el render fet segons el que indica l'arbre de Lightcuts. A primera vista no s'observen diferències abismals i, per tant, vol dir que l'aproximació que s'està aconseguint és bona i la feina s'està fent correctament. Tot i això, si ens hi fixem una mica podem observar que, en certs punts, hi ha diferències de colors. En general aquestes diferències passen allà a on hi ha llums properes de colors molt diferenciats. Els punts més clars a on es veu aquest fet és sobre les dues estanteries i sobre l'escultura situada al costat de la porta.



(a) Render orignal de l'UE4



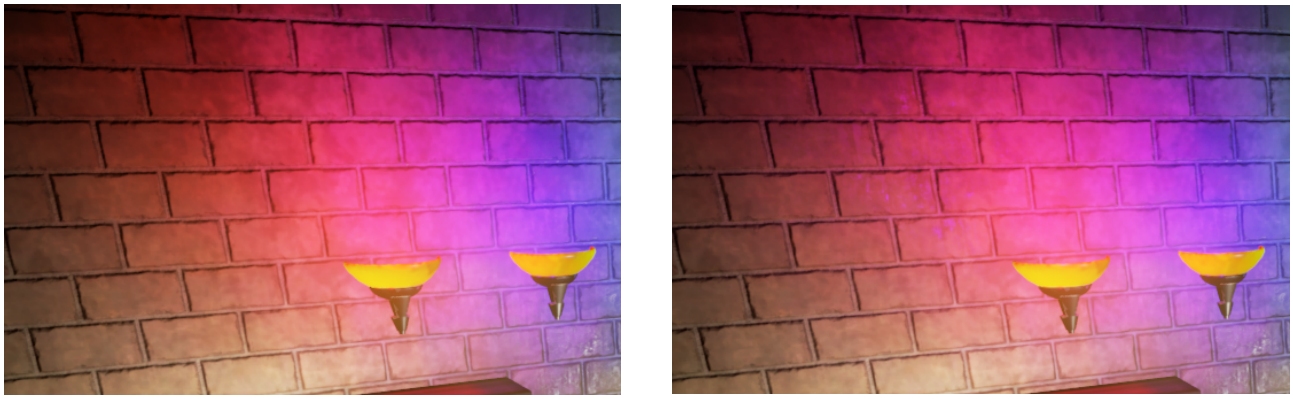
(b) Render utilitzant l'arbre de Lightcuts

Figura 10.1: Comparació de resultats

Aquestes diferències venen produïdes per com està construït l'arbre de Lightcuts. Com que cada vegada que es construeix una nova llum s'agafa la suma d'intensitat dels dos fills, així com un radi més gran que englobi els dos fills, provoca que la nova llum arribi més lluny amb més força. Això no és rellevant si renderitzem un punt proper a les llums, ja que s'agafen les llums originals de l'escena i el càlcul és correcte. D'altra banda, quan el punt és més llunyà, agafem la llum creada a partir de les altres dues, la qual té una major intensitat i radi d'acció. Això provoca que, per exemple, a l'estanteria de l'esquerra (Figura 10.1b) s'observi un sutil augment

del color rosa, donat que la llum creada a partir de les llums vermella i blava agafa el color combinat. Per tant, el fet que la paret es vegi una mica més rosa en punts llunyans ve definint per com està construit l'algoritme. Això també podem trobar-ho a l'estanteria dreta així com a l'escultura al costat de la porta, a on s'aprecia que el color verd-blau resultant de les dues llums en aquest cas arriba fins al límit de la porta.

Aquestes diferències les podem apreciar més bé a la Figura 10.2. Podem observar com a la Figura 10.2b el color rosat de la paret arriba fins a punts més llunyans de la imatge que al render original (Figura 10.2b). Podem apreciar més bé aquest fet a la part esquerra de les imatges, a on les llums vermella i blava no afecten i per tant s'agafa la nova llum creada a partir d'aquestes dues llums. Tot i això, com podem apreciar pel fet de necessitar una imatge comparativa, en la qual s'ha hagut de fer un augment de les imatges per a veure clarament les diferències, indica que els errors d'aproximació que ens podem trobar són molt petits. A més, cal tenir en compte que aquí s'està comparant una imatge estàtica, fet que no passarà mai en un joc, ja que s'estarà continuament en moviment. Això encara farà més difícil observar aquests petits detalls, per la qual cosa podem donar poca importància als errors d'aproximació que poden aparèixer.



(a) Render orignal de l'UE4 (b) Render utilitzant l'arbre de Lightcuts

Figura 10.2: Zoom per apreciar els errors d'aproximació

Com ja s'ha comentat, estem buscant obtenir una aproximació del que seria el càlcul de llum òptim, per la qual cosa aquests efectes de llum obtinguts són esperats. Per tant, podem dir que s'ha complert l'objectiu d'implementar la tècnica Lightcuts en un entorn en temps real com és l'UE4 i, per la qual cosa, podria ser utilitzada en un videojoc aconseguint uns resultats semblants als que s'aconseguirien utilitzant totes les llums.

D'altra banda, si parlem del tema d'eficiència, els resultats també han estat els esperats, ja que tal com s'explica a l'article original de la tècnica Lightcuts [1] (Annex 14.1), l'eficiència d'aquest algoritme és $O(\log n)$. Si comparem aquest cost amb el de l'algoritme original $O(n)$, observem clarament que quan es tingui una n gran, el temps d'execució per obtenir els resultats d'il·luminació seran menors que utilitzant l'algoritme original. Això és a causa d'utilitzar menys llums per aquells punts de l'escena que queden més allunyats dels punts de llum. Al estar lluny, es pot aproximar el valor de la il·luminació sense obtenir grans errors i, per tant, no gastar tants recursos fent operacions sobre totes les llums.

Així doncs, podem posar un gran nombre de llums a l'escena sense haver-nos de preocupar sobre el cost que suposaran a la hora de renderitzar cada frame. Per tant, això permetrà millorar altres aspectes del joc, ja sigui afegint més objectes, millorant la resolució de textures

o afegint més personatges controlats per intel·ligència artificial, entre altres. Tot i així, tenint en compte que els errors d'aproximació són poc visibles i que, tot i ser petit, sempre hi haurà una disminució en el temps de càlcul, pot ser bona idea implementar la tècnica en un entorn en temps real si es volen utilitzar únicament llums dinàmiques.

10.2 Validesa legal de l'aplicació

Sempre que es desenvolupa una aplicació cal tenir en compte la validesa legal d'aquesta, és a dir, cal que compleixi la legislació actual pel que fa a la Llei Orgànica de Protecció de Dades de Caràcter Personal (LOPD) així com la Llei de Serveis de la Societat de la Informació i Comerç Electrònic (LSSICE).

En el nostre cas, l'aplicació no emmagatzema cap tipus d'informació personal de l'usuari, per la qual cosa es compleix l'esmentat per la LOPD. D'altra banda, si parlem del comerç electrònic també es compleix el que diu la LSSICE, ja que l'aplicació no inclou cap servei de pagament.

Capítol 11

Conclusions

Per assolir els objectius d'aquest projecte es van marcar unes tasques a fer (Secció 1.3). Aquests objectius s'han pogut dur a terme, tot i trobar-nos problemes durant el desenvolupament.

L'objectiu principal d'implementar la tècnica d'il·luminació Lightcuts en un entorn en temps real com és el motor Unreal Engine 4 s'ha complert satisfactòriament. La tècnica funciona i el motor no té problemes d'estabilitat que puguin ser produïts pels canvis introduïts. D'altra banda, el procés d'implementació va ser tediós i problemàtic, principalment per la falta de documentació oficial del codi. Tot i que l'empresa del motor proporciona una documentació del codi, aquesta és reduïda i en molts casos deixa sense explicació classes i mètodes. Fins i tot s'han trobat punts de la documentació on aquesta és anticuada i no s'adequa al contingut actual del codi. Tot això provoca que el procés d'investigació hagi tingut una duració més llarga del que es podia esperar, provocant que la distribució de tasques inicial fos totalment diferent de la final.

Aquest procés d'investigació també cal tenir-lo en compte, ja que es va iniciar el projecte sense tenir cap coneixement del codi intern del motor. Si es tenien coneixements sobre el funcionament com a usuari de l'UE4, però aquests no servien de res a la hora d'iniciar el projecte. Així doncs, el procés inicial per trobar a on inserir el nou codi, estudiar tot el mòdul de Lightmass, així com el renderer de l'UE4 es va convertir en un procés el qual va durar durant tot el desenvolupament del treball final.

De cares als resultats obtinguts amb la tècnica de Lightcuts, aquests són correctes i podem comprovar que la tècnica es pot implementar correctament en un entorn en temps real. Tot i això, el pre-procés necessari per a calcular l'arbre de Lightcuts és molt lent i pot ser millorat si s'implementen alguns canvis (Secció 12). Tot i això, els resultats que s'obtenen en temps real són satisfactoris i compleixen els objectius de la tècnica: obtenir una il·luminació aproximada de l'escena utilitzant menys llums allà a on no sigui necessari. Tal com podrem haver observat, alguns punts de l'escena contenen errors d'aproximació molt petits, però això és esperat per la naturalesa de la tècnica.

Finalment, cal analitzar l'eficiència de la tècnica. En aquest apartat s'han obtingut els resultats esperats inicialment. Aquest resultat no és sorprenent ja que la tècnica ens assegura aconseguir un algoritme amb eficiència aproximadament $O(\log n)$ a causa de no utilitzar totes les llums disponibles per a renderitzar els punts més allunyats de les llums. D'aquesta manera, s'obté un càlcul d'il·luminació més eficient que l'algoritme original. Això permetrà afegir més llums a l'escena mantenint un cost de càlcul menor que utilitzant la tècnica original. Per tant, a l'hora de decidir si utilitzar la tècnica cal tenir en compte el nombre de llums que hi haurà a l'escena, tenint en compte que la millora en el temps d'eficiència es comença a notar en gran

mesura quantes més llums trobem a l'escena. Tot i així, veient que els errors d'aproximació són poc visibles i que, tot i ser petit, sempre hi haurà una disminució en el temps de càlcul, pot ser bona idea implementar la tècnica en un entorn en temps real si es volen utilitzar únicament llums dinàmiques.

Per tant, per a concloure, podem veure clar que la tècnica s'ha implementat correctament i el seu funcionament és l'esperat inicialment. A més, la seva eficiència és bona, ja que redueix els temps càlcul d'il·luminació en gran mesura quan hi ha moltes llums a l'escena. A part, els errors d'aproximació que apareixen no són gens evidents a simple vista, tal com s'ha pogut comprovar a l'apartat de resultats (Secció 10). Finalment, aquesta tècnica està pensada per a funcionar quan hi ha un nombre de llums molt elevat en una escena, la qual cosa pot no passar en un videojoc. De la mateixa manera, la tècnica de Lightcuts està pensada per a funcionar en temps real, la qual cosa comporta que totes les llums de l'arbre seran renderitzades en temps real. Aquí cal tenir en compte que molt sovint en videojocs s'utilitzen Lightmaps per tal de pre-calcular una gran part de la il·luminació estàtica, com pot ser la llum solar, per la qual cosa el nombre de llums dinàmiques d'una escena no serà mai molt elevat. Per tant, tot i funcionar la tècnica, els beneficis que comporta només són visibles a partir d'un nombre de llums dinàmiques molt elevat, per la qual cosa podem trobar-nos en que no passi a la hora de crear el joc. Així mateix, cal pensar si els errors d'aproximació que es troben al renderitzar valen la pena tenint en compte que la reducció del temps de càlcul serà només apreciable al arribar a un nombre de llums elevat.

Capítol 12

Treball futur

En aquest capítol presentarem una sèrie de millores o treballs futurs els quals poden millorar la feina aquí obtinguda, ja que ni molt menys el resultat fet és un treball tancat.

Així doncs, els punts més importants que podem trobar dins els possibles treballs futurs són:

- **Millora el pre-procés:** Millorar el temps de càlcul del pre-procés podria ser una de les millores possibles, ja que, tot i no ser important aquest temps, sempre és beneficiós aconseguir uns temps de càlcul menors. En aquest cas, es poden implementar mesures tals com gestionar les llums segons uns clústers basats en distàncies, que es calculin abans d'iniciar el càlcul de l'arbre. Això permetria no haver de provar tantes permutacions possibles de llums i aconseguir l'arbre més ràpidament.
- **Varis arbres en una mateixa escena:** Una altra millora podria ser crear un mètode pel qual no es calculi tot un arbre per una escena, sinò que es divideixin les llums de l'escena i es calculin diferents arbres. D'aquesta manera, si creem un volum que encapsuli una habitació, podem calcular l'arbre de Lightcuts només d'aquella habitació. Això permetria aconseguir uns resultats molt semblants a utilitzar només un arbre, però reduïria molt el pre-procés així com el temps de render en temps real, ja que només es passaria l'arbre que afecta a un punt.
- **Afegir més varietat de llums a l'algoritme:** Una de les tasques que es podria considerar important, seria afegir més tipus de llums possibles a l'algoritme. Això vol dir que es podria crear l'arbre utilitzant llums puntuals, focals o direccionals. D'aquesta forma, es podria reduïr encara més el cost d'il·luminació global sobre cada frame, ja que totes les llums serien renderitzades segons l'algoritme de Lightcuts.

Aquests dos punts comentats serien els més importants a tenir en compte per a treballs futur. Sobretot el segon punt pot ser beneficiós pel fet que permetria col·locar moltes més llums en una escena i separar els arbres d'aquestes afegint volums rectangulars o esfèrics al voltant de les llums de les quals volem calcular un arbre de Lightcuts. Així mateix, el treball queda obert a moltes altres millores les quals poden modificar els resultats obtinguts i millorar l'eficiència final.

Capítol 13

Bibliografia

[1] Adam Arbree Kavita Bala Michael Donikian Bruce Walter, Sebastian Fernandez and Donald P. Greenberg. Lightcuts: A scalable approach to illumination. *SIGGRAPH*, 2005.

[2] Epic Games. Unreal engine api reference, . URL https://docs.unrealengine.com/latest/INT/API/index.html.

[3] Epic Games. Unreal engine 4 documentation, . URL https://docs.unrealengine.com/latest/INT/index.html.

[4] Microsoft. High level shading language. URL https://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx.

Capítol 14

Annexos

14.1 Article original Lightcuts

To appear in the ACM SIGGRAPH conference proceedings

Lightcuts: A Scalable Approach to Illumination

Bruce Walter Sebastian Fernandez Adam Arbree Kavita Bala Michael Donikian Donald P. Greenberg
*Program of Computer Graphics, Cornell University**

Abstract

Lightcuts is a scalable framework for computing realistic illumination. It handles arbitrary geometry, non-diffuse materials, and illumination from a wide variety of sources including point lights, area lights, HDR environment maps, sun/sky models, and indirect illumination. At its core is a new algorithm for accurately approximating illumination from many point lights with a strongly *sublinear* cost. We show how a group of lights can be cheaply approximated while bounding the maximum approximation error. A binary light tree and perceptual metric are then used to adaptively partition the lights into groups to control the error vs. cost tradeoff.

We also introduce reconstruction cuts that exploit spatial coherence to accelerate the generation of anti-aliased images with complex illumination. Results are demonstrated for five complex scenes and show that lightcuts can accurately approximate hundreds of thousands of point lights using only a few hundred shadow rays. Reconstruction cuts can reduce the number of shadow rays to tens.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture;

Keywords: many lights, raytracing, shadowing

Figure 1: Bigscreen model: an office lit by two overhead area lights, two HDR flat-panel monitors, and indirect illumination. Our scalable framework quickly and accurately computed the illumination using 639,528 point lights. The images on the monitors were also computed using our methods: lightcuts and reconstruction cuts.

1 Introduction

While much research has focused on rendering scenes with complex geometry and materials, less has been done on efficiently handling large numbers of light sources. In typical systems, rendering cost increases linearly with the number of lights. Real world scenes often contain many light sources and studies show that people generally prefer images with richer and more realistic lighting. In computer graphics however, we are often forced to use fewer lights or to disable important lighting effects such as shadowing to avoid excessive rendering costs.

The lightcuts framework presents a new scalable algorithm for computing the illumination from many point lights. Its rendering cost is strongly sublinear with the number of point lights while maintaining perceptual fidelity with the exact solution. We provide a quick way to approximate the illumination from a group of lights, and more importantly, cheap and reasonably tight bounds on the maximum error in doing so. We also present an automatic and locally adaptive method for partitioning the lights into groups to control the tradeoff between cost and error. The lights are organized into a light tree for efficient partition finding. Lightcuts can handle non-diffuse materials and any geometry that can be ray traced.

Having a scalable algorithm enables us to handle extremely large numbers of light sources. This is especially useful because many other difficult illumination problems can be simulated using the illumination from sufficiently many point lights (e.g., Figure 1). We demonstrate three examples: illumination from area lights, from high dynamic range (HDR) environment maps or sun/sky models, and indirect illumination. Unifying different types of illumination within the lightcuts framework has additional benefits. For example, bright illumination from one source can mask errors in approximating other illumination, and our system automatically exploits this effect.

A related technique, called reconstruction cuts, exploits spatial coherence to further reduce rendering costs. It allows lightcuts to be computed sparsely over the image and intelligently interpolates between them. Unlike most interpolation techniques, reconstruction cuts preserve high frequency details such as shadow boundaries and glossy highlights. Lightcuts can compute the illumination from many thousands of lights using only a few hundred shadow rays. Reconstruction cuts can further reduce this to just a dozen or so.

The rest of the paper is organized as follows. We discuss previous work in Section 2. We present the basic lightcuts algorithm in Section 3, give details of our implementation in Section 4, discuss different illumination applications in Section 5, and show lightcut results in Section 6. Then we describe reconstruction cuts in Section 7 and demonstrate their results in Section 8. Conclusions are in Section 9. Appendix A gives an optimization for spherical lights.

2 Previous Work

There is a vast body of work on computing illumination and shadows from light sources (e.g., see [Woo et al. 1990; Hasenfratz et al. 2003] for surveys). Most techniques accelerate the processing of individual lights but scale linearly with the number of lights.

Several techniques have dealt explicitly with the many lights problem. [Ward 1994] sorts the lights by maximum contribution and then evaluates their visibility in decreasing order until an error

*email: {bjw,spf,arbree,kb,mike,dpg}@graphics.cornell.edu

bound is met. We will compare lightcuts with his technique in Section 6. [Shirley et al. 1996] divide the scene into cells and for each cell split the lights into important and unimportant lists with the latter very sparsely sampled. This or similar Monte Carlo techniques can perform well given sufficiently good sampling probability functions over the lights, but robustly and efficiently computing these functions for arbitrary scenes is still an open problem. [Paquette et al. 1998] present a hierarchical approach using light trees similar to the ones we will use. They provide guaranteed error bounds and good scalability, but cannot handle shadowing which limits the applicability. [Fernandez et al. 2002] accelerate many lights by caching per light visibility and blocker information within the scene, but this leads to excessive memory requirements if the number of lights is very large. [Wald et al. 2003] can efficiently handle many lights under the assumption that the scene is highly occluded and only a small subset contribute to each image. This subset is determined using a particle tracing preprocess.

Illumination from HDR environment maps (often from photographs [Debevec 1998]) is becoming popular for realistic lighting. Smart sampling techniques can convert these to directional point lights for rendering (e.g., [Agarwal et al. 2003; Kollig and Keller 2003]), but typically many lights are still required for high quality results.

Instant radiosity [Keller 1997] is one of many global illumination algorithms based on stochastic particle tracing from the lights. It approximates the indirect illumination using many virtual point lights. The resolvable detail is directly related to the number of virtual lights. This makes it a perfect fit with lightcuts, whereas previously it was largely restricted to quick coarse indirect approximations. [Wald et al. 2002] use it in their interactive system and added some clever techniques to enhance its resolution.

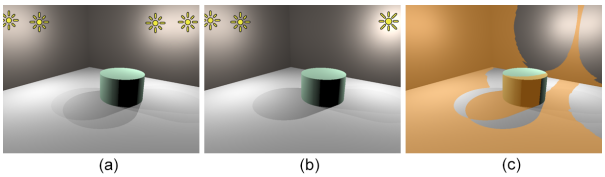
Photon mapping is another popular, particle-based solution for indirect illumination. It requires hemispherical final gathering for good results, typically with 200 to 5000 rays per gather [Jensen 2001, p.140]. In complex scenes, lightcuts compute direct and indirect illumination, using fewer rays than a standard hemispherical gather.

Hierarchical and clustering techniques are widely used in many fields. Well known examples in graphics include the radiosity techniques (e.g., [Hanrahan et al. 1991; Smits et al. 1994; Sillion and Puech 1994]). Unlike lightcuts, these compute view-independent solutions that, if detailed, can be very compute and storage intensive. Also since they use meshes to store data, they have difficulty with some common geometric flaws, such as coincident or intersecting polygons. Final gather stages are often used to improve image quality. The methods of [Kok and Jansen 1992] and [Scheel et al. 2001; Scheel et al. 2002] accelerate final gathers by trying to interpolate when possible and only shooting shadow rays when necessary. While very similar in goals to reconstruction cuts, they use heuristics based on data in a radiosity link structure.

Numerous previous techniques have used coherence and interpolation to reduce rendering costs. Reconstruction cuts' novelty and power comes from using the lightcuts framework. Like several previous methods, reconstruction cuts use directional lights to cheaply approximate illumination from complex sources. [Walter et al. 1997] use them for hardware accelerated walkthroughs of precomputed global illumination solutions. [Zaninetti et al. 1999] call them light vectors and use them to approximate illumination from various sources including area lights, sky domes, and indirect.

3 The Lightcuts Approach

Given a set of point light sources $\mathbb{S}$, the radiance L caused by their direct illumination at a surface point $\mathbf{x}$ viewed from direction ω is a



(a) (b) (c)

Figure 2: A simple scene with 4 point lights. (a) The exact solution. (b) Approximate solution formed by clustering the two lights on the right. (c) The orange region shows where the exact and clustered solutions are indistinguishable. Errors are typically largest near the lights and where their visibility differs.

product of each light's material, geometric, visibility and intensity terms summed over all the lights:

$$L_{\mathbb{S}}(\mathbf{x},\omega) = \sum_{i\in\mathbb{S}} \overbrace{M_i(\mathbf{x},\omega)}^{\text{material}} \underbrace{G_i(\mathbf{x})}_{\text{geometric}} \overbrace{V_i(\mathbf{x})}^{\text{visibility}} \underbrace{I_i}_{\text{intensity}} \tag{1}$$

The cost of an exact solution is linear in the number of lights since these terms must be evaluated for each light. To create a scalable, sublinear method, we need a way to approximate the contribution of a group of lights without having to evaluate each light individually.

Let us define a *cluster*, $\mathbb{C} \subseteq \mathbb{S}$, to be a set of point lights along with a representative light $j \in \mathbb{C}$. The direct illumination from a cluster can be approximated by using the representative light's material, geometric, and visibility terms for all the lights to get:

$$\begin{aligned} L_{\mathbb{C}}(\mathbf{x},\omega) &= \sum_{i\in\mathbb{C}} M_i(\mathbf{x},\omega)\, G_i(\mathbf{x})\, V_i(\mathbf{x})\, I_i \\ &\approx M_j(\mathbf{x},\omega)\, G_j(\mathbf{x})\, V_j(\mathbf{x}) \sum_{i\in\mathbb{C}} I_i \end{aligned} \tag{2}$$

The cluster intensity ($I_{\mathbb{C}} = \sum I_i$) can be precomputed and stored with the cluster making the cost of a cluster approximation equal to the cost of evaluating a single light (i.e. we have replaced the cluster by a single brighter light). The amount of cluster error will depend on how similar the material, geometric, and visibility terms are across the cluster. A simple example is shown in Figure 2.

Light Tree. No single partitioning of the lights into clusters is likely to work well over the entire image, but dynamically finding a new cluster partitioning for each point could easily prove prohibitively expensive. We use a global light tree to rapidly compute locally adaptive cluster partitions. A *light tree* is a binary tree where the leaves are individual lights and the interior nodes are light clusters containing the lights below them in the tree. A *cut* through the tree is a set of nodes such that every path from the root of the tree to a leaf will contain exactly one node from the cut. Thus each cut corresponds to a valid partitioning of the lights into clusters. An example light tree and three different cuts are shown in Figure 3.

While every cut corresponds to a valid cluster partitioning, they vary greatly in their costs and the quality of their approximated illumination. We need a robust and automated way to choose the appropriate cut to use locally. As the cuts will vary across the image, some points, or pixels, may use a particular cluster to reduce costs while others replace it with its children for increased accuracy. Such transitions could potentially cause objectionable image artifacts. To prevent this, we only use clusters when we can guarantee that the approximation error introduced by the cluster will be below a perceptual visibility threshold. Weber's law [Blackwell 1972] is a standard, well-known perceptual result that says the minimum perceptible change in a visual signal is roughly equal to a fixed percentage of the base signal. Under worst case conditions, humans can detect changes of just under 1%, though in practice,

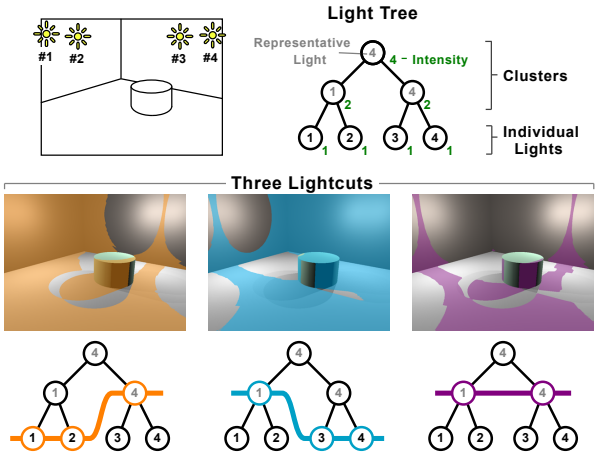


Figure 3: A light tree and three example cuts. The tree is shown on the top with the representative lights and cluster intensities for each node. Leaves are individual lights while upper nodes are progressively larger light clusters. Each cut is a different partitioning of the lights into clusters (the orange cut is the same as Figure 2). Above each cut, the regions where its error is small are highlighted.

the threshold is usually higher. In our experience, an error ratio of 2% results in no visible artifacts across a wide variety of scenes and was used for all our results. Changing this value can be used to vary the tradeoff between performance and accuracy.

Choosing Lightcuts. Using a relative error criterion requires an estimate of total radiance before we can decide whether a particular cluster is usable. To solve this difficulty, we start with a very coarse cut (e.g., the root of the light tree) and then progressively refine it until our error criterion is met. For each node in the cut we compute both its cluster estimate (Equation 2) and an upper bound on its error (Section 4.1). Each refinement step considers the node in the current cut with the largest error bound. If its error bound is greater than our error ratio times the current total illumination estimate, we remove it from the cut, replace it with its two children from the light tree, compute their cluster estimates and error bounds, and update our estimate of the total radiance. Otherwise, the cut obeys our error criterion and we are done. We call such a cut, a *lightcut*.

To make this process more efficient, we require that the representative light for a cluster be the same as for one of its two children. This allows us to reuse the representative light's material, geometric and visibility terms when computing that child. We use a heap data structure to efficiently find the cluster node in the cut with the highest error bound. If present in the cut, individual lights (i.e. light tree leaf nodes) are computed exactly and thus have zero error.

Our relative error criterion overestimates the visibility of errors in very dark regions. For example, a fully occluded point would be allowed zero error, but even at black pixels sufficiently small errors are not visible. Therefore, we also set a maximum cut size and, if the total number of nodes on the cut reaches this limit, stop further refinement. We chose our maximum cut size of 1000 to be large enough to rarely be reached in our results and then only in dark regions where the extra error is not visible.

4 Implementing Lightcuts

Our implementation supports three types of point lights: omni, oriented, and directional. Omni lights shine equally in all directions from a single point. Oriented lights emit in a cosine-weighted hemispherical pattern defined by their orientation, or direction of maximum emission. Directional lights simulate an infinitely far away source emitting in a single direction. All lights have an intensity I_i.

Building the Light Tree. The light tree groups point lights together into clusters. Ideally, we want to maximize the quality of the clusters it creates (i.e. combine lights with the greatest similarity in their material, geometric and visibility terms). We approximate this by grouping lights based on spatial proximity and similar orientation.

We divide the point lights by type into separate omni, oriented, and directional lists and build a tree for each. Conceptually though, we think of them as part of a single larger tree. Each cluster records its two children, its representative light, its total intensity $I_{\mathbb{C}}$, an axis-aligned bounding box, and an orientation bounding cone. The cone is only needed for oriented lights. Although infinitely far away, directional lights are treated as points on the unit sphere when computing their bounding boxes. This allows directional lights to use the same techniques as other point lights when building light trees and, more importantly, later for bounding their material terms M_i.

Similarity Metric. Each tree is built using a greedy, bottom-up approach by progressively combining pairs of lights and/or clusters. At each step we choose the pair that will create the smallest cluster according to our cluster size metric $I_{\mathbb{C}}(\alpha_{\mathbb{C}}^2 + c^2(1-\cos\beta_{\mathbb{C}})^2)$, where $\alpha_{\mathbb{C}}$ is the diagonal length of the cluster bounding box and $\beta_{\mathbb{C}}$ is the half-angle of its bounding cone. The constant c controls the relative scaling between spatial and directional similarity. It is set to the diagonal of the scene's bounding box for oriented lights and zero for omni and directional lights.

The representative light for a cluster is always the same as for one of its children and is chosen randomly based on the relative intensities of the children. Each individual light is its own representative. Thus the probability of a light being the representative for a cluster is proportional to its intensity. This makes the cluster approximation in Equation 2 unbiased in a Monte Carlo sense. However once chosen, the same representative light is used for that cluster over the entire image. Tree building, by its very nature, cannot be sublinear in the number of lights, but is generally not a significant cost since it only has to be done once per image (or less if the lights are static).

4.1 Bounding Cluster Error

To use the lightcuts approach, we need to compute reasonably cheap and tight upper bounds on the cluster errors (i.e. the difference between the exact and approximate versions of Equation 2). By computing upper bounds on the material, geometric, and visibility terms for a cluster, we can multiply these bounds with the cluster intensity to get an upper bound for both the exact and approximated cluster results. Since both are positive, this is also an upper bound on the cluster error (i.e. their absolute difference).

Visibility Term. The visibility of a point light is typically zero or one but may be fractional (e.g., if semitransparent surfaces are allowed). Conservatively bounding visibility in arbitrary scenes is a hard problem, so we will use the trivial upper bound of one for the visibility term (i.e. all lights are potentially visible).

Geometric Term. The geometric terms for our three point light types are listed below, where $\mathbf{y}_i$ is the light's position and ϕ_i is the angle between an oriented light's direction of maximum emission and direction to the point $\mathbf{x}$ to be shaded.

Light Type	Omni	Oriented	Directional
$G_i(\mathbf{x}) =$	$\frac{1}{\|\mathbf{y}_i - \mathbf{x}\|^2}$	$\frac{\max(\cos\phi_i, 0)}{\|\mathbf{y}_i - \mathbf{x}\|^2}$	1

(3)

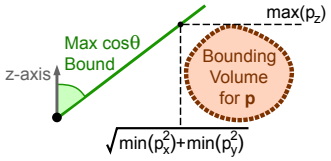


Figure 4: Bounding the minimum angle (and hence maximum cosine) to a bounding volume. See Equation 4.

The upper bound for directional lights is trivial since their geometric factor is always one. Omni lights are also easy. We just compute the minimum distance between the point **x** and the bounding volume of the cluster. Oriented lights are more complex because of the additional cosine factor. We could use the trivial cosine upper bound of one, but we prefer a tighter bound.

Let's start with the simpler problem shown in Figure 4. For any point $\mathbf{p} = [p_x, p_y, p_z]$, let θ be the angle between the vector from the origin to **p** and the z-axis. We want to find an upper bound on $\cos\theta$ over all points in some bounding volume. For any point **p**, we have $\cos\theta = \frac{p_z}{\sqrt{p_x^2+p_y^2+p_z^2}}$. To create an upper bound, we first replace the numerator by the maximum value of p_z within the bounding volume and then choose the p_x and p_y values to minimize or maximize the denominator depending on the sign of the numerator to get[1]:

$$\cos\theta \leq \begin{cases} \frac{\max(p_z)}{\sqrt{\min(p_x^2)+\min(p_y^2)+(\max(p_z))^2}} & \text{if } \max(p_z) \geq 0 \\ \frac{\max(p_z)}{\sqrt{\max(p_x^2)+\max(p_y^2)+(\max(p_z))^2}} & \text{otherwise} \end{cases} \quad (4)$$

To apply this to bounding $\cos\phi_i$ for oriented lights, we transform the problem as illustrated in Figure 5. First consider every point pair [**x**, $\mathbf{y}_i$] in the cluster and translate both points by $-\mathbf{y}_i$. This translates all the lights to the origin but spreads the point **x** across a volume with the same size as the cluster's bounding volume (the bounding volume's shape is the same but inverted). Second apply a coordinate transform that rotates the z-axis to match the axis of the cluster's orientation bounding cone. Now we can use Equation 4 to compute the minimum angle between the volume and the cone's axis. If this angle lies inside the bounding cone then we can only use the trivial upper bound of one, but if it lies outside then ϕ_i must be at least as large as this angle minus the cone's half-angle.

Material Term. The material term M_i is equal to the BRDF (Bidirectional Reflectance Distribution Function) times the cosine of the angle between the vector, $\mathbf{y}_i - \mathbf{x}$, and the surface normal at **x**. We have already described bounding the cosine of the angle to a bounding volume (Figure 4), so that only leaves the BRDF to be bounded.

Our current system supports three types of BRDF components: diffuse or lambertian, Phong [Phong 1975], and isotropic Ward [Larson 1992]. Multiple components can be summed when creating a BRDF. A diffuse BRDF component is simply a constant (i.e. does not depend on viewing or illumination directions) and so is trivial to bound. Phong components vary with the cosine of the angle between the vector to the light and the mirror reflection direction, raised to some exponent. We reuse the cosine bounding machinery already described to bound any Phong components. This same approach can be adapted to any similar BRDF component that is symmetric about an axis.

The isotropic Ward BRDF is not symmetric about any axis, because it is based on the half-angle (i.e. the angle between the surface normal and a vector halfway between the viewing and lighting vectors).

[1]Note if p_x ranges from -2 to 1 then $\max(p_x) = 1$ and $(\max(p_x))^2 = 1$ but $\min(p_x^2) = 0$ and $\max(p_x^2) = 4$.

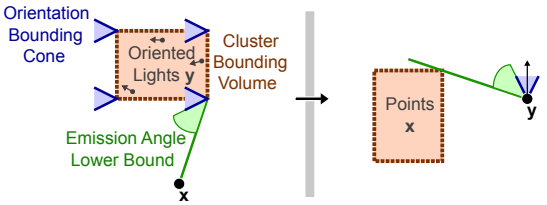


Figure 5: Conceptual transformation for bounding the emission angle (green) with respect to a cluster of oriented lights.

We have also developed a reasonably cheap and tight way to bound the minimum half-angle to a cluster. Details are in [Walter 2005]. We tested all our bounds numerically to confirm that they are valid.

In principle, lightcuts can work with any BRDF, as long as there is a good method to bound its maximum value over a cluster. Delta components (e.g., a mirror) are handled by the standard ray tracing method of recursively tracing reflected and/or refracted rays.

5 More Illumination Applications

Once we have a scalable solution for robustly approximating the illumination from large numbers of point lights, we can apply it to other difficult problems. The examples included here are illumination from: area lights, high dynamic range environment maps, and indirect illumination. Integrating different illumination types in the lightcuts framework has many advantages. For example, lightcuts automatically reduce the accuracy of one component when the error will be masked by strong illumination from another component.

Area Lights. Illumination and soft shadows from area lights are difficult to compute, and standard techniques often scale poorly with the size and number of area lights. A common approach is to approximate each area light using multiple point lights, however, the number required varies considerably depending on the local configuration. Locations near area lights or in their penumbra require many point lights while others require few. Many heuristics have been proposed (e.g., subdivide down to some fixed solid angle), but these may not work everywhere and often require manual parameter tweaking. In our system, each light can be converted into a conservatively large number of points. The lightcut algorithm will automatically and adaptively choose the number of samples to actually use locally. Any diffusely-emitting area light can be approximated by oriented lights on its surface. For spherical lights, an even better method using omni lights is described in Appendix A.

HDR Environment Maps. High dynamic range environment maps are a popular way to capture illumination from real world environments and apply it to synthetic scenes. Computing accurate illumination, especially shadows, from them can be very expensive.

The most common approach is to convert the environment map into a discrete set of directional lights. One critical question is how many directional lights to use. Using too few causes image artifacts such as blocky shadows, while using too many increases rendering costs substantially. For example, [Agarwal et al. 2003] suggest using 300 directional lights as generally adequate. In our experience, 300 is sufficient for isolated objects, but rendering complete scenes with significant occlusion and/or narrow glossy BRDFs may require 3000 or more. The lightcuts approach can handle such large numbers of lights much more efficiently than previous approaches.

Indirect Illumination. Indirect illumination (also called global illumination) is desirable for its image quality and sense of realism, but is considered too expensive for many applications. Much research has gone into increasing their physical and mathematical ac-


Lightcut Image


Reference Image


Error Image

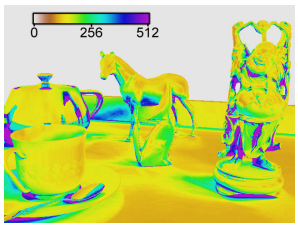

16 x Error Image

Point Lights	Avg Cut Size	Avg Shadow Rays	Time	Reference
4608	264	259	128s	1096s

Figure 6: Kitchen scene with direct light from 72 area sources.


Lightcut Image


Reference Image



Cut Size (False Color)


16 x Error Image

Point Lights	Avg Cut Size	Avg Shadow Rays	Time	Reference
3000	(187) 132	(168) 118	54s	424s

Figure 7: Tableau scene illuminated by an HDR environment map. In parentheses are averages over only pixels containing geometry.

curacy, but there is also interest in lower cost approximations with the tradeoff of lower fidelity, as long as they do not introduce objectionable image artifacts (e.g., [Tabellion and Lamorlette 2004]).

Instant radiosity is one such method. It first tracks light particles as they probabilistically scatter through a scene. Then virtual point lights are created at these scatter locations such that their aggregate illumination simulates the effect of indirect illumination. There is a cost vs. quality tradeoff in choosing the number of virtual lights. Using more lights reproduces more detail in the indirect illumination, but also increases cost of evaluating so many lights. Prior results have mostly been limited to tens or at most hundreds of virtual lights (e.g., [Keller 1997; Wald et al. 2002]). With our scalable algorithm, we can use thousands or even millions of virtual lights to reproduce more detail in the indirect illumination.

Prior instant radiosity results were limited to diffuse-only indirect because of the small number of virtual indirect lights used. Freed from this restriction, our system can also include some glossy indirect effects. Only a BRDF's diffuse component is used when converting particle hits to virtual oriented point lights, but unlike prior systems, we use the full BRDF at the points they illuminate.

When using instant radiosity, one needs to be aware of its inherent limitations. It cannot reproduce some types of indirect illumination (e.g., caustics); other methods must be used if these are desired. It also has difficulties with short range and/or glossy indirect effects though using more virtual lights helps. The particle tracing is a stochastic process and thus there is considerable randomness in the positions of the virtual lights. The contribution of any particular light is largely noise, and it is only by combining the results over many lights that we get statistically reliable information. Without a noise suppression technique, this noise would be highly visible in locations whose illumination is easily dominated by one or a few lights (e.g., concave corners or sharp glossy reflections)

The noise suppression is typically accomplished by limiting the maximum contribution from a virtual light. This biases the results but is preferable to objectionable noise artifacts. [Keller 1997] used hardware that clamped values to a maximum of 255 and [Wald et al. 2002] had the user specify a minimum distance to use when computing the geometric factor of an indirect light. The latter does not work with non-diffuse BRDFs, so we apply a clamping threshold to the total contribution, $M_i G_i V_i I_i$, of individual indirect lights. We do not clamp the contribution of clusters since they will be handled by normal lightcut refinement if too bright. This clamping threshold can be user-specified[2] or automatically determined as follows.

Our automatic clamping limits each indirect light to contribute no more than a fixed percentage of the total result. Since this clamping threshold is computed from the same noisy particle data, we must take care that it doesn't accentuate the noise. We use half the error ratio parameter from the lightcut selection/refinement process (e.g., 1% if the error ratio is 2%). During lightcut refinement, we keep track of any indirect nodes on the cut that may require clamping. In our experience, automatic clamping requires negligible overhead, works at least as well as a manually-tuned clamping threshold, and is one less parameter for the user to worry about.

6 Lightcut Results

In this section we demonstrate the results of applying our lightcuts implementation to five scenes with varying illumination. All results use an error ratio of 2% and a maximum cut size of 1000 nodes. All images in this section have a resolution of 640x480 with one eye ray per pixel, however this sometimes requires shading multiple points due to reflections and transparency. Timings are for a single workstation with two 3 GHz Xeon processors, though our system is also capable of efficiently using multiple machines.

Error. Lightcuts conservatively bound the maximum error per cluster to ensure that individual cluster refinement transitions are not visible. Theoretically, the errors from many different clusters could still sum up to a large and visible error. In practice we have not found this to be a problem for two reasons. The per cluster error bounds are worst case bounds; the actual average cluster error is much less than our 2% bound. Also uncorrelated errors accumulate much more slowly than correlated errors would (i.e. $O(\sqrt{N})$ vs. $O(N)$). During tree building, we randomize the choice of representative lights specifically to ensure that the cluster errors will be

[2] $\frac{1}{1000}$ of the image white point is a reasonable starting point.


Kitchen


Tableau


Grand Central


Temple

Model	Polygons	Number of Point Lights				Per Pixel Averages		Tree	Image
		Direct	Env Map	Indirect	Total	(d+e+i) Lightcut Size	Shadow Rays	Build	Time
Kitchen	388552	(72) 4608	5064	50000	59672	(54+244+345) 643	(**0.8**%) 478	3.4s	290s
Tableau	630843	0	3000	10000	13000	(0+112+104) 216	(**1.1**%) 145	0.9s	79s
Grand Central	1468407	(820) 38400	5064	100000	143464	(73+136+480) 689	(**0.33**%) 475	9.7s	409s
Temple	2124003	0	5064	500000	505064	(0+185+437) 622	(**0.07**%) 373	44s	225s
Bigscreen	628046	(4) 614528	0	25000	639528	(83+0+222) 305	(**0.04**%) 228	46s	98s

Figure 9: Lightcut results for 640x480 images of our five scenes with all illumination components enabled. Average cut sizes include how many came from each of the three components: direct, environment map, and indirect. Average shadow rays per pixel is also shown as a percentage of the total number of lights. The aliasing (e.g., windows in Grand Central) is due to using only one eye ray per pixel here.

statistically uncorrelated. Essentially, lightcuts provide a stochastic error bound rather than an absolute one; large total errors are possible but very unlikely.

The kitchen model, shown in Figure 6, is based on part of an actual house. It contains 72 area sources, each approximated using 64 point lights for a total of 4608 point lights. Even a close examination reveals no visible differences between the lightcut result and a reference image that evaluated each point light exactly. The error image appears nearly black. Magnifying the errors by a factor 16 shows that, as expected, the errors are generally larger in brighter regions and consist of many discontinuities caused by transitions between using a cluster and refining it. The lightcut image took 128 seconds with an average cut size of 264 and 259 shadow rays per pixel, while the reference image took much longer at 1096 seconds with an average of 3198 shadow rays per pixel. Shadow rays are not shot to lights whose material or geometric terms are zero.

The tableau model, in Figure 7, has several objects with different glossy materials on a wooden tray and lit by a captured HDR environment map (the Kitchen map from [Debevec 1998], not related to our kitchen model). We used the technique of [Agarwal et al. 2003] to convert this map to 3000 directional lights for rendering. Again there are no visible differences between the reference image and the lightcut image. A visualization shows the per pixel cut size (i.e. the number of nodes on the lightcut). Cut size corresponds closely to rendering cost and tends to be largest in regions of high occlusion.

Scalability. As the number of point lights increases, lightcuts scale fundamentally better (i.e. sublinearly vs. linearly) than prior techniques. To demonstrate this, we varied the number of point lights in the kitchen and tableau examples above by changing the number of point lights created per area light and directional lights created from the environment map, respectively. Image times vs. number of point lights are shown in Figure 8 for both lightcuts and the naive (reference) solutions. We also compare to [Ward 1994] which we consider the best of the alternative approaches with respect to generality, robustness, and the ability to handle thousands of lights.

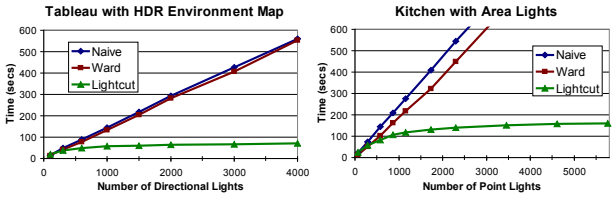


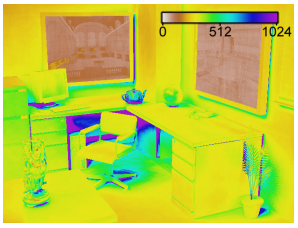
Figure 8: Lightcut performance scales fundamentally better (i.e. sublinearly) as the number of point lights increase.

Ward's technique computes the potential contribution of all the lights assuming full visibility, sorts these in decreasing order, and then progressively evaluates their visibility until the total remaining potential contribution falls below a fraction of the total of the evaluated lights (10% in our comparison). Since shadow rays (i.e.visibility tests) are usually the dominant cost, reducing them usually more than compensates for the cost of the sort (true for the kitchen and just barely for tableau). However, the cost still behaves linearly as shown in plots. Suppose we had 1000 lights whose unoccluded contributions would be roughly equal. We would have to check the visibility of at least 800 lights to meet a 10% total error bound[3], and this number grows linearly with the lights. This case is the Achilles' heel of Ward's technique, or any technique that provides an absolute error bound. Lightcuts superior scalability means that its advantage grows rapidly as the number of lights increase.

Mixed Illumination. We can use this scalability to compute richer and more complex lighting in our scenes as shown in Figure 9. Grand Central is a model of the famous landmark in New York City and contains 220 omni point lights distributed near the ceiling of the main hall and 600 spherical lights in chandeliers in the side hallways. The real building actually contains even more lights. Temple is a model of an Egyptian temple and is our most geometrically complex at 2.1 million triangles. Such geometric complexity

[3]Using the optimal value of 0.5 for the visibility of the untested lights.


Bigscreen Model



Cut Size (False Color)

Figure 10: Bigscreen model. See Figure 9 for statistics.

causes aliasing problems because we are using only one eye ray per pixel. The next section will introduce reconstruction cuts that allow us to perform anti-aliasing at a reduced cost. The bigscreen model (Figure 10) shows an office lit by two overhead area lights (64 points each) and by two large HDR monitors displaying images of our other scenes. The monitors have a resolution of 640x480 and were simulated by converting each of their pixels into a point light.

We've added a sun/sky model from [Preetham et al. 1999], which acts like an HDR environment map, to the kitchen, Grand Central, and temple. The sun is converted into 64 directional lights and the sky into 5000 directional lights distributed, uniformly over the sphere of directions. We also added indirect illumination to all the models using the instant radiosity approach. In the kitchen and Grand Central models only a fraction of particles from the sun/sky make it through the windows into the interior; many indirect lights end up on the outside of the buildings. With our scalable algorithm, we can compensate by generating more indirect lights. Temple uses the most indirect lights because its model covers the largest area.

With lightcuts the number of lights is not strongly correlated with image cost. Instead the strongest predictor of image cost is the degree of occlusion of the light sources. Thus the kitchen and particularly Grand Central are the most expensive because of the high degree of occlusion to their light sources especially the sun/sky. Although temple and bigscreen have more point lights and geometry, they are less expensive due to their higher visibility.

Shadow rays are the dominant cost in lightcuts and consume roughly 50% of the total time, even though this is the most optimized part of our code (written in C while the rest is Java). Computing error bounds consumes another 20% and 10% is shading. Various smaller operations including tree traversal, maintaining the max heap, updating estimates, etc. consume the rest. Tree building starts to be significant for the largest numbers of lights but there are much faster methods than our simple greedy approach.

Bound Tightness. The ability to cheaply bound the maximum contribution, and hence the error, from a cluster is essential for our scalability. Using tighter (and probably more expensive) bounds might reduce the average cut size and potentially reduce overall costs. In Figure 11, we show the cut sizes that would result if we had exact bounds on the product of the geometric and material terms $G_i M_i$. This is the best we can hope to achieve without bounds on visibility. Overall, our bounds perform very well. Indirect lights are harder to bound due to their wider dispersal, but our bounds still perform well. Significant further gains will likely require efficient conservative visibility bounds. This is possible in many specific cases (e.g., see [Cohen-Or et al. 2003]), but an unsolved problem in general.

Model	Illumination	Point	Avg Cut Size	
		Lights	Lightcut	Exact $G_i M_i$ Bound
Kitchen	Direct	4608	264	261
	+ Indirect	54608	643	497
Tableau	Env Map	3000	132	120
	+ Indirect	13000	216	153
Temple	Sun/Sky	5064	294	287

Figure 11: How tighter bounds would affect lightcut size.

7 Reconstruction Cuts

Reconstruction cuts are a new technique for exploiting spatial coherence to reduce average shading costs. The idea is to compute lightcuts sparsely over the image (e.g., at the corners of image blocks) and then appropriately interpolate their illumination information to shade the rest of the image. Although lightcuts allow the scalable computation of accurate illumination from thousands or millions of point lights, they still typically require shooting hundreds of shadow rays per shaded point. By taking a slightly less conservative approach, reconstruction cuts are able to exploit illumination smoothness to greatly reduce the shading cost.

The simplest approach would be to simply interpolate the radiances from the sparse lightcuts (e.g., Gouraud shading), but this would cause objectionable blurring of high frequency image features such as shadow boundaries and glossy highlights. Many extensions of this basic idea have been proposed to preserve particular types of features (e.g., [Ward and Heckbert 1992] interpolates irradiance to preserve diffuse textures and [Křivánek et al. 2005] use multiple directional coefficients to preserve low frequency gloss), but we want to preserve all features including sharp shadow boundaries.

To compute a reconstruction cut at a point, we first need a set of nearby samples (i.e. locations where lightcuts have been computed and processed for easy interpolation). Then we perform a top-down traversal of the global light tree. If all the samples agree that a node is occluded then we immediately discard it. If a node's illumination is very similar across the samples then we cheaply interpolate it using impostor lights. These are special directional lights designed to mimic the aggregate behavior of a cluster as recorded in the samples. Otherwise we default to lightcut-style behavior; refining down the tree until the errors will be small enough and using the cluster approximation from Equation 2, including shooting shadow rays.

Interpolating or discarding nodes, especially if high up in the tree, provides great cost savings. However when the samples straddle a shadow boundary or other sharp feature, we revert to more robust but expensive methods for the affected nodes. Because reconstruction cuts effectively include visibility culling, they are less affected by high occlusion scenes than lightcuts are.

Samples. A sample k is created by first computing a lightcut for a point $\mathbf{x}^k$ and viewing direction ω^k. This will include computing a radiance estimate $\tilde{L}_n^k$ at every light tree node n on the lightcut using Equation 2. For each node above the cut, we define $\tilde{L}_n^k$ as the sum of the radiance estimates of all of its descendants on the cut. Similarly, the total radiance estimate $\tilde{L}_{\mathrm{T}}^k$ is the sum over all nodes in the cut.

To convert a lightcut into a sample, we create *impostor directional point lights* (with direction $\mathbf{d}_n^k$ and intensity γ_n^k) for each node on or above the cut. Their direction mimics the average direction of incident light from the corresponding cluster or light, and their illumination exactly reproduces the radiance estimate $\tilde{L}_n^k$ at the sample point. Impostor lights are never occluded and hence relatively inexpensive to use (i.e. both visibility and geometric terms equal one).

For nodes on the cut, the impostor direction $\mathbf{d}_n^k$ is the same as from its representative light. For nodes above the cut, $\mathbf{d}_n^k$ is the average of the directions for its descendants on the cut, weighted by their respective radiance estimates. Using this direction we can evaluate the impostor's material term M_n^k and its intensity γ_n^k. We also compute a second intensity Γ_n^k based on the total radiance estimate and used to compute relative thresholds (i.e. relative magnitude of the node compared to the total illumination).

$$\gamma_n^k = \tilde{L}_n^k(\mathbf{x}^k, \omega^k) \,/\, M_n^k(\mathbf{x}^k, \omega^k) \tag{5}$$

$$\Gamma_n^k = \tilde{L}_{\mathrm{T}}^k(\mathbf{x}^k, \omega^k) \,/\, M_n^k(\mathbf{x}^k, \omega^k) \tag{6}$$

Occasionally we need an impostor directional light for a node below a sample's lightcut. These are created as needed and use the same direction as their ancestor on the cut, but with impostor's intensity diminished by the ratio between the node's cluster intensity $I_{\mathbb{C}}$ and that of its ancestor.

Exact sample size depends on the lightcut, but 24KB is typical. Samples store 32 bytes per node on or above the lightcut which includes 7 floats (1 for Γ_n^k and 3 each for $\mathbf{d}_n^k$ and γ_n^k) and an offset to its children's data if present. We decompose the image into blocks so that only a small subset of the samples is kept at any time.

Computing a Reconstruction Cut. Given a set of samples, we want to use them to quickly estimate Equation 1. Reconstruction cuts use a top-down traversal of the global light tree. At each node visited, we compute the minimum and maximum impostor intensities γ_n^k over the samples k, and a threshold τ_n, the lightcut error ratio (e.g.,2%) times the minimum of Γ_n^k over the samples. Then we select the first applicable rule from:

1. **Discard.** If $\max(\gamma_n^k) = 0$, then the node had zero radiance at all the samples and is assumed to have zero radiance here as well.
2. **Interpolate.** If $\max(\gamma_n^k) - \min(\gamma_n^k) < \tau_n$ and $\min(\gamma_n^k) > 0$, then we compute weighted averages of $\mathbf{d}_n^k$ and γ_n^k to create an interpolated impostor directional light. The estimate for this node is then equal to the material term for the interpolated direction times the interpolated intensity.
3. **Cluster Evaluate.** If $\max(\gamma_n^k) < \tau_n$ or if we are at a leaf node (i.e. individual light), then we estimate the radiance using Equation 2. This includes shooting a shadow ray to the representative light if the material and geometric terms are not zero.
4. **Refine.** Otherwise we recurse down the tree and perform the same tests on each of this node's two children.

While the above rules cover most cases, a few refinements are needed to prevent occasional artifacts (e.g., on glossy materials). First, we disallow interpolation inside glossy highlights. The maximum possible value for a diffuse BRDF is $1/\pi$. When computing the material term for an interpolated impostor, if its BRDF value is greater than $1/\pi$, then the direction must lie inside the gloss lobe of a material and we disallow interpolation for that node. Second, cluster evaluation is only allowed for nodes that are at, or below, at least one of the sample lightcuts. Nodes below all the sample lightcuts do not use interpolation, since they have no good information to interpolate. Third, if the result of a cluster evaluation is much larger than expected, we recursively evaluate its children instead.

Image Blocks. The image is first divided into 16x16 pixel blocks which do not share sample information. This keeps storage requirements low and allows easy parallel processing. To process each block, we initially divide it into 4x4 pixel blocks but may divide it further based on the following tests.

To compute a block, we compute samples at its corners, shoot eye rays through its pixels, and test to see if the resulting points match the corner samples. If using anti-aliasing, there will be multiple eye rays per pixel. A set of eye rays is said to match if they all hit surfaces with the same type of material and with surface normals that differ by no more than some angle (e.g., 30 degrees). We also use a cone test to detect possible local shadowing conditions that require block subdivision (e.g., see Figure 12). For each eye ray, we construct a cone starting at its intersection point and centered around the local surface normal. If the intersection point for any other eye ray lies within this cone, then the block fails the cone test. The cone test uses true geometric normals and is unaffected by shading effects such as bump maps or interpolated normals.

If the eye rays do not match and the block is bigger than a pixel, then we split the block into four smaller blocks and try again. If the block is pixel-sized, we relax the requirements, omit the cone test, and only require that each eye ray match at least two nearby samples. If there are still not enough matching samples, then we compute a new sample at that eye ray.

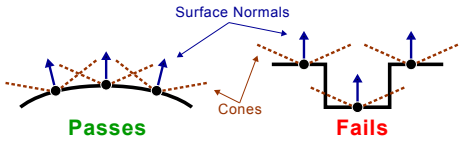


Figure 12: Block cone test. (Right) fails because the upper two points lie within the cone of the lower point. (Left) passes because no points lie within the cones defined by the other points.

After block subdivision we compute a color for each remaining eye ray using reconstruction cuts. For blocks larger than a pixel we use the four corners as the set of nearby samples and use image-space bilinear interpolation weights when interpolating impostors. Eye rays within pixel-sized blocks use their set of matching samples and interpolation weights proportional to the world-space inverse distance squared between the surface point and the sample points.

8 Reconstruction Cut Results

In this section we present some results of applying the reconstruction cuts technique to our scenes. Result images and statistics are shown in Figure 13. These results are adaptively anti-aliased [Painter and Sloan 1989] using between 5 and 50 eye rays per pixel. The sparse samples are computed using lightcuts with the same parameters as in Section 6. Most of the shading is done using reconstruction cuts that interpolate intelligently between the samples.

By exploiting spatial coherence, reconstruction cuts can shade points using far fewer shadow rays than lightcuts. While a lightcut requires a few hundred shadow rays, on average a reconstruction cut uses less than fourteen in our results. In fact, most of our shadow rays are used for computing the sparse samples (lightcuts) even though there are 15-25 times more reconstruction cuts. This allows us to generate much higher quality images, with anti-aliasing, at a much lower cost than with lightcuts alone. For the same size images, the results are both higher quality and have similar or lower cost than those in Section 6. Moreover for larger images, rendering cost increases more slowly than the number of pixels.

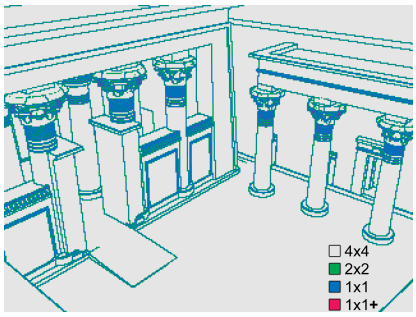
Samples are computed at the corners of adaptively sized image blocks (between 4x4 and 1x1 pixels in size) and occasionally within a pixel when needed (shown as 1x1+ in red). As shown for the temple image, most of the pixels in all images lie within 4x4 blocks. An average of less than one sample per pixel is needed even with anti-aliasing requiring multiple eye rays per pixel. We could allow larger blocks (e.g., 8x8) but making the samples even sparser where they are already sparse has less benefit and can be counterproductive because reconstruction cut cost is strongly related to the similarity of the nearby samples.

Because the accuracy of reconstruction cuts relies on having nearby samples that span the local lighting conditions, there is a possibility of missing small features in between samples. While this does happen occasionally, it is rarely problematic. For example in a few places, the interpolation smooths over the grooves in the pillars of the temple, but the errors are quite small and we have found that people have great difficulty in noticing them. Future improvements in the block refinement rules could fix these errors.

A Metropolis solution and its magnified (5x) differences with our result are shown in Figure 14. Metropolis [Veach and Guibas 1997] is considered the best and fastest of the general purpose Monte Carlo solvers. Its result took 13 times longer to compute than ours and still contains some visible noise. The main differences are the noise in the Metropolis solution and some corner darkening in our

Kitchen

Tableau



Temple (Reconstruction Block Size)

Grand Central

Temple

Model	Point Lights	Per Pixel Averages		Avg Shadow Rays Per Pixel		Avg Per Reconstruction Cut		Image Time	
		Eye Rays	Samples	Samples	Recon Cuts	Shadow Rays	Interpolations	1280x960	640x480
Kitchen	59672	5.4	0.29	143	50	**9.1**	14.4	672s	257s
Tableau	13000	5.4	0.23	50	41	**10.6**	17.8	298s	111s
Grand Central	143464	6.9	0.46	225	93	**13.3**	11.5	1177s	454s
Temple	505064	5.5	0.25	91	52	**9.4**	6.0	511s	189s
Bigscreen	639528	5.3	0.25	64	24	**4.6**	15.0	260s	98s

Figure 13: Reconstruction cut results for 1280x960 images (except where noted) of our scenes. Images are anti-aliased using 5 to 50 eye rays per pixel adaptively. Samples (lightcuts from Figure 9) are computed sparsely and most shading is computed using reconstruction cuts to interpolate intelligently. Reconstruction cuts are much less expensive and require many fewer shadow rays than lightcuts. Bigscreen image is shown in Figure 1.

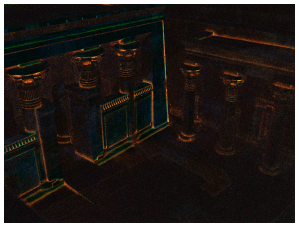
result. The latter is due to instant radiosity not reproducing very short range indirect illumination effects.

9 Conclusion

We have presented lightcuts as a new scalable unifying framework for illumination. The core component is a strongly sublinear algorithm for computing the illumination from thousands or millions of point lights using a perceptual error metric and conservative per cluster error bounds. We have shown that it can greatly reduce the number of shadow rays, and hence cost, needed to compute illumination from a variety of sources including area lights, HDR environment maps, sun/sky models, and indirect illumination. Moreover it can handle very complex scenes with detailed geometry and glossy materials. We have also presented reconstruction cuts that further speed shading by exploiting coherence in the illumination.

There are many ways in which this work can be improved and extended. For example, extending to additional illumination types (e.g., indirect components not handled by instant radiosity), integrating more types of light sources including non-diffuse sources and spot lights, developing bounds for more BRDF types, more formal analysis of lightcuts stochastic error, further refinement of the reconstruction cut rules to exploit more coherence, and adding conservative visibility bounds to further accelerate lightcuts.

Metropolis Solution



5 x Difference

Figure 14: Comparison with metropolis image for temple.

Acknowledgments

Many thanks to the modelers: Jeremiah Fairbanks (kitchen), Will Stokes (bigscreen), Moreno Piccolotto, Yasemin Kologlu, Anne Briggs, Dana Getman (Grand Central) and Veronica Sundstedt, Patrick Ledda, and the Graphics Group at University of Bristol (temple). This work was supported by NSF grant ACI-0205438 and Intel Corporation. The views expressed in this article are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U. S. Government.

References

AGARWAL, S., RAMAMOORTHI, R., BELONGIE, S., AND JENSEN, H. W. 2003. Structured importance sampling of environment maps. *ACM Transactions on Graphics 22*, 3 (July), 605–612.

BLACKWELL, H. R. 1972. Luminance difference thresholds. In *Handbook of Sensory Physiology*, vol. VII/4: Visual Psychophysics. Springer-Verlag, 78–101.

COHEN-OR, D., CHRYSANTHOU, Y. L., SILVA, C. T., AND DURAND, F. 2003. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics 9*, 3, 412–431.

DEBEVEC, P. 1998. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, 189–198.

FERNANDEZ, S., BALA, K., AND GREENBERG, D. P. 2002. Local illumination environments for direct lighting acceleration. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, 7–14.

HANRAHAN, P., SALZMAN, D., AND AUPPERLE, L. 1991. A rapid hierarchical radiosity algorithm. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, vol. 25, 197–206.

HASENFRATZ, J.-M., LAPIERRE, M., HOLZSCHUCH, N., AND SILLION, F. 2003. A survey of real-time soft shadows algorithms. In *Eurographics*, Eurographics, Eurographics. State-of-the-Art Report.

JENSEN, H. W. 2001. *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd.

KELLER, A. 1997. Instant radiosity. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, 49–56.

KOK, A. J. F., AND JANSEN, F. W. 1992. Adaptive sampling of area light sources in ray tracing including diffuse interreflection. *Computer Graphics Forum (Eurographics '92) 11*, 3 (Sept.), 289–298.

KOLLIG, T., AND KELLER, A. 2003. Efficient illumination by high dynamic range images. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, 45–51.

KŘIVÁNEK, J., GAUTRON, P., PATTANAIK, S., AND BOUATOUCH, K. 2005. Radiance caching for efficient global illumination computation. *IEEE Transactions on Visualization and Computer Graphics*.

LARSON, G. J. W. 1992. Measuring and modeling anisotropic reflection. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, vol. 26, 265–272.

PAINTER, J., AND SLOAN, K. 1989. Antialiased ray tracing by adaptive progressive refinement. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, vol. 23, 281–288.

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.

PHONG, B. T. 1975. Illumination for computer generated pictures. *Commun. ACM 18*, 6, 311–317.

PREETHAM, A. J., SHIRLEY, P. S., AND SMITS, B. E. 1999. A practical analytic model for daylight. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, 91–100.

SCHEEL, A., STAMMINGER, M., AND SEIDEL, H.-P. 2001. Thrifty final gather for radiosity. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 1–12.

SCHEEL, A., STAMMINGER, M., AND SEIDEL, H. 2002. Grid based final gather for radiosity on complex clustered scenes. *Computer Graphics Forum 21*, 3, 547–556.

SHIRLEY, P., WANG, C., AND ZIMMERMAN, K. 1996. Monte carlo techniques for direct lighting calculations. *ACM Transactions on Graphics 15*, 1 (Jan.), 1–36.

SILLION, F. X., AND PUECH, C. 1994. *Radiosity and Global Illumination*. Morgan Kaufmann Publishers Inc.

SMITS, B., ARVO, J., AND GREENBERG, D. 1994. A clustering algorithm for radiosity in complex environments. In *Proceedings of SIGGRAPH 94*, Annual Conference Series, 435–442.

TABELLION, E., AND LAMORLETTE, A. 2004. An approximate global illumination system for computer generated films. *ACM Transactions on Graphics 23*, 3 (Aug.), 469–476.

VEACH, E., AND GUIBAS, L. J. 1997. Metropolis light transport. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, 65–76.

WALD, I., KOLLIG, T., BENTHIN, C., KELLER, A., AND SLUSALLEK, P. 2002. Interactive global illumination using fast ray tracing. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, 15–24.

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Interactive global illumination in complex and highly occluded environments. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, 74–81.

WALTER, B., ALPPAY, G., LAFORTUNE, E. P. F., FERNANDEZ, S., AND GREENBERG, D. P. 1997. Fitting virtual lights for non-diffuse walkthroughs. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, 45–48.

WALTER, B. 2005. Notes on the Ward BRDF. Technical Report PCG-05-06, Cornell Program of Computer Graphics, Apr.

WARD, G. J., AND HECKBERT, P. 1992. Irradiance gradients. In *Third Eurographics Workshop on Rendering*, 85–98.

WARD, G. 1994. Adaptive shadow testing for ray tracing. In *Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering)*, Springer-Verlag, New York, 11–20.

WOO, A., POULIN, P., AND FOURNIER, A. 1990. A survey of shadow algorithms. *IEEE Computer Graphics and Applications 10*, 6 (Nov.), 13–32.

ZANINETTI, J., BOY, P., AND PEROCHE, B. 1999. An adaptive method for area light sources and daylight in ray tracing. *Computer Graphics Forum 18*, 3 (Sept.), 139–150.

A Spherical Lights

Any diffuse area light can be simulated using oriented lights on its surface. For spherical lights, we have developed a better technique that simulates them using omni lights (which are a better match for far field emission). Placing omni lights on the surface of the sphere would incorrectly make it appear too bright near its silhouette as compared to its center. Similarly a uniform distribution inside the volume of the sphere would exhibit the reverse problem. However, by choosing the right volume distribution inside the sphere, we can correctly match the emission of a spherical light.

$$d(\mathbf{x}) = \frac{1}{\pi^2 R^2 \sqrt{R^2 - r^2(\mathbf{x})}} \tag{7}$$

The normalized point distribution $d(\mathbf{x})$ is defined inside a sphere of radius R where $r^2(\mathbf{x})$ is the squared distance from the sphere's center. The beauty of this distribution is that it projects to a uniform distribution across the apparent solid angle of the sphere when viewed from any position outside the sphere, which is exactly the property we need. We also need a way to generate random points according to this distribution. Given three uniformly distributed random numbers, ξ_1, ξ_2, ξ_3, in the range $[0,1]$ we can compute points with the right distribution for a sphere centered at the origin using:

$$\begin{aligned} x &= R\sqrt{\xi_1}\cos(2\pi\xi_2) \\ y &= R\sqrt{\xi_1}\sin(2\pi\xi_2) \\ z &= \sqrt{R^2 - x^2 - y^2}\sin(\pi(\xi_3 - 1/2)) \end{aligned} \tag{8}$$

The omni lights generated from spherical lights behave exactly like normal omni lights except that when computing their visibility factors, they can only be occluded by geometry outside the sphere (i.e. their shadow rays terminate at the surface of the sphere).