



**EPS**

Escola Politècnica

Superior

Projecte/Treball Fi de Carrera

**Estudi:** Eng. Tècn. Informàtica de Sistemes. Pla 2001

**Títol:** Algorismes de shading moderns: implementació i comparativa

**Document:** Memòria del projecte

**Alumne:** Bernat Muñoz Garcia

**Director/Tutor:** Mateu Sbert Casasayas

**Departament:** Informàtica i Matemàtica Aplicada

**Àrea:** LSI

**Convocatòria (mes/any):** 09 / 2013

Universitat de Girona  
Escola Politècnica Superior

Projecte Final de Carrera

# **Algorismes de shading moderns: implementació i comparativa**

per

**Bernat Muñoz Garcia**

Tutor: Mateu Sbert Casasayas

Girona, 2010-2013

*Als meus pares*

---

# Índex

---

<b>Índex</b>	<b>i</b>
<b>1 Introducció</b>	<b>1</b>
1.1 Introducció . . . . .	1
1.2 Motivació . . . . .	4
1.3 Propòsit . . . . .	6
1.4 Objectius . . . . .	7
<b>2 Estudi de viabilitat</b>	<b>10</b>
<b>3 Metodologia</b>	<b>16</b>
<b>4 Planificació</b>	<b>20</b>
<b>5 Conceptes previs</b>	<b>24</b>
5.1 Conceptes bàsics . . . . .	24
5.2 Gràfics 3D . . . . .	25
5.3 Arquitectura pipeline gràfica . . . . .	26
5.3.1 Fase d'aplicació . . . . .	29
5.3.2 Fase de geometria . . . . .	29
5.3.3 Fase de rasterització . . . . .	33
5.4 GPU . . . . .	37
5.4.1 Fases programables . . . . .	38
5.4.2 Vertex shader . . . . .	40
5.4.3 Pixel shader . . . . .	40
5.5 Aparença visual . . . . .	41
5.5.1 Fonts de llums . . . . .	42
5.5.2 Materials . . . . .	44
5.5.3 Shading . . . . .	46
5.5.4 Equació de shading . . . . .	50



<b>6</b>	<b>Requisits del sistema</b>	<b>53</b>
6.1	Pre-requisits . . . . .	53
6.1.1	Càrrega de dades . . . . .	53
6.1.2	Sistema de càmera . . . . .	54
6.2	Implementació d'algorismes de shading . . . . .	55
6.3	Comparació d'algorismes de shading . . . . .	57
<b>7</b>	<b>Estudis i decisions</b>	<b>58</b>
7.1	Maquinari . . . . .	58
7.2	Llibreries . . . . .	59
7.2.1	Llibreries gràfiques . . . . .	59
7.2.2	Llibreries d'imatges . . . . .	60
7.2.3	Altres llibreries . . . . .	61
7.3	Programari . . . . .	62
<b>8</b>	<b>Anàlisi i disseny del sistema</b>	<b>64</b>
8.1	Anàlisi . . . . .	64
8.2	Disseny . . . . .	65
<b>9</b>	<b>Implementació i proves</b>	<b>69</b>
9.1	Implementació dels algorismes de shading . . . . .	69
9.1.1	Forward shading . . . . .	70
9.1.2	Deferred shading . . . . .	72
9.1.3	Deferred lighting . . . . .	74
9.1.4	Inferred lighting . . . . .	76
9.2	Implementació dels mòduls . . . . .	78
9.2.1	Mòdul de càrrega . . . . .	78
9.2.2	Motor 3D . . . . .	80
9.2.3	Sistema de mesura . . . . .	83
<b>10</b>	<b>Implantació i resultats</b>	<b>85</b>
10.1	Procés de desenvolupament . . . . .	85
10.1.1	Primer sistema de pintat . . . . .	85
10.1.2	Càrrega bàsica d'arxius Collada . . . . .	86
10.1.3	Abstracció de la llibreria gràfica . . . . .	87
10.1.4	Creació del sistema de dibuixat . . . . .	88
10.1.5	Sistema de càmera . . . . .	89
10.1.6	Determinació de visibilitat . . . . .	90
10.1.7	Implementació de la resta d'algorismes de shading . . . . .	91
10.1.8	Eines de rendiment i memòria . . . . .	92
10.1.9	Implementació de materials més complexes . . . . .	93
10.1.10	Sistema de comparatives . . . . .	95

10.2 Resultats . . . . .	96
10.2.1 Creació d'un motor 3D . . . . .	96
10.2.2 Utilitzar tecnologies desconegudes . . . . .	97
10.2.3 Implementar 4 algorismes de shading i comparar-los . . . . .	97
<b>11 Conclusions</b>	<b>100</b>
11.1 Assoliment dels requisits . . . . .	100
11.1.1 Càrrega de dades . . . . .	100
11.1.2 Sistema de càmera . . . . .	101
11.1.3 Implementació d'algorismes de shading . . . . .	101
11.1.4 Comparació d'algorismes de shading . . . . .	103
11.2 Conclusions del projecte . . . . .	103
11.2.1 Collada . . . . .	103
11.2.2 Direct3D 9 . . . . .	104
11.2.3 Algorismes de shading . . . . .	105
11.2.4 Conclusions generals . . . . .	106
<b>12 Treball futur</b>	<b>107</b>
<b>Bibliografia</b>	<b>109</b>
<b>Índex de figures</b>	<b>112</b>
<b>Manual d'instal·lació i usuari</b>	<b>115</b>

# Capítol 1

---

## Introducció

---

### 1.1 Introducció

La informàtica gràfica és mou a passos de gegant. Pot semblar una generalització pròpia de gent no avesada a aquest món, però és una realitat pels que treballen amb ordinadors o dispositius electrònics cada dia. Influeix en la forma de treballar: cal mantenir un equilibri constant entre desenvolupament de solucions per la tecnologia actual, i la investigació pel futur.

L'equilibri entre les tècniques aplicables, els requisits de qualitat/preu i les expectatives del client potencial, són les pedres angulars d'una indústria que avança a un ritme vertiginós. Tot i això, cal tenir sempre present en quins punts és adequat fer un salt tecnològic, i en quins punts el públic de consum no necessita un salt qualitatiu, sinó una petita evolució.



Figura 1.1: Exemple d'*uncanny valley* (esquerra) i estil no realista (dreta)

Els humans percebem amb molta facilitat qualsevol informació visual, i tenim una gran facilitat per reconèixer patrons. Aquestes dues propietats ens permeten reconèixer amb molta facilitat quan un detall és irregular o fora de lloc. És important aquest detall, degut a que, fins i tot persones sense coneixements tècnics, poden reconèixer que quelcom falla en una imatge sintètica. Fins i tot, hi ha un terme en anglès que descriu el fet que, propers al foto-realisme però amb algun detall no reproduït de forma fidedigna, és perd l'intent de realisme. Això és conegut com *uncanny valley*, i és un dels motius que a vegades és busqui un estil menys realista en els gràfics per ordinador, degut a que els humans poden ignorar el realisme de l'imatge, si l'estil artístic s'allunya d'aquest (Figura 1.1). D'aquesta manera, una vegada s'accepta la falta de fotorealisme de la imatge, l'espectador pot identificar-se més fàcilment amb el que veu, i per tant, acceptar el conjunt.

Un altre element que cal tenir en compte de la visió humana és la freqüència amb la que l'ull pot captar imatges diferents. De forma més clara, mostrades una successió d'imatges diferents en un segon, quantes podríem arribar a diferenciar? Aquest nombre no és exacte, ja que cada humà és capaç de diferenciar una quantitat lleugerament diferent, però, de forma general, es considera que unes 24 o 25 imatges per segon és l'estàndard. És important conèixer la xifra, ja que per successions d'imatges sota d'aquesta quantitat notaríem una falta de fluïdesa, i per damunt pot donar-se el cas de no veure alguna imatge (per exemple, els humans difícilment notem un flaix amb una freqüència de 40 Hz). En el cas del món del cinema, una major freqüència pot significar més quantitat de cinta en suports analògics, o més capacitat requerida en el cas de suports digitals. En el cas de videojocs, més freqüència significa que hem de fer més càlculs per segon, la qual cosa pot significar haver de reduir la complexitat de les escenes que mostrem, reduir la quantitat de llums o la fidelitat dels materials reproduïts.

Un cop remarcats aquests detalls, és notable saber que, en la majoria de gèneres, tant l'indústria del videojoc com del cine, persegueixen el foto-realisme, ja que pot significar una major implicació de l'espectador/jugador amb el que veu. Assolir més realisme implica un major treball, tant sigui des del punt de vista artístic, és a dir, quan temps necessiten els artistes per modelar el món/personatges a representar, i quan temps de procés necessitem per avaluar aquest món virtual. Aquest món virtual, no és més que una definició de materials, llums i geometria. És en aquest punt, on difereix de forma molt notable l'aproximació del món del cinema i dels videojocs.

En el cas del món del cinema, la major part de la investigació ha estat com reproduir, ni que sigui de forma aproximada, com és comporta la llum i els materials a la realitat. Un avantatge del cinema, és que es disposa de tot el temps de procés (dins dels límits de la plausibilitat, tant temporal com econòmica) per tal de calcular les interaccions entre llum i material, cosa que permet centrar-se,

com hem dit, en una reproducció acurada de la realitat, essent l'eficiència dels càlculs una qüestió amb *menys importància*. Per exemple, en aquesta indústria, no és estrany que completar tots els càlculs d'un únic fotograma trigui desenes de minuts o hores. Per exemple, alguns fotogrames de la coneguda pel·lícula 'Bichos' van trigar 72 hores en completar-se. Com que el cinema no és interactiu, això no suposa un problema, ja que la quantitat de fotogrames a dibuixar és acotada i no varia un cop finalitzada la producció.

En canvi, els videojocs són interactius. Com hem dit, els humans són capaços de diferenciar unes 25 imatges per segon. Llavors, hem d'actualitzar l'estat del joc en  $1/25$  segons, o el que és el mateix, 40 mili-segons. Durant aquests 40 mili-segons, hem de calcular la resposta a les accions del jugador (per exemple, movent-se), l'evolució del joc (per exemple, la intel·ligència artificial) i dibuixar un nou fotograma. Tot i que no detallem tots els processos, podem veure que tenim un marge d'acció relativament limitat, comparat amb les desenes de minuts o hores del cinema.

Tenint una idea general de la principal limitació dels videojocs, i només tenint en compte la interacció entre llums i materials, comunament anomenat *shading* en anglès, i afegint el fet que cada cop el públic demana entorns més dinàmics i complexos, i per tant el marge per al pre-càlcul d'elements estàtics és menor, necessitem trobar solucions, el més òptimes possible, per mostrar aquests entorns dinàmics i altament complexos.

Anem a centrar-nos en l'objectiu de l'actual treball: algorismes de shading moderns. Com hem dit, volem el màxim realisme possible, amb unes limitacions força clares. Un dels principals problemes que ha trobat l'indústria a l'hora de realitzar imatges realistes, és com emular el comportament dels materials i la seva interacció amb les llums.

Tenint en compte aquest dos elements, ens hem volgut centrar en com fan els sistemes actuals de shading per incrementar el nombre de llums en una escena, sense incrementar les necessitats de computació fora del que el mercat pot assolir. Podríem haver-nos centrat en la representació dels materials, però hem cregut que el problema de l'increment del nombre de llums és més interessant, i també és un problema més acotat que no la reproducció fidedigna de diferents tipus de materials (exemple a la [Figura 1.2](#)), com poden ser els materials metàl·lics, la pell humana o els transparents.

La necessitat d'incrementar el nombre de llums que podem suportar ve senzillament donat per les necessitats de crear escenaris més fidedignes. Idealment, per moltes escenes, amb la llum del sol en tindríem prou, però calcular com rebota la llum a les superfícies és un problema obert en el món dels gràfics en temps real, així que s'emula afegint llums als punts on la llum rebotaria. És una aproximació molt crua a la realitat, però és la única que permet la capacitat de càlcul amb la que disposem actualment pels motors 3D en temps real. Així doncs,

incrementar el nombre de fonts de llum és la única forma que tenim de modelar com es comporta la llum al món real.

## 1.2 Motivació

Unes de les coses positives del món del software (si més no, a nivell dels programadors gràfics), és el fet que es comparteix de forma força oberta el coneixement. A nivell general, no és difícil conèixer com treballa un motor gràfic d'avantguarda, o fins i tot compartir-ne codi, a vegades abans que el motor gràfic surti a la llum. Tot i que pot semblar una mala pràctica, en aquest àmbit és un costum. Només cal veure la quantitat de llibres, blocs o conferències que detallen com funcionen les tècniques desenvolupades per les companyies amb més renom, dels quals a la bibliografia en podem trobar una bona llista. Gràcies a que les tècniques utilitzades només són idees de com resoldre un determinat problema (per exemple, com fer una aproximació plausible a cert efecte del món real), i són una petita part d'un sistema molt més complexe, fa que això mai suposi un problema real, almenys a nivell de competitivitat entre diferents companyies. A efectes pràctics, compartir les idees no és problemàtic, ja que l'execució, després, és molt més complexa de portar a terme.

Tenint aquest coneixement, i portant diversos anys veient les solucions aplicades a cada companyia i a cada producte, se'n destil·la una idea força important i senzilla a la vegada: cal un gran i especialitzat coneixement, per tal de ser competitiu. No cal ser un expert en cada una de les tècniques utilitzades al mercat, però cal conèixer-les el suficient per saber on trobar recursos i com implementar-les. No només cal conèixer al detall el llenguatge utilitzat, si no que cal conèixer

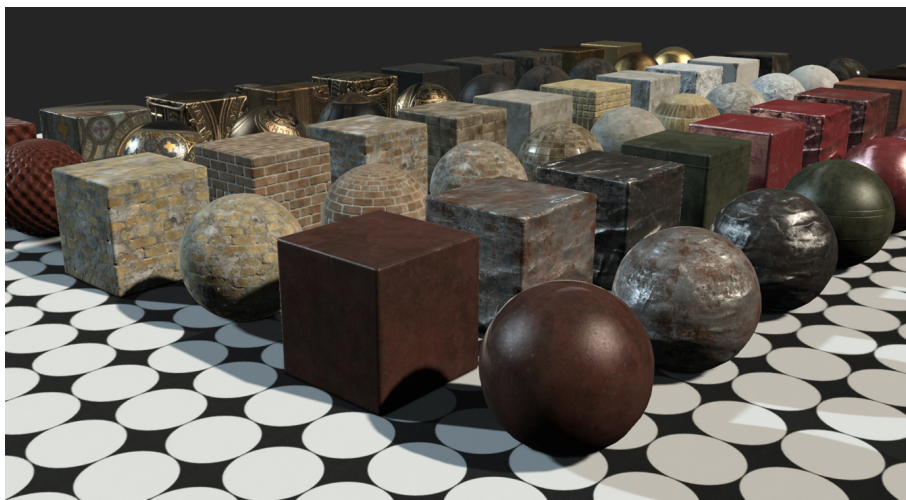


Figura 1.2: Exemple de reproduccions de diferents tipus de materials

molt bé el hardware sobre el qual executarem l'aplicació al detall, les (moltes) possibles tècniques aplicables, i tenir clares les limitacions de cada una. En una indústria que es renova constantment, és més difícil del que pot semblar.

Per sort, actualment, i gràcies a la xarxa, són molts els recursos a la mà de qualsevol programador, i l'accés a la informació és a l'abast de tothom. Això també provoca un petit problema, que és una de les idees base per la creació d'aquest treball: la informació no es troba centralitzada en un sol punt, sinó que és dispersa ens molts llocs, i a vegades la quantitat de recerca per tal d'estudiar una tècnica esdevé un procés llarg, que inclou part de buscar, part d'implementar, i part de prova-error. Una de les idees cabdals de l'actual treball és portar a terme aquesta recerca, i aportar, en un sol document, una comparativa de tècniques i tecnologies aplicades, junt amb una implementació que pugui rivalitzar amb una de comercial (dins de les possibilitats que un treball individual permet). A més, degut a la gran quantitat d'informació, la síntesi centralitzada que aporta el present treball, és molt important. Tenir una sola referència on consultar les tècniques és important, sobretot si també aporta una sèrie de referències externes amb les quals ampliar els coneixements més específics, si es desitja.

Centrant-nos en una orientació molt més personal i interessada de cara al futur, aquest treball esdevé un aparador de la meua capacitat tècnica i els meus coneixements, a part de poder també ser un sistema que em permetrà ser més conegut, ja que intenta ser una referència de les tècniques a implementar. També esdevé la meua petita aportació al món de la programació gràfica, del qual he agafat tantes coses, i he retornat tant poc.

Finalment, també hi ha una gran part d'experimentació. Vistes les motivacions prèvies del treball, és força clar que no és la meua primera aplicació gràfica, ni el meu primer gran projecte. Tant a nivell professional com personal, no és el primer motor 3D en el qual treballo, i algunes de les tècniques ja les conec prèviament. Per tal que l'actual treball sigui més interessant a nivell personal, i no una altra iteració en la implementació de tècniques que ja conec, he decidit desviar-me de les tecnologies que he utilitzat durant més d'una dècada. És possiblement el punt més perillós del treball, doncs m'endinso en terreny desconegut, que pot suposar certs problemes que hauré de resoldre.

Sempre mantenint-se dintre d'eines el més estàndard possibles, m'he desviat del meu mètode habitual de treball, per tal de que l'actual treball sigui un repte tant des de el punt de vista teòric (investigar les tècniques i comparar-les) com des del punt de vista tècnic (implementar-les amb tecnologies que desconec). Com en tot projecte interessant, el fet que hi hagi gran quantitat d'incerteses en el moment de començar, el fa perfecte per un enginyer: hem de treballar amb les possibilitats que ens dona el mercat, investigar aquestes possibilitats, cercant les que creiem més adequades, implementar el projecte, limitar l'abast del treball, analitzar quines decisions han estat correctes i quines no, i finalment, veure quines

noves possibilitats ofereix el mercat i si son interessants per futurs projectes.

### 1.3 Propòsit

El propòsit del projecte és divideix en dos parts, la primera, l'aprenentatge personal i la segona, crear implementacions que després la gent pugui consultar.

La primera, la part d'aprenentatge, perquè estic especialment interessat amb el que em pot aportar un projecte amb els objectius de l'actual. Essent un programador amb certa experiència, m'interessa allunyar-me de la meva zona de confort, per endinsar-me en camps on habitualment no m'endinsaria, tant sigui per limitacions de temps o altres. La idea és utilitzar la màxima quantitat de tecnologies amb les quals no estigui avesat. Als humans ens costa molt acceptar els canvis, per la qual cosa és més còmode seguir utilitzant les tecnologies que ja coneixem, a part de que no perdem la gran inversió d'hores que hem fet aprenent-les. Aprendre noves tecnologies i formats de dades és un dels grans propòsits del projecte, doncs m'aporta molt com enginyer.

També ofereix la oportunitat d'escriure un motor 3D des de zero. A nivell personal ja he tingut la oportunitat de fer-ho, però aquest ja porta en desenvolupament més de 6 anys i es compona de 50000 línies de codi, amb la qual cosa el marge per la experimentació es reduït. El fet de poder començar-ne un des de zero, em permet provar idees noves, que poden ser bones o dolentes, però que només aquest procés em pot permetre identificar. Com en qualsevol projecte llarg, s'han de prendre una serie de decisions inicials (que detallarem en capítols posteriors), que després poden ser o no correctes.

La segona, és que m'interessa crear quelcom que la gent pugui consultar en quan sigui acabat. Tot i que la documentació és important, crec que és cabdal donar al món implementacions funcionals d'allò documentat. Per desgràcia, és més fàcil trobar documentació dels sistemes que desenvoluparé en aquest projecte que no pas codi. En aquest punt vaig estar pensant en crear l'actual documentació en anglès, doncs el domino des de ben petit i no suposaria un esforç titànic, però al final vaig decidir crear-la en català, ni que el potencial public sigui menor, em dóna l'opció de referenciar-lo a gent amb menys coneixements d'anglès. Tot i això, el codi està documentat en anglès, ja que trobo incongruent fer-ho en català/castellà, doncs porto utilitzant l'anglès per escriure els comentaris des de fa 15 anys. A més, el codi és una eina igualment interessant a nivell de documentar, així que aquesta dualitat no hauria de ser problemàtica: aquelles persones que vulguin veure la implementació hauran de saber anglès per entendre els comentaris, però si no, sempre poden consultar la present memòria.



## 1.4 Objectius

En el moment de començar aquest projecte, portava 15 anys programant. Al cap dels anys, he entès que l'única manera de millorar i aprendre és plantejar-se reptes que et treguin de l'àrea de confort, i t'endinsin cap a zones desconegues.

L'objectiu d'aquest treball és desenvolupar un motor 3D senzill pels estàndards actuals, però ambicions per ser un projecte personal, que inclogués diversos algorismes de shading, algunes de les quals desconeixia totalment, i obligar-me a utilitzar tecnologies que desconeixia del tot.

Així, doncs, els meus objectius eren força clars:

- Desenvolupar un motor 3D des de zero, limitant l'abast per tal que sigui possible aconseguir-ho
- Utilitzar tecnologies que desconegui en el moment de començar
- Implementar quatre algorismes de shading i comparar-los

Anem a veure cada punt amb més detall.

Entenem com a motor 3D aquell que ens permeti dibuixar objectes 3D i mostrar-los en una pantalla de PC. Ha de tenir un sistema per moure el receptor o càmera, que és el punt des del que veiem els objectes, a desig de l'usuari. Les dades sobre els objectes que es dibuixaran poden crear-se directament utilitzant codi, però ens sembla molt més adequat utilitzar el sistema que és fa servir a les indústries relacionades, és a dir, que la descripció de l'escena i els elements que la componen provinguin d'arxius guardats en disc, de manera que un usuari sense coneixements de programació pugui visualitzar diferents escenes 3D. Existeixen molts motors 3D al mercat que podríem haver utilitzat, com per exemple **Unity** (Figura 1.3), que és gratuït per projectes d'aquest tipus. Hi ha diversos motius pels quals no els vaig utilitzar, però principalment, en el moment de començar aquest projecte, no oferien la flexibilitat necessària i anava en contra del meu propòsit d'experimentar amb noves maneres de dissenyar un motor 3D.

Quan em refereixo a utilitzar tecnologies desconegudes, vull centrar-me en l'elecció del format de les dades 3D i la llibreria gràfica utilitzada per comunicar-se amb el hardware. En el passat, pel format de dades, he treballat principalment amb formats propis, havent de crear, també, els programes necessaris per extreure les dades de les aplicacions de modelat 3D. Per aquest projecte, volia intentar utilitzar algun format que la indústria considerés estàndard, per treure conclusions de les seves possibilitats. A nivell de la llibreria gràfica per parlar amb el hardware, actualment hi ha dos eleccions a l'hora de desenvolupar motors 3D per a PC, i com ja en coneixia una, volia provar l'altra i veure els paral·lelismes, avantatges i desavantatges, ni que aquesta part no sigui important pel treball.

Finalment, la part més interessant del treball: algorismes de shading. En aquest cas seguim la tendència dels altres objectius, vull conèixer nous algorismes,

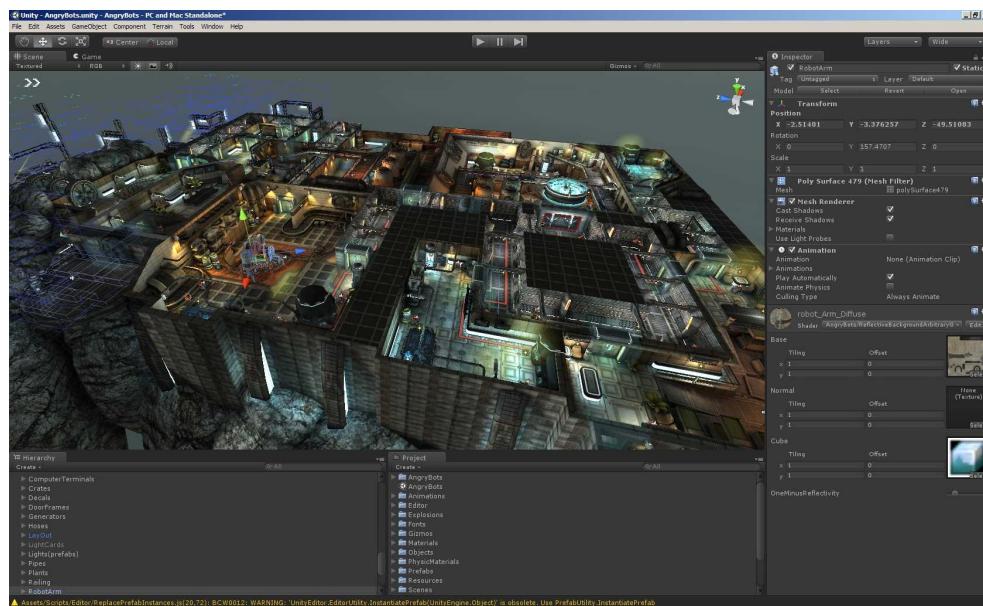


Figura 1.3: Editor del motor de jocs Unity

altres aproximacions per resoldre un problema tant obert com és el del shading en temps real. En el moment de començar el projecte, coneixia la meitat dels algorismes presentats, però mai havia fet una comparativa acurada entre ells i, a més, desconeixia els altres dos. De manera més concreta, m'interessava implementar els quatre algorismes següents (explicats amb més detall al capítol 9):

- **Forward shading:** L'algorisme clàssic. Quan es dibuixa cada objecte, es determina tota la il·luminació de les llums i la seva interacció amb el material. L'utilitzaré com a referència visual, doncs és fàcil d'implementar.
- **Deferred shading:** Separa la part de dibuixar objectes i d'il·luminar-los en dos fases. Teòricament escala millor quan incrementem el nombre de llums, però té un alt cost base.
- **Deferred lighting:** Separa la part de dibuixar objectes, calcular la il·luminació i aplicar-la. Funciona de forma molt semblant al Deferred Shading, però intenta utilitzar menys memòria.
- **Inferred lighting:** També separa la part de dibuixar objectes, calcular la il·luminació i aplicar-la. El pas de calcular la il·luminació es fa a baixa resolució, per a intentar accelerar el procés. És l'única tècnica a part del Forward Shading que suporta bé transparències.

L'objectiu en aquest cas, era arribar a implementar els quatre algorismes de shading, seguint sempre aconseguir que les diferències visuals entre ells fossin les menors possibles.

Un cop aconseguit implementar-los, que per si mateix ja és un bon repte, volia comparar-los, centrant-nos en el seu rendiment en diferents situacions, i la qualitat de les imatges poden arribar a crear, doncs els diferents algorismes de shading no comparteixen els mateixos objectius. Aquest últim objectiu busca obtenir una referència ràpida que ens permeti saber els punts forts i febles de cada tècnica amb d'una ullada ràpida.

## Capítol 2

---

# Estudi de viabilitat

---

Els motius que feien viable el present projecte son la meua experiència prèvia i la quantitat de software de qualitat, i sense cost, a l'abast de qualsevol persona amb una connexió a la xarxa. Per tal d'argumentar-ho, primer analitzarem els recursos necessaris, tant tecnològics com econòmics, d'un projecte com aquest, i veurem com tots eren coberts i difícilment esdevindrien un obstacle per la finalització del projecte.

Respecte a recursos, tant de hardware, software o documentals, necessitem el següent:

- **Un ordinador amb una targeta gràfica 3D:** Un tant per cent molt alt dels equips del mercat compleixen els requisits per tal de desenvolupar el projecte. Respecte a la part de comparar el rendiments dels diferents apartats del projecte (la part final del projecte), tenint en compte que és una comparació relativa, no hauria de ser problemàtica. Tot i això, durant la última dècada, tots els jocs (i fins i tot els sistemes operatius) han passat a requerir targetes 3D, per la qual cosa la qualitat de les targetes gràfiques més barates ha augmentat significativament. No és estrany que ordinadors molt barats incloguin targetes 3D amb unes prestacions que fa uns anys només podíem somniar.

Igualment, a nivell personal, no soc un usuari mitjà de hardware, ja que tinc certs requisits que no tots els equips del mercat compleixen, principalment pels projectes que desenvolupo a casa i, perquè no dir-ho, pels jocs als que jugo. Principalment sempre he valorat molt la targeta gràfica que tenia el PC que volia comprar.

En resum, un ordinador prou potent per desenvolupar i fer l'anàlisi del projecte és un recurs que tenia disponible.

- **Software per editar, compilar, enllaçar i depurar C++:** En aquest cas, tenim dos opcions clares, software open-source, que és gratuït, o bé software propietari, que té un cost determinat.

Per la part d'edició, tenim opcions lliures com Emacs o Vim, que son editors amb varies dècades de desenvolupament al darrere, amb centenars de milers d'usuaris i de provada utilitat. Tot i que la opció de Vim va ser valorada, la seva corba d'aprenentatge accentuada va fer que el descartés com una opció a considerar. Un altre editor teòricament menys potent, però molt útil com a complementari, és el Notepad++, que té suport per editar de forma còmoda molts llenguatges. És l'editor que he utilitzat per editar qualsevol arxiu que no contingués codi C++.

En quan a la part de compilar, enllaçar i depurar C++, la opció més clara és la combinació **GCC**, el compilador i enllaçador open-source més conegut, i **GDB**, el debugger més versàtil open-source. Serien les opcions a triar, si no fos per una iniciativa orientada a estudiants que comento a continuació.

Microsoft fa uns anys va començar una iniciativa (*Dreamspark*) per tal d'oferir part del seu software gratis a estudiants. Tot i que és un argument sobre el que no m'esplaiaré, crec personalment que l'entorn integrat **Visual Studio** és el millor per desenvolupar sobre la plataforma Windows, i comparat amb les solucions que ofereixen altres plataformes, és la millor existent, tenint en compte que és un entorn integrat molt versàtil i àgil. Les llicències d'ús comercial costen entre 600 i 16000 euros, cosa que estaria completament fora de l'abast de qualsevol estudiant, però tenint en compte l'iniciativa citada, crec que és la millor opció a triar.

- **Models 3D:** Els models 3D tenen quasi sempre un cost, però en el present projecte ens limitarem a utilitzar aquells que hagin estat alliberats pel seu ús gratuït, ja que hi ha un ventall prou ampli d'aquest com per portar a terme el present projecte. Opcions disponibles son el model de l'atri d'Sponza, molt utilitzat en comparatives d'algorismes d'il·luminació, la catedral Sibenik, el famós repositori de models 3D d'Stanford (on hi ha el reconegut model de l'Stanford Bunny, que podem veure a la [Figura 2.1](#)) o, de forma més general, les webs que venen models 3D normalment també n'ofereixen de gratuïts, com per exemple **TurboSquid**.
- **Llibreries complementàries:** No és pràctic (ni possible) escriure tot el codi necessari per desenvolupar un projecte amb l'abast del present, almenys en un temps raonable. Tant les empreses com els particulars es serveixen de llibreries molts específiques, que cobreixen certes necessitats. Com per exemple, llegir certs formats de text (*XML*, per exemple), informació de rendiment de la targeta, llegir o comprimir textures, o llibreries que ens permeten crear interfícies d'usuari de forma senzilla.

Respecte a aquestes llibreries, tenim un autèntic mar d'opcions, tant de gratuïtes com d'extremadament cares. Per cobrir les necessitats del projecte he pogut utilitzar llibreries sense cap cost, sense haver de renunciar a cap dels objectius. De fet, més aviat al contrari, hi ha algunes llibreries que s'han obviat, per poder implementar-les jo mateix, i així aprendre més detalls sobre les tecnologies i formats que inclou el projecte.

Hem estudiat detingudament que tinguéssim el codi disponible de totes les que potencialment volíem utilitzar. El motiu, és que si ens trobem amb algun problema, podem examinar el codi per entendre si estem utilitzant la llibreria d'una forma no adequada, o si és un problema de la llibreria, corregir-lo. Amb llibreries de codi tancat això és molt difícil o impossible, i per tant ho hem tingut molt en compte.

- **Bibliografia:** Per a desenvolupar el projecte, hi ha molta informació que podem trobar a Internet, però és possible que necessitem explicacions més formals o detallades que les que existeixen a la xarxa. Per cobrir aquesta necessitat, hi ha molts llibres tècnics a l'abast del programador interessat en gràfics, però quasi tots tenen un preu elevat, normalment entre 40 i 70 euros cadascun.

Tenim diverses alternatives si no ens podem permetre aquesta inversió. Una possible, és restringir les referències a utilitzar, limitant-nos només a aquelles que son a la xarxa. Pot trobar-se el cas que la quantitat/qualitat d'informació sobre un tema concret no sigui suficient, o bé no tinguem

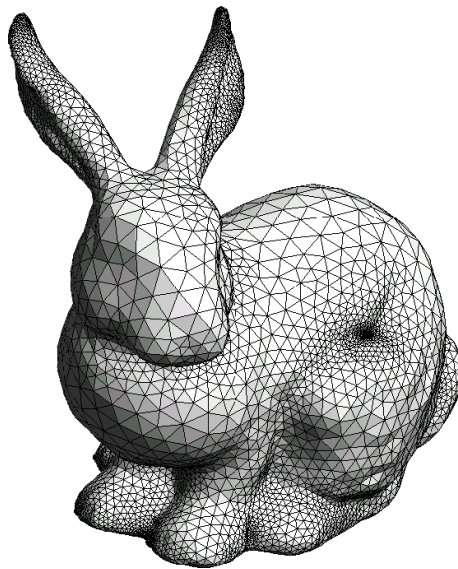


Figura 2.1: El famós model 3D Stanford Bunny

referències per a endinsar-nos més en la matèria, però és una solució senzilla. Per sort, com a alternativa, la majoria d'universitats tenen un nodrit grup de gràfics, amb molts llibres especialitzats a la biblioteca, doncs els usen com a referència constant, que podem consultar (com és el cas particular de la UDG), així que no caldria incloure-ho al pressupost. L'únic possible problema que podem tenir és inherent al fet que son llibres de préstec, amb la qual cosa poden no estar disponibles quan els necessitem.

Per sort, tornant al meu cas particular (com en el punt anterior), essent una persona interessada en la programació gràfica, he anat comprant llibres tècnics, relacionats amb aquest àmbit, al cap dels anys. A dia d'avui tinc la sort de tenir una nodrida llibreria que cobreix les necessitats del projecte.

- **Documentació de les tècniques:** Finalment, com estiguin de ben o mal documentats els algorismes que volem implementar és certament important. Aquest és possiblement el punt més feble, doncs a vegades la informació està repartida en diferents fonts, de forma incompleta, i altres cops, els algorismes només estan descrits de forma molt abstracta, amb la qual cosa pot ser complexa implementar-los.

A nivell de viabilitat, considerem que tenim la informació justa i necessària per implementar-los, tot i que segurament comportarà gran quantitat de recerca, proves i errors.

Com veiem, els recursos necessaris, almenys des de un punt de vista previ i a alt nivell, son coberts. Això ens deixa una única qüestió per tal de veure la viabilitat del projecte: tinc els coneixements necessaris per desenvolupar un projecte amb aquests requisits?

En capítols posteriors veurem una sèrie de conceptes per entendre que inclou un projecte de les dimensions de l'actual, però cal tenir en compte que s'aspira a desenvolupar un motor 3D de força complexitat, si més no, per portar-se a terme per un sol desenvolupador, i una vegada desenvolupat, utilitzar-lo per a fer una comparativa de tècniques. Només crear el motor 3D suportant una sola tècnica ja seria un projecte més que interessant i complex. Llavors perquè i com puc arribar a desenvolupar-ho?

La millor argumentació al respecte és la fascinació que sento pels gràfics 3D. Fa més de 15 anys que vaig descobrir que, una persona, a casa, podia crear mons 3D per on moure's. No calia esdevenir un elegit dintre una empresa, que em donés accés als secrets del món de les tres dimensions. Només necessitava uns quants llibres, voluntat i perseverança. Sé que pot semblar tòpic, però això va provocar un canvi, doncs tot i que tenia clara la meva *vocació*, dedicar-me a treballar fent jocs per ordinadors, ara tenia clar en que volia especialitzar-me: volia ser un programador centrat en gràfics.

Durant 15 anys, he dedicat gran part del meu temps lliure a estudiar sobre gràfics, implementar motors 3D de diferent complexitat, des de motors que funcionen completament per software com motors que utilitzen targetes gràfiques 3D, descobrir com funcionen les aplicacions per a crear continguts pels motors 3D i a parlar amb artistes 3D, per descobrir les necessitats des del seu punt de vista, i així aconseguir millors resultats finals. Fa uns pocs anys, la dedicació em va portar a aconseguir una feina en una empresa que es dedicava a desenvolupar un motor 3D, on encara vaig poder aprendre més. A dia d'avui, tinc la sort de treballar en una de les principals empreses de videojocs de Barcelona, on formo part de l'equip que desenvolupa la tecnologia dels nostres pròxims jocs.



Figura 2.2: Imatge creada en temps real utilitzant el primer motor comercial que vaig estar desenvolupant (circa 2007)

Això no intenta ser un exercici d'orgull, sinó que intento exemplificar que gran part dels coneixements previs pel treball, ja els tenia assolits en el moment de començar-ho. Per tal de detallar-ho, anem a enumerar els coneixements necessaris per implementar el projecte, i la meva experiència relacionada. Com que en capítols posteriors detallarem millor els conceptes i la implementació, en farem una visió molt superficial, centrant-nos ens dos aspectes, les dades d'entrada i el procés per mostrar-les a l'usuari:

- **Formats de dades:** Entenem per això els diferents formats de dades 2D i 3D utilitzats pel projecte. He treballat amb diferents formats per a les dades que contenen informació 2D i 3D, tant binàries com en text, i conec llibreries per carregar-los i, fins i tot, com programar jo mateix el necessari per carregar-los sense necessitar cap llibreria.



- **Procés de mostrar les dades a l'usuari:** Entenem per això com mostrem els gràfics 3D a l'usuari. Això inclou la decisió d'utilitzar una o altra llibreria gràfica (tot i que en PC, actualment només hi ha dos eleccions), entendre les bases matemàtiques que el procés de dibuixar inclou, i la forma més òptima d'implementar el motor 3D. Aquí, tornem a repetir l'experiència prèvia, doncs ja havia utilitzat una de les dues llibreries durant anys, ja tenia els coneixements d'àlgebra necessaris per crear un motor 3D, i acumulava l'experiència d'haver treballat en 3 motors comercials diferents, cadascun amb necessitats molt diferents (des de màquines PC de molt alta gamma, fins a consoles portàtils), que em donen una bona perspectiva.

Fins aquí hem vist la viabilitat del projecte, detallant cada un dels punts que hem considerat importants remarcar. Veiem que no tindrem problemes per la part de hardware, software o informació. La part més interessant, son les decisions que hem de prendre al començar el treball, ja que gràcies als coneixements previs i a la dificultat de la tasca, son decisions a prendre amb molta cura, ja que després n'haurem de patir les conseqüències.

## Capítol 3

---

# Metodologia

---

La metodologia de treball que s'ha seguit és dins de les anomenades metodologies àgils (*Agile*). La principal característica de les metodologies àgils és que la planificació inicial de tasques i estructura del desenvolupament és mínima, normalment modelant només una idea general de tot el sistema a implementar, potenciant un desenvolupament basat en iteracions curtes (normalment menys d'un mes per iteració), on és dissenyen i implementen parts petites de l'aplicació, que per si mateixes ja son funcionals. S'intenta que al final de cada iteració el producte sigui funcional, ni que les parts en que s'hagi treballat hagin de ser millorades en altres iteracions. Aquest sistema maximitza la possibilitats de canvis durant el desenvolupament de l'aplicació, i minimitza el risc global a errors en la planificació/disseny inicial, doncs al final de cada iteració és té una versió funcional que permet saber si s'han d'efectuar canvis o si l'aplicació té problemes de disseny.

Les metodologies àgils normalment tenen una persona designada com el **representant del client** que, en el cas d'aquest projecte, és el mateix que la persona que el desenvolupa, però quan aquest mètode s'aplica en empreses, és una persona dins de l'equip de desenvolupament que rep els requisits dels clients i els converteix en tasques a desenvolupar per l'equip. Durant les reunions d'equip per planificar la següent iteració, el representant del client afegeix o elimina elements a la llista de tasques a desenvolupar, i les prioritza, per tal que l'equip decideixi quines tasques cal fer durant la següent iteració.

A les metodologies àgils també és comú la celebració de reunions diàries, on els integrants de cada equip comenten en que van treballar el dia anterior i en que treballaran durant el dia. Aquestes reunions permeten transmetre a la resta de l'equip els possibles problemes de desenvolupament que cada integrant de l'equip ha trobat, i evitar possibles dependències entre membres de l'equip, que podrien bloquejar tasques.

De forma relacionada a aquestes metodologies, es potencia l'ús d'eines i tècniques per tal de facilitar el desenvolupament de les tasques, com son la programació basada en creació de petits tests abans de programar el codi en si mateix (**test-driven development**) o les eines d'integració continua, que creen contínuament versions de l'aplicació per tal de comprovar que aquesta segueix sent funcional. Podem veure, que si unim només aquestes dos, podem saber fàcilment durant tota la iteració si algun dels canvis efectuats ha provocat que l'aplicació no compleixi els requisits (o, directament, no funcioni).

Dins de les metodologies àgils, hem decidit utilitzar una variació al *desenvolupament basat en característiques*, en anglès, *Feature-Driven Development*, o **FDD**, com ens referirem a ell a partir d'ara. El FDD és un sistema ideat per *Jeff De Luca* el 1997, però la primera explicació de la metodologia la podem trobar al llibre **A Practical Guide to Feature-Driven Development** [19].

La metodologia FDD és basa en iteracions curtes, amb cinc fases diferenciades. Les dos primeres fases preparen un model general de tota l'aplicació, que detalla totes les característiques desitjades, i teòricament només s'executen un cop de forma seqüencial. Les tres fases restants s'iteren per cada una de les característiques que estem desenvolupant. Les cinc fases que la componen son les següents, segons el disseny original d'aquesta metodologia:

1. **Desenvolupament del model general:** Primer, es discuteix el sistema que es vol desenvolupar, a alt nivell, i tenint en compte el seu context. Un cop discutida la aplicació, es divideix aquesta discussió en models per dominis, o el que és el mateix, es divideix el sistema per tal que petits grups el modelin a alt nivell. Un cop efectuats els models, es discuteixen per domini, on un o la fusió de més d'un model s'utilitzen. Finalment, els models per domini s'uneixen per crear un model general.

Explicat de forma més senzilla, aquesta fase defineix el disseny general de la aplicació respecte les necessitats del producte i el seu context.

2. **Creació de la llista de característiques:** Un cop decidit el model / disseny general i amb els coneixements obtinguts a través del procés, s'identifiquen les característiques a implementar. Això es fa dividint cada domini en parts, anomenades *àrees amb subjecte*. Aquestes àrees, a la seva vegada, és divideixen en activitats, on cada activitat és un seguit de característiques per portar-la a terme. És important que cada característica es pugui completar en menys de dos setmanes, si no s'ha de dividir en característiques més petites.

El que fem, es dividir les diferents parts que hem identificat del sistema, fins identificar característiques isolades que podem implementar en menys de dos setmanes.

3. **Planificació de la característica:** En aquesta fase planifiquem la característica i l'assignem a un programador principal.
4. **Disseny de la característica:** El programador principal dissenya un grup de característiques que han de ser desenvolupades en dos setmanes. En la metodologia original, és modelen els mètodes i les classes que ha d'utilitzar la característica.
5. **Implementació de la característica:** Es programa la funcionalitat dins dels mètodes i classes dissenyats anteriorment, és comprova que aquesta funcioni respecte els requisits i proves automàtiques, i passa a afegir-se la característica al producte.

El procés de desenvolupament amb FDD, llavors, és basa en executar la fase 1 i 2, i després iterar les fases 3, 4 i 5 per cada característica del producte. Hi ha variacions que fan que la fase 3 planifiqui totes les característiques, i llavors només s'iteri sobre les fases 4 i 5 (a la Figura 3.1 podem veure aquesta variació).

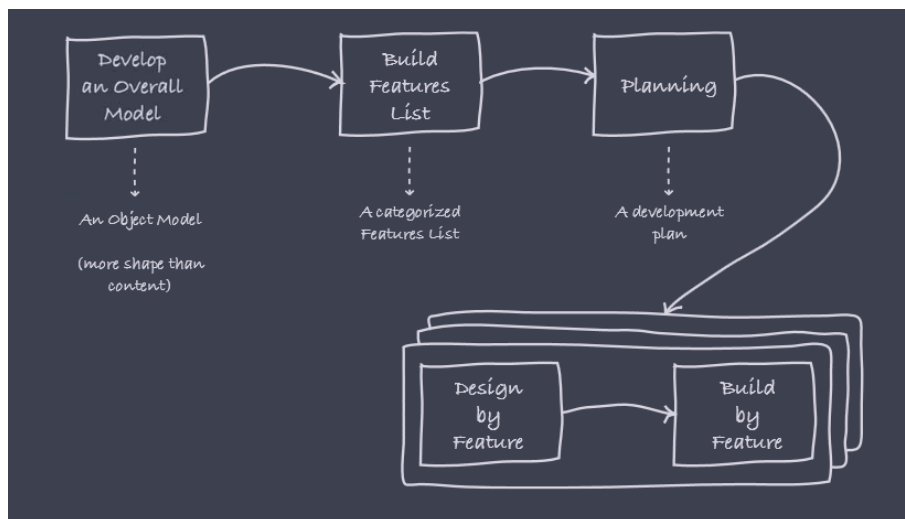


Figura 3.1: Procés de la metodologia basada en característiques (© 2005 Nebulon Pty. Ltd)

Hem utilitzat una metodologia molt semblant a aquesta, en que les fases 1 i 2 es van executar dissenyant tot el sistema a molt alt nivell i les característiques que volíem que complís, per després iterar per cada característica, implementant-les una a una. La principal variació és que no hem fet el disseny de classes i mètodes per característica utilitzant software de modelat de software, doncs creiem que és una forma arcaica de dissenyar sistemes. Hem fet un disseny general a alt nivell de l'organització dels mòduls i dades, i després hem implementat les característiques utilitzant un mètode més semblant al **vertical slice**, on hem creat una versió

incompleta de gran part del codi, per tal de comprovar el correcte disseny de l'aplicació a alt nivell, per després iterar en cada un dels components. També s'ha iterat molt utilitzant **top-down**, creant sistemes d'alt nivell amb la funcionalitat dels sistemes inferiors incompleta, per després implementar les capes inferiors.

Per exemplificar-ho amb la metodologia utilitzada per implementar el projecte, les primeres iteracions del projecte només permetien dibuixar objectes amb càmera fixa. Després es va dissenyar un sistema senzill per suportar diferents algorismes de shading, es van dissenyar uns pocs mètodes, i es va integrar al sistema. Finalment, un a un, cada un dels algorismes de shading van ser implementats en la seva forma més bàsica. Un cop tots els algorismes de shading suportaven certa característica de dibuixat, s'afegia una nova característica a cada una de les implementacions, es comprovava i s'iterava.

## Capítol 4

---

# Planificació

---

Anem a veure una planificació a alt nivell de com portarem a terme el projecte. Ho farem en concordança a la metodologia àgil, sense incidir en molts detalls, si no en una visió general del projecte i un seguit de característiques que volem. En capítols posteriors detallarem més les característiques i els requisits per considerar-les completades, mentre que en l'actual ens centrarem en una descripció breu de cada característica i el resultat esperat al completar-la.

Respecte a la metodologia FDD que hem seguit, primer hem de dividir el sistema en una serie de dominis o parts clarament divisibles, que podem modelar per separat, per després identificar les característiques que cada mòdul ha de tenir. Un cop tinguem una llista de característiques, farem una estimació del temps que trigarem a completar-les i quin resultat esperem de cada una d'elles.

### Separació del sistema en mòduls

Anem a veure quins mòduls considerem divisibles de forma neta, i explicarem de forma breu cada un d'ells. Volem un motor 3D, que pugui carregar diferents escenes, dibuixar-les utilitzant diferents algorismes de shading i navegar per elles. També volem treure mesures diverses de cada escena amb cada algorisme de shading, per després comparar-les. D'aquesta explicació a alt nivell identifiquem diversos mòduls:

- **Sistema per carregar escenes 3D:** Carregar dades del disc i convertir-les a un format fàcilment dibuixable. Volem suportar objectes, llums i càmeres
- **Sistema de control de càmera:** Posicionar i rotar la càmera respecte a l'entrada de l'usuari

- **Sistema de dibuixat 3D:** Dibuixar objectes 3D amb diferents algorismes. Que cada algorisme estigui clarament separat. Crear una abstracció de la llibreria gràfica
- **Sistema de mesura de rendiment:** Extreure dades de rendiment i nombre d'elements de cada tipus a l'escena

### Creació de la llista de característiques

Un cop identificats els quatre mòduls, necessitem identificar quines característiques volem per mòdul.

El mòdul que carrega escenes 3D ha de poder llegir dades que descriu geometries, llums i càmeres. Volem que la descripció d'aquestes dades estigui en un format estàndard. Les dades que ofereixi aquest mòdul han de ser fàcilment dibuixables, per tant s'han de processar si no ho són. La càrrega ha de ser ràpida, si algun procés és ineludiblement lent, cal ficar mecanismes per reutilitzar les dades processades. Una llista de alt nivell de les característiques d'aquest primer mòdul, seria:

- Càrrega de geometries
- Càrrega de llums
- Càrrega de càmeres
- Processament de les dades per dibuixar
- Identificar processos lents i guardar els resultats d'aquests per evitar tornar a processar-los

El mòdul del sistema de càmera, ens ha de permetre agafar les entrades d'usuari, tant sigui ratolí com teclat, i convertir-les a una descripció que pugui utilitzar el sistema de dibuixat. La seva llista de característiques seria:

- Processar entrades d'usuari
- Poder traslladar la càmera
- Poder rotar la càmera
- Generar una descripció utilitzable pel sistema de dibuixat

El mòdul de dibuixat 3D és molt més complexe, per això utilitzarem una visió de molt alt nivell. Primer, volem poder canviar la llibreria gràfica que utilitzem, si ho desitgem, per si aquesta esdevé obsoleta. Volem suportar diferents algorismes de shading, però mantenir totes les parts de dibuixat comuns fora de

l'implementació d'aquests algorismes, com per exemple determinar quins objectes son visibles. Volem poder utilitzar les dades provinents del sistema de càrrega i càmera, que conformaran l'estat del sistema de dibuixat. Volem suportar modes de depuració i informació extra de dibuixat, per facilitar el desenvolupament del sistema.

- Abstracció de la llibreria gràfica
- Implementació de l'abstracció gràfica amb una llibreria gràfica (per exemple, Direct3D9)
- Implementació d'un sistema de depuració gràfica
- Implementació d'un sistema per utilitzar les dades provinents del mòdul de càrrega
- Implementació dels algorismes no relacionats amb shading
- Implementació dels algorismes de shading

L'últim mòdul és el que s'ocupa de mesures de rendiment i altres comptadors. Volem poder guardar històrics de dades del rendiment en diferents punts de l'execució, junt amb altres, com per exemple el nombre de triangles o una captura del moment en que s'han consultat les dades. Una llista de característiques seria:

- Poder guardar dades de rendiment gràfic
- Poder guardar comptadors d'elements del mòdul de dibuixat
- Poder capturar la imatge generada pel mòdul de dibuixat

Un cop vistes la llista de característiques, només en queda estimar el temps que trigarem a completar cada part. D'acord amb la filosofia àgil, es considera que l'estimació del temps és orientativa, i que només després de diverses iteracions podem estimar de forma més exacta. Primer detallarem l'estimació del temps a nivell de mòdul, i després de les característiques.

- **Sistema per carregar escenes 3D:** 4-6 setmanes
- **Sistema de control de càmera:** 1-2 setmanes
- **Sistema de dibuixat 3D:** 9-12 setmanes
- **Sistema de mesures de rendiment:** 1-2 setmanes



Que en total, suma un màxim de 22 setmanes, uns **5 mesos i mig de feina**.

Ara anem a estimar les característiques dels dos mòduls més complexes, el mòdul de càrrega d'escenes i el de dibuixat, el temps invertit en els altres dos mòduls i les característiques que inclouen, son massa poques com per fer estimacions acurades. Començarem amb el sistema de càrrega:

- **Càrrega de geometries:** 2-3 setmanes
- **Càrrega de llums:** 2-3 dies
- **Càrrega de càmeres:** 2-3 dies
- **Processament de les dades per dibuixar:** 1 setmana
- **Identificar processos lents i guardar els resultats per reutilitzar-los:** 1 setmana

Podem veure que hi ha dos tasques que tenen estimacions per sota la setmana. Aquest tipus d'estimacions no son desitjables al fer les primeres planificacions, degut a que acostumen a ser errònies.

Respecte l'altre mòdul que volem detallar, el de dibuixat, considerem la següent estimació:

- **Abstracció de la llibreria gràfica:** 1 setmana
- **Implementació de l'abstracció gràfica amb una llibreria gràfica:** 1-2 setmanes
- **Implementació d'un sistema de depuració gràfica:** 1 setmana
- **Implementació d'un sistema per utilitzar les dades provinents del mòdul de càrrega:** 1-2 setmana
- **Implementació dels algorismes no relacionats amb shading:** 1 setmana
- **Implementació dels algorismes de shading:** 4-5 setmanes

Cal notar que aquestes estimacions son fetes respecte setmanes laborals de jornada completa, és a dir, 5 dies a la setmana, 8 hores al dia. Hem cregut més adient calcular-ho així, doncs ens fica en un marc on podem comparar amb altres estimacions, a diferencia de si comptem dies naturals, o pressuposem més o menys hores per dia.

## Capítol 5

---

# Conceptes previs

---

Donar una introducció al detall de tots els conceptes necessaris per entendre el treball amb profunditat és complexe i podria comprendre fàcilment 3 o 4 vegades l'abast de l'actual projecte. Per això, donaré una explicació d'alt nivell de tots els conceptes, i detallaré aquells que estan més estretament relacionats amb el treball. La resta, els referenciaré a la bibliografia.

L'actual treball es centra en els gràfics 3D generats per ordinador, amb una alta interactivitat per part d'un usuari utilitzant algun dispositiu d'entrada (com ara un ratolí o un teclat). Especialment, volem estudiar diferents maneres de, donada una o varies fonts de llum, com calcular la interacció d'aquestes amb els objectes que poblen el nostre món, el seu rendiment i avantatges/desavantatges.

Degut a que ens centrem en els gràfics per ordinador interactius i en temps real, ens centrarem en conceptes altament relacionats amb el hardware accelerador de 3D. Aquestes targetes, tal com el seu nom indica, ens permeten accelerar el pintat de gràfics 3D i gran quantitat d'operacions relacionades. No té sentit avui en dia discutir implementacions o tècniques basades en motors 3D software, ja que no són competitives a nivell de velocitat ni amb les targetes de més baix rendiment del mercat, i tenen poc lloc en la discussió de gràfics per ordinador en temps real.

### 5.1 Conceptes bàsics

Anem a veure una sèrie de conceptes bàsics, fonamentals per entendre la resta del capítol i el treball:

- **Primitiva de pintat/renderitzat:** Element bàsic de dibuix. Ens limitarem a considerar els punts, línies i triangles com a únics elements. A

efectes pràctics, sempre parlarem de triangles, doncs qualsevol polígon convex es pot convertir a triangles de forma trivial, i també qualsevol polígon no convex, tot i que no és una operació tant senzilla

- **Objecte o model:** Col·lecció de primitives de pintat. Porten sempre associat un o varis materials
- **Textura:** Imatge que s'utilitza sobre la superfície d'un objecte, per modificar-ne l'aparença o les propietats d'aquesta
- **Material:** Definició de les propietats de la superfície d'un objecte. Per exemple, el seu color, rugositat o transparència en son propietats típiques
- **Llum:** Element que emet llum. A la realitat, les llums sempre tenen volum, però ens els gràfics 3D se'n simplifica la definició a un element infinitament petit, principalment per motius de rendiment
- **Càmera:** Lloc, orientació i altres propietats de visualització, utilitzades per definir com s'observen els elements d'una escena
- **Escena:** Col·lecció d'objectes amb els seus materials, llums i càmeres
- **Frame:** Terme anglès que significa quadre. Col·loquialment, s'entén un frame com una imatge generada per un ordinador
- **Shading:** Terme anglès per designar el procés de determinar la interacció entre una llum i un material
- **Pixel:** Element únic d'una imatge. Normalment descriu un color, però pot descriure altra informació
- **Normal:** Vector perpendicular a una superfície

## 5.2 Gràfics 3D

A nivell senzill, els gràfics 3D només es tracten de, respecte a una escena tridimensional, generar una imatge bidimensional, com exemplifica la [Figura 5.1](#).

Normalment, es diu a l'encarregat del procés de generar una imatge respecte a una escena, la *pipeline gràfica*, i aquesta inclou diferents fases, l'arquitectura de la qual veurem a continuació. Detallarem només la seva funcionalitat i no com implementar cada una de les fases, degut a que bé ho veurem en detall en capítol posteriors o bé no en tenim control en l'actual generació de hardware 3D.

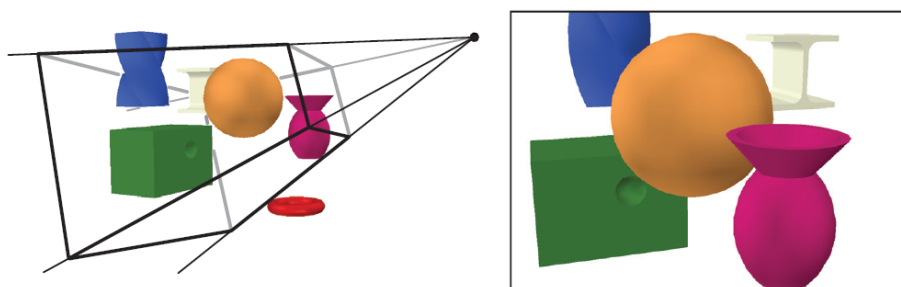


Figura 5.1: Projectió

### 5.3 Arquitectura pipeline gràfica

Utilitzarem el terme anglès *pipeline*, que significa, traduït de forma no literal, cadena de muntatge, per definir el procés de generar una imatge 2D des d'informació 3D. La part més important a entendre de la pipeline, és que és una cadena, i per tant, està formada per diferents fases. Aquestes fases estan estretament lligades, i cada una d'elles obté les dades de l'anterior. A nivell de rendiment, la velocitat d'aquesta cadena serà determinat per la fase més lenta, doncs esdevindrà el coll de botella.

Anem a exemplificar el concepte de coll de botella. Pensem en una cadena que munta contínuament ninots de joguina. Si, en ordre, tenim 3 fases, muntar braços, muntar cames i pintar, i triguin, corresponentment, dos minuts, dos minuts i tres minuts, no podrem muntar un ninot en menys de tres minuts, doncs les fases de muntar braços i cames hauran de restar parades esperant a pintar. Si no fos així, acumularíem ninots sense pintar fins a l'infinit. Entenem la fase de pintar com el coll de botella de la cadena, i si aconseguíssim reduir el temps de pintat, podríem produir més ninots per minut.

El sistema de cadena de muntatge també s'utilitza en gràfics per ordinador. Una aproximació d'alt nivell de les fases d'aquesta cadena, la podem trobar al *Real-Time Rendering* [1], on proposa una primera divisió en 3 fases, **aplicació**, **geometria** i **rasterització**, com podem veure a la [Figura 5.2](#). Cal notar que és un model de pipeline, després pot estar implementat de forma diferent en diferents targetes gràfiques, o motors gràfics. Per exemple, diferents fases poden esdevenir una de sola, o bé a l'inrevés, una sola fase pot esdevenir varies sub-fases.

Com podem veure a la part inferior de la [Figura 5.2](#), cada una d'aquestes fases, pot dividir-se en fases més petites o sub-fases, que llavors esdevindran una pipeline en si mateixa. Cada una d'aquestes sub-fases s'executarà de forma simultània amb la resta. Res evita executar sub-fases sense dependències entre elles de forma paral·lela, o duplicar-les, de manera que cada una de les sub-fases duplicades s'encarreguin d'un subconjunt de les dades, accelerant el processament

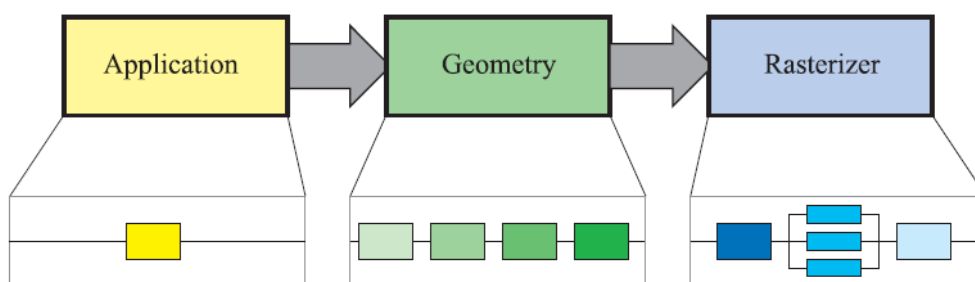


Figura 5.2: Pipeline

d'aquestes.

Abans de detallar cada una de les fases, veurem més en detall la velocitat de processament d'una pipeline, és a dir, el ritme al qual podem produir imatges (o renderitzar). I com aquesta és determinada per la fase més lenta de les tres citades. Una forma d'expressar comunament aquesta velocitat, és el nombre d'imatges per segon, o utilitzant les sigles angleses, *FPS (frames per second)*. Tot i això, en general, ens interessarà més saber el temps que triga en generar-se una imatge, que en el context de gràfics interactius, es mantindrà per sota d'un centenar de mili-segons, per la qual cosa normalment utilitzarem els mili-segons. Normalment utilitzarem els *FPS* per a informació mitjana de rendiment, i mili-segons per a mesurar el temps que triga a executar-se cada una de les fases.

La primera de les 3 fases, la **fase d'aplicació** normalment s'implementa en software, utilitzant la CPU de l'ordinador, que pot tenir varis nuclis, permetent que parts d'aquesta fase s'executin en paral·lel, accelerant-ne el procés. Tasques incloses dins de la fase d'aplicació son la creació de llistes del que hem de pintar, actualitzar l'animació o càlcul de físiques. Depenent de les necessitats de l'aplicació que estem implementant, poden incloure altres elements, per exemple, en jocs voldrem calcular la intel·ligència artificial dels enemics.

La segona fase, la **fase de geometria**, és l'encarregada de transformar les geometries, eliminar aquelles no visibles o retallar aquelles parcialment visibles. A nivell general, s'ocupa de calcular exactament que hem de pintar (les llistes creades anteriorment acostumen a ser super conjunts del que volem pintar, principalment per motius de rendiment), com ho hem de pintar i a on ho hem de pintar. A l'època de les primeres generacions de targetes 3D domèstiques, aquesta fase s'implementava total o parcialment utilitzant software. Això era degut a que la quantitat d'informació a processar en aquesta fase acostumava a ser molt inferior que en la següent fase (rasterització), i per tant no acostumava a ser el coll de botella. Des de fa més d'una dècada, aquesta fase és completament implementada a la GPU.

L'última de les fases d'alt nivell és la **fase de rasterització**, que s'ocupa

de generar la imatge final, amb la informació provinent de la fase anterior i dels càlculs que només és puguin fer per pixel. Aquesta és la primera fase que va ser implementada a les targetes 3D, degut a que és la fase que més fàcilment pot esdevenir el coll de botella, com exemplificarem a continuació.

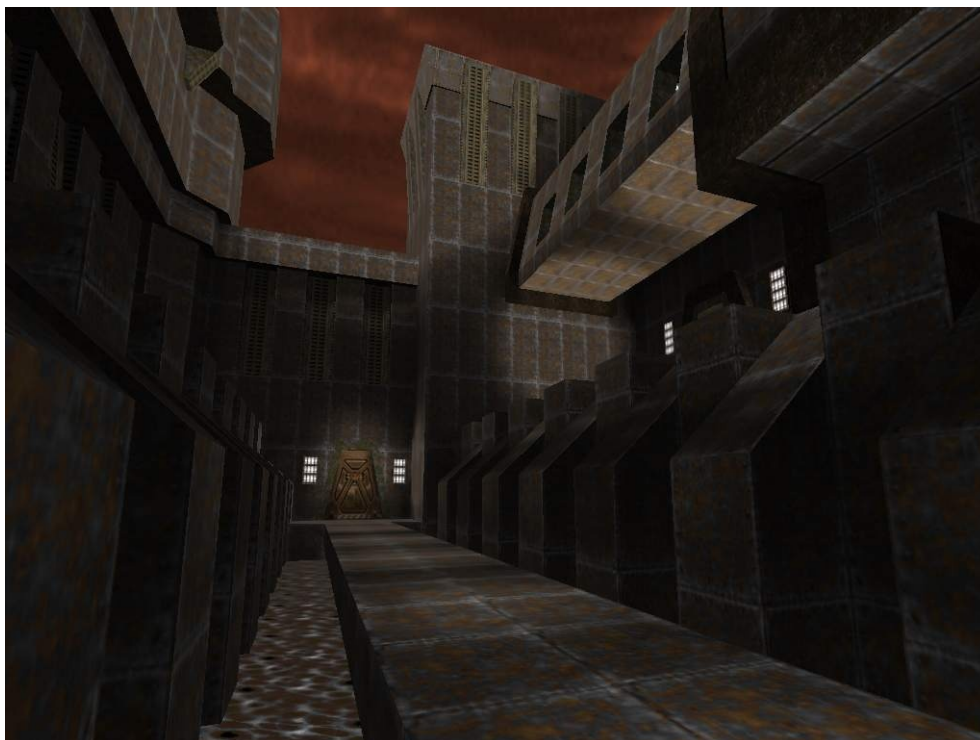


Figura 5.3: Quake 2

Suposem un frame amb una escena 3D típica del Quake 2 (Figura 5.3), un joc 1997, l'època de la gran explosió de les targetes gràfiques. Aquest frame té entre 600 i 900 polígons [25], polígons que pressuposarem compresos per un nombre baix de vèrtexs, 10, tot i que segurament superior a la mitja. Això ens dona un màxim de 9000 vèrtexs a processar per la fase de geometria. En canvi, la fase de rasterització, ha de pintar-ho a la pantalla. Suposem una resolució habitual de 1997, en el context del jocs, 640x480. Això ens dona 307200 pixels a processar per omplir tota la pantalla, 34 vegades més elements a processar que en l'altra fase. Altre cop, si pressuposem que triguem el mateix a processar un vèrtex que un pixel, podríem augment 34 vegades la quantitat de vèrtexs a processar, sense notar una davallada de rendiment, tenint en compte que les diferents fases de la pipeline s'executen de forma simultània.

Hi ha molts detalls que hem obviat en pro de la senzillesa de l'explicació, però és clar perquè la fase de rasterització va ser la primera en implementar-se en hardware. Un cop donada la visió general de les fases de més alt nivell, anem

a veure cada una d'elles en detall.

### 5.3.1 Fase d'aplicació

La fase d'aplicació és aquella sobre la que el desenvolupador té total control. Comparativament, i tenint en compte que ens centrem en una pipeline que utilitza GPUs, que tenen algunes fases implementades de forma immutable en hardware, aquesta és la fase amb més opcions. S'ocupa principalment de facilitar les dades a la fase de geometria i la informació necessària per la seva execució. A alt nivell, vol dir que hem de facilitar les primitives a pintar i les transformacions necessàries a aplicar.

Determinar les primitives a pintar normalment inclou la utilització d'algun sistema per determinar el set mínim de primitives visibles, les modificacions a l'estat del que anem a pintar respecte l'entrada de l'usuari (per exemple, un teclat) o l'animació d'objectes de l'escena, entre altres. Una visió molt simplificada, es veure-la com la fase en que tractem les dades a més alt nivell: a partir d'aquesta fase comencem a veure grups de primitives, transformacions i informació per rasteritzar, i tenim poca o cap informació lògica.

### 5.3.2 Fase de geometria

La fase de geometria tracta amb les primitives, els seus vèrtexs i com hem de transformar-los. Seguint el model proposat pel **Real-Time Rendering** [1], dividirem aquesta fase en les següents sub-fases: **transformació de model i vista**, **shading de vèrtexs**, **projecció**, **clipping** i **transformació a pantalla** (Figura Figura 5.4).

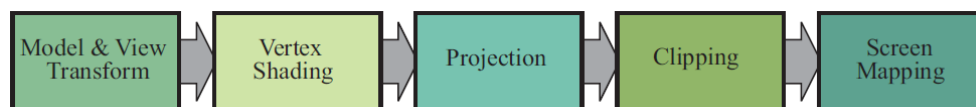


Figura 5.4: Sub-fases de la fase de geometria

La fase de geometria ens serveix per remarcar el que dèiem al introduir el model que estem detallant: diferents sub-fases del model poden acabar implementades en una de sola, doncs és habitual, en targetes gràfiques 3D, que les 3 primeres fases de geometria (transformació de model i vista, *shading* de vèrtexs i projecció) poden s'executin a la vegada.

Anem a veure les sub-fases amb més detall.

### Transformació de model i vista

Per portar a terme el procés de generar una imatge 2D a partir d'una llista d'objectes o primitives 3D, necessitem transformar-ne les propietats per un seguit d'espais vectorials (o sistemes de coordenades). Per comoditat i millor ús dels recursos, els objectes provinents de la fase d'aplicació venen en **espai d'objecte**, que és l'espai propi de cada objecte (a efectes pràctics, és l'objecte sense cap transformació). Des d'aquest espai ens interessa transformar l'objecte a l'**espai de món**, que és un espai comú, absolut i únic, compartit per tots els objectes. La transformació que s'aplica a un objecte per anar de l'espai d'objecte a l'espai de món, s'anomena **transformació de món** o **transformació de model**. L'avantatge d'aquests dos espais, és que podem tenir múltiples còpies (anomenades comunament instàncies) dels objectes, utilitzant varies transformacions de món per objecte, mentre que només necessitem mantenir un objecte, amb la conseqüent reducció d'ús de memòria.

Un cop amb els objectes en espai de món, la següent transformació interessant a aplicar és la **transformació de vista** o **transformació de càmera** (exemplificada a la Figura 5.5), amb la qual tindrem les propietats en **espai de vista**. Podem entendre l'espai de vista com aquell que té la càmera centrada en l'origen de coordenades, i té certes propietats que son molt interessants. Un cop tenim els objectes en espai de càmera, la projecció esdevé més senzilla, el càlcul de distàncies fins a la càmera esdevé trivial, així com el *clipping* de primitives.

Per entendre fases posteriors, és útil anomenar el volum que, dins l'espai de vista, és visible. Anomenem aquest volum com **volum de vista**. Aquest ve determinat per la posició de la càmera, l'angle d'obertura d'aquesta, i una distància màxima i mínima que restringeixen que podem veure (a la Figura 5.5 és el volum blau clar, que representa un volum d'una càmera amb projecció de perspectiva).

Tant la transformació de model com la de vista s'acostumen a descriure com a matrius 4x4, que ens permeten descriure de forma compacta una transformació afina.

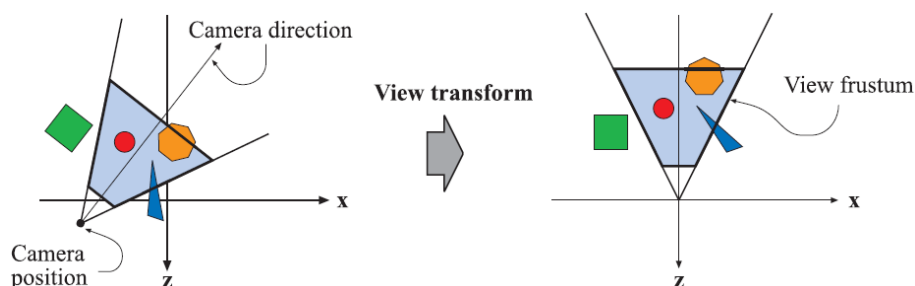


Figura 5.5: Transformació de vista



### Shading de vèrtexs

En aquesta fase implementem les parts de l'equació de shading, que utilitzem per pintar la primitiva, que es poden aplicar a nivell de vèrtex. Algunes parts de l'equació de shading es poden aplicar durant la fase de geometria, sobre els vèrtexs que s'està operant, i algunes s'apliquen durant la fase de rasterització, ja a nivell de pixel.

Els vèrtexs, a més de la posició, poden contenir més propietats, com la normal, coordenades de textura, o el color, entre altres que siguin necessàries per tal de descriure el material i aplicar el shading. Exemples d'operacions que podem portar a terme durant aquesta fase és calcular la incidència de la llum (si volem tenir il·luminació a nivell de vèrtex), modificar les coordenades de textura per tal d'animar-les o emular un efecte de distorsió (per exemple, l'últim permet crear un efecte d'aigua molt senzill i ràpid). Un cop aplicades les operacions necessàries a nivell de vèrtex, aquestes seran interpolades sobre la superfície de la primitiva, per tal de ser consumides per la fase de rasterització.

Finalment, i referenciant la secció anterior, aquests càlculs els podem aplicar en diferents espais, depenent de les necessitats. Això és possible perquè al transformar totes les coordenades dels elements implicats (càmera, llums, propietats dels vèrtexs), es mantenen les relacions relatives (angles i distàncies) entre elles. Per exemple, és pot utilitzar l'espai de vista, ja que no cal calcular la distància fins a la càmera, però a canvi no podem precalcular-la, ja que la càmera es pot moure, i teòricament podem perdre precisió respecte a fer els càlculs en espai de món.

### Projecció

La fase de projecció s'ocupa d'aplicar la **transformació de projecció**. Recordem que en l'entrada d'aquesta fase, les coordenades estan en espai de vista (la fase de shading no canvia l'espai de coordenades). La projecció normalment transforma les coordenades perquè estiguin compreses dins del **cub unitat** (delimitat dins l'espai amb mínim  $(-1,-1,-1)$  i màxim  $(1,1,1)$ ), i aquest espai s'anomena **espai de clipping** o **espai post-projectiu**. Dit d'altra manera, transforma el **volum de vista** al **cub unitat**. Normalment, una de les coordenades, la profunditat, no és útil després d'aquesta fase, a nivell de generar la imatge final, però s'acostuma a passar a les següents fases, per motius que explicarem més endavant, quan detallem les sub-fases de la fase de rasterització.

Les dos projeccions més usades, son la projecció ortogràfica i la de perspectiva (Figura 5.6). La projecció ortogràfica té un volum de vista en forma de capsa, i la seva principal característica és que les línies paral·leles es mantenen paral·leles després de la projecció. Per això, a vegades també se l'anomena projecció paral·lela.

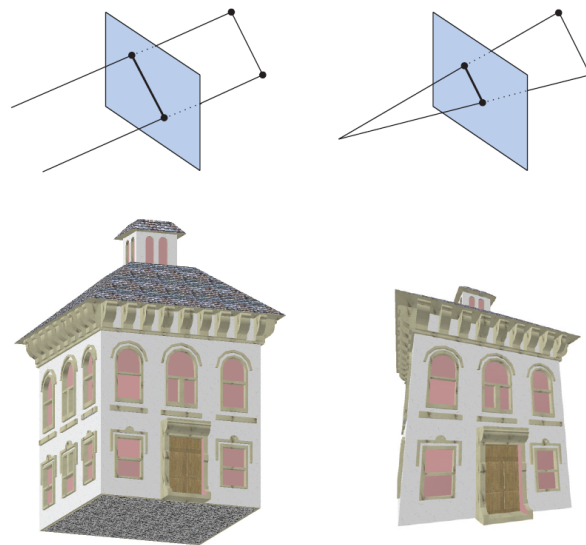


Figura 5.6: Transformacions de projecció

La projecció en perspectiva emula la forma que els humans veiem. El volum de vista d'una projecció en perspectiva és una piràmide amb base rectangular. Les línies paral·leles poden arribar a convergir a l'horitzó, després de transformar-les

La transformació de projecció també s'acostuma a representar amb una matriu  $4 \times 4$ .

### Clipping

Utilitzarem el terme anglès *clipping*, que té per traducció directa *retallar*, per denominar el procés que és dedica a retallar les primitives parcialment incloses dins del **volum de vista**. Quan una primitiva només és parcialment dins d'aquest volum, no tota la seva superfície ha de passar a la fase de rasterització, doncs cal recordar que les parts fora del volum no son visibles, i per tant no aporten informació a la imatge final.

Tècnicament és possible aplicar el *clipping* abans de les transformacions de vista i projecció, però fer-ho després té l'avantatge que les nostres coordenades visibles es troben dins del cub unitat, i per tant, determinar si una primitiva s'ha de retallar és molt senzill, doncs només cal comprovar si fa intersecció algun dels sis plans que delimiten el cub unitat. En podem veure un exemple a la [Figura 5.7](#).

### Transformació a pantalla

Després de la fase de *clipping*, un cop tenim les primitives retallades, necessitem transformar-les a l'espai de pantalla, doncs encara son coordenades tridi-

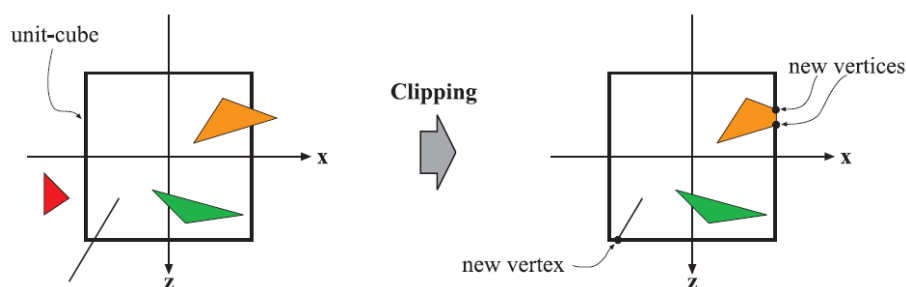


Figura 5.7: Clipping

mensionals dins del cub unitat. En cap cas transformarem la coordenada que representa la profunditat, doncs no és necessari per l'ús que en donarem més endavant. Des de les coordenades en **espai post-projectiu**, volem transformar les coordenades a les dimensions de la pantalla. Les coordenades en aquest espai les anomenarem **coordenades de pantalla**. Normalment, és una transformació composta d'escala i translació. Podem veure'n un exemple a la [Figura 5.8](#).

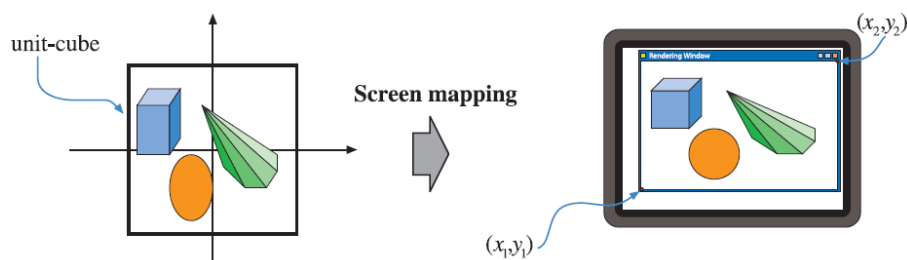


Figura 5.8: Transformació a pantalla

### 5.3.3 Fase de rasterització

L'entrada d'aquesta fase son els vèrtexs en coordenades de pantalla, junt amb la informació necessària complementària per aplicar el shading en espai de pixel. El que volem assolir és calcular els colors de cada pixel cobert per una primitiva, segons les equacions de shading, els vèrtexs transformats i les propietats de shading de cada primitiva.

De forma similar a la fase de geometria, aquesta fase és complexa i per tant la modelarem amb diverses sub-fases: **preparació de triangles**, **recorregut dels triangles**, **shading de pixels** i **fase d'unió**. ([Figura 5.9](#))

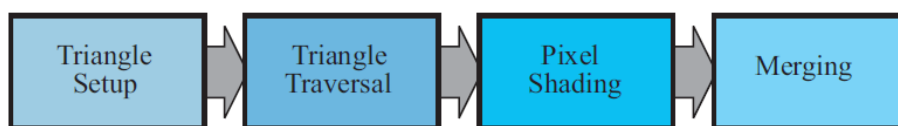


Figura 5.9: Sub-fases de la fase de rasterització

### Preparació de triangles

Aquesta fase s'ocupa de calcular els valors constants en tota la superfície del triangle. De forma efectiva, prepara totes les dades que és faran servir a les fases posteriors. És una fase important a tenir en compte, ja que significa un cost constant per triangle, sense importar la seva mida o cost de shading.

### Recorregut de triangles

En aquesta fase calculem les dades interpolades per cada pixel que és cobert per un triangle (Figura 5.10). Per cada pixel, hem d'interpoliar totes les propietats dels tres vèrtexs que componen el triangle, i guardar-les per la fase posterior. A aquest conjunt de dades se l'anomena fragment.

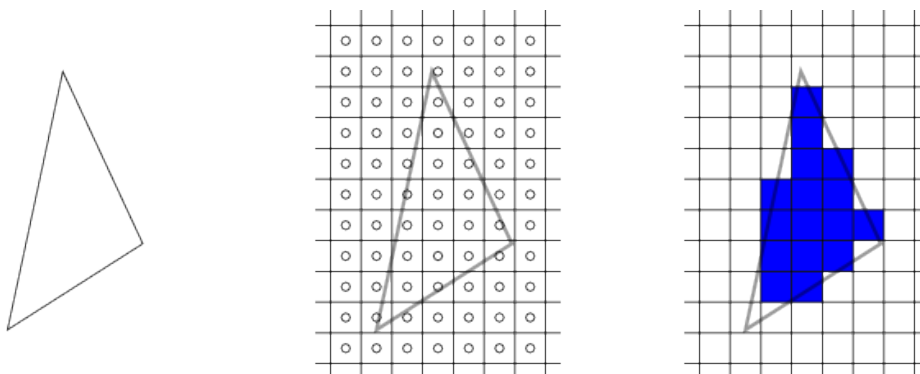


Figura 5.10: Preparació de triangles

Hi ha dos mètodes populars de fer el recorregut del triangle. El més popular en motors 3D software, calcula els punts inicial i final de cada línia, per llavors recórrer cada línia. Té com a principal avantatge que mantenim molt poques dades per línia, però és molt difícilment paral·lelitzable. Se l'acostuma a anomenar **recorregut d'arestes**, doncs anem recorrent les arestes del triangle, i interpolant les línies contingudes entre elles (Figura 5.11).

El mètode utilitzat en implementacions hardware és l'ús de les coordenades baricèntriques del triangle per a interpoliar les propietats dels vèrtexs. L'idea és calcular el rectangle que conté cada triangle, i llavors, podem utilitzar les coorde-

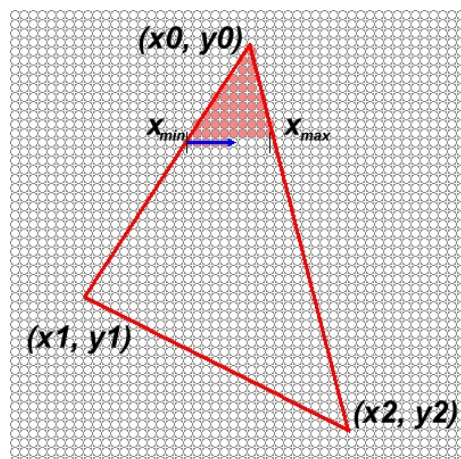


Figura 5.11: Recorregut d'arestes

nades baricèntriques, tant per determinar si un píxel és dins del triangle, com per interpolat les seves propietats. Té l'avantatge que és molt senzill d'implementar i paralelitzable, però és complex d'optimitzar.

### Shading de píxels

En aquesta fase fem tots els càlculs per píxel necessaris per avaluar l'equació de shading corresponent al material del fragment, respecte les propietats interpolades d'aquest. Si no estem avaluant la il·luminació per vèrtex, la calcularem en aquesta fase, que ens permet molta més flexibilitat, a costa d'un major cost. Una altra operació estretament lligada a aquesta fase, és el texturat, que no és més que aplicar una imatge sobre la superfície d'una/es primitiva/es (Figura 5.12).



Figura 5.12: Texturat

### Fase d'unió

La última sub-fase de la fase de rasterització s'ocupa d'unir/assignar la informació del color del fragment calculada durant el shading de pixels, amb els pixels continguts a l'imatge que finalment mostrarem a l'usuari.

La funcionalitat més destacable d'aquesta fase és la resolució de visibilitat o oclusió entre fragments, que és el procés de determinar, en el cas de múltiples fragments cobrint el mateix pixel (provinents de diferents primitives), quin té prioritat sobre l'altre. Normalment, voldrem que, per un pixel, el fragment més propers a la càmera sigui el visible. Si recordem la fase de geometria, guardàvem la profunditat dels vèrtexs, tot i que a partir de la fase de projecció no era clar l'ús que se'n donava. Amb la profunditat d'un fragment en pantalla, és pot veure que, donats varis fragments per pixel, podem comparar-les per determinar quin és més proper a la càmera. La forma més popular de portar a terme aquesta comparació és utilitzant un buffer de profunditat (també anomenat ZBuffer).

Un ZBuffer, és una imatge de les mateixes dimensions que la imatge on guardarem l'informació de color, però on emmagatzemem la profunditat del fragment més proper que hem pintat fins al moment. Quan un nou fragment és vol unir al buffer de color, es comprova la profunditat al ZBuffer. Si la profunditat del fragment d'entrada és menor a l'emmagatzemat al ZBuffer, i per tant, el fragment d'entrada és més proper a la càmera que el que teníem assignat anteriorment, assignem el color a la imatge de color, i la profunditat al ZBuffer. Altrament, descartem la informació del fragment d'entrada. Els avantatges d'utilitzar aquest sistema per determinar la visibilitat dels fragments, és que és molt senzill i podem dibuixar les primitives en qualsevol ordre, obtenint el mateix resultat. El principal desavantatge és que en el cas de primitives parcialment transparents, aquesta algorisme no és suficient, doncs l'ordre de dibuixat és important: primer hem de dibuixar totes les primitives opaques, i després les transparents de més llunyana a més propera, desactivant la resolució de visibilitat. Podem veure un exemple d'un buffer de profunditat amb la corresponent imatge generada a la Figura 5.13.

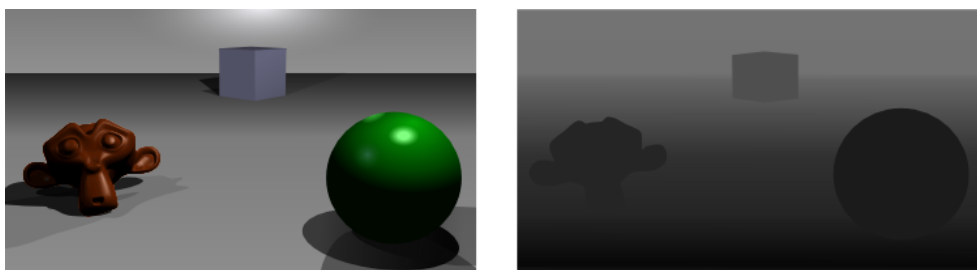


Figura 5.13: Buffer de color i de profunditat

En aquesta fase també tenim el suport per primitives parcialment transparents. Tot i que la imatge final només caldria que contingués colors, és molt útil també guardar la transparència/opacitat per pixel. Aquesta informació la podem utilitzar per evitar pintar fragments totalment transparents, o per barrejar la informació del fragment d'entrada amb la del que tenim emmagatzemada, poden aconseguir així fragments semi-transparents. Hi ha diferents formes de barrejar la informació d'entrada amb la emmagatzemada, per aconseguir diferents efectes, per exemple afegir de forma additiva, multiplicativa, o ponderada respecte la transparència de cada fragment.

Finalment, com aquesta fase és la que opera directament amb la imatge de sortida que mostrarem a l'usuari, i tenint en compte que el procés de generar aquesta imatge no és immediat, cal citar que normalment no treballem directament amb la imatge que veu l'usuari. L'esquema habitual és reservar prou memòria per dos imatges, i mentre és mostra una imatge, generar l'altra, i en el punt que està generada, intercanviar-les i repetir el procés. El terme utilitzat per aquest procés és *double-buffering*.

## 5.4 GPU

Tot i que podem implementar tota la pipeline gràfica utilitzant només software, com explica per exemple el llibre **Jim Blinn's Corner: A Trip Down the Graphics Pipeline** [3], actualment és més habitual que aquesta s'executi totalment en hardware dedicat, anomenat Graphic Processing Units (o GPUs). L'avantatge de tenir una implementació hardware de la pipeline gràfica és el rendiment, que és molt superior. A canvi, és perd certa flexibilitat.

Les primeres generacions de targetes 3D, només acceleraven la part final de la pipeline, ja que, com hem dit, és la que suposa normalment el coll de botella, però han anat incorporant altres parts de la pipeline, fins que avui en dia, és possible fins i tot utilitzar-les per implementar certes parts de la fase d'aplicació.

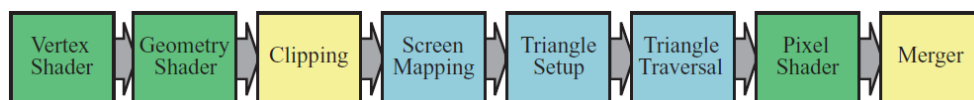


Figura 5.14: Pipeline hardware

Una altra evolució, a part de les fases implementades en hardware, és la capacitat per part del programador de configurar o canviar com operen certes fases. Les primeres generacions de targetes gràfiques oferien una pipeline amb funcionalitat fixa, que només permetia configurar certes opcions. L'evolució de les GPUs va anar introduint més opcions de configuració, fins al punt de permetre executar petites aplicacions en certes fases de la pipeline. Això va revolucionar les

possibilitats que oferien les targetes 3D, ja que, per exemple, és va passar d'unes poques opcions per canviar el shading aplicat a cada fragment, a que l'usuari pogués implementar l'equació de shading com cregués convenient. A l'actual generació de targetes, encara hi ha fases que només es poden configurar, i altres que son fixes, però es pot suposar que es mantindrà la tendència de l'última dècada de permetre que les fases siguin cada cop més configurables.

A la [Figura 5.14](#) podem veure les diferents fases de la pipeline que s'implementa normalment en hardware. Si ho comparem amb el model donat anteriorment, podem notar que falten varies fases. Això és degut a que varies de les fases descrites anteriorment s'executen de forma combinada, en una pipeline accelerada per hardware. La fase de **vertex shader** s'utilitza normalment per implementar les fases **transformació de model i vista**, **shading de vèrtexs** i **projecció**. La fase **geometry shader** ens permet operar, crear i destruir primitives, que en la pipeline detallada amb anterioritat s'havia de fer a la fase d'aplicació. La resta de fases corresponen a les del model vist anteriorment, amb fases que son completament fixes, com les de clipping, transformació a pantalla, preparació de triangles i recorregut de triangles, mentre que altres com la d'unió no son programables però es poden configurar.

Les dos fases que realment ens interessin en el context del projecte son la fase de **shading de vertex** (o vertex shader) i la fase de **shading de pixels** (o pixel shader), ja que podem implementar qualsevol algorisme de shading utilitzant només aquestes dos. Com ja hem dit, son fases totalment programables, així que explicarem què entenem per fases programables, per després explicar les particularitats de les dos fases.

#### 5.4.1 Fases programables

En totes les GPUs modernes, els nuclis que s'ocupen de processar les tres fases programables (pixel, vertex i geometry shader) comparteixen la mateixa arquitectura, és a dir, només es diferencien en les dades que tracten. Les primeres generacions de GPUs tenien nuclis diferents per cada fase, però es poc rellevant detallar-ho avui en dia.

A nivell de programació, els shaders normalment es programen en llenguatges molt semblants al C, que s'anomenen llenguatges de shading. Com a nota històrica, les primeres generacions de targetes s'havien de programar utilitzant directament assembleador, però avui en dia no és possible, almenys en PC. Els llenguatges de shading es converteixen en un llenguatge compilat intermig, semblant al que utilitzen llenguatges com Java o C#. En el moment de carregar els shaders compilats a la targeta, els drivers d'aquest interpreten el llenguatge compilat intermig, i el converteixen en instruccions específiques per la GPU. Això permet que el mateix llenguatge intermig pugui utilitzar-se en diferents targetes gràfiques.



A nivell del tipus de nuclis dins una targeta 3D, normalment es tracten de nuclis vectorials, que permeten operar sobre múltiples dades amb una sola operació (SIMD), principalment centrat en operar amb dades de tipus coma flotant (valors irracionals). Resulta adequat, doncs tractem principalment amb matrius o vectors de diferents dimensions, com per exemple transformacions, posicions, normals o colors, que són difícils de representar amb valors enters. També permeten, entre altres, seleccionar sobre quins elements operem d'un vèrtex, anomenat *swizzling*, per exemple, per comparar només el tercer element.

Permeten executar certes operacions que en gràfics són molt útils, com per exemple normalitzar un vector o calcular-ne el producte escalar, de forma molt eficient. La quantitat d'operacions matemàtiques implementades en hardware és extensa, doncs és l'únic ús que es dona a aquest nucli. Per exemple, comparat amb una CPU d'un ordinador, no tenen cap tipus d'instrucció per gestionar memòria.

Aquests nuclis també permeten control del flux del programa, és a dir, avaluar condicions que ens permeten executar o no certa part del codi. Comparat amb una CPU, l'ús que en podem donar és relativament limitat, doncs degut a l'arquitectura de les GPUs, la divergència en el temps d'execució entre diferents fluxos afecta molt negativament al rendiment.

Finalment, cal notar que tenint en compte que els models de shading són principalment utilitzats per artistes sense coneixements de programació, han proliferat els sistemes que generen el codi de shading amb eines visuals, com per exemple, editors de grafs jeràrquics, com podem veure a la [Figura 5.15](#). Els motors gràfics més populars ofereixen aquesta opció, doncs permet als artistes crear nous models de shading sense la necessitat d'un programador.

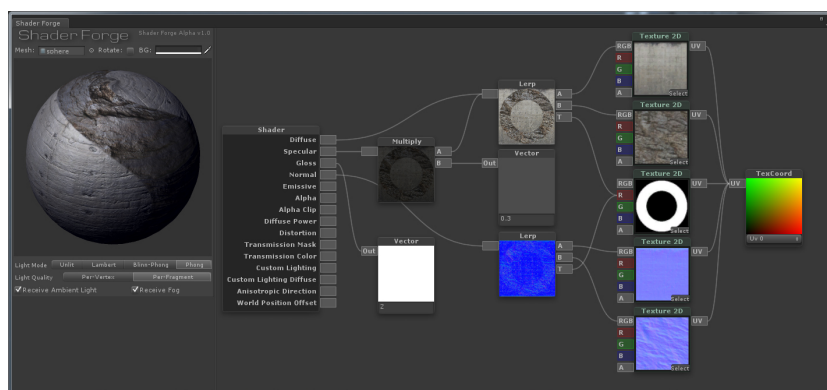


Figura 5.15: Editor de grafs per a generar shaders

Per més informació sobre les fases programables i la seva evolució històrica, recomano la lectura del **Real-Time Rendering** [1].

### 5.4.2 Vertex shader

Podem entendre un objecte triangular com un set de vèrtexs i l'informació de connectivitat, que ens indica quins vèrtexs hem d'unir per crear els triangles. El vertex shader només rep el set de vèrtexs, no té informació de connectivitat, i per tant opera amb ells a nivell individual. Les operacions es centren en les propietats del vèrtex, és a dir, crea, modifica o ignora, propietats com el color, la normal o les coordenades de textura.

Tal com hem dit abans, el vertex shader engloba tres fases de la pipeline, que transformen els vèrtexs d'espai d'objecte a espai de clipping. Aquesta serà l'única propietat que obligadament haurem de passar a les següents fases, mentre que podem ignorar la resta.

Usos típics del vertex shader són efectes de deformació a l'objecte, càlcul d'animació d'esquelets (com per exemple a la [Figura 5.16](#)) o fins i tot calcular il·luminació per vèrtex (es fa en GPUs de molt baix rendiment).

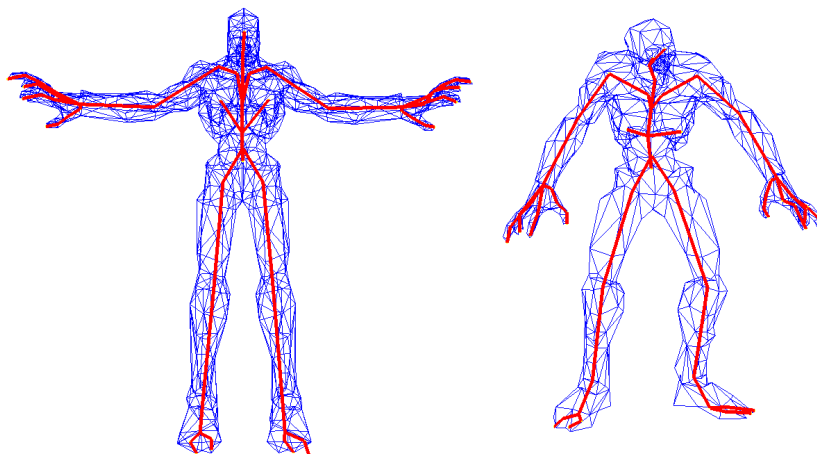


Figura 5.16: Exemple de geometria deformada segons un esquelet

Les dades de sortida d'aquesta, que són els vèrtexs amb les seves propietats (modificades o no), es passen a les següents fases, que amb la informació de connectivitat, generaran els triangles, i prepararan els fragments per ser consumits per la següent fase que detallarem, el **pixel shader**.

### 5.4.3 Pixel shader

El pixel shader consumeix els fragments resultant de les dades interpolades sobre la superfície de cada triangle, opera amb elles i genera sortides de color, que la següent fase, la d'unió, aplicarà sobre la imatge/buffer, depenent de com el pixel cobreix el fragment, si és opac o transparent, i la seva visibilitat. A més, el pixel shader també pot modificar la profunditat que es farà servir en la següent

fase per determinar la oclusió entre fragments, tot i que tots els fabricants de GPUs recomanen evitar-ho, ja que deshabilita certes assumpcions que acceleren el rendiment.

D'igual forma que en el vertex shader, el pixel shader opera sempre amb un únic fragment d'entrada, i sobre un únic pixel de sortida. Aquesta limitació, que podria semblar problemàtica a l'hora d'aplicar qualsevol tipus de filtre, no ho és tant, doncs podem sobreposar-la aplicant post-processos a la imatge.

Les GPUs normalment processen els pixels en blocs de 2x2 elements, ja que aporta l'avantatge de poder calcular els gradients en aquest bloc, fonamental per utilitzar textures amb mipmaps, entre altres usos.

Molt rellevant pel projecte és la capacitat del pixel shader de treure més d'un color per pixel, cadascun a una imatge diferent. Durant l'última dècada la capacitat de procés del pixel shader no ha fet més que incrementar, amb la qual cosa és practic poder fer més càlculs, a part del color. Tres de les tècniques en les que centrem el projecte són pràctiques gràcies a aquesta característica. A la [Figura 5.17](#) podem veure un exemple de la sortida d'un sol pixel shader que genera quatre sortides, que són 3 imatges de color i una imatge de profunditat.

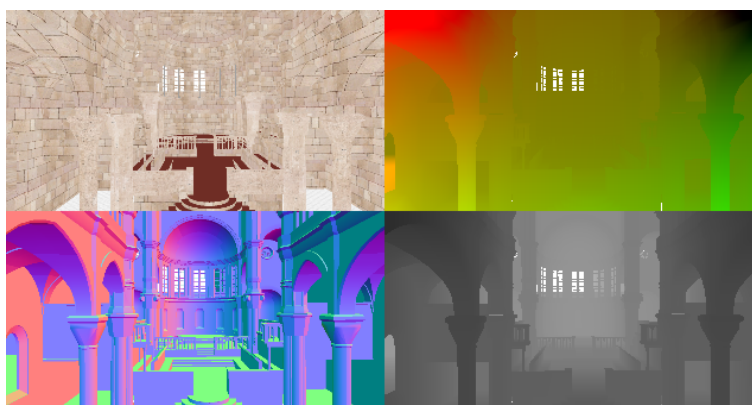


Figura 5.17: Exemple de multiples sortides per un unic pixel shader

## 5.5 Aparença visual

Per tal que un objecte sigui realista, no només ha de ser geomètricament acurat, sinó que l'aparença de la superfície també és molt important. En aquest apartat donarem una introducció als materials i a les llums, com es comporten al món real, i com podem modelar-les utilitzant GPUs programables.

Percebem l'aparença dels objectes al món real, respecte al següent procés:

- Les fonts de llum (natural o artificial) emeten llum

- La llum emesa interacciona amb els objectes de l'escena, on part és absorbida i part es torna a propagar
- Un receptor de llum (l'ull humà, una càmera, etc) absorbeix llum

A la [Figura 5.18](#) podem veure un exemple d'il·luminació on podem veure el tres punts anteriors. Primer, una font de llum, que en aquest cas emula el sol, il·lumina l'escena, que és la part més clara que podem veure. Aquest part més il·luminada, absorbeix part de la llum i en propaga una altra part (la quantitat i com la propaga depèn del material). La llum propagada il·lumina de nou l'escena (visible a la part esquerra de la [Figura 5.18](#)), que no és afectada directament per la única llum d'aquesta, i a més també es rebuda per la càmera.



Figura 5.18: Exemple d'il·luminació

### 5.5.1 Fonts de llums

Com es detalla als llibres de física, la naturalesa de la llum és complexa, i depèn de com s'observi, es comporta com una ona o com una partícula. A nivell dels gràfics per ordinador, ens interessa la teoria corpuscular, on la llum es tracta com una sèrie de partícules sense càrrega i sense massa, anomenats fotons, que porten radiació electromagnètica, i en la seva interacció amb la matèria, intercanvia energia amb aquesta. Entenem com una font de llum, aquell element que emet llum, però no l'absorbeix ni en canvia la direcció.

Hi ha moltes maneres de representar les fonts de llum, a nivell dels gràfics per ordinador, però ens centrarem en un model senzill. Dividirem les fonts de llum en tres tipus bàsics: llum direccional, llum puntual i llum de focus. Es tracta de grans simplificacions respecte a com funcionen les fonts de llum al món real, però permeten emular-les amb un cost de rendiment molt acotat. El color i intensitat d'una llum és una propietat comú a tots els tipus de llum, que normalment modelarem con un única propietat.

### Llum direccional

Les llums direccionals es consideren infinitament lluny, per la qual cosa els rajos de llum que emeten es poden considerar paral·lels. El principal exemple de llum direccional és el sol. Les llums direccionals es consideren mancades de posició, i normalment son definides únicament per la seva orientació. A la [Figura 5.19](#) podem veure una llum direccional.

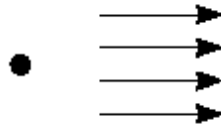


Figura 5.19: Llum direccional

### Llums posicionals

Els altres dos tipus de llums, son el que anomenem llums posicionals. Les llums posicionals s'anomenen així perquè considerem que tenen una posició a l'espai, el que a la realitat entendríem com que l'emissió de fotons es des d'un punt de l'espai. Això és una aproximació, doncs al món real les llums tenen volum.

Les llums puntuals emeten llum en totes direccions, com es pot veure a [Figura 5.20](#), mentre que les llums de focus restringeixen l'emissió de llum a una secció de l'espai, per exemple, un volum cònic d'aquest, com es pot veure a la [Figura 5.21](#).

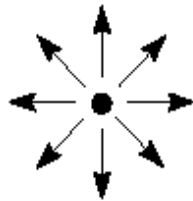


Figura 5.20: Llum omnidireccional

Les llums de focus, que normalment es defineixen respecte a un volum cònic, a part de la posició, tenen més propietats, com la seva direcció, l'angle d'obertura del con i l'atenuació des del centre d'aquest.

Físicament, al món real, la llum emesa per les llums posicionals sofreix una atenuació inversament proporcional al quadrat de la distància entre la font de llum i el receptor dels fotons emesos (en un entorn que no afecta als fotons). En les

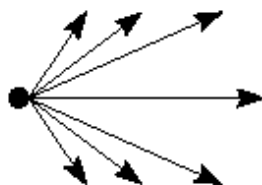


Figura 5.21: Llum spot

implementacions dels motors 3D s'acostuma a utilitzar aquest tipus d'atenuació per defecte, però també s'utilitzen altres tipus d'atenuació, bé sigui per motius artístics o limitacions del hardware.

### 5.5.2 Materials

A nivell dels gràfics 3D, el color que acaba arribant al receptor és resultat de l'interacció de la llum, que ja em descrit, amb la superfície dels objectes. Els materials són la descripció de com interacciona una superfície respecte la llum, i s'associa un material a cada objecte. Cada material és comprès per un seguit de propietats, que poden incloure textures, vertex i pixel shaders, i paràmetres variats. En aquest apartat descriurem com la llum interacciona amb les superfícies al món real i com ho modelem als gràfics per ordinador.

A nivell de materials, tota interacció entre la llum i la matèria és dividida en dos fenòmens: difusió i absorció.

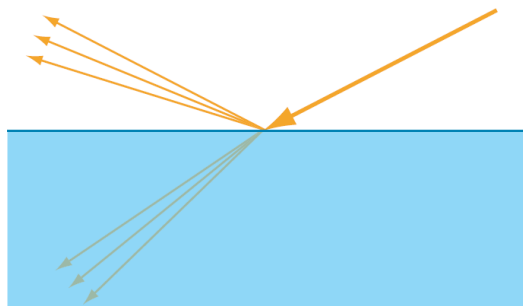


Figura 5.22: Difusió: reflexió i refracció

La difusió passa cada cop que la llum és troba amb una discontinuïtat òptica. Aquesta discontinuïtat pot ser la interfície entre dos substàncies amb diferents propietats òptiques, un canvi de densitat, etc. La difusió no canvia la quantitat de llum, només causa un canvi en la direcció de la llum, com podem veure a la [Figura 5.22](#). La discontinuïtat òptica més remarcable és la que trobem entre l'aire i la superfície d'un objecte. La difusió de la llum té dos direccions: cap a

dins de la superfície (refracció o transmissió) i cap enfora d'aquesta (reflexió). A la Figura 5.23 es pot veure aquesta interacció.

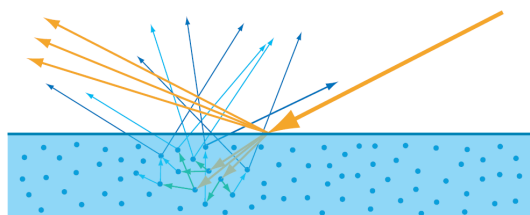


Figura 5.23: Interaccions llum-superfície

L'absorció passa dins de l'objecte i causa que la llum és converteixi en un altre tipus d'energia i desapareixi. Redueix la quantitat de llum, però no en canvia la direcció.

Els materials transparents els considerem filtres de color o atenuadors de les superfícies que tenen darrere seu. No modelem com funciona la llum realment al món real, on la llum que travessa objectes transparents en pot canviar la direcció o atenuar-la, ja que seria molt costós per aplicacions en temps real. La correcta representació d'objectes transparents està molt lligada a la **fase d'unió** citada anteriorment, ja que necessitem poder utilitzar el color emmagatzemat a la imatge final i el fragment d'entrada, i barrejar-los respecte a alguna equació.

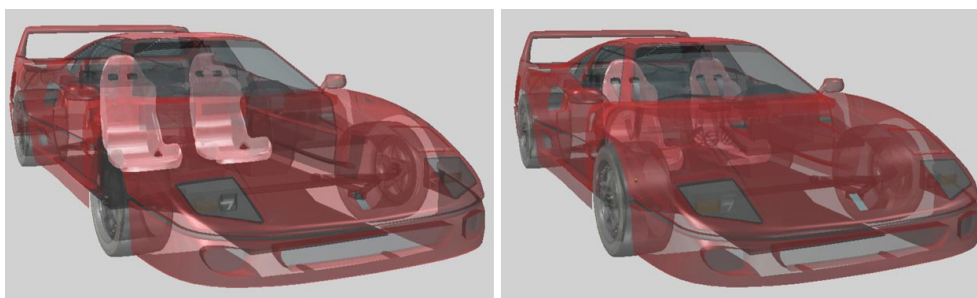


Figura 5.24: Diferència entre no ordenar transparències (esquerra) i ordenar-los (dreta)

Una particularitat d'aquest sistema per a representar materials transparents, és que els objectes representats han de ser ordenats de més llunyà a més proper, i pintar-los després dels opacs, ja que les equacions utilitzades per barrejar la informació de color són dependents de l'ordre. A la Figura 5.24 podem veure les irregularitats visibles si no ordenem correctament les superfícies transparents. Existeixen tècniques per evitar ordenar les transparències, ja que tot i que ordenar a nivell d'objecte és relativament senzill, fer-ho a nivell de triangles és molt costós a nivell de rendiment, i hi ha casos que són impossibles, com la intersecció de dos superfícies transparents. Aquestes tècniques permeten sobreposar tots aquest

problemes, però encara son lleugerament cares per ser utilitzades en la producció de jocs i altres aplicacions d'alt rendiment.

Tornant a la interacció entre llum i superfície, podem veure a la [Figura 5.23](#), que la llum reflectida a la superfície té diferent angle i color, que la llum que ha entrat dins la superfície, ha estat parcialment absorbida per aquesta, i difosa cap enfora de nou. Categoritzem aquest dos tipus utilitzant dos termes diferents a les equacions de shading. Anomenarem **component/terme especular** aquella part de la llum que ha estat reflectida a la superfície, i **component/terme difós** a aquella llum que ha entrat dins la superfície, absorbida i difosa. Podem veure un exemple dels dos termes per separat a la [Figura 5.25](#).



Figura 5.25: Les dos parts de l'il·luminació, terme difós (esquerra) i terme especular (dreta)

### 5.5.3 Shading

El **shading** és el procés d'utilitzar una equació per calcular la il·luminació visible pel receptor, basat en les propietats dels materials dels objectes, i les fonts de llum. És l'aspecte principal del projecte, pel qual veurem tot el necessari per entendre la implementació i la comparativa posteriors.

Com hem comentat quan descrivíem els materials, dividim l'equació de shading en dos termes, el difós i l'especular. Detallarem el terme difós i especular per separat, i després com combinar-los. A efectes de l'actual projecte, utilitzarem una equació de shading senzilla, anomenada **Blinn-Phong**, ja que ens volem centrar en la implementació. Existeixen equacions de shading molt més complexes que la que aquí comentarem. Tant el **Real-Time Rendering** [1] com el **Physically Based Rendering** [21] son excel·lents referències de diferents models de shading.



### Component difós

La part difosa intenta modelar el comportament mat d'un material. Si ens centrem en el model Blinn-Phong, l'equació que modela el component difós, està basada en una llei física anomenada Llei de Lambert, que diu que per superfícies difoses ideals, la quantitat de llum reflectida per un punt d'una superfície, es pot determinar respecte el cosinus de l'angle entre la normal  $\mathbf{n}$  en aquest punt, i el vector definit entre la font de llum i el punt de la superfície, el vector llum  $\mathbf{l}$ . Aquest angle l'anomenarem  $\phi$ . Una forma senzilla i ràpida de calcular l'angle entre dos vectors, és utilitzar el producte escalar:

$$A \cdot B = |A| \times |B| \times \cos \phi \quad (5.1)$$

Utilitzant aquesta propietat, i treballant amb vectors unitaris (la norma dels quals és 1), podem calcular l'angle entre dos vectors amb el producte escalar. Per tant, utilitzant el producte escalar, l'equació per calcular el terme difós és la següent:

$$I_{dif} = \mathbf{n} \cdot \mathbf{l} = \cos \phi \quad (5.2)$$

L'únic detall extra que hem de tenir en compte, és que quan a la normal  $\mathbf{n}$  i al vector llum  $\mathbf{l}$ , els separa un angle  $\phi$  superior a  $\pi/2$ , el valor del cosinus és negatiu. Geomètricament, això vol dir que la superfície no està orientada cap a la llum, cas en el qual no es desitjable que la llum l'afecti, per la qual cosa normalment no es tenen en compte valors negatius, quedant l'equació final:

$$I_{dif} = \max(\mathbf{n} \cdot \mathbf{l}, 0) = \max(\cos \phi, 0) \quad (5.3)$$

Per entendre el motiu de ser rere la llei de Lambert, primer hem d'introduir el concepte de la irradiància. La irradiància és la quantitat d'energia que passa per unitat d'àrea perpendicular a la superfície. Simplificant molt, podem entendre-ho com la quantitat de rajos de llum que ens arriben a una superfície, i podem entendre que estarà directament relacionat amb la quantitat de llum que aquesta superfície podrà reflectir.

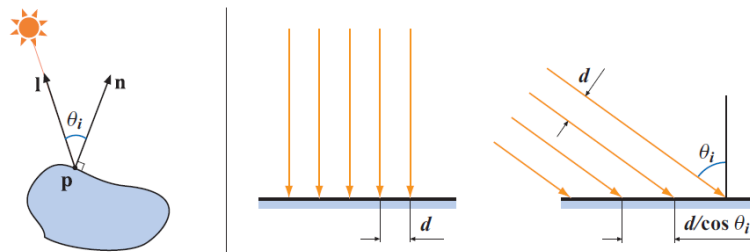


Figura 5.26: La llei de Lambert des d'un punt de vista geomètric

Si ens fixem en la **Figura 5.26**, a la imatge central, podem veure que la font de llum emet fotons que arriben perfectament perpendiculars a la superfície. La separació entre aquest rajos és  $\mathbf{d}$ . A la imatge de la dreta, els rajos arriben a la superfície amb un angle  $\theta$  respecte la normal d'aquesta, i per tant, aplicant trigonometria, podem veure que els rajos ara estan separats una distància  $d/\cos\theta$ , que sempre serà igual o major que la distància  $\mathbf{d}$ . Per tant, quan més increment l'angle de la llum incident amb la normal, major és la distància entre els fotons que arriben a la superfície, i per tant menor la llum que pot arribar a emetre. A més, aquesta és proporcional al cosinus de l'angle entre normal de la superfície i els vectors de la llum incident, cosa que explica la llei de Lambert.

Quan els fotons arriben a una superfície difosa, momentàniament son absorbits per aquesta. Depenent del color del fotó (que suposem el mateix de la font de llum) i el del material de la superfície, pot ser absorbit totalment o reflectit en una direcció aleatòria. A nivell de probabilitats, podem considerar la distribució de la direcció de reflexió com a homogènia, i per tant, igual en totes les direccions. Com que la llum reflectida difosa té igual probabilitat en totes les direccions, és independent del receptor d'aquesta, com ja es podia veure a les equacions, on en cap moment hem introduït cap terme dependent del receptor.

Com hem dit, el color dels fotons incidents i de la superfície tenen un especial paper respecte els que es reflecteixen, cosa que encara no hem introduït a l'equació. Si entenem  $l_{col}$  com el color de la llum, i  $m_{dif}$  com el color difós del material, i  $\otimes$  com la multiplicació per components:

$$I_{dif} = \max(n \cdot l, 0) \times (l_{col} \otimes m_{dif}) \quad (5.4)$$

On  $l_{col} \otimes m_{dif}$  representa el fet que una superfície difosa només reflecteix el fotons del mateix color que el material que la compona.

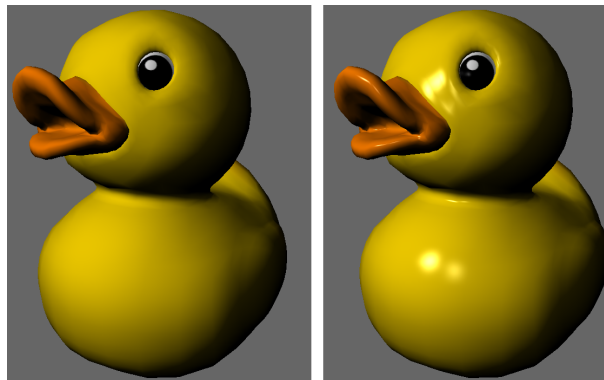


Figura 5.27: Component difós (esquerra) i component difós i especular (dreta)

### Component especular

A diferència del component difós, el component especular és altament direccional i està relacionat amb la posició del receptor, i com es **reflecteix** la llum a la superfície. Aquest component ens permet determinar la posició i direcció de les llums, a més de modelar les parts més lluminoses i punts brillants d'una superfície. Podem veure un exemple sobre la informació visual que aportar el component especular a la [Figura 5.27](#).

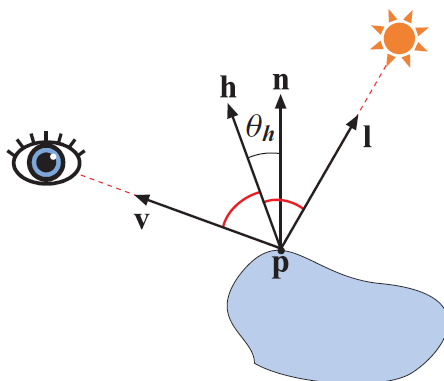


Figura 5.28: Vectors utilitzats per calcular el terme especular

Per a explicar com calcular el terme especular, primer hem d'introduir dos vectors nous, el *vector vista*  $\mathbf{v}$  i el *vector mig*  $\mathbf{h}$ , que podem veure a la [Figura 5.28](#). El vector vista  $\mathbf{v}$  és el vector que va del punt on incideix la llum fins al punt on es troba el receptor. El vector mig  $\mathbf{h}$  és el vector mig normalitzat entre  $\mathbf{l}$  (recordem, vector des del punt d'incidència fins a la llum), i  $\mathbf{v}$ :

$$\mathbf{h} = \frac{\mathbf{l} + \mathbf{v}}{\|\mathbf{l} + \mathbf{v}\|} \quad (5.5)$$

Respecte el qual introduïm el mètode presentat per Blinn [2] per calcular el component especular (que és una variació de l'*equació de shading de Phong* [22], que obviarem explicar):

$$i_{esp} = (\mathbf{n} \cdot \mathbf{h})^{m_{shi}} = (\cos \phi)^{m_{shi}} \quad (5.6)$$

La lògica rere aquesta equació és que el vector  $\mathbf{h}$  és la normal del pla que passa pel punt d'incidència, tal que el  $\mathbf{v}$  es reflectiria perfectament en el vector  $\mathbf{l}$ . Si tenim en compte que el terme especular modela la quantitat de llum reflectida a la superfície, podem veure que com més alineat estigui el vector  $\mathbf{h}$  amb la normal  $\mathbf{n}$  al punt d'incidència, més indica que la llum es reflecteix de la font de llum al receptor. Aquesta alineació és senzillament el cosinus de l'angle entre els vectors  $\mathbf{n}$  i  $\mathbf{h}$ , seguint la mateixa lògica que en el cas difós, que, recordem, podem calcular utilitzant el producte escalar. La popularitat de l'equació de

Blinn respecte l'original de Phong és degut a que aquesta calculava el vector  $\mathbf{l}$  reflectit, que és més intuïtiu, però també és més car de calcular.

De forma paral·lela al cas difós, hi ha casos en que el cosinus del *vector mig* i la normal serà negatiu, que geomètricament significa que la llum no està orientada a la superfície, i per tant no tenim en compte el terme especular.

$$i_{esp} = \max(n \cdot h, 0)^{m_{shi}} \quad (5.7)$$

Respecte a la qual ens queda explicar el terme  $m_{shi}$ . Aquest paràmetre descriu com de brillant és la superfície. Valors més grans volen dir que concentren més la brillantor, mentre que valors menors la difonen. A efectes pràctics, aquest paràmetre té poca validesa física, només es tracta d'una aproximació ràpida a les diferents mides de les reflexions especulars. Podem veure un exemple de diferents valors d'aquest paràmetre a la [Figura 5.29](#).

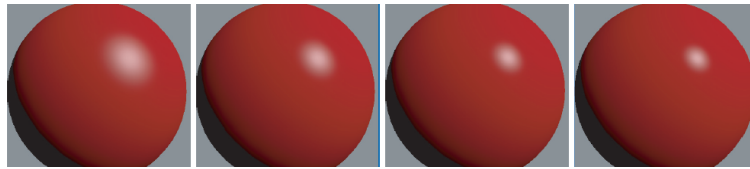


Figura 5.29: Diferents valors de la potencia especular de menor (esquerra) a major (dreta)

Finalment, i també de forma anàloga al terme difós, només els fotons amb el mateix color que la superfície es reflecteixen en ella, i per tant introduïrem dos termes més a l'equació per emular aquest comportament. Si entenem  $l_{col}$  com el color de la llum, i  $m_{esp}$  com el color especular del material, llavors l'equació final és:

$$i_{esp} = (\max(n \cdot h, 0)^{m_{shi}}) \times (l_{col} \otimes m_{dif}) \quad (5.8)$$

#### 5.5.4 Equació de shading

Fins ara hem detallat com calcular parts de l'equació de shading, ara anem a veure com calcularíem l'equació de shading completa, respecte als components que hem anat explicant. Aquesta equació determinarà com les llums d'una escena interaccionen amb els materials de les superfícies d'aquesta, i en determinaran (parcialment) el color final dels pixels corresponents a l'espai que ocupa cada objecte.

Explicarem un model d'il·luminació local, que vol dir que tota la il·luminació només depèn de les fonts de llum, no de la llum que prové d'altres superfícies. Per tal d'emular la llum que al món real prové d'altres superfícies, introduïrem un terme anomenat color ambient  $c_{amb}$ , que és un color constant per a tota la

escena. Hi ha aproximacions a l'equació de shading que també tenen paràmetres als materials per modular aquest valor, però creiem que amb aquest paràmetre és suficient, doncs tinguem en compte que son aproximacions molt crues a la realitat.

Respecte com funciona la llum i els termes explicats, és pot veure que el primer pas per calcular la il·luminació total, és sumar els termes difós i especular, en un terme que anomenarem il·luminació total, o  $i_{tot}$ :

$$i_{tot} = i_{dif} + i_{esp} \quad (5.9)$$

La majoria de bibliografia introdueix també un terme ambient per llum, però hem optat per obviar-lo, ja que considerem que amb l'aproximació d'un color ambient per escena ja és suficient.

Un cop hem calculat la il·luminació total per llum, cal tenir en compte l'atenuació que sofreix la llum respecte a la distància que recorre, que encara no hem introduït a l'equació. Al món real, la intensitat de la llum és inversament proporcional al quadrat de la distància que separa la llum amb la superfície que la rep. Només tenim en compte aquest tipus d'atenuació per llums posicionals, és a dir, de tipus puntual i de tipus focus, doncs les llums direccionals no tenen una posició respecte a la que atenuar, i a més, és considera que la atenuació que reben es pot descartar, doncs principalment modelen llums de gran potencia.

Si entenem  $l_{pos}$  com la posició de la llum, i  $\mathbf{p}$  com el punt on incideix la llum a la superfície, la atenuació és:

$$d_{atten} = \frac{1}{\|l_{pos} - p\|^2} \quad (5.10)$$

Modificant l'equació 5.9 per tenir en compte aquest terme, obtenim:

$$i_{tot} = d_{atten} \times (i_{dif} + i_{esp}) \quad (5.11)$$

Que seria la equació d'il·luminació per una llum puntual. Per a suportar llums de tipus focus, volem poder restringir on afecta la il·luminació, per tal que només la part interior del con sigui afectada per la llum. Com hem dit abans les llums de tipus focus tenen una direcció  $i_{dir}$ , que indica a on apunta, i els cosinus d'un angle d'obertura,  $i_{angcos}$ . Podem modelar el factor d'una llum de tipus focus tal que:

$$f_{fact} = \begin{cases} \max(i_{dir} \cdot -l, 0) & \text{si } (i_{dir} \cdot -l) < i_{angcos} \\ 0 & \text{altrament} \end{cases} \quad (5.12)$$

Amb la qual cosa, l'equació d'il·luminació per una llum de tipus focus seria:

$$i_{tot} = f_{fact} \times d_{atten} \times (i_{dif} + i_{esp}) \quad (5.13)$$

Podríem unificar els dos tipus de llums puntuals, afegint un tercer terme a la equació 5.12, que igualés  $f_{fact}$  a 1 en el cas que la llum no fos de tipus focus, però ho obviarem, doncs és senzill tractar els dos tipus de llums per separat, i ens dóna un petit avantatge de rendiment. Altres opcions que podríem afegir a les llums de tipus focus, és un paràmetre que controli la atenuació radial d'aquesta, que resulta interessant per evitar el tall dur que si no generarien.

Fins ara sempre hem tingut en compte la il·luminació aportada per una única llum, però una superfície pot ser afectada per més d'una llum. La il·luminació de diverses llums es comporta de forma additiva, per la qual cosa, si tenim  $n_{llums}$  llums afectant a un objecte, la seva il·luminació és pot calcular tal que:

$$i_{obj} = \sum_{k=1}^{n_{llums}} i_{tot}^k \quad (5.14)$$

On  $i_{tot}$  és la il·luminació total per cada tipus de llum. Si separem  $n_{llums}$  en els dos tipus de llums posicionals, nombre de llums puntuals  $n_{pnt}$  i nombre de llums focus  $n_{foc}$ , i expandim els termes anteriors, obtenim:

$$i_{obj} = \sum_{k=1}^{n_{foc}} (d_{atten}^k \times (i_{dif}^k + i_{esp}^k)) + \sum_{k=1}^{n_{pnt}} (f_{fact}^k \times d_{atten}^k \times (i_{dif}^k + i_{edp}^k)) \quad (5.15)$$

Finalment, com hem comentat, afegirem un terme que emula els rebots de llum a les superfícies que componen l'escena,  $c_{amb}$ , que és un color que aproxima cruament aquest fenomen. Com hem dit, és un terme global que no està afectat per els paràmetres dels objectes ni les llums, per la qual cosa s'afegeix a part dels sumatoris:

$$i_{obj} = c_{amb} + \sum_{k=1}^{n_{foc}} (d_{atten}^k \times (i_{dif}^k + i_{esp}^k)) + \sum_{k=1}^{n_{pnt}} (f_{fact}^k \times d_{atten}^k \times (i_{dif}^k + i_{edp}^k)) \quad (5.16)$$

Amb aquesta última ampliació, ja tenim definida l'equació de shading completa per objecte.

## Capítol 6

---

# Requisits del sistema

---

Aquest projecte té l'intenció d'assolir dos objectius clars: la implementació de certs algorismes de shading i comparar-los respecte a una serie de criteris. Aquesta és la visió a més alt nivell dels objectius del projecte que podem donar, però necessitem definir de forma molt més concreta aquests dos objectius per entendre els requisits per considerar-los complets. Així doncs, anem a veure els dos objectius per separat, detallar-los i extreure'n una llista de requisits per considerar-los.

### 6.1 Pre-requisits

D'acord amb el capítol 4, podem veure que la implementació d'algorismes de shading té dos passos previs, la càrrega de dades i la creació d'un sistema per moure la càmera per l'escena. Abans de veure els requisits de la implementació d'algorismes de shading, anem a veure aquests dos pre-requisits.

#### 6.1.1 Càrrega de dades

Volem un sistema de càrrega de dades que carregui un format de dades estàndard, que pugui llegir diferents tipus d'elements (objectes, llums, càmeres, jerarquies, animacions, etc), converteixi les dades a un format fàcilment dibuixable, i que sigui ràpid. Anem a veure cada punt per separat, i quins requisits considerem que han de complir per ser complets:

- **Càrrega d'un format estàndard:** Volem poder carregar un format que no sigui creat per nosaltres, si no que ja existeixin diverses aplicacions amb la capacitat per crear-lo. Volem que aquest format suporti com a mínim objectes i llums, que son el mínim que necessitem. Així doncs, els

requisits son llegir un format 3D, que diverses aplicacions 3D siguin capaces d'exportar, i que pugui descriure objectes i llums.

- **Lectura de diferents tipus d'elements:** Si carreguem un format estàndard és molt possible que suporti molts més elements dels que realment necessitem pel projecte. Considerarem que el carregador està complet si pot llegir objectes, llums i com es disposen en l'espai. També és altament desitjable poder llegir animacions i càmeres, per poder fer tests més complets i automatitzats. Els requisits son poder llegir objectes, llums i la seva disposició. Com a extra, és interessant llegir càmeres i animacions.
- **Conversió de dades:** Si llegim un format estàndard, es pot suposar que les dades no estan dissenyades per pintar-se de forma òptima, doncs normalment son formats d'intercanvi de dades entre diferents aplicacions de modelat 3D. Volem poder convertir les dades a un format optim per dibuixar. Això vol dir eliminar dades supèrflues, reordenar dades, o convertir-les a formats més adequats pel hardware. Els requisits mínims son obtenir dades que es puguin dibuixar, però amb extres molt interessants com eliminar dades redundants o conversió d'aquestes a millors formats.
- **Càrrega ràpida:** Durant el desenvolupament del projecte, el procés de càrrega de dades l'haurem de repetir moltes vegades, i pot ser que inclogui processos extremadament complexos, especialment a l'apartat de conversió de dades. Volem que la càrrega trigui el menys possible, per la qual cosa s'ha de dedicar temps a fer-ho possible. Això pot incloure optimitzar tota la part de càrrega de dades de disc, la conversió de dades, i desar dades ja processades, si el procés en si mateix és massa lent per fer-ho cada vegada. Els requisits son mantenir la càrrega i conversió de dades dins d'uns marges raonables, és a dir, trigar 15-30 segons en un ordinador de gamma mitja és acceptable, trigar 15 minuts no. Si el procés triga més, requeriria poder desar les dades processades i evitar haver-les de tornar a processar en successives execucions.

### 6.1.2 Sistema de càmera

Els requisits del sistema de càmera son senzills:

- **Llegir les entrades d'usuari i moure la càmera:** Volem un sistema que, respecte a les entrades d'usuari (ratolí o teclat), mogui una càmera en un espai 3D. Els requisits és que el sistema permeti moure's per tot l'espai 3D, així com rotar la càmera en qualsevol direcció. Volem que la càmera es mogui en la direcció que està orientada, per tal que la navegació sigui intuïtiva.



- **Generar una descripció de la càmera:** El client de les dades d'aquest sistema serà el motor 3D, així que volem oferir una descripció compacta de la posició i rotació de la càmera, i quines propietats de projecció té. Respecte al vist anteriorment, requerim que la càmera ens permeti consultar les matrius de càmera i projecció.

## 6.2 Implementació d'algorismes de shading

Amb els pre-requisits clars, podem idear els requisits pel sistema més complexe de tot el projecte. Cal entendre, que respecte a la metodologia utilitzada, els pre-requisits no cal que es compleixin totalment abans de començar aquesta part. Per exemple, hem acceptat començar a treballar en els algorismes de shading sense fer que la càrrega fos ràpida, o sense optimitzar les dades totalment, per així poder modelar una **vertical slice** de tota l'aplicació, cosa que valoro per damunt de seguir els requisits en un ordre estricte.

Segons les característiques que hem vist al capítol 4, anem una a una definint que requerirem de cada una d'elles per considerar-les completes. Seguint la llista de característiques:

- **Abstracció de la llibreria gràfica:** Volem crear una abstracció de la llibreria gràfica. Això vol dir que no utilitzarem directament la llibreria en qüestió, sinó una capa per damunt d'aquesta, que amaga molts detalls. El raonament rere a això, és que si la llibreria gràfica esdevé obsoleta durant el desenvolupament del projecte, la podem canviar per una altra sense que la transició sigui extremadament complexa. A més ens permet modelar l'accés al hardware 3D d'una forma bastant genèrica, sense les particularitats de cap llibreria, cosa que ens permet veure els punts forts i febles de cada una. Per tal de considerar aquesta tasca completa, volem una abstracció que ens permeti fer tot el que faríem amb una llibreria gràfica directament, sense que en cap moment sigui possible saber amb quina estem treballant.
- **Implementació de l'abstracció gràfica amb una llibreria gràfica:** Un cop creada una abstracció de la llibreria gràfica, volem implementar una versió d'aquesta. Això vol dir implementar tota la funcionalitat que l'abstracció ofereix de forma concreta, utilitzant una de les APIs del mercat. Els requisits d'aquesta tasca és concretar tota la funcionalitat que ofereix la abstracció amb una llibreria (com per exemple, Direct3D9).
- **Implementació d'un sistema de depuració gràfica:** Durant el desenvolupament del motor 3D, cometrem multitud d'errades de programació o de concepte que costaran d'entendre. Per tal d'ajudar-nos a la programació, ja tenim la eina de depuració associada al compilador, però per entendre problemes gràfics necessitem eines de depuració gràfica, com per exemple

un mode per veure l'escena només respecte a les seves arestes, poder veure els volums de les llums, o poder activar certs comptadors de depuració. Considerarem aquesta tasca completa quan tinguem suport per veure els objectes només respecte les seves arestes, puguem recarregar shaders sense reiniciar l'aplicació, i mostri certs comptadors per pantalla, com els nombre de triangles i vèrtexs dibuixats.

- **Implementació d'un sistema per utilitzar les dades provinents del mòdul de càrrega:** Per tal de manegar les dades provinents del mòdul de càrrega, crearem una serie d'abstraccions per a cada un dels elements, com objectes, llums, càmeres i la descripció general de l'escena. Aquest sistema ens ha de permetre llistar els elements d'una escena, preguntar-n'he propietats, i afegir o treure elements. Considerarem aquest sistema complet quan puguem manegar totes les dades que el carregador ens pot oferir, és a dir, com a mínim objectes i llums, però idealment també càmeres i animacions.
- **Implementació dels algorismes no relacionats amb shading:** Hi ha algorismes que no son dependents del tipus de shading que es vulgui utilitzar, com per exemple ordenar els objectes respecte certs criteris, o determinar els objectes no visibles. El considerarem complet si ens permet determinar quins objectes son visibles, de forma ràpida.
- **Implementació dels algorismes de shading:** La part més complexa de tot el treball. Volem implementar quatre algorismes de shading diferents, intentant que el resultat visual no difereixi en excés, sempre tenint en compte les particularitats de cada algorisme de shading, cada un d'ells suportant la mateixa quantitat de característiques, com per exemple, els mateixos tipus de llums o efectes per pixel. A part, volem iterar en cada algorisme per tal d'optimitzar-ho el millor possible, per tal que la posterior comparativa tingui sentit. Així, doncs, per considerar aquesta tasca completa, hem d'haver implementat quatre algorismes de shading amb el mateix nivell de característiques (que concretarem més endavant), amb una diferencia visual entre ells dins dels paràmetres acceptables de les particularitats de cada algorisme, i amb una implementació el més òptima possible.

Un cop complerts aquests requisits, tindrem un motor 3D amb suport de quatre algorismes de shading, amb diferències menors de qualitat entre ells, que ens permet carregar diferents escenes i moure's per elles utilitzant els dispositius d'entrada del PC

### 6.3 Comparació d'algorismes de shading

En aquest punt, i complerts els requisits, queda la part final del treball, que és crear una comparativa dels diferents algorismes. Volem que la comparativa compregui tres elements: rendiment, qualitat visual i dificultat d'implementació. Per poder fer la comparació amb aquests tres criteris, necessitem certes funcionalitats implementades al codi, que crearan les dades que podrem comparar:

- **Poder guardar dades de rendiment gràfic:** Volem poder comparar el rendiment de cada tècnica utilitzant una serie de característiques, com per exemple el que triga a dibuixar-se una imatge o com està utilitzant les diferents unitats de la GPU. Per considerar aquesta característica completa, hem de decidir i poder recollir aquests valors, guardant-los a disc per comparar-los posteriorment.
- **Poder guardar comptadors d'elements del mòdul de dibuixat:** Volem saber quins elements componen la imatge en cada moment, per a poder identificar els punts forts i febles de cada tècnica. Per exemple, pot ser que un algorisme funcioni millor quan dibuixem molts objectes, però empitjori si afegim moltes llums. Per tal efecte, considerarem aquesta característica acabada quan puguem desar la quantitat de llums, objectes i triangles, a disc, per després utilitzar-los per extrapolar comportaments de les diferents tècniques.
- **Poder capturar la imatge generada pel mòdul de dibuixat:** Volem saber la qualitat visual que dona cada tècnica, i per això hem d'analitzar les imatges generades per cada algorisme. Necessitem, per tant, poder desar aquestes imatges en un moment concret. Considerarem completa aquesta tasca quan puguem guardar les imatges generades a disc, per comparar-les després utilitzant qualsevol software d'edició d'imatges.

Un cop complertes aquestes tasques, disposarem d'informació suficient per a comparar les quatre tècniques respecte als criteris que hem esmentat anteriorment. Cal esmentar que la comparativa de qualitat visual i la de dificultat d'implementació es basaran també en el coneixement adquirit durant la realització del treball, especialment la segona. La qualitat visual de cada tècnica pot anar implícitament lligada a aquesta, doncs a vegades una tècnica intercanvia una millora de rendiment notable a canvi de reduir la qualitat visual lleugerament. Respecte a la comparativa de dificultat d'implementació, ho farem de força relativa a les altres tècniques.

## Capítol 7

---

# Estudis i decisions

---

Anem a descriure quin maquinari, llibreries i programari hem necessitat per portar a terme el projecte.

### 7.1 Maquinari

A nivell de hardware, hem utilitzat un PC que ara es podria considerar de gamma mitja, doncs és un equip que ara mateix té 3 anys, i varis milers d'hores d'ús. Anem a detallar-ne els components rellevants per a portar a terme el treball, i perquè els considerem rellevants:

- **CPU:** La CPU de l'equip és un *Intel Core 2 Duo a 2.4Ghz*. Per compilar codi i processar dades és important una CPU relativament potent. En aquest cas, aquesta CPU és una mica justa, considerant el que ofereix el mercat avui en dia. Igualment, tenint en compte que el motor 3D que desenvoluparem es mantindrà per sota de les 50000 línies de codi, no hauria de ser gens problemàtic re-compilar tot el codi. Un avantatge d'utilitzar una CPU de gamma mitja-baixa és que es persegueix en tot moment un codi net i òptim, doncs ens interessa que el projecte funcioni de forma fluida en aquest equip.
- **GPU:** La GPU d'aquest equip és una *NVidia Geforce GT 130M*. Per a executar els shaders que implementen els algorismes de shading és important tenir una GPU dedicada, doncs tot i que existeixen GPUs integrades al mateix silici de la CPU, aquestes ofereixen prestacions reduïdes. Aquesta GPU és adequada pel projecte que portem a terme, tot i ser lleugerament antiga, doncs els algorismes que implementarem es podran executar amb aquesta, amb certa fluïdesa. Si haguéssim d'incloure ombres dinàmiques o

post-processos pot ser que trobéssim problemes de rendiment, però no és el cas.

- **RAM:** Aquest equip disposa de 4 GB de RAM. Això és suficient per executar el sistema operatiu, el software que ens permet compilar codi, i l'aplicació que estem desenvolupant. En cap moment la quantitat de RAM ha estat problemàtica, ni en el moment de carregar les escenes de major mida.

Així, doncs, l'equip que teníem disponible cobria les necessitats per a desenvolupar i executar el projecte, de fet excedien el que considerariem el mínim per poder executar l'aplicació.

## 7.2 Llibreries

Discutir les llibreries utilitzades és una de les parts que considero més interessants del projecte, doncs hem decidit utilitzar algunes llibreries desconegudes per mi abans de començar el projecte.

### 7.2.1 Llibreries gràfiques

Per a utilitzar la targeta gràfica 3D necessitem utilitzar una llibreria, doncs dóna una capa d'abstracció entre l'aplicació i el hardware mateix. Si aquesta no existís, hauríem d'adaptar l'aplicació a cada targeta del mercat que volguéssim suportar, una feina titànica. En aquest cas no podíem decidir si utilitzar o no una llibreria, si no quina llibreria utilitzar. Si ens centrem en PC, tenim dos eleccions:

- **OpenGL** [26] [10]: És una llibreria multi-plataforma creada al gener de 1992, que ha rebut revisions periòdiques des de llavors. Utilitza un mecanisme d'extensions per tal de poder-se ampliar entre cada una de les revisions. Les revisions actualment són discutides i aprovades per un comitè de diverses empreses, anomenat Khronos Group, que vetllen per la creació d'estàndards oberts. S'utilitza en PCs, MACs i mòbils, entre altres dispositius.
- **Direct3D:** És una llibreria ideada per funcionar amb el sistema operatiu Windows, de la companyia Microsoft. També és la llibreria gràfica utilitzada per les consoles XBox i XBox360, i els mòbils amb el sistema operatiu Windows Phone. Rep revisions periòdiques, però no té un mecanisme d'extensions, si que no requereix certes funcionalitats base per tal de funcionar, amb la qual cosa el cicle de revisions sol ser més curt que el d'OpenGL.

Fins el moment de començar el projecte, només havia utilitzat extensivament **OpenGL**. Pel projecte vaig decidir utilitzar Direct3D. La principal motivació per

utilitzar-ho és aprendre, però també hi ha altres motius pels quals vaig creure adequat utilitzar-la. Com desenvolupo utilitzant principalment sistemes operatius de Microsoft, a nivell personal me's indiferent utilitzar OpenGL o Direct3D, doncs les dos funcionen correctament al meu equip habitual. A nivell professional, hi ha una divisió clara en el món del desenvolupament de video-jocs: per a mòbils s'utilitza una versió reduïda d'OpenGL, anomenada **OpenGL ES**, mentre que per PC la majoria de companyies utilitzen Direct3D. Tenint en compte que les dues llibreries ofereixen una funcionalitat relativament semblant, però amb filosofies d'ús diferents, creia molt interessant utilitzar Direct3D, amb un marge de risc relativament baix. A més, per tal de reduir més el risc, com hem dit al capítol anterior, vaig crear una capa d'abstracció, per poder canviar de llibreria de forma senzilla.

### 7.2.2 Llibreries d'imatges

Els formats amb el qual es desen habitualment les imatges a disc son força complexos. Acostumen a barrejar una bona quantitat d'algorismes de compressió, i sistemes de reducció de detalls menys significatius. Escriure un carregador per un format d'imatge com JPG o PNG és una feina complexa, i suportar totes les possibilitats que ofereix el format suposa una quantitat de feina considerable.

A part, les targetes 3D suporten formats d'imatge comprimida diferents als que estem acostumats com a usuaris del PC. Son formats amb menys compressió, però amb certes propietats que els fan ideals per utilitzar-los en textures i es poden implementar fàcilment al hardware. El principal avantatge que ofereixen és que ocupen menys en memòria, amb la qual cosa podem utilitzar més textures, gastant menys ample de banda i menys memòria de la targeta.

Tenim, doncs, dos necessitats. Volem poder llegir un ventall ampli de tipus d'imatges i volem poder comprimir-les a formats que les GPUs puguin utilitzar:

- **Llegir imatges:** Primer cal notar que Direct3D té una llibreria d'utilitats anomenada D3DX que té funcionalitat per carregar un ampli ventall d'imatges. Per totes les necessitat que no cobreix, hem decidit utilitzar la llibreria OpenIL (també anomenada DevIL) [28]. Aquesta llibreria ofereix una interfície C molt senzilla, modelada per ser molt semblant a OpenGL. L'he decidit utilitzar perquè ja la coneixia i suporta càrrega, conversió i guardat d'imatges. A més, és una llibreria open-source amb una llicència força liberal, pel qual no suposa cap cost utilitzar-la.
- **Comprimir imatges:** Per a comprimir imatges, volíem integrar una llibreria i no dependre d'una utilitat externa per comprimir les imatges. No disposem de moltes opcions, però la millor opció és la NVidia Texture Tools (NVTT) [4]. És un conjunt de llibreries amb una llicència molt poc restrictiva, creada per NVidia i utilitzada per moltes companyies. A més, té una

interfície molt senzilla d'utilitzar amb diferents modes de qualitat, la qual cosa ens permet fer les proves amb el mode de baixa qualitat (que va molt ràpid).

Amb aquestes dos llibreries cobríem totes les necessitats relacionades amb imatges. Respecte la càrrega d'imatges podríem haver optat per limitar-nos a un tipus concret d'imatge, el carregador del qual fos fàcil d'implementar (per exemple, el format TGA o BMP és molt fàcil de llegir), però la càrrega d'imatges és el tipus de cas en el que és interessant utilitzar llibreries, doncs normalment els format tenen moltes peculiaritats difícils de suportar.

Respecte la compressió d'imatges, la podríem haver ignorat completament, però si volem el màxim rendiment és quelcom molt important. Un dels grans colls de botella dels gràfics 3D és l'ample de banda, així que reduir la càrrega sempre és important. A més, la memòria de les targetes 3D és molt cara, amb la qual cosa acostumen a tenir mides molt més reduïdes que les memòries principals dels PCs, per tant és interessant conèixer mecanismes per aprofitar-la millor.

### 7.2.3 Altres llibreries

Hem utilitzat dos llibreries més, la primera perquè ens facilitava moltíssima una tasca feixuga, i una altra perquè ens donava accés a dades que d'altra manera no podríem conèixer.

#### Decodificador de XML

El format de dades 3D que hem elegit és el estàndard Collada [8]. Collada és un format 3D que descriu les dades utilitzant la codificació XML. La codificació XML és un serie de regles per descriure informació utilitzant un format de text, en podem veure un exemple a la [Figura 7.1](#).

```
<config>
  <windowSize width="1280" height="700" />
  <fullscreen on="false" />
</config>
```

Figura 7.1: Exemple senzill del format XML

Un dels objectius del projecte era carregar dades 3D des d'un format estàndard, però no volíem haver de desenvolupar el codi que entengués la codificació XML, doncs hauria incrementat molt el temps necessari per acabar el projecte: escriure un lector de XML robust és molt complexe.

En comptes d'això, vaig decidir utilitzar una llibreria que llegís XML, i a sobre d'aquesta llibreria programar el lector de Collada. Primer havia considerat la

llibreria **TinyXML** [27], que és una llibreria que ja coneixia, open-source i amb una llicència molt poc restrictiva. Utilitzant TinyXML eliminava el temps d'aprenentatge necessari, però al final vaig decidir utilitzar la llibreria **pugixml** [14], ja que també és open-source, amb una llicència poc restrictiva, i una interfície senzilla d'utilitzar. La decisió de canviar, la va motivar que segons les comparatives, era ordres de magnitud més ràpida que TinyXML, i que volia provar una llibreria de lectura XML diferent, doncs aquesta última era la única que coneixia.

### Lector de comptadors hardware

Una de les característiques que hem enumerat com necessària per considerar el projecte com a completat, és poder guardar dades de rendiment. La quantitat de mesures relacionades amb el rendiment que podem calcular, sense un sistema d'introspecció de la targeta, és limitat. Com que la majoria de la pipeline està implementada dins la GPU, sense informació de rendiment d'aquestes fases és molt difícil saber quins son els colls de botella, els avantatges i els desavantatges de cada algorisme de shading.

Per sort, NVidia ofereix dos solucions per tal de conèixer la càrrega de treball de les diferents fases de la pipeline implementada a la GPU. La solució més antiga que ofereixen és el **NVidia PerfKit** [17], que cobreix perfectament les necessitats del projecte: és gratuïta, molt senzilla d'utilitzar i amb bona documentació. L'únic inconvenient que té és que NVidia ja no en crea noves versions, doncs intenten que la gent utilitzi la nova llibreria **NSight**, però té requisits de hardware bastant elevats, i és molt més complexa de fer servir. Igualment, l'última versió que es pot descarregar ofereix tota la informació que necessitava.

## 7.3 Programari

Anem a llistar el programari utilitzat, junt amb l'ús que li hem donat i perquè hem decidit utilitzar-lo:

- **Microsoft Windows 7**: És un sistema operatiu molt estable.
- **Microsoft Visual Studio 2010**: Entorn integrat que permet editar codi C++, compilar-lo, enllaçar-lo i depurar-lo (**Figura 7.2**). Personalment crec que és el millor entorn integrat per desenvolupar codi, l'editor és ràpid i versàtil, el compilador genera bons executables i el depurador té una potencia espectacular. A més, és gratuït per estudiants.
- **Notepad++**: Un editor de text gratuït amb suport de remarcar sintaxi de diversos llenguatges, suport per macros, plug-ins (té un editor hexadecimal excel·lent) i tot allò que esperaríem d'un editor modern. És l'aplicació que



s'ha utilitzat per editar els shaders o editar fitxers Collada, i en general per qualsevol cosa que no fos codi C++.

- **Autodesk 3dsMax / Autodesk Maya:** Vam utilitzar llicències gratuïtes per estudiants d'aquestes aplicacions (que ara ja no son disponibles) per exportar les dades del format natiu a Collada. Vaig utilitzar aquestes dos aplicacions perquè son les que conec més, i oferien llicències gratuïtes per estudiants.
- **IrfanView:** Visor d'imatges, amb suport per carregar i guardar diferents formats. És gratuït per ús no comercial i extremadament àgil, comparat amb altres visors d'imatges.
- **Google SketchUp:** Aquesta aplicació de modelat gratuïta de Google permet carregar arxius Collada, i l'he utilitzat per comparar els resultats del meu carregador de Collada amb el d'aquesta aplicació. És força lent carregant arxius grans, però compleix molt bé l'estàndard Collada, amb la qual cosa és molt útil.
- **OpenOffice Calc:** És una aplicació per treballar amb fulles de càlcul. Ens servirà per analitzar les dades estadístiques. És gratuït i ofereix totes les prestacions necessàries per l'anàlisi que hem de fer.

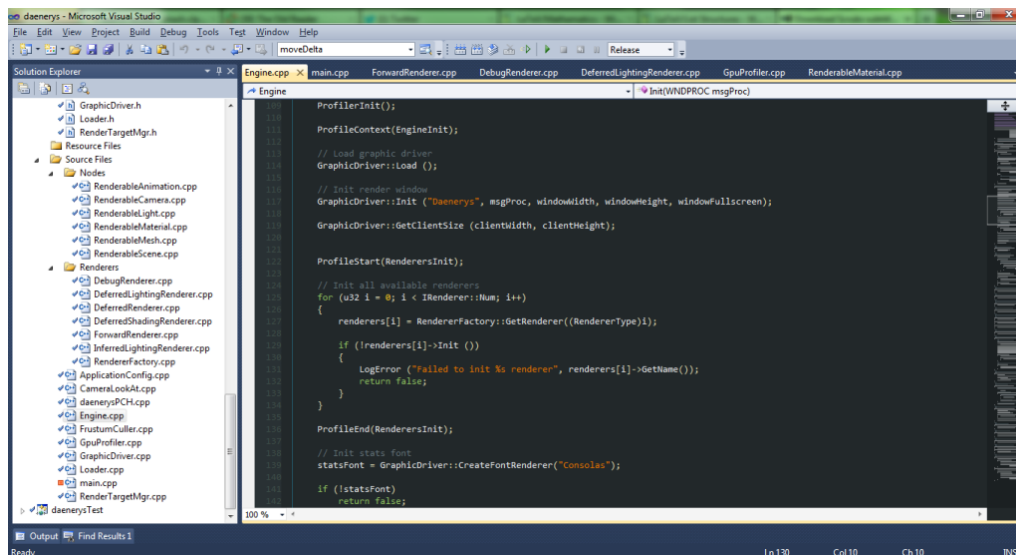


Figura 7.2: Projecte Final de Carrera obert al Visual Studio 2010

## Capítol 8

---

# Anàlisi i disseny del sistema

---

Com hem vist al capítol 3, hem utilitzat una metodologia àgil per desenvolupar el projecte. Dins les metodologies àgils s'intenta reduir la quantitat de planificació prèvia, i analitzar les tasques una a una. A nivell d'anàlisi i disseny del sistema, això ens ha suposat esquematitzar el projecte a molt alt nivell, sobretot tenint clar els punts de comunicació entre els quatre mòduls definits al capítol 4. Tant l'anàlisi com el disseny, intenten donar una visió senzilla i lleugerament idealitzada del sistema, per tal de poder entendre com m'he enfrontat al problema en qüestió, però en cap moment intenten definir les classes al detall ni tots els punts de comunicació entre elles: crec fermament que el disseny amb massa detall acaba essent sempre erroni, doncs hi ha centenars de detalls molt difícils de preveure en aquesta fase, cosa que concorda amb la metodologia àgil.

### 8.1 Anàlisi

Com ja hem vist en capítols anterior, el nostre sistema es pot dividir en quatre mòduls lògics diferents: càrrega de dades, sistema de càmera, motor 3D i sistema de comparativa.

El primer que hem d'analitzar són les necessitats de cada sistema, anàlisi molt semblant al que hem portat a terme al capítol 4, per la qual cosa el mantindrem concís:

- **Càrrega de dades:** Necessitem poder llegir dades d'un format codificat en XML, processar-les perquè siguin òptimament dibuixables i que el procés sigui ràpid.
- **Sistema de càmera:** Necessitem un sistema que generi informació de càmera respecte a les entrades d'usuari. No ens hauria d'importar com son

aquestes entrades ni com funciona el sistema, però hauria d'oferir informació concisa de com està configurada la càmera respecte a les accions de l'usuari.

- **Motor 3D:** Volem una abstracció de la llibreria gràfica, de manera que sigui indiferent quina utilitzem. Volem poder utilitzar les dades del carregador i del sistema de càmera, per a definir l'escena. Volem que tot allò no relacionat amb els diferents algorismes de shading estigui unificat, i que la quantitat de punts de comunicació amb els algorismes de shading sigui mínima.
- **Sistema de comparativa:** Aquest sistema ha d'obtenir les dades del hardware i del motor 3D. Ha d'actuar de manera no intrusiva al motor 3D, funcionant com un client d'aquest i no dins d'aquest.

A alt nivell he modelat l'aplicació utilitzant el flux de la [Figura 8.1](#).

## 8.2 Disseny

En quant al disseny, primer ens centrarem en les dades. Les dades d'entrada no cal dissenyar-les, doncs utilitzem un format anomenat Collada [8] per la descripció d'escenes, i formats ja existents per les imatges. Respecte el flux que es pot veure a la [Figura 8.1](#), només hem de decidir en quin format desaré a disc les dades estadístiques.

Al capítol 7 he inclòs una aplicació per editar fulles de càlcul, i deia que la faria servir per analitzar les dades estadístiques. Així doncs, he de desar les dades en un format que em sigui fàcil de carregar a la fulla de càlcul. El format més senzill amb aquests requisits és l'anomenat **CSV, Comma Separated Values**. El CSV és un format de text on els valors es separen per comes. És un format molt senzill de crear, i per tant, ideal pel projecte.

Respecte el disseny de l'aplicació, vaig fer un modelat de classes a alt nivell, amb els mètodes que vaig creure necessaris ([Figura 8.2](#)).

Aquest disseny té 7 classes principals, que s'ocupen de la càmera, de la descripció d'escena del carregador i del motor 3D, els algorismes de shading, l'abstracció de la llibreria gràfica, el sistema de comparatives i el motor, que ho uneix tot.

La classe **Engine** és la que s'ocuparà d'unir tota la funcionalitat. És el que fins ara enteníem com a motor 3D, tot i que també és l'element que fa de nexa a altres sistemes.

La classe **CameraLookAt** implementarà un sistema per poder moure la càmera respecte les entrades d'usuari. En principi, només la utilitzarà la classe **Engine**, i serà aquesta última qui li demanarà informació sobre el seu estat.

Les classes **ColladaScene** i **RenderableScene** contenen diferents representacions de les dades 3D. La primera té les dades en un format molt semblant a com son desades en el format Collada, amb la diferencia que les dades ja han estat

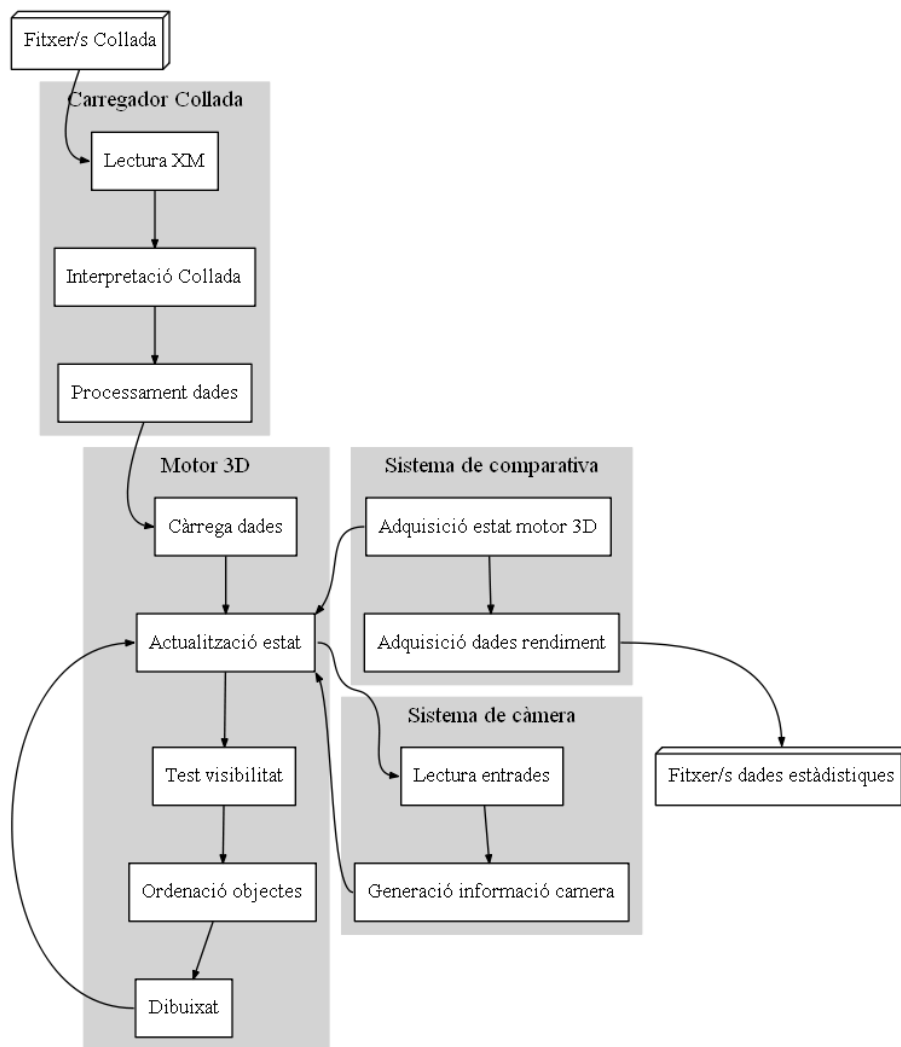


Figura 8.1: Flux de l'aplicació a alt nivell

processades per poder-se dibuixar. La segona és una representació de les dades que ens permet dibuixar-les de forma immediata, doncs conté recursos carregats al hardware. La diferència principal entre les dos classes és el tipus d'accés que ens dóna a les dades, i on resideixen aquestes.

La classe **IRenderer** és una interfície comú a tots els algorismes de shading, que com podem veure, és molt senzilla. Com ja hem dit en capítols anteriors, és molt important que les implementacions dels algorismes de shading siguin auto-contingudes. D'aquesta classe hereten els quatre algorismes de shading.

La classe **IGraphicDriver** és una interfície per implementar les abstraccions de la llibreria gràfica. D'aquesta hereten les implementacions de la llibreria gràfica.

Aquestes classes representen el projecte a alt nivell, i al voltant d'aquestes hem implementat el sistema.

La majoria de l'aplicació utilitza la classe **Engine** com a nexa, però en pro de la netedat del codi, hem utilitzat el patró Singleton [5], en aquells casos que utilitzar la classe Engine suposés complicar massa el codi.

L'altre decisió de disseny important és qui és propietari de certs objectes, especialment quan aquests son compartits. Per exemple, les textures i els materials poden ser compartits per diferents objectes, i volem no tenir diverses còpies de les mateixes dades en memòria. Per tal d'evitar-ho, durant la càrrega d'aquests, és comprova si ja s'han carregat, i si és així, es comparteixen. El problema llavors esdevé saber en quin punt podem eliminar els objectes. Per això, hem utilitzat una estructura anomenada **Smart Pointer** [12], que ens permet mantenir un comptador de referències, i així saber quan la textura o el material no és referenciat per cap objecte, i per tant es pot destruir.

Per la construcció de les classes que implementen els algorismes de shading, hem utilitzat el patró de *factoria abstracta*. D'aquesta manera, durant el desenvolupament, el motor no coneixia quins *algorismes de shading* podia arribar a crear, si no que els demanava a la factoria. L'avantatge, junt amb l'interfície creada per aquests, és que afegir nous algorismes de shading és extremadament fàcil. Fins i tot, en podem crear versions de depuració durant el desenvolupament que després podem desactivar.

Finalment, s'ha utilitzat àmpliament la llibreria estàndard de funcions *templitzades* de C++, que ens ofereix accés a diferents tipus d'estructures de dades molt útils per a consultar dades de forma ràpida, com per exemple mapes o sets, que tenen temps d'accés amb un cost asimptòtic acotat.

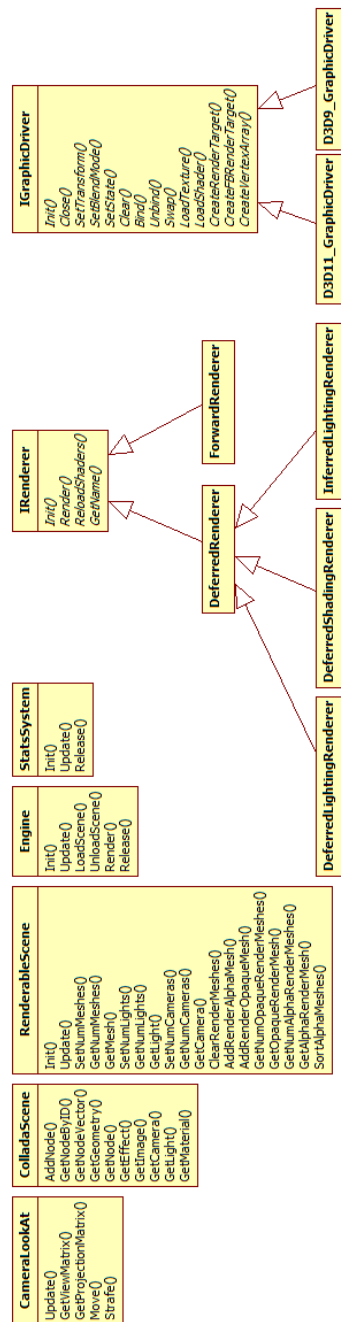


Figura 8.2: Disseny inicial de classes de l'aplicació

## Capítol 9

---

# Implementació i proves

---

Fins ara només he parlat d'algorismes de shading, sense detallar exactament com funcionen. En aquest capítol detallaré la implementació dels quatre algorismes de shading, i els seus avantatges i desavantatges teòrics.

A més, parlaré de parts de l'implementació dels quatre mòduls que han resultat problemàtiques i quines solucions he aplicat. Finalment, comentaré parts que, tot i no ser problemàtiques, em semblin especialment interessants i dignes de comentar.

### 9.1 Implementació dels algorismes de shading

Com hem comentat al capítol 5, avui en dia podem avaluar l'equació de shading tant utilitzant *vertex shaders* com *pixel shaders*. De fet, el més habitual és avaluar una part de l'equació per vèrtex, i la resta per fragment. La separació acostuma a venir donada per les necessitats de rendiment i qualitat, però simplificant, podem dir que qualsevol propietat de la geometria ens interessarà tractar-la per vèrtex, mentre que la majoria de propietats dels materials les acabarem tractant per fragment. Seguint amb la simplificació, sempre ens interessarà avaluar la il·luminació per fragment, ja que si no la il·luminació dependrà de la quantitat de triangles que tingui la geometria, com podem veure a la [Figura 9.1](#).

No entrarem en més detall de quines parts ens interessa implementar per vèrtex o per pixel, doncs l'elecció no és senzilla. A nivell de les següents explicacions, ens centrarem en l'ús de *pixel shaders*, doncs els quatre algorismes intenten reduir la càrrega que aquests reben.

A nivell d'efectes, als quatre algorismes hem suportar el mateix:

- **Textures difoses:** Donen l'aparença base als objectes, sobre elles apliquem l'il·luminació.

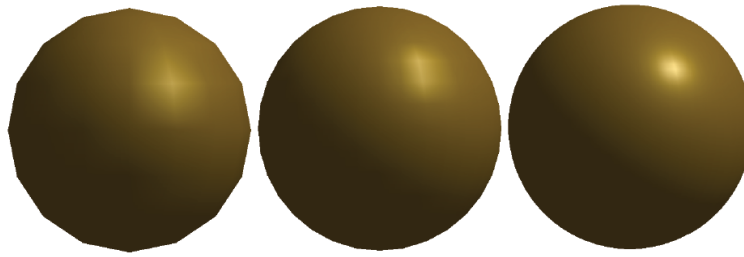


Figura 9.1: Una esfera il·luminada per vèrtex. D'esquerra a dreta, les esferes tenen 256, 1024 i 16384 triangles

- **Textures especulars:** Permeten modular on apliquem la il·luminació especular.
- **Textures de normals:** Permeten tenir normals per pixel, donant molt més detall a la superfície.
- **Brillantor especular:** Com de concentrades son les reflexions especulars
- **Llums puntuals:** Amb suport de color i radi a on afecta
- **Llums focus:** Amb suport de color, radi a on afecta i radi d'obertura

### 9.1.1 Forward shading

Aquest és l'algorisme de rasterització clàssic, que s'ha fet servir de forma quasi exclusiva durant més de 10 anys en jocs, aplicacions de CAD i altres entorns que requereixen interactivitat.

L'algorisme clàssic a alt nivell és el següent:

- **Per cada llum**
  - **Per cada** objecte afectat per la llum
    - \* Dibuir la objecte

La idea és que anem afegint la il·luminació que les diferents llums aporten a l'objecte. El problema és que potencialment hem de re-dibuir la mateixa geometria molts cops, si aquesta és afectada per moltes llums. Per tal de solucionar-ho, normalment s'utilitza la següent variació:

- **Per cada objecte**
  - Determinar les llums que afecten l'objecte
  - Dibuir l'objecte



La qual cosa ens permet dibuixar els objectes només un cop, però té diversos desavantatges. El primer problema és que hem de determinar quines llums afecten a quins objectes. Determinar quins objectes afecten a quines llums, en una escena amb milers d'objectes i desenes de llums, esdevé un problema no trivial. Es converteix fàcilment en un problema més complex que determinar quins objectes son visibles en una escena.

El segon problema, és que pot ser que a l'objecte l'afectin més llums que podem dibuixar pintant un sol cop (per exemple, la pipeline fixa de les primeres GPUs només permetia 8 llums per objecte), amb la qual cosa, haurem de fer una versió de l'algorisme híbrida amb l'anterior:

- **Per cada** objecte
  - Determinar les llums que afecten l'objecte
  - **Mentre** no haguem dibuixat l'objecte amb totes les llums
    - \* Dibuir l'objecte

I en aquest cas, tornariem a tenir el problema de dibuixar l'objecte varies vegades.

El principal problema que té i que va impulsar el desenvolupament dels altres algorismes que tractem al projecte, és que no tenim manera de saber quines parts de la geometria son afectades per quines llums. Com hem dit, determinar quins objectes afecten a quines llums és complex: determinar quins triangles d'una geometria son afectats per una llum és converteix ràpidament en un problema intratable. Com que no podem determinar exactament quines zones d'un objecte son afectades per una llum pot ser que estiguem fent molts càlculs innecessaris.

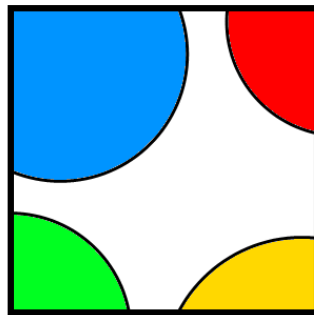


Figura 9.2: Exemple d'un dels problemes amb Forward Shading

Imaginem que el rectangle de la Figura 9.2 és l'objecte que volem dibuixar, i les zones de color son zones on afecten les llums. Si utilitzem la llum de la zona de color blau com exemple, i tenint en compte que no tenim forma de

determinar on afecta aquesta, haurem de processar tot el rectangle per calcular-ne la il·luminació. Si comparem la mida del rectangle amb la mida de la zona on afecta la llum de color blau, podem veure que hem processat molta part del rectangle on la llum no afecta, fent càlculs completament innecessaris. El mateix passarà amb la resta de llums.

Un altre problema que té a nivell d'implementació, és que sofreix el que anomenem **explosió combinatòria de shaders**, que ara explicarem. Quan utilitzem shaders, hem de programar el suport per cada tipus d'efecte que vulguem suportar. Per exemple, si volem suportar diferents tipus de llums, ombres, normals per pixel o altres. Llavors, si volem dibuixar objectes amb diferents configuracions, hem de donar suport a poder activar o desactivar cada efecte. Això fa, que per un simple shader existeixin moltes configuracions diferents, per exemple, llum puntual amb ombres, o també, una llum focus afectant un material amb color pla i normals per pixel. Ràpidament es pot veure que hem de poder crear totes les permutacions possibles dels efectes suportats, si volem poder activar o desactivar efectes. A més, si volem dibuixar les llums en blocs, per optimitzar el rendiment, encara tenim més permutacions. A aquesta gran quantitat de possibles combinacions és al que ens referíem. Hi ha diferents formes d'implementar un sistema que pugui suportar aquesta combinatòria, però quasi totes es basen en tenir una descripció pseudo-modular de tots els efectes i després generar les permutacions.

Respecte a l'implementació del projecte, hem optat per programar un shader que suporta tots els efectes, que es poden activar amb directives del compilador de shaders. Les permutacions les generem (al vol) mentre s'executa l'aplicació. Això ens permet evitar generar les desenes de milers de permutacions, que trigarien molt a compilar. A canvi, generar cada permutació durant l'execució suposa un petit cost, que es negligible dins del temps de càrrega de l'aplicació.

També s'ha fet una simplificació de la determinació de quines llums afecten a quins objectes, doncs s'ha utilitzat només el radi de les llums puntuals. Es podria haver programat el suport per detectar la intersecció entre un con i una caixa, la qual cosa ens hauria permès determinar més exactament quines llums de tipus focus afecten a quins objectes, però degut a que la majoria de llums son de tipus puntual, no ho he cregut necessari.

### 9.1.2 Deferred shading

El Deferred Shading intenta solucionar el problema exemplificat a la [Figura 9.2](#). L'algorisme de és el següent:

- **Per cada** objecte
  - Dibuixar propietats relacionades amb l'il·luminació de l'objecte
- **Per cada** llum

- Consultar propietats d'il·luminació i avaluar la il·luminació de la llum

La idea és separar el dibuixat d'objectes de la il·luminació, passant d'un cost asimptòtic  $O(m*n)$ , a un cost teòric  $O(\max(m,n))$ , si entenem  $m$  com el nombre d'objectes i  $n$  el nombre de llums. Òbviament això és una simplificació, però intuïtivament podem veure que aquest algorisme serà més escalable si augmentem el nombre de llums.

Per tal de separar quan dibuixem objectes i llums, el primer que és fa és guardar les propietats dels objectes visibles en diversos buffers, com podem veure a la Figura 9.3. Per tal d'avaluar la il·luminació de forma completa, i tal com hem vist al capítol 5, necessitarem com a mínim la posició i la normal, per cada pixel. Normalment, es guarda per pixel l'informació de color sense il·luminar, la normal, la brillantor especular i la profunditat. Aquest conjunt d'imatges s'anomena **Geometry Buffer** (GBuffer) perquè emmagatzema la informació de les geometries.

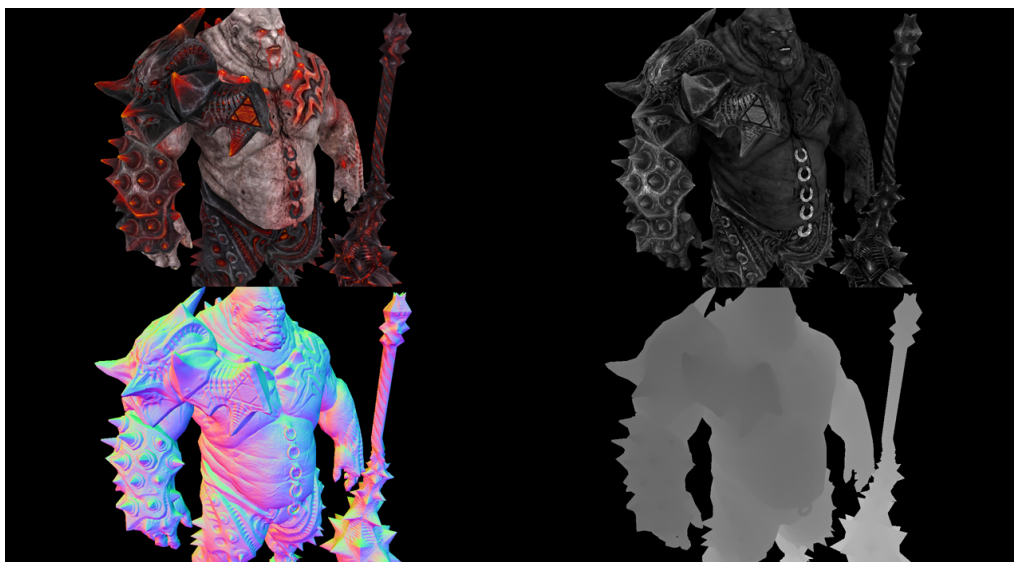


Figura 9.3: Exemple de GBuffer. Color (superior-esquerra), normals (inferior-esquerra), propietats especular (superior-dreta) i profunditat (inferior-dreta)

Un cop tenim el GBuffer construït, pintem totes les llums de l'escena. Per cada llum, construïm un volum que la conté, i el projectem en pantalla. Els pixels coberts per aquest volum projectat, són els que potencialment poden ser il·luminats. Per cada pixel, consultem al GBuffer la profunditat, respecte la qual podem reconstruir la posició del pixel [20]. Un cop tenim la posició, podem il·luminar com ho faríem normalment, doncs podem consultar totes les propietats per il·luminar al GBuffer que hem emplenat anteriorment.

El principal avantatge que ofereix aquest algorisme, és que només il·luminem exactament els píxels que son afectats per una llum, així que en cap moment fem càlculs superflus. Això permet que aquest algorisme escali molt bé respecte la quantitat de llums que afecten un objecte. Com que només treballen amb la relació entre llum i píxel, no cal determinar en cap moment quines llums afecten a quin objecte, que descarrega molt la CPU.

Un dels desavantatges que té, és que el GBuffer consumeix molta memòria. Normalment s'utilitzen 3-4 imatges de la mateixa resolució que estem pintant, cosa que pot sumar varies desenes de megabytes fàcilment. A més, consultar les dades per cada píxel suposa consultar el GBuffer, amb la qual cosa incrementem la càrrega al bus de memòria. El bus de memòria, històricament, incrementa la seva velocitat de forma molt més lenta que la resta d'unitats (com els nuclis de computació o les CPUs), amb la qual cosa pot significar un problema. Per això, normalment es busquen formes d'empaquetar les dades de la manera més compacta possible, aconseguint reduir l'ample de banda necessari, a canvi de la necessitat d'empaquetar i desempaquetar els valors. [23]

Un altre desavantatge, és que no suporta transparències de forma nativa. Com hem dit abans, les transparències afegeixen informació després dels objectes opacs, i poden haver-ne varies per píxel. Al GBuffer només podem guardar informació d'un objecte per píxel, amb la qual cosa no podem guardar informació de diverses transparències una damunt l'altra. La solució habitual és mantenir shaders semblants als d'un algorisme **Forward Shading**, e il·luminar amb ells després de la fase de **Deferred Shading**.

Finalment, la fase d'il·luminació sempre utilitza el mateix shader, que ve donat per la configuració del GBuffer. Això és un avantatge i un desavantatge. Per una part, només hem d'escriure variacions d'efectes en la fase de creació del GBuffer, que és l'únic punt on poden haver-n'hi. A la fase d'il·luminació sempre il·luminem respecte els paràmetres del GBuffer, la qual cosa vol dir que només tenim un tipus de material. Això és un desavantatge perquè és difícil definir superfícies diferents (metalls, pell, etc) amb un sol tipus de material.

La implementació del projecte no utilitza cap tipus d'empaquetament, perquè la quantitat d'atributs que havíem de suportar no cabien de cap manera en menys buffers, així que hem preferit mantenir una millor qualitat.

### 9.1.3 Deferred lighting

La generació actual de consoles (XBox360 i PS3) té relativament poca memòria per a gràfics (especialment la XBox360), i un ample de banda lleugerament limitat (especialment la PS3), amb la qual cosa una implementació estàndard de **Deferred Shading** era difícil. L'algorisme **Deferred Lighting** utilitza menys memòria a costa d'haver de dibuixar els objectes varies vegades:

- **Per cada objecte**
  - Dibuixar propietats mínimes de l'objecte per a poder il·luminar
- **Per cada llum**
  - Consultar propietats d'il·luminació, acumular termes difós i especular per separat
- **Per cada objecte**
  - Consultar termes difós i especular, i il·luminar amb propietats d'objecte

La filosofia d'aquest algorisme és molt similar al Deferred Shading: primer guardem les dades per il·luminar i després calculem l'aportació de les llums. La diferència en aquest cas, és que generem un GBuffer molt més petit, només amb les normals i la profunditat (la meitat que el cas típic amb Deferred Shading).

La il·luminació es guarda en dos buffers, un que conté el terme difós i un altre que conté el terme especular (Figura 9.4), i llavors es fa una última passada per unir aquests dos buffers amb la informació de color dels objectes, i altres propietats, com per exemple una màscara que indica on aplicar el terme especular.

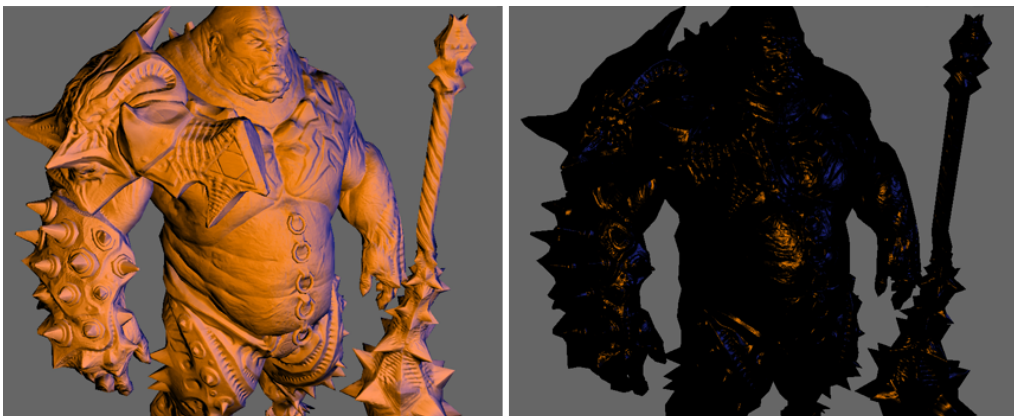


Figura 9.4: Exemple de buffers d'acumulació. Terme difós (esquerra) i terme especular (dreta)

Com avantatges, consumeix menys memòria, tot i que cal mantenir dos buffers per la il·luminació que anem calculant. També, s'han de llegir menys dades del GBuffer, amb la qual cosa fem menys càrrega sobre el bus de dades. Per realment consumir menys, s'acostuma a empaquetar les dades, doncs podem arribar a codificar la normal i la profunditat utilitzant un sol buffer, i podem guardar una aproximació del terme difós i especular amb un únic buffer. El problema, es

que impliquen una rebaixa de qualitat, que depenent del que haguem de dibuixar, pot no ser acceptable.

Comparteix la resta de desavantatges del Deferred Shading: no suporta directament transparències i no suporta múltiples tipus de materials.

La implementació del projecte utilitza dos buffers per guardar les normals i la profunditat, i també dos buffers per acumular la il·luminació. La única particularitat és que guarda la il·luminació utilitzant una escala que va del 0 al 2, en comptes de l'habitual 0 al 1. Vaig decidir perdre precisió a canvi de més rangs, ja que així aconseguia els mateix tipus d'il·luminació que altres algorismes de shading.

#### 9.1.4 Inferred lighting

Hem comentat que tant el **Deferred Shading** com el **Deferred Lighting** no suporten transparències, i que l'ús de memòria és elevat en el primer. El **Inferred Lighting** busca solucionar el problema amb les transparències i reduir els requisits de memòria.

L'algorisme és molt semblant al del Deferred Lighting:

- **Per cada** objecte
  - Dibuixar propietats mínimes per il·luminar i diferenciar l'objecte
- **Per cada** llum
  - Consultar propietats d'il·luminació, acumular termes difós i multiplicador especular
- **Per cada** objecte
  - Consultar termes d'il·luminació i il·luminar amb propietats d'objecte

La base és el Deferred Lighting, però utilitzant un GBuffer de menor resolució que la pantalla. L'idea és que la freqüència de la majoria d'il·luminació és prou baixa com per poder reduir els buffers utilitzats per il·luminar, sense que es noti al resultat final. La imatge final no es modifica, té la mateixa resolució que la pantalla.

El problema que podria tenir, és que com tenim els buffers d'il·luminació a menys resolució que la imatge final, quan dibuixéssim els objectes a la última part de l'algorisme, podríem crear errors (Figura 9.5). Això passa perquè al consultar els buffers de menys resolució, podem estar llegint informació provinent d'objectes que no son l'actual. Això és especialment dolent si estem utilitzant filtrat bilineal, que al llegir una textura consulta més d'un pixel.

Per això, aquest algorisme introdueix la **Discontinuity Sensitive Filtering** (DSF). El DSF es basa en guardar dades extres al GBuffer durant la seva creació



Figura 9.5: Exemple d'errors d'interpolació utilitzant Inferred Lighting

(Figura 9.6), de manera que en el moment d'aplicar l'il·luminació, en la fase final, puguem detectar discontinuïtats fàcilment, com per exemple diferències d'objecte o de profunditat. Amb aquesta informació, podem crear un filtre bilineal que només consulti la informació correcta. Aquest mateix filtre també permet il·luminar objectes transparents, però no ho detallarem perquè escapa a l'abast del projecte.

A més, a l'article original [15], els creadors de l'algorisme utilitzen un buffer d'il·luminació únic, guardant el color del terme difós, i el terme especular com un únic valor que representa un multiplicador. Després, en la passada final es restaura el terme especular, utilitzant aquest multiplicador amb el valor del terme difós. És una aproximació que pot resultar incorrecta, però té l'avantatge que només requereix un buffer per acumular la il·luminació.

Els desavantatges que té aquest algorisme, és que el filtre DSF no és gaire barat, comparat amb el filtrat bilineal, que és fa per hardware. A més, tot i que per separat és difícil notar que la il·luminació està a resolució més baixa, si ho comparem amb altres algorismes, és bastant notable. Tampoc funciona gaire bé quan la il·luminació té alta freqüència, com és el cas de reflexions especulars amb normals per pixel.

La implementació del projecte utilitza la profunditat i un identificador d'objecte, com a elements extres al GBuffer, que després son utilitzats al filtre DSF. Respecte la reducció, el GBuffer i els buffer d'il·luminació tenen unes dimensions equivalents al 70% de la resolució de pantalla, que significa aproximadament la meitat de pixels a processar.



Figura 9.6: Buffer DSF que conté la profunditat discreta i un identificador d'objecte

## 9.2 Implementació dels mòduls

### 9.2.1 Mòdul de càrrega

El mòdul de càrrega es dedica a llegir el format Collada, que està codificat utilitzant XML, i a extreure'n dades. Després dona format a aquestes dades perquè el mòdul de dibuixat pugui utilitzar-les fàcilment. Hi ha dos detalls de la implementació mòdul de càrrega interessants a comentar, doncs han resultat complexes d'implementar: la lectura de gran quantitat de dades, i el processat d'aquestes.

#### Lectura de dades

Simplificant, el format Collada, és una descripció de dades en format text. Les dades en format text tenen una sèrie d'avantatges respecte a les dades binàries, però les principals és que són fàcils de llegir i d'editar pels humans. Això és molt pràctic, doncs analitzar-ne el format pot ser tan fàcil com obrir un editor de text, i llegir. També és molt pràctic per fer proves, doncs amb un senzill editor de text podem modificar parts de l'arxiu. Aquesta propietat s'ha utilitzat de forma extensiva durant el desenvolupament del projecte, la qual cosa n'avalua la utilitat.

El problema del format text és que és difícil de processar. Els ordinadors treballen de forma òptima amb dades binàries, de manera que hem de convertir les dades de format text a binari, procés extremadament delicat i lent. A la taula 9.1 podem veure un exemple senzill d'un valor en diferents codificacions. En aquesta taula es pot veure que la codificació binària, a part d'altres propietats que no detallaré, necessita menys valors per ser representada.

Els formats que descriuen dades 3D contenen majoritàriament vèrtexs i les



Format text	1.2345
Codificació en text (hexadecimal)	003DEB01C400
Codificació binària (hexadecimal)	3F9E0419

Taula 9.1: Exemple d'un valor decimal en diferents codificacions

seves propietats, podríem dir que entre el 70 i el 80 per cent ho son. Els vèrtexs i les seves propietats, de forma simplificada, no son més que llistes de valors decimals.

En resum, per llegir el format Collada, hem de llegir una gran quantitat de valors decimals en format text i convertir-los a la seva representació binària. Optimitzar aquest procés ha inclòs diverses tasques, que enumero:

- **Rutina de conversió de decimals en format text a binari:** La llibreria estàndard de C inclou una funció que converteix un decimal representat en format text a format binari. Com que és una funció estàndard, suporta tots els possibles valors decimals. El format Collada no utilitza tots els tipus de valors, amb la qual cosa és possible escriure una funció que només tracti el subconjunt que empra, a més d'altres optimitzacions de més baix nivell.
- **Evitar còpies:** Degut a que tractem amb moltes dades, és molt important evitar qualsevol còpia de memòria. Per això, el sistema de conversió de dades només recorre les dades al anar fent el canvi de codificació, evitant fer còpies. Això ha estat gràcies a la rutina programada al punt anterior i a un disseny orientat de forma expressa a no copiar mai dades.

Aquestes dos optimitzacions han permès rebaixar el temps de càrrega en un ordre de magnitud.

### Processat de dades

Un cop tenim les dades carregades, hem de processar-les per tal que siguin dibuixables. Normalment, els formats de dades 3D contenen aquestes de la manera més semblant al software de modelat que s'ha utilitzat per crear-les. Això suposa un problema, doncs acostumen a tenir, entre altres, vèrtexs duplicats, geometries que no estan descrites en base a triangles, o polígons degenerats (que no tenen superfície). Al procés que s'encarrega d'eliminar vèrtexs duplicats, polígons degenerats i a triangularitzar polígons, se l'anomena **consolidació**.

El procés de triangularitzar polígons és relativament senzill si aquests son convexos i també el d'eliminar polígons degenerats. Cap dels dos acostuma a ser problemàtic a nivell de rendiment.

Eliminar vèrtexs duplicats acostuma a ser més problemàtic. Comparar dos vèrtexs suposa comparar totes les propietats que el componen, i cal tenir en

compte que un vèrtex pot tenir 4 o 5 propietats, essent cada propietat un vector de 3 o 4 elements. Per cada vèrtex, hem de comprovar que no és igual a tots els altres vèrtexs, la qual cosa el converteix fàcilment en un problema amb un cost asimptòtic quadràtic, almenys per la implementació trivial.

Per resoldre aquest problema de forma ràpida, hem optat per crear valors **hash** de cada vèrtex. Aquest valor **hash** es pot entendre com un identificador únic del vèrtex i les seves propietats. Llavors, utilitzant una estructura de tipus mapa, on la clau és el **hash**, i el valor és l'índex del vèrtex, podem saber ràpidament si tenim un vèrtex duplicat, i a quin índex redirigir-ne les referències.

## 9.2.2 Motor 3D

Deixant de banda la interessant implementació dels algorismes de shading, el motor 3D té certs detalls interessants. Parlarem de les utilitats que hem programat per accelerar el desenvolupament, per mesurar el rendiment i el correcte ús de la memòria. També de l'algorisme per determinar els objectes visibles.

### Mesura de rendiment

Ja he dit diverses vegades que era molt important el rendiment del motor 3D, per tant necessitava utilitats que em permetessin comprovar els colls de botella. A tal efecte, vaig programar el que s'anomena un **profiler**. Un profiler, ens permet saber quantes vegades s'ha cridat una funció, els temps acumulat i el temps mitjà per crida, qui ha cridat una funció i altres dades. Els profilers poden implementar-se com a aplicacions externes, com és el cas del conegut VTune [11], o com a funcionalitat interna, com és el cas del projecte.

El profiler del projecte utilitza un sistema de marques per determinar les àrees on volem mesurar el rendiment. Bàsicament marquem l'inici i el fi de blocs de codi, que es registren al profiler. En qualsevol moment, el profiler permet escriure un resum de rendiment a disc, per poder-lo examinar (Figura 9.7). És un sistema senzill i potent, que cobreix les necessitats del projecte.

Profile point	Time (ms)	Calls
LoadCollada	660	1
ConvertFromCollada	75	1
ConvertCameras	0	1
ConvertLights	0	1
ConvertMeshes	74	1
LoadTexture	24	25
ConvertAnimation	0	27

Figura 9.7: Extracte del resum creat pel profiler del projecte

### **Sistema de memòria**

Un dels avantatges i desavantatges de programar en C++ és el seu model de memòria. És molt eficient, doncs ens permet un accés a baix nivell. Ens permet decidir quan reservem la memòria i quan la alliberem, però hem de ser molt prudents recordant d'alliberar-la. Això que pot semblar trivial, és un problema comú, i per tant vaig decidir crear un sistema per evitar-ho, al projecte.

El projecte inclou una re-implementació de les funcions de memòria, que s'ocupen de mantenir una llista de quina memòria és reservada, i des de quin punt del codi s'ha reservat. El sistema ens permet llistar en qualsevol moment quanta memòria estem utilitzant i qui la ha reservat. Per tal que el motor pogués carregar les dades en calent, sense haver-lo de reiniciar, calia assegurar-se que al descarregar una escena, tota la memòria associada a ella fos descarregada, cosa que aquest sistema permet comprovar. A més, aquest sistema pot ser la base per utilitats molt més avançades, com detecció de corrupció de memòria, però es fora de l'abast del projecte. En resum, aquest sistema ens permet assegurar l'ús correcte de la memòria, que és molt important per a programar aplicacions estables.

### **Recàrrega de shaders en calent**

Finalment, una decisió de disseny que ha influït de forma notable en la implementació: poder recarregar els shaders durant l'execució de l'aplicació.

Com hem dit, carregar fitxes Collada pot ser un procés lent, que pot arribar a trigar fins a 15 segons, un cop optimitzat. El procés de programar shaders acostuma a ser força més complexe que programar C++, doncs no tenim eines per depurar, i moltes vegades hem de fer una gran quantitat de proves. Si per cada petita prova es trigués 15 segons, seria difícil aconseguir bons resultats.

A tal efecte, des del principi, tot el motor 3D està pensat per poder recarregar els shaders en calent. De fet, gran part dels gràfics d'aquesta memòria s'han creat aplicant petites modificacions als shaders, per a poder veure diferents components.

Pot semblar un detall d'implementació trivial, però crec que sense aquesta funcionalitat hauria d'haver invertit entre 10 i 100 vegades el temps que he necessitat per desenvolupar els shaders.

### **Test de visibilitat d'objectes**

El problema a solucionar és senzill: donats diferents objectes repartits en un espai 3D i una càmera, com determinem quins son visibles? És important determinar quins son visibles, doncs cada objecte que no ho és, no aporta informació a la imatge final generada, i per tant, processar-lo és innecessari.

Els algorismes de visibilitat és una àrea dels gràfics per ordinador que ha vist una gran quantitat de recerca, doncs determinar ràpidament quins objectes son visibles és la base per a un motor 3D ràpid.

Un exemple n'és el **Potentially visible set**, utilitzat pels motors creats per Id Software (famosa per jocs com Doom o Quake), que utilitza un pre-procés per calcular quins objectes son visibles des de diferents cel·les de l'espai 3D. Després, durant l'execució, només cal consultar les dades de visibilitat. Un altre exemple és l'ús de **Portals**, que es basa en dividir l'espai en regions comunicades per finestres o portals. Tant el **Real-Time Rendering** [1] com el **Game Engine Architecture** [7] contenen explicacions d'aquests i altres algorismes de visibilitat.

Hem utilitzat l'algorisme més bàsic, el **Frustum Culling**. Normalment, en motors 3D avançats, s'utilitzen sistemes de visibilitat com els esmentats als paràgraf anterior, i s'utilitza el Frustum Culling per acabar de filtrar les llistes resultants dels altres algorismes.

La primera part del procés és determinar el volum visible, que com hem dit al capítol 5, depèn del tipus de projecció. Per tal de simplificar l'explicació, ens centrarem en la projecció de perspectiva, però el mateix aplica a la projecció paral·lela. El volum visible, utilitzant una projecció de perspectiva és, de forma simplificada, una piràmide. En anglès, aquest volum piramidal s'anomena **frustum**, d'aquí el nom de l'algorisme. Un cop hem determinat aquest volum, hem de comprovar quins objectes interseccionen amb aquest. Com que comprovar triangle a triangle és molt car, normalment s'acostumen a utilitzar volums contenidors, com per exemple capses o esferes (exemple a la Figura 9.8).

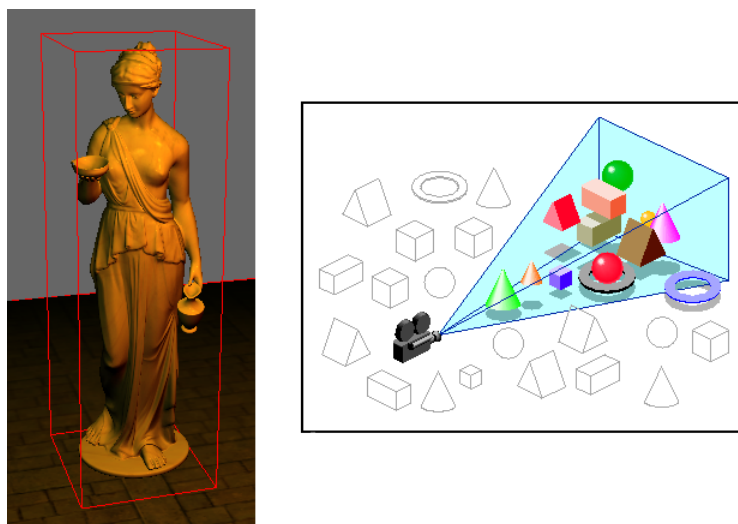


Figura 9.8: Capsa contenidora (esquerra). Esquema de Frustum Culling (dreta)

La implementació del projecte, durant la càrrega de geometries del motor 3D, crea capsas contenidores en espai d'objecte, per a cada una de les geometries (hi ha un exemple visible a l'esquerra de la Figura 9.8). Aquestes capsas sempre estan alineades als eixos de coordenades de l'espai de món, és a dir, no tenen cap tipus de rotació.

Un cop s'ha de determinar la visibilitat, es calculen els sis plans que té el frustum. Aquest plans es calculen de manera que estan orientats cap a l'interior del frustum. Això normalment és fa orientant el pla de manera que, si calculem la distància d'un punt a l'interior del frustum, obtinguem una distància positiva. Per cada objecte, s'obté la capsa contenidora (que ja havíem calculat), es transforma a l'espai de món, i es calcula la posició dels seus vuit vèrtexs respecte els plans del frustum. Si algun dels vèrtexs és a l'interior del frustum, vol dir que la capsa contenidora és visible, i per tant marquem l'objecte com a visible.

Un dels desavantatges d'aquesta tècnica és que pot arribar a ser lenta si tenim una quantitat massiva d'objectes, per la qual cosa, com hem dit abans, normalment es filtren els objectes amb alguna altra tècnica. Un altre desavantatge és que dóna falsos positius, doncs el volum contenidor és només una aproximació de l'objecte. Tot i això, aquest algorisme és molt senzill d'implementar i una peça fonamental de qualsevol motor 3D.

### 9.2.3 Sistema de mesura

El sistema de mesura es dedica a recollir dades, tant de rendiment com de diverses mesures de l'escena. A més, també pot executar tests automàtics. El concepte darrere d'aquests tests automàtics, es que puguem carregar una escena, seleccionar un algorisme de shading, activar el test, i que aquest canviï la quantitat, mida i distribució de les llums de l'escena. Així, el mòdul, a nivell senzill, permet guardar dades i crear una configuració de llums, doncs és necessari per poder crear tests automàtics.

La part de crear llums de forma automàtica, és interessant, ja que hem de definir com cobrir l'escena el millor possible. L'algorisme que he implementat, per tal d'omplir l'espai de llums, és el següent:

1. **Determinem el volum de l'escena:** Recorrem les capsas contenidores dels objectes, convertint-les a espai de món. Desem els punts màxims i mínims que anem trobant.
2. **Determinem la separació entre llums:** La separació depèn de les mides de la capsa contenidora i el nombre de llums que volem distribuir en cada eix.
3. **Determinem el radi de les llums:** Per tal que les llums cobreixin tot l'espai, determinem quin dels tres eixos de separació de llums és el màxim.

Un cop coneixem aquesta separació *màxima*, el radi de cada llum és la meitat d'aquesta separació.

4. **Generem les posicions de les llums:** Calculem la posició *base* d'una llum. Aquesta és la posició mínima de la caixa que conté tota la escena, més el radi de llums. És important afegir el radi, o part de les llums quedaran fora del volum. Respecte a aquesta posició base, només cal anar afegint increments de la separació, calculats al punt 2.

A la [Figura 9.9](#) podem veure un exemple de la distribució de llums, on he reduït lleugerament els radis de les llums perquè és pugués entendre visualment.

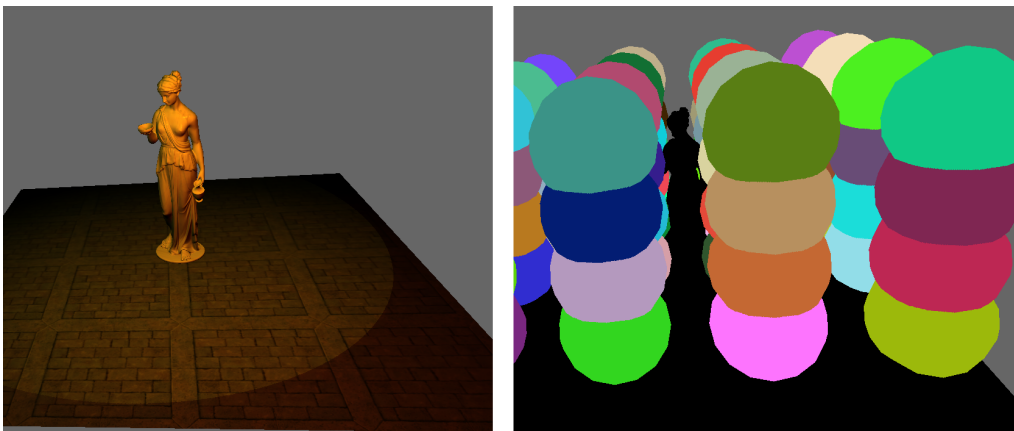


Figura 9.9: Escena amb les llums originals (esquerra). Esquema amb llums generades, representades com esferes de colors (dreta)

El test automàtic que he inclòs al sistema de mesura, varia la quantitat de llums generades, de manera que es pot comprovar com funciona un algorisme de shading, respecte a diferents quantitats de llums i mides d'aquestes. També he inclòs suport per moure i canviar el radi de les llums generades, que resulta útil per crear situacions de prova menys artificials.

## Capítol 10

---

# Implantació i resultats

---

Ara detallarem el procés de desenvolupament i els resultats obtinguts. Primer faré un anàlisi històric de com ha evolucionat el projecte, explicant perquè he treballat en cada tasca, i l'estat del projecte en aquell moment. A l'apartat de resultats, discutiré i mostraré l'estat dels diferents algorismes de shading, i inclouré una petita comparativa entre els diferents algorismes.

### 10.1 Procés de desenvolupament

Anem a veure, en format històric, el procés de desenvolupament. Per tal de detallar el procés, explicaré el que vaig desenvolupar en cada moment, quina era la motivació per desenvolupar aquella part, i a quin estat portava el projecte. On escaigui, a més, ho exemplificaré amb imatges. Essent un projecte centrat en els gràfics, crec que la millor manera de mostrar l'evolució del projecte és utilitzar imatges.

#### 10.1.1 Primer sistema de pintat

El primer que es va crear, tenint en compte que estava utilitzant una llibreria gràfica que no coneixia, va ser crear un sistema que em permetés dibuixar en pantalla.

Utilitzant la documentació i els exemples continguts dins del **DirectX SDK** (que és el conjunt de llibreries dins del qual es distribueix Direct3D 9), vaig aprendre com dibuixar punts i triangles a la pantalla, i com col·locar una càmera. Llavors, vaig crear un projecte que generava una càmera i un objecte utilitzant codi, amb la qual cosa podia comprovar el correcte funcionament del meu codi.

Aquesta primera tasca em va suposar una introducció a la llibreria gràfica Direct3D 9, a més de poder avaluar quina funcionalitat matemàtica necessitaria

en un futur. En aquest punt, tenia una aplicació que, utilitzant la pipeline fixa, dibuixava un objecte en pantalla.

### 10.1.2 Càrrega bàsica d'arxius Collada

Un cop tenia un sistema que podia pintar objectes, vaig decidir començar a carregar arxius Collada. Aquesta tasca tenia dos objectius: una primera visió del format Collada i un anàlisi del flux de dades dins el carregador.

El primer que vaig fer va ser triar una llibreria que permetés llegir còmodament arxius codificats en XML. Com he comentat anteriorment, vaig decidir utilitzar **pugixml**. Un cop triada la llibreria respecte les seves prestacions, vaig crear un esquelet del carregador, i vaig intentar carregar un arxiu Collada senzill utilitzant la llibreria. Aquest pas només servia per assegurar que la llibreria no donava cap error al carregar l'arxiu, degut a algun problema de lectura o un detall de la codificació no suportat.

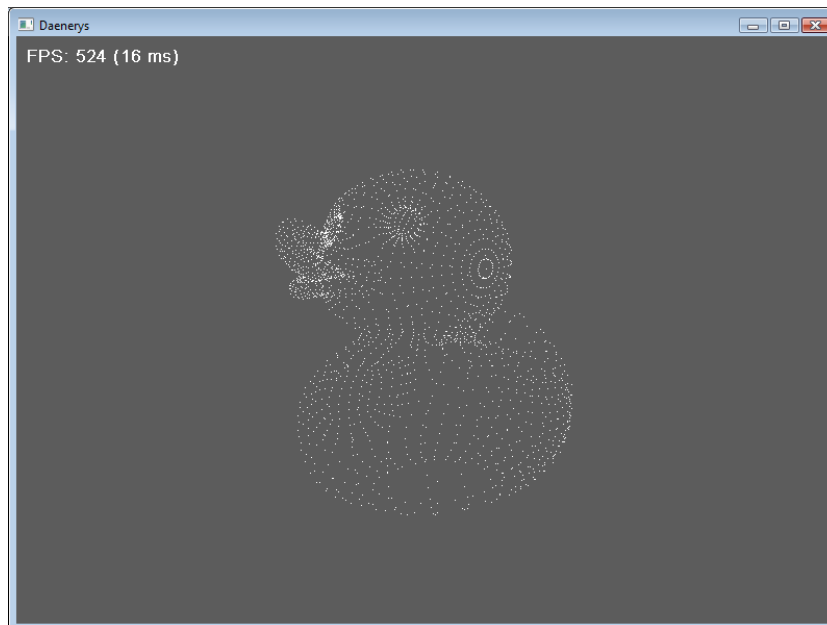


Figura 10.1: Primer objecte Collada dibuixat pel projecte

Un cop amb l'arxiu carregat utilitzant pugixml, el següent pas va ser identificar el punt on es definien les posicions dels vèrtexs i extreure-les. Aquest procés em va endinsar en la especificació del format Collada [9], a més de començar a definir exactament la implementació del carregador. Un dels grans avantatges de la codificació XML és que es molt fàcil ignorar parts que no entenem o no volem convertir, la qual cosa em va facilitar la tasca. Vaig crear un carregador que només llegia la posició dels vèrtexs, ignorant qualsevol altre element. Un cop



tenia els vèrtexs carregats, vaig adaptar la part encarregada de pintar perquè els pogués dibuixar, com es pot veure a la [Figura 10.1](#).

Un cop tenia el primer objecte dibuixat, i la primera implementació del carregador, vaig anar afegint-li funcionalitat poc a poc, a la vegada que també desenvolupava la part de pintat. En aquesta primera fase vaig incorporar suport per més propietats de vèrtexs, a més de triangles, polígons i llums. El resultat al final d'aquesta fase és el visible a la [Figura 10.2](#).



Figura 10.2: Primer objecte il·luminat que el projecte va dibuixar

A partir d'aquesta versió bàsica del carregador, es va anar iterant el carregador respecte les necessitats del sistema de dibuixat i del projecte. Es va afegir suport per càmeres, animacions i més propietats de vèrtex, entre altres.

Durant aquesta fase, el motor de pintat només utilitzava la pipeline fixa de la GPU. Això vol dir que no vaig necessitar programar cap shader, ni la funcionalitat per utilitzar-los, cosa que permetia tenir un motor de pintat extremadament senzill, tot i que poc flexible. Com que volia començar a programar shaders, la següent fase era crear la infraestructura necessària per poder crear-los.

### 10.1.3 Abstracció de la llibreria gràfica

Un cop el carregador de dades va assolir un estat bàsic, volia començar a implementar un sistema de pintat més elaborat. La idea era deixar de crear un sistema de pintat que evolucionava de forma orgànica, doncs s'utilitzava per comprovar el correcte funcionament del carregador, i dirigir la implementació cap al sistema dissenyat prèviament.

Per tal d'implementar el sistema de pintat, vaig creure convenient crear abans una abstracció de la llibreria gràfica. Tot i que pot semblar una pèrdua de temps, crear una abstracció em permetria el luxe de canviar la llibreria gràfica per sota d'aquesta capa, sense haver de canviar el motor.

El punt decisiu va ser poder estar protegit respecte a possibles problemes amb Direct3D 9. Com he comentat al capítol 7, Direct3D és una llibreria sense un sistema d'extensions. A més, la llibreria acostuma a canviar força la seva interfície d'ús, entre versions. En el cas hipotètic que necessités alguna característica no suportada a Direct3D 9, hauria de canviar a Direct3D 10, Direct3D 11, o fins i tot OpenGL. Allunyar el motor 3D dels detalls de l'interfície de la llibreria gràfica utilitzada era molt important.

Un altre avantatge, i que també va ser part de la decisió, és que utilitzant una abstracció, es pot crear una versió simplificada de la llibreria gràfica. D'aquesta manera, el motor 3D utilitza el hardware amb una interfície molt més senzilla. Això permet que, per exemple, la càrrega d'un shader esdevingui una sola funció, amb una comprovació d'errors trivial, mentre que si utilitzéssim directament la llibreria gràfica, potser serien 5 o 6 crides a funció, enterbolint molt el codi que es dedica a dibuixar.

#### 10.1.4 Creació del sistema de dibuixat

Un cop tenia una primera versió de l'abstracció gràfica, i una implementació d'aquesta utilitzant Direct3D 9, era el moment d'implementar el sistema de dibuixat.

La idea general era crear un sistema central, que servís de nexa i tingués tota la funcionalitat comuna, com inicialitzar la llibreria gràfica (a través de l'abstracció), utilitzar les dades del mòdul de càrrega, determinar visibilitat o mostrar dades de rendiment en pantalla, tot el que no fos molt estretament relacionat amb els algorismes de shading.

Un cop creada una versió bàsica d'aquest sistema central, calia determinar com implementaríem la fase d'aplicació dels algorismes de shading, és a dir, la càrrega de shaders, preparació de paràmetres, etc. Com hem vist al capítol de disseny, volíem una interfície molt petita per cada algorisme de shading, així que es va crear una factoria d'instàncies, i es va crear una *classe interfície*, que cada algorisme de shading havia d'implementar.

L'avantatge d'aquest sistema es que podia afegir nous algorismes de shading molt fàcilment, a més que aquests s'ocupen del mínim possible. De fet, en un principi, només es va implementar l'algorisme de **Forward Shading**, que és el més ràpid d'implementar.

Dos detalls importants que es van implementar en aquest punt, són la recarrega dels shaders i la possibilitat de canviar d'algorisme de shading utilitzat per dibuixar. Poder recarregar shaders, significa que podem iniciar l'aplicació, i anar editant els shaders amb un editor de text extern; un cop hem fet els canvis adients, tornem a l'aplicació i, pitjant una tecla, es recarreguen els arxius dels shaders. Això permet fer proves molt ràpid, essent una eina de depuració i desenvolupament increïblement útil.

Poder canviar d'algorisme de shading de forma ràpida, permet veure inconsistències entre ells. Com el seu funcionament és molt diferent, poder canviar entre un i altre ràpidament ens permet veure els errors. Si ajuntem aquesta característica amb la recarrega de shaders, podem corregir errors d'implementació dels algorismes de shading, de forma molt àgil i còmoda.



Figura 10.3: Primera implementació de Forward Shading utilitzant shaders, amb suport de textures, llums i il·luminació difosa

Quan vaig acabar aquesta fase, tenia un motor de pintat força senzill, però ben estructurat, que permetia dibuixar objectes il·luminats amb Forward Shading (Figura 10.3). A més, el codi estava preparat per afegir la resta d'algorismes de shading.

### 10.1.5 Sistema de càmera

En aquest punt, va començar a ser útil tenir un sistema de càmera, doncs volia poder comprovar errors de dibuixat o de càrrega.

Vaig implementar un sistema de càmera que utilitza l'entrada d'usuari per moure's per l'espai. El ratolí permet rotar la càmera, i el teclat avançar cap endavant, enrere, a l'esquerra o a la dreta. A més, com gran part del desenvolupament es feia amb l'aplicació oberta mentre s'editaven shaders al Notepad++, necessitava un mecanisme per poder activar o desactivar el sistema de càmera, ja que si no, es podia modificar la càmera, sense voler, al canviar de finestra.

Un cop acabada aquesta fase, el sistema de dibuixat ja tenia el suport per a utilitzar les dades del mòdul de càmera, cosa que permetia inspeccionar les escenes 3D a voluntat.

### 10.1.6 Determinació de visibilitat

Un cop perfilat el sistema de dibuixat i de càmera, volia incloure un sistema per dibuixar només aquells objectes realment visibles. Com hem dit al capítol anterior, vaig decidir utilitzar Frustum Culling.

Per a portar a terme aquesta fase necessitava dos elements: les dades de la càmera i les capsas contenidores dels objectes. La primera ja la havia resolt a la fase anterior, així que només faltava crear les capses.

Quan el sistema de pintat rep les dades del mòdul de càrrega, i abans de copiar-les a la targeta gràfica, analitza les geometries per extreure les capses que contenen cada una d'elles. Aquesta informació és guarda junt amb cada geometria. Un cop teníem les capses i la informació de càmera, vaig crear un sistema per extraure els plans del frustum.

Una vegada tenim els plans, només cal recórrer tots els objectes de l'escena, i comprovar si les seves capses estan, almenys parcialment, dins del frustum. Si ho estan, marquem l'objecte com a visible, propietat que serà consultada per la funció de dibuixat.

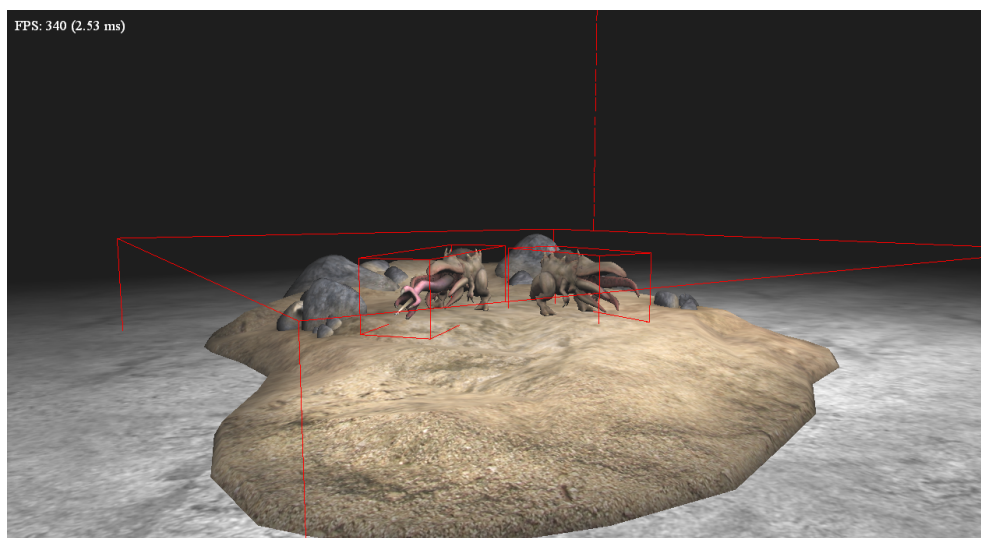


Figura 10.4: Primera versió del projecte amb capsas contenidores i Frustum Culling

En aquest moment, el projecte podia pintar objectes amb Forward Shading, moure la càmera, i determinar quins objectes eren visibles des d'aquesta, per pintar-los o no. En aquest punt, disposava d'eines més que suficients per començar a implementar la resta d'algorismes de shading.

### 10.1.7 Implementació de la resta d'algorismes de shading

En aquest punt, es va implementar la versió bàsica dels tres algorismes de shading restants: Deferred Shading, Deferred Lighting i Inferred Lighting.

La implementació d'aquests tres algorismes es va retardar fins a tenir un sistema estable i amb eines de depuració eficients per desenvolupar-los. A diferència del Forward Shading, aquests tres algorismes son força difícils d'implementar. La dificultat prové del fet que poden fallar per errors de precisió, per dades mal copiades o en un espai vectorial incorrecte, en contra del Forward Shading, que és molt més directe.

Al començar aquesta fase, l'algorisme de Forward Shading suportava il·luminar objectes amb diverses llums i materials bàsics. Per materials bàsics entenc una textura difosa, modulada amb els termes difós i especular, de les llums que influencien l'objecte. Per donar per completada aquesta fase, volia implementar els tres algorismes, i que suportessin el mateix que el de Forward Shading.

A parts iguals, el temps dedicat a aquesta tasca va dedicar-se a implementar els algorismes, i a depurar-los fins que fossin visualment iguals entre ells.

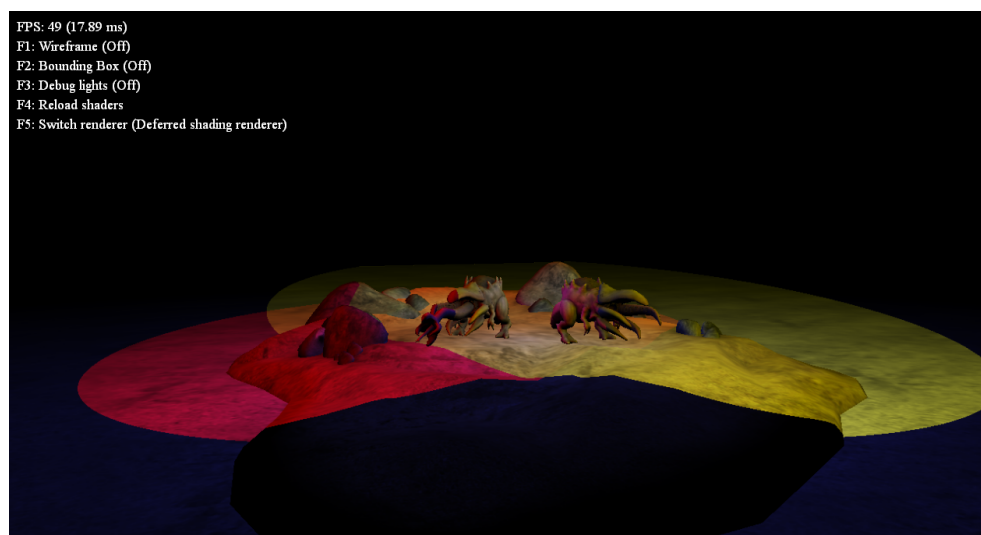


Figura 10.5: Primera versió de l'algorisme Deferred Shading

Un cop aquesta fase es va concloure, el projecte tenia una implementació bàsica dels quatre algorismes de shading. A més, podia canviar fàcilment entre ells per comprovar que eren visualment iguals, podia navegar l'escenari per inspeccionar possibles errors, i tenia eines de depuració per a detectar l'origen dels errors i corregir-los.

### 10.1.8 Eines de rendiment i memòria

Arribat a aquest punt, tenia una versió completa (però senzilla) del projecte. Vaig creure necessari parar el desenvolupament de noves característiques, i polir les existents. Aquesta procés de polir quasi sempre es divideix en dos fases: augmentar el rendiment i reduir el consum de memòria. A tal efecte, en aquest punt vaig desenvolupar tres utilitats, integrades dins del projecte.

La primera utilitat, va ser un sistema que em permetés saber en qualsevol moment qui consumia memòria. Hem detallat aquest sistema anteriorment, així que em centraré en l'ús. El primer que vaig fer, es aconseguir que aquest sistema escrivís un resum de la memòria no alliberada en el moment de tancar l'aplicació, la qual cosa em permetia fer una primera passada de neteja. Un cop vaig completar aquesta primera neteja, vaig ficar comprovacions en certs punts del codi, per comprovar que la quantitat de memòria utilitzada no s'incrementés durant l'execució. Aquesta segona passada em va permetre saber que l'aplicació, un cop carregada, mantindria un ús de memòria estable.

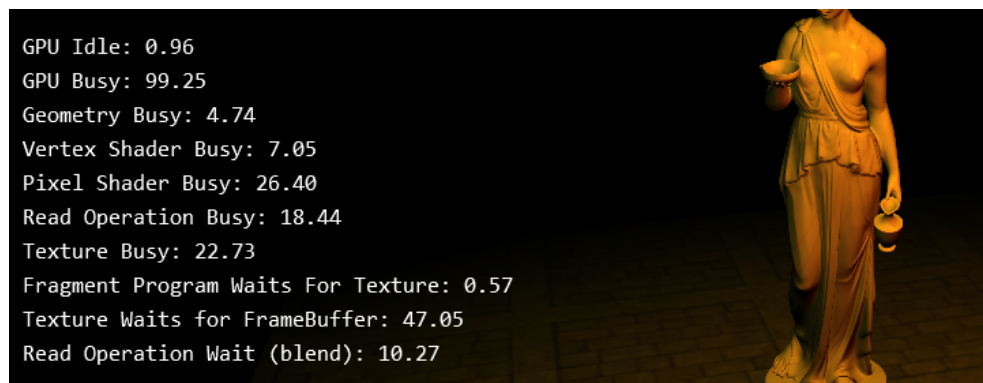


Figura 10.6: Profiler de la GPU. Mentre s'executa l'aplicació, és pot consultar a la part inferior esquerra de la pantalla

La següent eina, és un profiler de CPU (és a dir, pel codi que s'executa a la CPU), del qual també hem parlat al capítol anterior, així que ens centrarem en el seu ús. Vaig afegir marques a tot el sistema de càrrega, per identificar quines parts de la càrrega eren les més lentes. Una vegada identificades aquestes parts, es va fer un anàlisi curós de possibles solucions, i es van optimitzar. Durant el procés d'optimització s'anava comprovant que aquesta tingués l'efecte desitjat, és a dir, que el codi funcionés més ràpid. Després de cada passada d'optimització, es tornava a avaluar el resum de rendiment del profiler. Un cop vaig creure que el rendiment del sistema de càrrega era prou ràpid, vaig comprovar el sistema de pintat, però com no fa cap tipus de procés complex (a nivell de CPU), no hi havia cap lloc on optimitzar pogués donar un resultat notable.

L'última eina és un profiler de GPU. Un profiler de GPU és lleugerament diferent d'un de CPU, doncs ens dóna informació de càrrega de les fases de la pipeline de la GPU (Figura 10.6). Aquest profiler utilitza una llibreria de NVidia que ens permet llegir aquesta informació. Utilitzant aquesta informació es pot optimitzar els algorismes: per exemple, si veiem que la GPU no està ocupada quasi al 100%, vol dir que no li estem donant dades suficients perquè processí. O si veiem que el *pixel shader* és el coll de botella, podem intentar fer càlculs al *vertex shader*. Aquest profiler també és útil per al mòdul de comparativa d'algorismes, doncs em permet comparar quines fases són el coll de botella, per quins algorismes.

Un cop acabada aquesta tasca, el projecte havia vist diverses rondes d'optimització, amb la qual cosa la càrrega era quasi un ordre de magnitud més ràpida, s'havien eliminat casos on es reservava memòria i no s'alliberava, i s'havien analitzat, i corregit on va ser possible, els colls de botella dels algorismes de shading.

### 10.1.9 Implementació de materials més complexes

Un cop arribat a aquest punt, només quedava afegir suport per materials més complexos a tots els algorismes de shading. Fins ara, cada material només es composava d'una textura (o un color pla) que definia el color de la superfície, i d'un paràmetre que en definia la brillantor. Com a objectius d'aquesta fase, era suportar una textura per modular el terme especular i normals per pixel (també anomenat **normal-mapping**), en tots els algorismes de shading.

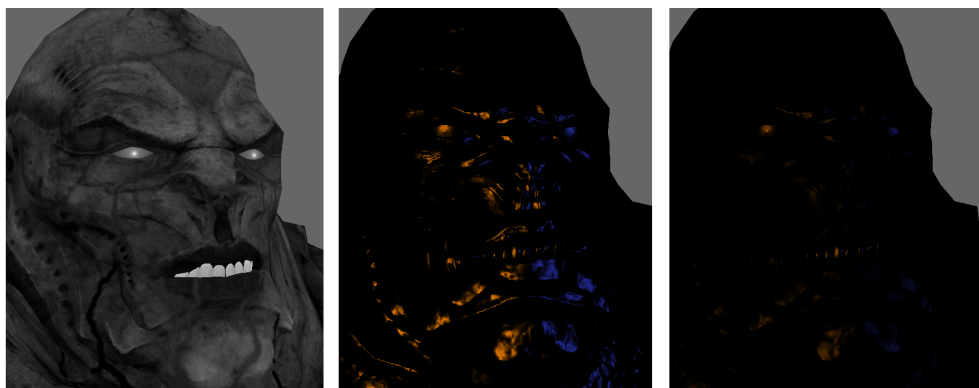


Figura 10.7: Textura per modular especular (esquerra). Sense modulació d'especular (centre). Amb modulació d'especular (dreta)

La textura per modular l'especular és important per evitar l'anomenat *efecte plàstic*. Aquest efecte es produeix quan una superfície mostra una il·luminació especular realment exagerada, la qual cosa fa que sembli de plàstic, o extremadament mullada. Una solució és utilitzar una textura que moduli el terme especular,



de manera que podem reduir la quantitat d'aquest, en certes zones. Un exemple és pot veure a la [Figura 10.7](#).

Les normals per pixel són molt més interessants. Ens permeten obtenir il·luminació d'altra freqüència respecte una geometria amb pocs triangles. Explicat d'una altra manera, ens permet obtenir una il·luminació que simula una geometria molt detallada. De forma simplificada, la tècnica es basa en tenir una textura que defineix les normals, que consultarem per pixel, en comptes d'utilitzar la normal que ens arriba interpolada del *vertex shader*. No explicaré els detalls específics de la tècnica, doncs son complexes i fora de l'abast del treball, a més, hi ha diverses maneres d'implementar un sistema com aquest. Una bona referència és el **Real-Time Rendering** [1]. A la [Figura 10.8](#) podem veure la diferència en la qualitat de la il·luminació que aporten les normals per pixel.

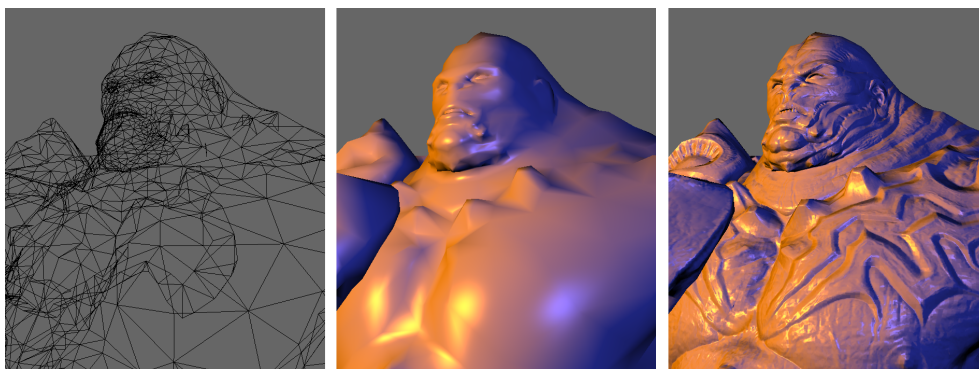


Figura 10.8: Geometria que compona l'objecte (esquerra). Il·luminació amb normals per vèrtex (centre). Il·luminació amb normals per pixel (dreta)

Supportar aquests nous tipus de materials, a tots els algorismes de shading, és difícil. El principal problema, és que tots aquells que utilitzen un GBuffer (recordem, Deferred Shading, Deferred Lighting i Inferred Lighting) sofreixen problemes de precisió. Aquest problema de precisió no es noten amb il·luminació de baixa freqüència, és a dir, utilitzant normals per vèrtex, però si utilitzant normals per pixel. A tal efecte, buscant possibles solucions, vaig acabar utilitzant una codificació anomenada **Best Fit Normals** [13], que permet mantenir millor precisió, que també millora la qualitat quan tenim normals per vèrtex.

Un vegada acabada la implementació d'aquests materials, tenia el motor 3D preparat per a portar a terme les comparatives, doncs ja suportava els quatre algorismes de shading, la càrrega de Collada, materials complexos i un sistema de càmera. A més, durant el procés, vaig acabar de corregir totes les possibles diferències de dibuixat entre els algorismes.



### 10.1.10 Sistema de comparatives

El sistema de comparatives va ser l'últim a desenvolupar-se, doncs necessitava que la resta del projecte estigués molt avançada. El sistema de comparatives fa dos tasques: crea casos d'estrès pels algorismes de shading i guarda l'estat del motor 3D. Ja hem discutit el sistema d'estrès, que s'ocupa de generar diverses quantitats de llum a l'escena, així que ens centrarem en la part de guardar l'estat del motor.

Per tal de comparar el rendiment i els colls de botella del motor, he seleccionat una sèrie de valors de l'estat del motor. Aquests valors provenen tant de la configuració d'escena (nombre de llums, nombre de materials, nombre de triangles) com del profiler de GPU (càrrega de les diferents fases de la GPU).

El sistema de comparatives, permet guardar l'estat en un moment determinat, i guardar-lo en un arxiu CSV, que després es pot carregar amb qualsevol aplicació de fulles de càlcul. Per fer el procés menys farragós, he inclòs al mòdul de comparatives un sistema de *test automàtics*.



Figura 10.9: Exemple de llums generades amb el test automàtic

El test automàtic va generant configuracions de llums, incrementant el seu nombre a cada generació. Per exemple, podem definir un test automàtic que generi totes les configuracions entre 1 llum i 64 llums, en increments de 4 en 4. Abans de generar una nova configuració de llums, es guarda l'estat del motor. Un cop generada una nova configuració de llums, el test espera un temps per tal que el motor s'estabilitzi, i torna a generar una nova configuració. Quan el test ha acabat, guarda les dades en un CSV i atura el test. A l'exemple anterior, després d'haver guardat l'estat corresponent a una configuració de 64 llums, el

test finalitza.

A més, el test automàtic anomena els fitxers CSV amb una composició del nom de l'algorisme i del nom del fitxer Collada carregat, amb la qual cosa es poden identificar fàcilment. Utilitzant aquests CSVs, he creat els gràfics que es poden veure a l'apartat de resultats (10.2).

## 10.2 Resultats

A nivell de resultats, repassarem el projecte respecte als objectius plantejats a la introducció.

### 10.2.1 Creació d'un motor 3D

Hem aconseguit un motor 3D amb un rendiment realment bo, i que és una bona base per a futures ampliacions. Té eines de depuració que permeten identificar ràpidament errors, tant siguin en la programació de shaders, codi C++ o en l'ús de memòria. El motor permet determinar quins objectes no són visibles i controlar una càmera a l'espai 3D amb total llibertat. Hi ha una clara separació entre els tipus de dades del motor i les provinents del carregador, amb la qual cosa podem canviar l'actual carregador o afegir-ne de nous.

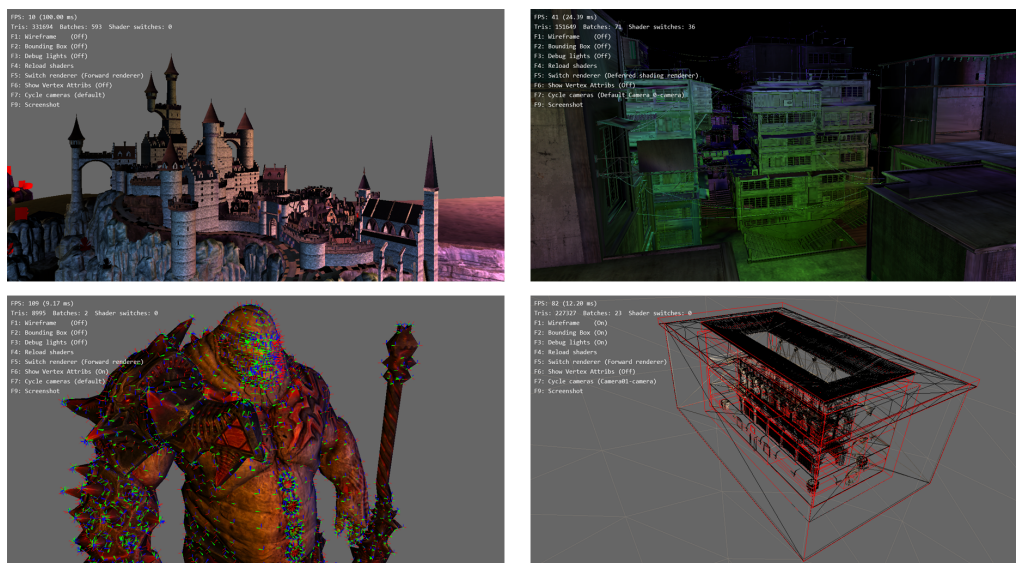


Figura 10.10: Càrrega d'escenes complexes (imatges superiors). Eines de depuració gràfica (imatges inferiors)

Considero l'objectiu assolit, és un motor 3D molt interessant i ampliable, per haver estat desenvolupat per una sola persona.

### 10.2.2 Utilitzar tecnologies desconegudes

Durant el desenvolupament del projecte, he hagut de familiaritzar-me, principalment, amb dos tecnologies que desconeixia: l'estàndard Collada, i Direct3D 9.

Respecte al primer, Collada, és el format de dades que el projecte carrega. He desenvolupat un carregador que suporta geometries descrites de diferents formes, diferents tipus de materials, gran quantitat de propietats per vèrtexs, llums, animacions, càmeres i jerarquia de l'escena. Durant el procés de crear el carregador, he arribat a conèixer el format en profunditat, a més de conèixer les aplicacions que el poden exportar i l'estat d'aquests exportadors. A més, el carregador creat té un rendiment excel·lent, trigant menys de 15 segons a carregar escenes de 200MB, que és un molt bon resultat.

Durant el desenvolupament d'aquest carregador, he pogut aprendre quins problemes té Collada i els seus exportadors, la qual cosa plasmaré a l'apartat de conclusions.

La segona tecnologia, Direct3D 9, és la llibreria gràfica sobre la qual funciona tot el projecte. En veritat, com ja he explicat, el motor 3D utilitza una abstracció damunt de Direct3D 9, però igualment he hagut d'aprendre com funciona la llibreria gràfica.

Durant el procés, m'he familiaritzat amb la extensa i excel·lent documentació que Microsoft ofereix (a la que es pot accedir online i offline) i els exemples que proporciona dins del paquet de desenvolupament. A més, he pogut contrastar la llibreria amb una altra que ja coneixia, OpenGL. El procés m'ha permès entendre els punts bons i dolents d'aquesta llibreria, que es distribueix en revisions tancades, sense mecanismes d'ampliació. A més té una interfície C++, i un sistema de control d'errors molt diferent a OpenGL.

Considero que el treball amb les dos tecnologies ha estat molt enriquidor. Crec que sense desenvolupar un projecte com aquest no podria haver après com funcionen les dos llibreries, ni podria fer-ne una crítica acurada, posició en la qual crec estar ara. Així doncs, considero aquest objectiu assolit.

### 10.2.3 Implementar 4 algorismes de shading i comparar-los

Els objectius en aquest cas eren tres: implementar els algorismes, aconseguir que les diferències visuals entre ells siguessin mínimes i comparar-los. Anem punt per punt.

El primer objectiu era aconseguir implementar els quatre algorismes. He aconseguit crear implementacions dels quatre algorismes, que renderitzen materials força complexes. A més, he intentat optimitzar-los el màxim possible. A més, he separat la implementació dels quatre algorismes de la resta del motor 3D, per tal que sigui fàcil estudiar-los. Considero aquest primer objectiu complert.

El segon objectiu, aconseguir que les diferències visuals entre ells siguin mínimes, ha estat possiblement el més complex de portar a terme. Un dels avantatges de tenir quatre algorismes de shading diferents, és que pots comprovar errors comparant-los entre ells, si tens la certesa que almenys un és correcte. Això a vegades ha jugat en contra meva, doncs la implementació que considerava de 'referència' no era correcta. A més, els quatre algorismes son molt diferents, per definició, especialment respecte la precisió de moltes dades, la qual cosa complica força la implementació. Igualment, com es pot veure a la Figura 10.11, crec que he aconseguit uns molt bons resultats, i considero l'objectiu també assolit.

Finalment, per la comparativa, he creat un sistema que em permet guardar un resum de l'estat del motor en qualsevol moment. Podem guardar diversos estats, i escriure'ls a disc en format CSV. Com a utilitat, he creat un sistema per automatitzar els tests de rendiment, de manera que podem provar diferents escenes i algorismes en un entorn 100% reproducible, la qual cosa proporciona dades perfectes per comparar. A nivell d'exemple, podem veure tant la Figura 10.12 com la Figura 10.13, gràfics creats en base als CSV exportats pel projecte. Considero que el sistema creat és un molt útil per crear comparatives, i per tant, assolit aquest objectiu.

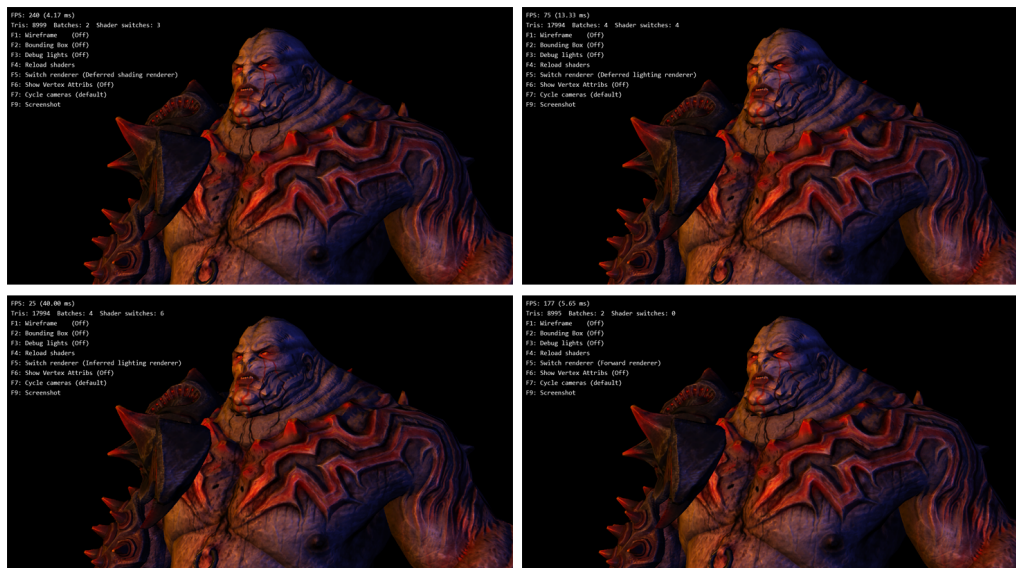


Figura 10.11: Els quatre algorismes de shading dibuixant un objecte amb un material complex. Deferred Shading (superior-esquerra), Deferred Lighting (superior-dreta), Inferred Lighting (inferior-esquerra) i Forward Shading (inferior-dreta)

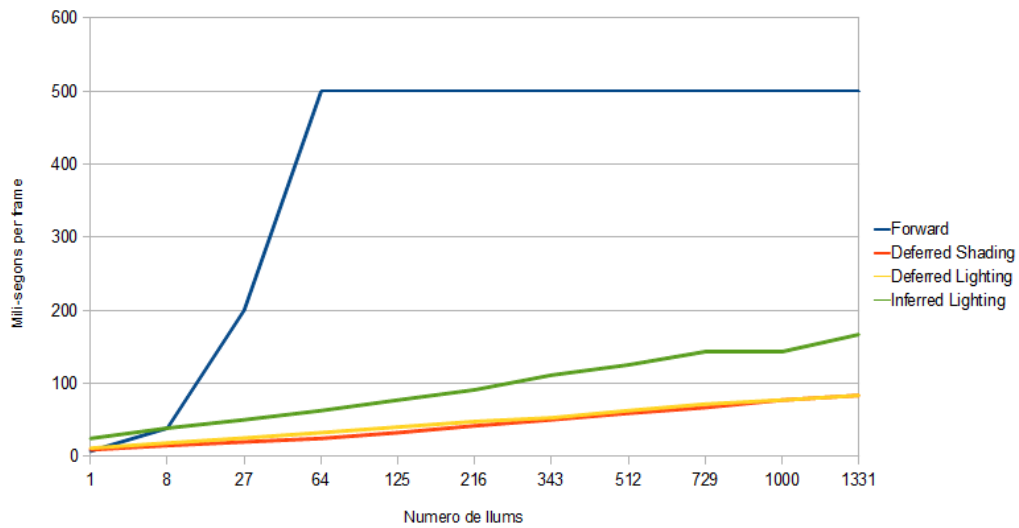


Figura 10.12: Gràfica que relaciona el temps que triga a dibuixar-se una imatge amb el nombre de llums, pels quatre algorismes

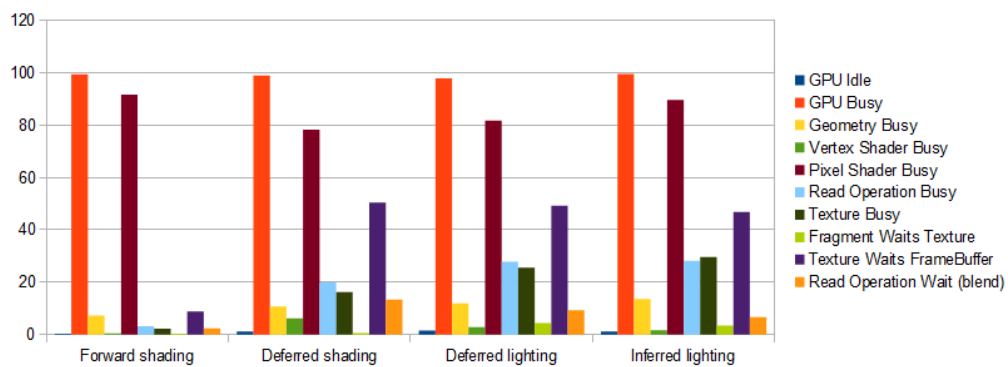


Figura 10.13: Per una mateixa configuració llums i escena, us de la GPU dels quatre algorismes

## Capítol 11

---

# Conclusions

---

En aquest capítol discutiré els requisits vistos al capítol 6, respecte a l'aplicació que he desenvolupat. Detallaré, requisit per requisit, que he desenvolupat i perquè crec que l'he assolit o no. Finalment, com a colofó a l'anàlisi d'aquest capítol, i al que he fet al capítol 10 sobre els objectius, extraure una sèrie de conclusions sobre el projecte desenvolupat i les tecnologies utilitzades.

### 11.1 Assoliment dels requisits

Anem a discutir els requisits seguint el mateix esquema utilitzat al capítol 6. Enumeraré els requisits de cada un dels punts, discutiré el que he desenvolupat en relació a aquest, i explicaré perquè crec que compleix el requisit. Cal notar que aquesta és una visió del projecte respecte als requisits, que no ha de complir cronològicament el desenvolupament del projecte, anàlisi que ja hem fet al capítol 10.

#### 11.1.1 Càrrega de dades

Els requisits eren els següents:

1. Càrrega d'un format estàndard
2. Lectura de diferents tipus d'elements
3. Conversió de dades
4. Càrrega ràpida

El sistema de càrrega de dades, carrega arxius en el format Collada, que és un esquema de dades codificat amb XML. El format Collada és un estàndard

promogut per **Sony** i el grup **Khronos**. El format suporta molts tipus d'elements diferents, com per exemple, objectes, llums, càmeres o jerarquies.

El carregador creat és capaç de llegir objectes, llums, càmeres, jerarquies i animacions. A més, processa les dades dels objectes, animacions i jerarquies, perquè siguin dibuixables de manera fàcil i òptima. El carregador s'ha provat amb quasi 50 arxius, de diferents fonts, per tal d'assegurar-ne la estabilitat i la capacitat de carregar arxius creats de diferents formes. A més, s'ha dedicat una gran quantitat de temps a optimitzar el codi, amb la qual cosa és capaç de carregar arxius de 200MB de dades en menys de 15 segons, a l'equip de desenvolupament (descriu al capítol 7).

Considero que he complert tots els requisits, de fet, he sobrepassat per molt el segon requisit.

### 11.1.2 Sistema de càmera

Els requisits eren els següents:

1. Llegir les entrades d'usuari i moure la càmera
2. Generar una descripció de la càmera

El sistema de càmera creat permet utilitzar el ratolí per rotar la càmera, i el teclat per desplaçar-se en l'espai. El desplaçament sempre és respecte a la orientació actual de la càmera. A més, per tal de no destorbar durant el desenvolupament, el sistema s'activa i es desactiva amb una tecla, de manera que és difícil canviar la posició de la càmera accidentalment.

El mòdul de càmera permet que el client d'aquest en configuri les propietats, com la obertura de la càmera o la posició inicial, entre altres. També ofereix les matrius de càmera i projecció, per tal que el client de la classe pugui convertir objectes 3D a l'espai de projecció de la càmera. Aquestes matrius son una descripció molt compacta de la configuració de la càmera.

Considero que he complert els requisits, doncs sobrepassen les necessitats inicials.

### 11.1.3 Implementació d'algorismes de shading

Els requisits eren els següents:

1. Abstracció de la llibreria gràfica
2. Implementació de l'abstracció gràfica amb una llibreria gràfica
3. Implementació d'un sistema de depuració gràfica

4. Implementació d'un sistema per utilitzar les dades provinents del mòdul de càrrega
5. Implementació dels algorismes no relacionats amb shading
6. Implementació dels algorismes de shading

Després de crear un primer esbós del sistema de dibuixat, amb Direct3D 9, s'ha dissenyat una capa d'abstracció, per tal que el motor 3D no utilitzés directament la llibreria gràfica, i per tant, fos fàcil canviar-la. Un cop creada la capa d'abstracció, s'ha programat una implementació utilitzant Direct3D 9. Mitjançant aquest mecanisme, el motor ha pogut utilitzar una interfície molt senzilla per enviar comandes al hardware 3D.

Posteriorment, s'ha creat el codi perquè el hardware 3D pogués utilitzar les dades provinents del sistema de càrrega. Com que les dades ja venien en un format adequat, aquest codi es dedica principalment a copiar les dades a la memòria del hardware 3D.

Amb les dades copiades a la targeta 3D, he implementat el primer algorisme de shading, **Forward Shading**. Per tal que la implementació dels algorismes de shading fos el més senzilla possible, s'han dissenyat per fer únicament les operacions estrictament relacionades amb cada algorisme. Junt a la implementació bàsica d'aquest, s'ha dissenyat i implementat un sistema que permet recarregar shaders i canviar entre els algorismes de shading. Per ajudar al desenvolupament, s'ha creat un mode que només mostra les arestes dels objectes, i un altre que mostra els volums de les llums.

Un cop amb el primer algorisme de shading creat, s'ha implementat un algorisme de determinació de visibilitat, el **Frustum Culling**. Aquest algorisme funciona de forma completament deslligada als de shading. Per depurar el funcionament del Frustum Culling, s'ha implementat un sistema per dibuixar les capses dels objectes en pantalla.

Finalment, s'han implementat les versions bàsiques dels tres algorismes de shading restants, és a dir, **Deferred Shading**, **Deferred Lighting** i **Inferred Lighting**. Un cop aquestes implementacions han assolit el mateix nivell que de Forward Shading, s'han implementat materials més complexes a tots els algorismes, de forma paral·lela, assegurant en tot moment que les diferències visuals entre ells fossin mínimes. Durant el procés s'ha utilitzat de forma extensiva la recarrega de shaders, demostrant-ne la utilitat.

Considero els requisits complerts, doncs he creat una implementació dels algorismes de shading molt ràpida, i amb unes diferències visuals mínimes entre ells, a més del codi de motor necessari per implementar-los.



### 11.1.4 Comparació d'algorismes de shading

Els requisits eren els següents:

1. Poder guardar dades de rendiment gràfic
2. Poder guardar comptadors d'elements del mòdul de dibuixat
3. Poder capturar la imatge generada pel mòdul de dibuixat

En aquest apartat hi ha hagut una petita desviació, doncs la captura d'imatges s'ha creat durant el desenvolupament dels algorismes de shading. Això respon a la necessitat de mantenir un històric del projecte per elaborar la present memòria.

A part d'aquesta desviació, s'ha creat un mòdul, que utilitzant una llibreria externa, permet consultar l'estat de les diferents fases de la pipeline de la GPU. Durant el desenvolupament del motor 3D s'han creat comptadors per assegurar el bon funcionament del motor i el carregador de dades (per exemple, per comprovar el nombre de triangles i vèrtexs d'un objecte).

Amb els dos orígens de dades citats (el mòdul de consulta de la GPU i els comptadors), ja tenia totes les dades necessàries per poder comparar els algorismes. Per tal de facilitar la comparació, s'ha afegit funcionalitat per guardar les dades en format CSV, pel seu posterior anàlisi en qualsevol aplicació de fulles de càlcul. A més, per que la creació d'aquestes dades fos encara més senzilla, s'ha creat un sistema de tests automàtics.

Tot i la desviació, he complert els objectius de forma plena, fins i tot creant utilitats per a fer més còmoda l'extracció de dades.

## 11.2 Conclusions del projecte

Acabem de veure que he complert tots els requisits i els motius pels quals considero que és així. Un cop he fet aquest anàlisi, m'agradaria discutir el projecte, les decisions preses al principi, i el que he après durant el procés.

Em centraré en les dos parts que em van motivar més a començar el projecte, i que per tant, tenien major interès per mi: aprendre noves tecnologies, i analitzar a fons quatre algorismes de shading. Respecte les noves tecnologies, discutiré el format Collada i la llibreria Direct3D 9. Respecte els quatre algorismes de shading, en faré una comparativa distesa, centrant-me en la dificultat d'implementació, la qualitat visual i la capacitat de millora dels algorismes.

Finalment faré uns petits apunts generals.

### 11.2.1 Collada

El format Collada es va crear amb la intenció de ser un estàndard de referència d'intercanvi de dades entre aplicacions de modelat 3D, però no ha acabat

d'aconseguir-ho. El principal problema és que la seva especificació és molt poc detallada en diferents apartats, però resulta especialment dolent en el cas dels rangs de dades. En cap moment s'especifica els rangs de possibles valors, la qual cosa fa impossible l'intercanvi exacte de dades entre dos aplicacions de modelat 3D diferents.

Respecte al seu us al dia a dia, el fet que la especificació no tingui una versió binària és un inconvenient important, si considerem que escenes relativament senzilles poden ocupar desenes de megabytes. Això és problemàtic no només a nivell de l'espai de disc que requereixen, sinó també per la memòria necessària per carregar-los i el temps de procés per a convertir-los a una representació més compacta. En comparació, el format **FBX** (un format propietat d'**Autodesk**), ofereix la possibilitat d'exportar les dades en format binari o text.

Finalment, el major problema, és que les companyies que hi ha rere aquest estàndard, només han creat l'especificació. No han col·laborat amb els desenvolupadors encarregats de crear els exportadors des de les eines de modelat 3D. El resultat d'això és que la qualitat dels exportadors i els fitxers que exporten, és molt irregular. Durant les proves del carregador de Collada, vaig trobar diversos fitxers (creats amb un exportador molt popular) que incomplien completament l'estàndard.

En resum, ha estat molt interessant aprendre sobre aquest format, però no en recomanaria l'ús en cap projecte de mida mitja o gran. En comptes, recomanaria que s'avalués el format FBX, o bé es creessin exportadors propis.

### 11.2.2 Direct3D 9

Al moment de començar el projecte, ja existia Direct3D 10 i Direct3D 11, però vaig decidir utilitzar Direct3D 9. La decisió es va prendre després de mirar diferents estadístiques de hardware, on Direct3D 10 i Direct3D 11 encara eren suportats per una porció petita dels equips, en comparació a Direct3D 9.

El principal problema és que Direct3D no té un mecanisme d'extensions, com té OpenGL. Això vol dir, que Direct3D 9 ofereix una llista de característiques tancada, amb una especificació molt detallada. Qualsevol cosa fora d'aquesta especificació no és suportada, i s'ha de canviar a la versió que si ho suporta. L'inconvenient és que normalment hi ha canvis radicals de l'interfície entre versions, amb la qual cosa el canvi és una operació delicada i que requereix molt de temps. Tant és així, que Microsoft ofereix guies (força llargues) per convertir el codi d'una versió a la següent.

Durant el desenvolupament del projecte m'he trobat en alguna ocasió amb una limitació que no em permetia avançar. Per sort, els fabricants de hardware han creat mecanismes 'no oficials' de poder utilitzar alguna funcionalitat fora de l'especificació. [24]

Si tornés a començar el desenvolupament, segurament utilitzaria Direct3D 10 o Direct3D 11, especialment la segona, doncs crec que son millors inversions del meu temps.

### 11.2.3 Algorismes de shading

Fent un anàlisi ràpid, el resum és que si hem de dibuixar escenaris amb poques llums, la millor elecció és el **Forward Shading**, i si hem de dibuixar moltes llums, el **Deferred Shading**. Anem a analitzar els altres dos algorismes, i perquè crec que son pitjors opcions.

El **Deferred Lighting** es basa en utilitzar menys memòria, per tal de guanyar velocitat i reduir requisits. El problema és que ho fa a costa d'haver de dibuixar tots els objectes dos vegades, un cop durant la creació del GBuffer, i a la fase final d'aplicar l'il·luminació. A més, tampoc utilitza molta menys memòria perquè, idealment, hem de mantenir dos buffers, per mantenir el terme difós i el terme especular. Tal com es pot veure a la gràfica comparativa del capítol 10, normalment té pitjor rendiment que el Deferred Shading. Vaig provar de codificar la informació d'il·luminació en un sol buffer, però els errors d'il·luminació eren notables i ho vaig haver de descartar. És una petita decepció, doncs era una tècnica que m'interessava molt. Seguiré investigant-la, per si existeixen detalls d'implementació que no conec.

El **Inferred Lighting** crec, directament, que és una mala tècnica. El guany teòric que es produiria al calcular la il·luminació en una resolució menor, es perd al filtrar els buffers en la passada final que l'aplica. A més, provoca errors molt visibles amb objectes amb normals per pixel. Finalment, si mirem la gràfica comparativa del capítol anterior, en cap moment té millor rendiment que els altres dos algorismes dissenyats per a una gran quantitat de llums, és a dir, el Deferred Shading i el Deferred Lighting.

En canvi, el **Forward Shading** és molt fàcil d'implementar, i és perfecte per fer proves. És molt directe, doncs al mateix moment s'avalua tota l'equació de shading. Té com a principal problema que escala molt malament respecte al numero de llums. Si el nombre de llums que ha de suportar un motor 3D és reduït, és l'elecció clara.

L'altra bona opció és el **Deferred Shading**, tot i que és més complex d'implementar que l'anterior. Té un cost base força elevat, que correspon a omplir el GBuffer, però a partir d'aquest cost escala molt bé, com es pot veure a la gràfica comparativa del capítol anterior. És un algorisme molt interessant d'implementar i optimitzar, doncs estressa moltes parts de la pipeline més complexes d'entendre que el Forward Shading, la qual cosa suposa haver d'aprendre molt més sobre com funciona una GPU. A més, convertir un renderer que utilitza Deferred Shading a Deferred Lighting (o Inferred Lighting), és molt menys complicat que no pas la transició de Forward a Deferred Shading.

Tot i això, seguiré llegint sobre aquestes tècniques, i altres que no entraven al projecte i comento al capítol 12. Ni que alguns dels algorismes de shading que he implementat no han resultat donar el rendiment esperat, crec que és molt valuós tenir una opinió documentada dels problemes que tenen.

#### 11.2.4 Conclusions generals

Finalment, voldria posar per escrit l'opinió general sobre el desenvolupament del projecte, doncs crec que és interessant, després de les meves decisions inicials i els resultats finals.

Crec que desenvolupar aquests tipus de projectes es molt important, doncs s'aprèn les bases dels motors 3D. Tot i que moltes vegades, per desenvolupar un joc, és millor opció utilitzar un motor ja creat i amb bones eines, conèixer com funciona teòricament un motor 3D per sota és molt important. Ni que finalment acabem utilitzant un motor 3D no propi, aquests coneixements permeten fer-ne un millor us.

A nivell de les decisions, tot i creure que l'elecció de Collada i Direct3D 9 no van ser les millors, però com he dit al principi de la memòria, només la creació d'un projecte d'aquestes dimensions permet saber-ho del cert, així que estic content d'aquestes decisions. Quan comenci un altre projecte, no tornaré a utilitzar aquestes tecnologies, però ara tinc prou informació per saber perquè no utilitzar-les.

## Capítol 12

---

# Treball futur

---

Les possibles vies d'ampliació d'un motor 3D, son pràcticament infinites. Crearé una petita llista de les possibles característiques o algorismes que inclouria al projecte si el seguís desenvolupant:

- **Suport per transparències:** Ara mateix no es suporten objectes transparents i seria interessant implementar-ho. A l'algorismes de Forward és trivial de fer, als altres és bastant més complexe, sobretot al d'Inferred Lighting.
- **Suport per anti-alias:** L'anti-alias és un procés que redueix les 'vores dentades' dels objectes. Aquestes vores dentades son molt visibles, especialment quan la càmera o els objectes es mouen, així que es desitjable implementar tècniques que ho redueixin. La millor opció seria implementar un dels filtres de post-procés que ara utilitzen la majoria de jocs, doncs funcionarien en qualsevol algorisme de shading, i es podria implementar sense tocar les implementacions d'aquests.
- **Suport d'ombres:** Ara mateix no hi ha suport per a llums que projecten ombres. La recerca sobre el dibuixat d'ombres és molt interessant i un tema molt ampli, que donaria per un projecte complet.
- **Renderitzat físicament realista:** L'actual implementació d'il·luminació és una aproximació molt senzilla a com funciona la interacció entre llum i material. El renderitzat físicament realista intenta modelar millor el comportament real de la llum. És un camp de recerca també molt interessant, que també donaria per un projecte complet.
- **Altres algorismes de shading:** Durant la creació d'aquest projecte, han aparegut nous algorismes de shading, com per exemple **Forward+** [16], o

el **Clustered Forward i Deferred Shading** [18], que serien interessants d'implementar.

Altres idees de futur que no necessiten explicació, son la creació d'una implementació de la llibreria gràfica utilitzant Direct3D 11, que el mòdul de comparatives pugui generar gràfiques o suport per carregar altres tipus d'arxius, fins i tot, utilitzant alguna llibreria com **assimp** [6].

---

# Bibliografia

---

- [1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008. [cited at p. 26, 29, 39, 46, 82, 94]
- [2] James F. Blinn. Models of light reflection for computer synthesized pictures. *ACM Computer Graphics (SIGGRAPH '77)*, 11(2):192–198, July 1977. [cited at p. 49]
- [3] James F. Blinn. *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*. Morgan Kaufmann, 1996. [cited at p. 37]
- [4] Ignacio Castano and Simon Brown. Nvidia texture tools. <http://code.google.com/p/nvidia-texture-tools/>. [cited at p. 60]
- [5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, January 1995. [cited at p. 67]
- [6] Alexander Gessler, Thomas Schulze, and Kim Kulling. Assimp: Open asset import library. <http://assimp.sourceforge.net/>. [cited at p. 108]
- [7] J. Gregory. *Game Engine Architecture*. Ak Peters Series. A K Peters, 2009. [cited at p. 82]
- [8] Khronos Group. Collada. <http://www.collada.org>. [cited at p. 61, 65]
- [9] Khronos Group. Collada specification 1.4.1. [http://www.khronos.org/files/collada\\_spec\\_1\\_4.pdf](http://www.khronos.org/files/collada_spec_1_4.pdf). [cited at p. 86]
- [10] Khronos Group. Opengl api. <http://www.opengl.org/documentation/>. [cited at p. 59]
- [11] Intel. Intel vtune. <http://software.intel.com/en-us/intel-vtune-amplifier-xe/>. [cited at p. 80]

- [12] Nicolai M. Josuttis. *The C++ standard library: a tutorial and reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. [cited at p. 67]
- [13] Anton Kaplanyan. Cryengine 3: reaching the speed of light. [http://www.crytek.com/download/AdvRTRend\\_crytek.ppt](http://www.crytek.com/download/AdvRTRend_crytek.ppt). [cited at p. 94]
- [14] Arseny Kapoulkine. pugixml. <http://www.pugixml.org>. [cited at p. 62]
- [15] Scott Kircher and Alam Lawrance. Inferred lighting: Fast dynamic lighting and shadows for opaque and translucent objects. [http://www.dsvolution.com/gdc/downloads/SIGGRAPH2009\\_KircherLawrence\\_InferredLighting.zip](http://www.dsvolution.com/gdc/downloads/SIGGRAPH2009_KircherLawrence_InferredLighting.zip). [cited at p. 77]
- [16] Jay McKee. Technology behind amd's - leo demo. [http://developer.amd.com/wordpress/media/2012/10/AMD\\_Demos\\_LeoDemoGDC2012.ppsx](http://developer.amd.com/wordpress/media/2012/10/AMD_Demos_LeoDemoGDC2012.ppsx). [cited at p. 107]
- [17] NVidia. Nvidia perfkit. <http://developer.nvidia.com/nvidia-perfkit>. [cited at p. 62]
- [18] Ola Olsson, Markus Billeter, and Ulf Assarsson. Clustered deferred and forward shading. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics conference on High-Performance Graphics, EGGH-HPG'12*, pages 87–96, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association. [cited at p. 108]
- [19] Stephen Palmer and Mac Felsing. *A Practical Guide to Feature-Driven Development*. Prentice Hall, 2002. [cited at p. 17]
- [20] Matt Pettineo. Scintillating snippets: Reconstructing position from depth, Mach 2009. <http://mynameismjp.wordpress.com/2009/03/10/reconstructing-position-from-depth/>. [cited at p. 73]
- [21] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010. [cited at p. 46]
- [22] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975. [cited at p. 49]
- [23] Aras Pranckevičius. Compact normal storage for small g-buffers. <http://www.aras-p.info/texts/CompactNormalStorage.html>. [cited at p. 74]
- [24] Aras Pranckevičius. D3d9 gpu hacks. <http://aras-p.info/texts/D3D9GPUHacks.html>. [cited at p. 104]



- [25] Fabien Sanglard. Quake 2 source code review 4/4, September 2011. [http://www.fabiensanglard.net/quake2/quake2\\_opengl\\_renderer.php](http://www.fabiensanglard.net/quake2/quake2_opengl_renderer.php). [cited at p. 28]
- [26] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009. [cited at p. 59]
- [27] Lee Thomason, Yves Berquin, and Andrew Ellerton. Tinyxml. <http://www.sourceforge.net/projects/tinyxml/>. [cited at p. 62]
- [28] Denton Woods, Nicolas Weber, and Meloni Dario. Openil / devil. <http://openil.sourceforge.net>. [cited at p. 60]

---

# Índex de figures

---

1.1	Exemple d' <i>uncanny valley</i> (esquerra) i estil no realista (dreta) . . .	1
1.2	Exemple de reproduccions de diferents tipus de materials . . . . .	4
1.3	Editor del motor de jocs Unity . . . . .	8
2.1	El famós model 3D Stanford Bunny . . . . .	12
2.2	Imatge creada en temps real utilitzant el primer motor comercial que vaig estar desenvolupant (circa 2007) . . . . .	14
3.1	Procés de la metodologia basada en característiques (© 2005 Ne- bulon Pty. Ltd) . . . . .	18
5.1	Projecció . . . . .	26
5.2	Pipeline . . . . .	27
5.3	Quake 2 . . . . .	28
5.4	Sub-fases de la fase de geometria . . . . .	29
5.5	Transformació de vista . . . . .	30
5.6	Transformacions de projecció . . . . .	32
5.7	Clipping . . . . .	33
5.8	Transformació a pantalla . . . . .	33
5.9	Sub-fases de la fase de rasterització . . . . .	34
5.10	Preparació de triangles . . . . .	34
5.11	Recorregut d'arestes . . . . .	35
5.12	Texturat . . . . .	35
5.13	Buffer de color i de profunditat . . . . .	36
5.14	Pipeline hardware . . . . .	37
5.15	Editor de grafs per a generar shaders . . . . .	39
5.16	Exemple de geometria deformada segons un esquelet . . . . .	40
5.17	Exemple de múltiples sortides per un únic pixel shader . . . . .	41
5.18	Exemple d'il·luminació . . . . .	42
5.19	Llum direccional . . . . .	43

5.20	Llum omnidireccional . . . . .	43
5.21	Llum spot . . . . .	44
5.22	Difusió: reflexió i refracció . . . . .	44
5.23	Interaccions llum-superfície . . . . .	45
5.24	Diferencia entre no ordenar transparències (esquerra) i ordenar-los (dreta) . . . . .	45
5.25	Les dos parts de l'iluminació, terme difós (esquerra) i terme especular (dreta) . . . . .	46
5.26	La llei de Lambert des d'un punt de vista geomètric . . . . .	47
5.27	Component difós (esquerra) i component difós i especular (dreta) . . . . .	48
5.28	Vectors utilitzats per calcular el terme especular . . . . .	49
5.29	Diferents valors de la potencia especular de menor (esquerra) a major (dreta) . . . . .	50
7.1	Exemple senzill del format XML . . . . .	61
7.2	Projecte Final de Carrera obert al Visual Studio 2010 . . . . .	63
8.1	Flux de l'aplicació a alt nivell . . . . .	66
8.2	Disseny inicial de classes de l'aplicació . . . . .	68
9.1	Una esfera il·luminada per vèrtex. D'esquerra a dreta, les esferes tenen 256, 1024 i 16384 triangles . . . . .	70
9.2	Exemple d'un dels problemes amb Forward Shading . . . . .	71
9.3	Exemple de GBuffer. Color (superior-esquerra), normals (inferior-esquerra), propietats especular (superior-dreta) i profunditat (inferior-dreta) . . . . .	73
9.4	Exemple de buffers d'acumulació. Terme difós (esquerra) i terme especular (dreta) . . . . .	75
9.5	Exemple d'errors d'interpolació utilitzant Inferred Lighting . . . . .	77
9.6	Buffer DSF que conté la profunditat discreta i un identificador d'objecte . . . . .	78
9.7	Extracte del resum creat pel profiler del projecte . . . . .	80
9.8	Capsa contenidora (esquerra). Esquema de Frustum Culling (dreta) . . . . .	82
9.9	Escena amb les llums originals (esquerra). Esquema amb llums generades, representades com esferes de colors (dreta) . . . . .	84
10.1	Primer objecte Collada dibuixat pel projecte . . . . .	86
10.2	Primer objecte il·luminat que el projecte va dibuixar . . . . .	87
10.3	Primera implementació de Forward Shading utilitzant shaders, amb suport de textures, llums i il·luminació difosa . . . . .	89
10.4	Primera versió del projecte amb capsas contenidores i Frustum Culling . . . . .	90

10.5	Primera versió de l'algorisme Deferred Shading . . . . .	91
10.6	Profiler de la GPU. Mentre s'executa l'aplicació, és pot consultar a la part inferior esquerra de la pantalla . . . . .	92
10.7	Textura per modular especular (esquerra). Sense modulació d'especular (centre). Amb modulació d'especular (dreta) . . . . .	93
10.8	Geometria que compon l'objecte (esquerra). Il·luminació amb normals per vèrtex (centre). Il·luminació amb normals per pixel (dreta) . . . . .	94
10.9	Exemple de llums generades amb el test automàtic . . . . .	95
10.10	Càrrega d'escenes complexes (imatges superiors). Eines de depuració gràfica (imatges inferiors) . . . . .	96
10.11	Els quatre algorismes de shading dibuixant un objecte amb un material complex. Deferred Shading (superior-esquerra), Deferred Lighting (superior-dreta), Inferred Lighting(inferior-esquerra) i Forward Shading (inferior-dreta) . . . . .	98
10.12	Gràfica que relaciona el temps que triga a dibuixar-se una imatge amb el nombre de llums, pels quatre algorismes . . . . .	99
10.13	Per una mateixa configuració llums i escena, us de la GPU dels quatre algorismes . . . . .	99

---

# Manual d'instal·lació i usuari

---

## Instal·lació

Els requisits dels projecte son:

- **Sistema operatiu:** Windows XP SP2 o superior (recomanat Windows 7)
- **Targeta gràfica:** NVidia Geforce 8 o superior (recomanat NVida Geforce 200 o superior)
- **RAM:** 256MB lliure (recomanat 512MB)

El projecte no necessita instal·lació. Només cal descomprimir l'arxiu ZIP amb el que s'ha empaquetat. Durant l'execució del projecte, aquest pot escriure dades a disc, així que és important que es descomprimeixi en un directori amb permisos d'escriptura. Per exemple, Windows 7 i Windows Vista protegeixen el contingut del directori **Arxius de Programa** i el directori arrel de cada disc dur, contra escriptura.

## Manual d'usuari

Per iniciar el projecte, només cal executar el binari **daenerys.exe**. El primer cop la inicialització del projecte pot ser lenta, doncs s'han de pre-processar certes dades. El projecte carrega automàticament una escena 3D, amb la qual podem navegar utilitzant la següent configuració:

- **Tecles W/S:** Avançar/retrocedir respecte a la orientació de la càmera
- **Tecles A/D:** Desplaçar-se a la esquerra/dreta respecte a la orientació de la càmera
- **Tecla Shift:** Mentre està pulsada, desactiva el bloqueig de rotació de la càmera
- **Moviment de ratolí:** Rota la càmera
- **Roda d'scroll del ratolí:** Augmenta/disminueix la velocitat de desplaçament

A més de poder moure's per la escena, podem canviar d'algorisme de shading, d'escena, càmera, i activar diferents opcions de depuració:

- **Tecles F5/Shift+F5:** Següent/previ algorisme de shading
- **Tecles E/Q:** Següent/prèvia escena
- **Tecla F1:** Activa/desactiva el mode que dibuixa només les arestes dels triangles que componen l'escena
- **Tecla F2:** Activa/desactiva el mode que mostra les capsas contenidores dels objectes
- **Tecla F3:** Activa/desactiva el mode que mostra els volums de les llums
- **Tecla F4:** Recarrega els shaders de l'actual algorisme de shading
- **Tecla F6:** Activa/desactiva el mode que mostra certes propietats de vèrtex

- **Tecla F7:** Si l'escena té més d'una càmera, permet canviar entre elles
- **Tecla F9:** Captura la imatge generada per l'algorisme de shading i la desa al sub-directori **shots**
- **Tecla F11:** Inicia un test automàtic. Aquest procés triga entre 2 i 3 minuts en completar-se.
- **Esc:** Tanca l'aplicació

El projecte té una llista tancada d'escenes, però permet carregar arxius utilitzant la línia de comandes (per exemple, *daenerys.exe elMeuArxiu.dae*), o senzillament arrossegant-los damunt l'executable del projecte.