UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Implementació de deferred shading

Autor: Albert Millàs Roura.
Data: 16 de Gener del 2015

Director: Antonio Chica Calaf
Departament: Ciències de la Computació.

Titulació: Enginyeria Informàtica.
Centre: Facultat d'informàtica de Barcelona (FIB)
Universitat: Universitat Politècnica de Catalunya (UPC)

Introducció…………………………………………………………………..3

Conceptes…………………………………………………………………....4

Forward Rendering…………………………………………………………4

Deferred Shading……………………………………………………..……..7

Forward+……………………………………………………………………10

CUDA………………………………………………………………………..12

Estructura bàsica……………………………………………………………...16

Forward…………………………………………………………………………18

Blending……………………………………………………………………...18

Texture buffer object………………………………………………………22

Deferred Shading………………………………………………………………24

Forward+…………………………………………………………………………29

CUDA………………………………………………………………………….32

Resultats…………………………………………………………………………35

Comparació de mètodes……………………………………………………35

Forward Render……………………………………………………………...40

Deferred shading…………………………………………………………….41

Forward+……………………………………………………………………..43

Bibliografia………………………………………………………………………44

Introducció

La il·luminació juga un paper molt important en donar realisme a un escenari 3D. Això fa que a vegades siguin necessàries un gran nombre de llums per poder simular una il·luminació realista, o per poder tenir molt efectes visuals concurrents.

El fet de tenir un elevat nombre de llums en una escena afecta notablement al rendiment de l'aplicació gràfica, per això s'han desenvolupat vàries tècniques que permeten augmentar el nombre de llums mantenint uns nivells d'eficiència acceptables. En aquest projecte ens centrarem a implementar i analitzar l'eficiència d'aquests algoritmes a mesura que incrementem el nombre de llums de l'escena.

Per facilitar el procés de creació de l'aplicació i poder centrar-nos en els algoritmes de render, hem utilitzat vàries llibreries. Les més rellevants són:

OpenGL	API que ens permet renderitzar escenes utilitzant la GPU.
Qt	Framework que ens permet gestionar la finestra de l'aplicació.
Assimp	Llibreria que ens facilita la importació de models 3D.
GLM	Llibreria que ens permet utilitzar funcionalitats de GLSL en la CPU.
CUDA	Conjunt d'eines de desenvolupament que ens permet utilitzar la GPU per altres tasques a part del renderitzat gràfic
Thrust	Llibreria de CUDA que implementa algoritmes generals altament optimitzats.

Taula amb un resum de les llibreries usades.

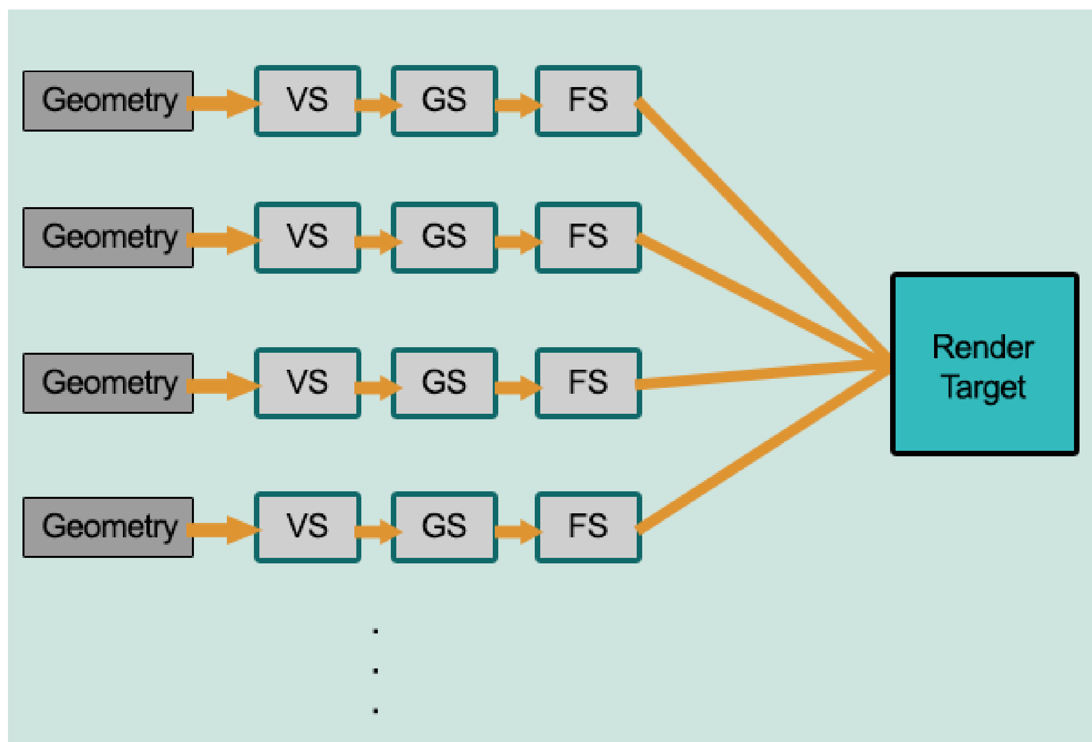
En les primeres versions de OpenGL, el pipeline gràfic era fix i la mateixa API ens donava una sèrie de crides que permetien configurar com volíem que es pintés l'escena. Entre altres opcions, podíem configurar si volíem que OpenGL apliqués il·luminació i quines llums volíem que hi hagués a l'escena. Aquest sistema era molt senzill i inicialment era suficient, però ràpidament van aparèixer limitacions importants: en ser una crida fixe el nombre de llums està molt limitat, no permetent generalment més de 8 llums simultànies.

Conceptes

Forward Rendering

Aquest és el mètode utilitzat en el pipeline fix de OpenGL per pintar una escena. És un mètode senzill però efectiu per calcular la il·luminació en escenes amb un nombre reduït de llums.

Forward rendering és un mètode que podríem anomenar **exhaustiu**, ja que bàsicament consisteix a calcular l'efecte de totes les llums per cada punt o píxel de l'escena sense excepció. Això implica que **a cada píxel de cada polígon de l'escena se li aplica l'efecte de totes les llums**, encara que aquestes visiblement afectin només una part petita de l'escena.



La geometria només passa un cop pel pipeline gràfic, l'iluminació s'efectua al Fragment Shader.

Sabent això no és difícil imaginar que el rendiment de forward rendering decaurà dràsticament a mesura que augmenti la complexitat de l'escena i el nombre de llums que conté aquesta.

OpenGL limita el seu pipeline fix en un nombre costant depenent del hardware disponible, generalment aquest valor és de 8 llums, però això no és una limitació de forward rendering en si, sinó una decisió de disseny per part d'OpenGL. Es poden fer implementacions de forward rendering amb capacitat per a més llums. Per tal de poder oferir una comparació vàlida amb altres sistemes de shading necessitem més de les 8 llums clàssiques que ofereix OpenGL, per tant hem estudiat i provat diverses implementacions que ens permeten tenir un nombre prou gran de llums i hem escollit la millor per realitzar la comparativa amb els altres algoritmes. Els mètodes que hem provat són: utilitzar solament blending, shaders amb uniforms (i blending) i texture buffers.

Una de les etapes del pipeline gràfic de OpenGL és el que es coneix com a **blending**. En aquesta etapa es **decideix com afecta el color del punt** que estem processant amb el que ja hi ha pintat al framebuffer. Tenim l'avantatge que OpenGL ens permet configurar com volem que es faci aquesta etapa, això ens pot ser molt útil, ja que ens permetrà pintar l'escena múltiples cops i anar acumulant els resultats.

En concret, la primera tècnica que vam provar consisteix a utilitzar només el pipeline fix de OpenGL amb blending. Per fer això el que fem és **pintar l'escena múltiples vegades canviant la configuració de les 8 llums en cada pintat i fer blending per anar acumulant els resultats fins a haver calculat totes les llums.**

Aquest procés ens permet tenir un nombre il·limitat de llums, el problema és que a part del cost del forward rendering, hem d'afegit tot el cost de repintar l'escena cada 8 llums i fer el blending. Aquest cost incrementa dràsticament quan tenim un nombre elevat de llums, ja que incrementem el nombre de repintats.

Això fa que aquest mètode no sigui vàlid per comparar algoritmes de shading, ja que afegim molt cost extra al forward rendering. Vist això, sembla que amb el pipeline fix no en tindrem prou, per tant haurem de **programar shaders que implementin forward rendering i permetin rebre més paràmetres.**

En la primera versió del shader vam decidir **enviar la informació de cada llum com una variable de tipus uniform.** Això donava bons resultats però ràpidament vam veure que **OpenGL limita el nombre de uniforms que podem passar al shader** i, tot i que permetia un nombre molt més elevat de llums que la versió standard (al voltant de les 100 llums), no era suficient per ser comparat amb els altres mètodes.

Una opció que vam provar va ser **utilitzar aquest shader junt amb el blending que utilitzàvem inicialment**, però tot i que millorava significativament respecte a la primera versió, seguíem afegint cost extra al forward rendering a mesura que havíem de pintar més cops l'escena per poder calcular totes les llums.

Finalment vam descobrir la manera adequada de passar tota la informació necessària al shader: **Buffer textures**. Bàsicament és un **tipus de textura que s'envia al shader com una uniform i s'utilitza com si fos un buffer, permetent enviar grans quantitats d'informació**. Aquestes textures tenen una capacitat molt gran i són més que suficients per enviar tota la informació necessària per fer la il·luminació.

Aquest últim mètode ens permet passar un gran nombre de llums al shader sense necessitat de pintar l'escena múltiples cops i fer blending, evitant afegir sobrecàrrega quan augmentem el nombre de llums. Això ens permetrà comparar amb precisió l'eficiència d'aquest mètode amb els altres.

Tot i haver superat la limitació del nombre de llums, el mètode de **forward rendering és un mètode exhaustiu el que fa que difícilment sigui prou eficient a mesura que afegim llums a l'escena**, per això presentem dos algoritmes que creiem que poden oferir millors resultats: *deferred shading* i *forward+ rendering*.

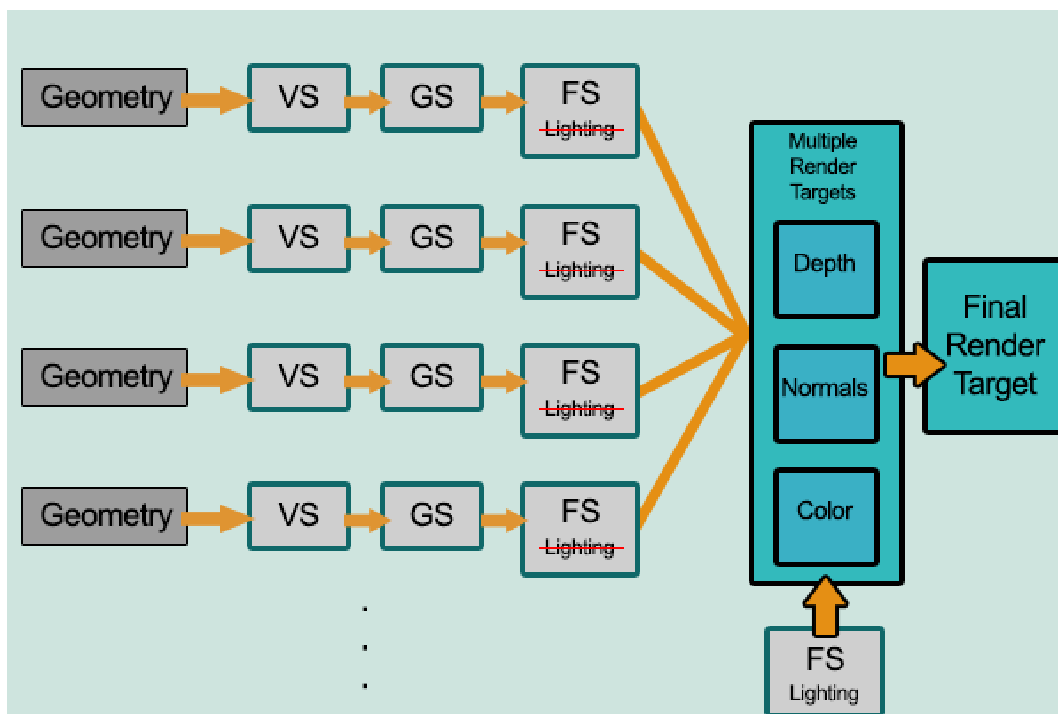
A la secció de resultats d'aquest document s'inclou un apartat que compara l'algoritme que utilitza uniforms i blending amb el que utilitza buffer textures per poder apreciar l'efecte que té el blending a l'hora de calcular la il·luminació de l'escena. Les comparacions amb la resta de mètodes es fan amb buffer textures, ja que és la versió que ha donat millors resultats.

Deferred Shading

Deferred shading és una tècnica més avançada que forward rendering que permet reduir la complexitat del procés d'il·luminació i ombrejat, especialment en escenes amb moltes llums dinàmiques. Aquest mètode s'usa habitualment per fer aplicacions gràfiques amb uns requisits d'il·luminació elevats, per exemple per fer simulacions a temps real o videojocs moderns.

Aquest mètode **consisteix bàsicament a pintar l'escena en dos torns. En el primer torn es calcula la geometria de l'escena** i se n'extreu tota la informació necessària. Posteriorment es fa **un segon torn on s'usa la informació extreta de l'escena per processar les llums**.

A la primera passada, anomenada *geometry pass*, li e**nviem als shaders la informació de la geometria de l'escena**, és a dir, els diferents polígons i vèrtexs que els formen. En aquesta passada no s'envia cap informació sobre les llums de l'escena. **Els shaders processen aquesta informació i retornen un conjunt de textures amb informació sobre els píxels visibles** per l'espectador. Aquesta informació pot ser per exemple les coordenades del píxel o el seu color.



Es coneix com a MRT o Multiple Render Targets la funcionalitat de les targetes gràfiques de retornar vàries textures d'un renderitzat.

Aquestes textures són molt útils perquè contenen tota la informació geomètrica de l'escena. Un dels principals avantatges d'aquest procés és que **ja s'han descartat tots els píxels no visibles** i només tenim informació dels píxels visibles, el que farà que tots els càlculs fets a partir d'aquestes textures siguin només sobre les parts visibles de l'escena.

La segona passada, anomenada *lighting pass*, és la que **s'encarrega d'aplicar la il·luminació**. Per aconseguir-ho **renderitzem cada llum com una esfera, calculem a quins píxels del viewport** (secció de l'escena que veu l'espectador) **afecta i apliquem els efectes de la llum** utilitzant la informació de les textures obtingudes al *geometry pass*.

En renderitzar cada llum com una esfera ens assegurem que **només apliquem els afectes de la llum en l'àrea d'influència d'aquesta** i no en tota l'escena, fet que suposa un gran avantatge si aquestes llums afecten una part reduïda del viewport.

No obstant això, aquests avantatges no venen sense **inconvenients**. Entre ells la **dificultat de fer objectes transparents**, ja que en fer el geometry pass no se sap si un objecte és transparent o no i només s'exporta la informació dels píxels més propers a l'observador. Un altre problema que hi ha amb deferred shading és que **no es poden utilitzar alguns dels algoritmes d'antialiasings** més populars. També cal remarcar que **en renderitzar l'escena dos cops estem afegint un sobrecost al procés** normal de render, fet que pot fer que en alguns casos deferred shading no obtingui els millors resultats. La majoria d'aquests desavantatges poden ser resolts.

En resum:

1- Geometry Pass	**2- Lighting pass**
Processem la geometria sense tenir en compta il·luminació	Apliquem l'efecte de cada llum als píxels pertinents

Deferred shading pinta l'escena en dues passades.

Avantatges	**Inconvenients**
- Evitem fer càlculs d'il·luminació en zones ocultes. - Evitem fer càlculs d'il·luminació en zones on la llum no afecta.	- Hem de renderitzar l'escena dos cops: implica cert cost afegit. - Dificultat per processar objectes transparents. - No suporta tots els algoritmes d'antialiasing.

El principal avantatge és en eficiència, però té algunes complicacions, tot i que la majoria d'aquestes tenen solució.

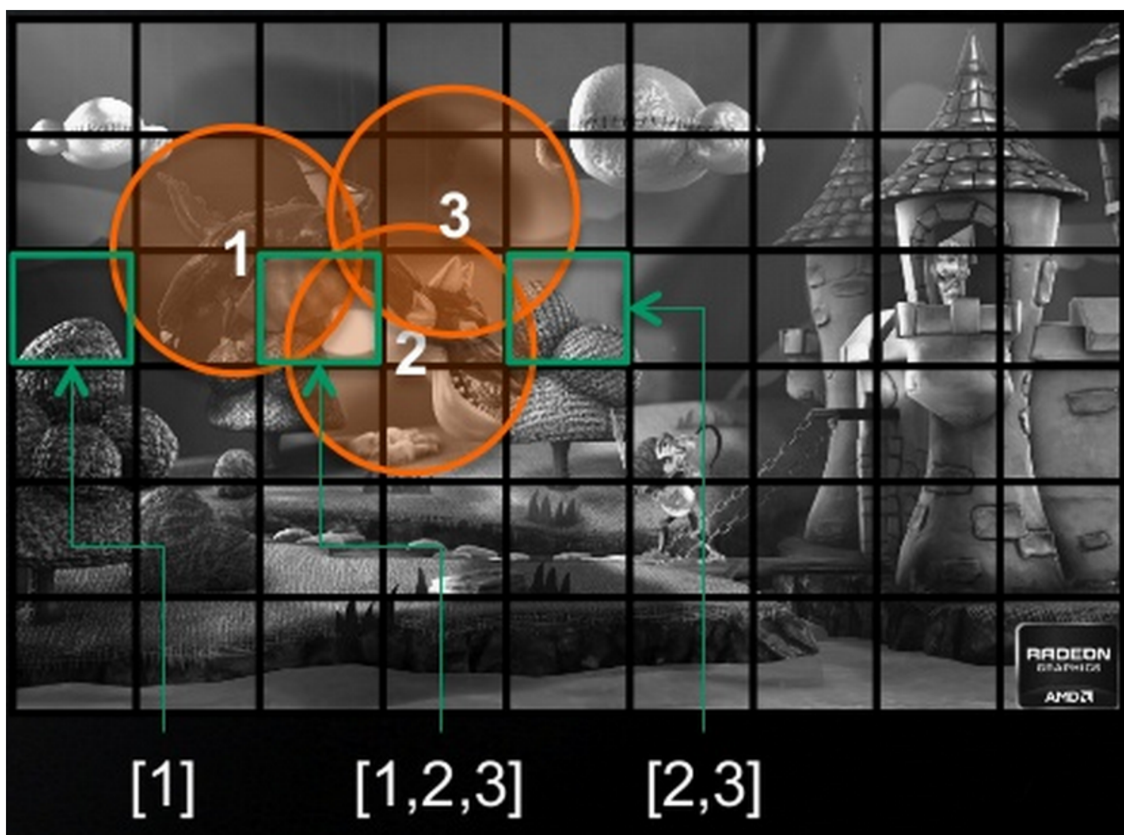
Podem concloure que, tot i que deferred shading té algunes complicacions quan el comparem amb forward rendering, **sembla que aquest mètode serà molt més eficient en escenes complexes amb gran quantitat de llums amb poca intensitat i poca superposició entre elles**. En l'apartat de resultats estudiarem com afecta el rendiment de deferred shading en augmentar el nombre de llums i la seva àrea d'influència.

Forward+

Forward+ (també conegut com a tiled forward rendering) és un algoritme en general menys conegut que deferred shading, aquest fet es deu al fet que encara no s'ha utilitzat gaire en entorns comercials, ja que es tracta d'un algoritme relativament nou que encara està en procés de demostrar tot el seu potencial. La nostra implementació en basa en la presentació de AMD feta a Eurographics 2012.

Com el seu nom indica, **Forward+** intenta seguir un camí semblant a forward rendering en el sentit que només **renderitza l'escena un cop**. La principal diferencia entre els dos, és que Forward+ fa un **preprocés anomenat *light culling***, que permet saber al shader quines llums és possible que afectin el píxel que està dibuixant.

El light culling consisteix bàsicament a dividir el viewport en una graella, llavors calculem la projecció de cada llum sobre el viewport i calculem quines caselles intersequen aquesta projecció **guardant la ID de la llum en cada casella afectada**.



Imatge mostrant una divisió del viewport en una graella, cada cela conte una llista de les llums que l'intersequen.

Quan renderitzem l'escena, a l'hora de calcular el color d'un píxel podem saber en quines coordenades del viewport es troba i per tant en quina casella pertany, el que ens permet **utilitzar la matriu obtinguda en el *light culling* per saber quines llums poden afectar el punt**, deixant de banda les que estan massa lluny o tenen massa poca intensitat.

Aquest preprocés és molt costós, ja que hem de projectar cada llum i calcular quines caselles l'intersequen, per sort **podem configurar la resolució de la graella fins a trobar el punt òptim de rendiment**. A més, **aquest tipus de càlcul és potencialment paral·lelitzable** i en el següent apartat explicarem quines eines hem utilitzat per aconseguir-ho.

A pesar del costós preprocess, aquest mètode té l'avantatge que només cal renderitzar la geometria un cop. Bàsicament al fer un forward render calculant només les llums necessàries, **estem obtenint les avantatges de deferred shading però eliminant el sobrecost de fer dos renders i acumular els resultats**. A més, forward+ no té els problemes que ens trobàvem amb deferred shading, ja que fonamentalment és un forward rendering millorat.

Sembla que **aquest algoritme podria superar deferred shading pel que fa a eficiència**. El principal coll d'ampolla és el light culling i és on caldrà intentar optimitzar l'algoritme per veure si realment podem millorar els resultats obtinguts amb deferred shading. En l'apartat de resultat compararem aquests dos algoritmes i discutirem els resultats.

CUDA

CUDA és un conjunt d'eines de desenvolupament creat per nVidia amb l'objectiu de permetre **utilitzar la potència de càlcul de les GPUs** per altres tasques a part del renderitzat gràfic. Les GPUs tenen la característica que estan dissenyades per maximitzar el throughput, és a dir, per **executar el màxim nombre d'operacions en paral·lel possibles**, a diferència de les CPUs que intenten minimitzar la latència, és a dir, executar poques operacions paral·leles molt ràpid. Hi ha molts processos que poden ser dissenyats per funcionar en paral·lel, de manera que podem millorar la seva eficiència dràsticament.

En el nostre cas, podem utilitzar CUDA per accelerar el procés *del light culling* que fèiem a la tècnica de Foward+. Aquest procés suposava el principal coll d'ampolla de l'algoritme, ja que requeria iterar tota la graella, que depenent de la resolució de les caselles podia ser molt gran, per cada llum. Aquest procediment és potencialment paral·lelitzable, fet que milloraria notablement l'eficiència de Forward+.

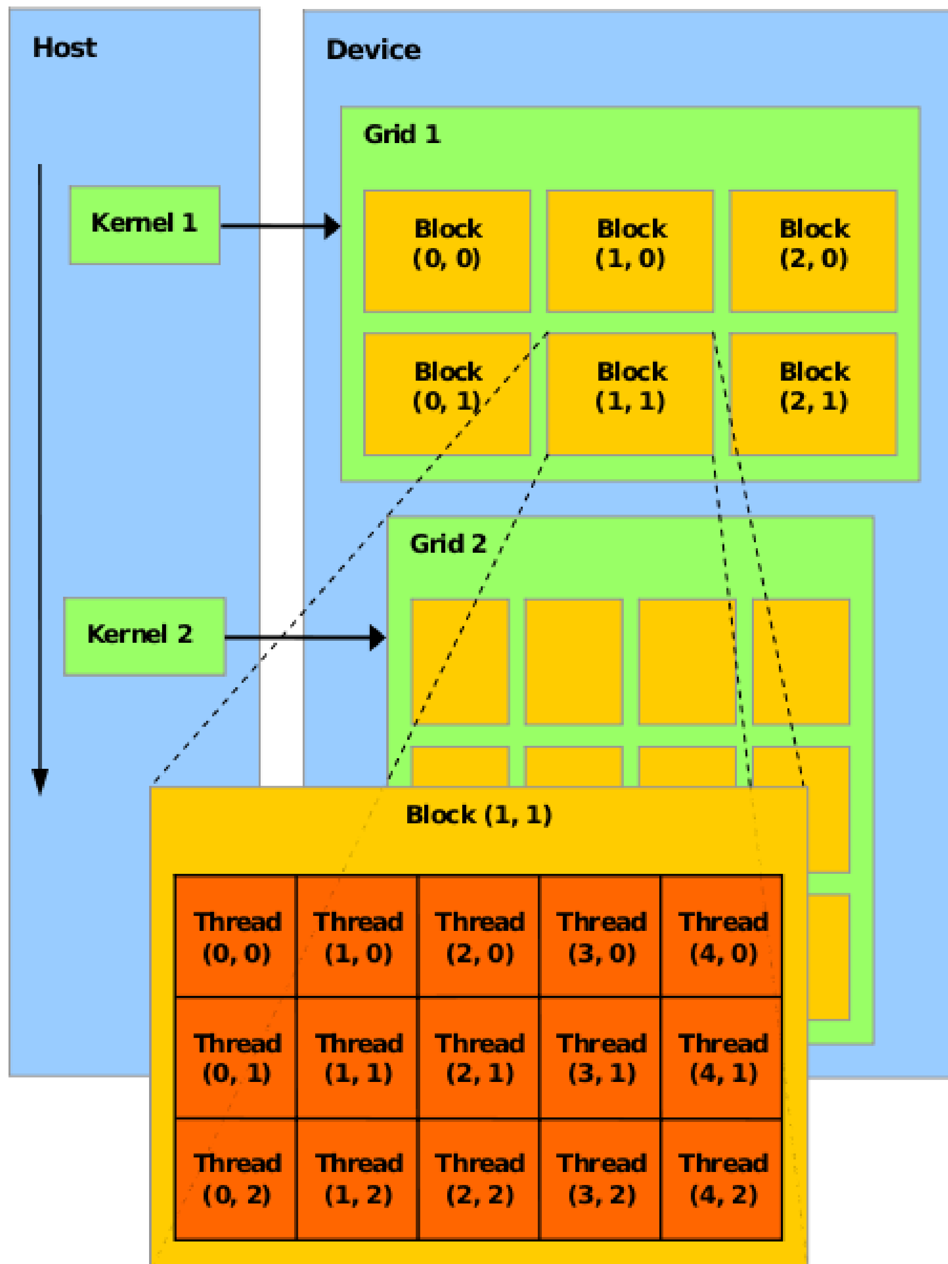
Per entendre més endavant la implementació de Forward+ en versió GPU, explicarem com funciona en general l'arquitectura de CUDA.

Inicialment cal fer la distinció entre *host* i *device*. Anomenem **host a la zona de codi i memòria que s'executa a la CPU** i controla el flux del programa. **La zona de codi i memòria que s'executa en la GPU l'anomenarem device**, és l'encarregat de crear el varis threads i executar-los en paral·lel.

El flux d'un programa CUDA típicament és el següent: **el *host* copia la informació necessària a la memòria del *device*** (més endavant explicarem com funciona la jerarquia de memòria de CUDA), això és degut al fet que el *device* no pot accedir a la memòria del host, per tant cal primer copiar tota la informació abans **d'executar el kernel.** Un kernel bàsicament és una **funció que es crida des del host i s'executa en el *device* amb varis threads en paral·lel**. Finalment el host ha de **copiar el resultat de l'execució del kernel de memòria del device a la de host** per poder-hi accedir.

Quan executem un kernel de CUDA, volem que **s'executi en el màxim paral·lelisme** possible, és a dir, que tinguem el màxim nombre de threads concorrent executant-se a la GPU. **CUDA agrupa els threads en thread blocks**, que bàsicament agrupa una seria de threads que comparteixen un tipus de memòria anomenat *shared memory*. **La GPU s'encarrega d'assignar cada thread block en un dels Streaming**

Multi-processor (SM) que tinguin prou capacitat per executar els seus threads, els SMs vindrien a ser les unitats de procés que utilitza la GPU per executar els blocks, **dintre d'un SM tots els threads del block s'executen en paral·lel. Cal fer una bona repartició de threads per block i nombre de blocks** per poder obtenir el màxim paral·lelisme possible, el host quan crida el kernel és el responsable de dir-li al *device* quants blocks i quants threads per block han d'executar-se.



Gestió de threads en CUDA: el host llença els kernels, cada kernel te una graella de blocks i cada blocks varis threads, tots els thread d'un block s'executen en paral·lel.

Abans hem dit que el host és l'encarregat de copiar la informació necessària a la memòria del device, és important doncs saber com funciona la jerarquia de memòria de CUDA per entendre com podem gestionar-la de manera eficient.

Principalment hi ha 3 tipus de memòria en CUDA: *Global*, *Shared* i *Local*.

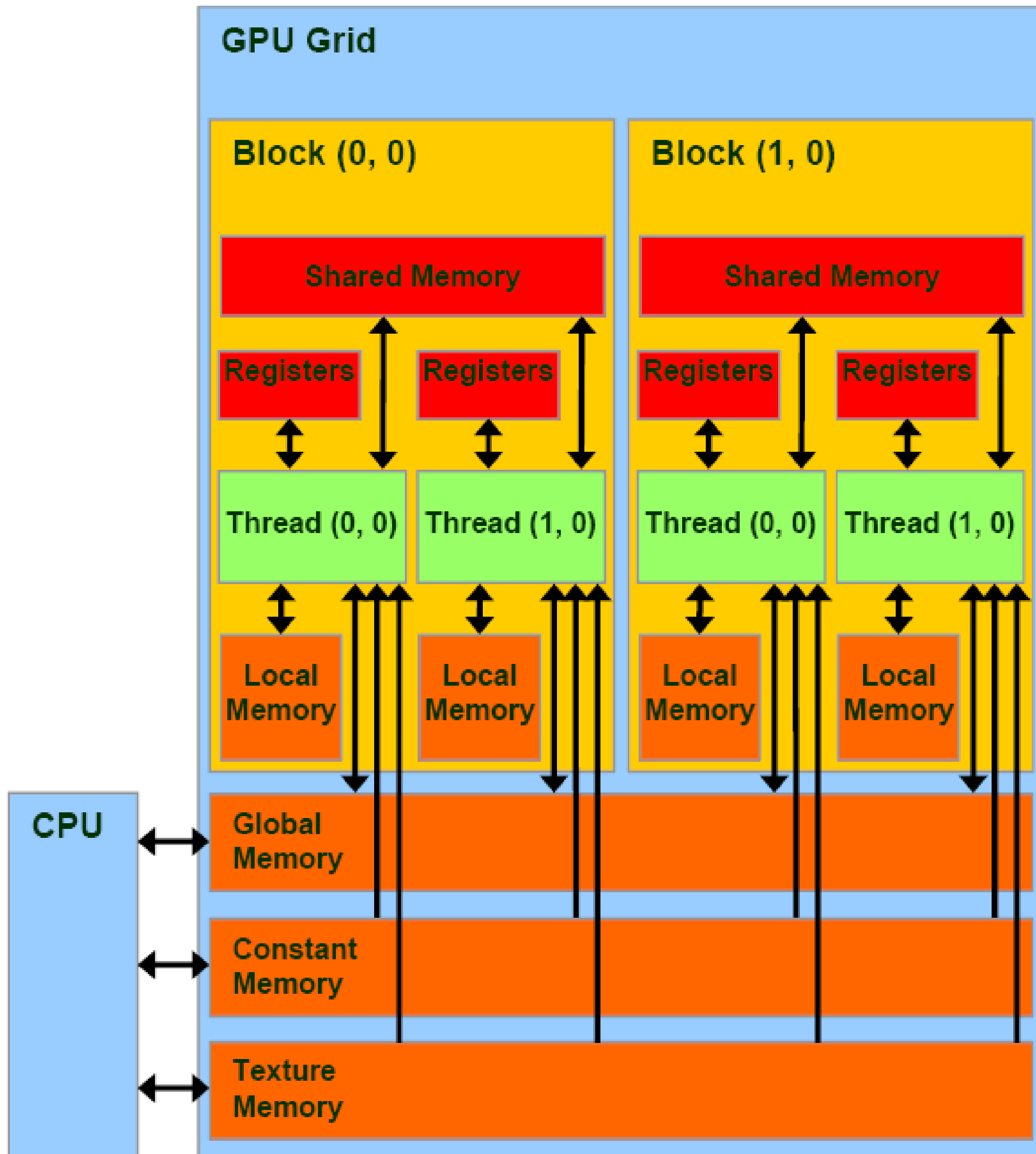
La memòria global és compartida per tots els thread blocks que s'estan executant al kernel, és a dir, **tots els threads poden llegir i escriure** en aquest tipus de memòria. A més **disposem d'una crida de l'API que ens permet assignar més memòria global al kernel i copiar informació de memòria de host a memòria global de *device***. El principal inconvenient d'aquest tipus de memòria és que és el tipus més lent que disposem en CUDA. A més hem de tenir en compte que tots els threads poden accedir o modificar aquesta secció de memòria, el que fa que hàgem de **sincronitzar d'alguna manera els accessos a memòria** per evitar que un thread llegeixi dades incorrectes per culpa de l'ordre que han accedit a la memòria, per exemple un thread podria llegir un resultat just abans que un altre thread l'escribis, fent que el valor que ha llegit el primer thread sigui incorrecte.

Cada thread block disposa d'una memòria compartida o shared pròpia, que pot ser accedida i modificada per tots els thread pertanyents al block. El principal avantatge d'aquesta memòria **és que és vàries ordres de magnitud més ràpida que la memòria global**, fet que la fa molt atractiva si hem de compartir la mateixa informació entre un grup de threads.

Per acabar **cada thread té una memòria local**, que bàsicament consisteix en totes les variables que es declaren dintre del thread. Aquest és el **tipus de memòria més ràpid que hi ha en CUDA.**

En resum, podríem ordenar els tipus de memòria segon la seva velocitat d'accés de la següent manera:

Local (per thread) > Shared (per block) >> Global (tots els threads)

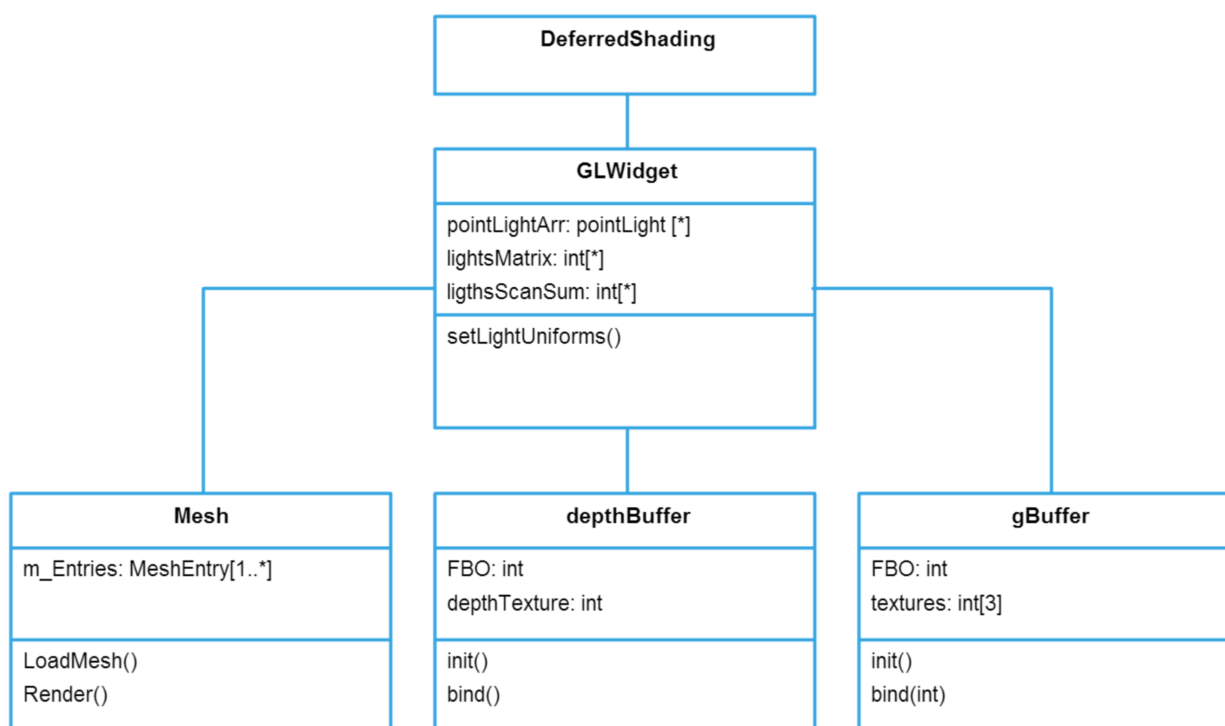


Jerarquia de memòria en CUDA. Per ordre de velocitat d'accés: cada thread té la seva memòria local, cada block té la seva memòria compartida accessible per tots els seus threads i tots els threads tenen accés a la memòria global.

Abans hem dit que l'últim pas d'un programa CUDA era copiar les dades de memòria del device al host. En el nostre cas, un cop tinguem els resultats del *light prepass* a la CPU els enviarem un altre cop a la GPU perquè el shader els pugui utilitzar, el que fa que hàgim de fer moltes còpies innecessàries entre la CPU i la GPU. Per sort **CUDA permet operar i compartir memòria amb OpenGL**, el que fa que evitem la còpia de device a host al final de l'execució del programa, ja que el shader llegirà directament la informació de la memòria global de CUDA.

Estructura bàsica

En aquest apartat explicarem molt breument les parts més rellevant de l'estructura de l'aplicació que hem desenvolupat per implementar els diversos algoritmes de render comentats. Cal notar que no es tracta d'un diagrama complet i que s'han obviat moltes parts que no són rellevants per a les explicacions posteriors. També cal dir que en aquest apartat no farem explicacions detallades dels components, els elements més importants de cada algoritme s'explicaran en els seus respectius apartats.



La classe deferredShading és la que inicia el programa i s'encarrega de creat tots els elements de Qt i gestionar els esdeveniments associats amb aquests. Aquesta classe, tot i que és important per l'execució del programa, no influeix en la implementació dels algoritmes de renderitzat i per tant no entrarem en detall en el seu funcionament.

Un dels components que crea la classe anterior és un **GLWidget**, que **s'encarregarà de crear la finestra on pintarem la nostra escena amb OpenGL**. És, per tant, la classe central de tots els procediments que afecten l'escena i l'encarregada d'executar els diferents algoritmes de render així com totes les crides a OpenGL. Aquesta classe és molt extensa, només explicarem els elements que puguin ser més interessant conèixer.

El primer atribut que hem ficat en aquesta classe és l'array que conté la informació de les llums que figuren a l'escena. Aquesta estructura és important perquè conte tota la informació que utilitzaran els shaders quan hagin de calcular la il·luminació de l'escena. Els dos atributs següents s'explicaran en més detall en l'apartat de Forward+, els hem inclòs aquí com a referència.

Les funcions utilitzades per realitzar els diferents renders les explicarem en els seus apartats corresponents. No obstant això, hi ha una sèrie de funcions amb el nom setXXXUniforms() que **són les encarregades d'enviar als shader informació general**, com per exemple els paràmetres de la llum ambient o el nombre de llums de l'escena. Aquestes funcions varien segons el tipus de render perquè han d'enviar la informació a diferents shaders, a més pot ser que alguns paràmetres variïn lleugerament.

La classe GLWidget usa algunes classes auxiliar. La més notable és **la classe Mesh, que conté informació de la geometria que volem renderitzar**. Bàsicament conte una llista de MeshEntry, aquestes entrades contenen informació sobre els vèrtexs i els materials que utilitza la malla del model. La classe té dues funcions rellevants: l'operació **LoadMesh() que ens permet carregar la geometria des d'un fitxer**; i la funció **Render() que s'encarrega d'enviar la geometria al pipeline d'OpenGL per a ser renderitzada.**

També hi ha una **classe *gBuffer* que gestiona els buffer que utilitzarem en l'apartat de Deferred Shading**. La funció **init() s'encarrega d'inicialitzar els buffers** i la veurem detalladament en l'apartat de Deferred Shading. La funció **bind bàsicament canvia el framebuffer que fem servir per renderitzar l'escena** depenent del paràmetre que li passem:

- GBUFFER_DRAW: fica el framebuffer que conté els gbuffers en mode escriptura.
- GBUFFER_READ: fica el framebuffer que conté els gbuffers en mode lectura.
- GBUFFER_DEFAULT: restaura el framebuffer d'OpenGL.
- GBUFFER_READ_TEX: activa i enllaça les textures corresponents als G-Buffers per poder ser llegides per OpenGL.

La classe **depthBuffer és una classe auxiliar anàloga a la de gBuffer però només conté un depth buffer**.

Forward

En aquest apartat explicarem la implementació de forward rendering que hem usat per fer les comparacions amb la resta de mètodes de render. Com ja hem explicat en l'apartat de conceptes, la limitació de llums d'OpenGL ens suposa un problema, ja que necessitem poder tenir un nombre elevat de llums per fer comparacions vàlides. Per aconseguir això hem provat diversos algoritmes, en aquest apartat explicarem dos d'ells perquè creiem que és interessant veure com es pot resoldre el mateix problema de vàries maneres i poder comparar les seves característiques, especialment l'eficiència.

Blending

Aquesta primera implementació consisteix en **processar packs de llums enviats als shaders mitjançant uniforms, utilitzant el blending d'OpenGL per acumular els resultats**.

Per poder fer el blending correctament **necessitem fer un render inicial que escrigui al depth buffer**, per tant necessitarem una inicialització una mica especial:

```
// Gbuffer gBufferDS
gBufferDS->bind(GBUFFER_DEFAULT);
glDepthMask(GL_TRUE);
glEnable(GL_DEPTH_TEST);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glUseProgram(shaderProgramBlend);
```

glwidget.cpp línies 642-648.

La primera crida s'encarrega d'assegurar-se que estem usant el framebuffer d'OpenGL i no un dels G-Buffers que explicarem més endavant en la secció de Deferred Shading.

Per fer **el primer render necessitem assegurar-nos que s'escrigui al *depth buffer***, això és important perquè per poder fer blending necessitarem aplicar a un offset al valor de profunditat en posteriors renders, ja que sinó els fragments serien descartats automàticament.

Un cop configurat OpenGL podem començar el render, per limitacions de nombre de uniforms de GLSL **utilitzem intervals de 100 llums**. Per enviar les llums adequades al shader hem implementat una funció la següent funció:

```
setLightsUniformsBlend(unsigned int l, unsigned int h, float offset)
```

On *l* i *h* són respectivament el límit inferior i superior de l'interval de llums que volem enviar, la funció s'encarrega de fer les crides necessàries a *glUniform* per enviar les variables necessàries als shaders. Aquesta funció a més accepta un paràmetre extra que és l'offset que volem aplicar al zBuffer. En la primera passada aquest valor serà 0 doncs volem que el depth s'escrigui correctament al depth buffer, més endavant utilitzarem aquest paràmetre per poder acumular la resta de llums.

Exemple de depth buffer, cada píxel té un valor associat a la distància de l'observador.

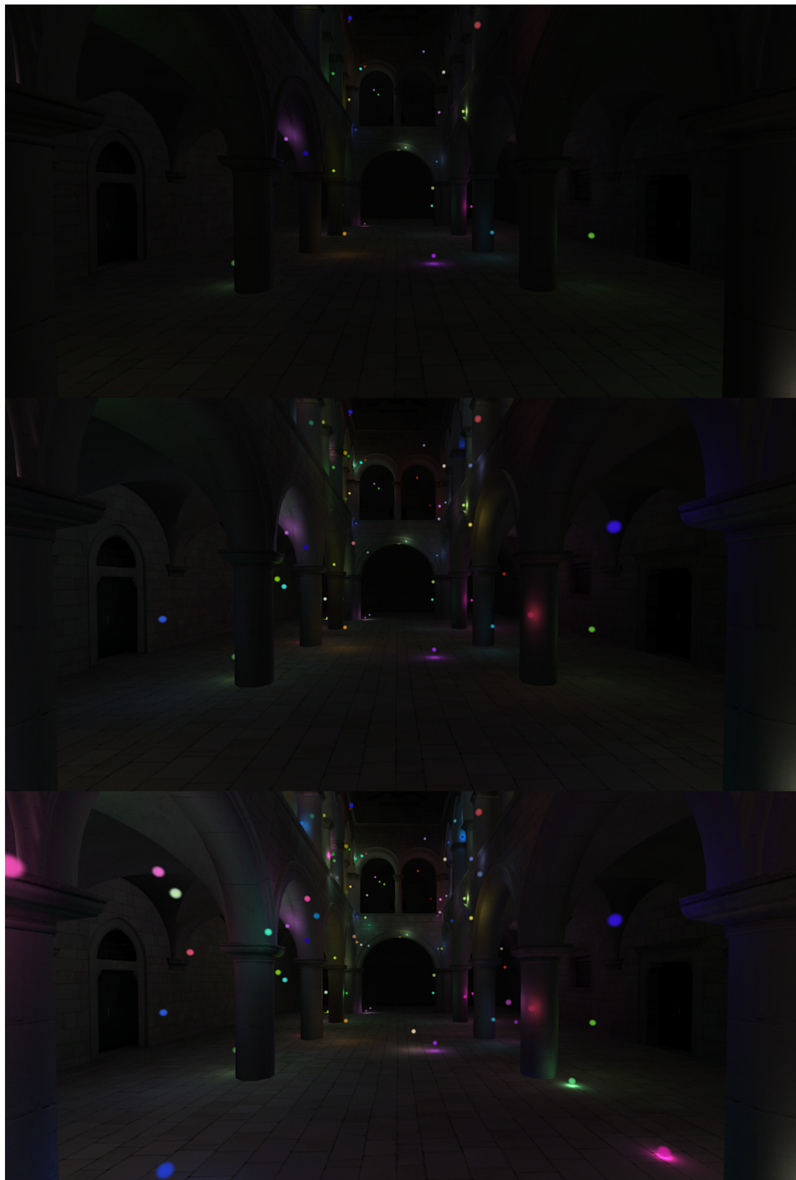
Un cop tenim això hem de començar **acumular les següents llums sobre aquestes**, per tant cal configurar OpenGL de la següent manera:

```
glEnable(GL_BLEND);
glBlendEquation(GL_FUNC_ADD);
glBlendFunc(GL_ONE, GL_ONE);
glDepthMask(GL_FALSE);
```

glwidget.cpp línies 673-676.

Activem el flag de blend de OpenGL i el configurem perquè sumi els colors del framebuffer amb els que surten del pipeline, donant igual importància als dos. A més hem de **desactivar l'escriptura del depth buffer**, perquè ja el tenim omplert amb la primera crida i no volem modificar-lo.

Fet això només cal **iterar sobre el nombre restant d'intervals i anar renderitzant-los**, OpenGL s'encarregarà d'anar acumulant els resultats fins a tenir totes les llums pintades. Cal notar que en aquesta part **cal donar-li un valor negatiu a la crida *setLightsUniformsBlend*** per assegurar-nos que els fragments nous no són descartats en estar a la mateixa profunditat, en el nostre cas hem aplicat un offset de -0.00001.



Imatge en la qual podem aprecia el blending per aconseguir renderitzar 300 llums.

Per acabar, és interessant veure com funciona el fragment shader que utilitzem per ombrejar l'escena, ja que és el focus del problema que ens obliga a fer blending.

Els uniforms que enviem al fragment shader són els següents:

```
uniform ambientLight aLight;
uniform directionalLight dLight;
uniform pointLight pointLights[N_MAX_LIGHTS];
uniform int nLights;
uniform sampler2D sampler;
uniform float zOffset;
```

forward_shader_blend.fs línies 28-33.

El struct ambientLights conté informació de color i intensitat, directionalLight inclou a més una direcció i per acabar pointLight conté informació del color, la intensitat, el radi, la posició i la funció d'atenuació de la llum. La resta de paràmetres són per ordre: el nombre de llums que s'han de pintar, la textura que conté la component difusa i l'offset a aplicar al zBuffer.

El problema principal és l'array *pointLights*, ja que a la que afegim moltes llums s'arriba al límit de uniforms que podem enviar a GLSL.

La resta del shader consisteix a fer un càlcul de l'efecte de cada llum al punt que no copiarem aquí. L'output del shader és el següent:

```
gl_FragColor =  texture2D(sampler, texCoord0.xy)*
(ambientColor+dLightDiffuseColor+pTotalLightDiffuseColor);
gl_FragDepth = gl_FragCoord.z+zOffset;
```

forward_shader_blend.fs línies 65-67.

Amb aquest sistema podem veure que **obtenim resultats correctes**. No obstant això, el **blend és molt ineficient i fa que cada cop que afegim un interval més de llums decaigui l'eficiència de l'algoritme**. En l'apartat de resultats veurem més en detall aquest fet. A causa de aquesta ineficiència, presenten una alternativa a utilitzar blending: utilitzar *texture buffer objects*.

Texture buffer object

El principal canvi que afegim en aquest mode respecte a l'anterior és substituir l'array de uniforms del fragment shader, anomenat *pointLights*, per una sola uniform de tipus *samplerBuffer*. Aquest canvi comporta diversos efectes en el procés de render com veurem tot seguit.

La part d'OpenGL se simplifica considerablement, ja que no cal pintar les llums per intervals i fer blending. **Inicialitzem el procés** de la mateixa manera que fèiem en mètode anterior: ens **assegurem que el framebuffer que usem és el de OpenGL i que l'escriptura al depth buffer està activada**, al fer només una passada de render no necessitem desactivar-la.

Abans de fer el render cal actualitzar l'estat de les llums al *texture buffer*, anomenem al *texture buffer* al buffer de opengl anomenat GL_TEXTURE_BUFFER, i al tipus de textura del mateix nom. Amb les següents crides omplim el buffer i la textura amb la informació necessària:

```
glBindBuffer(GL_TEXTURE_BUFFER,TB);
glBufferData(GL_TEXTURE_BUFFER,nLights*sizeof(pointLight),
            &pointLightsArr[0],GL_DYNAMIC_COPY);

glBindTexture(GL_TEXTURE_BUFFER,LTB);
glTexBuffer(GL_TEXTURE_BUFFER,GL_RGB32F,TB);
```

glwidget.cpp línies 413-417.

Aquestes crides el que fan és actualitzar el *texture buffer* amb el contingut que tenim a l'array *pointLightsArr*, que es correspon a la informació de les llums de l'escena. Posteriorment s'indica que la textura corresponent al GL_TEXTURE_BUFFER utilitzi la informació que acabem de copiar al buffer i especifica quin format de textura volem usar, com que molts dels nostres paràmetres consisteixen en grups de 3 floats (posició, color…) hem escollit GL_RGB32F.

Un cop fet això només **cal assignar aquesta textura a una unitat de textura de OpenGL i passar-li al shader el seu ID**, aquí hem escollit fer servir GL_TEXTURE1.

```
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_BUFFER, LTB);
glUniform1i(lightsTexBufferLocation, 1);
```

glwidget.cpp línies 633-635.

Un cop fet això ja podem **enviar la geometria a renderitzar**. Cal notar que el shader que fem servir ha canviat lleugerament, ja que hem modificat la manera com enviem les llums. Els paràmetres d'entrada d'aquest fragment shader són els següents:

```
uniform ambientLight aLight;
uniform directionalLight dLight;
uniform int nLights;
uniform sampler2D sampler;
uniform samplerBuffer lightsTexBuffer;
```

forward_shader.fs línies 18-22

Com podem veure l'únic canvi que em fet és **substituir l'array de tipus pointLight que teníem per un sol uniform de tipus samplerBuffer** anomenat *lightsTexBuffer*. Aquest és el buffer a OpenGL em omplert amb l'informació de les llum. **Per poder accedir al valors fem servir la funció *texelFetch*.**

```
vec3 lColor = texelFetch(lightsTexBuffer,i*4).rgb;
```

forward_shader.fs línia 45

Aquest és un exemple de com obtenir el color la llum i-ésima, cada llum té 4 paràmetres i cadascun d'aquestes té 3 elements de tipus float, tal com hem especificat a OpenGL anteriorment en fer el linkatge del buffer amb la textura.

Un cop extraiem tots els paràmetres que necessitem de la llum, la resta del shader és el mateix que en el mètode anterior.

Aquest mètode **visualment dona uns resultats idèntics als del mètode anterior però ens estalviem haver de fer blending**, fet que accelera notablement el procés de renderitzat tal com veurem en la secció de resultats.

Deferred Shading

Com ja hem comentat en l'apartat de conceptes, **deferred shading consisteix a separar el pintat de la geometria i la il·luminació**. Per poder obtenir la informació de la geometria necessitarem una sèrie de buffers, que anomenarem G-buffers, en els quals desarem aquesta informació que posteriorment usarem per fer la il·luminació de l'escena.

En iniciar l'aplicació cal **crear i preparar els G-buffers**, per facilitar això hem creat una classe gbuffer que ho gestiona.

```
glGenFramebuffers(1, &FBO);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, FBO);
```

gbuffer.cpp línies 14-15

Per començar cal que **generem un nou framebuffer** i que li indiquem a OpenGL que l'utilitzi com a draw buffer, això vol dir que aquests framebuffer s'utilitzarà com a objectiu quan fem el render de l'escena.

```
glGenTextures(GBUFFER_N_TEXTURES, textures);

for (int i = 0; i < GBUFFER_N_TEXTURES; ++i){
	glBindTexture(GL_TEXTURE_2D, textures[i]);
	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB32F, w, h, 0, GL_RGB, GL_FLOAT, 
NULL);
	[...]
	glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0+i, 
GL_TEXTURE_2D, textures[i], 0);
}
```

gbuffer.cpp línies 18-28

Un cop inicialitzat el framebuffer hem de **generar les textures que utilitzarem com a GBuffers**. Els framesbuffers tenen una sèrie de "*attachment points*" que ens permeten indicar-li al framebuffer quines textures han de fer servir quan reben informació d'OpenGL. A nosaltres ens interessen els attachments de tipus GL_COLOR_ATTACHMENT*i*, ja que el que volem és guardar el resultat del render. Altres *attachement point* ens permetrien, per exemple, guardar la profunditat de l'escena. Sabent això, cal **configurar les textures d'acord amb els requeriments del framebuffer i assignar-les al GL_COLOR_ATTACHMENTi corresponent**.

Per últim cal **indicar-li a OpenGL quins colors buffers pot escriure** en el framebuffer:

```
GLenum drawBuffers[] = {GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1,
                        GL_COLOR_ATTACHMENT2};
glDrawBuffers(GBUFFER_N_TEXTURES,drawBuffers);
```

gbuffer.cpp línies 37-38

Amb això ja **tenim els Gbuffers llestos per poder fer el *geometry pass***, la configuració d'aquesta passada de render és bastant senzilla:

```
glUseProgram(shaderProgramDeferredGeo);
gBufferDS->bind(GBUFFER_DRAW);
glDepthMask(GL_TRUE);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);
glDisable(GL_BLEND)
```

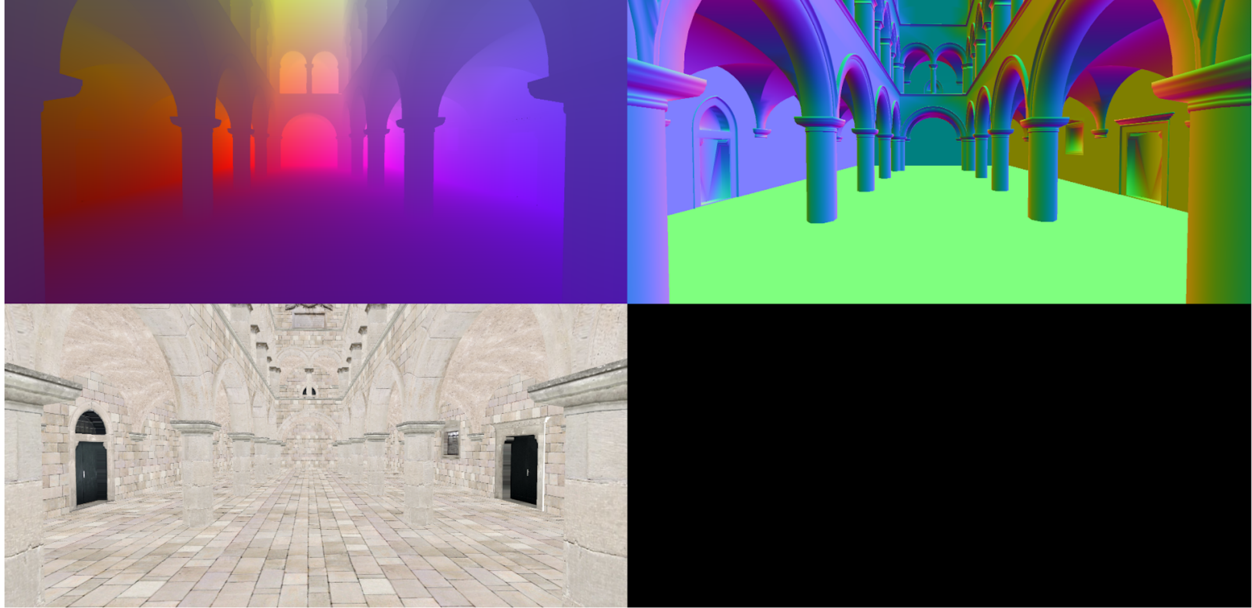
glwidget.cpp línies 711-716

El que fem és indicar-li a OpenGL que utilitzi el framebuffer que hem configurat amb els GBuffers i tota la resta ho configurem com si fos un render normal, els shaders s'encarregaran d'omplir els GBuffers correctament. Arribat a aquest punt ja podem **enviar la geometria al pipeline d'OpenGL**.

```
gl_FragData[0] = vec4(position0,1.0);
gl_FragData[1] = vec4(normalize(norm0),1.0);
gl_FragData[2] = vec4(texture2D(sampler, texCoord0.xy).xyz,1.0);
```

deferred_shader_geo.fs línies 12-14

El fragment shader associat amb aquesta passada l'únic que ha de fer és **copiar la informació necessària al target** que toca, això es coneix com a *Multiple Render Targets*. Cada output es correspon amb un GL_COLOR_ATTACHMENT*i* al framebuffer. La informació que extreu el fragment shader és, per ordre: posició, normal i component difusa.



Exemple dels gBuffers utilitzats, a dalt a l'esquerra es veu la posició, a dalt a la dreta les normals i abaix a l'esquerra la component difusa.

Arribat a aquest punt ja **tenim el Gbuffers omplerts amb la informació necessària**. El següent pas consisteix a aplicar la il·luminació a partir dels G-buffers, això es coneix com a *lighting pass*. Per fer l'acumulació de les llums **dibuixarem l'esfera envoltant de cada llum i farem blending del resultat** de calcular il·luminació de cada una.

```
glDepthMask(GL_FALSE);
glDisable(GL_DEPTH_TEST);

glUseProgram(shaderProgramDeferredLight);
glEnable(GL_BLEND);
glBlendEquation(GL_FUNC_ADD);
glBlendFunc(GL_ONE, GL_ONE);

gBufferDS->bind(GBUFFER_READ_TEX);
glClear(GL_COLOR_BUFFER_BIT);

setLightPassUniforms();

for (unsigned int i = 0; i < nLights; i++)
drawPointLight(pointLightsArr[i]);
```

glwidget.cpp línies 721-736

Inicialitzem la segona passada desactivant el test de profunditat, ja que volem pintar totes les llums, **i activem blending**. També li indiquem a la classe encarregada de gestionar els G-buffer que prepari les **textures que contenen la informació dels G-buffers per a ser llegides i configuri el framebuffer que estem utilitzant per al de OpenGL**. La funció *setLightPassUniforms()* s'encarrega **d'enviar-li al shader quines textures ha de fer servir**. Finalment **pintem cada llum com una esfera**, el shader s'encarregarà de pintar correctament l'efecte de la llum amb la informació del G-buffers.

```
uniform sampler2D positionBuffer;
uniform sampler2D normalBuffer;
uniform sampler2D diffuseBuffer;
uniform vec2 screenSize;
uniform pointLight pLight;
```

deferred_shader_light.fs linies 11-15

Al shader li enviem els G-Buffers així com les dimensions del viewport i la informació de la llum que estem processant.

```
float screenLocX = gl_FragCoord.x/screenSize.x;
float screenLocY = gl_FragCoord.y/screenSize.y;
vec2 screenLoc = vec2(screenLocX,screenLocY);

vec3 position = texture2D(positionBuffer,screenLoc).xyz;
```

deferred_shader_light.fs linies 20-23

Per accedir a la informació del punt que estem processant hem de **calcular en quines coordenades del viewport ens trobem i accedir a la posició corresponent dels Gbuffers**. La resta del shader s'encarrega de fer els càlculs de llum.

Exemple de imatge pintada usant deferred shading.

Com podem veure a la imatge, **el resultat final és el mateix que en el cas de forward rendering**. Cal notar però que en aquesta implementació de deferred shading no hem tingut en compte problemes com el de les transparències, per tant, si hi hagués algun objecte transparent, no es veuria correctament, aquest problema té solució però no entrava dintre l'abast del projecte fer-ne la implementació.

Hem vist com amb aquest mètode **només executem el fragment shader en els punts dins de l'esfera delimitant de la llum**. Això fa que s'hagin de processar molts menys fragments en escenes amb un gran nombre de llums de poca intensitat. A més com que als G-Buffers només guardem els punts mes propers, **no hem de calcular la il·luminació dels punts amagats per altres parts de la geometria**, fet que també contribueix a millorar l'eficiència.

Forward+

La idea principal de Forward+ consisteix a dividir el viewport en una graella i calcular quines llums afecten cada casella. Aquest procés es coneix com a *light prepass* i és la part més important d'aquest algoritme. Un cop tenim aquesta informació, podem pintar l'escena amb una sola passada de render utilitzant els resultats del *light prepass*.

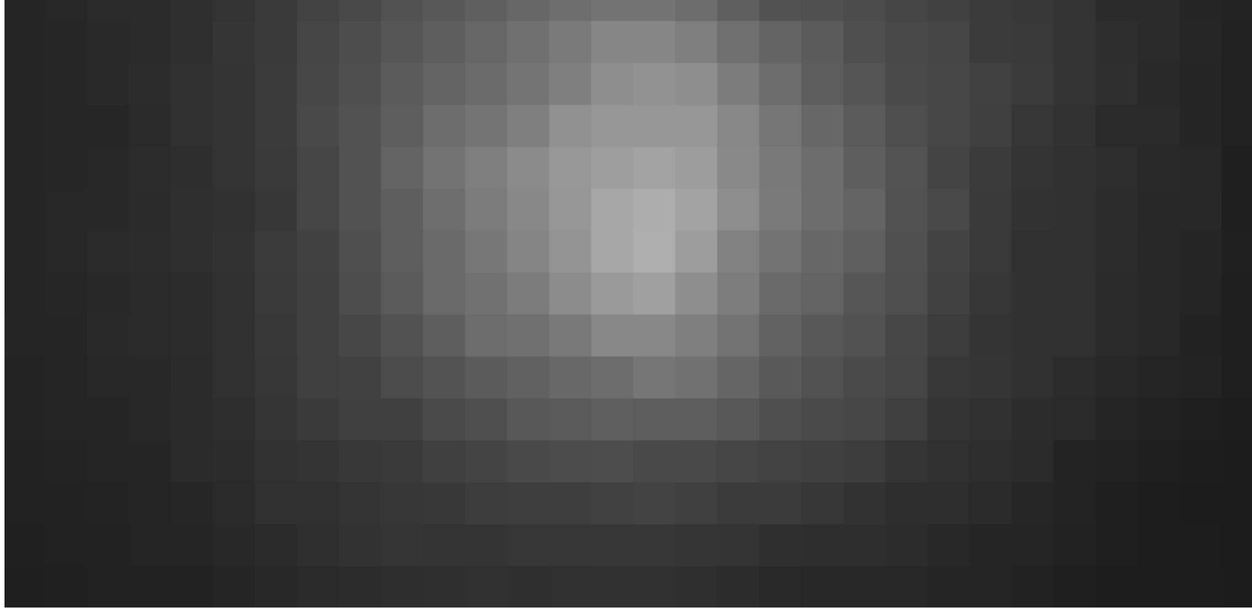
Per poder fer aquest render el shader necessitarà, a part dels paràmetres que enviàvem per fer un forward render, la llista de IDs que hi ha a cada casella. Per poder fer això utilitzarem **dos arrays: un en el que guardarem els IDs de les llums**, que anomenarem light grid, **i un altre que es correspondrà a un scan sum del nombre de llums de cada casella**.

Per omplir la graella de llums necessitem fer uns quants passos. **Primer de tot s'ha de calcular la projecció de cada llum en el viewport**, el que volem és trobar la posició del centre de l'esfera projectat en el pla del viewport i el seu radi projectat.

Un cop tenim la llum projectada, hem de **calcular a quines caselles de la graella afecta**, per fer això cal recórrer tota la graella i comprovar si la distància de la casella al centre de la projecció és més petita que el radi. En cas afirmatiu escrivim l'ID de la llum a la posició corresponent de la *light grid*.

Aquest procediment es fa per totes les llums i al final tenim la graella omplerta amb la informació necessària. Ara només falta **fer la scan sum del nombre d'element de les caselles de graella** per poder compactar aquesta array i enviar-li al shader el mínim nombre d'elements possibles.

Amb la scan sum feta podem **compactar la matriu de llums** de manera que no hi hagi cap espai en blanc en l'array de IDs que enviarem al shader.



Representació de la light grid, cada casella es pinta de color més clar com més llums afecten aquella posició..

Finalment podem **enviar la informació al shader**. Com que aquests array contindran un elevat nombre d'elements, **utilitzarem textures buffers** per enviar les dades.

```
GLuint TB[2];
glGenBuffers(2,TB);

glBindBuffer(GL_TEXTURE_BUFFER,TB[0]);
glBufferData(GL_TEXTURE_BUFFER,sizeof(_lightsMatrix),&_lightsMatrix[0],GL
_DYNAMIC_COPY);

glBindTexture(GL_TEXTURE_BUFFER,LGTB);
glTexBuffer(GL_TEXTURE_BUFFER,GL_R32I,TB[0]);

glBindBuffer(GL_TEXTURE_BUFFER,TB[1]);
glBufferData(GL_TEXTURE_BUFFER,sizeof(_lightsScanSum),&_lightsScanSum[0],
GL_DYNAMIC_COPY);

glBindTexture(GL_TEXTURE_BUFFER,SSTB);
glTexBuffer(GL_TEXTURE_BUFFER,GL_R32I,TB[1]);

glDeleteBuffers(2,TB);
```

glwidget.cpp linies 1219-1234

Fet això ja podem **enviar la geometria al pipeline d'OpenGL**. Les principals diferències en els shader són les següents.

```
uniform isamplerBuffer lightsGrid;
uniform isamplerBuffer scanSum;
```

forward_plus_shader.fs linies 25-26

Tenim dos uniforms de tipus sampler buffer més que en forward render. Per defecte, GLSL implementa els seus tipus com a floats, per a indicar-li que estem utilitzant enters cal afegir una *i* davant del tipus.

```
float nCol = ceil(screenSize.x/TILE_SIZE);
float xTile = floor(gl_FragCoord.x/TILE_SIZE);
float yTile = floor(gl_FragCoord.y/TILE_SIZE);
int tile = int(xTile+nCol*yTile);
```

forward_plus_shader.fs linies 30-33

El primer que cal fer és **calcular en quina coordenada del viewport estem pintant i determinar quina casella li correspon**.

```
int lightsBegin, lightsEnd;
if (tile == 0) lightsBegin = 0;
else lightsBegin = texelFetch(scanSum,tile-1);
lightsEnd = texelFetch(scanSum,tile);

for (int i = lightsBegin; i < lightsEnd; ++i) {
    int lightId = texelFetch(lightsGrid,i);
    // Process light
```

forward_plus_shader.fs linies 35-43

Un cop sabem en quina casella ens trobem **podem obtenir de la *lights grid* el rang de llums que afecta aquella casella gràcies a la *scan sum***. Llavors obtenim els IDs de les llums que cal processar i fem els càlculs d'il·luminació corresponents amb aquestes llums.

Exemple de imatge pintada usant Forward+.

Aquest procés ens permet fer un ***forward render* limitant el nombre de llums aplicades a cada fragment, fet que millora molt la velocitat d'aquest procés**. Faltarà veure si el *light culling* és prou eficient per superar els altres algoritmes de pintat que hem vist. Un dels avantatges que tenim amb forward+ és que **podem configurar la mida de la graella** per trobar el punt òptim entre el cost del light culling i la millora en temps de render. Això es veurà en més detall en l'apartat de resultats.

Com hem pogut veure, el procés del *light culling* ha d'iterar múltiples cops la graella de llums. Aquest procés es pot paral·lelitzar utilitzant CUDA, fet que podria millorar el rendiment d'aquesta tècnica.

CUDA

Un dels avantatges que té l'ús de CUDA és la seva **interoperabilitat amb OpenGL**. Això vol dir que podem compartir zones de memòria en la GPU entre CUDA i OpenGL, fet que ens estalvia fer transferències de memòria entre la CPU i la GPU que tenen un cost elevat.

```
GLuint CUDABuffers[2];
cudaGraphicsResource *resource[2];
```

glwidget.h linies 210-211

Per poder fer servir la interoperabilitat entre CUDA i OpenGL **necessitem dues variables**: la primera serà **la variable que indicarà a OpenGL l'ID del buffer**. La segona és un tipus propi de CUDA, per simplificar direm que es tracta de l'**ID que CUDA dona al buffer que hem creat** en OpenGL. Per tant **els dos IDs apuntaran al mateix buffer, la diferència és que un s'usarà en OpenGL i l'altre en CUDA.**

```
      glBindBuffer(GL_TEXTURE_BUFFER, CUDABuffers[1]);
      glBufferData(GL_TEXTURE_BUFFER, sizeof(_lightsScanSum), NULL,
GL_DYNAMIC_COPY);
      glBindTexture(GL_TEXTURE_BUFFER,SSTB);
      glTexBuffer(GL_TEXTURE_BUFFER,GL_R32I,CUDABuffers[1]);
```

glwidget.cpp linies 444-447

Primer de tot cal **inicialitzar els buffers amb OpenGL**. Aquest buffer ja el podríem utilitzar en OpenGL, però si volem poder utilitzar-lo en CUDA necessitarem un pas més.

```
cudaGraphicsGLRegisterBuffer(&resource[1], CUDABuffers[1],
cudaGraphicsRegisterFlagsWriteDiscard);
```

glwidget.cpp linia 486

Aquesta crida **associa un buffer d'OpenGL amb un *resource* de CUDA**. Un cop fet això, CUDA ja té registrat el buffer d'OpenGL i pot accedir a la seva adreça de memòria.

```
cudaGraphicsMapResources(2, resource, NULL);
launch_kernel(pointLightsArr, [...], resource[1]); // CUDA
cudaGraphicsUnmapResources(2, resource, NULL);
```

glwidget.cpp linies 1256-1261

Tot i que tant OpenGL com CUDA tenen registrats els buffers, no es pot accedir a aquests buffer des del dos alhora. Per poder utilitzar els buffers en CUDA cal cridar la funció *cudaGraphicsMapResources()*. Aquesta funció fa un **mapejat del resource perquè CUDA hi pugui accedir. Un cop acabat l'ús del buffer per part de CUDA cal fer la crida corresponent perquè OpenGL pugui utilitzar el buffer.**

```
cudaGraphicsResourceGetMappedPointer((void**) &d_lightsScanSum, &size,
lightsScanSumResource);
```

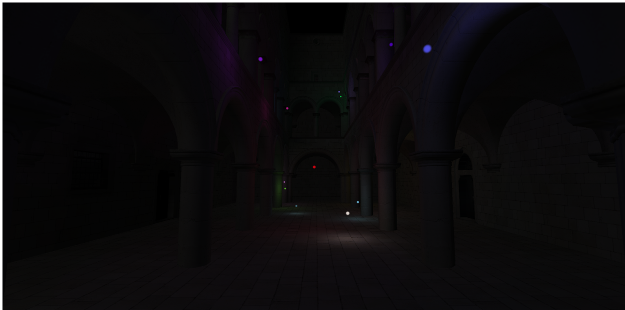
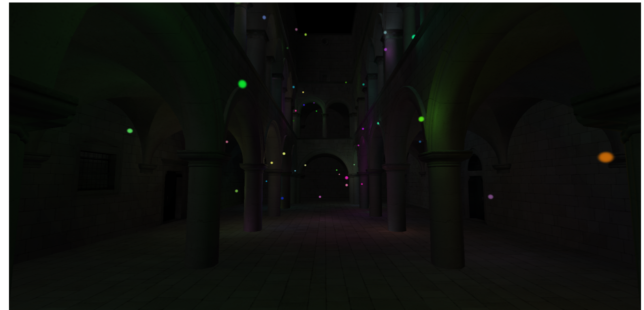
Per acabar, un resource no és un punter a memòria sinó una estructura pròpia de CUDA utilitzada per identificar un recurs. **Per obtenir el punter a memòria de device necessitem fer una crida de l'API de CUDA** que donat un cudaGraphicsResource ens retorna el punter al recurs en memòria de device i la seva mida en bytes. Aquest punter és el que utilitzarem en els kernels per omplir els buffers. Aquests mateixos buffers són els que més endavant utilitzaran els shaders per fer el renderitzat.

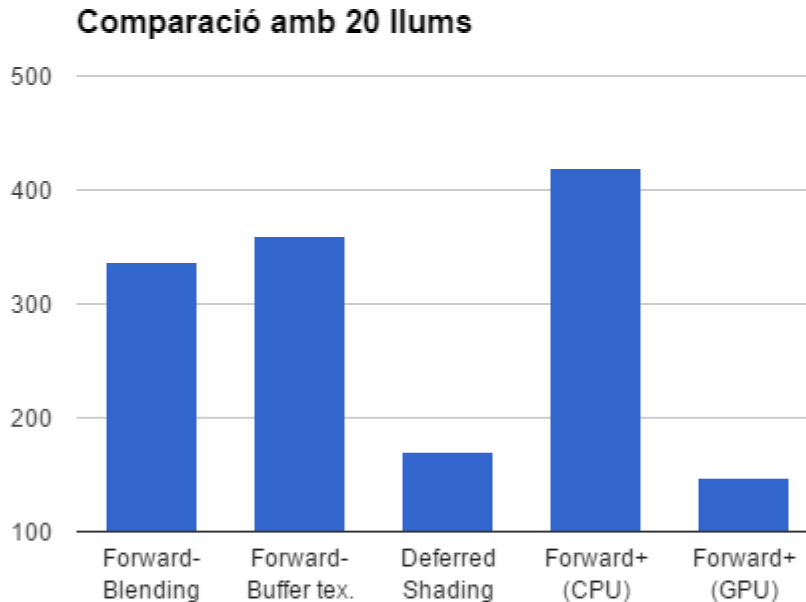
Arribat a aquest punt ja podem executar l’algoritme en CUDA, la implementació del light prepass paral·lelitzat no l'explicarem en detall, ja que segueix la mateixa estructura que la versió en sèrie però fent canvis específics perquè el resultat es pugui executar en varis threads simultanis.

Resultats

Comparació de mètodes

En aquest apartat analitzarem el rendiment obtingut dels diferents mètodes que hem implementat. L'Objectiu principal és veure com varia l'eficiència dels diferents algoritmes a mesura que incrementem el nombre de llums de l'escena. Les proves que realitzarem consistiran a configurar l'escena amb un nombre incremental de llums, tal com es veu a les imatges mostrades i compara els resultats obtinguts entre els diferents mètodes.

<table>
<tr><td>
20 llums</td><td>
100 llums</td></tr>
<tr><td>
300 llums</td><td>
1000 llums</td></tr>
</table>

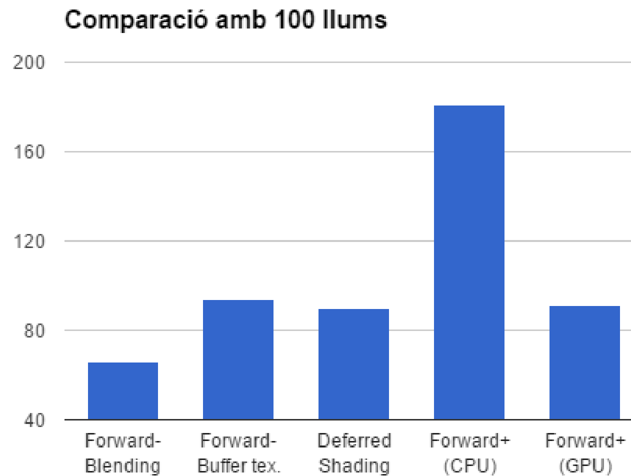


Aquest és el joc de llums més petit que utilitzarem, aquesta prova ens servirà per veure quins algoritmes són millor en situacions de poc estrès on el nombre de llums és reduït.

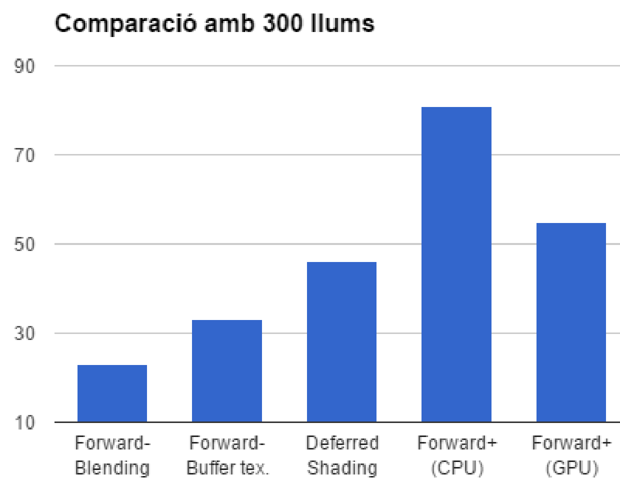
Podem veure com les implementacions de forward rendering i forward+ són molt més eficients que deferred shading, a més sembla que la versió en GPU de forward+ tampoc dona bons resultats.

En el cas de deferred shading, això és degut al fet que, com que hi ha un nombre molt petit de llums, el sobrecost que afegeix deferred shading en renderitzar l'escena dos cops i fer blending no surt a compte, ja que forward render encara pot suportar aquesta càrrega de llums i en fer només un render aconsegueix millors resultats.

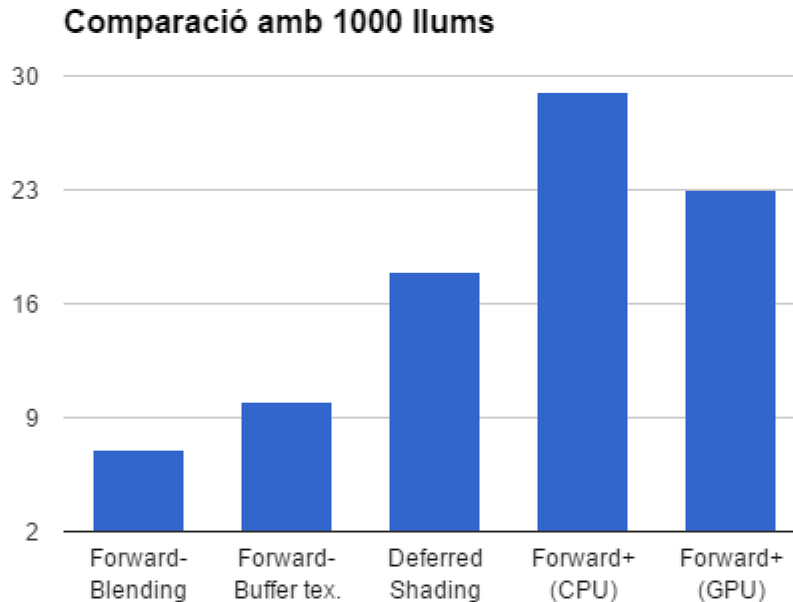
Amb forward+ CUDA segurament passa una cosa semblant, el cost d'executar en paral·lel un nombre petit de threads és molt alt, ja que, a part del cost de crear i gestionar els threads, les instruccions a GPU en general tarden molt més a executar-se.



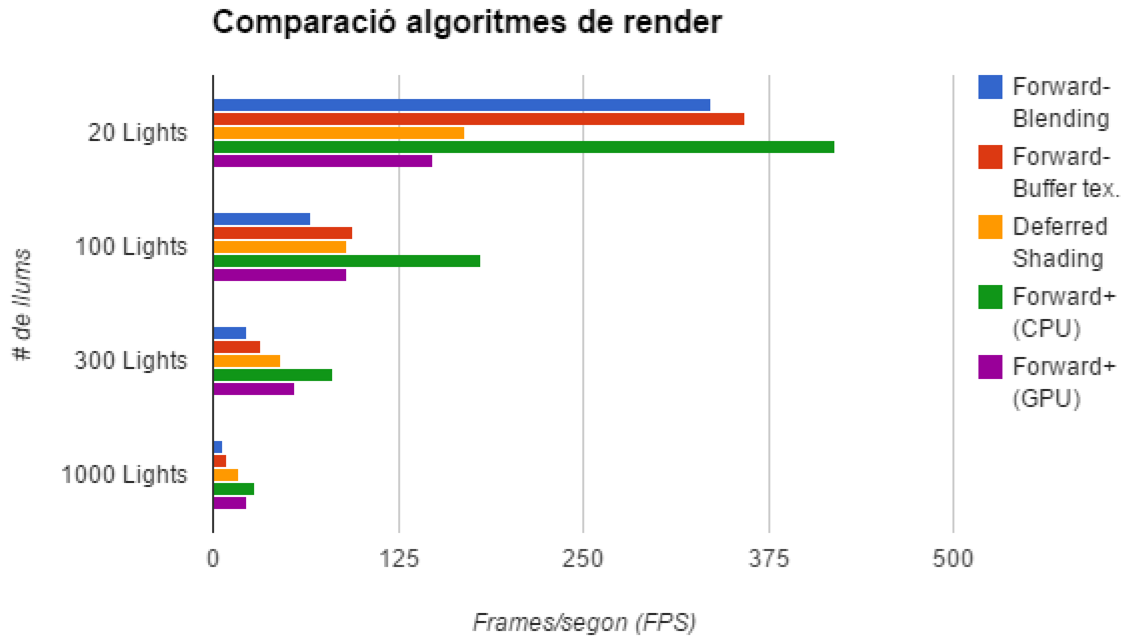
Amb 100 llums forward rendering començar a tenir problemes i observem una dràstica caiguda del seu framerate, especialment amb la tècnica de blending. A causa d'aquesta davallada podem veure com deferred shading ja iguala a la implementació de forward rendering que utilitza buffer textures. Podem veure també que Forward+ es desmarca clarament de la resta, i que la versió de GPU aconsegueix mantenir-se a la mateixa altura que deferred shading, fent una considerable millor respecta el cas anterior.



Podem veure que a mesura que incrementen el nombre de llums les implementacions de forward rendering tradicionals es queda cada cop més enrere, particularment la de blendig, ja que cada cop ha de renderitzar més paquets de llums. Deferred shading es manté clarament per sobre d'aquests dos mètodes, però la implementació de Forward+ en GPU ja l'ha aconseguit superar. Mentrestant, Forward+ segueix sent l'algoritme que dóna millors resultats.



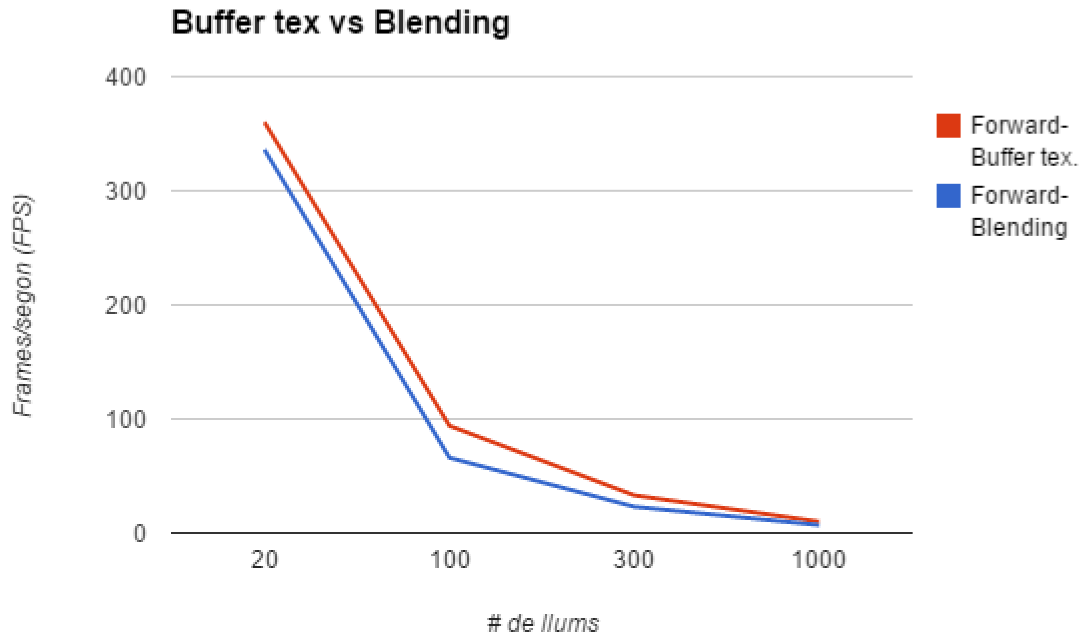
Per acabar, observem que amb 1000 llums dinàmiques la implementació de Forward+ en CPU és la que ofereix millors resultats, seguit de la versió en GPU. Això és contradictori amb la idea que teníem inicialment d'optimitzar el procés utilitzant CUDA. No obstant això, podem veure que lentament la versió de GPU s'acosta al rendiment de la versió CPU a mesura que augmentem el nombre de llums. La principal causa d'això és que en fer créixer la quantitat d'informació a processar obtenim més beneficis del paral·lelisme de la GPU. Això ens fa sospitar que la implementació que hem fet amb CUDA no acaba d'estar ben paral·lelitzada i que una possible millor paral·lelització en els algoritmes que usem podria fer que aquesta versió millores notablement. A part d'això, cal notar que la mida de les caselles de la graella també afecta al rendiment d'aquests algoritmes, ho veurem en detall més endavant.



En resum podem veure com Forward+ és indiscutiblement l'algoritme que ha donat millors resultats al llarg de tots els jocs de proves. Per contra, forward rendering ha estat el que més ha sofert en augmentar el nombre de llums. Deferred shading, tot i no mostrar un rendiment massa bo en els casos més petits, ha escalat molt bé en augmentar el nombre de llums a l'escena.

Forward Render

En aquesta secció compararem les dues tècniques de forward rendering que hem implementat. Hem resumit la informació dels dos mètodes en la següent gràfica.

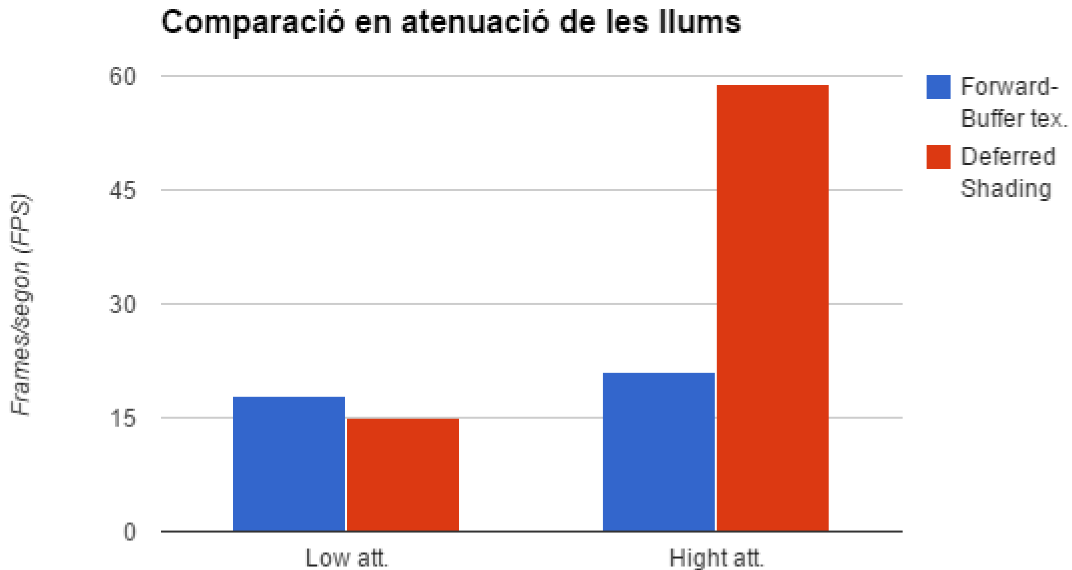


Com ja pensàvem, la implementació amb buffer textures és millor en tots els casos respecte a la de blending, degut precisament al fet que el blending és una operació molt costosa.

Deferred shading

El principal avantatge que té deferred shading sobre forward rendering és que només fa el càlcul la il·luminació en els píxels on pot afectar la llum. Intuïtivament sembla que en situacions on les llums ocupin poc espai en pantalla deferred shading serà molt més eficient que forward rendering. Per veure com afecta la mida de la llum hem implementat un test amb dues escenes: una on les llums tenen molt poca atenuació i per tant ocupen pràcticament tot el viewpot, i un altre on les llums tenen una atenuació molt alta i ocupen poc espai al viewport.

 Atenuació baixa	 *Atenuació alta*

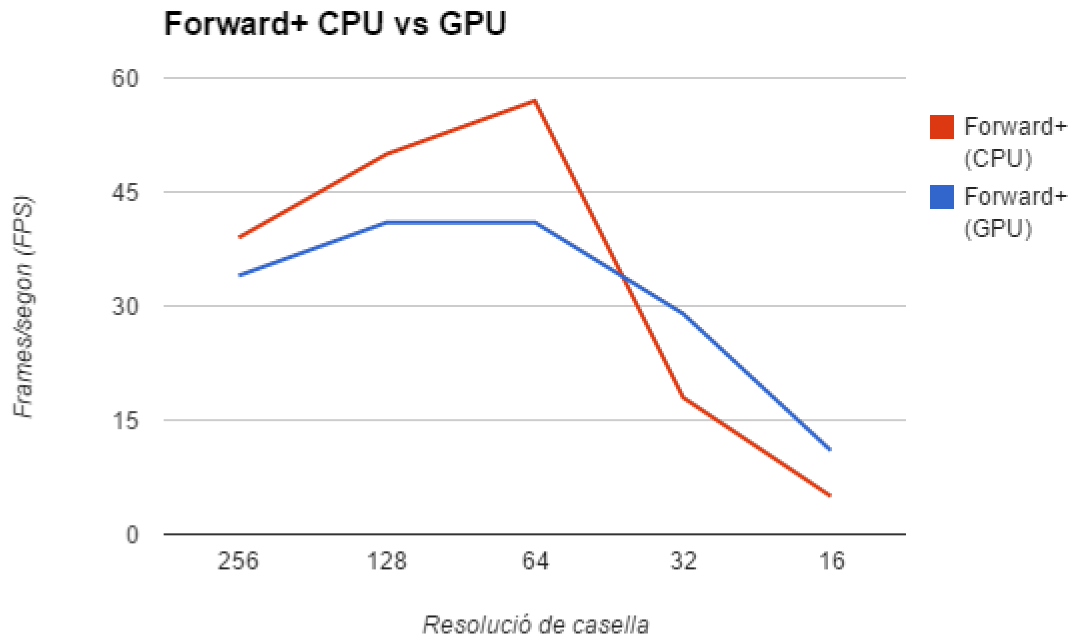


Podem veure que en el cas amb poca atenuació forward rendering és una mica més eficient que deferred shading, això és degut al fet que en tenir llums que ocupen tota la pantalla estem fent blending de tots els punts del viewport per cada llum. Com ja hem comentat anteriorment, el procés de blending és molt costós i això fa que no aprofitem les avantatge de deferred shading per culpa de la gran quantitat de blending.

En canvi en el cas amb molta atenuació podem veure com deferred shading pràcticament triplica en eficiència a forward, que no es veu gaire beneficiat per la reducció de mida de les llums. Les llums, en afectar una part petita del viewport, permeten que deferred shading no hagi de comprovar un gran nombre punts de l'escena múltiples cops, fet que accelera el procés de renderitzat enormement.

Forward+

Hi ha un aspecte de forward+ que encara no hem analitzat: la resolució de les caselles de la graella. Com més petita sigui la casella més precisió tindrem en el nombre de llums que afecten un fragment, però en tenir un nombre més elevat caselles farem més costós el light culling.



En la gràfica podem veure com amb resolucions altes de casella obtenim poques avantatges en fer el renderitzat, ja que la precisió és molt baixa. L'eficiència dels algoritmes millora fins al punt en què el light culling comença a ser més costós que les millores que obtenim en temps de render. És important veure que la versió de GPU obté millor eficiència que la versió en CPU a mesura que fem més petites les caselles. Això ens fa creure que podríem implementar un algoritme paral·lel millor i més eficient que aprofites les avantatge de tenir una graella més fina sense que l'increment del cost del light culling sigui tan problemàtic.

Bibliografia

[1] “Chapter5 - OpenGL Programming Guide” *glprogramming.*
<http://www.glprogramming.com/red/chapter05.html>

[2] Christen, Martin. “Clockworkcoders Tutorials - Per Fragment Lighting” *OpenGL. (*2007)
<https://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/lighting.php>

[3] “Deferred Shading” *wikipedia*.
<http://en.wikipedia.org/wiki/Deferred_shading>

[4] Owens, Brent. “Forward Rendering vs. Deferred Rendering” *tutplus*. (28 Oct 2013)
<http://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342>

[5] Etay Meiri, “Tutorial 35 - Deferred Shading” *ogldev*.
http://ogldev.atspace.co.uk/www/tutorial35/tutorial35.html

[6] Koonce, Rusty “Deferred Shading in Tabula Rasa”, GPU Gems 3.
<http://http.developer.nvidia.com/GPUGems3/gpugems3_ch19.html>

[7] Filion, Dominic and McNaughton, Rob “StarCraft II - Effects & Techniques”, SIGGRAPH, Advances in Real-Time Rendering in 3D Graphics and Games Course 5, Los Angeles, California. (2008)
<http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/01/Chapter05-Filion-StarCraftII.pdf>

[8] Valient, Michal “The RenderingTechnology of Killzone 2”, GDC, Moscone Center, San Francisco. (23-27 Març 2009)
<http://www.guerrilla-games.com/presentations/GDC09_Valient_Rendering_Technology_Of_Killzone_2.pdf>

[9] Anderson Johan “DirectX 11 Rendering in Battlefield 3” *GDC*. DICE (2011)
<http://dice.se/wp-content/uploads/GDC11_DX11inBF3_Public.pdf>

[10] Harada, Takahiro; Mckee, Jay; Yang, Jason C. “Forward+: Bringing Deferred Lighting to the Next Level” EUROGRAPHICS. (2012)
<https://amd.app.box.com/s/9g70td498gmxz5lq8py5>

[11] Luebke, David; Owens, John; Roberts, Mike; Lee, Cheng-Han “Introduction to Parallel Programming” *udacity*. NVIDIA.
<https://www.udacity.com/course/cs344>

[12] Sanders, Jason and Kandrot, Edward. CUDA by example, Addison-Wesle (2011)

[13] “CUDA Toolkit Documentation” *nvidia*.
<http://docs.nvidia.com/cuda/index.html>

[14] van Oosten, Jeremiah “Inroduction to CUDA 5.0” *3dgep*. (26 Oct 2012)
http://3dgep.com/introduction-to-cuda-5-0/