UPC - FIB

TFG

MEMÒRIA

Motor de sistemes de partícules executat a la GPU

Autor:
CRISTIAN RODRÍGUEZ VÁZQUEZ

Supervisor:
ANTONIO CHICA CALAF

15 de juny de 2013

Índex

I Introducció **1**

1 Sistemes de partícules **1**
1.1 Objectiu 2
1.2 Motivació 2

2 Context **3**
2.1 Software de modelatge 3
2.2 Motors gràfics i *game engines* 3

II Capacitats prèvies **5**

3 OpenGL 4.0 **5**
3.1 El *pipeline* programable 5
3.2 Transform feedback 6

4 Disseny *software* **8**
4.1 Classe *ParticleSystemContainer* 8
4.2 Classe *ParticleSystem* 8
4.3 Classe *Texture* 9
4.4 Components 9

III Implementació **11**

5 Fase de *render* **11**
5.1 Billboards 11
5.1.1 *Point sprites* 12
5.1.2 *Geometry shader billboards* 13
5.2 *Soft particles* 13
5.2.1 *Test* manual de profunditat 15
5.2.2 Funcions de suavitzat 17
5.3 Partícules animades 21
5.3.1 Memòria *cache* de textures 23
5.3.2 Atles de textures i LODs (level of detail) 25
5.4 *Normal mapped billboards* 28
5.4.1 *Normal mapping* 29
5.4.2 Model d'il·luminació de Phong 31

5.4.3 Adaptació als *billboards* 35

6 Fase d'actualització **36**
6.1 Atributs dels sistemes de partícules 36
6.2 Càlculs de la física a la GPU 39
6.2.1 Equacions físiques 39
6.2.2 Càlcul de col·lisions 41
6.3 Inicialització de les partícules a la GPU 43
6.3.1 Funcions de soroll 43
6.3.2 *Pseudo random number generators* (PRNG) 44

IV Resultats **46**

7 Proves de rendiment **46**
7.1 *Hardware* utilitzat 46
7.2 *Speed up* 47

8 Conclusions **49**
8.1 Assoliment dels objectius 49
8.2 Treball futur i millores 49

V Gestió del projecte **51**

9 Metodologia de desenvolupament **51**
9.1 Tasques i procediments 51
9.2 Eines de seguiment i validació 52

10 Planificació **54**
10.1 Fase incial 54
10.2 Tècnica de Soft particles 56
10.3 Tècnica de partícules animades 57
10.4 Tècnica de PRNG *Pseudo Random Number Generators* a la GPU 57
10.5 Tècnica de normal mapped billboards 58
10.6 Colisions externes de les partícules 58
10.7 Proves de rendiment entre una versió de CPU i la versió de GPU 59
10.8 Fase final 59

11 Pressupost **60**
11.1 Recursos humans 60
11.2 Costos hardware 61
11.3 Costos software 61
11.4 Cost de les instal·lacions 62
11.5 Cost total 62

Índex de figures

1 Exemples de sistemes de partícules 1
2 Diagrama de funcionament de transform feedback 6
3 Pipeline gràfic de OpenGL 4.0 7
4 Diagrama uml de l'arquitectura emprada 9
5 Esquema de les dos orientacions tradicionals de billboards . . 11
6 Generació dels 4 vèrtexs que formen un billboard 13
7 Comparativa entre no aplicar soft particles (esquerre) i aplicar-la (dreta) 14
8 Exemple de textura de profunditats d'una escena 3D 15
9 Comparativa entre el filtre GL_NEAREST (esquerre) i GL_LINEAR (dreta) 17
10 Casos que cal tractar per cada partícula 18
11 Comparativa entre una funció lineal (esquerre) i una funció quadràtica (dreta) 19
12 Funció exponencial de l'equació 1 20
13 Funció per trossos utilitzada 20
14 Exemple d'atles de textures amb els sprites d'una flama 22
15 Exemple de foc aplicant els sprites 15 23
16 Esquema de la jerarquia de memòria d'una CPU genèrica . . . 24
17 Esquema de la jerarquia de memòria d'una GPU genèrica . . . 24
18 Problema de l'ampliació i la reducció en textures 26
19 Exemple de LOD generat a partir de l'atles 14 26
20 Comparació entre el pitjor filtre i el millor filtre 27
21 Exemple de *nomral mapping* aplicat (esquerre) i del seu corresponent normal map (dreta) 29
22 Exemple de dos espais tangents diferents sobre una superfície . 30
23 Model de lambert 32
24 Exemple d'oclusió (esquerre) amb la correcció de l'equació 8 (dreta) 32
25 Comportament de la llum especular en un mirall perfecte . . . 33
26 Comportament de la llum especular en un mirall imperfecte . 33
27 Model de Phong 34
28 Caiguda de color que implica n a l'expressió $\cos^n \alpha$ 34
29 Exemple d'aplicació del model d'il·luminació de Phong 34
30 Resultat de la il·lumianció en un sistema de partícules de fum 36
31 Comportament del vector refelctit respecte un vector incident en una superfíce 42
32 Exemple de textura generada amb Perlin *noise* 44

33 Diagrama de Gantt 55

Índex de taules

1 Especificacions de la CPU utilitzada 47
2 Especificacions de la GPU utilitzada 47
3 Resultats de rendiment 48
4 taula amb els costos de recursos humans 60
5 taula amb els costos del hardware 61
6 taula amb els costos del software 61
7 taula amb la suma dels costos de les instal·lacions 62
8 taula amb la suma dels costos totals 62

Resum

En aquest TFG (treball de fi de grau) explorarem el món de les sistemes de partícules orientat a l'àmbit dels gràfics per computador. La particularitat que hi ha és que els càlculs d'equacions físiques es faran en paral·lel usant la GPU. A més a més explicaré com he desenvolupat els sistema explicant les diferents tècniques aplicades per aconseguir realisme visual.

Pel desenvolupament he utilitzat *OpenGL 4.0* juntament amb *GLSL 4.0* que m'ofereixent les funcionalitats necessàries per programar la GPU tant per la visualització com pels càlculs de propòsit general. Aquesta particularitat és possible gràcies a la nova característica *transform feedback* que ens ofereixen les últimes versions de *OpenGL*.

Resumen

En este TFG (trabajo de final de grado) exploraremos el mundo de los sistemas de partículas orientado al àmbito de los gràficos por computador. La particularidad que hay es que los cálculos de las ecuaciones físicas se hacen en paralelo usando para ello la GPU. Además, explicaré como he desarrollado el sistema explicando las diferentes técnicas aplicadas para conseguir realismo visual.

Para el desarrollo he usado *OpenGL 4.0* juntamente con *GLSL 4.0* que me ofrecen las funcionalidades necesarias para programar la GPU tanto para la visualización como para los cálculos de propósito general. Esta particularidad es posible gracias a la nueva característica *transform feedback* que nos ofrecen las últimas versiones de *OpenGL*.

Abstract

In this paper we are going to explore the world of particle systems in the computer graphics scope. The particularity is that the physic computations are carried on the GPU to exploit the parallelism of this device. Furthemore, I am going to explain how I developed the system explaining the variety of techniques applied to achive visual realism.

To develop it I have used *OpenGL 4.0* with *GLSL 4.0* which offers the necessary functionalities for carry out the visualization GPU programming and the general purpose GPU programming. This particularity is possible due to *OpenGL*'s new versions feature, *transform feedback*.

Part I
Introducció

1 Sistemes de partícules

En el món de les aplicacions gràfiques en temps real sovint es requereixen eines que facilitin la visualització d'efectes gràfics. En el món dels gràfics per computador s'utilitzen sistemes de partícules per tal d'aconseguir aquest tipus d'efectes.

Un sistema de partícules consisteix, per tant, en la simulació física i visualització d'un conjunt d'entitats molt petites que, en conjunt, intenten simular fenòmens que impliquen un cert grau de borrositat i que no es poden pintar a la pantalla amb les tècniques convencionals. Podem tenir com exemples 1 de sistemes de partícules explosions, boira, pluja, fum, foc, etc.

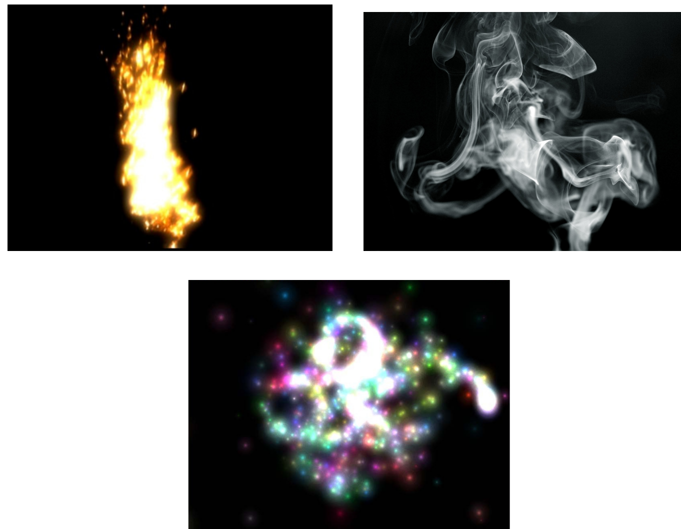

Figura 1: Exemples de sistemes de partícules

Un dels principals problemes que tenen és que es necessiten fer molts càlculs ja que cada partícula necessita ser processada. Aquests càlculs, però, són massivament paral·lels i és el que fan els motors d'avui dia, aprofitar l'arquitectura de les GPUs que ens ofereixen el paral·lelisme necessari per dur a terme aquest tipus de tasques.

1.1 Objectiu

L'objectiu d'aquest projecte consisteix en desenvolupar un motor de sistemes de partícules sobre el qual es pugin provar i desenvolupar diverses tècniques que s'apliquen en aquests sistemes tant de visualització com de *render*. També vull comprovar quin és el *speedup* que es pot guanyar usant una GPU respecte la CPU a les tècniques aplicades.

1.2 Motivació

Els motius pels quals vaig escollir aquest projecte, a part de l'interés algorísmic i tecnològic que hi tinc, són bàsicament dos. El primer es que les llibreries *open source* que existeixen no estan mantenides ni actualitzades des de fa un temps. Cal sumar, també, que la documentació és escassa. Un exemple es *SPARK* [4] que no millora des d'abril del 2011.

L'altre motiu és que els motors actuals tenen llicències privades que requereixen la compra per la seva utilització com es per exemple el cas de la majoria de motors per fer videojocs: Unity 3D [5], Unreal Engine [7], Cry engine [2], etc.

Per aquestes raó crec que es útil invertir un esforç en aquest projecte i donar la oportunitat als usuaris d'aquest tipus de tecnologies d'utilitzar una llibreria que utilitzi les tècniques més actuals de manera gratuïta i amb una bona documentació que no només expliqui com s'ha d'utilitzar, sinó també que expliqui com s'ha desenvolupat.

A més a més, com que es tracta d'un projecte *open source*, existeix la possibilitat que la comunitat de desenvolupadors contribueixi en el manteniment i adició de noves característiques i ampliacions que facin de tot plegat un motor competitiu amb els ja existents que he esmentat prèviament.

Finalment dir que es tracta d'enfatitzar en la implementació i *testing* de tècniques variades que ja s'han usat en altres sistemes reals i adaptar-les a *OpenGL 4.0* que és la API gràfica que he usat pel seu desenvolupament. És un alicient, també, que amb aquesta tecnologia no hi ha cap motor de renom fet i crec que és important intentar ser pioner desenvolupant en una API que tot i que avui dia només la suporten completament pocs dispositius serà l'eina que s'usarà per projectes futurs.

2 Context

Abans de dur a terme un projecte que produeix algun tipus de producte és important analitzar quin és el context en el que es troben productes similars, fer esment de a qui va dirigit i fer un estudi previ de les eines que s'usaran en el desenvolupament.

En els següents apartats presentaré quin és aquest context tenint en compte aspectes com les persones a les quals va dirigit, l'estat actual d'alguns dels sistemes ja existents i les referències bibliogràfiques que faran possible la realització del motor.

Existeixen uns quants escenaris en els que és important l'ús d'aquests tipus de sistemes. Estan relacionats amb el món de la computació gràfica. En els següents subapartats parlaré una mica de qui podria estar interessat en eines d'aquest tipus.

2.1 Software de modelatge

Sovint, els artistes que utilitzen aplicacions de modelatge 3D (com per exemple *blender* [1]) necessiten aquests tipus de sistemes per executar simulacions físiques entre diferents objectes per aconseguir un modelat més realista.

Exemples de casos d'ús podrien ser el modelat del cabell o inclús la creació d'escenes 3D en les que apareguin efectes que ho requereixin. També cal considerar tots aquells efectes que poden estar associats a una animació.

El fet d'incorporar aquesta característica en una etapa de modelatge fa possible que certs efectes es puguin precalcular per no haver de fer-ho en temps real, és a dir, que sigui un *render offline*. La integració de la eina que he desenvolupat, però, no està destinada a integrar-se en aquests tipus de softwares ja que m'he centrat en *render online*, tot i que podria fer-se com a treball futur. En definitiva hi ha tota una branca de utilització de sistemes de partícules al món de l'animaciò 2D i 3D on es poden arribar a simular sistemes de partícules molt realistes.

2.2 Motors gràfics i *game engines*

En el món de desenvolupament d'aplicacions gràfiques s'utilitzen eines per fer el desenvolupament més àgil. Es tracten dels motor gràfics (si es que

només donen suport a la visualització) o *game engines* (si a més de visualització ofereixen altres aspectes com pot ser el càlcul de físiques). És una idea similar a la dels *frameworks* en el desenvolupament de software en general.

A diferència dels softwares de modelatge comentats al subapartat anterior, el que es vol en aquests tipus de sistemes és, normalment, que els sistemes de partícules siguin dinàmics i interactius. Per aconseguir això la etapa de render no pot estar pre-calculada i es fa en temps real. Totes les tècniques i algorismes usats en aquests tipus de sotwares són completament diferents.

És comú que en aquests sistemes existeixi un component que es dediqui a la utilització de sistemes de partícules. La majoria de *engines* que utilitzen les estratègies d'implementació més eficients i actuals per aconseguir bons efectes especials són de llicència privada. Un exemple d'aquests és *Unreal cascade* [6], que és la part de *Unreal engine* que proporciona els sistemes de partícules.

El meu projecte, per tant, beneficia a totes aquelles persones que desenvolupin aplicacions gràfiques de qualsevol tipus que no vulguin o puguin pagar les llicències dels motors, ja que vull que sigui distribuït sota una llicència *open source*. Un clar exemple de públic potencial interessat són els *indie developers*, que fan jocs independets i amb molta freqüència utilitzen eines *open source*.

Part II
Capacitats prèvies

Abans d'entrar en els detalls d'implementació de les tècniques aplicades en aquest apartat es preten explicar quins són els aspectes tecnològics que cal saber per poder entendre correctament el funcionament de la implementació del projecte.

3 OpenGL 4.0

Com ja hem vist, una de les motivacions d'aquest projecte és fer el desenvolupament amb la quarta versió de OpenGL. A priori no sembla que no hi hagi giares implicacions però realment hi ha punts que s'han de considerar respecte openGL 2.0, que és la versió més popular.

3.1 El *pipeline* programable

El primer canvi important que cal comentar, afegit a partir de la versió 3.0, és que ja no existeix el *pipeline* gràfic fix. Es tracta d'un canvi important ja que per utilitzar la API es força l'ús de *GLSL*, que és el llenguatge compatible amb *OpenGL* per la programació de *shaders* que s'executin a la GPU. Amb altres paraules, cal programar obligatòriament totes les etapes bàsiques del *pipeline*, és a dir, el processament de vèrtexs i de fragments, que en versions anteriors tenien una funció fixe implementada.

La primera implicació que té això és que no hi ha cap manera de enviar a renderitzar geometria amb les comandes *glBegin* i *glEnd* que han passat a estar *deprecated*. L'alternativa a això han estat els VBO (*vertex buffer objects*) que ens permeten emmagatzemar dades a la memòria de la GPU. El motiu per forçar l'ús d'aquestes estructures és el de minimitzar transferències de dades pel bus PCI (*peripheral component interconect*) que són el coll d'ampolla de les aplicacions gràfiques en general.

D'aquesta manera, el nou paradigma que cal aplicar és que la informació de cada vèrtex es decideix per programa. Dit d'una altra manera, no estem forçats a tenir estructures com un vector normal, un color, una posició, etc. És el programador qui decideix quines dades tindrà el vèrtex i com les vol

interpretar i usar.

Sembla més difícil fer-ne ús de la nova API però val la pena donat l'increment d'eficiència que es pot obtenir fent una gestió explícita de la memòria. Un cop enteses quines són les implicacions que se'n deriven d'usar aquesta tecnologia parlaré de la nova característica per la qual he decidit usar-la.

3.2 Transform feedback

Els VBO permeten emmagatzemar informació a la GPU però aquesta per defecte només permet ser llegida. El que ens permet *transform feedback*, explicat al llibre [18], és poder manegar les dades completament amb *GLSL* sense que el client tingui que intervenir-hi. Tenint present la figura 2 intentarem entendre com funciona aquest tècnica.

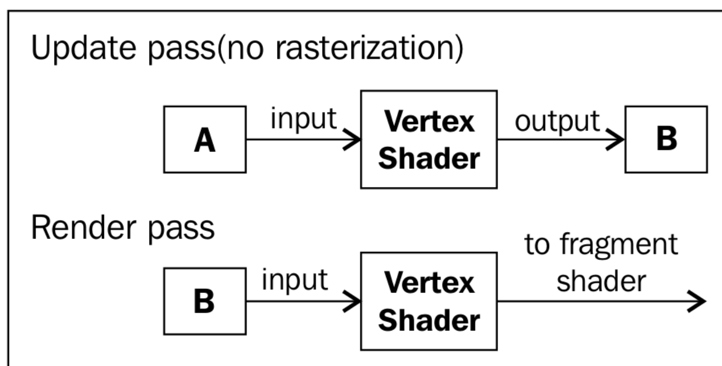

Figura 2: Diagrama de funcionament de transform feedback

Primer de tot necessitem dos VBOs que continguin les dades replicades, anomenem-los A i B. En cada fase de render necessitarem cadascun dels *buffers* tingui un rol concret, input o output. Primerament faré que el buffer A sigui el *input* i el B *output* per conveni. Això es necessari perquè un VBO en OpenGL només es pot o escriure o llegir en un render.

El que tindrem és que cada *frame* necessitem dos renders, un per actualitzar un VBO tenint com a entrada l'altre i l'altre fase per visualitzar, en el cas que necessitem una visualització, els resultats en pantalla tenint com a entrada del *shader* el *buffer* B. Finalment, al següent *frame* caldrà invertir els rols d'aquests VBOs, és a dir, B passarà a ser de lectura i A d'escriptura. Cal apreciar que la part del *pipeline* [3] de la rasterització en endavant no la

necessitem per tant es pot descartar.

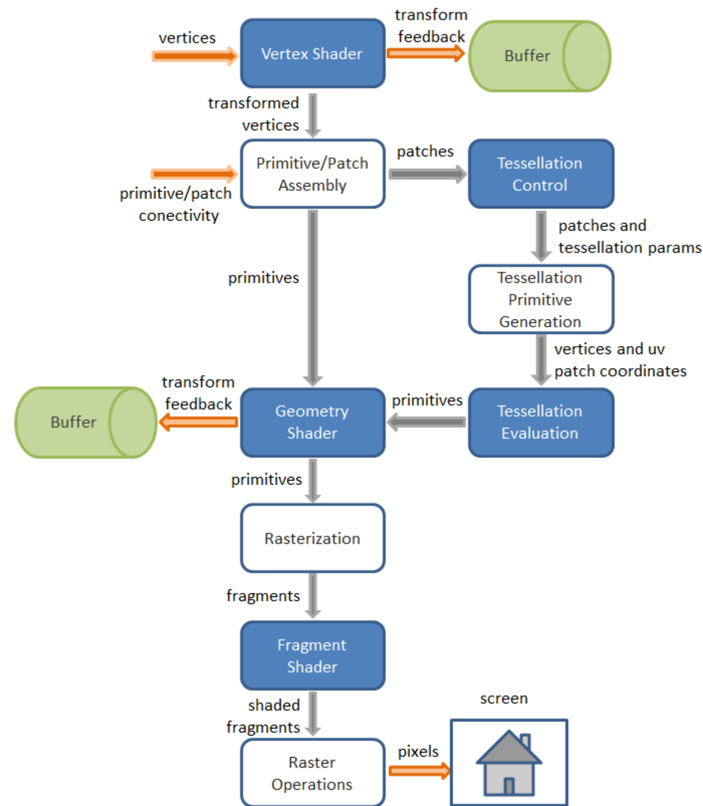

Figura 3: Pipeline gràfic de OpenGL 4.0

Com es pot apreciar, amb aquesta manera de treballar podem enviar unes dades inicials a la GPU i anar-les operant usant sempre només la seva memòria. Un cop acabats els càlculs intermedis, si es necessita saber quin és el resultat final, només cal saber quin dels dos VBO era d'escriptura en l'últim *frame* de càlcul i enviar les dades un sol cop pel bus PCI i processar-les de la manera que necessitem.

Aplicant *transform feedback* al càlcul de les físiques dels sistemes de partícules podem aconseguir un sistema que exploti el paral·lelisme que ens ofereixen les GPU per un problema massiva-ment paral·lel com el que és motiu d'estudi en aquest TFG, els sistemes de partícules.

4 Disseny *software*

Donat que l'objectiu del TFG es centra més en provar diferents tècniques que aconsegueixin el major realisme visual possible no s'ha acabat fent un motor molt complet i complex que ofereixi moltes opcions d'ús. Tot i això, a continuació presentaré el disseny *software* que he aplicat per tal d'aconseguir un sistema el màxim obert a modificacions futures.

L'objectiu de l'arquitectura en un sistema d'aquest estil és aconseguir una plataforma en la qual afegir noves funcionalitats sigui el més fàcil possible. La idea és tenir molts mòduls que es puguin combinar per aconseguir el sistema de partícules desitjat a cada cas. El problema és que treballar amb *shaders* de *GLSL* té una sèrie d'implicacions que no permeten que sigui fàcil el disseny. A la figura 4 es mostra el diagrama UML del disseny del motor. A continuació faré una petita explicació de tot plegat.

4.1 Classe *ParticleSystemContainer*

Aquesta és la classe que dóna accés a tot el sistema. Conté totes les funcions necessàries per gestionar els sistemes de partícules. L'objectiu, com ja diu el seu nom, és fer de contenidor de tots els sistemes que vulguem usar. Aquesta decisió té sentit ja que normalment un sistema de partícules no està composat per un sol sistema aïllat sinó que es composa de diversos sistemes molt diferents. Un exemple pot ser una foguera on s'hi pot trobar el sistema del foc, el fum i les espurnes. L'explicació de que fan algunes de les funcions sortiran més endavant.

4.2 Classe *ParticleSystem*

Es tracta de la classe base del motor. Ens permet definir com és un sistema de partícules aïllat. Conté el recull de funcions que determinen el comportament que tindrà el sistema i les seves característiques visuals. Es pot decidir quines de les tècniques implementades escollir per la visualització. Cal notar que atributs físics com la posició i la velocitat no apareixen com a membres de la classe. Això és perquè aquesta gestió la fa la GPU com ja veurem en apartats posteriors.

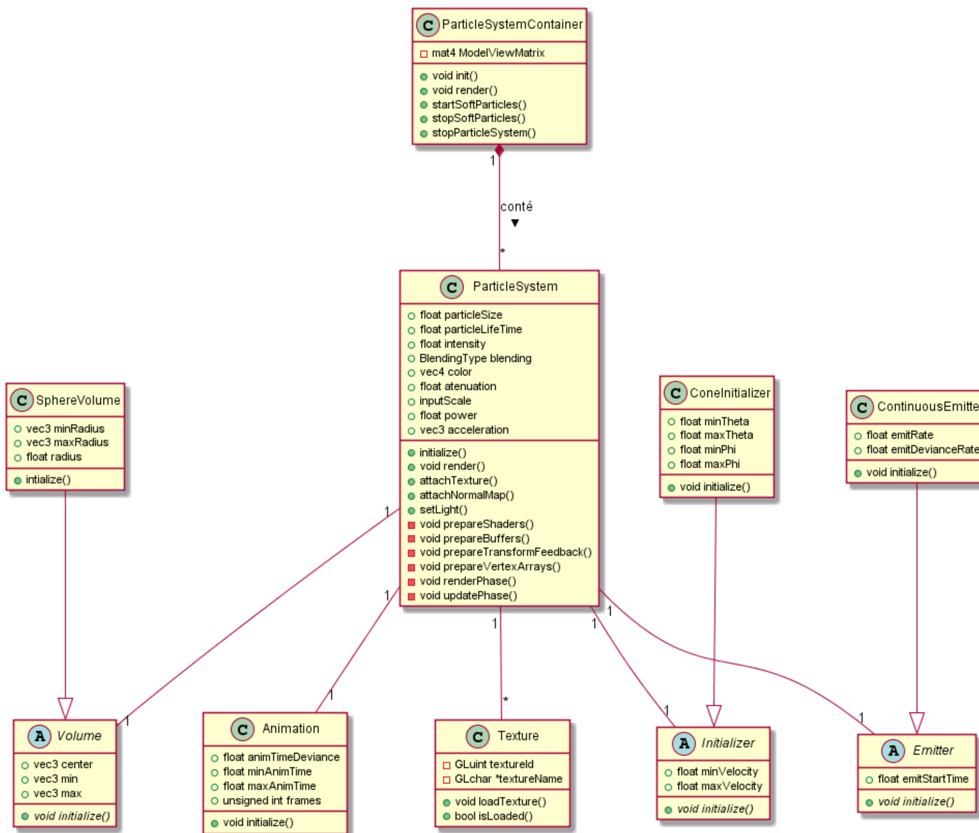

Figura 4: Diagrama uml de l'arquitectura emprada

4.3 Classe ***Texture***

Aquesta classe és l'encarregada de gestionar les textures que s'utilitzin als sistemes de partícules. Tenim funcions de carrega d'aquestes i atributs del nom que usarà al *shader* i l'identificador que usa *OpenGL* per referenciar-la.

4.4 Components

Aquesta és la part més interessant de l'arquitectura. La idea es tracta de, poder afegir a la classe *ParticleSystem* els components que s'encarreguin de la inicialització que determinarà el comportament futur de les partícules. Cadascun d'aquests components hereta d'una classe base abstracta de manera que si es vol ampliar el sistema només cal programar classes que, de manera similar a les existents, inicialitzin cadascuna de les parts que cal inicialitzar per donar un comportament diferent al sistema. Tenim:

- *Emitter*: aquest component dicta quins seran els temps de naixement de les partícules. En el component d'exemple, *ContinuousEmitter*, les partícules neixen una darrere l'altra de manera contínua.
- *Initializer*: és l'encarregat de donar les magnituds que composen els vectors de velocitat inicial de cada partícula. *ConeInitializer* retorna vectors que segueixen un con parametritzat pels angles θ i ϕ.
- *Volume*: diu quines seran les posicions inicials de les partícules en néixer acotades dins un volum. Com exemple de volum he posat un volum esfèric a *SphereVolume*.
- *AnimationBase*: serveix per definir la manera d'animar *sprites* en les partícules. Els detalls s'explicaran en la part pràctica del projecte.

A la pràctica aquest sistema no acaba de ser cert per una raó i és que finalment els paràmetres que s'inicialitzin en aquestes classes ha d'acabar arribant a la GPU cosa que és equivalent a dir que s'ha de tenir codi específic als *shaders*.

Com a treball futur seria interessant l'opció de dissenyar un llenguatge de programació dissenyat per definir comportaments i acabés generant les classes que hereten d'aquestes que he exposat, com per exemple seria *ConeInitializer*, i generés també el codi *GLSL* corresponent. Tal i com es fa ara el sistema no acaba de ser obert ja que existeix aquest acoblament inevitable entre el codi *C++* i el codi *GLSL*.

Part III
Implementació

En aquesta part del document explicaré la part tècnica d'implementació. Es divideix en dos blocs ben diferenciats. El primer, corresponent a la fase de *render* tracta la temàtica relacionada amb la visualització realista dels sistemes. El segon, fase d'*update* es centra en temes relacionats amb càlculs de propòsit general, físiques i actualització de les partícules.

5 Fase de *render*

Un sistema de partícules en el que tenim n partícules movent-se amb un comportament qualsevol per sí sol normalment no gaudeix de gaire realisme. La naturalesa dels *billboards* per defecte ens aporta una sèrie de problemes que cal resoldre per tal d'aconseguir l'objectiu desitjat, que cal tenir clar que varia en cada cas.

5.1 Billboards

Un *billboard* consisteix en una primitiva geomètrica (normalment un *quad*) que té una posició en l'espai molt concreta. La seva característica principal és que estan, o bé orientats sempre a l'observador, o bé orientats al *viewport*, tal i com es veu a la figura 5. Hi ha variants d'aquesta tècnica en que s'apliquen rotacions sobre un eix però sense perdre mai la orientació al observador.

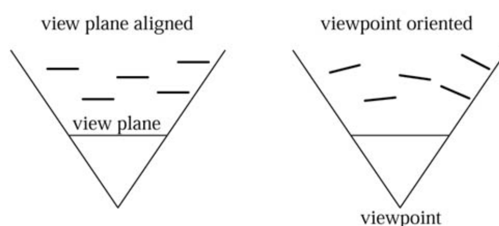

Figura 5: Esquema de les dos orientacions tradicionals de billboards

No només s'utilitzen en el context dels sistemes de partícules. En general, es tracta d'una tècnica basada en imatge que intenta oferir molt detall d'allò que es vol representar. Normalment tenen associada una posició i una textura, a més a més d'un *shader* si es que es volen aplicar efectes. Hi ha altres variants que usen més d'un polígon per *billboard*, per exemple per representar

vegetació.

Per sistemes de partícules és molt comú usar un sol polígon *quad* texturat i orientat al observador, ja que ens assegura que la partícula sempre serà visible i que tots en conjunt donen sensació de volum. Usant aquesta tècnica tenim l'avantatge que la geometria es redueix considerablement. La clau està en que, com podem associar coordenades de textura, ens permet representar entitats que semblin molt complexes amb pocs vèrtexs. Per contra, la generació de fragments que en resulta pot provocar problemes de rendiment si el *fragment shader* és mínimament complex.

Una de les coses que cal decidir a l'hora d'usar *billboards* és com els creem i posicionem a la nostra aplicació. La manera més directe és fer els càlculs adients al client de *OpenGL* tenint en compte la matriu de transformació entre coordenades de món i coordenades de *clipping*. El resultat seria que enviaríem a pintar 4 vèrtexs per cada *billboard*. Per tal de reduir encara més la geometria enviada pel bus *PCI Express* que connecta la CPU amb la GPU podem usar altres estratègies.

5.1.1 *Point sprites*

OpenGL ens proporciona *point sprites* que són una implementació dels *billboards* orientats a la càmara amb funció fixe, és a dir, per *hardware*. D'aquesta manera podem enviar a pintar només els centres dels *billboards* (només 1 vèrtex) de manera que aconseguim reduir la geometria.

Usant aquesta característica és a l'etapa de rasterització del *pipeline* gràfic on s'aplica la funció fixa que ens ofereix la API per generar els 4 vèrtexs orientats al punt on es situa l'observador i generar els fragments corresponents. Els *point sprites* són parametritzables i són molt fàcils d'usar.

Podem especificar el tamany en píxels, tot i que tenen un màxim de tamany que varia en funció de la GPU. Se'ls hi poden aplicar textures i s'autoassignen les coordenades de textura. Es poden consultar dins dels *shaders* mitjançant la instrucció *glPointCoords*. També, un cop establert el tamany, el podem modificar els *shaders* amb la variable *built in glPointSize*. Per més informació veure [3].

El principal avantatge que tenim usant aquest sistema és que la generació dels *billboards* és molt ràpida. Per contra la limitació de tamany pot ser un problema, ja que hi ha contextos en els que necessitem partícules grans. A

més a més, s'ha de gestionar a mà el tamany en funció de la profunditat dins del *shader*. És per això que m'he platejat més alternatives.

5.1.2 *Geometry shader billboards*

Dins el *pipeline* gràfic disposem d'una etapa que ens permet generar nova geometria a partir de la geometria que prové del processador de vèrtexs. Es tracta del *geometry shader*. Seguint una mica l'esquema anterior la idea és seguir enviant només punts a pintar de manera que quan entrin a l'etapa d'aquest *shader* puguem generar els corresponents *billboards* [18].

La manera de fer-ho és molt senzilla. Tenint el centre P, que és el vèrtex que hem enviat a dibuixar, hem de generar 4 vèrtexs tal i com mostra la figura 6 en coordenades de càmera. D'aquesta manera assegurem que estem orientant els *billboards* al observador.

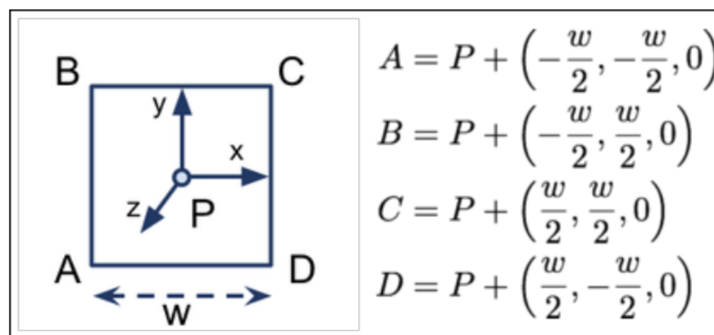

Figura 6: Generació dels 4 vèrtexs que formen un billboard

D'aquesta manera eliminem les restriccions de tamany que ens genera la metodologia anterior a canvi, però, d'un *overhead* en el rendiment ja que estem enfrontant un programa nostre contra una implementació *hardware*. Cal destacar també que la funcionalitat que ofereixen els *geometry shaders* de crear i destruir vèrtex en mig del *pipeline* crea una complexitat *hardware* afegida dins la GPU que fa que el rendiment baixi. La solució doncs és usar una tècnica o una altra en funció de les necessitats que ens plantegi el sistema de partícules que volem representar.

5.2 *Soft particles*

Sovint, per la naturalesa que tenen els *billboards*, donen problemes a la hora de visualitzar-los. A la majoria d'aplicacions gràfiques els sistemes de

partícules interactuen i es renderitzen juntament amb una escena. Quan aquests intersecten amb aquesta, generen artefactes que fan que es vegi la intersecció dura.

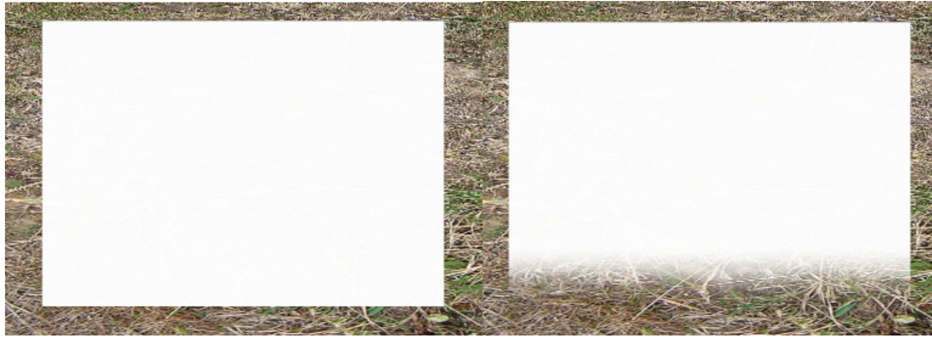

Figura 7: Comparativa entre no aplicar soft particles (esquerre) i aplicar-la (dreta)

Per tal de millorar aquest aspecte existeix una tècnica anomenada partícules suaus (o *soft particles*) que evita aquest problema explicada a [12]. La idea que hi ha darrere és fer que aquestes interseccions estiguin més difuminades jugant amb el paràmetre alfa de cada *billboard* en funció de la distància de cada fragment generat amb l'escena. Veiem amb més detall en què consisteix.

Els sistemes de partícules sovint es composen de *billboards* transparents. En aquests cassos és important tenir una sèrie de consideracions a l'hora de fer el *render* [8]:

- Pintar els objectes translúcids els últims. És necessari per tal que els objectes que hi ha darrere un objecte translúcid apareguin.
- Desactivar el test de *z-buffer*. Si no es desactivés, els píxels que es renderitzessin darrere un objecte translúcid es descartarien, fet que és incorrecte ja que a causa del *blending* el color del píxel corresponent hauria de ser modificat.
- Enviar els objectes translúcids ordenats per profunditat. Si no es fa d'aquesta manera el *blending* resultant és incorrecte. Aquest últim tema no el tractaré en aquest TFG ja que en el cas dels sistemes de partícules el guany que obtindríem aplicant la ordenació dels *billboards* no és molt gran respecte l'esforç que s'ha d'aplicar per fer una implementació d'un algorisme d'ordenació en GPU.

Si apliquem les consideracions anteriors sense cap mena de restricció el resultat és que no hi ha cap tipus de *test* que descarti aquells fragments que no s'haurien de veure. En altres paraules, l'únic que aconseguiríem és que només es veiessin els píxels de les partícules que no tinguessin res davant respectant, però, les condicions exposades a diferència que el test de profunditat normal. Aprofitant la tècnica que tractem en aquest punt implementaré aquesta comprovació de manera manual.

5.2.1 ***Test* manual de profunditat**

Les partícules s'han de renderitzar al final de tots el objectes d'una escena en cada render. Si s'ha de fer el test de profunditat manualment cal alguna manera de comprovar si cadascun dels fragments generats per les partícules s'ha de descartar o no. Per fer-ho he aplicat una tècnica molt comú en gràfics.

Consisteix en guardar el *z-buffer* en una textura associada a un FBO (*frame buffer object*) amb resolució del tamany del *viewport*. Quan renderitzem les partícules al *fragment shader* cal consultar la textura de profunditats. Per accedir la textura hem de veure quina és la correspondència entre el píxel al *viewport* i la coordenada de textura correcte.

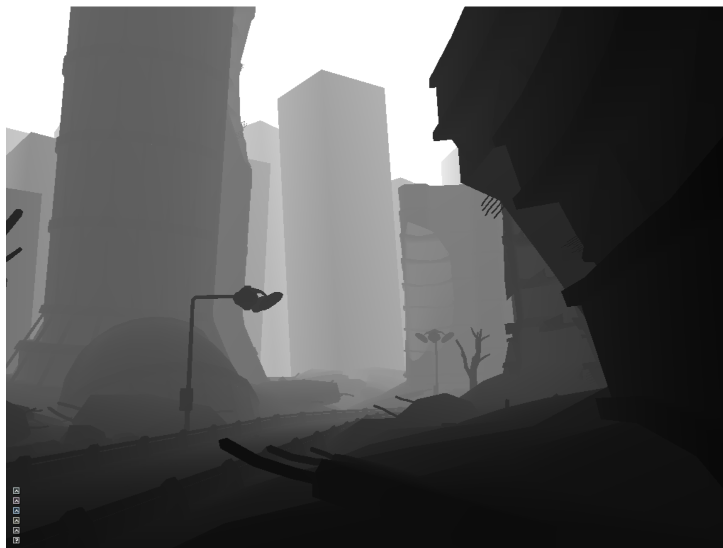

Figura 8: Exemple de textura de profunditats d'una escena 3D

Les coordenades que obtenim al consultar la profunditat del fragment estan en espai de dispositiu, és a dir, que la coordenada x horitzontal es correspon

amb el píxel $x \in [0, viewort\ width]$ i la coordenada y vertical es correspon amb el píxel $y \in [0, viewport\ height]$. Les coordenades de textura s i t, per contra, estan normalitzades al interval $[0, 1]$ per tal que sigui independent del tamany d'una textura accedir a un *texel*. Es necessari, doncs, al *shader* normalitzar les coordenades del fragment que estem tractant per tal de saber quin *texel* li correspon exactament a aquell píxel.

La textura que emmagatzema les profunditats de l'escena ha d'estar parametritzada de manera que s'adapti a les necessitats. En primer lloc només necessitem un valor en comptes dels 4 components que ens pot oferir una textura. Per tant, cal dir-li a *OpenGL* que el que contindrà la textura és el component de profunditat per tal que interpreti que els 32 bits que normalment s'usen pel color (8 bits per cada component *rgba*) l'interpreti com un de sol. També cal especificar que la textura emmagatzemi nombres reals de coma flotant (*floats*) ja que és el tipus que emmagatzema el *z-buffer* sinó per defecte es guarda bytes.

Cal assignar també el filtre adequat. Els filtres, en textures, s'usen per obtenir valors aproximats quan la textura que es vol aplicar i la primitiva (triangle) sobre el qual es vol aplicar no tenen una correspondència 1:1 en les 2 dimensions de la textura s i t. Hi ha diversos tipus de filtres. Els dos més bàsics són:

- GL_NEAREST: de tots els *texels* possibles agafa el que estigui més proper a les coordenades que li passem.
- GL_LINEAR: dels 4 *texels* veïns més propers al *texel* que consultem fa una interpolació lineal.

En el nostre cas la correspondència sí és 1:1 ja que la textura té la resolució que tingui el *viewport*, per tant, per estalviar la operació d'interpolació que fa GL_LINEAR, tot i que a les GPUs actuals es fa per *hardware* i està molt optimitzada, és millor escollir GL_NEAREST. Finalment cal fer una última consideració, tot i que no és obligatòria, a l'hora d'escollir la funció de *clamp* al marc de la textura.

El *clamp* a la bora de la textura el que determina és a quin valor s'ha d'arrodonir quan una textura es repeteix i estem a la bora. En principi això no hauria de passar mai ja que la textura és exactament del mateix tamany del *viewport*. És, igualment bona pràctica fer-ho. Per defecte els valors s'arrodoneixen a 0 que en el *z-buffer* vol dir més a prop. El que fem es dir-li a

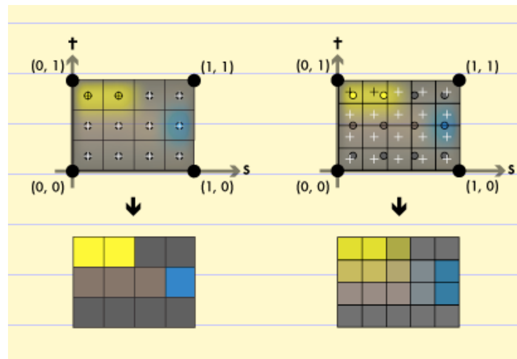

Figura 9: Comparativa entre el filtre GL_NEAREST (esquerre) i GL_LINEAR (dreta)

OpenGL que arrodoneixi a 1 i així s'estableix aquest valor per defecte en cas que es necessiti.

Fetes aquestes consideracions veiem l'algorisme que s'aplica. Hem de fer el que faria un *test* de *z-buffer* manualment. Si el fragment està per sobre del que ja hi havia escrit al *buffer* el processem. En cas contrari no s'ha de veure en pantalla i el descartem. El pseudocodi associat seria el següent:

```
coordenadesNormalitzades := (posicioFragment.x/viewport.x,
   posicioFragment.y/viewport.y)
profunditatActual := consultaTextura(coordenadesNormalitzades)
si (profunditatActual < profunditatFragment) descarta
sino processa
```

Aquesta tècnica implica que a cada *frame* calgui un render addicional previ al render final de tota l'escena per tal de guardar-ne les profunditats a la textura corresponent. Feta aquesta preparació necessària ja estem en disposició d'aplicar l'algorisme de partícules suaus tal i com explica l'apartat següent.

5.2.2 Funcions de suavitzat

La clau per destruir les interseccions dures entre les partícules i l'escena és aplicar una funció que donat un fragment d'un *billboard* retorni la transparència addicional que cal aplicar-li. Aquesta funció ha de rebre dos paràmetres: la diferència entre la profunditat del fragment de la partícula i la de l'escena que vulguem considerar pel suavitzat i un valor constant que serveixi com

a modificador per la quantitat de transparència que es vol aplicar. Tenint això en ment cal fixar-se en quins cassos tenim. Un fragment en el tema que estem tractant pot trobar-se en 3 situacions que han d'acabar definint el comportament de la funció:

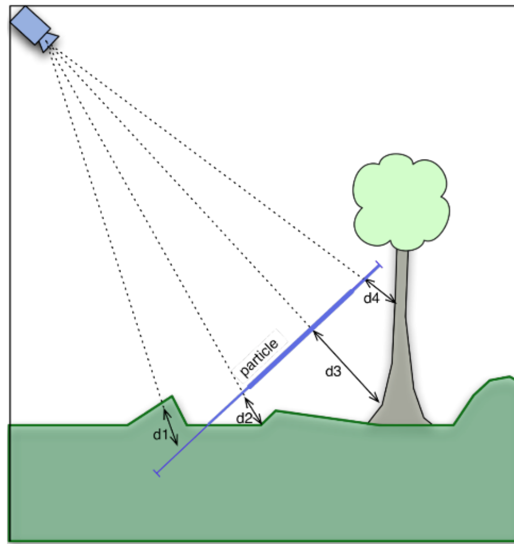

Figura 10: Casos que cal tractar per cada partícula

- Aplicar transparència absoluta o descartar fragments per darrera de l'escena des de l'observador, que és el que hem aplicat al apartat anterior.
- Quant més proper està el fragment a l'escena més transparent cal fer-lo. És el cas de $d2$ i $d4$ a la figura 10.
- Quant més llunyà estigui el fragment de l'escena més opac ha de ser, fins al punt que si és prou llunya no se i apliqui cap α. Seria el cas de $d3$ a la figura 10

Cal dissenyar una funció que respecti aquestes característiques. Donat que el valor α de transparència està contingut al interval $[0, 1]$ la funció de suavitzat també s'ha de moure en aquest espai d'imatges. La funció més trivial que es pot aplicar per aconseguir allò que es vol es una funció lineal:

```
\label{lineal}
c = clamp((profunditatEscena - profunditatParticula) *
    valorEscala, 0.0, 1.0);
```

La funció l'he expressat directament en codi *GLSL*. La variable c ve de contrast, que es com he decidit anomenar el fet de suavitzar. La funció clamp de *GLSL* el que fa es acotar el primer paràmetre en el rang de valors comprés en els dos paràmetres següents. En aquest cas volem que el resultat de la operació del primer paràmetre s'escali entre 0 i 1. El contrast és el valor que finalment multiplicarà el component α del color del fragment que estem processant en aquell moment. El valor d'escala, permet definir quina serà la distància màxima amb l'escena que volem tenir en compte per suavitzar el fragment.

Figura 11: Comparativa entre una funció lineal (esquerre) i una funció quadràtica (dreta)

L'avantatge que tenim d'aplicar aquesta funció és que el càlcul que implica és molt senzill. Per contra, la naturalesa que té la funció *clamp* fa que es generin discontinuïtats molt visibles al suavitzat com es pot veure a la figura 11. Tenint present aquest fet cal buscar una alternativa que faci un *fade* més suau. El que necessitem, doncs, és una funció corva de manera que els canvis no siguin bruscos. Una solució que és pot aplicar és la de l'equació següent:

$$f(d) = 1 - 2^{2 \cdot clamp(d,0,1)^c} \tag{1}$$

La funció de la figura 12 presentada és molt fàcil d'implementar. El paràmetre d es correspon amb la diferència entre la profunditat del fragment i la profunditat de l'escena de manera anàloga a la primera funció. Tot i que millora el resultat de la primera funció, la asimetria que presenta pot seguir generant discontinuïtats al suavitzat. Cal notar que aquesta millora té un cost computacional lleugerament més elevat respecte la senzillesa de la funció lineal. Per tal de tenir una funció simètrica que ens arregli aquestes discontinuïtats la solució és dissenyar una funció per trossos com la de la figura 13.

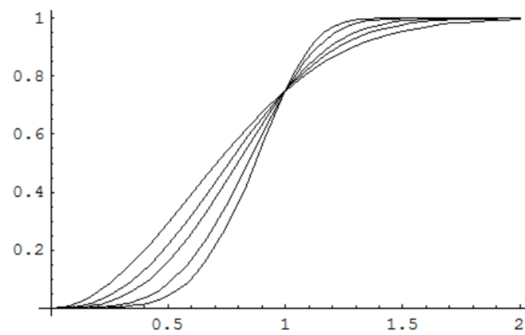

Figura 12: Funció exponencial de l'equació 1

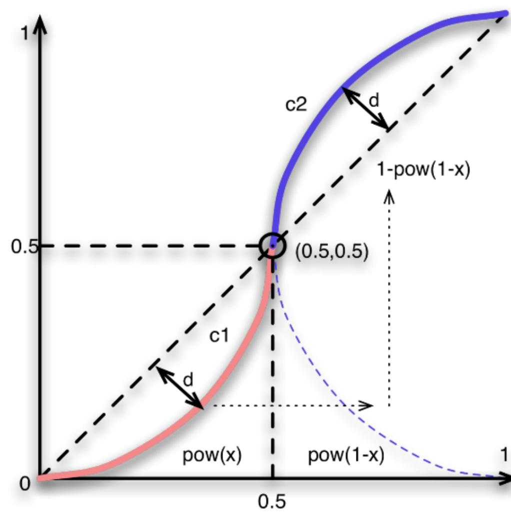

Figura 13: Funció per trossos utilitzada

Com es pot veure, aquesta funció es composa de dos trossos on el primer tros que va des de $x = 0$ fins a $x = 0.5$ i es una funció que té com exponent un paràmetre anomenat *contrast power* (tros vermell) i el segon tros és el complementari de la mateixa funció elevat al complementari del exponent i que engloba la resta de la funció (tros blau). El paràmetre x es correspon, un cop més amb la diferència de profunditats. Per que es vegi una mica millor considerem el següent pseudocodi:

```
C = 0.5*pow(clamp(2*(( diff > 0.5) ? 1 - diff : diff, 0, 1),
    ConstrastPower)
C = (diff > 0.5) ? 1 - C : C
```

El que es fa es agafar com a base el valor de la diferència de profunditats o el seu complementari en funció de si es troba per sobre o per sota de la meitat de la funció. Finalment de la mateixa manera aplicarem el complementari o no al resultat aplicant la mateixa condició.

Notar que *ContrastPower* serà el paràmetre que decidirà la quantitat de diferència de distància que considerarem per aplicar el suavitzat. La implementació d'aquesta funció és la més costosa però alhora la que ofereix millors resultats. Es per aquest motiu que vaig decidir incorporar les tres funcions al motor per que el desenvolupador pugui escollir qualsevol de les 3 funcions depenent de les seves necessitats.

Gràcies a aplicar aquesta tècnica s'ofereix la possibilitat de donar molt més realisme a sistemes en els que hi ha una intenció prèvia de que les interaccions amb l'escena no es notin. Clars exemples són el foc o el fum. Serà l'artista el que, jugant amb els paràmetres explicats, aconsegueixi els resultats que més s'adaptin al que es vol aconseguir tant visualment com en temes de rendiment.

5.3 Partícules animades

Hi ha sistemes de partícules en els que es necessita amagar el fet que la seva composició estigui feta de partícules molt petites. Un exemple d'això podrien ser els fluids. Per poder fer simulació de fluids hi ha, però, dos aspectes que fan que la càrrega computacional sigui massa gran com per poder fer la simulació en temps real amb una màquina normal i corrent:

- La quantitat de partícules ha de ser massa gran.
- Els càlculs que determinen el comportament (com per exemple forces internes) suposen un cost molt elevat.

El foc per exemple també es pot considerar un fluid però avui dia hi ha molts videojocs on es pot veure el foc. La explicació és que es fan aproximacions molt més simples de computar però que imiten aproximadament el comportament que s'esperaria a la realitat. Una d'aquestes tècniques consisteix en animar cada partícula amb *sprites* tal i com s'explica a [14]. Fer-ho així fa que la quantitat de partícules necessàries sigui molt menor i el comportament depengui en certa manera una mica més de les animacions escollides. Per contra, requereix més treball per part d'un artista a l'hora de dissenyar el

sistema.

Figura 14: Exemple d'atles de textures amb els sprites d'una flama

El problema en aquest cas rau en com gestionar el problema de tenir diferents textures que ens representin cadascun dels *frames* que volem que tingui l'animació de les nostres partícules. La primera solució que vaig aplicar va ser usar un *array* de textures de 2 dimensions. Consisteix en una sola estructura de dades que conté diversos *slides* o làmines on s'emmagatzemen textures 2D normals i corrents. La peculiaritat que tenen és que es pot consultar qualsevol d'aquests *slides* amb una tercera coordenada de textura. Conceptualment es pot veure com una estructura de tres dimensions.

Si totes les partícules es visualitzen al mateix *frame* d'animació alhora el resultat no es gaire satisfactori ni realista ja que, en efectes com pot ser el foc, tot tendeix a ser més heterogeni. Per tal d'evitar aquest fet cada partícula s'inicialitza amb un *frame* aleatori d'animació associat, afegint també que la durada de cada *frame* té una durada amb una desviació també aleatòria, i d'aquesta manera es forma un sistema molt més heterogeni que millora el resultat visual. De la mateixa manera es pot escollir el temps de cada frame, també amb una certa desviació.

Finalment, per tal que les transicions entre *sprites* siguin més suaus, faig un *blending* amb una funció lineal similar a l'explicada a l'apartat de la tècnica *soft particles* entre el *frame* actual i el següent de cada partícula. Un exemple del resultat de tot plegat és el de la figura 15. Aquestes dos últimes millores

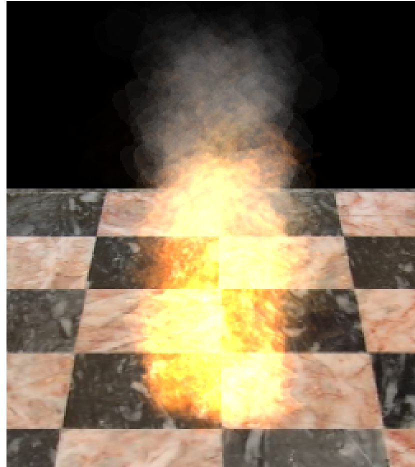

Figura 15: Exemple de foc aplicant els sprites 15

de diversitat visual tenen, però, un impacte molt fort en el rendiment de l'aplicació. El motiu és l'el·lecció d'emmagatzemar els *frames* en *arrays* de textures 2D. Veiem una explicació més detallada d'aquest fet.

5.3.1 Memòria *cache* de textures

L'arquitectura d'una GPU no és com l'arquitectura que té una CPU d'avui dia donat que els requisits són totalment diferents. Hi ha diverses diferències entre els dos xips. Una de les més destacables sens dubte és la de la jerarquia de memòria que usa cadascuna. En una CPU volem que les latències amb memòria siguin el més baixes possible per la qual cosa hi ha una jerarquia de memòria molt complexa composada per diversos nivells de memòries *cache* que intenten amagar el cost d'accés que existeix per accedir a memòria principal.

Com es pot apreciar a la figura 16, cadascun dels nuclis d'una CPU tenen la seva unitat de control, ALUs amb suport per coma flotant i un banc de registres. Seguidament destaquem els diferents nivells de memòria de la jerarquia, L1, L2, L3 i memòria principal.

Per contra, en una GPU formada de molts petits nuclis de càlcul les necessitats són molt diferents. No importa quina sigui la latència, el que es vol es tenir ocupades aquestes unitats de càlcul el màxim temps possible. En altres paraules, el *target* és aconseguir un ample de banda entre el xip gràfic i la seva memòria més gran possible sense importar tant la latència que existeixi. És per aquest motiu que, per les dades, és a dir, vèrtexs i fragments, els

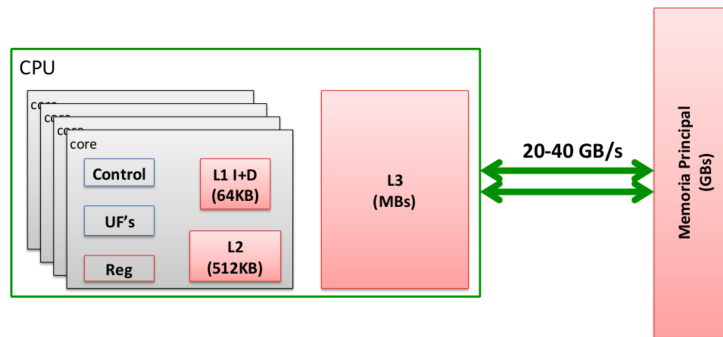

Figura 16: Esquema de la jerarquia de memòria d'una CPU genèrica

nuclis no disposen de cap jerarquia de memòria *cache*.

En l'únic context que té sentit disposar d'una memòria *cache* en un dispositiu gràfic és per les textures [10]. La manera en que aquestes s'emmagatzemen en memòria és molt especial. No és un problema anàleg al d'emmagatzemar una matriu per tal que a cada línia de *cache* estigui alineada. El que determina el posicionament de la textura en memòria és la fase de rasterització. La jerarquia de nivells, però, continua essent molt més bàsica que la d'una CPU.

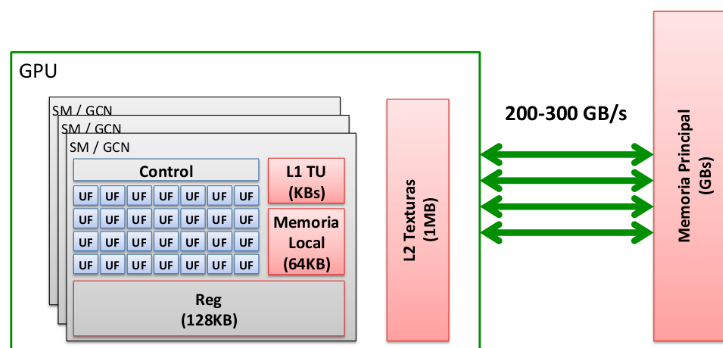

Figura 17: Esquema de la jerarquia de memòria d'una GPU genèrica

Normalment només disposem de 2 nivells abans de memòria principal en les targes més decents tal i com es veu a 17. Els diferents nuclis disposen de moltes unitats de coma flotant que comparteixen una cache de primer nivell L1 com es veu a 17. Seguidament hi ha un segon nivell L2 el qual comparteixen tots els nuclis i posteriorment la memòria principal. Notar que hi ha una diferència molt gran també en l'ample de banda que ens ofereixen els dispositius.

El motiu d'aquesta decisió de no invertir més esforços en aquestes memòries és que quan pintem primitives normalment a cada *batch* se li apliquen les mateixes textures amb les mateixes coordenades. Per aquest motiu, la decisió d'aplicar 2 *frames* interpolats i diferents per cada partícula fa que es trenqui aquest esquema i el resultat es que hi hagin més fallades de *cache* que les que s'haurien de produir. Es per això que vaig haver de descartar la comoditat de accedir a les textures per coordenada s i t pel color i w pel *frame* i usar un altre tipus d'estratègia.

5.3.2 Atles de textures i LODs (level of detail)

Finalment la solució a tots els problemes anteriors ha estat tenir una textura més gran que contingui tots els *frames*. Aquesta manera de encapsular diverses textures és molt comú al món dels gràfics i es coneix amb el nom d'atles de textures.

Ara en comptes de seleccionar el *frame* que es vol usar amb una tercera coordenada el que val fer és saber quines coordenades de textura té associades cada *frame*. Hi ha dos paràmetres que determinen el nombre de files i columnes de *frames* que conté la textura. La textura resultant s'ha forçat que sigui potència de dos ja que, tot i que les APIs gràfiques suporten textures que no tenen aquesta restricció, segueix essent millor de cara al rendiment.

Amb aquests sistema només cal carregar una textura a *cache* a diferència del mètode anterior on es necessitaven carregar tantes textures com *frames* volguéssim que tingués l'animació en el pitjor dels cassos. Tot i això un aspecte molt important a tenir en compte és que el tamany de la textura que s'hagi de consultar ha de ser el més petit possible. Es bastant obvi ja que quan més gran sigui el conjunt de dades que moguem més cost de memòria es genera.

El problema que sorgeix si utilitzem textures molt petites es que si ens apropem massa al sistema de partícules apareixen problemes d'ampliació, ja que cada *texel* (unitat que composa una textura) correspon a més d'un píxel. De la mateixa manera, si ens allunyem massa i les partícules es veuen molt petites tenim el problema contrari, a cada píxel li correspon més d'un *texel*.

Per solucionar-ho el que es fa es tenir diferents nivells de detall anomenats LODs. Bàsicament un LOD consisteix en una versió simplificada d'un recurs, en aquest cas una textura. Per decidir quin LOD cal escollir en cada cas hi

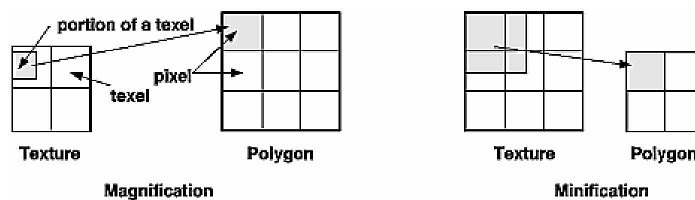

Figura 18: Problema de l'ampliació i la reducció en textures

ha diverses tècniques i tipus de filtrats diferents. En un cas general els filtrats necessaris per tal que la visualització sigui correcte en totes les orientacions d'una primitiva han de ser molt sofisticats.

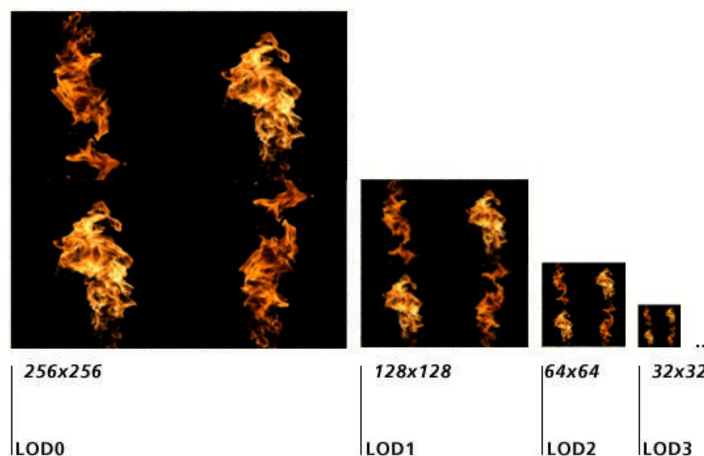

Figura 19: Exemple de LOD generat a partir de l'atles 14

Al cas dels *billboards* tot és molt més simple ja que sempre els veiem amb la mateixa relació d'aspecte i orientació al espai 3D. La solució és aplicar el filtre adequat tant en problemes d'ampliació com de reducció, és a dir GL_NEAREST o GL_LINEAR tal i com s'ha explicat a la subsecció anterior, però tenint en compte quin subnivell de detall aplicar a cada cas en funció de la distància a la que ens trobem. *OpenGL* ens ofereix 4 tipus de filtres bàsics que es diferencien per com es filtra el LOD i un cop tenim el LOD filtrat com es filtra la subtextura resultant:

- GL_NEAREST_MIPMAP_NEAREST: Decideix quin és el LOD més adequat amb el filtre GL_NEAREST i seguidament torna a aplicar el mateix filtre sobre el LOD escollit. En total consulta 2 texels.

- GL_LINEAR_MIPMAP_NEAREST: Decideix quin és el LOD més adequat amb el filtre GL_NEAREST i seguidament aplica el filtre GL_LINEAR sobre el LOD escollit. En total consulta 4 texels.
- GL_NEAREST_MIPMAP_LIENAR: Interpola els dos LODs més adequats amb el filtre GL_LINEAR i seguidament aplica el filtre GL_NEAREST sobre el texel previament interpolat. En total consulta 4 texels.
- GL_LINEAR_MIPMAP_LINEAR: Interpola els dos LODs més adequats amb el filtre GL_LINEAR i seguidament aplica el filtre GL_LINEAR sobre el texel previament interpolat. En total consulta 8 texels.

El filtre finalment escollit ha d'estar equilibrat entre la qualitat obtinguda i nombre de *texels* consultat ja que pot suposar un cost addicional molt elevat. El primer de tots, que és el més lleuger quant a càrrega computacional, té uns resultats no gaire satisfactoris visualment donada la senzillesa amb la que decideix quin píxel és més adequat, que és simplement per proximitat.

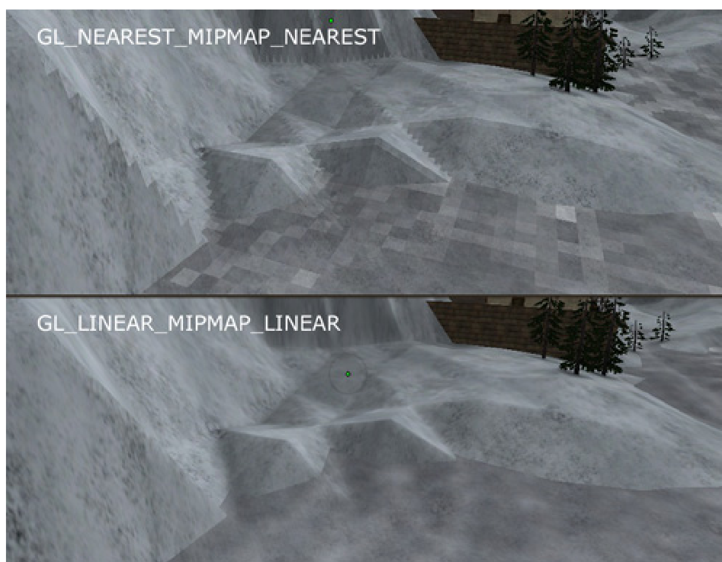

Figura 20: Comparació entre el pitjor filtre i el millor filtre

Si escollim l'últim el rendiment baixarà però a canvi d'augmentar la qualitat del *render*. Per tant el que determina el filtre més adient depèn de les condicions que ens proposem. Si volem tenir sistemes de partícules amb moltes partícules probablement el millor filtre visual ens donarà problemes, sobretot si no es disposa d'una GPU molt potent com és el cas, per exemple, de les GPUs que venen integrades amb les CPU. Per tant, la solució és ajustar el filtre a les necessitats que ens imposi el sistema que volem modelar.

5.4 *Normal mapped billboards*

Com a totes les aplicacions gràfiques que pretenguin ser realistes, la il·luminació és un tema molt important. El repte és que estem tractant amb *billboards*, per tant totes les tècniques d'il·luminació per vèrtex (que són les més eficients) les hem de descartar ja que la geometria que enviem a la GPU són només vèrtexs a l'espai 3D que acaben essent *quads*. Totes les tècniques d'il·luminació el que tracten és la manera de calcular el color de cada píxel. Per fer-ho existeixen molts models d'il·luminació que es poden aplicar en cassos molt diversos. Hi ha dos tipus d'il·luminació:

- Directa: simula els efectes que provocar una llum situada en un lloc concret directament sobre els objectes de l'escena.
- Global: té en compte els rebots que sofreix la llum, tant llum ambient com una llum de puntual.

En aquest TFG només m'he centrat en la il·luminació directa d'una sola llum puntual esfèrica. Les variables vectorials que intervenen en el càlcul de la il·luminació normalment són el vector normal a la superfície il·luminada i el vector de llum incident. Altres són la distància a la font de llum i l'angle que formen els dos vectors anteriors. El com s'usin aquestes variables depèn del model d'il·luminació que vulguem aplicar. Una altra característica que tenen en compte aquests models són els components que té la llum i els materials sobre els quals incideix:

- Ambient: quantifica la quantitat d'un color concret que se li aplica a un material comptant una llum omnidireccional.
- Difusa: quantifica la quantitat de llum que físicament rebotaria en totes direccions en incidir sobre una superfície i quin és el color que absorbiria el material o emetria la llum.
- Especular: diu quin és el color que es reflectiria en incidir sobre un material. S'aplica a superfícies reflectants.
- Índex de brillantor: coeficient que dicta com ha de ser l'àrea sobre la qual incideix la llum especular. Aquest paràmetre només s'aplica a materials.

El tema de la il·luminació s'aplica a aquells sistemes de partícules que només absorbeixen reflectint o refractant la llum ja que hi ha altres que directament ells per si mateixos són una font de llum. Aquests últims no els consideraré

tampoc. Un exemple de sistema de partícules que es veu afectat per la il·luminació és el fum. La millor manera d'il·luminar correctament la geometria és enviar models amb moltes primitives, quantes més millor. Això no és viable en sistemes de partícules perquè serien models massa complexos i, a més a més, molts.

Hem decidit, per aquest motiu, la tècnica més adient per resoldre el problema ha de ser alguna basada en il·luminació per píxel. Això té el desavantatge que la complexitat del càlcul depèn del nombre de fragments generats, i normalment se'n generen molts. Tot i això a la pràctica es poden aconseguir rendiments bastant bons. El que cal torbar és la manera adequada de fer que cada píxel varïi el seu color respectant el detall que poden arribar a tenir les textures. La tècnica que he usat i presento a continuació és *normal mapping*.

5.4.1 *Normal mapping*

En general *normal mapping* (o *bump mapping*) és una tècnica que pretén simular el comportament de la llum dels detalls de rugositat d'una malla poligonal. Un dels avantatges que té i que el fa molt usat és que no requereix de molts polígons per dur a terme aquesta tasca. El que es fa és usar un *normal map* que no és més que una textura on els components r, g, b emmagatzemen els components x, y, z d'un vector normal pertorbat per cada píxel. Finalment s'aplica aquesta normal pertorbada en comptes d'aplicar el vector normal a la superfície.

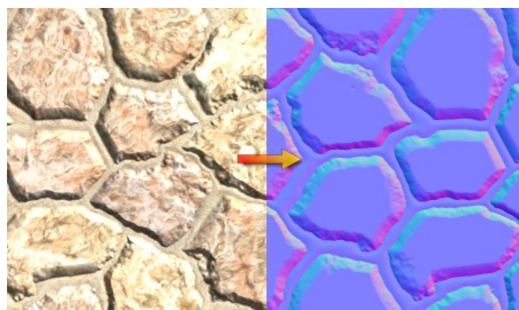

Figura 21: Exemple de *nomral mapping* aplicat (esquerre) i del seu corresponent normal map (dreta)

Aquestes textures es poden produir de moltes maneres. Una d'elles es a partir d'un *height map*, una textura que emmagatzema l'alçada de cada píxel. Una altra molt utilitzada és a partir d'un model 3D fet en algun *software* de

modelatge.

Els *normal maps* són interpretats en espai tangent. En aquest espai de coordenades l'origen es situa al punt de la superfície on estem calculant la il·luminació i el vector normal es correspon amb l'eix $z = (0, 0, 1)$. Per tant, els eixos x i y són tangents a la superfície. A la imatge 22 podem veure dos espais tangents diferents per dos punts diferents.

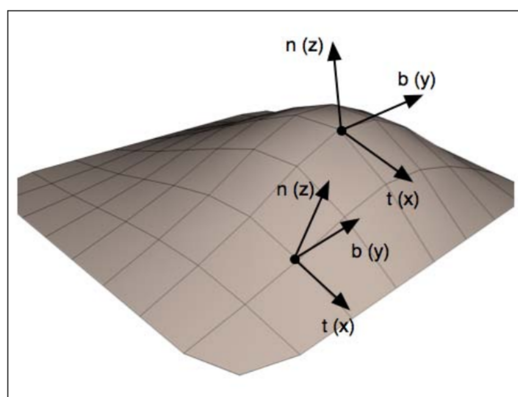

Figura 22: Exemple de dos espais tangents diferents sobre una superfície

L'avantatge que obtenim d'usar un sistema de coordenades diferent és que l'aplicació d'aquestes normals pertorbades és independent del sistema de coordenades en el que estiguem operant. Així, ens estalviem transformar el vector normal, sumar-li la normal pertorbada i re-normalitzar. Per tal que la tècnica funcioni cal avaluar el model de reflexió que haguem escollit en aquest espai de coordenades.

Per fer això, cal transformar els vectors normals usats a espai tangent. El lloc més idoni per fer aquesta tasca és el *vertex shader*. Per definir la transformació que passa de l'espai de coordenades de càmara a espai tangent necessitem 3 vectors normalitzats co-ortogonals definits, també, en coordenades de càmara que defineixin l'espai tangent. Com ja he dit, l'eix z es correspon amb el vector normal, l'eix x l'anomenem vector tangent i l'eix y vector binormal. Un punt P definit en coordenades de càmara es pot transformar a espai tangent aplicant:

$$\begin{bmatrix} S_x \\ S_y \\ S_z \end{bmatrix} = \begin{bmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{bmatrix} \cdot \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \tag{2}$$

En l'equació 2, S és el punt en espai tangent i P el punt en espai de càmara.

Per tal d'aplicar aquesta transformació al *vertex shader*, normalment el programa *OpenGL* ha de proveir al menys dos dels 3 vectors que defineixen l'espai tangent a la posició del vèrtex. Usualment es proveeix el vector normal n i el vector tangent t. El vector binormal, doncs, es pot calcular com el producte vectorial d'aquests anteriors:

$$b = n \times t \tag{3}$$

Els vectors tangents normalment s'inclouen com a dades addicionals a les estructures dels models 3D. Si aquest vector no el tenim disponible el podem aproximar derivant-los a partir de variacions de les coordenades de textura a través de la superfície (veure [11] per més informació). De tota manera, com que estem tractant amb *billboards* podem fer simplificacions com veurem més endavant.

5.4.2 Model d'il·luminació de Phong

La manera com es comporta la llum depèn del model d'il·luminació que vulguem aplicar. Per al sistema he decidit usar el model de Phong [13]. La el·lecció ve de que permet determinar el color final d'un píxel usant totes les components vistes anteriorment: ambient, difús i especular amb el coeficient de brillantor. Dir com es comporta la llum, per tant, és anàleg a dir com reacciona un material concret davant d'una llum concreta. En aquest model, es calcula el color com l'acumulació de cadascun dels 3 components.

$$A = I_a K_a \tag{4}$$

La primera part de la formula calcula la contribució que té la llum ambient sobre els objectes. Com a llum ambient considerem aquella llum que afecta a tots els punts de manera equitativa. A l'equació 4, I_a representa la intensitat de la llum ambient i K_a el coeficient de reflexió de llum ambient del material. En les següents parts de la formula aplicaré la mateixa notació. Per la part difusa, Phong utilitza el model de Lambert:

$$D = I_p K_d \cos(\theta) \tag{5}$$

Un cop més, I_p és la intensitat del punt de llum, K_d el coeficient de reflexió difusa del material i θ l'angle que formen el vector normal N i el vector que va del punt que estem tractant a la font de llum L. Usant el producte escalar de dos vectors:

$$\vec{v_1} \cdot \vec{v_2} = \|\vec{v_1}\| \|\vec{v_2}\| \cos(\theta) \tag{6}$$

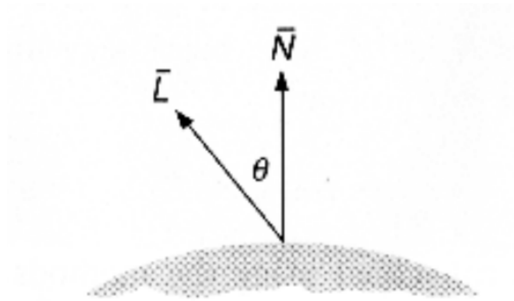

Figura 23: Model de lambert

En cas que N i L estiguin normalitzats podem reescriure l'equació d'il·luminació:

$$D = I_p K_d (N \cdot L) \tag{7}$$

L'equació 7 encara dona problemes ja que hi ha cassos en els que es poden donar oclusions (imatge 24) que cal tractar de la següent manera:

$$D = I_p K_d \; max(N \cdot L, 0) \tag{8}$$

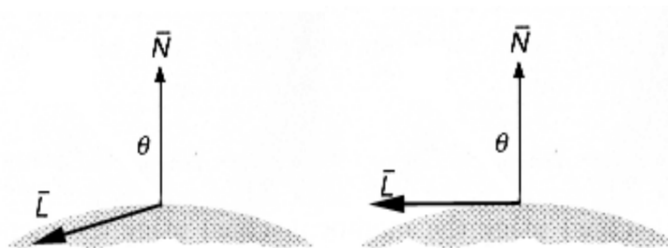

Figura 24: Exemple d'oclusió (esquerre) amb la correcció de l'equació 8 (dreta)

Per acabar d'afegir realisme cal tenir en compte que la llum no arriba amb la mateixa intensitat a tots els punts. Es per això que cal un factor d'atenuació f_{att} que varïi en funció de la distància i que modifiqui la quantitat de color difusa que cal aplicar. La caiguda de intensitat de llum obeeix la llei del quadrat invers, la seva intensitat disminueix exponecialment en funció de la distància:

$$f_{att} = \frac{1}{d_L^2} \tag{9}$$

On d_L és la distància del punt il·luminat a la font de llum. Finalment la llei de lambert completa obeeix:

$$D = f_{att} I_p K_d \ max(N \cdot L, 0) \tag{10}$$

Aquest mètode, però, en escenes renderitzades per computador ens dóna una rang de valors molt petit de llums amb les que poder treballar. Es per això que es sol usar un factor d'atenuació lineal amb una distància màxima d'atenuació.

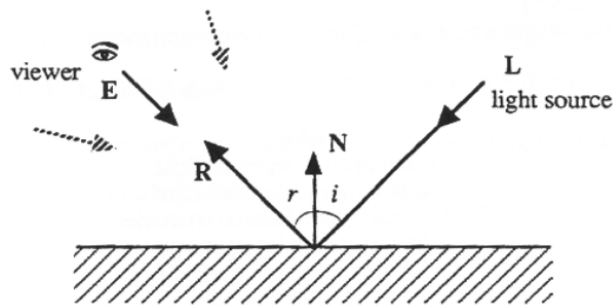

Figura 25: Comportament de la llum especular en un mirall perfecte

Ja només queda calcular la contribució de la llum especular, que és el que ens dona l'equació de Phong. Com es pot apreciar a la figura 25 el vector E és el que surt del punt d'observació al punt il·luminat i es compleix que $r = i$. El cas és que normalment no ens trobarem miralls perfectes i per tant, el comportament de la llum especular serà similar al de la figura 26 i és el que tracta l'equació de Phong.

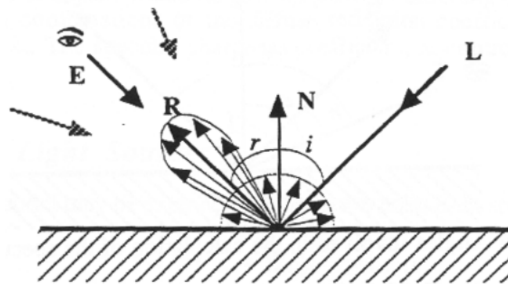

Figura 26: Comportament de la llum especular en un mirall imperfecte

Al diagrama 28 podem veure que, de manera anàloga al cas de Lambert, l'angle θ és el que forma el vector d'incidència de la llum respecte la normal

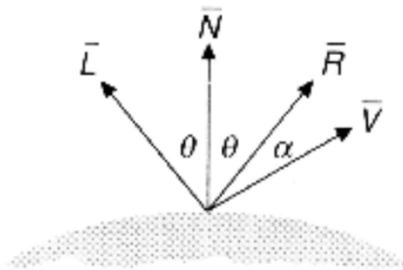

Figura 27: Model de Phong

a la superfície. V representa el vector E de l'observador canviat de sentit. L'angle α és el que formen el vector reflectit R i el vector de l'observador V. Phong va postular que la reflexió especular màxima s'assoleix quan l'angle α tendeix a 0. D'aquí va derivar que la caiguda de llum especular depèn d'aquest mateix angle, governat pel terme $\cos^n \alpha$.

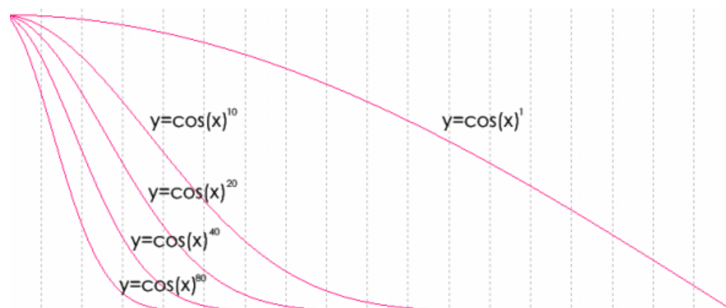

Figura 28: Caiguda de color que implica n a l'expressió $\cos^n \alpha$

$$S = f_{att} I_p K_s (R \cdot V)^n \tag{11}$$

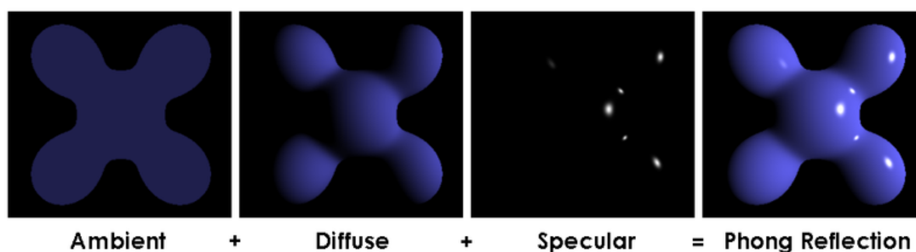

Figura 29: Exemple d'aplicació del model d'il·luminació de Phong

De tot plegat se'n deriva l'equació 11 molt similar a l'equació 10. El que varia és K_s que ara representa el coeficient especular, els vectors R i V ja comentats

i n com a coeficient de brillantor. Finalment l'equació d'il·luminació final que aplicarem a cada punt serà la suma de cada component. Els resultats de cada pas es mostren a la figura 30.

$$I = A + D + S \tag{12}$$

$$I = I_a K_a + f_{att} I_p K_d \ max(N \cdot L, 0) + f_{att} I_p K_s (R \cdot V)^n \tag{13}$$

5.4.3 Adaptació als *billboards*

Com que estem tractant amb *billboards*, com ja he comentant, es poden fer simplificacions a l'hora de calcular els vectors tangent i binormal que juntament amb la normal formen l'espai tangent. Com que els *billboards* sempre estan orientats a la càmara no cal que es passi des de l'aplicació *OpenGL* al *shader* el vector tangent. Tenim el vector normal en coordenades d'ull que és $(0, 0, 1)$. Per calcular el vector tangent podem aplicar el següent pseudocodi:

```
tangent = normalitza((tamanyParticula, 0, 0) + centreParticula)
    - centreParticula);
```

El que faig és saber quin és el vector corresponent a partir de dos punts: l'extrem dret del quadrat i el centre. Això es pot fer perquè disposem d'aquesta informació en coordenades d'ull. Per acabar només necessitem aplicar la transformació a espai tangent als vectors L i V explicats a l'apartat anterior. Finalment, al *fragment shader* s'ha d'aplicar el model d'il·luminació ja presentat.

Figura 30: Resultat de la il·lumianció en un sistema de partícules de fum

6 Fase d'actualització

Vista la fase de render només queda preocupar-se de l'evolució que segueixen les partícules des del moment en que néixen fins la seva mort i reutilització. En aquest apartar tractaré tots aquests temes sempre orientant la implementació a la GPU.

6.1 Atributs dels sistemes de partícules

En un sistema de partícules un dels paràmetres que influeix és el nombre de partícules que el sistema pot arribar a processar alhora [17]. Inicialment em plantejava la idea de tenir un nombre de partícules base que es pogués anar modificant dinàmicament en funció de l'execució sense tenir un màxim fixat. Realment es poden arribar a aconseguir resultats molt bons. Per contra s'introdueixen problemes de gestió de memòria dinàmica que el farien més ineficient.

Per solucionar aquest tema finalment he escollit l'opció de tenir un nombre de partícules màxim fixat. És el paràmetre α de la transparència el que determina si la partícula es viva i cal ser processada o no. Amb aquest mètode hi ha punts de l'execució en el que estem gastant més memòria de la necessària però tenim l'avantatge que no existeix la ineficiència de la memòria dinàmica.

Com ja hem vist, per tant, les partícules segueixen un cicle de vida en el que poden estar visibles o no. El factor clau que determina quin serà el

seu comportament durant l'execució és la inicialització dels paràmetres que defineixen la partícula. Aquests són paràmetres relacionats amb la física. Podem distingir-los entre dos tipus de cara a la implementació:

- Varibles per cada partícula.
  - Poisició inicial: vector de 3 components que representa la posició inicial del centre de la partícula al espai 3D.
  - Posició actual: vector de 3 components que representa la posició actual del centre de la partícula al espai 3D.
  - Velocitat incial: vector de 3 components que representa la velocitat inicial del centre de la partícula al espai 3D.
  - Velocitat actual: vector de 3 components que representa la velocitat actual del centre de la partícula al espai 3D.
  - Temps de naixement.
- Constants per totes les partícules.
  - Temps transcorregut: temps que ha passat des de la inicialització del sistema fins el moment.
  - Increment de temps: temps transcorregut entre el *frame* i i el *frame* $i + 1$.
  - Temps de vida: temps que ha de passar per tal que la partícula hagi de morir o tornar-se a usar.
  - Acceleració: vector de 3 components que represneta l'acceleriació de totes les partícules en un *frame* i.

Hi ha paràmetres que essencialment han de ser diferents per cada partícula per poder resoldre el problema com la posició i velocitat inicial. Si no fós d'aquesta manera no es podrien crear volums a partir de *billboards* i aquest és l'objectiu principal. La posició i velocitat actuals se'n deriven dels atributs inicials com a conseqüència dels càlculs que se'ls apliquen tal i com veurem més endavant.

Finalment, si el temps de naixement no fós obligatori fer-lo variable, totes les partícules naixerien alhora, cosa que de vegades és el que es vol, però perdríem opcions en la generació d'efectes variats. Per exemple, si volem simular fum, és estrictament necessari que les partícules sorgeixin progressivament.

Per les variables que són comunes a totes les partícules, evidentment, tant el temps transcorregut com l'increment de temps han de ser-ho. En canvi les dos variables restants, temps de vida i acceleració, no estan tan clarament ubicades. Quantes més variables diferents per cada partícula tingui el sistema, més ric serà quant a realisme. Inicialment aquesta era la meva idea però vaig topar-me amb un problema que no deixava dur-lo a terme.

La programació de GPU és un món complex com a conseqüència de la hetereogeneïtat de la que disposem al mercat. A diferència de la programació tradicional en CPU, l'arquitectura de cada tarja pot ser completament diferent fins al punt que els resultats entre dos targes no tenen perquè ser equivalents. Això és normal ja que les GPU estan pensades per fer gràfics i no és tant important que, per exemple, el color dels píxels que es vegin per pantalla sigui una mica diferent.

De la mateixa manera que els resultats dels càlculs varien també ho fan el nombre d'estructures màxim que la API gràfica suporta. Com ja hem vist als inicis d'aquest document a l'apartat *transform feedback*, per usar aquesta tècnica es necessiten VBOs. Hi ha un màxim de VBO que suporta la meva tarja gràfica per fer servir aquesta tècnica (en el meu cas es 4).

Una primera solució seria usar VBO empaquetats. Això consisteix en que en comptes de tenir un VBO per cada atribut només en tenim un en el que, mitjançant *offsets*, introduïm totes les dades. Per exemple, es poden usar les 3 primeres posicions per la posició, les 3 següents per la velocitat i la última pel temps de naixement i aquests 7 *floats* serien la informació que li entraria a cada vèrtex.

El problema rau en que en el món dels VBO no és trivial saber quin mètode, si empaquetant o no, et donarà més eficiència en temps [9]. En definitiva és un terreny en el que cal experimentar i en el meu cas la opció empaquetada era més dolenta. Cal sumar també el fet que les dades que varien per cada partícula han d'estar duplicades per tal de poder fer el *swap buffers* pels motius que ja s'han vist i això és un cost que pot arribar a ser molt gran en memòria ja que normalment estem tractant amb sistemes formats per moltes partícules.

La solució no és única però jo m'he decantat per, simplement, fer que aquest parell de variables siguin comuns a totes les partícules i mantenir els VBO separats. Els motius d'aquesta decisió són els següents:

- Que el temps de vida varïi per cada partícula no és tan crític per obtenir bons resultats ja que ajustant els altres paràmetres es poden arribar a aconseguir efectes similars.

- L'acceleració en qualsevol sistema físic és el resultat de la interacció de diverses forces amb el sistema tal i com ens diu la llei de Newton $F = m \cdot a$. Hi han maneres d'aconseguir que sembli que s'apliquen diferents acceleracions a les partícules per modificar el seu comportament. Ho veurem amb més detall al següent apartat.

6.2 Càlculs de la física a la GPU

La física ens dona les eines quantitatives per descriure els comportaments de la matèria. En un sistema de partícules normalment s'apliquen les lleis de la física per determinar quina és l'evolució del sistema en funció del temps. Es podrien distingir dos maneres en d'aplicar la física en les aplicacions de gràfics per computador:

- Simulació: es tracta d'aplicacions en els que el resultat generat ha de ser el més fidel possible a la realitat i normalment la complexitat i quantitat dels càlculs fan que a dia d'avui en molts contextos no es puguin dur a terme en temps real. Exemples de coses a calcular no trivials serien col·lisions inter-partícules, centre de masses, etc. Normalment es situen en l'àmbit científic i d'investigació.

- Aproximació: volen cobrir el món dels gràfics en temps real, per tant, a diferència dels anteriors els càlculs que efectua són aproximacions que no pretenen imitar amb absoluta fidelitat el que passaria en un entorn real. El que es busca, per tant, és que els càlculs siguin eficients i ens permetin tenir un *framerate* acceptable. Un clar exemple serien els videojocs.

En aquest TFG estem ubicats en el segon tipus d'aplicacions. L'objectiu que cal assolir és que, utilitzant els atributs estudiats al apartat anterior, fer els càlculs necessaris per saber com hem d'actualitzar les partícules usant la GPU per explotar el seu paral·lelisme. Per fer-ho *OpenGL 4.x* ens ofereix *transform feedback* tal i com hem vist anteriorment.

6.2.1 Equacions físiques

Del món de la cinemàtica i la dinàmica coneixem una sèries de lleis que ens permeten saber com actualitzar un cos en moviment en funció del temps [18].

Realment l'únic que ens influeix és saber quina posició té cada partícula a cada instant de temps. L'equació que ens ho resol seria la següent:

$$P = P_0 + vt + \frac{1}{2} \cdot at^2 \tag{14}$$

On P és la posició, P_0 la posició inicial, v la velocitat inicial, a l'acceleració i t el temps. Aquesta manera de calcular la posició és totalment correcte però introdueix limitacions. Bàsicament ens força a que les variables que depenen del temps (velocitat i acceleració) siguin constants durant tot el moviment fet que ens fa perdre llibertats a l'hora de definir comportaments.

Es podria usar l'equació 14 en un sistema on per exemple l'acceleració pogués canviar al llarg del temps però implicaria haver de detectar al *vertex shader* cada canvi d'aquesta variable i actualitzar les posicions i velocitats inicials de totes les partícules. Hi ha un altre conjunt d'equacions que ens soluciona aquest problema permetent-nos simular les variables importants pas a pas:

$$P_{i+1} = P_i + \vec{v} \cdot \Delta t \tag{15}$$

$$\vec{v}_{i+1} = \vec{v}_i + \vec{a} \cdot \Delta t \tag{16}$$

Els subíndex $i + 1$ i i volen dir *frame* següent i *frame* actual. L'avantatge que tenim és que si per exemple l'acceleració canvia en un *frame* determinat la manera com calculem cada actualització ens dona un resultat correcte. Per contra cal que ens guardem més variables d'estat per cada partícula. He resolt el problema guardant només la velocitat tal i com explica l'apartat anterior.

La justificació, a part dels temes relacionats amb la memòria ja explicats, es que segons en quins cassos de sistemes, com poden ser fum i foc, podem fer una aproximació que simula una acceleració diferent per cada partícula en funció de la seva distància al emissor tal i com es demostra a [14].

La idea és molt simple. Es tracta de aplicar un percentatge de l'acceleració en funció d'una condició, en el cas dels sistemes descrits funciona bé la distància al emissor de la partícula, és a dir, la seva posició inicial. Haver de tenir guardada la posició inicial no és un afegit ja que la necessitem per re-utilitzar la partícula un cop es morta com veurem al següent apartat.

El que necessitem per fer això és separar els valors que volem que siguin constants de l'acceleració, com podria ser la gravetat, dels valor variables

amb el temps, com per exemple l'acceleració resultant de la força del vent aplicada en un moment donat.

$$\vec{a}_t = \vec{a}_c + \vec{a}_v(t) \cdot f(x) \tag{17}$$

L'acceleració total $\vec{a}_t$ es calcularia com la suma de l'acceleració constant $\vec{a}_c$ i l'acceleració variable $\vec{a}_v(t)$ modificada per una funció que es mou en l'espai d'imatges $[0, 1]$ i que varia en funció d'una llista de paràmetres x que en aquest cas pot ser la distància al emissor i una distància màxima.

A partir d'aquí, en funció dels resultats que es vulguin obtenir, existeixen infinites funcions que ens donaran un comportament diferent. Per resoldre el cas del foc i el fum sota la influència de la força del vent he aplicat una funció lineal que, quan es troba a una distància màxima parametritzada del emissor aplica la tota l'acceleració (és a dir, que la funció retorna 1) i quan la partícula està aprop del emissor retorna un valor que tendeix a 0 o bé 0 si és just a la posició d'emissió.

Per veure exemples de paràmetres que li poden entrar a la funció es podria considerar la opció de parssar-li la edat de la partícula, és a dir el temps que porta viva menys el temps en que ha nascut, de manera que quan sigui 0 s'apliqui només l'acceleració constant i a mesura que li arribi la mort s'apliqui gairebé el total de l'acceleració variable. En definitiva, veiem que podem modificar els comportaments d'una manera aproximada, senzilla i controlada.

6.2.2 Càlcul de col·lisions

Un altre dels aspectes que influeix en el càlcul de la posició de cada partícula en la seva actualització és la detecció de col·lisions amb possibles elements de l'escena. Aquesta branca per si sola és un món molt complex que per si sol podria donar tema suficient com per fer un tfg només tractant aquest tema. Per tal de poder fer una visualització realista és important veure com interaccionen els elements rígids de l'escena entre si.

Com ja he comentat, per fer-ho bé caldria comprovar les col·lisions entre tots el elements de l'escena que considerem rígids i les partícules no són una excepció. Per simplificar, en aquest sistema he fet que les només es considerin les interaccions amb elements de l'escena. A més a més ho aplicaré només a esferes d'un radi determinat i plans infinits sobre el pla xz estàtics.

El motiu de tractar només *colliders* tan simples és que, tot i que per CPU ja hi han algorismes molt optimitzats per dur a terme aquesta tasca, per fer-ho en

GPU no és trivial i he volgut centrar més el meu treball en temes relacionats amb la visualització. Per detectar la col·lisió hem de fer la comprovació de manera diferent pel pla i per l'esfera. Pel cas de l'esfera caldria aplicar el següent pseudocodi:

```
novaPosicio = posicioActual + velocitat * deltaTime
normalSuperficie = novaPosicio - centreEsfera
distancia = modul(normalSuperficie)
if (distancia < radiEsfera) {
    calculaRebot()
}
```

Calculem la nova posició que tindrà la partícula en el *frame* de la manera que ja hem vist al apartat anterior. Un cop la tenim podem calcular la distància d'aquesta nova posició al centre de l'esfera, que és conegut i si aquesta distància es menor que el radi significarà que hi ha col·lisió i hem de calcular el rebot. De la mateixa manera el pseudocodi per la detecció amb el pla seria:

```
novaPosicio = posicioActual + velocitat * deltaTime
if ((novaPosicio < alturaPla AND posicioActual > alturaPla) OR
    (novaPosicio > alturaPla AND posicioActual < alturaPla)) {
    calculaRebot()
}
```

Com que només comprovem col·lisions amb plans als eixos xz només cal veure si l'alçada del pla està continguda entre la posició actual i la nova posició. Finalment pel càlcul del rebot he aplicat la fórmula del vector reflectit:

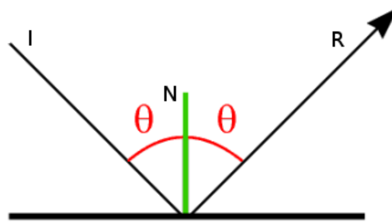

Figura 31: Comportament del vector refelctit respecte un vector incident en una superfíce

$$\vec{R} = \vec{I} - 2(\vec{N} \cdot \vec{I}) \cdot \vec{N} \tag{18}$$

On $\vec{R}$ és el vector reflectit en detectar una col·lisió, $\vec{I}$ és el vector incident

corresponent a la velocitat i $\vec{N}$ és el vector normal a la superfície de col·lisió normalitzat. Per controlar la elasticitat del xoc he afegit un coeficient de elasticitat que va entre 0 (xoc totalment inelàstic) i 1 (xoc totalment elàstic) de manera que es varia la magnitud de la velocitat resultant.

Al motor, podem passar-li un pla i una esfera en forma de variable *uniform* de *GLSL* de manera que totes les partícules tinguin els mateixos *colliders* associats. Cal que aquests es mantinguin estàtics ja que si els fem variar la seva posició seguint alguna equació física indicaria que tenen associada una velocitat fet que faria els càlculs anteriors erronis.

6.3 Inicialització de les partícules a la GPU

Tal i com està plantejat el motor, cada sistema de partícules sempre està format per un nombre fixe de partícules que poden estar vives (visibles) o no. El plantejament que cal fer ara és, per sistemes cíclics en els que es vol que les partícules puguin ser re-utilitzades, quina és la manera de re-inicialitzar-les.

La primera tècnica i més simple de totes consisteix en tenir guardada la posició i velocitat inicials on la partícula ha nascut i un cop la seva edat supera el temps de vida tornar a assignar els mateixos valors inicials. Evidentment aquesta manera funciona però el resultat és que el sistema sempre té el mateix comportament en cada cicle en cas que no apliquem acceleracions variables. Per tal de fer-ho més variat caldria assignar valors nous que respectin les condicions inicials.

Per tal de tenir sistemes que no siguin repetitius amb el temps la solució està en re-inicialitzar les partícules amb valors aleatoris que entrin en el rang de valors definits en la inicialització al client de *OpenGL*. Com que aquest càlcul també té un paral·lelisme massiu un cop més voldrem aprofitar el *hardware* de la GPU. El problema que existeix és que hi ha molt poques GPUs al mercat que ens ofereixin una unitat de nombres aleatoris per la qual cosa no és un estàndard a *GLSL*.

6.3.1 Funcions de soroll

La solució coneguda existent es basa en aplicar funcions de soroll, funcions que reben un cert nombre de paràmetres i ens retornen un valor pseudoaleatori. Aquestes funcions van ser introduïdes per Perlin al 1985. Les característiques d'aquesta funció són les següents:

- No existeix una correlació entre valors d'entrada a la funció distants.

- Es repetitiva i determinista. Els mateixos valors d'entrada produeixen el mateix resultat.

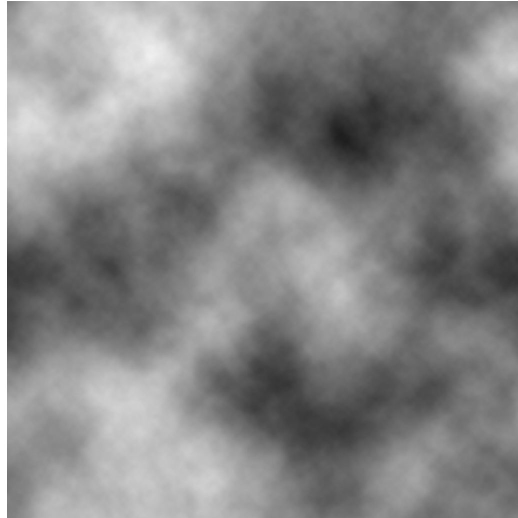

Figura 32: Exemple de textura generada amb Perlin *noise*

El problema que té el Perlin *noise* i les seves millores i modificacions posteriors és que són funcions contínues per valors d'entrada propers i aquesta característica, tot i que els valors que ens retorna la funció són pseudoaleatoris, no és satisfactòria ja que el que volem es trencar tota correlació possible.

6.3.2 *Pseudo random number generators* (PRNG)

Un PRNG, tal i com s'explica a [15] consisteix en un tipus de funció de soroll que pretén, també, donar nombres que semblin aleatoris. Les seves característiques són similars al que he comentat anteriorment però a principal diferència és que s'intenta trencar amb la continuïtat dels valors que retorna la funció per valors d'entrada propers. Bàsicament es tracta de definir una funció de *hash* 1-D de la manera següent:

$$hash(x) = x^2 modM \tag{19}$$

La variable x es correspon amb el paràmetre d'entrada de una dimensió i M és un número primer. Com a M utilitzo el número 61 ja que al *paper* explica que dóna bastants bons resultats a la pràctica. Amb aquesta funció tal qual el que tenim és que per un paràmetre determinat obtenim un valor pseudoaleatori de sortida. En el cas que estic tractant, però, necessito inicialitzar dos vectors de 3 components. El que es pot fer és extendre aquesta funció

1-D a N-D mitjançant la següent definició recursiva:

```
Si N == 1 -> hashN(v) = hash(x1)
Sino -> hashN(v) = hash(hashN(x1, x2, ..., xN-1}) + x_N)
```

En la definició, $v = (x_1, x_2, ..., x_N)$ és un vector de tamany N. Aquesta nova definició donats N valors d'entrada retorna un valor entre 0 i M. Aquest valor valor es pot normalitzar al interval $[0, 1)$ i reescalar-lo posteriorment al interval $[min, max]$ definit des de el client per la primera inicialització.

Com que la funció només retorna un valor i el que necessitem són 3 valors tant per la nova posició com per la nova velocitat inicials a la implementació final executo 3 cops la funció *hashN* amb 4 paràmetres. Si considerem la inicialització de la posició, els 3 primers paràmetres es corresponen amb la posició inicial.

Donat que aquesta funció sempre retorna la mateixa sortida per una entrada donada afegeixo un quart paràmetre variable en cada inicialització. El candidat perfecte és el temps transcorregut. Amb això obtinc el component x de la nova posició. Per determinar y i z repeteixo el mateix procés permutat els components de la posició inicial. És a dir, en comptes de passar com a paràmetre l'ordre x, y i z passo y, x, z i z, y, x per exemple. Pel cas de la velocitat el procés és totalment anàleg al de la posició.

Part IV
Resultats

Un dels objectius del projecte era comprovar el *speed up* que es pot aconseguir usant la GPU per computar els càlculs de la física del sistema de partícules. En les següents seccions en faré un anàlisi al respecte.

7 Proves de rendiment

La mesura de rendiment que he escollit per posar a prova el sistema és el nombre de partícules màxim que es poden executar a 60fps (*frames* per segon). El motiu és que la GPU realment no pot anar a més de 60fps ja que es correspon amb els 60Hz del refresc de la pantalla per tant tot i que s'enviés més informació en un segon només es podrien mostrar com a molt 60 imatges cada segon.

El programa executat el que fa és emetre un sistema de partícules petites on totes les partícules comencen en temps 0. Cada partícula equival només a un punt de un tamany molt petit per tal d'intentar simular el rendiment màxim assolible. Per tal de comprovar únicament quina és la potència de càlcul he executat també el mateix programa sense el render per poder fer una comparativa.

En resum he fet 4 execucions: CPU amb render, CPU sense render, GPU amb render i GPU sense render. Per entendre una mica el perquè dels resultats obtinguts a la prova cal fer un petit incís del *hardware* usat. El temps sense actualització no té sentit tractar-lo ja que en amdós cassos serà equivalent perquè es fa amb la GPU i el temps seqüencial de l'aplicació és gairebé negligible.

7.1 *Hardware* utilitzat

L'equip amb el qual he desenvolupat el projecte i he fet proves, com ja s'ha vist al apartat de pressupost, és un ordinador portàtil *ASUS k55v*. Els components en els que m'he centrat són en la CPU i la GPU que són les unitats de càlcul usades en ambdós cassos.

Com es pot veure a la taula 7.1 apareixen les especificacions de la CPU i a la taula 7.1, les de la GPU. Les principal diferència a *grosso modo* és que la

Número de processador	i7-3610QM
Nuclis	4
Threads	8
Velocitat de rellotge	2.3GHz
Freqüència turbo màxima	3.3GHz
Intel smart cache L3	6MB
Canals amb memòria	2
Ample de banda màxim amb memòria	25.6 GB/s

Taula 1: Especificacions de la CPU utilitzada

Model	nVIDIA gt 630M
Arquitectura	Fermi
Nuclis CUDA	96
Rellotge del nucli	672MHz
Rellotge de *shaders*	1344MHz
Rellotge de memòria	900MHz
Ample del bus amb memòria	128-bit DDR3/GDDR5
Ample de banda màxim amb memòria	32.0
Memòria	2GB
DirectX	11
OpenGL	4.1
Suport del bus	PCI Express 2.0

Taula 2: Especificacions de la GPU utilitzada

CPU pocs nuclis comparats amb la GPU però la seva freqüència és molt més alta.

7.2 *Speed up*

A la taula 7.2 es mostren els resultats del rendiment que ens ofereix el sistema en el context explicat. A priori m'esperava que els resultats fossin molt més elevats ja que les GPU en un cas general donen millors resultats que una CPU en un problema massivament paral·lel. La realitat, però, és que el *speed up* és de 6,54x en el cas que no hi ha render i 2x en el cas que si de l'execució de GPU respecte la de CPU.
Com es pot apreciar, el render és clarament el coll d'ampolla en l'execució de la versió de GPU. El motiu és que es generen molts més fragments que no pas vèrtexs i això implica que si el *fragment shader* és complex, tal i com és el cas ja que s'apliquen diverses tècniques al *fragment shader*. En canvi, a la

	CPU	**GPU**
Sense render	320.000	2.092.000
Complet	315.000	615.000

Taula 3: Resultats de rendiment

versió de CPU no és tant important donada la poca diferència de partícules que es poden processar amb o sense render.

Aquest fet fa que sigui molt difícil determinar quin és el cost real del render ja que depèn totalment de la grandària de les partícules. En definitiva el que s'obté és que amb la GPU usada es pot arribar a executar un sistema de partícules gairebé ideal a una velocitat de $36.9 \cdot 10^6 particules/s$.

El motiu d'aquests rendiments tan pobres es deu a que la característica més important d'una GPU per executar aquest tipus d'algorismes és l'ample de banda que té amb la memòria del dispositiu. Si mirem un cop més les taules 7.1 i 7.1 es veu que l'ample de banda que té la CPU es de 25.6GB/s i la GPU 32.0GB/s. Això és molt determinant ja que s'ha de moure una gran quantitat de dades i, probablement, els nuclis de la GPU es passin més temps esperant dades que no pas fent càlculs.

$$T_{particula} = 8floats \cdot 2 = 128bytes \tag{20}$$

L'equació de dalt mostra quin és el tamany en *bytes* d'una partícula. Són bàsicament 8 *floats* que es corresponen amb els atributs del sistema. Cada *float* ocupa 8*bytes* en una arquitectura x64 com la del meu equip. Finalment es multiplica per dos perquè les dades han d'estar replicades en 2 *buffers*.

$$A = NumParticules \cdot T_{particula} = 36.9 \cdot 10^6 particules/s \cdot T_{particula} = 4,732GB/s \tag{21}$$

Per saber l'ample de banda A complet que necessitarà tot el sistema només cal multiplicar el tamany d'una partícula per la velocitat de partícules que es mouen per segon. Al cas límit vist anteriorment l'ample de banda necessari són 4,732GB/s, que ja es veu que és bastant considerable si tenim en compte que l'ample de banda de les especificacions de la GPU és teòric i no assolible a la pràctica. Cal valorar que el problema és totalment escalable i augmentant la potència de la GPU obtindríem millors resultats.

8 Conclusions

Un cop acabat el projecte cal fer una avaluació objectiva per comprovar si s'han assolit correctament els objectius i quins aspectes es podrien millorar d'alguna manera.

8.1 Assoliment dels objectius

Per recordar-ho, l'objectiu principal era desenvolupar un motor de sistemes de partícules amb *OpenGL* i usant la GPU per fer càlculs de físiques així com provar diferents tècniques de visualització associats a aquests sistemes. El problema que té aquest objecitu és que és molt ampli i potser caldria haver-lo acotat una mica més.

Quant a la part de visualistazió val a dir que cadascuna de les tècniques s'ha implementat correctament i amb l'èxit visual que podia assolir. Cal tenir en compte que en alguns cassos el resultat és molt depenent de la part artística (textures) que s'empri. En el meu cas no disposava d'una font d'art suficientment adaptada a les meves necessitats.

Per la part purament de càlcul, crec que ha quedat demostrat que es pot utilitzar una API gràfica com és la d'*OpenGL 4.0* amb *GLSL 4.0* per la part de GPU per explotar el paral·llisme d'aquests dispositius. Cal notar que no és una API pensada per càlculs de propòsit general sinó per gràfics per computador.

8.2 Treball futur i millores

El món dels sistemes de partícules és molt ampli, fins al punt que hi podrien haver diversos TFGs en paral·lel que finalment posant-los en conjunt millorессin un producte comú. Jo en aquest subapartat em centraré només en els temes que jo he tractat.

La part del disseny d'un motor que utilitzi la GPU és una tasca complexa de dur a terme perquè crea moltes dependències amb el codi que s'executara al dispositiu gràfic. Tal i com està desenvolupat el producte acutalment, cada modificació que afecti al comportament implica crear una classe *C++* i el codi *GLSL* associat. Això implica que es necessitin coneixements del llenguatge i les GPUs en general. També cada modificació implica que s'augmenti la

complexitat dels *shaders* i com a conseqüència que baixi el rendiment.

Es per això que considero molt interessant de cara al futur la implementació d'un llenguatge de *scripting* que permeti definir comportaments i propietats d'un sistema de partícules i que generi el codi necessari, sobretot *GLSL*, que s'acabi executant al programa. La possibilitats de sistemes de partícules que es podríen generar creixerien bastant.

Quant als aspectes visuals un aspecte important que es podria tractar és el de considerar les partícules esfèriques, explicat a [16]. No he entrat en el tema perquè una millora de la tècnica *soft particles* que he aplicat. També donar suport a altres efectes com la pluja o la sang però en definitiva altres sistemes més específics.

Part V
Gestió del projecte

Aquest apartat està dedicat a la metodologia i planificació aplicades per desenvolupar el projecte així com els seus costos associats.

9 Metodologia de desenvolupament

A continuació parlaré d'aquells aspectes que modelen la manera de treballar que he seguit i que han permès desenvolupar un producte de qualitat que satisfà al client complint els terminis.

9.1 Tasques i procediments

Per desenvolupar el projecte, ha calgut dur a terme una sèrie de tasques les quals en el meu cas són majoritàriament independents les unes de les altres, excepte la primera que permet posar en marxa la resta:

- Aprenentatge autònom de les característiques de OpenGL 4 i glsl 4.
- Aprenentatge autònom sobre tècniques de desenvolupament d'APIs en C++.
- Implementació de la base del motor usant els coneixements adquirits.
- *Testing* del la base del motor.
  - Lectura d'articles relacionats amb *geometry shader billboards*
  - Implementació de la tècnica.
  - *Testing* d'aquesta part.

  - Lectura d'articles relacionats amb *soft particles.*
  - Implementació de la tècnica.
  - *Testing* d'aquesta part.

  - Lectura d'articles relacionats amb animació de partícules mitjançant sprites.
  - Implementació de la tècnica.
  - *Testing* d'aquesta part.

  - Lectura d'articles relacionats amb generació de nombres aleatoris a la GPU.
  - Implementació de la tècnica.
  - *Testing* de les tècniques anteriors simulant fum i foc.

  - Lectura d'articles relacionats amb *normal mapped billboards.*
  - Implementació de la tècnica
  - *Testing* d'aquesta part.

  - Adició de col·lisions amb plans i esferes.
  - *Testing* d'aquesta part.

  - Implementació d'una versió de CPU del motor.
  - Proves de rendiment i comparatives associades.

- Assoliment de la fase beta del software.
- *Testing* de la beta.
- Finalització del motor.

9.2 Eines de seguiment i validació

Tota la planificació està estructurada i refelctida a un diagrama de gantt on apareix la planificació amb tots els canvis corresponents des de l'inici amb la durada real de les tasques. Cal dir que inicialmente la jornada de treball constava de 8h cosa que s'ha hagut de reduïr a 4h per temes laborals fora de l'àmbit acadèmic.

El seguiment del projecte l'ha supervisat el tutor amb el que hi han hagut una sèrie de reunions, incloent la reunió de la fita de seguiment, per tal d'assegurar que cadascuna de les etapes establertes s'ha assoleit correctament.

A continuació mostraré quin ha estat el criteri de validació que estableix quins són els requisits concrets que cal assolir per tal de completar cadascuna de les iteracions descrites anteriorment i, per suposat, el projecte sencer. Són els següents:

- Implementació de la base del motor

  - Disseny software adient considerant les tecnologies que s'usaran.

- API mínima que permeti crear i usar un sistema de partícules senzill que utilitzi la tècnica de *transform feedback* i *geometry shader billboards*.

- Implementació de *soft particles*.
  - Adició de sistemes de partícules més concrets relacionats amb la tècnica, com per exemple fum.
  - Funcionament efectiu de la tècnica.
  - Adició al disseny software de la nova característica.

- Implementació de partícules animades.
  - Adició de sistemes de partícules més concrets relacionats amb la tècnica, com per exemple foc.
  - Permetre que es puguin introduïr un nombre qualsevol de *sprites* al sistema.
  - Funcionament efectiu de la tècnica.
  - Adició al disseny software de la nova característica.

- Implementació de un generador de nombres aleatoris a la GPU.
  - Utilització eficaç de la tècnica per tal de tenir sistemes de partícules més variables en el temps en 3 dimensions.

- Implementació de *normal mapped billboards*.
  - Adició de sistemes de partícules amb partícules visualment més complexes a partir de primitives senzilles, com per exemple quads.
  - Funcionament efectiu de la tècnica.
  - Adició al disseny software de la nova característica.

- Addició de col·lisions de les partícules amb plans i esferes.
  - Pemetre tenir com a mínim un pla xz i una esfera estàtics amb els quals les partícules col·lisionin.
  - Que la interacció amb aquestes desencadeni un rebot realista acord amb la física.

- Proves de rendiment i comparativa entre CPU i GPU.

  - Fer un joc de proves que permeti determinar a grans trets quin dels dos sistemes es millor amb un resultat quantitatiu.

- Assoliment de la fase beta del motor.

  - Producte final preparat per ser testejat amb totes les característiques implementades.
    * Capacitat de generar sistemes de partícules predefinits: foc, fum, espurnes, etc.
    * Capacitat de generar sistemes de partícules genèrics.

10 Planificació

He fet la planificació en un diagrama de Gantt utilitzant *gantter* tal i com mostra la figura 10. En aquest apartat faré una descripció en detall de que consisteix cada tasca. He agrupat les tasques en 4 fases que conformen un objectiu comú molt específic per facilitar el control i modificacions del flux de desenvolupament del projecte. A cada tasca he assignat els recursos corresponents. Totes tenen un recurs comú que es la meva mà d'obra i la utilització de PC portàtil que faré servir. Algunes en concret tenen la documentació i el software necessàris per dur-les a terme.

10.1 Fase incial

Aquesta és la fase de posada en marxa del projecte. La podem identificar al diagrama com les tasques que tenen el color vermell. La idea que hi ha darrere és aconseguir una base sòlida que en un futur pugui créixer i millorar per assolir l'objectiu final. Les tasques són les següents:

- Instal·lació i configuració de l'entorn de treball: En aquest apartat caldrà posar a punt les eines que utilitzaré al llarg del desenvolupament. Es tracten del sistema operatiu linux i *qtCreator* com a IDE. Aquesta tasca també ha implicat la instal·lació de *bumblebee*, un projecte *open source* que permet executar les aplicacions amb la GPU dedicada a linux. Ha estat necessari ja que l'equip que he utilitzat té una GPU integrada que no és vàlida per desenvolupar.

- Aprenentatge de OpenGL 4: Aquesta tasca té la missió d'aprendre els aspectes més importants que té la API de *OpenGL 4* per al desenvolupament. En concret, el llenguatge *glsl 4* pel desenvolupament de

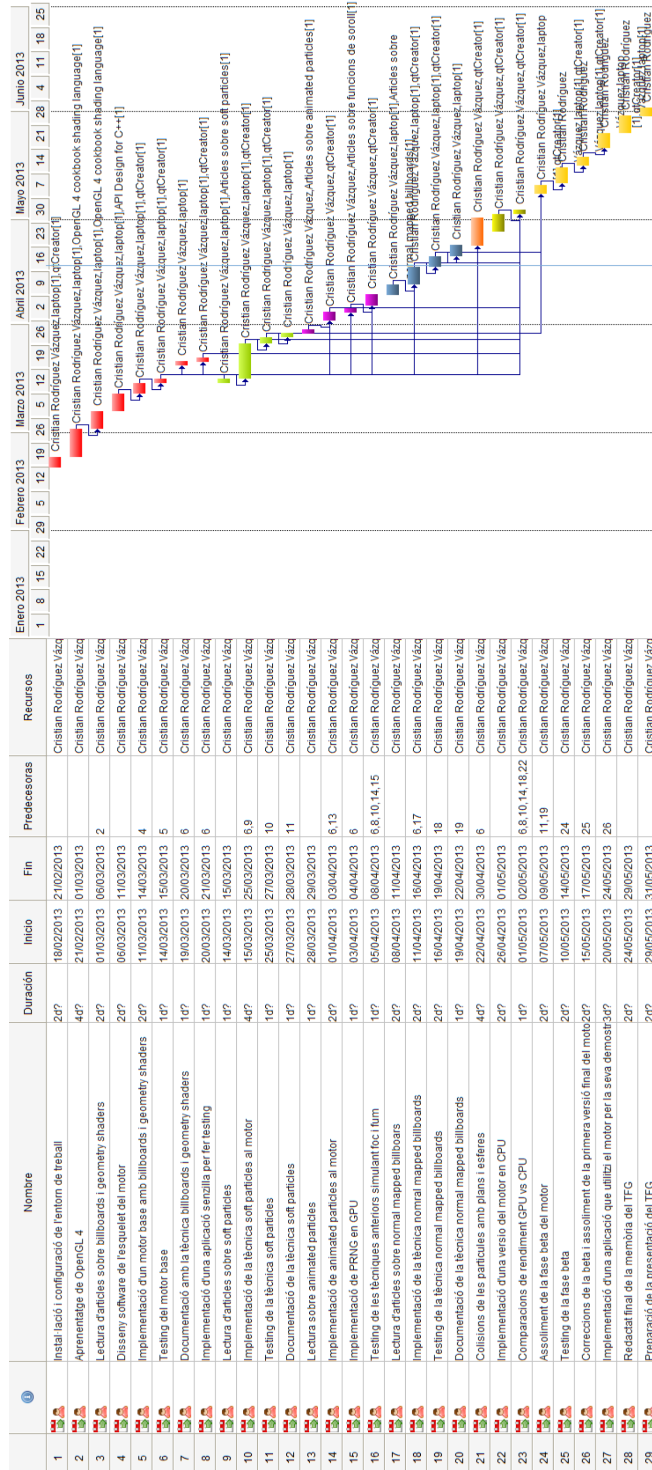

Figura 33: Diagrama de Gantt

shaders. La informació serà extreta del *OpenGL 4 cookbook*, que és el recurs associat.

- Lectura d'articles sobre *geometry shader billboards*: L'aspecte principal d'aquesta tasca és entendre els aspectes més la tècnica base que permet enviar punts a la GPU i generar els *billboards* dins del *pipeline* gràfic.
- Disseny software de l'esquelet del motor: Es tracta de raonar quina seria l'arquitectura del software adequada per tal de tenir una bona API com a resultat. Requereix un llibre que he escollit perquè em permet aprendre bones estratègies pel meu cas que es diu *API Design for C++*.
- Implementació d'un motor base amb billboard instancing: Aplicant els coneixements i tècniques adquirits caldrà implementar el disseny anterior. Es important que aquesta tasca es faci en el temps estimat perquè si no la resta s'endarreria massa. És possible que durant la implementació es trobin errors en el disseny software i calgui tornar a revisar la tasca anterior.
- Testing del motor base: Aquesta part inclou un disseny d'un joc de proves per comprovar que tot funciona correctament.
- Documentació del motor base amb la tècnica billboard instancing: En aquesta tasca ha calgut redactar la part de la memòria que expliqui els detalls d'implementació.
- Implementació d'una aplicació senzilla per fer testing dels diferents apartats: La idea es construir un esquelet que permeti provar les noves característiques que se li vagin afegint al motor.

Les fases posteriors a la inicial estan destinades a introduir totes les millores possibles dins del producte.

10.2 Tècnica de Soft particles

En aquesta fase el projecte he afegit al motor base producte de la fase anterior una tècnica de visualització que aconsegueix sistemes de partícules més realistes. Les tasques que engloba són les següents:

- Lectura d'articles sobre soft particles: De manera anàloga al cas anterior cal documentar-se abans de fer la implementació. S'inclou com a recurs tota la documentació associada a aquesta tècnica.

- Implementació de la tècnica soft particles al motor: Un cop estudiada i entesa la tècnica cal afegir-la al projecte. Un cop més, aquesta part és una de les més delicades ja que és una de les que determina que el projecte es completi o no. En cas de tenir molts problemes caldria parlar amb el tutor per replanificar aquesta part d'alguna altra manera.

- Testing de la tècnica soft particles: Aquesta tasca consisteix en crear diferents jocs de proves que demostrin que s'ha implementat correctament la tècnica. Aquests jocs de proves s'integraran amb l'aplicació de testing de la que he parlat abans.

- Documentació de la tècnica soft particles: En aquest punt redactaré a la memòria tot allò que sigui rellevant sobre aquesta tècnica i el cas particular de la implementació que hagi fet.

Si qualsevol dels punts es torna problemàtic tinc llibertat per replantejar la planificació i passar a la fase següent ja que són dos tasques independents que es poden paral•lelitzar.

10.3 Tècnica de partícules animades

Com ja s'ha vist al apartat corresponent, aquesta tècnica permet donar dinamisme a cada partícula per separat de manera que en conjunt es puguin aconseguir efectes més variats. Aquesta característica ha implicat les següents tasques:

- Lectura i documentació sobre la tècnica: fase en la que ha calgut fer una investigació prèvia de com implementar la tècnica llegint articles que en parlin al respecte. Com a recursos associats, doncs, es troba tot aquest tipus de documentació.

- Implementació i testing de la tècnica: en aquesta fase s'han aplicat tots els coneixements adquirits en l'apartat anterior per afegir-los al motor i fer-ne proves al respecte.

10.4 Tècnica de PRNG *Pseudo Random Number Generators* a la GPU

En aquests apartat hem implementat un PRNG per tal de re-inicialitzar les partícules amb valors pseudo-aleatoris a la GPU amb la idea de mantenir aquesta política de fer tots els càlculs independents de l'aplicació client. Les tasques associades són:

- Lectura i documentació sobre PRNG: en aquest punt ha calgut cercar informació sobre aquests tipus de generadors de nombres i s'associen els recursos de la documentació trobada.
- Implementació i testing de PRNG a la GPU: aplicant els coneixements adquirits al apartat anterior s'ha implementat aquesta característica al sistema.

10.5 Tècnica de normal mapped billboards

Dins aquest apartat de tècniques implementaré la tècnica de normal mapped billboards per aconseguir una il•luminació més realista. El procediment a seguir és molt similar al de la fase anterior:

- Lectura d'articles sobre normal mapped billboars: Consisteix en una recerca sobre la tècnica amb la finalitat de poder entendre-la per la seva posterior implementació. Tenim com a recursos, doncs, tots els documents trobats al respecte.
- Implementació de la tècnica nomral mapped billboards: Un cop completada la tasca anterior, s'ampliarà el motor afegint aquesta funcionalitat. Si cal fer canvis en el disseny software deguts a algun error es revisarà l'esquelet.
- Testing de la tècnica normal mapped billboards: Utilitzant, com sempre, l'aplicació per fer testing es comprovarà que s'han implementat correctament les característiques.
- Documentació de la tècnica normal mapped billboards: Per completar aquesta etapa caldrà redactar a la memòria les explicacions corresponents al treball realitzat .

10.6 Colisions externes de les partícules

Un cop acabada la il·luminació que ja està implementada i planificada des del document inicial hem decidit que seria interessant implementar col·lisions de les partícules amb plans i esferes. Per tal de dur a terme aquest punt cal seguir els següents passos:

- Repàs de la física que hi intervé: cal tenir en compte les formules físiques que s'han d'aplicar al sistema per tal de dur a terme la posterior implementació. Com a recursos associats, per tant, tenim tota la documentació que expliqui aquesta temàtica.

- Implementació i testing de les col·lisions: aplicant, un cop més, la temàtica del apartat anterior consistirà en afegir aquesta funcionalitat al motor.

10.7 Proves de rendiment entre una versió de CPU i la versió de GPU

Finalment hem afegit una petita part que faci un *profiling* de quin és el guany que s'obté al implementar la versió de GPU respecte un software tradicional de CPU ja que creiem que es important demostrar els avantatges que ens dona aplicar aquestes tècniques.

- Implementació de la versió de CPU: generar una versió de CPU molt senzilla que ens permeti realitzar les mateixes tasques que la versió anterior de GPU.
- Comparacions de rendiment: en aquest punt, amb la versió desenvolupada en la tasca anterior, caldrà testejar quin és el guany de càrrega que suporta el motor basat en GPU respecte el basat en CPU.

10.8 Fase final

Arribats al punt en que s'han implementat les dos tècniques anteriors i més ampliacions, si s'escau, és el moment de finalitzar el desenvolupament del projecte. Tenim els següents apartats:

- Assoliment de la fase beta del motor: L'objectiu es obtenir una primera versió completa del software que ho tingui tot.
- Testing de la fase beta: Aquí provaré que tot en conjunt funciona a la perfecció i tractaré de detectar els diferents bugs que puguin aparèixer.
- Correccions de la beta i assoliment de la primera versió final del motor: Detectats els errors en la fase de testing, els arreglaré per tal de tenir la primera versió definitiva del motor.
- Implementació d'una aplicació que utilitzi el motor per la seva demostració: De cara la presentació final és important tenir alguna cosa que ensenyar i és el que intentaré en aquesta fase.
- Redactat final de la memòria del TFG: Acabaré de revisar i redactar el que calgui de la memòria del TFG.

- Preparació de la presentació del TFG: Prepararé el conjunt de transparències i fare els entrenaments que convinguin per fer la presentació final del TFG.

11 Pressupost

A continuació mostraré un resum on es vegi quin es el preu estimat que costaria desenvolupar el projecte associat a la planificació feta anteriorment en cas de que hagués d'estar finançat per algú. Per fer-ho veurem quines són cadascuna de les despeses implicades.

11.1 Recursos humans

En la taula següent faig un recull de quins són els rols implicats en tot el projecte amb els seus sous corresponents:

Rol	Salari (€/h)	Nombre d'hores (h)	Cost total (€)
Cap de projecte	21	496	10416
Analista/Dissenyador	12,50	176	2200
Programador	18	448	8064
Tester	13	64	832
			21512

Taula 4: taula amb els costos de recursos humans

Tots els costos d'aquesta taula són fixes i directes. Hi ha un salari diferent per cada rol que actua al desenvolupament del projecte. El cap de projecte treballa el conjunt d'hores total que dura el projecte, ja que sempre ha de dur un control exhaustiu del seu desenvolupament i tornar a planificar en cas que sigui necessari.

Els altres rols, només es necessiten en tasques més específiques del seu àmbit. Tan el dissenyador com el programador han de realitzar les tasques d'aprenentatge dels algorismes que s'hauran d'implementar, per tant, aquestes són compartides per ambdós. Tot i això, el dissenyador fa menys hores que el programador donat que l'arquitectura software del motor no és la part més important en aquest projecte.

Finalment, tinc un rol dedicat exclusivament a testejar que el software funciona després de cada fase de implementació.

11.2 Costos hardware

Aquí incloure totes les despeses relacionades amb l'equip hardware necessari per dur a terme el projecte:

Material	Preu (€)	Quantitat	Cost total (€)
Asus k55V notebook	751,46	1	751,46
			751,46

Taula 5: taula amb els costos del hardware

Aquest és el preu de l'equip que usaré per desenvolupar tot el projecte de manera que crec necessari inclou-re el seu cost al pressupost.

11.3 Costos software

Veiem doncs una llista del software que serà necessari en aquest projecte:

Software	Preu (€)	Observacions
Ubunutu	0	Software lliure
QtCreator	0	Software lliure
Google Chrome	0	Software lliure
Gantter	0	Software lliure
Microsoft Windows 7	0	Acord MSDNAA
Microsoft Visual Studio 2012	0	Acord MSDNAA
	0	

Taula 6: taula amb els costos del software

El cost total d'aquesta part es nul per dos raons. La primera, és que disposo de software lliure per fer la gran majoria de tasques. La segona, es que les eines de Microsoft que utilitzaré les distribueix la universitat mitjançant l'acord MSDNAA que té amb l'empresa.

11.4 Cost de les instal·lacions

Una despesa que cal tenir en compte és el consum de l'entorn de treball. La taula següent ho mostra:

Recurs	**Preu (€)**
Electricitat	1200
Aigua	300
Lloguer	3200
	4700

Taula 7: taula amb la suma dels costos de les instal·lacions

Aquests preus són estimats, ja que majoritàriament he treballat en espais públics com biblioteques o aules de la universitat. En un cas real caldria estimar aquesta part més acuradament.

11.5 Cost total

A la següent taula mostro l'acumulació dels costos que conformen el cost total del projecte:

Recurs	**Preu (€)**
Recursos humans	21512
Hardware	751,46
Software	0
Instal•lacions	4700
	26963,46

Taula 8: taula amb la suma dels costos totals

En definitiva aquest és el recompte final del que més o menys costaria desenvolupar el projecte en un entorn real.

Referències

[1] *Blender - http://www.blender.org/.*

[2] *Cry engine 3 - http://www.crytek.com/cryengine.*

[3] *OpenGL 4 references pages - http://www.opengl.org/sdk/docs/man/.*

[4] *SPARK Particle engine - http://sourceforge.net/projects/sparkengine/.*

[5] *Unity 3D game engine - http://spanish.unity3d.com/.*

[6] *Unreal Cascade particle engine - http://www.unrealengine.com/en/features/cascade/.*

[7] *Unreal engine - http://www.unrealengine.com/.*

[8] Opengl particle rendering. https://graphics.stanford.edu, 2011.

[9] Vbo best practices. http://www.opengl.org/wiki, October 2012.

[10] Kekoa Proudfoot Homan Igehy, Matthew Eldridge. *Prefetching in a Texture Cache Architecture.* ACM, New York, 1998.

[11] Eric Lengyel. Computing tangent space basis vectors for an arbitrary mesh, 2001.

[12] Tristan Lorach. Soft particles. NVIDIA, January 2007.

[13] Jonathan Macey. Illumination models. Bournemouth university, September 2008.

[14] Hubert Nguyen. *GPU Gems*, chapter 6. Fire in the Vulcan Demo. NVIDIA, 2004.

[15] Marc Olano. *Modified noise for evaluation on graphics hardware.* Eurographics/ACM SIGGRAPH, July 2005.

[16] Gábor Szijártó Tamás Umenhoffer, László Szirmay-Kalos. Spherical billboards and their application to rendering expl. 2007.

[17] John van der Burg. Building an advanced particle system. Gamasutra.com, 2000.

[18] David Wolf. *OpenGL 4.0 Shading Language Cookbook.* Packt publishing, 2011.