Universitat de les Illes Balears

Treball Final de Grau

ENGINYERIA INFORMÀTICA

Ampliació del motor de videojocs per a Android

SERGI CALAFAT GASTALVER

Tutor
José María Buades Rubio

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, 5 de setembre de 2016

M'agradaria expressar el meu major agraiment, en primer lloc, al meu tutor del projecte, *José María Buades Rubio*, per animar-me a correr la marató, gracies a ell he aconseguit arribar a la meta. També agrair molt el seu suport i ajuda a *Carolina Marí Martínez*, per el seu ajut en la revisió ortogràfica d'aquest document, i al meu gran amic, *Àxel Sánchez Salom*, que m'ha ajudat fent alguns gràfics per al joc final. I finalment, també agrair als meus pares, per estar sempre allà i per no haver-me posat mai cap pega en que estudiés el que volia.

Sumari

Sumari iii

Índex de figures v

Acrònims vii

Resum ix

1 Introducció 1
1.1 Context . . . 1
1.2 Motivacions . . . 3
1.3 Opcions disponibles . . . 4
1.4 Estructura del treball . . . 4

2 Motor de Videojocs 7
2.1 Què és un Motor de Videojocs? . . . 7
2.2 Història del MV[1] . . . 7
2.3 Límits dels MVs . . . 8
2.4 Arquitectura d'un MV . . . 9
2.4.1 Capa de Hardware . . . 9
2.4.2 Capa de *Drivers* . . . 11
2.4.3 Capa del Sistema Operatiu . . . 11
2.4.4 Capa de SDKs[2] i Middleware Third-Party . . . 11
2.4.5 Capa d'Independència de la Plataforma . . . 12
2.4.6 Capa del Sistema Principal . . . 12
2.4.7 Capa de Gestió de Recursos . . . 13
2.4.8 Capa del Motor de Renderitzat . . . 13
2.4.9 Capa d'Eines de Depuració . . . 14
2.4.10 Capa de Col·lisions i Físiques . . . 14
2.4.11 Capa d'Animació . . . 15
2.4.12 Capa de Dispositiu d'Interfície Humana . . . 15
2.4.13 Capa de So . . . 15
2.4.14 Capa de Networking/Multi-jugador . . . 16
2.4.15 Capa de *Gameplay Foundation System* . . . 16
2.4.16 Capa de Subsistemes Específics del Joc . . . 17

[1] Motor de Videojocs
[2] Software Development Kits

2.5 Estat del MV a l'inici del projecte 17

3 Requisits **19**
3.1 Càmeres amb Moviment 19
3.2 Scene culling 19
3.3 Parallax Scrolling 20
3.4 Anti-Aliasing 22

4 Implementació **25**
4.1 Càmeres amb Moviment 25
4.2 Scene-Culling 27
4.3 Parallax Scrolling 33
4.4 Anti Aliasing 35

5 Videojoc d'Exemple **41**
5.1 Menús del joc 41
5.1.1 Menú principal 41
5.1.2 About 41
5.2 Joc 43
5.2.1 Protagonista 43
5.2.2 Controls 43
5.2.3 Objectiu 43
5.2.4 Primera zona 44
5.2.5 Segona zona 44
5.2.6 Fons del joc 44
5.2.7 Derrota 45
5.2.8 Victòria 46

6 Conclusions **47**
6.1 Opinió Personal 47
6.2 Resultats Obtinguts 47
6.3 Possibles Millores 48

Bibliografia **49**

ÍNDEX DE FIGURES

1.1 Estimació dels beneficis de la industria dels videojocs del 2015 fins al 2019. 2
1.2 Mitja de diners (en dòlars) que es gasten a l'any els usuaris d'alguns dels jocs més importants per a mòbil. 3

2.1 Comparativa de diversos MV segons la seva re-usabilitat. 8
2.2 Diagrama que mostra l'estructura general d'un MV i com és posicionen els seus diversos components uns respecte als altres formant capes. 10
2.3 Arquitectura típica del "motor de renderitzat". 13

3.1 Concepte de *scene culling*. 20
3.2 Concepte de *parallax scrolling*. 21

4.1 Exemple de *bounding sphere* i *bounding box* per a un objecte tridimensional qualsevol. 28
4.2 Imatge de la zona de visió d'una càmera ortogràfica i la visibilitat de dos objectes. 29
4.3 Forma de la zona de visió d'una càmera perspectiva, anomenada frustum. 30
4.4 Càlcul de l'ample i l'alt dels plans *near* i *far*. 30
4.5 Visibilitat de diversos objectes per la càmera i falsos positius. 32
4.6 Diverses passes de l'algoritme FXAA[3]. 36
4.7 Comparació d'un fragment de l'escena sense AA[4] (esquerra) o emprant FXAA (dreta). Fotografies del joc amb molt de zoom per poder apreciar l'efecte. 40

5.1 Menú principal del joc. 42
5.2 Escena secundaria amb informació sobre el joc i el seu creador. 42
5.3 Primera zona del joc, on es troben els enemics. 44
5.4 Segona zona del joc, on hi ha que esquivar. 45
5.5 Pantalla de derrota. 45
5.6 Pantalla de victòria. 46

[3]Fast Approximate Anti-Aliasing
[4]Anti-Aliasing

Acrònims

TFG Treball Final de Grau

MV Motor de Videojocs

FPS First Person Shooter

SO Sistema Operatiu

SDK Software Development Kit

API Application Programming Interfaces

IA Intel·ligència Artificial

HID Human Interface Device

NPC Non-Player Characters

AA Anti-Aliasing

FXAA Fast Approximate Anti-Aliasing

Resum

El projecte desenvolupat i exposat en aquest document consisteix en l'ampliació i millora d'un motor de videojocs per a la plataforma Android. L'objectiu d'aquest és el de facilitar la creació i edició de videojocs, separant un poc el videojoc de tots els procediments de baix nivell que aquest requereix per funcionar, i estalviant molt de temps als desenvolupadors del joc, que es poden centrar en el que s'haurien de centrar, que és en desenvolupar el joc.

El motor de videojocs base, el qual s'ha estès durant aquest projecte, havia estat desenvolupat per altres alumnes anys anteriors. Encara que ja tenia molta funcionalitat implementada, també tenia encara moltes mancances i components extra que es podien implementar, i encara segueix podent-se millorar molt més després de realitzar aquest projecte.

La meva tasca en aquest project ha estat bàsicament la implementació de quatre diversos punts al motor. Implementar algun altre tipus de càmeres que seguissin al personatge principal (per que aquest es pugui desplaçar per l'escenari), filtrar els objectes que no fossin visibles per pantalla per tal de que el motor no els dibuixàs (i per tant gastes recursos inútilment), aplicar l'efecte de *parallax scrolling* que s'explica a la memòria (bàsicament és un efecte que dona una sensació de profunditat a entorns 2D) i finalment aplicar algun tipus de sistema de *Anti-Aliasing* a temps real.

Finalment, el treball també ha inclòs la realització d'un petit videojoc de mostra implementat amb el motor de videojocs millorat, que fa ús de les noves millores introduïdes al motor.

CAPÍTOL 1

INTRODUCCIÓ

L'objectiu d'aquest primer capítol es dotar al lector d'una visió general del projecte realitzat. Per a això primer es donarà un poc de context al projecte realitzat, després s'exposaran les diverses línies que estaven disponibles per desenvolupar el projecte, quines d'aquestes es van seleccionar y els objectius que s'han complit amb la realització d'aquesta pràctica.

Finalment, per tancar aquest capítol farem una petita exposició de la resta de continguts que seguiran a aquest capítol en aquesta memòria.

1.1 Context

Durant els últims anys, amb el gran impuls que han pres els dispositius mòbils, s'ha propulsat molt el desenvolupament de software per a tals dispositius. Avui en dia es desenvolupen aplicacions de tota mena per a *smartphones, tablets, smartwatches...*, com poden ser per exemple alarmes, aplicacions de missatgeria, aplicacions per fer documents, per a manejar fulls de càlcul, etc. I entre totes les possibilitats que han obert aquests dispositius per al desenvolupament d'aplicacions, també s'han vist molt impulsats els videojocs, que han trobat en els dispositius mòbils un nou (nínxol de mercat).

Tan important es el mercat dels jocs per a dispositius mòbils que tot sol ja representa el 39% dels beneficis totals de la indústria dels videojocs, com es pot observar a la figura 1.1, esperant-se que produeixi 38.84 bilions de dòlars al llarg d'aquest any[1].

Aquestes xifres tenen explicació per dos motius generalment:

1. **El gran mercat al qual donen accés els dispositius mòbils:** avui en dia tothom té com a mínim un mòbil al seu abast, hi ha molta gent que fins i tot té més d'un dispositiu, com pot ser un mòbil i una tablet o mòbil i smartwatch. Per tant, com

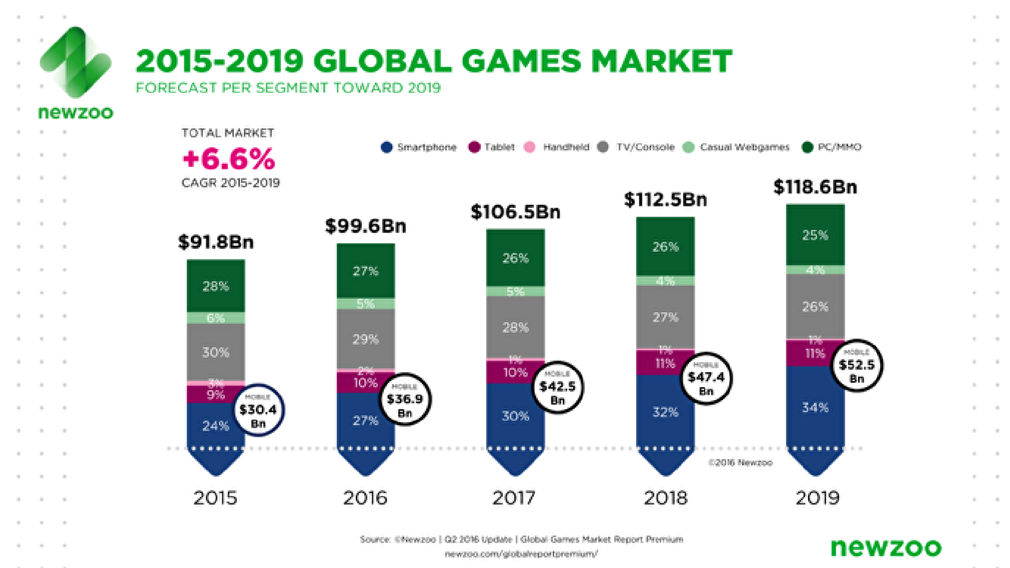

Figura 1.1: Estimació dels beneficis de la industria dels videojocs del 2015 fins al 2019.

és tan gros el nombre d'usuaris d'aquests dispositius es normal veure que es pot treure molt profit venent un producte a tots aquests usuaris, a més d'una forma molt senzilla com és posar una aplicació a les tendes virtuals del dispositiu al qual pretens vendre aquesta aplicació.

2. **El model de negoci:** Els nous models de negoci que s'han desenvolupat en els últims anys treuen el màxim profit possible a aquest gran nombre d'usuaris. Bàsicament parlo de dos models de negoci: freemium (o gratuït amb micro-transaccions) i gratuït amb anuncis. Com que aquests models no costen res a l'usuari (al menys inicialment) es més senzill captar a un gran nombre d'usuaris, dels quals al final en treus profit ja sigui fent que vegin anuncis per poder seguir jugant o bé venent-lis contingut extra del joc un cop aquests ja estan enganxats al joc i per tant son més propensos a comprar algun tipus de contingut (freemium). Aquests models de negoci permeten generar grans quantitats de diners a jocs de mòbil (que son molt més senzills actualment que els jocs d'ordinador o de videoconsola) degut al major nombre d'usuaris potencials que tenen aquest tipus de jocs.

Essent un mercat tan important i prominent el dels videojocs i concretament el dels videojocs per a dispositius mòbils, és clara la importància que té el tòpic dels motors de videojocs per a dispositius mòbils i els beneficis que pot generar aquest tipus de software donada la quantitat de desenvolupadors de jocs per mòbil que hi ha actualment.

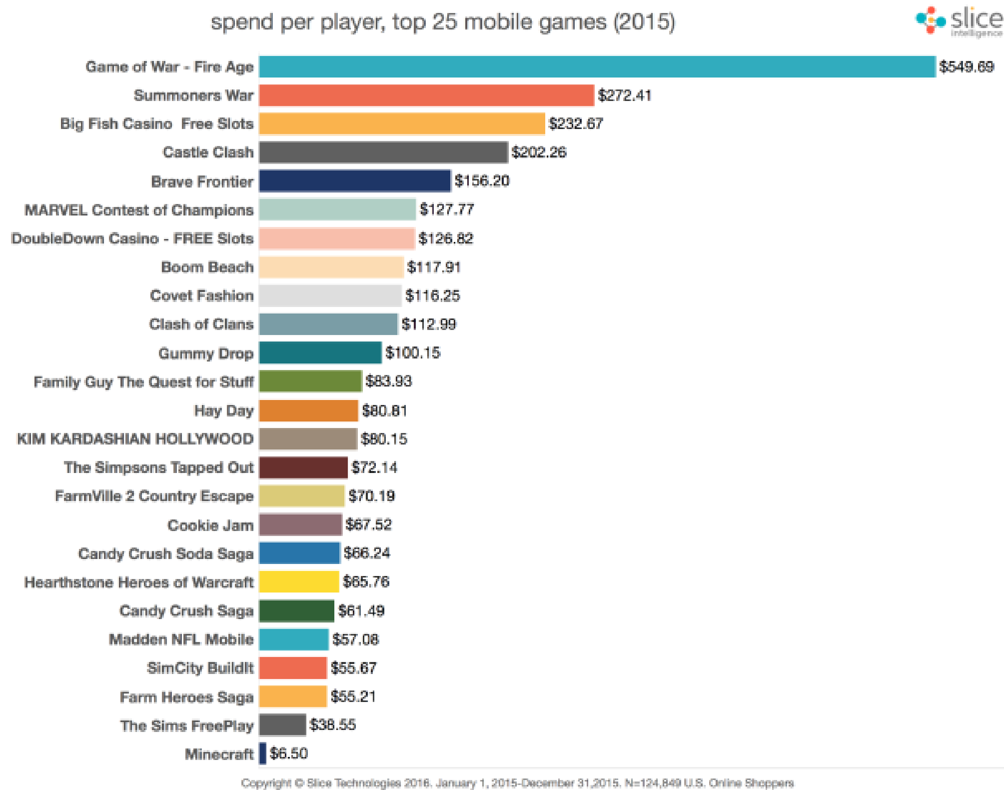

Figura 1.2: Mitja de diners (en dòlars) que es gasten a l'any els usuaris d'alguns dels jocs més importants per a mòbil.

1.2 Motivacions

Quan vaig haver de triar TFG[1] vaig mirar totes les ofertes que hi havia disponibles per veure quina em feia més ganes, la guanyadora va ser aquesta, òbviament, per els següents motius:

- **Videojocs:** Desde petit he estat jugador de videojocs i sempre han estat entre les meves aficions al llarg de la meva vida. Per mi no són simplement videojocs, consider que són art.
- **Motor de Videojocs:** És un tema que m'interessa molt, ja que venen a ser les entranyes dels videojocs i com aficionat d'aquest món sempre m'ha interessat com son per dins els videojocs. Quan havia de triar el TFG ja havia emprat alguns MV i per tant m'interessava encara més saber com funcionen.
- **Aprenentatge:** Òbviament esperava aprendre alguna cosa durant la realització del TFG, i aprendre encara que sigui el mínim sobre els MV, em pareixia prou interessant i atraient com per decidir treballar sobre el tema.

[1] Treball Final de Grau

1.3 Opcions disponibles

En agafar aquest TFG s'em van presentar diverses opcions sobre les que treballar al MV, van ser les següents:

- Afegir ombres
- Afegir més models d'il·luminació
- Permetre models animats
- Realitzar scene-culling
- Afegir efectes de post-processat
- Permetre jugar a OUYA amb el seu comandament
- Permetre scripting
- Crear un editor
- Millorar el mòdul de físiques
- Fer que sigui multithread
- Millorar el mòdul de so

D'aquestes possibilitats vaig acabar agafant la de realitzar Scene-culling, encara que al final he acabat dedicant-me, a més de al punt inicial, a millorar la gestió de càmeres, afegir *backgrounds* amb els que es pugui aplicar un efecte de *parallax scrolling* i a afegir efectes de post-processat. A més d'això, el treball també ha consistit en l'elaboració d'un joc de prova que demostri les capacitats del MV en el que he treballat.

1.4 Estructura del treball

Per posar fi a aquest primer capítol introductori de la memòria, donarem un poc de visió del que vendrà a continuació. Els pròxims capítols tracten el següent:

- **Motor de Videojocs:** Aquest capítol posa les bases de coneixement sobre les que hem treballat, explicant el concepte principal de MV i en què consisteix aquest, de quins elements està compost i, a més, una petita explicació de com es trobava el motor quan vaig començar a treballar en ell.
- **Requisits:** A aquest punt es trobaran exposats els problemes als que m'he enfrontat durant aquest treball, el que tenia que aconseguir i algunes explicacions generals dels conceptes amb els que he treballat.
- **Implementació:** En aquest capítol s'exposaran les diverses solucions que han sorgit per resoldre els problemes explicats al capítol anterior i entre les quals he hagut de decidir per resoldre el problema. A part d'això, com es d'esperar, s'explica la solució adoptada per a cada tema, amb fragments de codi i imatges representatives.

- **Videojoc d'exemple:** Aquí mostraré el videojoc que he implementat sobre el MV que he millorat, i com s'empren les noves característiques desenvolupades del motor.

- **Conclusions:** En aquest capítol faré un anàlisi dels resultats que he obtingut realitzant aquest treball i donaré la meva opinió personal d'aquesta experiència.

- **Bibliografia:** Aquí es recolliran els diversos llibres, articles, webs, noticies y documents en general als quals referenciaré durant la memòria i que m'han ajudat i servit com a base per a realitzar aquest treball.

CAPÍTOL 2

Motor de Videojocs

2.1 Què és un Motor de Videojocs?

Un MV és una plataforma de software dissenyada per la creació i desenvolupament de videojocs. Els desenvolupadors els empren per crear tot tipus de jocs per a videoconsoles, dispositius mòbils i ordinadors personals.

El nucli de funcionalitat inclou, habitualment, un "motor de renderitzat", un "motor de físiques"(detecció de col·lisions i resolució de col·lisions), gestió de so, scripting, animacions, intel·ligència artificial, streaming, gestio de la memòria, concurrencia i altres components que s'exposaran a la secció 2.4.

2.2 Història del MV

El terme "Motor de Videojocs"va sorgir a meitat dels anys 90, en referencia als jocs FPS[1] (jocs de disparar en primera persona), com per exemple el clàssic Doom desenvolupat per id Software. Doom va ser un dels primers jocs en construir-se amb una clara i ben definida separació entre el nucli dels seus components de software (com per exemple el sistema de renderitzat de gràfics tridimensionals, o el sistema de so) i els detalls propis del joc (com per exemple els recursos artístics, els diferents mapes del joc o les regles del joc) que creaven l'experiència del jugador.

El valor d'aquesta separació es va evidenciar quan els desenvolupadors van començar a llicenciar nous jocs i crear nous productes tan sols canviant l'art, els diferents mons del joc, les armes, personatges, vehicles i regles dels jocs casi sense fer canvis al "motor"del joc.

[1] First Person Shooter

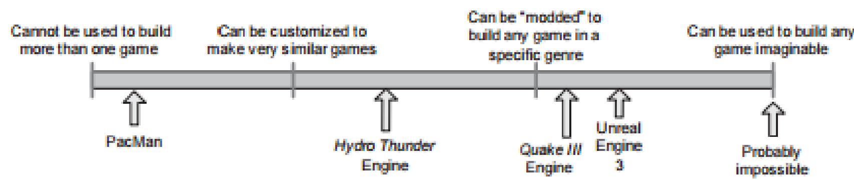

Figura 2.1: Comparativa de diversos MV segons la seva re-usabilitat.

Això també va marcar el naixement de la comunitat de "*mod*"'s, un grup de jugadors i petits estudis independents que crearen nous jocs a partir de modificar jocs existents, emprant eines gratuïtes proporcionades pels propis creadors del joc original.

Ja a finals dels 90, alguns jocs com per exemple *Quake III Arena* o *Unreal* ja es desenvolupaven pensant directament en la re-usabilitat i el *modding*. Els "motors"es feren molt customitzables a partir de llenguatges de scripting (un llenguatge de scripting es aquell que suporta scripts, que són programes escrits per a un entorn d'execució que automatitza l'execució de tasques) com per exemple "Quake C"de id Software, i la venta de llicencies d'us dels MV va començar a ser una font viable de beneficis per als desenvolupadors que els creaven.

Avui en dia, els desenvolupadors de videojocs poden comprar una llicencia d'algun MV i reutilitzar grans parts del software d'aquest per tal de construir jocs. Encara que aquestes llicencies siguin bastant cares, pot ser molt més econòmic que desenvolupar tots els components del MV. Tant es així que avui en dia hi ha companyies que tan sols es dediquen a desenvolupar i mantenir un motor de videojocs, com es el cas de Unity, un MV desenvolupat per *Unity Technologies* que permet desenvolupar jocs multi-plataforma (que es poden executar a diverses plataformes com per exemple PC i videoconsoles) i que avui en dia compta amb més de 5.5 milions d'usuaris[2], per veure quin abast tenen aquestes eines avui en dia.

2.3 Límits dels MVs

La línia que separa un joc del seu "motor"sovint no està molt definida. Alguns MV fan un clara distinció mentre que d'altres pràcticament no els intenten separar. Es pot dir que una arquitectura dirigida per les dades és el que diferencia un MV d'un videojoc que no és un "motor". És a dir, quan un joc conté lògica o regles del joc hard-codetjades, o empra codi específic per renderitzar certs tipus d'objectes, resulta difícil o impossible reemprar aquest software per fer un joc diferent. El terme MV per tant, es reserva per software que es pot usar per a crear diferents jocs sense grans modificacions.

Òbviament els motors existents no són o exclusius d'un joc o es pot crear qualsevol tipus de jocs amb ells. Tot el contrari, és una distribució més continua segons les seves capacitats de reutilització. A la figura 2.1 podem observar una comparativa d'alguns MV i com es posicionen segons la seva reusabilitat.

Es pot pensar que un MV podria ser un software de propòsit general capaç de executar qualsevol joc imaginable, però aquest ideal encara no s'ha aconseguit (i potser mai s'aconseguirà).

La majoria dels MV es creen amb cura i amb un propòsit fixat per tal de poder executar un joc concret a una plataforma concreta, i s'ha de tenir clar que com més general és el motor, és a la vegada manco òptim per a un joc en condicions concretes.

Això ocorre perquè dissenyar un software eficient sempre ens duu a fer balanços i decisions complexes que s'han de basar en assumpcions sobre com s'emprarà aquest software o sobre quin hardware s'executarà (per exemple un "motor de renderitzat"que s'ha dissenyat per manejar petits entorns interiors no manejarà tan be grans entorns exteriors).

Amb l'evolució dels computadors que cada vegada son més ràpids i l'evolució també de les targetes gràfiques, juntament amb els nous algoritmes i estructures de dades més eficients per al renderitzat, cada cop són més febles les diferencies entre els diversos motors gràfics dels jocs de diferents gèneres, però encara així sempre existirà el balanç entre la generalitat i la optimalitat del MV.

2.4 Arquitectura d'un MV

Aquest apartat pretén donar una visió general dels diversos components dels quals es composa un MV, que es poden observar conjuntament a la figura 2.2.

Els MV estan estructurats en un sistema de capes, on les capes superiors depenen de les capes inferiors. D'aquesta manera encara que es modifiqui alguna capa tan sols es veuen afectades per les modificacions les capes superiors a aquesta, per tant si es modifiquen les capes superiors no hi ha que fer cap més modificació del "motor". I, a més, aquesta estructura ens permet afegir capes sobre les capes superiors del "motor"sense afectar al funcionament d'aquest[3].

Començarem a desgranar les capes i explicar breument el seu paper al MV desde les capes més baixes fins a les més altes.

2.4.1 Capa de Hardware

Aquesta capa representa el sistema de l'ordinador o de la consola sobre el que el joc s'executarà. Alguns sistemes típics poden ser, per exemple, Microsoft Windows, ordinadors basats en Linux, un iPhone de Apple, una Playstation 4 o una Wii U.

Molts dels principis de disseny i desenvolupament són comuns per a qualsevol joc independentment de la plataforma on aquest s'acabi executant, no obstant, els desenvolupadors sempre duen a terme optimitzacions al MV per tal de millorar la eficiència del joc a les plataformes que es tenguin com a objectiu.

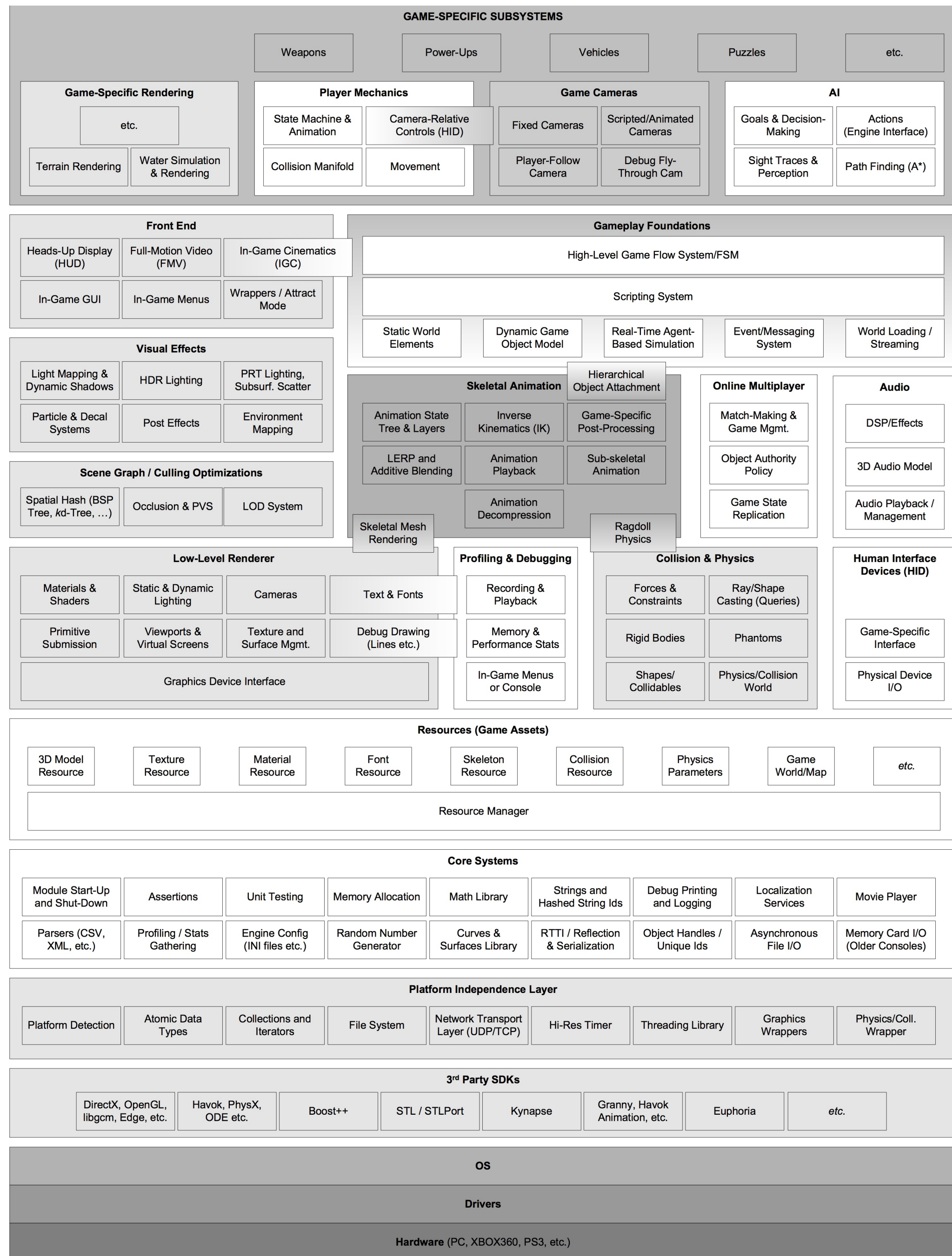

Figura 2.2: Diagrama que mostra l'estructura general d'un MV i com és posicionen els seus diversos components uns respecte als altres formant capes.

2.4.2 Capa de *Drivers*

Els *drivers* son components de software de baix nivell proveïts pel sistema operatiu o el fabricant del hardware que s'empren per manejar els recursos de hardware disponibles i així evitar al sistema operatiu i als altres nivells més alts del MV els detalls de comunicar-se amb els models concrets de hardware que es tenen disponibles.

2.4.3 Capa del Sistema Operatiu

A un ordinador el SO[2] s'executa tot el temps i s'encarrega d'organitzar l'execució de múltiples programes a un sol ordinador, un dels quals serà el joc. El SO més comú, com és, Microsoft Windows, empra una tècnica de divisió temporal per tal de compartir el hardware amb els múltiples programes que s'executin, això significa que un joc d'ordinador mai pot assumir que té el control absolut del hardware, ha de conviure amb la resta de programes del sistema.

En canvi, a una videoconsola el SO habitualment era tan sols una petita llibreria que es compilava directament dins l'executable del joc i per tant el joc podia emprar com a propi tot el hardware de la màquina. No obstant, a les consoles modernes com per exemple la Xbox One o Playstation 4 aquest ja no és el cas, sinó que el SO d'aquestes màquines també pot pausar e interrompre l'execució del joc, o bloquejar alguns recursos del sistema. Així que actualment cada cop es menor la diferencia entre el desenvolupament per a ordinador i per a videoconsola.

2.4.4 Capa de SDKs i Middleware Third-Party

Així com en altres projectes software, el desenvolupament d'un MV es sol recolzar en llibreries externes i SDKs per a proporcionar una determinada funcionalitat. Tot i això, i encara que generalment aquest software està bastant optimitzat, alguns desenvolupadors prefereixen personalitzar-lo per tal d'adaptar-lo a les seves necessitats particulars, especialment en videoconsoles i portàtils. Dins aquesta capa conviuen diversos tipus de llibreries que podem dividir en: estructura de dades i algoritmes, gràfics, col·lisions i físiques, animació de personatges i intel·ligència artificial.

Un bon exemple de llibreria per al maneig de estructures de dades seria STL (Standard Template Library). Aquesta llibreria de plantilles estàndard de C++, que representa alhora el llenguatge més extés actualment per al desenvolupament de videojocs, degut majorment a la seva portabilitat i eficiència.

A l'àmbit dels gràfics 3D, hi ha un gran nombre de SDKs que ajuden a solventar determinats aspectes comuns als videojocs, com és el renderitzat 3D. Els exemples més clars són OpenGL i DirectX (OpenGL és multi-plataforma, mentre que DirectX es exclusiva dels sistemes Microsoft). Aquest tipus de llibreries tenen com a principal objectiu ocultar aspectes de més baix nivell de les targetes gràfiques, presentant una interfície comú per a totes elles.

[2] Sistema Operatiu

2.4.5 Capa d'Independència de la Plataforma

Molts de MV es desenvolupen amb l'objectiu d'executar jocs a més d'una plataforma de hardware. Grans companyies com per exemple Activision/Blizzard o Ubisoft sempre desenvolupen els seus jocs per una gran varietat de plataformes, ja que això els exposa a un mercat de compradors molt major. Típicament els desenvolupadors de jocs que tan sols desenvolupen jocs per a una plataforma exclusiva són desenvolupadors propietaris de la plataforma per la qual desenvolupen, com per exemple Naughty Dog, que es propietat de Sony.

Per tant la majoria de MV tenen una capa d'independència de la plataforma. Aquesta capa esta per sobre del hardware, els *drivers*, el SO i la resta de Software de tercers i evita a la resta del "motor"de tenir que conèixer sobre quina plataforma s'executarà el producte final. Això s'aconsegueix encapsulant o reemplaçant les funcions més emprades de les llibreries estàndard, les cridades al SO i a altres APIs[3], i per tant s'aconsegueix un comportament consistent en les diverses plataformes de hardware destí.

2.4.6 Capa del Sistema Principal

Cada MV i, en general, tota gran i complexa aplicació de C++ requereix una gran quantitat de utilitats de software, així que aquestes es categoritzen sota el nom de "Sistema Principal". A continuació es llisten uns quants exemples de les utilitats que sol aportar aquesta capa:

- **Depuració:** proporciona eines per tal de facilitar la depuració del codi i evitar errors no desitjats. Habitualment aquest codi s'elimina de la versió final del producte.

- **Maneig de la memòria:** tots els MV implementen els seus propis sistemes de maneig de la memòria per tal d'assegurar assignacions i alliberacions de la memòria d'alta velocitat, i per limitar la fragmentació de la memòria.

- **Llibreries matemàtiques:** els videojocs acostumen a emprar moltes matemàtiques. Per tal, els MVs tenen com a mínim una, si no varies, llibreries matemàtiques. Aquestes proporcionen facilitats per a matemàtiques de vectors i matrius, trigonometria, operacions geomètriques, esferes, etc.

- **Estructures de dades i algoritmes:** a no ser que es decideixi cenyir-se exclusivament a algun paquet de software de tercers, es sol requerir un conjunt d'eines per tal de manejar estructures de dades fonamentals (llistes enllaçades, arrays dinàmics, arbres binaris...) i algoritmes (cerca, ordenació...). Aquestes solen estar fetes a mà per tal de minimitzar o eliminar per complet l'assignació dinàmica de memòria i assegurar un rendiment òptim a l'execució sobre la plataforma objectiu.

[3] Application Programming Interfacess

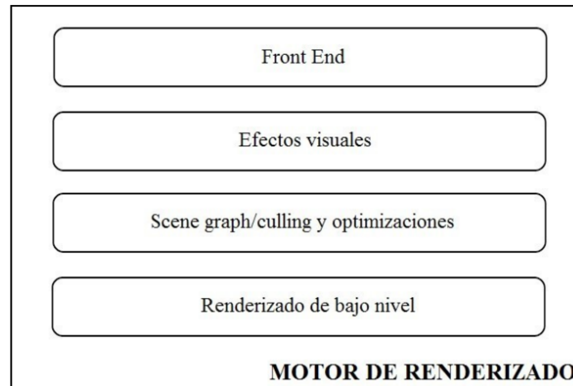

Figura 2.3: Arquitectura típica del "motor de renderitzat".

2.4.7 Capa de Gestió de Recursos

Aquesta capa proveeix una interfície comú per tal de poder accedir a tots els tipus de recursos del joc o altres dades d'entrada del "motor". Alguns "motors"fan això d'una manera molt centralitzada i consistent, mentre que d'altres ho fan a la seva manera (habitualment deixant-li al programador accedir directament als fitxers del disc o fins i tot als arxius comprimits).

2.4.8 Capa del Motor de Renderitzat

Aquesta capa és una de les més grans i més complexes de qualsevol motor de videojocs. Els motors de renderitzat es poden construir de múltiples maneres, no hi ha cap forma acceptada de fer-ho, no obstant, la majoria dels motors de renderitzat moderns comparteixen alguns conceptes fonamentals de disseny, conduits en gran part pel disseny del hardware de gràfics 3D del qual depenen.

Una tàctica comú i efectiva de dissenyar el motor de renderitzat és emprar una arquitectura de capes com la que es pot observar a la figura 2.3.

Renderitzat de baix nivell
La capa de renderitzat de baix nivell reuneix les diverses funcionalitat associades a la representació gràfica dels diversos objectes que participen en un determinat entorn com per exemple la càmera, primitives de renderitzat, materials, textures, etc. El principal objectiu d'aquesta capa és renderitzar les diverses primitives geomètriques el més ràpidament possible, sense tenir en compte possibles optimitzacions, com per exemple, quines parts de l'escena són visibles per la càmera.

Scene graph culling i optimitzacions
És la capa immediatament superior a la de renderitzat de baix nivell. És la responsable de seleccionar quines parts de l'escena s'enviaran a la capa de renderitzat de baix nivell. Aquesta optimització permet incrementar el rendiment del motor de renderitzat, ja que limita el nombre de primitives geomètriques que es renderitzaran.

Típicament aquestes optimitzacions consisteixen en emprar estructures de dades de subdivisió espacial per tal de fer més eficient el renderitzat, ja que aquestes permeten determinar d'una forma més ràpida el conjunt d'objectes parcialment visibles. Aquesta capa també s'encarrega d'emprar mètodes de *culling* per tal de determinar els objectes que no són visibles ja que algun altre objecte els tapa.

Efectes visuals

Aquesta capa maneja les necessitats específiques de renderitzat de les partícules, els sistemes de calcomanies y altres efectes visuals. Aquests dos sistemes mencionats acostumen a esser dos components del sistema de renderitzat i actuen com a entrades per a la capa de renderitzat de baix nivell.

D'altra banda, els sistemes de *light mapping*, *environment mapping* i ombres es manegen internament al "motor de renderitzat". Els efectes de post-processat de pantalla completa s'implementen o bé com a part del "motor de renderitzat"o bé com un component apart que opera sobre els buffers de sortida del "motor de renderitzat".

Front End

Aquesta capa agrupa els diversos tipus de gràfics 2D que es superposen a la escena 3D als videojocs, com son, per exemple, el HUD (Heads-up Display), menús interns, una consola de comandaments, altres eines de desenvolupament o una GUI (Graphical User Interface) per al joc que permeti al jugador manipular el seu inventari, configurar unitats per a una batalla o realitzar qualsevol altre tipus de tasca.

2.4.9 Capa d'Eines de Depuració

Els videojocs són sistemes a temps real, i com a tals els desenvolupadors habitualment necessiten mesurar el rendiment dels seus jocs per tal d'optimitzar-los. A més d'això, la memòria pot ser un recurs escàs, per tant els desenvolupadors també fan ús d'eines d'anàlisi de la memòria.

Aquesta capa encapsula aquestes eines i també inclou eines per a *debuguejar* els videojocs, tals com *debug drawing*, algun sistema intern o consola, i l'habilitat de gravar i reproduir contingut del joc per tal de provar-lo i arreglar-lo.

2.4.10 Capa de Col·lisions i Físiques

La detecció de col·lisions es important a tots els jocs, sense ella els objectes penetrarien els uns als altres i seria impossible interactuar amb el món virtual d'una forma normal. Alguns videojocs també inclouen algun sistema realista o semi-realista de simulació dinàmica. Aquesta capa s'encarrega per tant de detectar la col·lisió, determinar la col·lisió i resoldre-la.

En l'actualitat, la majoria de companyies empren motors de físiques desenvolupats per *Third-Parties*, integrant-los en el mateix motor.

2.4.11 Capa d'Animació

Qualsevol videojoc que tengui personatges orgànics o semi-orgànics (humans, animals, robots. . .) necessita un sistema d'animació. Hi ha 5 tipus bàsics d'animació que s'empren a videojocs, aquests són:

- Animació de Sprites/Textures
- Animació jeràrquica de cossos sòlids
- *Skeletal Animation*
- *Vertex Animation*
- *Morph Targets*

D'aquests el més emprat als videojocs avui en dia és la *skeletal animation*, que consisteix en permetre que un model 3D pugui ser posat per un animador mitjançant un senzill sistema d'ossos, així a mesura que els ossos es mouen també ho fan els vertex del model 3D.

2.4.12 Capa de Dispositiu d'Interfície Humana

Aquesta capa s'encarrega de processar les accions d'entrada dels jugadors, que s'obtenen a partir de HIDs[4] com per exemple el ratolí, el teclat, un joypad o altres controladors. També s'encarrega, a vegades, de donar alguna informació de sortida a l'usuari, per exemple fent vibrar el controlador o emetent algun só de sortida a través del controlador.

Aquest component sovint es dissenya de tal forma que separa els detalls de baix nivell dels diversos tipus de controladors d'una plataforma específica de hardware dels controls d'alt nivell que empra el videojoc, i amb això també sol proporcionar algun tipus de mecanisme per tal de permetre al jugador final assignar els diversos controls físics dels que disposa a les funcions lògiques que empra el videojoc.

2.4.13 Capa de So

L'àudio és tan important com els gràfics a qualsevol MV, encara que sovint sol rebre menys atenció que el renderitzat, les físiques, l'animació, la IA[5] i les mecàniques de joc.

Els "motors de so"solen variar molt en refinament, els de Quake o Unreal per exemple són molt bàsics, per tant molts d'equips de desenvolupament els modifiquen o els reemplacen per complet per alguna altra solució completa.

Cal mencionar que inclús si s'empra un "motor de so"preexistent, cada joc acostuma a necessitar bastant desenvolupament, integració i posta a punt per tal de produir àudio de gran qualitat al producte final.

[4] Human Interface Devices
[5] Intel·ligència Artificial

2.4.14 Capa de Networking/Multi-jugador

Molts jocs permeten a múltiples jugadors jugar dins un sol món virtual. De fet hi ha companyies que s'han especialitzat en aquest tipus de jocs i tan sols produeixen jocs multi-jugador, com per exemple Blizzard Entertainment, propietat d'Activision/Blizzard, que a partir del seu increïblement popular MMORPG (Massive Multiplayer Online Role Playing Game) World of Warcraft, tan sols ha produït jocs multi-jugador.

Aquest tipus de videojocs es poden dividir en 4 tipus bàsics:

- **Multi-jugador d'una pantalla:** dos o més HIDs es connecten a una única màquina. Cada jugador controla a un personatge, tots els personatges es troben al mateix món virtual i una única càmera els manté a tots en pantalla a la vegada. Un exemple d'aquest tipus pot ser Smash Bros.
- **Multi-jugador de pantalla partida:** múltiples personatges es troben a un únic món virtual, amb múltiples HIDs connectats a una única màquina però cada jugador amb la seva pròpia càmera. Per tant la pantalla es divideix en diverses seccions per tal de que cada jugador pugui veure el seu personatge. N'és un exemple el joc Mario Kart.
- **Multi-jugador online:** múltiples màquines s'interconnecten a través de la xarxa, cada màquina representa un jugador.
- **Jocs multi-jugador online massius:** un videojoc on poden estar jugant simultàniament fins a mils de milers d'usuaris, tots dins un únic món virtual que mantenen un gran nombre de servidors.

Encara que els jocs multi-jugador solen ser molt similars als jocs d'un jugador en molts aspectes, el fet de tenir que tractar amb múltiples jugadors té un gran impacte en el disseny de certs components del MV. Per tant, és molt recomanable, si es possible, dissenyar les capacitats de multi-jugador des de l'inici del desenvolupament del joc.

2.4.15 Capa de *Gameplay Foundation System*

Aquesta capa agrupa tots els mòduls relatius al funcionament intern del joc. D'aquesta forma es permet la definició de les regles que governen el món virtual conjuntament amb les habilitats del personatge, els objectes del món i els objectius del joc.

També seveix com a capa d'aillament entre les capes inferiors i tot el funcionament propi del joc. Per aconseguir això es separa la lògica del joc de la implementació emprant un sistema de scripting (llenguatge d'alt nivell que s'empra per a definir el comportament del joc).

A més, és a aquesta capa on s'integra el sistema d'esdeveniments dels quals la principal funció és comunicar els diversos objectes, independentment del seu tipus o naturalesa. Aquests esdeveniments consisteixen en una estructura de dades que conté informació rellevant de manera que la comunicació està precisament guiada pel contingut de l'esdeveniment i no per el de l'emissor o el receptor del mateix.

2.4.16 Capa de Subsistemes Específics del Joc

Per sobre de la resta de components del MV els desenvolupadors i dissenyadors del joc cooperen per tal d'implementar el joc en sí. Aquests subsistemes dels jocs són, per tant, habitualment molt nombrosos, molt variats i específics del joc que s'està desenvolupant.

En són exemples les mecàniques del personatge del jugador, els sistemes de càmeres del joc, la IA que controlarà als NPCs[6] o els sistemes d'armes.

Si s'hagués de definir on es troba la línia que separa el "motor"del joc, aquesta es trobaria entre aquesta capa i l'anterior, però a la realitat mai hi ha una separació clara i sovint el coneixement propi del tipus de joc que es desenvolupa passa a través de la capa de *Gameplay Foundation System* fins a capes més internes del "motor".

2.5 Estat del MV a l'inici del projecte

Aquí llistaré com es trobava cada una de les capes del nostre MV quan vaig començar el projecte.

- **Capa de Hardware:** Permet l'execució de jocs sobre dispositius Android, inclou OUYA i MOJO (petites micro-consoles per a jugar a jocs d'android).
- **Capa de *Drivers*:** Iniciada.
- **Capa del Sistema Operatiu:** Iniciada però requereix d'una ampliació.
- **Capa de SDKs i Middleware Third-Party:** Iniciada i ampliada diversos cops. S'anirà ampliant a mesura que s'afegeixin funcionalitats al MV.
- **Capa d'Independència de la Plataforma:** Capa en expansió.
- **Capa del Sistema Principal:** Iniciada i completa de moment. A mesura que s'introudeixin noves característiques al MV s'haurà d'ampliar.
- **Capa de Gestió de Recursos:** Desenvolupada de manera extensible.
- **Capa del Motor de Renderitzat:** Iniciat, amb quatre tipus de postprocessat, encara hi ha espai per a millores.
- **Capa d'Eines de Depuració:** No n'hi ha.
- **Capa de Col·lisions i Físiques:** Bastant desenvolupada, el MV es capaç de gestionar les col·lisions i informar de quines es produeixen perque es resolguin.
- **Capa d'Animació:** Tan sols inclou animació per transformació.
- **Capa de Dispositiu d'Interfície Humana:** Iniciada, amb alguns elements, pero encara es podria millorar bastant.

[6]Non-Player Characterss

- **Capa de So:** En un estat molt bàsic.
- **Capa de Networking/Multi-jugador:** No n'hi ha, el motor tan sols dona suport a jocs d'un sol jugador de moment.
- **Capa de *Gameplay Foundation System*:** Desenvolupada, amb un sistema de scripting que permet separar-la del core del MV.
- **Capa de Subsistemes Específics del Joc:** Desenvolupada, igual que el punt anterior, es basa en un sistema de scripting per als desenvolupadors del joc.

CAPÍTOL 3

Requisits

Aquest capítol exposarà cada un dels temes amb els que he tractat al treball per tal de millorar el MV que se'm va proporcionar per a realitzar el treball. Dividiré aquest capítol en 4 seccions, una per cada un dels problemes més generals que he tractat: afegir càmeres amb moviment, realitzar *scene-culling* per tal d'optimitzar el renderitzat de l'escena, aplicar un sistema de Parallax Scrolling per poder afegir *backgrounds* més rics a les escenes i aplicar una tècnica de *anti-aliasing* per tal de dissimular l'efecte d'aliàsing que es produeix al renderitzar elements.

3.1 Càmeres amb Moviment

El primer que vaig realitzar per al MV va ser la creació d'un nou tipus de càmera que realitza un seguiment d'algun actor que li sigui especificat, per tal de que l'actor designat sempre sigui visible per pantalla.

A més, no tan sols volíem que la càmera seguís a l'actor i el mantengués centrat en pantalla en tot moment, sinó que volíem que la càmera tan sols es desplaçàs a partir de certs marges de moviment. És a dir, que tan sols es mogués en una certa direcció si l'actor s'havia allunyat una certa distancia del centre de la càmera.

3.2 Scene culling

A la materia de renderitzat 3D el terme de *culling* s'empra per descriure el descart de qualsevol tipus d'objectes (objectes, triangles, píxels, etc) que no contribueixen a la imatge final[4]. Existeixen moltes tècniques que descarten objectes a diverses fases del renderitzat, algunes d'aquestes són plenament de software a la CPU, mentre que d'altres empren també ajuda del hardware (GPU) i algunes estan implementades internament dins les targetes gràfiques.

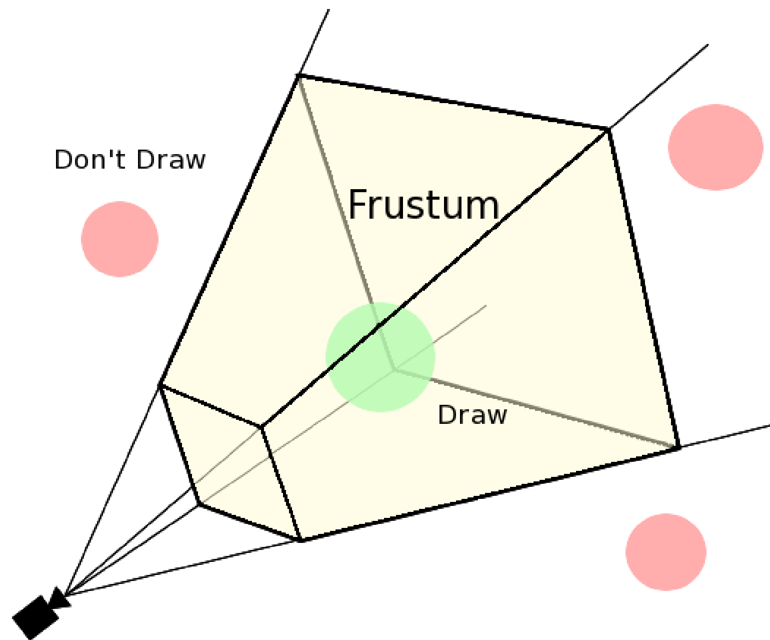

Figura 3.1: Concepte de *scene culling*.

Per tal d'evitar el màxim de processament que sigui possible, el millor és descartar el més prest possible i descartar el màxim d'objectes possibles, però per una altra banda el procés de *culling* tampoc ha de costar massa rendiment o memòria, per tant per tenir el millor rendiment possible, s'ha de maximitzar el rendiment del *culling* minimitzant al màxim el seu cost.

Habitualment el *culling* es duu a terme al nivell de les cridades al dibuixat dels objectes, no es sol anar més abaix, al nivell de triangles o més enllà, ja que sovint això és ineficient. Això vol dir que de vegades pot tenir sentit dividir grans objectes en múltiples parts per tal de que no totes les parts es tenguin que renderitzar conjuntament.

En el nostre cas l'única part del *culling* que teníem que realitzar era la de descartar tots els objectes que, encara que es trobin a escena, no són visibles per la càmera. Això, que no te perquè ser molt costós computacional-ment, redueix enormement el treball de renderitzar la imatge final, sobretot per escenes molt grosses aquest procediment és absolutament indispensable, ja que sinó pintaríem infinitat d'objectes que igualment al final no són visibles per ningú. Es pot veure una representació visual d'això a la figura 3.1.

3.3 Parallax Scrolling

El *parallax* es defineix com al desplaçament aparent d'un objecte observat degut a qualcun canvi en la posició de l'observador. L'efecte de *parallax scrolling* en 2D, llavors, és aquell on la posició de l'observador tan sols varia sobre els eixos X i Y. Per tant, tan sols la velocitat i la ubicació d'un objecte canviaran amb la posició de l'observador.

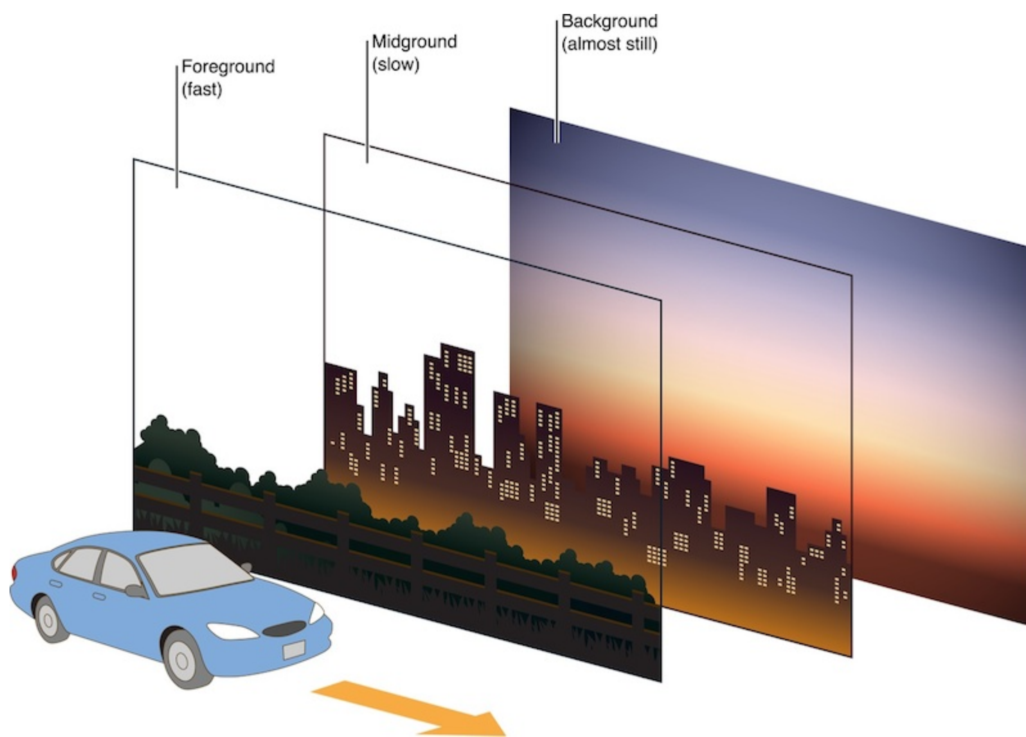

Figura 3.2: Concepte de *parallax scrolling.*

El primer joc que va emprar aquest efecte és el *Moon Patrol* fet per *Takashi Nishiyama*, però aquesta tècnica ja existia a l'animació des de l'any 1933. La tècnica consistia en que, emprant una càmera multi-pla, els animadors eren capaços de crear un efecte tridimensional no estereoscòpic, que donava la il·lusió de profunditat a l'observador fent que els diferents recursos artístics es moguessin a diferents velocitats en relació amb la distancia percebuda respecte de la càmera.

Així és com s'aconsegueix aquest efecte als videojocs moderns, però enlloc d'emprar una càmera multi-pla s'empren escenes on es defineixen múltiples capes i tan sols una càmera o vista. Dividint els elements de fons i de front d'un joc en diverses capes és possible controlar la velocitat i la posició de cada un d'aquests elements basant-se en la capa a la qual es troba. Per tant, l'observador, que en aquest cas és el jugador, i la càmera es centren en un punt o un objecte particular, i totes les capes es mouen d'acord a la seva profunditat.

Aquest punt focal, que sol coincidir amb el personatge principal del joc, es mou per tant a velocitat "normal", i les capes es mouen segons la profunditat que els hi vulguem donar. Quan més profunda volem que sigui una capa, menor serà la velocitat dels objectes d'aquesta capa, mentre que les capes més properes es mouran més ràpidament. Això produeix una il·lusió de profunditat al jugador, que fa l'escena (que està en 2D) paregui molt més immersiva.

Per tant, el nostre objectiu a aquest punt era crear un sistema de capes per a renderitzar diverses capes de fons al nostre videojoc, dotant aquestes capes d'un moviment relatiu al moviment de la càmera i aconseguint aquest efecte de *parallax scrolling*. Podem veure una representació visual d'aquesta explicació a la figura 3.2.

3.4 Anti-Aliasing

Per parlar d'Anti-Aliàsing primer hem de definir el que coneixem com a aliàsing. L'aliàsing es com s'anomena l'efecte de "serra"(*jagging*) que es produeix a les arestes dels gràfics per ordinador. En termes més científics els aliàsings són artefactes causats per tasses de mostreig que són menors que el doble d'altes que la freqüència (com exposa el teorema del mostreig de Nyquist i Shannon). Les mostres són punts infinitesimals que s'empren per calcular el color del píxel. Quan no s'empra anti-aliasing tan sols hi ha una mostra al centre de cada píxel.

Hi ha dues formes bàsics de aconseguir reduir els efectes de l'aliàsing, i tots els mètodes de anti-aliàsing existents es basen en aquests mètodes. Són els següents:

1. Augmentar la tassa de mostreig (com per exemple es fa a les tècniques MSAA, SSAA o CSAA).

2. Difuminar les arestes o els contrasts (com per exemple a les tècniques de MLAA, FXAA o SMAA), que també es coneix com a *Post-AA o (post-anti-aliasing)*.

A continuació explicarem, de forma general, en què es basen algunes de les tècniques d'anti-aliasing.

- **FSAA (Full Screen AA) o SSAA (Super Sampling AA):** aquesta tècnica consisteix en augmentar el nombre de mostres que s'empren i calcular el color de cada píxel fent un promig de les mostres que aquest píxel conté, això resulta en píxels que tenen una mescla dels colors que en realitat són a dins.

  Aquesta es la forma d'AA que dona uns millors resultats a les imatges finals, reduint molt efectivament els efectes d'aliasing i sense produir cap tipus de difuminat a l'imatge final. La seva principal contrapartida és el gran rendiment que necessita, ja que l'imatge s'ha de renderitzar com a mínim el doble de gran en cada eix i per tant és com si renderitzassis cada imatge 4 cops.

- **MSAA (Multi Sampling AA):** consisteix en tan sols prendre més mostres a la geometria que intentam dibuixar. És a dir, el buffer que es necessita té el mateix tamany que per a SSAA (amb espai per guardar múltiples mostres per píxel), però tan sols s'escriuen les mostres que cobreix la geometria que anam a dibuixar, per tant, a les arestes, tan sols s'escriurà sobre algunes de les mostres, mentre que al centre de la geometria s'escriuran totes. Però a SSAA el fragment shader s'executarà un cop per cada mostra (produint diversos resultats per cada mostra dins un mateix píxel), mentre que a MSAA el shader tan sols s'executa

un cop per cada píxel i el resultat es comparteix entre totes les mostres que pertanyen a tal píxel. Per això, requereix menys potència computacional que el SSAA.

- **EQAA (Enhanced Quality AA) i CSAA (Coverage Sample AA):** intenten millorar la qualitat de MSAA. La forma en que ho fan es bastant complexa així que tan sols afegirem que s'intenta incrementar el nombre de mostres però mantenint el mateix nombre de mostres de color, profunditat i *stencil*.
- **MLAA (Morphological AA) i FXAA (Fast Aproximate AA):** aquestes ja són tècniques de post-processat que empren filtres de distorsió. Primer detecten contrasts (les arestes) de la imatge i llavors els distorsiona. Això redueix considerablement l'efecte de "serra"(*jagging*) que produeix l'aliasing. Aquesta és la forma menys costosa computacionalment de AA, i per tant és la que sovint s'empra als videojocs, sobretot de videoconsoles, que solen tenir menys potència.
- **SMAA (Subpixel Morphological AA):** és una evolució del MLAA. La detecció de aliasing està millorada i és més propera a la que s'empra en MSAA. Com a resultat SMAA és encara molt menys costós que MSAA i també redueix l'efecte de "serra", però no distorsiona tant la imatge com FXAA o MLAA.
- **TXAA (Temporal AA):** és una forma molt complexa de AA que es basa en combinar la imatge actual amb la d'un instant de temps anterior per tal de tenir més mostres amb les quals treballar per cada píxel. No obstant també requereix un gran rendiment per part del computador que l'hagi d'emprar.

En aquest punt, per tant, el nostre objectiu era implementar alguna d'aquestes tècniques per al nostre motor de videojocs en Android.

CAPÍTOL 4

Implementació

4.1 Càmeres amb Moviment

Per a implementar el requisit exposat a l'apartat 3.1 podria haver optat per molts de tipus de càmeres, com per exemple:

- Càmera en tercera persona. Càmera a la que es veu el personatge en tot moment, sovint a una certa distància fixa d'aquest.
- Càmera en segona persona. Una càmera que seguís al actor principal desde rere, mostrant a aquest parcialment.
- Càmera amb perspectiva isomètrica. On la càmera es situa molt per sobre el personatge, i per tant sens permet veure una gran part del terreny.

En el nostre cas ens hem decantat per càmeres en tercera persona que enfoquen al personatge desde un lateral, permetent veure tant a l'actor com al món com es sol veure als jocs 2D de plataformes.

En quant al moviment de la càmera teníem les següents possibilitats:

- Una càmera amb seguiment. Que segueixi els moviments de l'actor.
- Una càmera interactiva. Que a més de seguir a l'actor principal, aquesta càmera sigui configurable fins a cert punt, podent modificar l'orientació o be la distància d'aquesta a l'actor.
- Càmera sobre rails. Té una ruta concreta predefinida i no es pot sortir d'aquesta.

I d'aquests tipus de moviments de càmera vam decidir que la càmera tan sols realitzàs un seguiment simple del personatge principal.

També vam decidir, però, donar uns certs marges de moviment al personatge en els quals la càmera no es desplaça. A la càmera 2D mentre el personatge està dins aquests límits, la càmera està quieta, mentre que a la càmera 3D si el personatge està dins aquests límits la càmera rota per seguir centrant la imatge sobre aquest.

Per tant, un cop decidides les càmares que anava a implementar, el primer que vaig fer va ser generar una nova classe de java, que vaig anomenar *FollowCamera*.

```
public class FollowCamera extends Camera {

    private String targetId;
    private Actor target;
    private float maxElapseX;
    private float maxElapseY;
    private Point3D prePos;

    public FollowCamera(GameEngineContext gec, String
       name, String targetId, Point3D eye, Point3D look,
       Point3D up, boolean ortho, float maxElapseX, float
       maxElapseY)

    @Override
    public void simulate(float delta)

    public String getTargetId()

    public void setTarget(Actor target)

}
```

Com es pot observar es va incloure a la classe els següents atributs:

- **targetId:** el id de l'actor al qual la càmera ha de seguir.
- **target:** una referència a l'actor que es segueix. Hi ha l'id i també la referència, ja que quan declaram la càmera, l'actor pot ser que no estigui carregat a l'escena, així que guardam a cada càmera l'id de l'actor que ha de seguir i un cop s'han carregat tots els actors a escena s'assigna la referència a aquest actor a partir del seu id.
- **maxElapseX:** aquest atribut indica a la càmera quin és el màxim desplaçament que es permet a l'actor respecte al centre de la càmera a l'eix de les X.
- **maxElapseY:** aquest atribut indica a la càmera quin és el màxim desplaçament que es permet a l'actor respecte al centre de la càmera a l'eix de les Y.
- **prePos:** posició prèvia de l'actor al frame anterior per tal de veure cap a on s'està movent l'actor i moure la càmera en conseqüència.

En quant als mètodes tenim:

- **FollowCamera:** el constructor que s'encarrega de generar una càmera d'aquest tipus i popular tots els seus atributs.
- **simulate:** mètode sobre-escrit a la classe pare, que actualitza aquesta càmera per cada recorregut del bucle de simulació general.
- **getTargetId:** mètode que retorna l'id de l'actor que ha de seguir la càmera per tal de cercar-lo i posteriorment assignar la seva referència.
- **setTarget:** mètode que assigna l'actor al qual seguirà aquesta càmera i que a més inicialitza la posició prèvia d'aquest.

Bàsicament tot el comportament que tendrà aquesta càmera ens ve definit, per tant, al mètode de simulació. Dins d'aquest mètode hem definit dos tipus diferents de comportament de la càmera segons si aquesta és o no ortogràfica, es a dir, si és una càmera per a 2D o per a 3D.

El funcionament de les dues no és gaire diferent, si és una càmera ortogràfica es comprova si l'actor està superant els marges de moviment definits per aquesta càmera, i si els està excedint es desplaça la càmera en la direcció en la qual s'està desplaçant l'actor, tant a les X com a les Y.

En canvi, si la càmera no és ortogràfica el que canvia és que mentre que l'actor no excedeixi els límits definits per la càmera, aquesta enlloc de desplaçar-se per seguir-lo rotarà, i tan sols un cop l'actor sobrepassi els marges definits la càmera el seguirà.

Per acabar aquesta secció només falta afegir que també s'ha hagut de modificar el *parser* de l'escena per tal de poder declarar el nou tipus de càmeres i també s'ha definit un tipus enumerat per tal de definir quin tipus de càmera volem generar.

4.2 Scene-Culling

Per tal de determinar quins actors són visibles a l'escena en un determinat instant de temps, ens hem decantat per dues tècniques diferents, depenent de si la càmera és o no ortogràfica, com al punt anterior.

Però el primer que hem hagut de fer és, per tots els actors que carregam a l'escena, definir un cos tridimensional simple que contengui tots els vèrtex de l'actor, per tal de testejar aquest darrer amb la zona visible de l'escena. Aquest pas és necessari ja que comprovar la visibilitat dels actors havent de comprovar tots els seus vèrtex respecte de la zona visible seria increïblement costos computacionalment.

Això es sol fer mitjançant un d'aquests dos tipus de cosos, una *bounding sphere* o un *bounding box*, és a dir, una esfera o una caixa. En el nostre cas hem decidit emprar *bounding boxes* ja que consideram que engloben millor els actors de la nostra escena, on hi ha terreny, que és molt llarg però quasi no te altitud. Per tant una esfera ocuparia

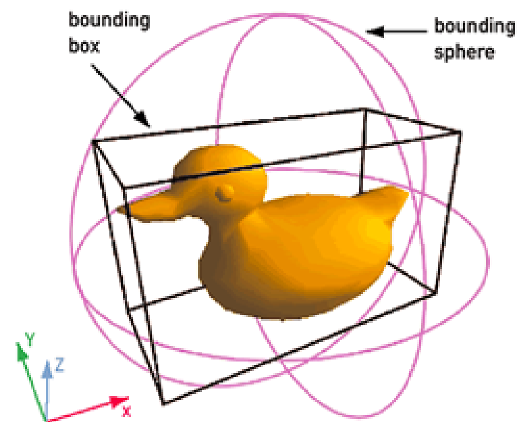

Figura 4.1: Exemple de *bounding sphere* i *bounding box* per a un objecte tridimensional qualsevol.

una gran zona d'aquest actor on l'actor no hi seria present, mentre que una caixa englobaria casi a la perfecció tan sols la zona on es troba l'actor.

Cal dir que les comprovacions de visibilitat per una esfera solen ser més simples, però en el nostre cas al menys, l'us d'una caixa donarà molts millors resultats pel motiu exposat al paràgraf anterior. A la figura 4.1 podem veure una representació gràfica d'aquests dos volums contenidors.

També hem definit un mètode de la càmera que es diu *isInSight* i que rep com a paràmetres el *bounding box* de l'actor i la posició central d'aquest, i hem afegit una cridada a aquest mètode per la càmera activa i per tots els actors de la escena al bucle principal de pintat de l'escena, per tal d'evitar pintar els que no siguin visibles per la càmera.

Càmera ortogràfica

La solució per les càmeres ortogràfiques ha estat definir 4 punts com a limits de la càmera als eixos X i Y. Hem pres aquesta solució assumint que la càmera serà una càmera 2D estàndard que estarà completament direccionada a l'eix Z i per tant ens bastarà comprovar que l'actor es troba dins el rectangle que defineix la càmera.

Per tal objectiu, al mètode *isInSight* feim quatre comprovacions, una amb cada un dels límits de l'escena, i si l'actor es troba al costat exterior d'algun d'ells llavors sabem que aquest actor no es visible per la càmera. Això s'il·lustra a la figura 4.2

Hauríem d'haver comprovat també que l'actor estigués contingut dins el plans *near* i *far*, per tal de no dibuixar objectes a darrera de nosaltres i molt enfora, però ho vam deixar ja que en general, i en el cas del joc implementat, els jocs 2D no acostumen a jugar gaire amb la profunditat, i per tant tots els actors es trobaran entre aquests dos plans de l'eix Z.

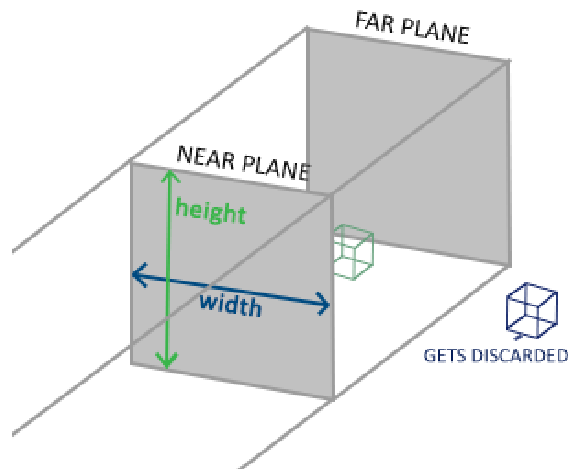

Figura 4.2: Imatge de la zona de visió d'una càmera ortogràfica i la visibilitat de dos objectes.

Càmera perspectiva
Per les càmeres perspectives teníem varies possibilitats:

- Simplificar la zona de visió a una similar a la càmera ortogràfica i aplicar el mateix mètode que a l'anterior.
- Comprovar els vèrtex contra les 4 línies exteriors dels eixos X i Y de la càmera.
- Comprovar la visibilitat dels actors contra l'àrea de visió de la càmera perspectiva.
- Solucions més complexes, com per exemple, emprar arbres de quatre fills per tal de determinar les zones visibles de l'escena.

De totes aquestes solucions al final ens vam decantar per comprovar la visibilitat dels actors contra l'àrea de visió de la càmera perspectiva, el qual es tracta d'una forma tridimensional que es coneix com a *frustum* (figura 4.3), i per tant, a aquesta tècnica se la anomena *frustum culling*[5].

El primer de tot és calcular la forma d'aquest frustum, per això començam calculant l'altura y amplada dels plans *near* i *far*. Aquest càlcul és duu a terme quan s'inicialitza la càmera perspectiva, emprant la mateixa informació que passam a les funcions de OpenGL per crear la càmera.

```
Hnear = 2 * tan(fov / 2) * nearDist;
Wnear = Hnear * ratio;

Hfar = 2 * tan(fov / 2) * farDist;
Wfar = Hfar * ratio;
```

H representa l'alçada i W l'ample, i això es calcula per als dos plans, la informació que rebem es el *fov* (field of view) o l'angle de visio de la càmera, la distancia de la

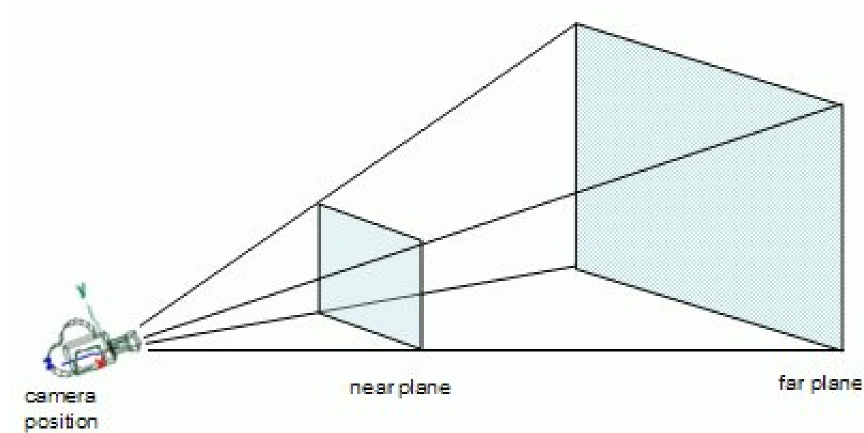

Figura 4.3: Forma de la zona de visió d'una càmera perspectiva, anomenada frustum.

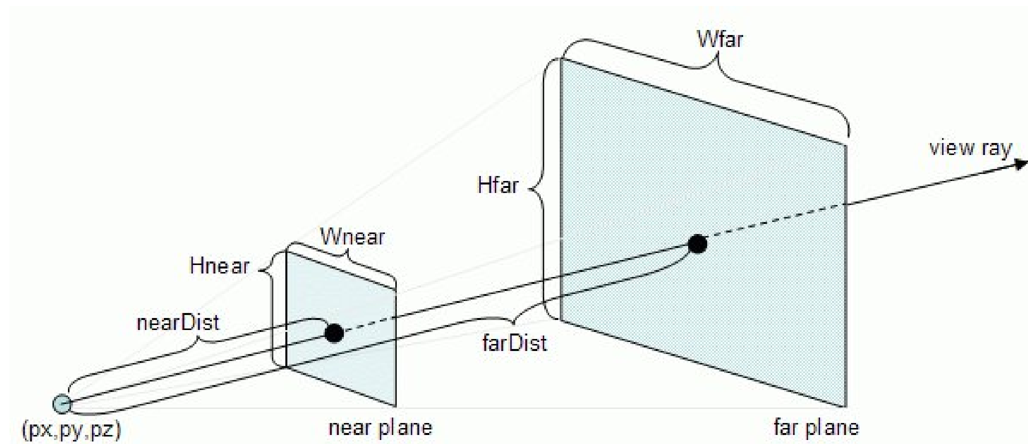

Figura 4.4: Càlcul de l'ample i l'alt dels plans *near* i *far*.

càmera als dos plans o nearDist i farDist, i el *aspect ratio* de la càmera (la relació entre l'ample i l'alt de la projecció de la càmera). La figura 4.4 il·lustra aquest càlcul.

Ara haurem de calcular els eixos de la càmera i, a partir d'aquests, els sis plans que defineixen el *frustum*, que s'hauran de re-calcular cada cop que la càmera es mogui o roti. Per definir un pla tan sols necessitam un punt que pertanyi al pla i la normal al pla. Per tant, el càlcul dels sis plans del *frustum* serà el següent:

```
Z = eye - look;
Z.normalize();

X = up * Z;
X.normalize();

Y = Z * X;
```

```
//Centres dels plans near i far.
nc = eye - Z * nearD;
fc = eye - Z * farD;

//L'eix Z s'empra com a normal dels plans near i far.
pl[NEAR].assignarNormalIPunt(-Z,nc);
pl[FAR].assignarNormalIPunt(Z,fc);

aux = (nc + Y*nh) - eye;
aux.normalize();
normal = aux * X;
pl[TOP].assignarNormalIPunt(normal,nc+Y*nh);

aux = (nc - Y*nh) - eye;
aux.normalize();
normal = X * aux;
pl[BOTTOM].assignarNormalIPunt(normal,nc-Y*nh);

aux = (nc - X*nw) - eye;
aux.normalize();
normal = aux * Y;
pl[LEFT].assignarNormalIPunt(normal,nc-X*nw);

aux = (nc + X*nw) - eye;
aux.normalize();
normal = Y * aux;
pl[RIGHT].assignarNormalIPunt(normal,nc+X*nw);
```

Als càlculs anteriors falta aclarir que eye, look i up són els tres vectors que defineixen la càmera, amb els noms usuals, i que nh i nw són l'ample i l'alt del pla *near* que hem calculat prèviament quan es defineix la càmera.

Així tendrem finalment perfectament definida la zona de visió de la càmera, o el seu *frustum*. Ara tan sols ens queda exposar els càlculs que hem de realitzar per tal de comprovar si una determinada *bounding box* es troba dins o fora de l'àrea de visió de la càmera.

Alhora de definir els plans els hem definit de tal manera que les seves normals apuntin cap a l'interior del *frustum* de visió, així per conèixer si un objecte es troba dins de l'àrea de visió o no tan sols haurem de computar la distancia amb signe de l'objecte al pla. Si l'objecte es troba al costat del pla al que apunta la normal llavors la distancia al pla serà positiva, sinó la distancia serà negativa. Per tant, si un objecte es troba a una distancia positiva dels sis plans llavors podem afirmar que aquest es troba dins del *frustum*.

A la figura 4.5 podem observar 3 objectes, el verd es troba dins el *frustum*, el groc es troba parcialment dins el *frustum* i el taronja es troba a l'exterior. De la figura groga es pot observar que tots els seus punts es troben a l'exterior del *frustum*, però encara així

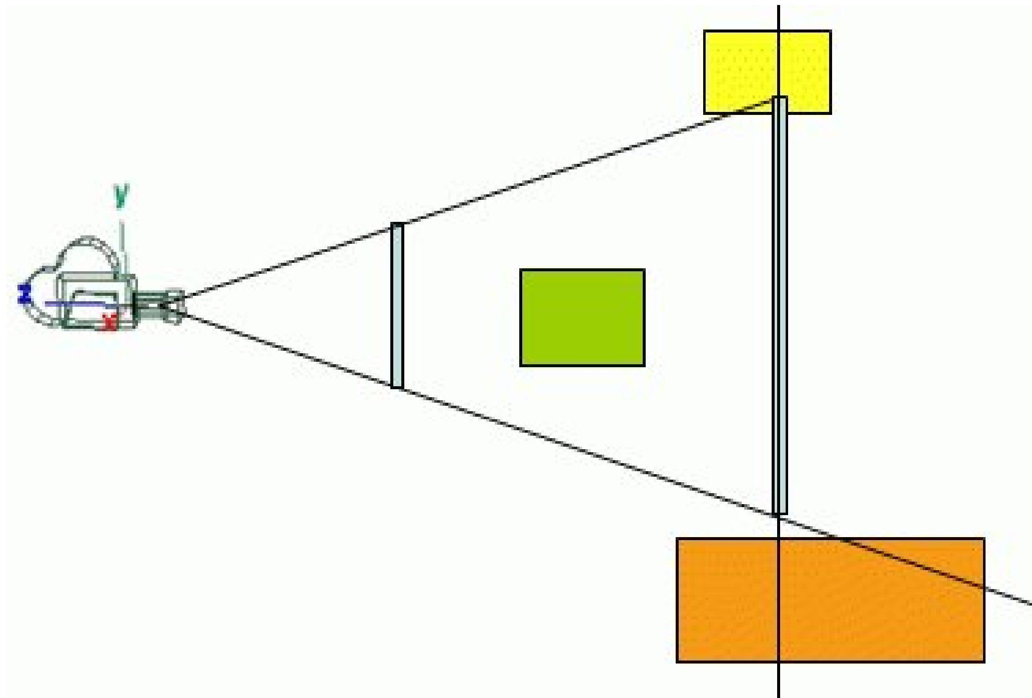

Figura 4.5: Visibilitat de diversos objectes per la càmera i falsos positius.

aquesta es troba parcialment dins de la zona de visió, i per tant no pot ser descartada.

Per tant, per solucionar aquest problema haurem de tenir en compte que per descartar una caixa, tan sols podem descartar-la si tots els seus punts es troben fora d'un mateix pla. Així en aquest cas no descartaríem l'objecte groc. Però aquesta solució té un efecte secundari, i és que la caixa taronja, encara que es troba completament fora del *frustum* no es descartarà i donarà un fals positiu. Però per tal de no incrementar excessivament el cost de computar la visibilitat aquests objectes es deixaran com a positius, ja que és menor el cost de renderitzar-los que no el de calcular si realment són a dins del *frustum*.

Per tant, el codi que ens quedarà per tal de comprovar si un objecte és o no dins la zona de visió de la càmera serà similar a aquest:

```
int out = 0, in = 0;

//Per cada pla.
for (int i = 0; i < 6; i++) {
    out=0; in=0;
    //Per cada cantó de la caixa que comprovam.
    for (int j = 0; j < 8 && (in == 0 || out == 0);
       j++) {
        //Si el cantó de la caixa es dins o forà del
           pla.
```

```
            if (pl[i].distance(bb.getVertex(j)) < 0)
                out++;
            else
                in++;
        }
        //Si tots els cantons són fora.
        if (in == 0) return false;
    }
    return true;
```

I per tant tan sols considerarem fora del *frustum* els objectes que tenguin tots els cantons del seu *bounding box* fora d'algun dels sis plans que delimiten el *frustum*.

4.3 Parallax Scrolling

Per al sistema de *parallax scrolling*, un cop conegut el que era, el que vaig tenir que decidir principalment va ser com implementar el sistema de capes.

Per a això em vaig trobar amb dues opcions:

1. Dibuixar les diverses capes de forma ordenada, de la més llunyana a la més propera, i totes abans de dibuixar cap altre component de l'escena, ja que sinó aquestes apareixerien per sobre, ocultant segurament la majoria d'elements de l'escena.

2. Emprar el buffer de profunditat de OpenGL per tal de dibuixar les capes a una profunditat concreta que ens indiqui l'usuari del motor. Així també es podrien dibuixar algunes de les capes del *parallax* per sobre dels altres elements, no com al punt anterior.

Finalment em vaig decantar per la primera opció, ja que generalment no sols voler capes de *parallax* a la part frontal de l'escena, i així es simplifica el seu ús al motor, simplement has de definir l'ordre en que vols les capes i aquestes ja s'imprimeixen bé al fons, no t'has de preocupar de a quina profunditat sobre l'eix de les Z es troben els objectes de l'escena, de si l'eix de les Z va en direcció positiva o negativa cap a la càmera, etc.

Així que el primer que vaig fer per començar a montar el sistema per permetre l'efecte de *parallax scrolling* al nostre motor va ser estendre el funcionament del component que en aquell moment tenia el motor, que era una classe anomenada *Background*. Així que vaig heretar aquesta classe per crear la classe *Background Scroller*, de la qual a continuació en mostraré la declaració:

```
public class BackgroundScroller extends Background {
    private float factor;
    private float preCamX;
    private float scrX = 0;
    private float scrWidth = 0.2f;
```

```
    private int order;
}
```

Bàsicament aquests cinc atributs són els únics que es necessitaven per aplicar aquest efecte. Cal dir abans de res més que tan sols s'ha aplicat l'efecte de *parallax* sobre l'eix de les X, però que modificar-ho per fer el mateix efecte sobre l'eix de les Y seria trivial amb el codi implementat.

Els atributs afegits serveixen pel següent:

- **factor:** o la velocitat a la qual volem que es desplaci aquesta capa respecte del desplaçament de la càmera.
- **preCamX:** posició prèvia de la càmera a l'eix de les X, per comprovar si s'ha desplaçat i en quin sentit s'ha desplaçat aquesta.
- **scrX:** variable que guarda la posició on es troba aquesta capa.
- **scrWidth:** tamany relatiu respecte a l'ample de la capa que volem que sigui visible en tot moment per pantalla, és a dir, si és 0.2 vol dir que veurem un 20% del tamany total de la capa a la pantalla.
- **order:** un enter per determinar la posició de cada una de les possibles capes respecte de les altres.

Amb aquesta nova classe ja podríem definir capes mòbils amb el motor, ara fa falta que es puguin declarar aquestes a una escena, per això varem haver d'ampliar el *parser* de l'escena amb un nou tipus de objecte, el *background-scroller*, que requeria donar una imatge per la capa (que és el que es dibuixarà per pantalla), el factor, el tamany relatiu respecte a l'ample de la imatge i l'ordre respecte de les altres capes.

Ara tan sols falta encarregar-se del pintat de totes aquestes noves capes, per això es va definir a la classe *Object2D*, que era la que s'encarregava de pintar el background, un mètode anomenat *drawBackground*, que tan sols s'encarrega d'ordenar totes les capes pel seu atribut ordre, i a cridar al mètode de pintat de cada una d'aquestes en l'ordre correcte, dibuixant primer les capes més llunyanes i avançant fins a les més properes.

I l'únic que va quedar per definir va ser el mètode de dibuixat de les capes. Aquest és bastant similar al del *background* que hi havia prèviament, però tenint en compte que tan sols es dibuixa una secció del background i calculant aquesta secció respecte al moviment de la càmera.

Per a això cal definir a OpenGL que s'ha de repetir sobre l'eix de les X, per tal de que aquestes textures puguin fer cicles sense cap interrupció. Això es fa amb la següent cridada:

```
GLES20.glTexParameteri(GLES20.GL_TEXTURE_2D,
    GLES20.GL_TEXTURE_WRAP_S, GLES20.GL_REPEAT);
```

Això defineix que per a l'actual textura carregada, aquesta es repeteixi sobre l'eix de les X. Un cop això està definit faltava activar la transparència de OpenGL (per a poder veure les capes inferiors) i crear un *shader* que tengués en compte aquest valor de transparència. Per el primer basta amb les següents instruccions:

```
GLES20.glEnable(GLES20.GL_BLEND);
GLES20.glBlendFunc(GLES20.GL_SRC_ALPHA,
   GLES20.GL_ONE_MINUS_SRC_ALPHA);
```

I el *shader* creat és el següent:

```
#version 100

precision mediump float;

uniform sampler2D u_Texture;
uniform vec4 a_Color;

varying vec2 v_TexCoordinate;

void main() {
    if (a_Color.a < 0.1) discard;
    gl_FragColor = texture2D(u_Texture, v_TexCoordinate)
       * a_Color;
}
```

És un *shader* molt senzill, només s'ha de tenir en compte que si un píxel té un nivell de transparència molt baix s'ha de descartar.

I amb això ja podem generar backgrounds molt més riques amb el nostre MV i emprar l'efecte de *parallax scrolling* per dotar a un joc 2D que creem d'una certa sensació de profunditat.

4.4 Anti Aliasing

Com hem exposat a la secció 3.4 per aquest punt del treball havíem d'aplicar una tècnica de anti-aliasing al nostre MV per tal que es pogués reduir l'efecte d'aliasing dels jocs a temps real.

L'elecció de quina tècnica emprar va ser molt senzilla en aquest punt, ja que els jocs d'Android no poden exigir gaire rendiment, perque es solen haver d'executar sobre *smartphones* que, per norma general, no solen tenir gaire potència de processament. Per tant vaig decidir implementar la tècnica menys costosa computacionalment, el FXAA. També cal senyalar que aquesta és la tècnica de post-AA més senzilla, la qual cosa també va facilitar la decisió.

L'algoritme de FXAA fa el següent[6], il·lustrat a la figura 4.6 d'esquerra a dreta i de dalt a abaix:

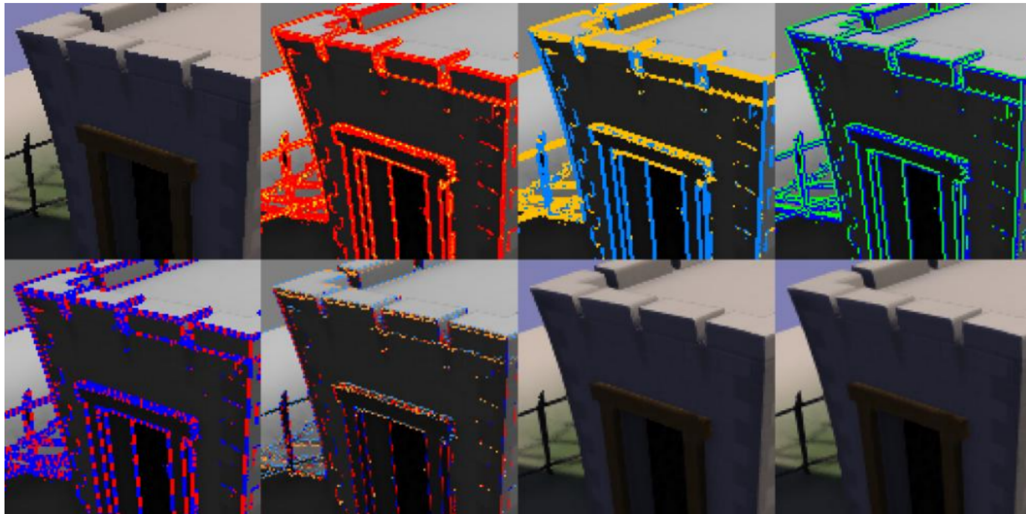

Figura 4.6: Diverses passes de l'algoritme FXAA.

1. FXAA rep com a entrades la informació de color rgb no lineal, que converteix internament en una estimació escalar de la luminància per a la lògica del *shader*.

2. Es comprova el contrast local per tal d'evitar processar tot allò que no siguin arestes. Les arestes detectades es mostren en vermell, canviant fins a groc per representar la quantitat detectada de subpíxel aliasing.

3. Els píxels que passen la prova de contrast local es classifiquen llavors en horitzontals, en daurat a la imatge, o verticals, en blau a la imatge.

4. En aquest punt, donada la orientació de l'aresta, la parella de píxels amb el major contrast es selecciona, en blau/verd a la imatge.

5. L'algoritme llavors cerca el fi de l'aresta en ambdues direccions sobre l'aresta, cercant un canvi significatiu en la mitja de luminància de les parelles de píxels amb alt contrast sobre l'aresta. Es marquen ambdues direccions, la negativa en vermell i la positiva en blau.

6. Un cop trobats els finals de l'aresta, la posició del píxel sobre aquesta es transforma en un desplaçament de 90º perpendicular a l'aresta per tal de reduir l'aliasing (en vermell/blau per al desplaçaments horitzontals i daurat/celeste per als desplaçaments verticals.

7. Ara la textura d'entrada és re-mostreja donat aquest desplaçament.

8. Finalment, es mescla un filtre passabaix (*lowpass filter*) depenent de la quantitat detectada de subpíxel aliasing.

Així, amb aquestes 8 passes, l'algoritme FXAA produeix molt bons resultats, i té dos grans avantatges:

1. Suavitza les arestes a tots els píxels de la pantalla, incloent aquells dins de textures *alpha-blended* (alpha-blending és la combinació del canal de alpha o transparència amb altres capes d'una imatge, per tal d'aconseguir translucidesa) i aquells resultants d'efectes de *shader*.

2. És molt ràpid. Algunes de les primeres versions de FXAA eren el doble de ràpides que aplicar 4x MSAA.

Així que començam amb la nostra implementació. El primer va ser implementar un *vertex* i un *fragment* o *píxel shader* que implementassin aquest algoritme. Així que vaig començar fent una còpia d'una de les classes que aplicaven algun efecte de post-processat del motor (ja que FXAA s'aplica com a un efecte de post-processat sobre l'escena renderitzada), com per exemple la classe *Sepia*.

Aquesta classe ja tenia implementat el codi per passar als *shaders* la escena renderitzada que es troba guardada a un buffer i el codi per dibuixar aquesta escena sobre un quadrat que ocupa tota la pantalla. Així doncs, l'únic que hauria de canviar d'aquesta classe seria la informació que es passava al *shaders*. També vaig haver d'afegir codi a la classe *ShaderManager* per tal de poder aplicar el nou *shader*.

Un cop preparat tot això vaig crear ambdos *shaders* i els vaig anomenar FXAA.vert i FXAA.frag respectivament. Primer explicarem el *vertex shader* que és el més senzill dels dos.

```
#version 100
precision highp float;

uniform vec2 u_Resolution;

attribute vec4 a_Position;
attribute vec2 a_TexCoord;

varying vec4 v_Position;
varying vec2 v_TexCoord;

varying vec2 v_rgbNW;
varying vec2 v_rgbNE;
varying vec2 v_rgbSW;
varying vec2 v_rgbSE;
varying vec2 v_rgbM;

void texcoords(vec2 fragCoord, vec2 resolution,
            out vec2 v_rgbNW, out vec2 v_rgbNE,
            out vec2 v_rgbSW, out vec2 v_rgbSE,
            out vec2 v_rgbM) {
    vec2 inverseVP = 1.0 / resolution.xy;
    v_rgbNW = (fragCoord + vec2(-1.0, -1.0)) * inverseVP;
    v_rgbNE = (fragCoord + vec2(1.0, -1.0)) * inverseVP;
```

```
    v_rgbSW = (fragCoord + vec2(-1.0, 1.0)) * inverseVP;
    v_rgbSE = (fragCoord + vec2(1.0, 1.0)) * inverseVP;
    v_rgbM = vec2(fragCoord * inverseVP);
}

void main(void) {
    vec2 fragCoord = a_TexCoord * u_Resolution;
    texcoords(fragCoord, u_Resolution, v_rgbNW, v_rgbNE,
       v_rgbSW, v_rgbSE, v_rgbM);

    v_TexCoord = a_TexCoord;
    gl_Position = a_Position;
}
```

El que fa aquest *shader* és calcular les coordenades de la textura que es passaran al *fragment shader* i computa a més les coordenades de la textura dels píxels diagonals respecte al píxel que es tractarà al següent *shader*, això ho feim aquí per tal d'optimitzar el rendiment del *fragment shader*. La posició dels vèrtex simplement es passa com a atribut i directament a OpenGL via la variable predefinida gl_Position.

I ara passem al *fragment shader*, que és on es duu a terme l'algoritme de FXAA.

```
#version 100
precision highp float;

uniform sampler2D tex0;
uniform vec2 u_Resolution;

const float FXAA_SPAN_MAX = 8.0;
const float FXAA_REDUCE_MUL = 1.0/8.0;
const float FXAA_REDUCE_MIN = 1.0/128.0;

varying vec2 v_TexCoord;

varying vec2 v_rgbNW;
varying vec2 v_rgbNE;
varying vec2 v_rgbSW;
varying vec2 v_rgbSE;
varying vec2 v_rgbM;

void main() {
    //Calculam el color dels píxels diagonals respecte
       l'actual i l'actual mateix.
    vec3 rgbNW = texture2D(tex0, v_rgbNW).xyz;
    vec3 rgbNE = texture2D(tex0, v_rgbNE).xyz;
    vec3 rgbSW = texture2D(tex0, v_rgbSW).xyz;
    vec3 rgbSE = texture2D(tex0, v_rgbSE).xyz;
    vec4 texColor = texture2D(tex0, v_rgbM);
```

```
vec3 rgbM  = texColor.xyz;

//Calculam la luminancia de cada un d'aquests píxels.
vec3 luma = vec3(0.299, 0.587, 0.114);
float lumaNW = dot(rgbNW, luma);
float lumaNE = dot(rgbNE, luma);
float lumaSW = dot(rgbSW, luma);
float lumaSE = dot(rgbSE, luma);
float lumaM  = dot(rgbM,  luma);

//Calculam la luminància mínima i màxima entre totes
   aquestes.
float lumaMin = min(lumaM, min(min(lumaNW, lumaNE),
   min(lumaSW, lumaSE)));
float lumaMax = max(lumaM, max(max(lumaNW, lumaNE),
   max(lumaSW, lumaSE)));

mediump vec2 dir;
dir.x = -((lumaNW + lumaNE) - (lumaSW + lumaSE));
dir.y =  ((lumaNW + lumaSW) - (lumaNE + lumaSE));

float dirReduce = max((lumaNW + lumaNE + lumaSW +
   lumaSE) * (0.25 * FXAA_REDUCE_MUL),
                      FXAA_REDUCE_MIN);


mediump vec2 inverseVP = vec2(1.0 / u_Resolution.x,
   1.0 / u_Resolution.y);

float rcpDirMin = 1.0 / (min(abs(dir.x), abs(dir.y))
   + dirReduce);
dir = min(vec2(FXAA_SPAN_MAX, FXAA_SPAN_MAX),
          max(vec2(-FXAA_SPAN_MAX, -FXAA_SPAN_MAX),
          dir * rcpDirMin)) * inverseVP;

vec2 fragCoord = v_TexCoord * u_Resolution;
vec3 rgbA = 0.5 * (
    texture2D(tex0, fragCoord * inverseVP + dir *
       (1.0 / 3.0 - 0.5)).xyz +
    texture2D(tex0, fragCoord * inverseVP + dir *
       (2.0 / 3.0 - 0.5)).xyz);
vec3 rgbB = rgbA * 0.5 + 0.25 * (
    texture2D(tex0, fragCoord * inverseVP + dir *
       -0.5).xyz +
    texture2D(tex0, fragCoord * inverseVP + dir *
       0.5).xyz);
```

Figura 4.7: Comparació d'un fragment de l'escena sense AA (esquerra) o emprant FXAA (dreta). Fotografies del joc amb molt de zoom per poder apreciar l'efecte.

```
    float lumaB = dot(rgbB, luma);

    vec4 color;
    if ((lumaB < lumaMin) || (lumaB > lumaMax)) color =
        vec4(rgbA, texColor.a);
    else color = vec4(rgbB, texColor.a);

    gl_FragColor = color;
}
```

Aquesta implementació es basa en una implementació[7] de la versió 2 de FXAA desenvolupada per Timothy Lottes, creador de l'algoritme. Hi ha algunes modificacions, sobretot en la instrucció d'agafar les dades de la textura, ja que a la implementació original s'emprava la funció *texture2DLod*, però a la versió de GLSL que inclou OpenGL ES aquesta funció no és suportada, per tant s'ha hagut d'implementar emprant tan sols la funció *texture2D*.

Un cop l'algoritme va ser funcional (i per poder veure realment el seu efecte, ja que aquest efecte sol ser bastant subtil, sobretot si no es sap on mirar) es va afegir un nou botó per al jugador a l'escena de joc, que activa i desactiva aquest efecte. A la figura 4.7 podem observar els resultats, s'observen sobretot a la part inferior de la plataforma (la línia que separa la plataforma de la lluna de fons) on es veu molt més suau quan s'està aplicant l'algoritme FXAA. Cal mencionar que encara que aquest és l'algoritme de AA menys costos, es nota una bona baixada de FPS (Frames Per Second) quan l'activam.

CAPÍTOL 5

Videojoc d'Exemple

Per tal de donar una mostra de l'ús del nostre MV, amb totes les noves funcionalitats afegides, hem desenvolupat un petit videojoc d'exemple que funciona sobre aquest motor.

Aquest capítol consistirà en una explicació d'aquest joc i de les diverses funcionalitats que desenvolupades que s'han aplicat aquí.

5.1 Menús del joc

5.1.1 Menú principal

Tan sols obrir el joc se'ns presenta la pantalla que veim a la figura 5.1. Aquesta pantalla es un senzill menú de joc que conté els següents elements:

- **Nom del joc:** el primer element visible, que es troba per sobre de les opcions disponibles és el nom del joc.
- **Play:** botó que inicia el joc, aquest carregarà l'escena del joc.
- **About:** botó que carregarà una petita escena amb informació del creador del joc.
- **Exit:** botó per tancar aquesta aplicació.
- **Fons de l'escena:** el fons, que està implementat mitjançant el sistema de capes que hem creat per poder emprar l'efecte de *parallax scrolling*. En aquest cas però, cap capa té moviment.

5.1.2 About

Quan des d'el menú principal es prem el botó de "about", llavors es carrega l'escena mostrada a la figura 5.2. Aquesta escena mostra informació sobre els creadors del joc i

Figura 5.1: Menú principal del joc.

Figura 5.2: Escena secundaria amb informació sobre el joc i el seu creador.

permet tornar al menú principal sempre que es vulgui. Així, igual que a l'escena anterior, aquesta escena consta de:

- **Nom del joc:** aquesta escena manté el nom del joc, ja que tampoc hi havia gaire més informació que compartir.
- **Text:** el text central ens indica els creadors del joc.
- **Back:** botó per tornar al menú principal.
- **Fons de l'escena:** igual que al menú principal, aquest fons consta de varies capes, totes elles sense moviment.

5.2 Joc

5.2.1 Protagonista

El protagonista d'aquest joc és un petit pollastre que intenta evitar tots els perills del món. L'única habilitat que té és la de fer uns grans bots.

5.2.2 Controls

El joc inclou quatre botons i un joystick. Aquests tenen les següents funcions:

- **Joystick:** permet el moviment del personatge.
- **Botar:** el botó inferior dret, és el que fa que el nostre pollastre boti.
- **Canvi de càmera:** el botó superior dret, canvia entre una càmera ortogràfica i la càmera perspectiva. Ambdues segueixen al personatge principal i són les implementades que s'han explicat a l'apartat 4.1.
- **FXAA:** botó que activa l'efecte de post-processat de anti-aliasing, que aplica l'algoritme implementat a la secció 4.4 FXAA.
- **Back:** botó que sempre ens permet tornar al menú principal del joc.

5.2.3 Objectiu

L'objectiu d'aquest petit joc es tan sols arribar fins a una bandera que hi ha a l'escena. Un cop el personatge principal fa contacte amb aquesta es considera que s'ha guanyat el joc.

Però arribar a la bandera no serà fàcil, ja que pel camí hi ha varis enemics i obstacles que intentaran que no puguis arribar a la meta.

El joc es pot dividir en dues petites zones que es desgranaran a les dues pròximes seccions.

Figura 5.3: Primera zona del joc, on es troben els enemics.

5.2.4 Primera zona

A aquesta zona, el jugador haurà d'evitar als enemics que s'aniran acostant cap a la seva posició lentament. Per passar-los té dues opcions, o bé botar per sobre i passar d'ells o bé els hi pot botar sobre el cap, la qual cosa els elimina de l'escena durant la resta d'aquesta partida.

Un cop sortejats els enemics, el jugador arribarà a la segona zona.

5.2.5 Segona zona

Aquesta zona consisteix en esquivar bales que plouen desde dalt, amb la dificultat afegida de que aquestes tenen un color molt similar a zones del fons, per tant hi ha moments on són molt difícils de divisar.

La bona noticia es que un cop passada aquesta petita zona ja s'arriba a la bandera, i per tant, un cop s'ha passat aquesta zona, un ha guanyat el joc.

5.2.6 Fons del joc

El fons del joc empra l'efecte de *parallax scrolling* que hem desenvolupat. Consta de set capes diferents i cada una d'elles es mou a diferent velocitat per tal de donar la impressió de profunditat.

La capa més posterior, o la que conté la lluna, és la que es mou més lentament, mentre que la capa més propera, el terra blanquinós, és la que es mou a major velocitat.

Figura 5.4: Segona zona del joc, on hi ha que esquivar.

Figura 5.5: Pantalla de derrota.

Totes aquestes capes es van movent, cada una a la seva velocitat, cada cop que es mou la càmera.

5.2.7 Derrota

Si ens topam amb algun enemic (sense botar-li a sobre) o col·lisionam amb alguna de les bales, llavors hem perdut la partida.

Figura 5.6: Pantalla de victòria.

Quan perdem se'ns mostra la imatge de la figura 5.5 i tan sols se'ns permet tornar al menú principal.

5.2.8 Victòria

En canvi, si aconseguim arribar a la bandera final veurem la imatge de la figura 5.6. Encara que les accions que podrem realitzar a partir d'aquí seran les mateixes que a la derrota, tornar al menú principal.

CAPÍTOL 6

CONCLUSIONS

6.1 Opinió Personal

El TFG ha estat una bona proba final per posar en pràctica molts dels coneixements que he adquirit durant aquests anys a la carrera.

M'ha agradat molt, de principi a fi, el tema del meu TFG, ja que m'ha permès aprendre moltes coses de com són els MV internament, i conseqüentment com estan fets molts de videojocs.

Ha estat molt satisfactori personalment poder arribar al final d'aquest repte i veure el motor funcionant amb totes les noves funcionalitats. El desenvolupament del joc final ha estat la part més divertida, a més d'haver estat relativament molt simple per el fet de disposar del motor, el qual ha fet molt trivial molts dels aspectes de la creació del joc (donant-nos ja implementades el sistema de físiques, càmeres mòbils, suport per a models 3D i textures, suport per a múltiples capes de fons, etc).

En general, ha estat una experiència molt enriquidora personal i professionalment, i no me'n he penedit en cap moment del TFG que vaig triar.

6.2 Resultats Obtinguts

Totes les millores realitzades al motor són funcions molt útils que ara tendrà aquest. Les tornarem a exposar aquí indicant què impliquen aquestes funcions per al motor.

- Ara el motor disposa de càmeres que s'encarreguen de seguir elles mateixes a un actor determinat.

- El scene-culling és una funció que encara que sigui difícil de notar per a nosaltres, farà que el motor tingui un millor rendiment. Ara ja no pintarà absolutament tots els objectes que es trobin a una escena.
- Es poden crear fons d'escenes 2D molt més rics gràcies al sistema de capes del *parallax scrolling* i dotar a jocs 2D fets amb el motor d'aquesta sensació de profunditat que ens dona la tècnica.
- I finalment amb el FXAA es disposa d'una bona solució a temps real contra els problemes de aliasing, que encara que té un cert cost de rendiment, millora gratament els gràfics que aconseguim per pantalla.

6.3 Possibles Millores

Encara que ja som varis els alumnes que hem treballat en aquest MV, i, encara que ja té moltes coses, encara hi ha moltíssimes millores que s'hi podrien realitzar.

Algunes d'aquestes serien:

- **Afegir un sistema d'animació**
- **Suport per a altres tipus de formats 3D:** ara mateix tan sols pot carregar models en format .obj de *WaveFront*.
- **Millorar el motor de so:** que encara es troba en un estat molt bàsic.
- **Implementar altres shaders:** com per exemple per a *normal-mapping* o *specular-mapping*.
- **Aplicar occlussion-culling**
- **Implementar una interficie d'usuari per a l'ús del motor**
- **Desenvolupar eines per a la creació de jocs multijugador amb el motor**

BIBLIOGRAFIA

[1] J. Brightman, “Mobile to overtake pc in 99.6bn dollars global games market,” *gamesindustry.biz*, april 2016. [Online]. Available: http://www.gamesindustry.biz/articles/2016-04-21-mobile-to-overtake-pc-in-usd99-6bn-global-market-newzoo 1.1

[2] “Unity estadistics.” [Online]. Available: https://unity3d.com/es/public-relations 2.2

[3] J. Gregory, *Game Engine Architecture*. Wellesley, Mass: A K Peters, 2009. 2.4

[4] A. Johnson, “Culling Explained,” 2013. [Online]. Available: http://docs.cryengine.com/display/SDKDOC4/Culling+Explained 3.2

[5] “3d tutorials.” [Online]. Available: http://www.lighthouse3d.com/ 4.2

[6] T. Lottes, “FXAA,” february 2009. [Online]. Available: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf 4.4

[7] “FXAA Demo.” [Online]. Available: http://www.geeks3d.com/20110405/fxaa-fast-approximate-anti-aliasing-demo-glsl-opengl-test-radeon-geforce/3/ 4.4

[8] “Opengl es glsl specification,” may 2009. [Online]. Available: https://www.khronos.org/registry/gles/specs/2.0/GLSL_ES_Specification_1.0.17.pdf

[9] “Stack overflow.” [Online]. Available: http://stackoverflow.com/