UdG EPS Escola Politècnica Superior

Memòria de P/TFC de les enginyeries informàtiques

Estudi: Eng. Tècn. Informàtica de Gestió. Pla 2001

Títol: Desenvolupament de prototips (Android/LibGDX i Unity), per a la producció d'un videojoc d'acció lateral amb plataformes (Ouroboros)

Document: Memòria

Alumne: Jaume Calm Serra

Director/Tutor: Gustavo Patow

Departament: Informàtica i Matemàtica Aplicada

Àrea: LSI

Convocatòria: Juny 2014

Índex

Figures

1. Introducció, motivacions, propòsit i objectius del projecte

Actualment la indústria dels videojocs mou més diners que la indústria del cinema i la música junts. Per exemple, als EEUU la música va arribar gairebé als 7 bilions de USD($), el cinema 10.6 i els videojocs amb gairebé 23; tot l'any 2013. En tot el món, els ingressos vas ascendir a més de 70 bilions amb una audiència estimada de 1.231 milions de jugadors.

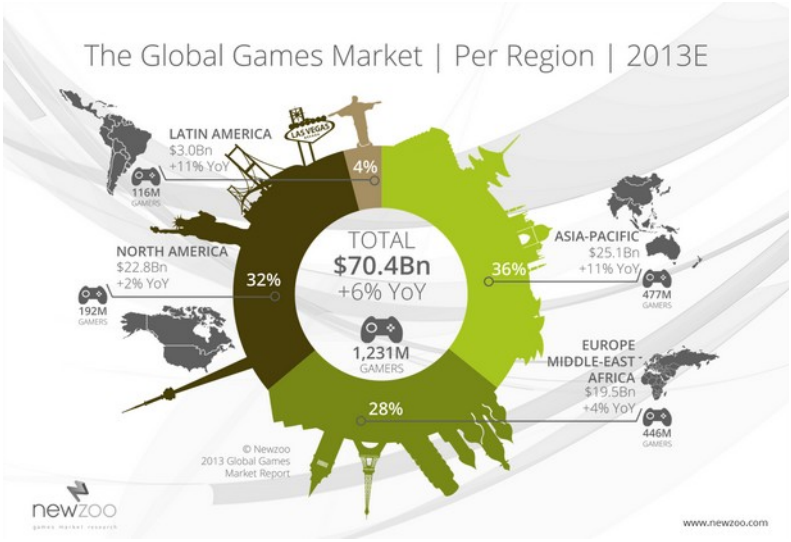


Figura 1: El mercat global de videojocs per regions

Els jocs d'acció són un gènere de videojocs que emfatitzen desafiaments, com per exemple, la coordinació ull-mà i el temps de reacció.

En un joc d'acció, el jugador controla un **avatar** del protagonista. Aquest avatar ha de navegar per un **nivell**, superant obstacles i batallant contra enemics amb diferents atacs. Al final del nivell, o d'una sèrie de nivells, el jugador sovint s'ha d'enfrontar a un gran enemic, típicament anomenat "**Boss**", que és més complexe i desafiant que altres enemics. Els atacs dels enemics i els obstacles redueixen la vida, o el nombre de vides de l'avatar, i el joc s'acaba quan el jugador es queda sense vides, o quan el jugador supera amb èxit tots els nivells del joc.

Aquest gènere inclou diferents subgèneres, i per aquest joc ens fixarem en dos d'ells; els denominats "**Beat 'em up**" i els **jocs de plataformes**.

1.1 Propòsit i objectius

L'objectiu d'aquest projecte és la creació de dos **prototips**, un per **Android** i l'altre per **Unity**, establint les bases per a la producció d'un **videojoc** d'acció lateral (Beat 'em up) amb plataformes (puzles) anomenat "**Ouroboros**".

Android és un sistema operatiu basat en Linux, designat primerament per mòbils tàctils (smartphones) i tabletes. En concret s'utilitzarà el SDK (Software Development Kit) dins de l'entorn de programació **Eclipse** amb llenguatge **Java**, i les bases d'un **framework** anomenat **LibGDX**.

Unity, en canvi, és un motor de videojocs multi-plataforma amb un entorn de desenvolupament integrat, del que nosaltres utilitzarem la versió en **Javascript.**

Es volen explorar les dues plataformes per tal d'esbrinar quina de les dues vies és la més idònia de cares a la producció final d'un joc.

1.2 Beat 'em up

Aquesta categoria de jocs presenten combats cos a cos entre el protagonista i un **nombre inusualment gran d'enemics** més dèbils. Tradicionalment prenen lloc en escenaris 2D i en els que l'acció es desenvolupa desplaçant-se lateralment.

Figura 2: Knights of the Round

Aquesta serà la categoria predominant del joc que volem desenvolupar, i l'objectiu principal és oferir una jugabilitat accessible per a tothom a l'hora de superar els reptes que es proposin dins els nivells, però suficientment profunda/complexa pels jugadors més experts i exigents.

1.3 Jocs de plataformes

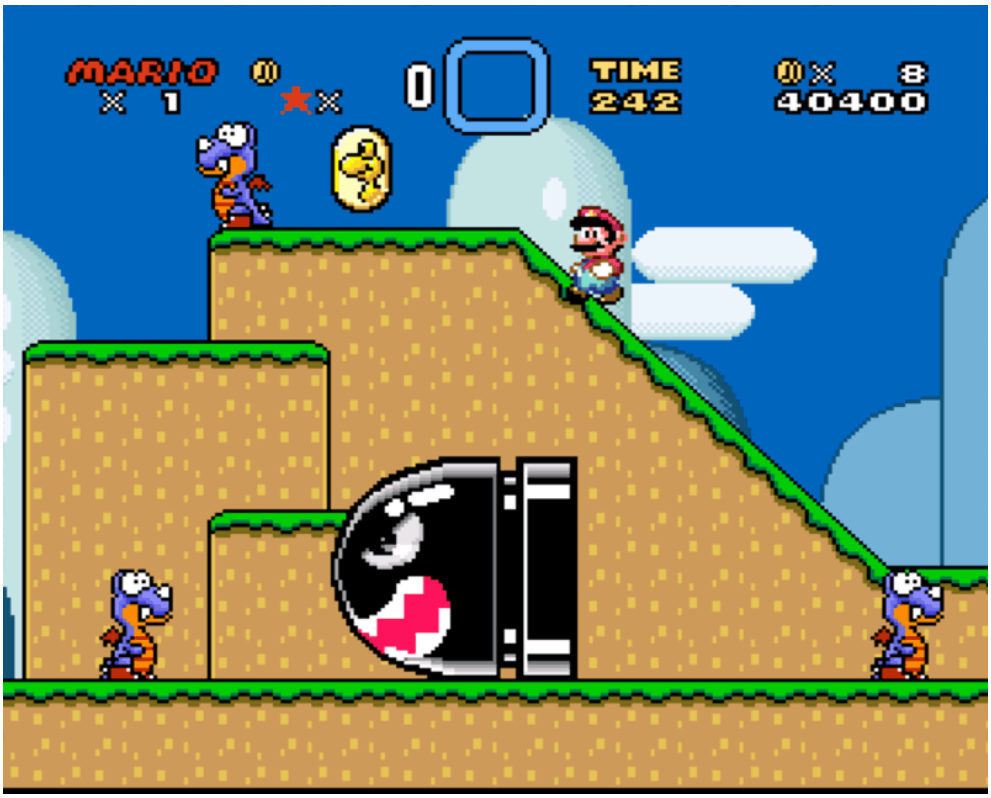


Figura 3: Super Mario World

Els jocs de plataformes estan basats en guiar un avatar saltant per diferents plataformes, sobre obstacles, o ambdós per avançar en el joc. Aquests desafiaments es poden anomenar com “**puzles de salt**” i es poden presentar moltes situacions depenent de les habilitats i/o característiques de l'avatar.

Aquesta serà la categoria secundaria dins el joc, i es volen incorporar puzles de salt que facin pensar al jugador per compaginar les etapes d'acció contra enemics.

1.4 Motivació Personal

Des de que era molt petit sempre m'han fascinat els videojocs, segurament pel fet de poder explorar i controlar entitats d'un món imaginari ple d'elements atípics i sorprenents.

Ja més endavant m'han continuat fascinant, sobretot pel fet que agrupen les dues professions o camps que més m'han agradat: la informàtica i el disseny artístic.

El meu objectiu, algun dia, és de poder crear la meva pròpia empresa i poder-me guanyar la vida dissenyant videojocs.

Figura 4: Logo de l'estudi de videojocs que s'està creant

1.5 Estructura del document

Aquest document esta estructurat en 15 capítols, que són els següents:

1. **Introducció, motivacions, propòsit i objectius del projecte.** En aquest capítol s'explicarà el perquè del desenvolupament d'aquest projecte, quins són els objectius proposats i com s'ha organitzat el desenvolupament.
2. **Estudi de viabilitat.** En aquest capítol es justifiquen els paràmetres que fan possible el desenvolupament del projecte.
3. **Metodologia.** Aquest capítol conté una explicació de la metodologia utilitzada.
4. **Planificació.** En aquesta etapa es defineix l'estratègia seguida per arribar als objectius plantejats.
5. **Marc de treball i conceptes previs.** En aquest capítol es descriuen els diversos aspectes relacionats amb el desenvolupament general del projecte, que ajudaran a

entendre millor els següents capítols.

6. **Requisits del sistema.** En aquest capítol es defineixen els requisits dels software, els quals recullen, a grans trets, els objectius de l'aplicació juntament amb les funcionalitats que es volen obtenir.
7. **Estudis i decisions.** Aquesta secció conté una descripció de les eines utilitzades i l'ús que se'ls hi ha donat, tant de llibreries i motors, com de software.
8. **Anàlisi i disseny del sistema.** Aquest apartat proporciona una comprensió precisa de les necessitats del sistema. És a dir, s'encarrega de la investigació del problema a resoldre, però no es materialitza la solució. Es tradueixen requeriments anomenats en capítols anteriors a un llenguatge més formal.
9. **Implementació i proves.** En aquest capítol es dona a conèixer com s'ha construït l'aplicació, les classes i els mètodes implementats que resulten més significatius per la comprensió del funcionament del videojoc.
10. **Resultats.** En aquest capítol es mostren proves d'execució de l'aplicació per tal de veure'n les funcionalitats finals.
11. **Conclusions.** En aquest apartat s'exposaran les conclusions extretes un cop finalitzat el projecte.
12. **Treball futur.** En aquesta secció s'exposa tot allò que es podria millorar en el videojoc, o ampliar-lo.
13. **Bibliografia.** Aquest capítol conté referències utilitzades per el desenvolupament del projecte.
14. **Annexos.** Aquesta secció conté les definicions dels tecnicismes més freqüentment utilitzats, així com explicacions i experiments no indispensables per la comprensió global del projecte.
15. **Manual d'usuari i/o instal·lació.** Aquesta secció inclou les especificacions del funcionament del producte.

2. Estudi de viabilitat

Per desenvolupar aquest projecte no es requereix d'una gran infraestructura però si de molt de **programari**, un **hardware** adequat i **temps**. Aleshores anem a desglossar poc a poc cada un d'aquests aspectes per finalitzar amb els **costos**.

En tot moment comptarem amb dos escenaris, Android i Unity; així doncs primer hem de fer una diferenciació del que es vol aconseguir a cada una de les plataformes.

El prototip d'Android, amb el suport del framework LibGDX, comptarà amb **gràfics** i animacions en **2D**, típicament anomenats "**sprites**", realitzats a mà amb una tècnica anomenada "**pixel art**", seguint un estil "**retro**".

Figura 5: Personatge de la versió d'Android/LibGDX

El prototip de Unity, per altre banda, comptarà amb **models** i animacions en **3D**, seguint un estil **modern**.

Figura 6: Personage de la versió de Unity

2.1 Programari

Donat que es volen implementar dos prototips diferents, cada un d'ells comptarà amb un conjunt de programari específic. L'únic que comparteixen les dues vies és el sistema operatiu, en aquest cas, **Windows 7** Ultimate Edition.

2.1.1 Android

- Eclipse Indigo
- Android SDK
- GraphicsGale (Free Edition)
- Painter 23

2.1.2 Unity

- Unity
- Creation Kit
- 3DsMax / NifSkope
- FBX Converter
- Photoshop

2.2 Hardware

Ja que s'utilitzaran programes d'edició 3D es necessita un ordinador força potent; tinguen en compte els estàndards actuals, aquesta és la màquina amb la que treballarem:

- Asus Rampage III GENE
- Intel Core i7 960 3.20Ghz Box Socket 1366
- Gigabyte GeForce GTX 560 Ti OC 1024MB GDDR5
- Exceleram Black Sark DDR3 1600 PC3-12800 12GB 3x4GB CL9

2.3 Temps

Aquest projecte consta d'una durada aproximada de 9 mesos en els quals s'estipula que, de mitjana, es treballarien 4h/dia o a mitja jornada.

2.4 Costos

Tinguen en compte tot els elements que hem anomenat fins ara i centrant-nos en els que hi ha una despesa, el cost del projecte seria el següent:

Programari	Microsoft Windows 7 Ultimate	175,00 €
	3DsMax 2012	3.900,00 €
	Photoshop Cs5	1.390,84 €
Hardware	Asus Rampage III GENE	176,27 €
	Intel Core i7 960 3,20Ghz Box Socket 1366	208,47 €
	Gigabyte GeForce GTX 560 Ti OC 1024MB GDDR5	178,81 €
	Exceleram Black Sark DDR3 1600 PC3-12800 12GB 3x4GB CL9	77,92 €
	Seagate Barracuda 7200,12 1TB SATA3	42,37 €
	NZXT H2 Classic Series Negra	76,27 €
	Scythe Ninja 3	31,31 €
	SilverStone Strider Plus ST1000-P 1000W	133,90 €
	Asus DRW-24B3ST Grabadora DVD 24X Negra OEM	22,00 €
Temps	9 Mesos * 30 dies/Mes * 4 hores/dia * 10 €/hora	10.800,00 €
Total		**17.213,16 €**

Figura 7: Taula de costos

No s'han inclòs dins el cost final altres elements com podrien ser l'electricitat o el lloguer d'un local/oficina. Tampoc cap llicència de Unity ja que s'utilitzarà la versió gratuïta.

En total, el desenvolupament d'aquest projecte ens suposaria un cost de **17.213,16 €**.

3. Metodologia

Donat que els videojocs compten amb un gran nivell d'interactivitat entre tots els elements que el formen, s'ha optat per una metodologia que reforça a cada pas el treball fet anteriorment i la descriurem de la següent manera:

1. Escollir la feina a desenvolupar.
2. Decidir quines eines de programació s'utilitzaran.
3. Aprendre el funcionament de les eines i llenguatges de programació escollits.
4. Estructurar la feina en parts segons les funcions que s'hagin de realitzar.
5. Desenvolupar la part corresponent seguint l'ordre d'estructura del treball.
6. Fer comprovacions per confirmar que el funcionament és correcte al finalitzar cada part.
    - Si al fer les comprovacions el resultat no és el desitjat, es tornarà al punt 5 per realitzar els canvis oportuns a l'última part desenvolupada o a les anteriors, si és necessari.
    - Si al fer les comprovacions el resultat és el desitjat, es desenvoluparà la següent part tornant al punt 5. Una vegada s'hagin finalitzat totes les parts amb les respectives comprovacions, s'iniciarà el punt 7.
7. Unir totes les parts desenvolupades i comprovar que el funcionament és correcte.
    - Si al fer les comprovacions el resultat no és l'esperat, es tornarà al punt 5 per realitzar els canvis oportuns a l'última part desenvolupada o a les anteriors, si és necessari.
    - Si al realitzar les comprovacions el resultat és l'esperat, s'iniciarà el punt 8.
8. Generar diferents models d'exemple per comprovar que el funcionament és l'esperat.
    - Si al fer les comprovacions el resultat no és el desitjat, es tornarà al punt 5 per realitzar els canvis oportuns a l'última part o a les anteriors, si és convenient.
    - Si al realitzar les comprovacions el resultat és el desitjat, s'iniciarà el punt 9.
9. Arrodonir la documentació feta al llarg del projecte.

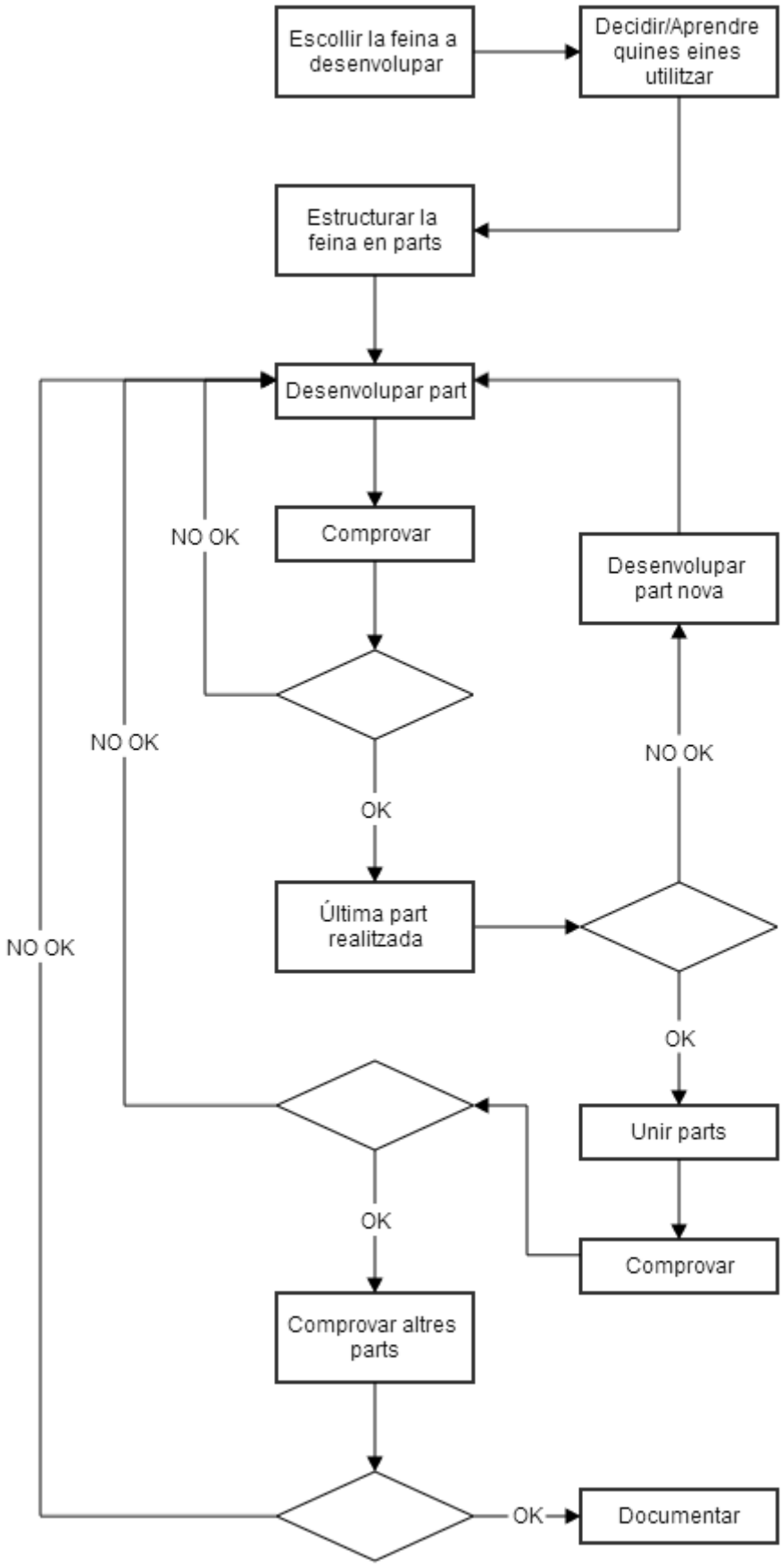


Figura 8: Diagrama de la metodologia a seguir

4. Planificació

Per realitzar la planificació, diferenciarem entre els diferents estadis sobre els que haurem de passar. Així doncs, tindrem les següents feines a desenvolupar:

1. Definició de les mecàniques jugables.
2. Estudi de les plataformes Android / LibGDX i Unity.
3. Estudi del programari per a la creació dels elements visuals.
4. Disseny / Creació dels elements visuals.
5. Disseny / Implementació dels algorismes de col·lisions.
6. Disseny / Implementació dels algorismes de control del personatge.
7. Disseny / Implementació dels algorismes d'intel·ligència artificial dels enemics.
8. Verificació i proves dels algorismes implementats.
9. Elaboració de la documentació.

D'aquesta manera s'estima un mes per a la realització de cada tasca treballant a mitja jornada, podent esser flexible i dedicar més temps a les proves, quan es compleixen els objectius abans d'hora, i treballar més hores quan no s'estiguin assolint els terminis. Per consegüent obtenim un període total, teòric, de 9 mesos (Figura 9).

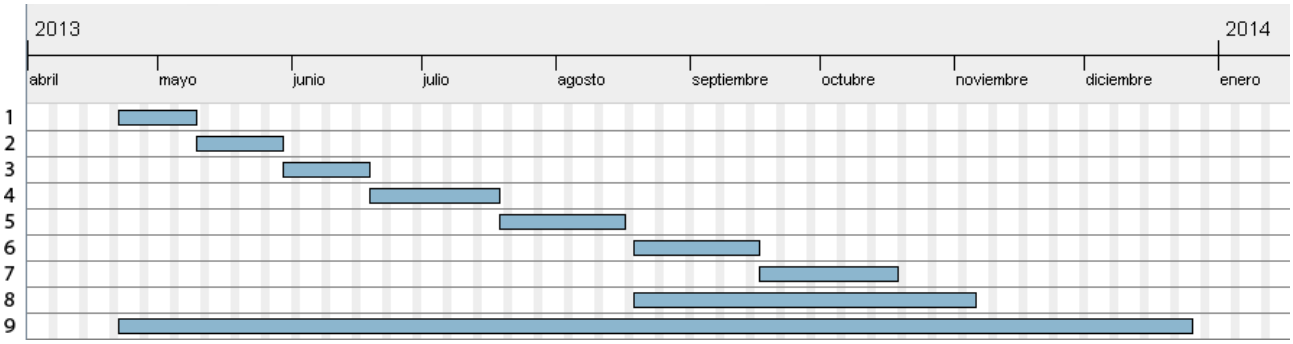


Figura 9: Diagrama de Gantt

4.1 Definició de les mecàniques jugables

La primera tasca a realitzar constarà en la definició dels moviments del personatge que controlarà el jugador, i quins seran els controls que permetran realitzar aquestes accions; també s'analitzarà / plantejarà com hauran de ser els escenaris per tal d'aprofitar els moviments del personatge.

4.2 Estudi de les plataformes Android / LibGDX i Unity

En aquest apartat es comencen a utilitzar les dues plataformes, cada una per separat, realitzant proves i tutorials amb projectes establerts per tal d'aprendre el funcionament i l'estructura de les mateixes.

4.3 Estudi del programari per a la creació dels elements visuals

Dins d'aquesta tasca s'inclouen els estudis necessaris de la tècnica “Pixel Art”, per realitzar gràfics i animacions en 2D, i de tècniques per dissenyar models i animacions en 3D, amb el corresponent programari per tots dos casos, esmentat a l'apartat 2.1.

4.4 Disseny / Creació dels elements visuals

Desenvolupament dels gràfics 2D i models 3D del personatge principal, enemics i escenari, en les respectives plataformes.

4.5 Disseny / Implementació dels algorismes de control del personatge

En aquesta tasca s'implementaran tots els algorismes que refereixen al control del personatge principal; desplaçaments, atacs, moviments especials, etc.

4.6 Disseny / Implementació dels algorismes de col·lisions

Aquí s'implementaran els algorismes que permetran als diferents elements en escena (personatge principal, escenari, enemics) interactuar entre ells, detectant quan col·lisionen o es realitzen accions que afecten als altres.

4.7 Disseny / Implementació dels algorismes de la “IA” dels enemics

En aquest apartat s'implementaran tots els algorismes que es refereixen al control per intel·ligència artificial dels enemics, i els patrons de conducta respectius.

4.8 Verificació i proves dels algorismes implementats

L'última des les tasques de desenvolupament serà la de comprovar que els algoritmes realment realitzin el que han de fer.

4.9 Documentació

La documentació és una tasca que s'ha de portar a terme constantment durant el desenvolupament del projecte, i hi haurà una discussió sobre quin dels dos prototips seria l'escollit per l'entrada a producció del videojoc “Ouroboros”.

4.10 Resultats estimats

S'espera tenir, al menys, un prototip funcional sobre el que desenvolupar una demostració, del que seria un primer pas per a la creació del videojoc "Ouroboros".

5. Marc de treball i conceptes previs

En el desenvolupament de videojocs normalment s'utilitza un motor de joc (Game Engine) degudament adaptat a les necessitats dels desenvolupadors, de manera que puguin optimitzar la creació del videojoc sense preocupar-se de com el joc interactuarà amb el hardware del dispositiu pel que s'estigui creant.

Les funcionalitats dels motors de joc solen incorporar, avui en dia, un motor de generació d'imatges (rendering) per gràfics en 2D i/o models 3D encarregats de fer que es vegin per pantalla tots els elements, un motor de físiques o detector de col·lisions per determinar quan els elements de l'escena intersequen entre ells, llibreries de so per crear tota mena d'efectes, editors d'animació per donar vida als models, estructures de dades i altres recursos per desenvolupar intel·ligència artificial, gestor de memòria, etc.

S'ha de destacar que normalment es poden trobar motors específics per diferents tipologies de joc, però també en trobem de genèrics amb moltes prestacions. S'ha de valorar en cada desenvolupament quina és la millor manera de treballar i quines eines seran necessàries per assolir tots els objectius.

Així doncs, per aquest projecte s'han diferenciat clarament dues vies o opcions a seguir:

- Desenvolupar un motor específic pel videojoc que volem crear.
- Utilitzar un motor ja establert sobre el que treballar.

5.1 Desenvolupant un motor específic - Android i LibGDX

Aquesta via ens permetrà obtenir una vista completa de tot el procés de creació del joc, ja que partirem pràcticament de zero i haurem de construir/modificar bona part del motor per adequar-lo a les nostres necessitats.

Per tal d'obtenir els millors resultats en el menor temps possible, s'ha optat per utilitzar un llenguatge de programació amb el que ja s'hagi treballat amb anterioritat i se'n tingui bon coneixement; Java.

Per aquesta raó s'ha escollit treballar amb Android, ja que en la programació de les aplicacions s'utilitza Java a través del SDK (Software Development Kit). Compta amb l'avantatge que l'entorn de programació recomanat és Eclipse, un editor també conegut.

Figura 10: Android

Es van començar a buscar exemples de codi, de jocs creats amb aquesta tecnologia, fins que es va trobar el framework LibGDX.

Figura 11: LibGDX

Aquest framework, en el seu moment, només constava d'una sèrie de llibreries o classes, donant com a punt de partida un parell d'aplicacions d'exemple. D'aquesta manera es va començar a treballar; es va agafar l'exemple d'un senzill joc de plataformes i es va començar a desenvolupar, estudiant-lo i modificant-lo.

Figura 12: Super Jumper

5.2 Utilitzant un motor establert - Unity

Aquesta opció ens permetrà no haver-nos de preocupar pel desenvolupament de parts específiques del motor, i centrar-nos purament en la codificació del joc.

Es va fer una valoració de diferents motors: Unreal Development Kit, Cry Engine i Unity3d.

UDK ha estat el motor més utilitzat dins la passada generació de consoles i disposa d'una comunitat molt gran al darrere per solucionar qualsevol problema que pugui sorgir. És possible aconseguir una llicència gratuïta però si es comercialitza el joc s'han de pagar 99$ i un 25% dels ingressos.

Cry Engine va ser la resposta de l'empresa Crytek al UDK, més potent i amb millor interfície. Tot i així si no s'està versat en tècniques de desenvolupament de videojocs o de producció audiovisual en 3D la corba d'aprenentatge és molt alta. S'ha de pagar una quota mensual de 9.90 $/€ i altres mòduls que el complementen resulten molt cars.

Unity, el que s'adequa millor a les nostres exigències. Disposa d'un sistema molt fàcil per generar prototips, gràcies a una interfície gràfica molt visual, on es disposen tots els elements simplement arrastrant-los a l'escena i editant les seves característiques posteriorment amb l'ajuda d'scripts.

Figura 13: Unity

No és tant potent com els els altres 2, però a part d'esser gratuït (és possible obtenir una versió professional per una quota anual o comprant una llicència.), també disposa d'una gran varietat d'opcions a l'hora de publicar/compilar els projectes per diferents dispositius i plataformes, i una gran quantitat d'exemples d'altres projectes i tutorials.

6. Requisits del sistema

En aquest capítol seran descrits els requeriments, el quals expliquen a grans trets els objectius de l'aplicació, juntament amb les funcionalitats desitjades.

6.1 Requisits funcionals

Els serveis o opcions que oferiran els prototips a desenvolupar són:

- Iniciar una partida.
- Pausar la partida.
- Reiniciar la partida.
- Finalitzar l'aplicació.
- Moure el personatge principal per l'escenari
- Utilitzar les accions del personatge principal dins l'escenari

6.1.1 Iniciar una partida

En ambdós prototips podrem iniciar la partida a través d'un menú just a l'iniciar l'aplicació.

6.1.2 Pausar la partida

Al prototip d'Android/LibGDX podrem pausar la partida amb una icona dins la interfície. Al pausar activarem un menú. Al prototip de Unity s'ha decidit activar el menú sense pausar la partida.

6.1.3 Reiniciar la partida

En ambdós prototips podrem reiniciar la partida a través d'un menú. En cas d'Android/LibGDX es farà a través del menú activat al pausar la partida, i en el cas de Unity ho farem a través d'un menú activat dins la partida, sense pausar-la.

6.1.4 Finalitzar l'aplicació

Es podrà finalitzar l'aplicació seguint el mateix procediment que en el punt anterior.

6.1.5 Moure el personatge principal per l'escenari

El personatge principal es mourà sempre en un pla x/y (Figura 14), poguent córrer i saltar.

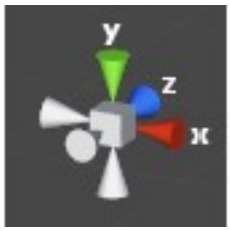


Figura 14: Eixos de coordenades x, y i z

6.1.6 Utilitzar les accions del personatge principal dins l'escenari

El personatge principal podrà estar quiet, córrer, saltar, realitzar un doble salt a l'aire, caure, fer càrregues i atacar, amb el que podrem interactuar amb els enemics.

Pot ésser que en el transcurs del desenvolupament s'afegeixin més accions en qualsevol de les versions.

6.2 Requisits no funcionals

L'entorn en el que aquest projecte es desenvoluparà, consta únicament d'un sol ordinador amb el programari i les prestacions següents.

Per tal que els prototips funcionin sense problemes es requereix el següent programari:

- Windows 7
- Eclipse
- Android SDK

El hardware utilitzat per a la implementació i proves dels prototips és el següent:

- Asus Rampage III GENE
- Intel Core i7 960 3.20Ghz Box Socket 1366
- Gigabyte GeForce GTX 560 Ti OC 1024MB GDDR5
- Exceleram Black Sark DDR3 1600 PC3-12800 12GB 3x4GB CL9

7. Estudis i decisions

Aquí anomenarem els programes i llibreries utilitzades per realitzar aquest projecte. Es mostren les eines que han permès la implementació dels prototips, des de les més bàsiques fins aquelles que han servit simplement de suport.

7.1 Eclipse

Eclipse és un programa compost per un conjunt d'eines de programació de codi obert multi-plataforma per desenvolupar aplicacions de qualsevol tipus. Aquesta plataforma, típicament ha estat utilitzada per crear entorns de desenvolupament integrats (Integrated Development Environment, IDE), aplicacions que proporcionen eines per facilitar la codificació de software; i en concret en el cas que ens ocupa, com l'IDE de Java anomenat **Java Development Toolkit** (JDT) i el compilador que s'integra com part d'Eclipse. La versió que s'ha utilitzat per a la realització d'aquest projecte és la "Indigo"(20110615-0604).

L'entorn de desenvolupament integrat (IDE) utilitza mòduls (plug-in) per proporcionar a l'usuari les funcionalitats que realment necessiti, i permeten escriure qualsevol extensió desitjada.

S'ha utilitzat per escriure el codi del prototip Android (Java) en 2D del projecte.

Figura 15: Eclipse

7.2 Android SDK

L'SDK (Software Development Kit) d'Android, inclou un conjunt d'eines de desenvolupament. Comprèn un depurador de codi, llibreries, un simulador de telèfons, documentació, exemples de codi i tutorials. La plataforma integral de desenvolupament (IDE) oficial és Eclipse juntament amb el complement ADT (Android Development Tools plugin), per crear i depurar aplicacions. A més a més es poden controlar dispositius Android que estiguin connectats.

Les actualitzacions de l'SDK estan coordinades amb el desenvolupament general d'Android. També suporta versions antigues d'Android, per si els programadors necessiten instal·lar aplicacions en dispositius ja obsolets o més antics.

S'ha utilitzat juntament amb l'Eclipse per testejar el prototip Android sobre dispositius.

Figura 16: Android SDK

7.3 LibGDX

LibGDX és un entorn de treball per el desenvolupament de videojocs multi-plataforma basat en llenguatge Java, que utilitza llibreries OpenGL, funciona amb Windows, Linux, Mac OS, Android, iOS i navegadors amb WebGL habilitat, i disposa de la seva pròpia API (Application Programming Interface).

S'ha utilitzat com a base del prototip android en 2D, agafant un dels projectes d'exemple i adaptant i/o redissenyant cada una de les parts

Ara ens centrarem en l'arquitectura d'Android amb el framework LibGDX.

7.3.1 Marc de treball de l'aplicació

L'aplicació és responsable d'engegar el joc i fer-lo córrer en una plataforma específica. Instanciarà els sub-mòduls, crearà una finestra a l'escriptori o una activitat dins d'un dispositiu Android, i s'encarregarà del fil d'execució de la interfície d'usuari (**UI thread**), que s'executarà quan hi hagi alguna interacció de l'usuari amb l'aplicació.

S'interactua amb l'aplicació de 3 maneres:

- Accedint als sub-mòduls.
- Consultant informació, com per exemple, sobre la plataforma sobre la que està corrent.
- Registrant un "ApplicationListener".

Aquest últim element permet rebre notificacions del cicle de vida de l'aplicació i reaccionar-hi. Correspon al següent model, simplificat, del cicle de vida d'una aplicació Android (Figura 17).

”AplicationListener” consta dels següents métodes:

- **resume()** es cridarà una vegada a l'arrancar l'aplicació i cada vegada que l'aplicació retorni de l'estat de pausa.
- **pause()** es cridarà quan l'aplicació sigui interrompuda i pausada per un event extern, com una crida interna o quan l'usuari pressioni el botó “home”. L'aplicació podrà ésser represa o destruïda, depenent de l'acció de l'usuari.
- **destroy()** es cridarà quan l'aplicació hagi de finalitzar.

Cada vegada que succeeixi un d'aquests events del dispositiu, l'aplicació cridarà al respectiu mètode de l'AplicationListener registrat.

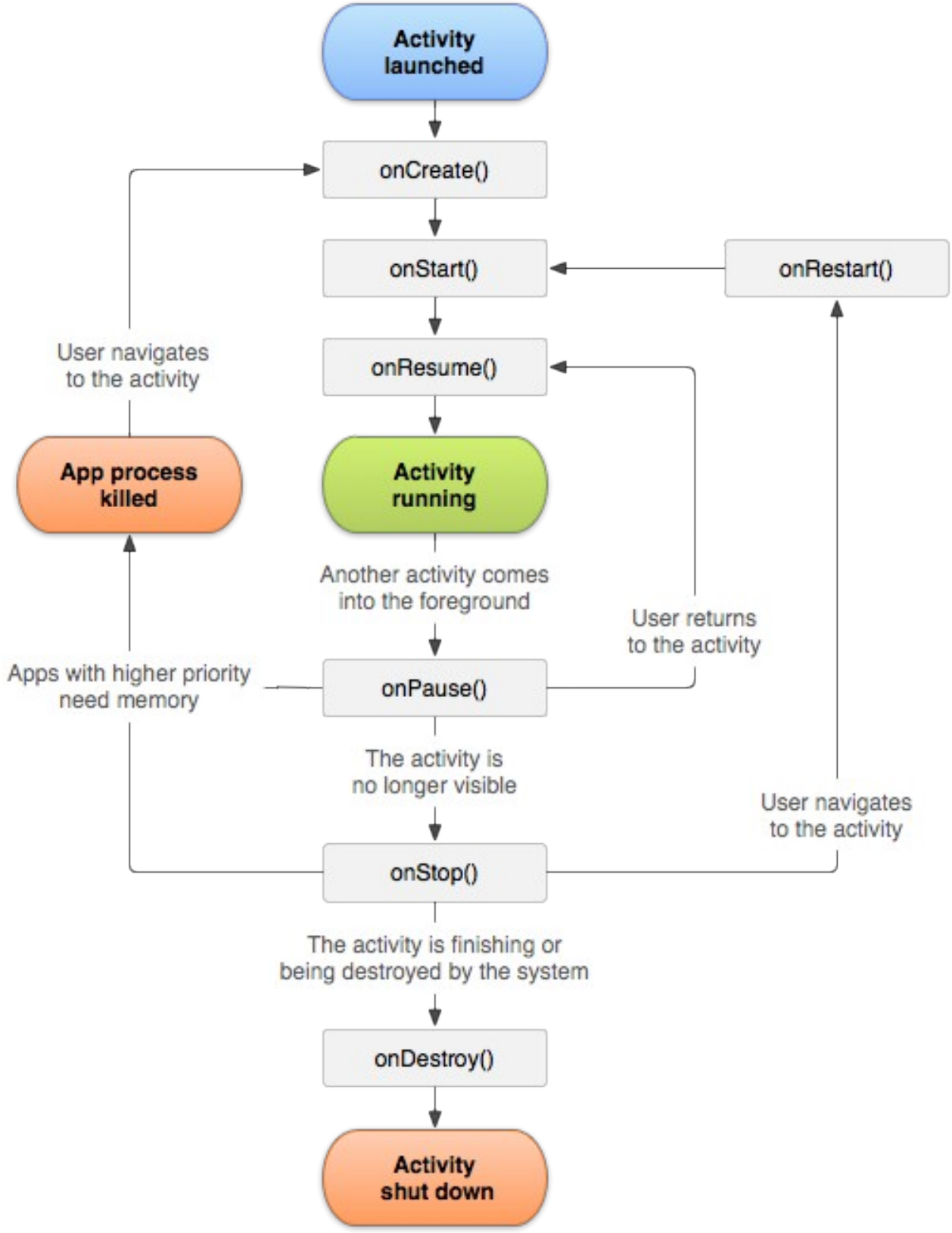


Figura 17: Diagrama d'activitat d'una aplicació Android

7.3.2 Mòdul de gràfics

Dins aquest mòdul, s'instanciarà el fil d'execució de renderitzat (**Rendering thread**). Aquest executa un bucle infinit que pot ésser pausat i reprès pels events del cicle vital de l'aplicació.

Es pot accedit a aquest fil d'execució a través de l'"AplicationListener", i reacciona a certs events que veiem a continuació:

"RenderListener" consta dels següents mètodes:

- **surfaceCreated()** es crida quan l'aplicació s'engega o quan es passa d'un estat de pausa a un estat de reprendre l'aplicació.
- **surfaceChanged()** es crida quan l'orientació de la pantalla canvia, o quan la finestra de l'escriptori es redimensiona.
- **render()** es crida contínuament. És el lloc on s'implementarà la lògica del joc conjuntament amb el renderitzat. Si l'aplicació és pausada, el fil d'execució també es pausa per tal que el mètode no es cridi.
- **dispose()** es crida quan l'aplicació hagi de finalitzar. Aquest és un lloc alternatiu per realitzar neteja de tasques, com registrar altes puntuacions o salvar altra informació.

7.3.3 Mòdul d'arxius

Ara explicarem com s'han de carregar els arxius de gràfics i àudio. Aquí és on entrarien en joc un parell de directoris predefinits a Android on es poden guardar aquest tipus de fitxers, però per aquest projecte realitzarem una aproximació diferent. El mòdul d'arxius diferencia entre 2 tipus d'arxius:

- Externs: Aquests fitxers serien els que es poden descarregar d'un servidor o fitxers que es vulguin crear per guardar puntuacions o configuracions. Aquests, en la versió per escriptori, es guardarien en relació al directori "home" de l'usuari, i en la versió Android es guardarien en relació al directori arrel de la targeta SD. Aquests fitxers poden ésser llegits i escrits.
- Interns: Aquests fitxers s'associen directament a l'arrel de l'aplicació enlloc del típic directoris d'assets, ja que podrien ser modificats. Aquests fitxers només poden ésser llegits.

En el nostre cas només farem servir arxius interns.

7.3.4 Mòdul d'entrada/sortida

Aquest mòdul ofereix accés als perifèrics d'entrada de la plataforma en la que la nostre aplicació s'estigui executant.

S'exposen els events tàctils i de teclat de dues maneres:

- Gestor d'events: Es pot registrar un "InputListener" amb la interfície d'entrada que rebrà tots els events tàctils i de teclat. Aquests events es generen en l'esmentat "UI thread" (7.3.1).
- Accés de sondeig: Es pregunta a la interfície d'entrada “UI” si s'està prement una tecla o l'usuari està tocant la pantalla.

Aquest últim mètode és el que hem escollit.

7.4 OpenGL

OpenGL és un acrònim que significa “Open Graphics Library”. És una API multi-llenguatge i multi-plataforma per escriure aplicacions o jocs que produeixen gràfics en 2D i 3D.

S'ha utilitzat la versió 1.0 (GL10) dins el framework LibGDX.

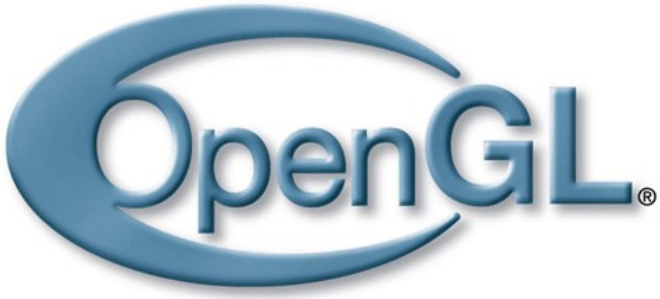


Figura 18: OpenGL

7.5 GraphicsGale (Free Edition)

GraphicsGale és una aplicació que permet, de forma rica i senzilla, editar i crear animacions i altres tipus de gràfics 2D, sobretot basats en “spriting i pixel art”.

S'ha utilitzat per crear les animacions del personatge principal amb estil pixel art.

Figura 19: GraphicsGale

7.6 Painter23

Painter23 és un editor gràfic que disposa d'una àmplia varietat d'efectes i eines amb les que retocar i editar imatges en multitud de formats diferents.

S'ha utilitzat per extreure el fons blanc de les animacions del personatge principal.

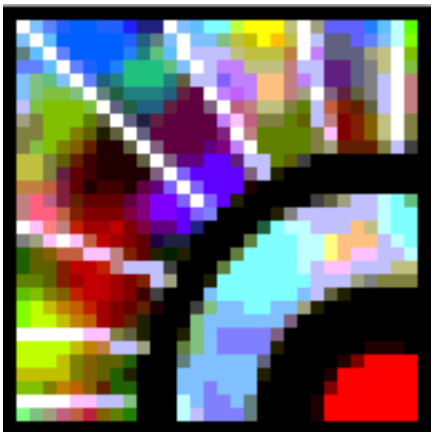


Figura 20: Painter23

7.7 Unity

Unity és un motor de videojocs multi-plataforma en el que també es poden crear animacions. Permet crear videojocs per Windows, OS X, Linux, Xbox 360, Playstation 3, Wii, Wii U, iOS, Android, Windows Phone 8, Blackberry 10 i Navegadors, fent que només s'hagi de desenvolupar una sola aplicació si es vol exportar el projecte a més d'una plataforma.

Suporta la integració amb 3ds Max, Maya, Zbrush i Photoshop, entre molts altres programes de disseny, i el motor gràfic utilitza Direct3D (Windows), OpenGL (Mac, Linux), OpengGL ES (Android, iOS), i APIs propietàries (WiiU).

S'ha utilitzat per la creació del prototip en 3D.

Ara ens centrarem en la seva arquitectura.

7.7.1 Mòduls principals

L'arquitectura de l'aplicació a nivell de subsistema és una xarxa d'objectes amb les dades encapsulades. El següent diagrama (Figura 21) il·lustra els subsistemes i les seves relacions.

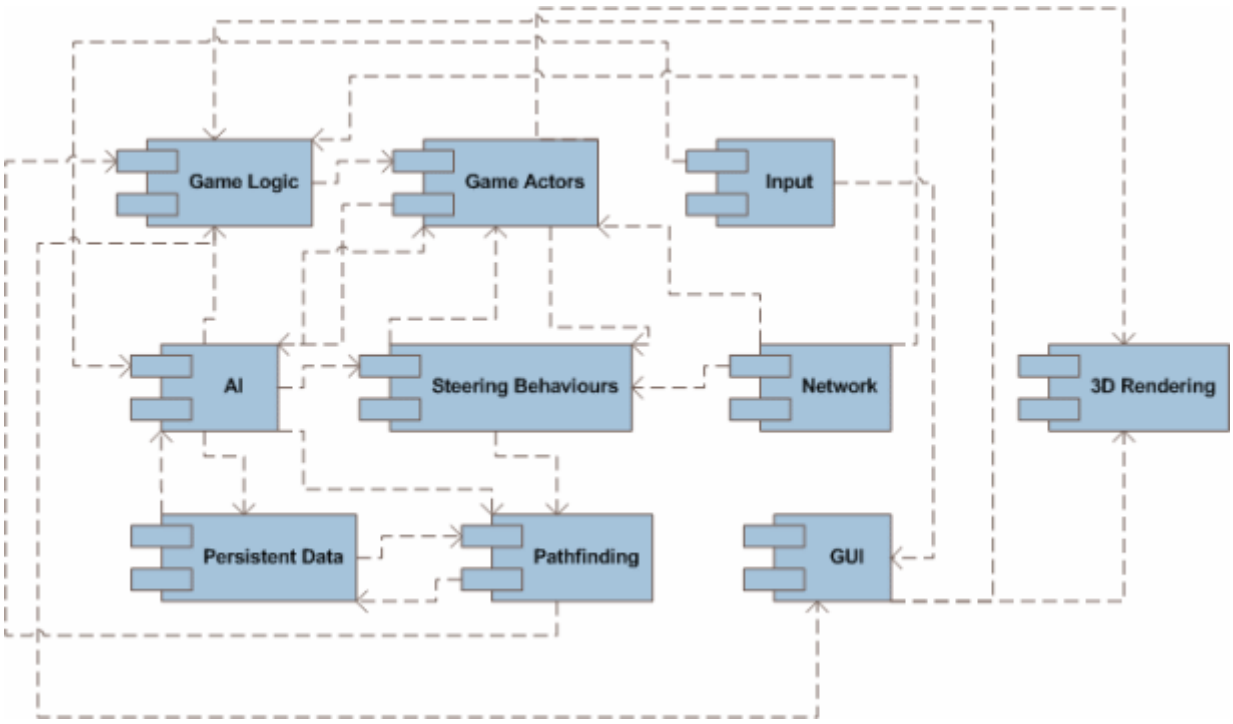


Figura 21: Subsistema Unity, Móduls i les seves relacions

7.7.2 Game Logic

Aquest mòdul s'encarrega de gestionar el jugador actual i la configuració de la intel·ligència artificial, els compte del temps i l'estat de l'aplicació (pausada, esperant per resposta de la xarxa ...)

7.7.3 AI (Intel·ligència Artificial)

El mòdul d'IA conté la lògica darrera de les unitats, grups i entitats no controlats per un jugador. Les unitats, grups i entitats fan ús de diverses conductes per a la cerca de camins o d'evasió d'obstacles i controls d'estat.

7.7.4 Persistent data

Aquest mòdul s'encarrega de guardar i carregar les dades que han d'estar disponibles entre les diferents sessions de joc. També emmagatzema les taules i grafs per a la cerca de camins del mòdul d'IA.

7.7.5 Game actors

Els actors de joc són terrenys, unitats o edificis dins el joc. Els seus models 3D es passen al sistema de renderitzat de Unity per a la visualització. Cada actor de joc posseeix referències al mòdul d'IA que controlen el seu comportament.

7.7.6 Steering behaviours

Els comportaments de direcció calculen les forces que influeixen en com i com de ràpid un actor de joc s'hauria de moure. Es poden utilitzar per evitar obstacles, moviment de grups o simples tasques de cerca.

7.7.7 Pathfinding

Aquest mòdul és responsable de la creació d'una xarxa de camins, obtenció de dades referent a obstacles i proporcionar una interfície per diferents peticions de cerques de camins. Per un millor rendiment alguna informació es guarda i es carrega des del disc.

7.7.8 Input

El mòdul d'entrada s'encarrega de realitzar el seguiment de l'entrada de dades proporcionada per l'usuari, la processa i genera una resposta.

7.7.9 Network

Aquest mòdul és el responsable de gestionar tots els estats dels actors en un joc en xarxa. Una altre responsabilitat és la de mantenir el joc consistent a totes les màquines per evitar incongruències.

7.7.10 GUI

La interfície gràfica d'usuari mostra tots els botons, menús, mini-mapa, temporitzadors de compte enrere i qualsevol altre element que formi part de la interfície. També és responsable de la funcionalitat d'aquests elements i interactua fortament amb el mòdul de lògica de joc per a aquest propòsit.

7.7.11 3D rendering

Aquest mòdul va íntimament lligat amb la càmera principal de l'escena i determina els objectes que han d'esser renderitzats. Unity encapsula la majoria dels detalls de renderitzat però també ofereix accés a través de "shaders" de píxels i vèrtexs (apliquen ombres/efectes de relleu).

7.8 Direct 3D

Direct 3D és una API disponible per sistemes operatius Microsoft Windows per escriure aplicacions o jocs que produeixen gràfics en 2D i 3D. Està associat amb la col·lecció de llibreries DirectX. El competidor principal és OpenGL.

Figura 22: DirectX

7.9 Creation Kit

Creation Kit és un programa amb el que es poden realitzar modificacions del videojoc “The Elder Scrolls: Skyrim”, permetent utilitzar els arxius del joc.

S'ha utilitzat per extreure models 3D, animacions i textures per reutilitzar-los en la creació del prototip amb Unity. Aquests models s'extreuen en format “**.NIF**” (veure apartat 7.11).

Figura 23: Creation Kit

7.10 3DsMax

3DsMax és un software de modelat i animació en 3D que proporciona una solució completa de modelat, animació, renderització i composició en 3D als creadors de jocs, cinema i altres medis audiovisuals.

S'ha utilitzat per importar i modificar els models extrets del Creation Kit.

Figura 24: 3DsMax

7.11 NifSkope

NifSkope és un programa gràfic que permet obrir arxius **.NIF**, visualitzar el seu contingut, editar-lo i guarda-lo de nou. Disposa d'un plug-in per 3DsMax en el que es poden realitzar canvis a propietats específiques d'un arxiu NIF i també importar i exportar-los.

Els arxius NIF estan vinculats a “GameBryo”, un motor de videojocs creat per “Numerical Design Limited”, i que conté tota la informació relacionada dels models 3D creats per ésser utilitzats en aquest motor.

S'ha utilitzat el plug-in per 3DsMax per tenir compatibilitat amb els arxius extrets del Creation Kit.

Figura 25: NifSkope

7.12 FBX Converter

FBX Converter és una eina d'Autodesk d'intercanvi de dades d'alta fidelitat entre diferents programes, que s'utilitza principalment per arxius que contenen dades de models en 3D amb una gran quantitat de variables referents a textures, superfícies, il·luminació, estructura, etc.

S'ha utilitzat per exportar i importar els models i animacions de 3DsMax a Unity.

Figura 26: FBX Converter

7.13 Photoshop

Photoshop es un programa en forma de taller de pintura i fotografia que treballa sobre un “llenç” per capes i que està destinat a l'edició, retoc fotogràfic i pintura a base d'imatges de mapa de bits. Disposa d'una gran quantitat d'eines i se'n fan plug-ins per tenir major compatibilitat amb altres programes.

S'ha utilitzat per modificar i crear textures pels models 3D, compatibilitat amb 3DsMax.

Figura 27: Photoshop

8. Anàlisi i disseny del sistema

Abans de descriure l'anàlisi i el disseny dels apartats especificats a la planificació, descriurem el diagrama de casos d'ús per tal de lligar la interacció dels usuaris amb tot el funcionament dels prototips.

Els prototips es basaran, simplement, en moure el nostre personatge per un escenari de proves, amb plataformes per les que moure's i saltar, en el que hi haurà un o varis enemics als que podrem derrotar amb moviments d'atac.

8.1 Diagrames de casos d'ús

Els nostres prototips comptaran amb unes senzilles interfícies amb les que, com a mínim, es podrà reiniciar el programa.

8.1.2 Unity

Amb el prototip de Unity, es dona la possibilitat de canviar la configuració de pantalla, de qualitat gràfica, i també de l'entrada del sistema. Aquestes opcions vénen ja donades per el propi editor al publicar el prototip. El procés és el següent:

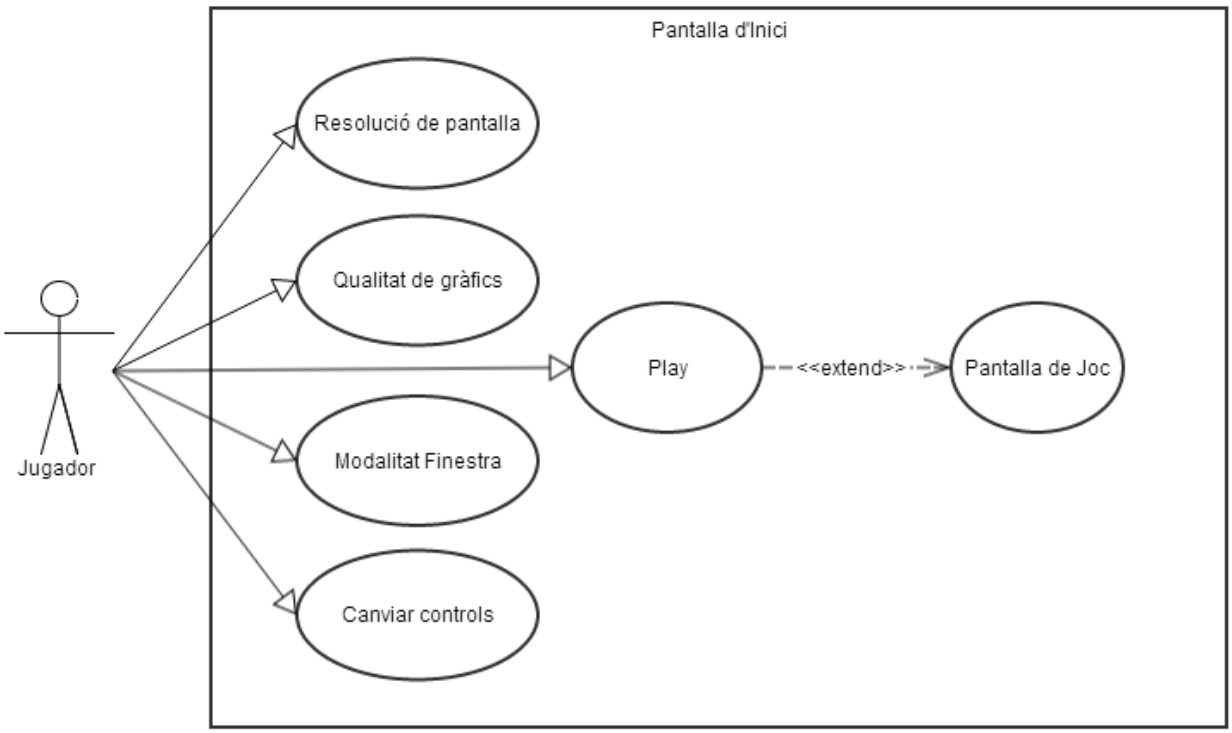


Figura 28: Diagrama de casos d'ús, Pantalla d'inici - Interfície Unity

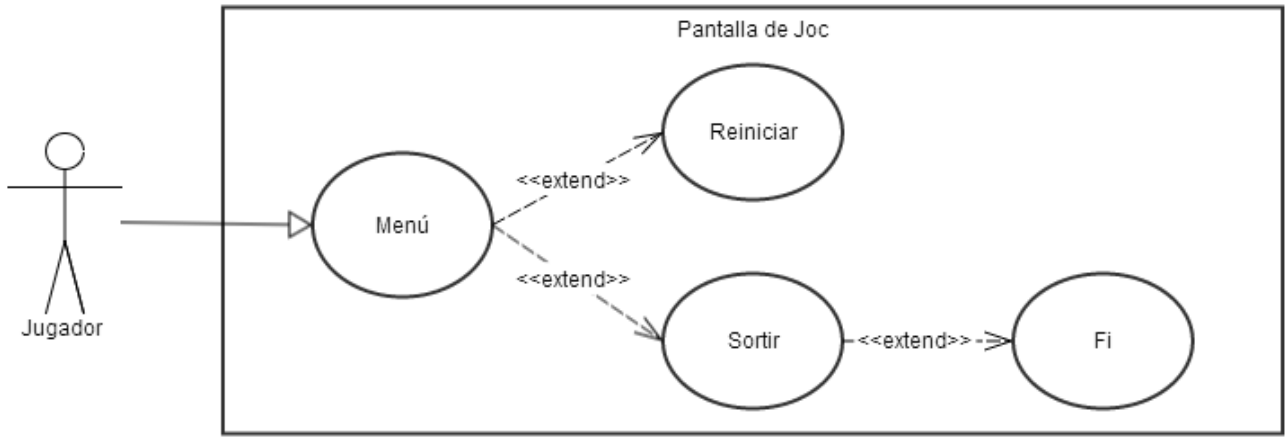


Figura 29: Diagrama de casos d'ús, Pantalla de joc - Interfície Unity

8.1.1 Android/LibGDX

Seguint amb el que s'ha explicat a l'apartat 6.1 tindrem els següents diagrames de casos d'ús del funcionament i/o interacció de l'aplicació amb l'usuari:

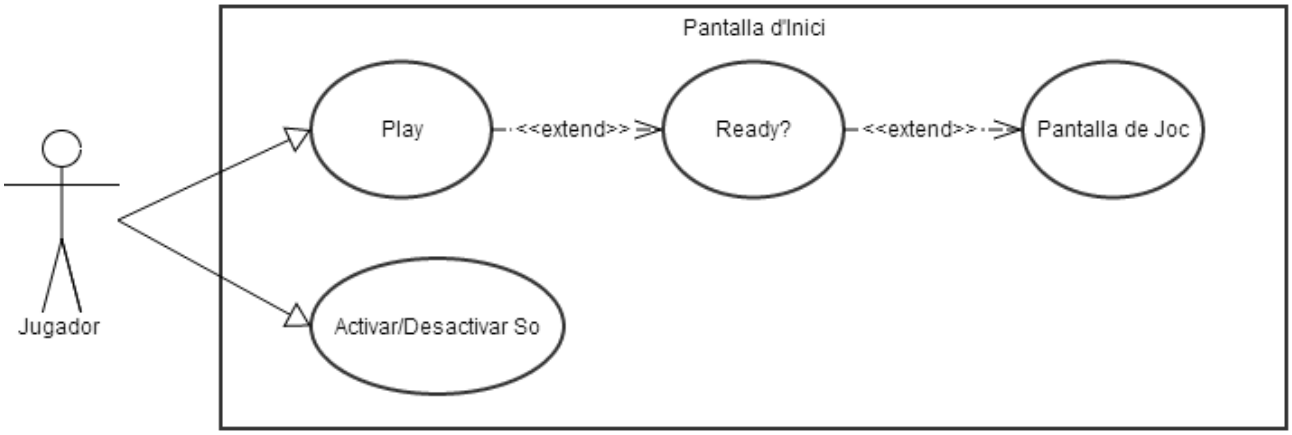


Figura 30: Diagrama de casos d'ús, Pantalla d'inici - Interfície Android

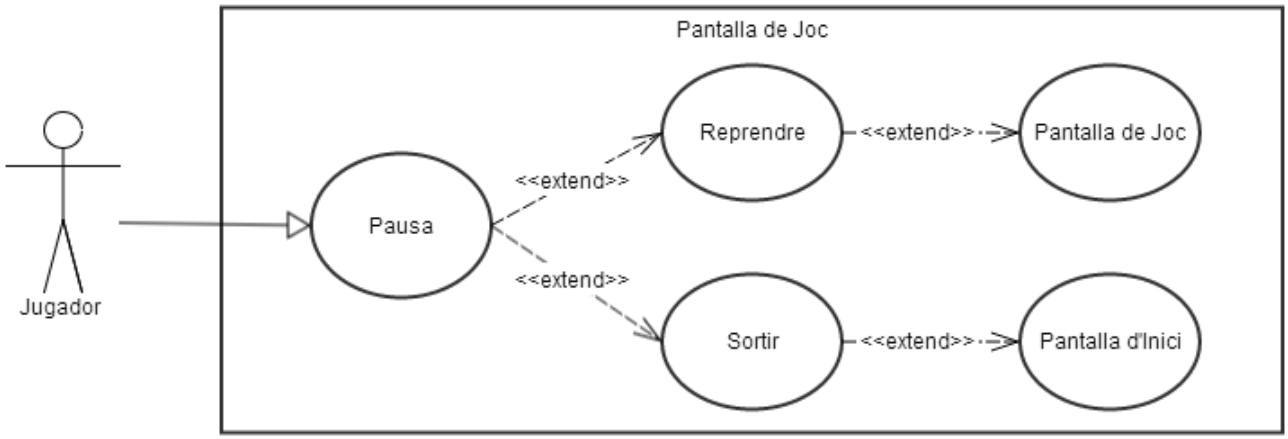


Figura 31: Diagrama de casos d'ús, Pantalla de joc - Interfície Android

8.2 Disseny dels elements visuals

Els elements visuals d'ambdós prototips no tindran res a veure entre ells, ja que el prototip d'Android serà a base d'sprites 2D, i el de Unity a base de models 3D.

8.2.1 Android/LibGDX

Per A reproduir qualsevol element en 2D, simplement es requereix un arxiu d'imatge en format "**.png**" ja que permet, amb el sistema de compressió, no tenir pèrdues de detall i utilitzar el canal "**alfa**" (RGBA) per obtenir transparències, i així poder mostrar siluetes definides i no només rectangulars.

Per a la reproducció d'animacions el que es requereix són els mateixos arxius d'imatge, però amb la peculiaritat que haurien de contenir un dibuix per cada instant de la seqüència d'animació que es vol realitzar.

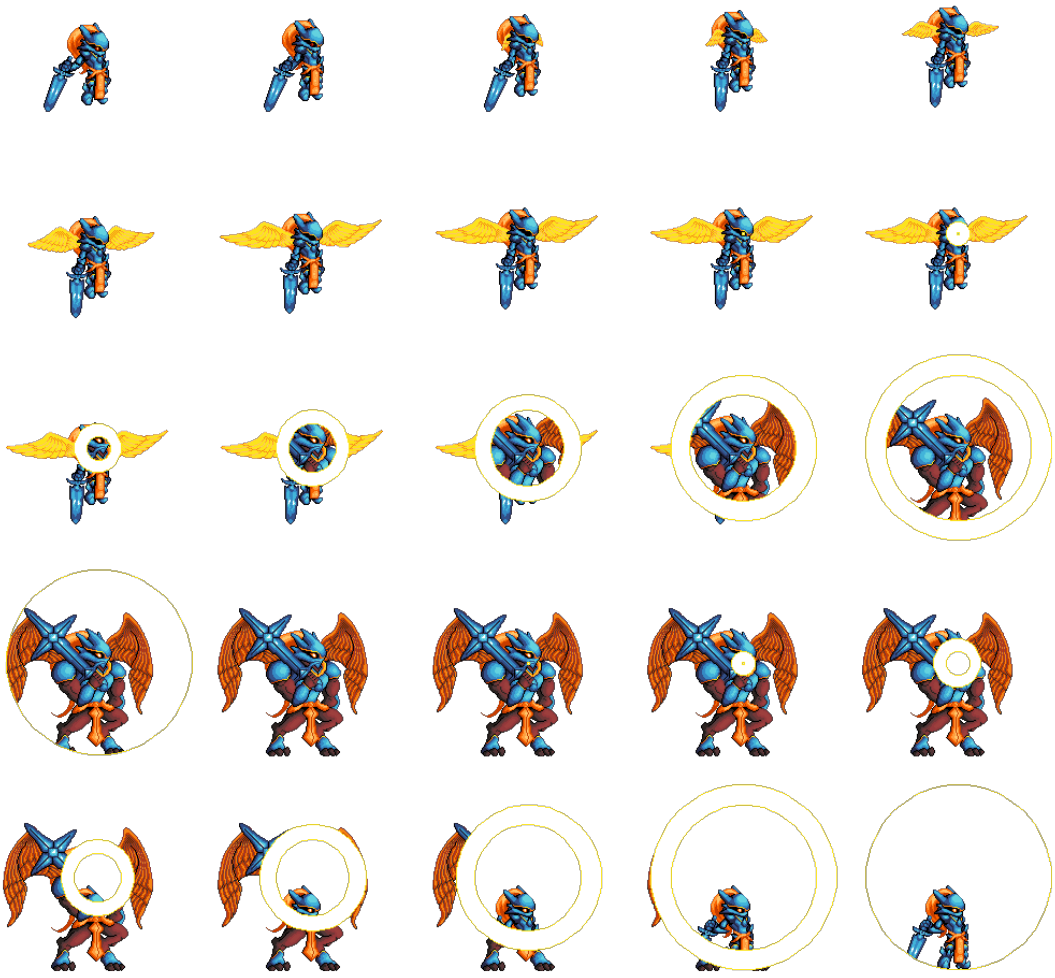


Figura 32: Arxiu d'imatge que conté una animació en 2D

El que es fa és “**mapejar**” aquesta imatge i configurar un objecte d'animació que contingui les coordenades de cada instant (Figura 32), així doncs hi pot haver més d'una animació en una mateixa imatge. També s'afegeix dins de l'objecte d'animació una altre variable, un valor numèric que determinarà la velocitat amb la que es canviarà d'imatge a l'iniciar l'animació.

Un cop tenim tots els elements dibuixats i configurats, es començarà a imprimir en pantalla els elements que es troben al fons de l'escena, i acabant pels que es troben més aprop; per últim es posen els elements de la interfície d'usuari si n'hi ha. Això es coneix com l'algorisme del pintor.

Figura 33: Pantalla de joc del prototip d'Android/LibGDX

8.2.3 Unity

Per reproduir qualsevol element en 3D, es necessita un arxiu que conté la informació del disseny: textures, materials, mapejat UV (les sigles UV denoten els eixos d'una textura perquè X, Y i Z ja es fan servir per representar un objecte 3D a l'espai.), la malla de polígons que el formen (Figura 34), i el seu esquelet intern (Figura 116).

Nosaltres no els crearem des de zero, a excepció de l'escenari de proves, sinó que utilitzarem dissenys d'un altre joc dels que n'haurem modificat la forma i textura. Així mateix també aprofitarem i modificarem les animacions, que es troben contingudes dins un altre arxiu.

Figura 34: Comparativa dels models original i final

A l'hora de posar-ho tot en marxa s'associaran vàries animacions a un model en el que també hi haurà associat un script amb totes les variables de control, de manera que quan escaigui es reproduirà l'animació corresponent.

Figura 35: Pantalla de joc del prototip de Unity

Un cop tenim tots els elements dissenyats i configurats, es necessitarà un objecte “càmera” que captarà l'escena i mostrarà els elements amb el punt de vista adequat (Figura 35).

8.3 Disseny del control del personatge

Pel control del personatge ens centrarem primer en determinar els moviments que volem que realitzi. Així doncs, un cop tenim una llista de tot el que el nostre personatge podrà fer, es determina un diagrama d'estats per tal de saber com es realitzarà la transició d'un estat a un altre (Figura 36).

Per exemple, si agafem l'estat del centre del diagrama “Córrer”, podrem veure que mentre el personatge està corrent podrà passar als següents estats: Quiet, Saltar, Caure i Càrrega.

Per altre banda, els estats que poden passar a l'estat “Córrer” són els següents: Quiet, Saltar, Doble salt, Caure, Càrrega i Atacar

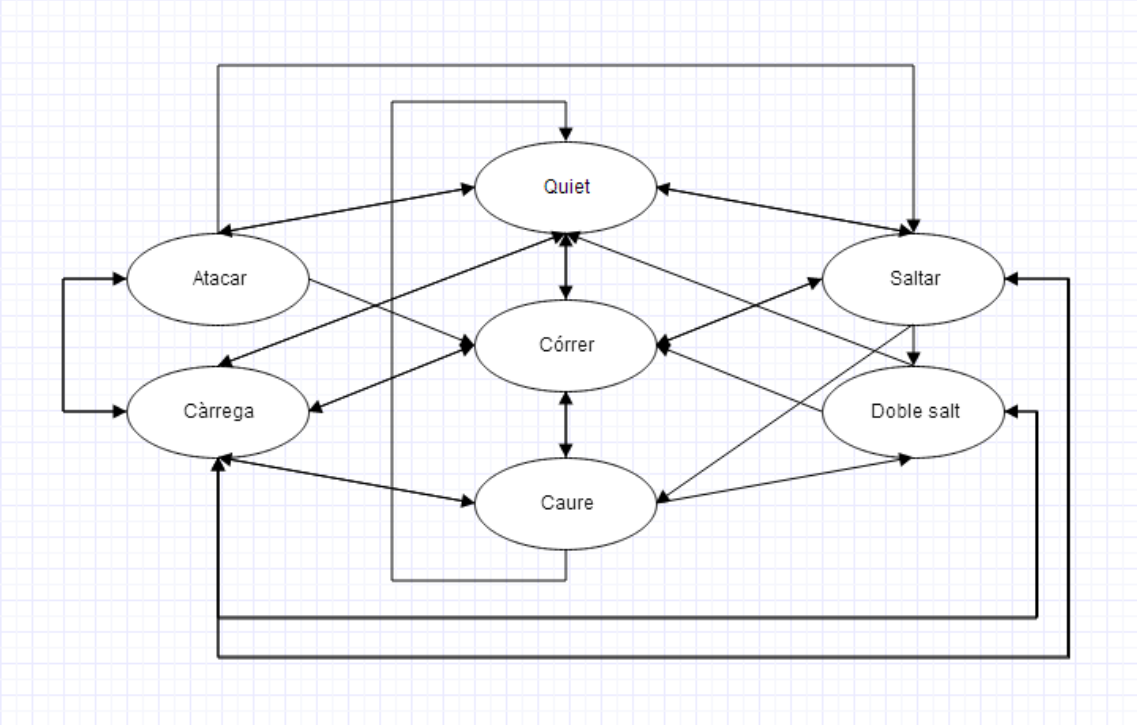


Figura 36: Diagrama d'estats del personatge

En la modificació de l'estat del personatge intervenen dos punts principals; l'entorn dins el mateix videojoc i el dispositiu d'entrada (Input) que utilitzarem per moure'l.

Així doncs, es crearà la lògica per l'entorn del videojoc, amb la gravetat i els elements que puguin interactuar amb el personatge. Aquesta serà la base sobre la que construirem l'altra lògica de control, pel dispositiu d'entrada, on li donarem les ordres.

Bàsicament el comportament del personatge és una combinació del que passa dins el món del videojoc amb les ordres que nosaltres li donem. Per exemple, si estem caiguen no som capaços d'atacar per molt que premem el botó corresponent (la gravetat -món del joc- té prioritat sobre atacar -dispositiu d'entrada-), però sí que podrem tornar a saltar (saltar -dispositiu d'entrada- té prioritat sobre la gravetat -món del joc-).

Aquestes prioritats les establim al diagrama d'estats; aleshores, si afegíssim una relació de “Caure” fins a “Atacar”, sí que podríem atacar mentre caiguéssim, i pel costat contrari si eliminéssim la relació existent de “Caure” fins a “Doble Salt” no podríem tornar a saltar mentre caiguéssim.

Un altre aspecte a destacar és el moment en el que deixem de tenir control sobre el personatge; això succeirà quan o bé sortim del terreny de la pantalla de joc(Android/LibGDX) o quan el personatge mori (Unity). Farem un cop d'ull a les fitxes de casos d'ús per entendre-ho millor.

Fitxa de cas d'ús: Sortida del terreny de joc (Android/LibGDX)	
Descripció	Quan el jugador es mou lliurement per l'escenari pot acabar sortint dels límits del terreny de joc i no tornar
Actor	Jugador
Pre-condició	L'actor es troba a la pantalla de joc
Flux Principal	1. Es mou el personatge utilitzant el dispositiu d'entrada 2. El personatge surt dels límits de la pantalla 3. El personatge no pot tornar al terreny de joc 4. Per acció de la gravetat la posició del personatge descendeix fins arribar a un límit establert
Post-condició	Es suprimeix el control sobre el personatge i es mostra la pantalla de fi de joc (game over)
Observacions	S'ha de poder reiniciar la partida o tornar a la pantalla d'inici

Fitxa de cas d'ús: Mort del personatge (Unity)	
Descripció	Quan el jugador s'enfronta a un grup d'enemics i es queda sense vida, o cau dins la lava, ha de morir
Actor	Jugador
Pre-condició	L'actor es troba a la pantalla de joc
Flux Principal	1. Es mou el personatge utilitzant el dispositiu d'entrada 2. El personatge s'enfronta a un grup d'enemics 3. El personatge es queda sense vida
Flux alternatiu	1. Es mou el personatge utilitzant el dispositiu d'entrada 2. El personatge cau dins la lava
Post-condició	Es suprimeix el control sobre el personatge, s'actualitza el seu estat a mort i es mostra un missatge de fi de joc (game over)
Observacions	S'ha de poder reiniciar la partida o executar el menú de joc

A partir d'aquí es creen i/o modifiquen les animacions que s'hauran d'utilitzar per cada un dels estats que s'han determinat, i es retoquen els paràmetres de la lògica fins a aconseguir el comportament desitjat.

8.4 Disseny del motor de col·lisions

Els motors de col·lisions d'ambdós prototips també seran diferents donades les característiques especials de cada motor. Només tindran en comú la física resultant, que configurarem segons les necessitats del joc. Permetran detectar i identificar quan el jugador interseca amb altres elements de l'escenari per poder-ne modificar el comportament si es creu oportú.

8.4.1 Android/LibGDX

El que requerirem per al motor de col·lisions és la gravetat, determinant quan i com un personatge ha d'ésser influenciat per aquesta força. També s'ha de determinar quines són les parts de l'escena que evitaran que el personatge caigui per la gravetat, les plataformes, i com interactuar amb elles.

Així doncs per fer que els personatges interactuïn amb les plataformes (col·lisionin) s'utilitzarà un algoritme d'intersecció de rectangles. Cada personatge i cada plataforma

tindrà una referència a un rectangle que s'utilitzarà per detectar si hi ha col·lisions entre ells (Figura 37).

Figura 37: Rectangles del personatge i plataformes, que utilitza el motor de col·lisions

8.4.2 Unity

Dins l'Unity el sistema de col·lisions serà molt més simple que no pas en Android, ja que no requerirem d'algoritmes propis per realitzar les col·lisions. El que farem serà proporcionar als models 3D dels personatges d'un "Character Controler" (Figura 38) que delimita l'espai, com si es tractés d'una bombolla al seu voltant.

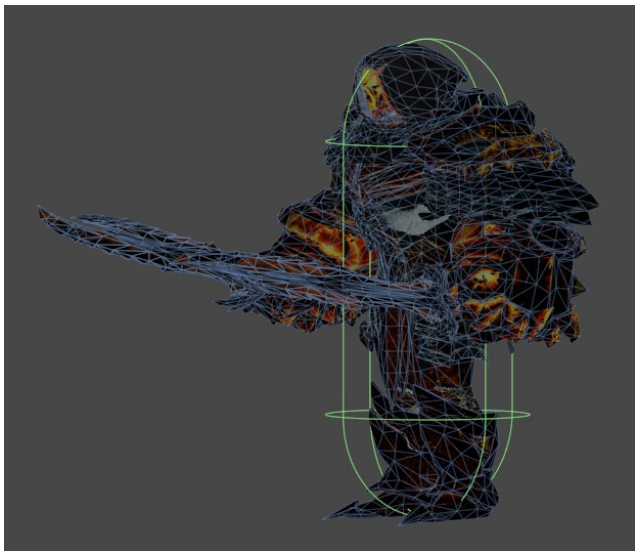


Figura 38: Character controller

El propi Unity utilitza aquests controladors de personatges per determinar com intersequen amb l'entorn i entre ells.

8.5 Disseny de la intel·ligència artificial

El procés que s'utilitzarà per fer que els enemics puguin arribar al jugador automàticament, es basarà en realitzar una representació matemàtica de l'escenari mitjançant un graf. Un graf és un conjunt d'objectes anomenats vèrtexs o nodes units per enllaços anomenats arestes o arcs, que permeten representar relacions entre elements d'un conjunt (veure Figura 39).

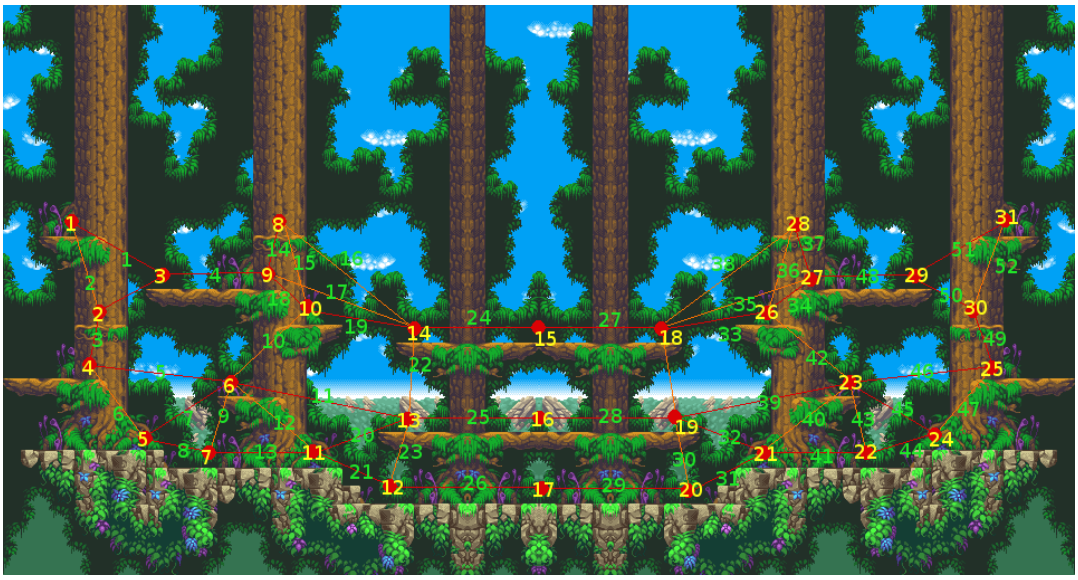


Figura 39: Escenari de proves Android amb un graf mapejat

Aquest graf en concret consta de 31 nodes (números en groc) i 52 arestes (números en verd). S'ha de destacar que hi ha dos tipus d'arestes; bidireccionals (vermelles) i unidireccionals (taronja). Les bidireccionals indiquen que els enemics podran anar i tornar entre els nodes de l'aresta, mentre que les unidireccionals només permetran anar des del node amb una posició més elevada fins al node amb la posició més baixa.

8.6 Diagrames de classes (Android/Libgdx)

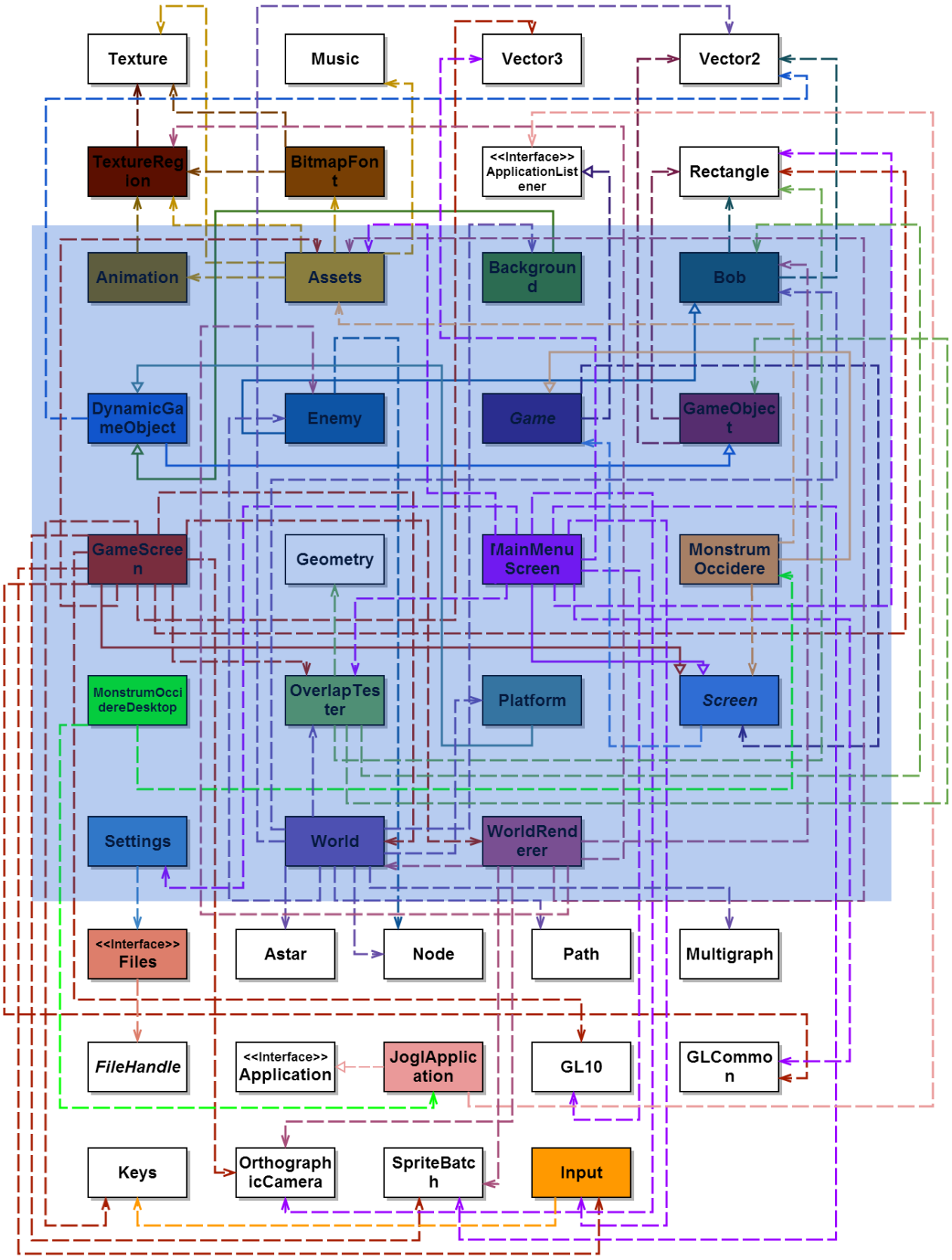


Figura 40: Diagrama de classes, Android/LibGDX

El diagrama de classes mostrat (Figura 40), conté les classes més destacades del prototip. S'han remarcat amb un fons blavós aquelles que formen part del prototip creat, mentre que la resta provenen de les llibreries utilitzades esmentades a l'apartat 7.

S'ha omès la multiplicitat per veure més clarament les relacions; les nomenarem als següents apartats on farem un repàs a les classes que formen part del prototip pròpiament dit.

8.6.1 Classe Animation

Conté les animacions basades en frames.

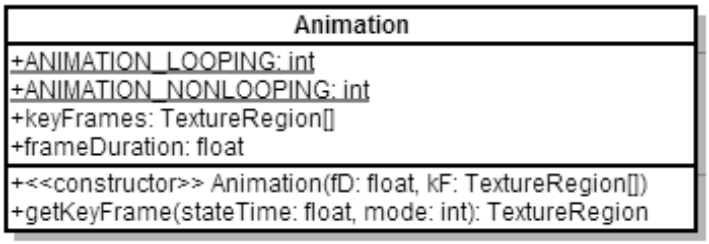


Figura 41: Classe Animation

Atributs

`+ANIMATION_LOOPING`: Constant que servirà per indicar si l'animació ha de crear un loop.

`+ANIMATION_NONLOOPING`: Constant que servirà per indicar si l'animació no ha de crear un loop.

`+keyFrames`: Taula de "TextureRegion" on es contindràn tots els frames de l'animació.

`+frameDuration`: Variable que indica el tems que s'ha de mostrar cada frame.

Mètodes

`+Animation(float frameDuration, TextureRegion ... keyFrames)`: Constructor.

`+getKeyFrame(float stateTime, int mode)`: Retorna el frame que pertoca de l'animació segons el temps i el mode entrats.

Multiplicitats

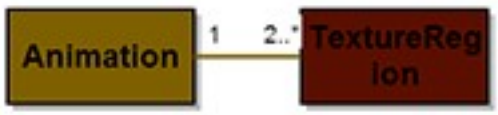


Figura 42: Multiplicitats Animation

8.6.2 Classe Assets

Conté tots els recursos gràfics i sonors necessaris per a l'aplicació.

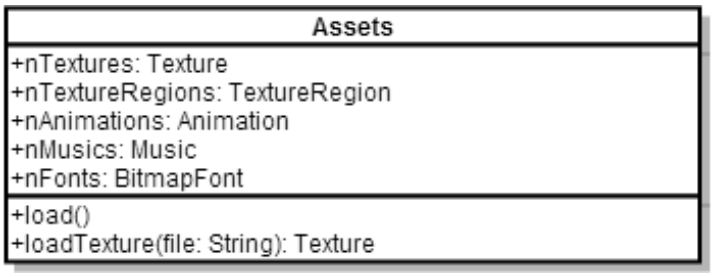


Figura 43: Classe Assets

Atributs

`+nTextures`: Objectes de tipus “Texture” que contindran imatges.

`+nTextureRegions`: Objectes de tipus “TextureRegion” que contindran coordenades d'una textura, determinant-ne un rectangle que contindrà un frame.

`+Nanimations`: Objectes de tipus “Animation” que contindran els frames.

`+nMusics`: Objectes de tipus “Music” que contindran arxius de música.

`+nFonts`: Objectes de tipus “BitmapFont” que contindran fonts de texte.

Mètodes

`+load()`: Carrega tots els recursos que necessitarà l'aplicació.

`+loadTexture(String file)`: Retorna la textura de nom “file”.

Multiplicitats

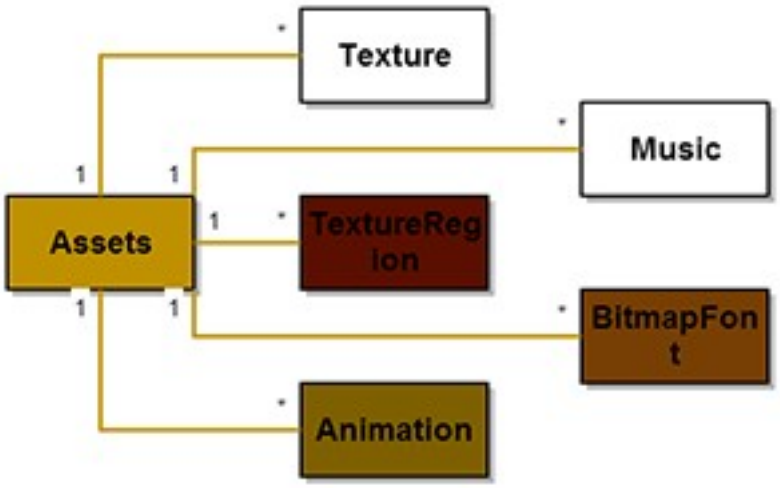


Figura 44: Multiplicitats Assets

8.6.3 Classe Background

Extén de "DynamicGameOject" i conté les dades d'un fons de pantalla de l'aplicació.

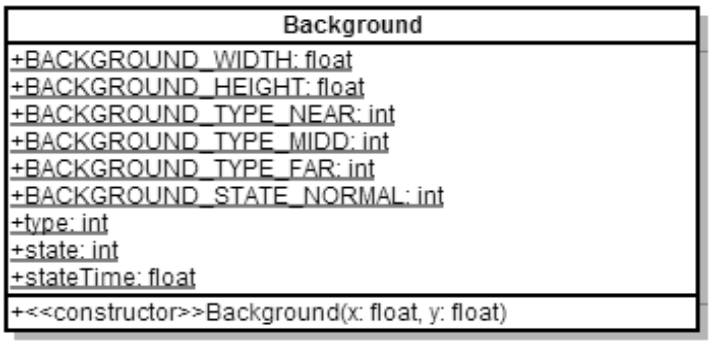


Figura 45: Classe Background

Atributs

`+BACKGROUND_WIDTH`: Mida constant de l'amplada del fons.

`+BACKGROUND_HEIGHT`: Mida constant de l'alçada del fons.

`+BACKGROUND_TYPE_NEAR`: Constant per indicar el tipus de distància del fons, propera.

`+BACKGROUND_TYPE_MIDD`: Constant per indicar el tipus de distància del fons, mitjana.

`+BACKGROUND_TYPE_FAR`: Constant per indicar el tipus de distància del fons, llunyana.

`+BACKGROUND_STATE_NORMAL`: Constant per indicar l'estat del fons.

`+type`: Variable per indicar el tipus de distància del fons.

`+state`: Variable per indicar l'estat del fons.

`+stateTime`: Variable per indicar el temps d'estat.

Mètodes

`+Background(float x, float y)`: Constructor.

Multiplicitats (cap)

8.6.4 Classe Bob

"Bob" és el nom donat al personatge principal; aquesta classe representa al personatge controlat pel jugador.

Figura 46: Classe Bob

Atributs

+BOB_STATE_JUMP: Constant per indicar l'estat en el moment de saltar.

+BOB_STATE_FALL: Constant per indicar l'estat en el moment de caure.

+BOB_STATE_GROUND: Constant per indicar l'estat quan s'està posicionat sobre un pla.

+BOB_STATE_RUN: Constant per indicar l'estat quan es corre.

+BOB_STATE_FALL_GROUND: Constant per indicar l'estat quan canvia de caure, a un pla.

+BOB_STATE_JUMPING: Constat per indicar l'estat quan s'està saltant.

+BOB_STATE_FALLING: Constat per indicar l'estat quan s'està caiguen.

+BOB_STATE_ATTACK: Constant per indicar l'estat quan s'està atacant.

+BOB_STATE_RUN_ATTACK: Constant per indicar l'estat quan s'està atacant i corrent.

+BOB_STATE_HITTED: Constant per l'estat indicar quan un atac l'ha impactat.

+BOB_STATE_DEAD: Constant per indicar l'estat quan s'està mort.

+BOB_STATE_DYING: Constant per indicar l'estat que s'està morint.

+BOB_STATE_DEADEND: Constant per indicar l'estat quan s'ha acabat de morir.

+BOB_STATE_CHARGE: Constant per indicar l'estat quan està realitzant una càrrega.

+BOB_STATE_SPIN_JUMP: Constant per indicar l'estat quan es realitza un doble salt.

+BOB_STATE_FINAL_ATTACK: Constant per indicar l'estat quan es realitza l'atac final.

+BOB_HITTED_VELOCITY: Constant per indicar la velocitat quan un atac l'ha impactat.

+BOB_CHARGE_VELOCITY: Constant per indicar la velocitat quan realitza una càrrega.

+BOB_JUMP_VELOCITY: Constant per indicar la velocitat del salt.

+BOB_MOVE_VELOCITY: Constant per indicar la velocitat quan es mou.

+BOB_WIDTH: Constant per indicar el gruix del personatge.

+BOB_HEIGHT: Constant per indicar l'alçada del personatge.

`+velocity`: Variable per indicar la velocitat (x,y) del personatge.

`+accel`: Variable per indicar l'acceleració (x,y) del personatge.

`+position`: Variable per indicar la posició (x,y) del personatge.

`+bounds`: Variable per indicar el perímetre i posició del personatge amb un rectangle.

`+boundsAnt`: Variable per indicar el perímetre i posició anterior del personatge amb un rectangle.

`+state`: Variable per indicar l'estat actual del personatge.

`+stateAnt`: Variable per indicar l'estat anterior del personatge.

`+lastPlatformHit`: Variable per indicar l'última plataforma sobre la que s'ha reposat.

`+lastPlatformTouch`: Variable per indicar l'última plataforma amb la que s'ha xocat.

`+numAttack`: Variable per indicar el número de l'atac que s'està utilitzant.

`+health`: Variable per indicar la vida restant.

`+stateTime`: Variable per indicar el temps que es troba en l'estat actual.

`+sideAnt`: Variable per indicar l'orientació anterior.

`+sideAtcked`: Variable per indicar de quin costat han atacat al personatge.

`+resetJump`: Variable per indicar si s'ha de poder tornar a saltar o no.

`+spinJump`: Variable per indicar si es pot realitzar el doble salt o no.

`+resetCharge`: Variable per indicar si es pot tornar a realitzar la càrrega o no.

`+landing`: Variable per indicar si s'està aterrant sobre una plataforma.

`+enemies`: Vector d'objectes que el personatge ha de considerar com enemics.

Mètodes

`+Bob(float x, float y)`: Constructor, crea el personatge principal a les coordenades x,y.

`+attack()`: Realitza una atac si es pot.

```
funcio attack()
	si (estat == estatQuiet)
		estat = estatAtacar
		tempsEstat = 0
		comprovarCol·lisioAtac()
	altrament si (estat == estatAtacar)
		numAtac++
		si (numAtac > numMaxAtac) numAtac = 1 fsi
		tempsEstat = 0
		comprovarCol·lisioAtac()
	fsi
ffuncio
```

Si es troba en estat quiet el canvia a atacar i comprova col·lisions d'atac; si es troba en estat d'atac augmenta el número d'atac per passar al següent i comprova col·lisions d'atac.

`+charge()`: Realitza una càrrega si es pot.

```
funcio charge()
	si (podemFerCarrega) llavors
		estatAnterior = estat
		estat = estatCarrega
		tempsEstat = 0
		comprovarCol·lisioAtac()
		podemFerCarrega = fals
	fsi
ffuncio
```

Es guarda l'estat actual per després de la càrrega, es passa a estat de càrrega, es comproven col·lisions d'atac i es desactiva la càrrega.

`-checkAttackCollisions()`: Mira quins enemics hi ha dins el rang d'atac i els fereix.

```
funcio checkAttackCollisions()
	actual: Enemic
	it: Iterador<Bob> = enemics.iterador()
	mentre (it.hiHagiSeguent()) fer
		actual = (Enemic)it.seguent()
		si (actual.noEsMort())
			si (actual.dinsRangAtac())
				actual.tocat()
			fsi
		fsi
	fmentre
ffuncio
```

`+hitPlatform()`: Controla la lògica d'estats del personatge al aterrar sobre una plataforma.

```
funcio hitPlatform()
	si (estat != estatCarrega)
		si (velocitat.x == 0)
			si (estat == estatCaiguent) aterrant = cert fsi
			si (aterrant)
				si (estat != estatAtacar && estat != estatAterrant)
					estat = estatAterrant
					tempsEstat = 0
				fsi
				si (tempsEstat > tempsAterrant) aterrant = fals fsi
				si (tempsEstat > tempsResset)
					podemFerSalt = cert
					podemFerDobleSalt = cert
				fsi
```

```
                altrament
                        si (estat != estatAtacar)
                                si (estat != estatMorint || estat != estatMort)
                                        si (estat != estatQuiet)
                                                estat = estatQuiet
                                                tempsEstat = 0
                                        fsi
                                        podemFerSalt = cert
                                        podemFerDobleSalt = cert
                                fsi
                        altrament
                                si (numAtac == numMaxAtac && tempsEstat >
                                    tempsAtac)
                                        estat = estatQuiet
                                        numAtac = 1
                                fsi
                        fsi
                fsi
        altrament
                si (estat == estatAtacat)
                        estat = estatQuiet
                        velocitat.x = 0
                altrament si (estat == estatMorint)
                        estat = estatMort
                        tempsEstat = 0
                        velocitat.x = 0
                altrament si (estat != estatCorrent)
                        estat = estatCorrent
                        tempsEstat = 0
                fsi
                si (tempsEstat > tempsResset)
                        podemFerCarrega = cert
                        podemFerSalt = cert
                        podemFerDobleSalt = cert
                fsi
        fsi
    altrament
        si (tempsEstat >= tempsCarrega)
                velocitat.x = 0
                estat = estatAnterior
        fsi
    fsi
ffuncio
```

Si no està carregant i es troba quiet es mira si està aterrant i quan de temps fa per poder tornar a activar els salts; també si està atacant i quin atac esta fent, i si no es troba quiet es mira si l'han atacat, si s'està morint o si estava corrent; també els temps per poder tornar a activar els salts i la càrrega.

Si esta carregant i ha passat el temps de càrrega es torna a l'estat anterior.

`+hitted(float side)`: Actualitza l'estat del personatge quan l'ha tocat un atac.

```
funcio hitted(side: float)
	si (salud > 0)
		salud =- 10
		estat = estatAtacat
	altrament si (salud == 0)
		estat = estatMorint
	fsi
	velocitat.y = velocitat_atacat
	velocitat.x = velocitat_atacat * side
	tempsEstat = 0
ffuncio
```

Si queda salud la redueix i es canvia l'estat a atacat, si no queda salud es canvia l'estat a morint. Es canvia la velocitat per representar el toc que ha rebut el personatge.

`+jump()`: Realitza un salt si es pot.

```
funcio jump()
	si (podemFerSalt)
		velocitat.y = velocitat_salt
		estat = estatSalt
		tempsEstat = 0
		podemFerSalt = fals
		podemFerDobleSalt = cert
		aterrant = fals
		numAtac = 1
	altrament si (podemFerDobleSalt)
		velocitat.y = velocitat_salt
		estat = estatDobleSalt
		tempsEstat = 0
		podemFerDobleSalt = fals
	fsi
ffuncio
```

Si es pot fer un salt es modifica la velocitat vertical amb la velocitat de salt, es canvia l'estat a saltar, es desactiva el salt, s'activa el doble salt, es marca que no esta aterrant i es resseteja el número d'atac. Si podem fer el doble salt es modificala velocitat vertical amb la velocitat de salt, es canvia l'estat a doble salt i es desactiva el doble salt.

`+noHitPlatform()`: Controla la lògica del personatge quan es troba a l'aire.

```
funcio noHitPlatform()
	si (estat != estatCarrega)
		si (estat != estatAtacat && estat != estatMorint)
			si (velocitat.y < 0)
				si ((velocitat.y >= velocitat_caure && estat !=
```

```
                        estatCaure) || (velocitat.y < velocitat_caure && estat
                        != estatCaiguent))
                              si (estat != estatCaure)
                                    estat = estatCaure
                              fsi
                              si (estat != estatCaiguent)
                                    estat = estatCaiguent
                              fsi
                              tempsEstat = 0
                              podemFerSalt = fals
                        fsi
                  altrament si (velocitat.y < velocitat_salt)
                        si (estat != estatSaltant)
                              estat = estatSaltant
                              tempsEstat = 0
                              podemFerSalt = fals
                              podemFerDobleSalt = cert
                        fsi
                  fsi
            fsi
      altrament
            si (tempsEstat >= tempsCarrega)
                  velocitat.x = 0
                  estat = estatAnterior
            fsi
      fsi
ffuncio
```

Si no està carregant i tampoc esta atacant ni s'està morint, es mira la velocitat vertical; si està descendent canvien l'estat a caiguent i desactivem el salt, i si estem ascendint canviem l'estat a saltant, desactivem el salt i activem el doble salt; si estem carregant i s'ha superat el temps de càrrega es canvia la velocitat horitzontal a 0 i es torna a l'estat anterior a la càrrega..

`touchPlatform()`: Controla la lògica del personatge al xocar contra una plataforma.

`update(float deltaTime)`: Actualitza la posició i la velocitat del personatge.

Multiplicitats

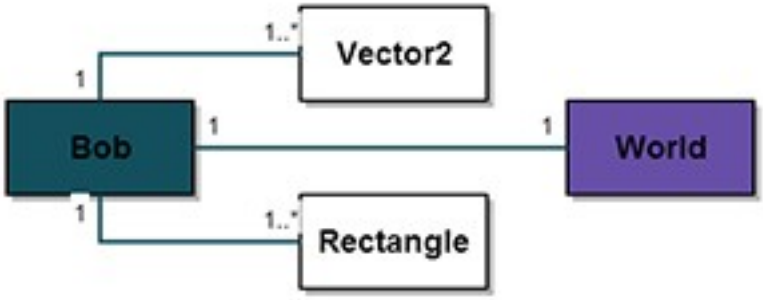


Figura 47: Multiplicitats Bob

8.6.5 Classe DynamicGameObject

Extén de "GameObject" i conté informació per objectes dinàmics de l'escena.

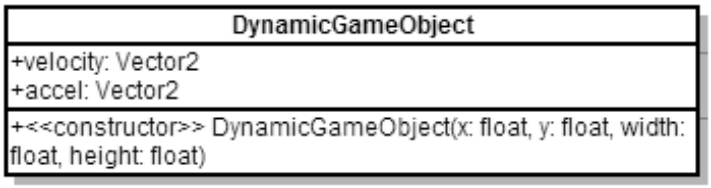


Figura 48: Classe DynamicGameObject

Atributs

`+velocity`: Variable que guarda la velocitat (x,y).

`+accel`: Variable que guarda l'acceleració (x,y).

Mètodes

`+DynamicGameObject(float x, float y, float width, float height)`: Constructor que crida el de la classe superior.

Multiplicitats

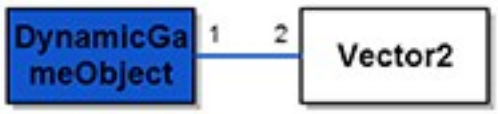


Figura 49: Multiplicitats DynamicGameObject

8.6.6 Classe Enemy

Extén de "Bob" i conté les dades necessàries pels enemics. S'ha fet que extengui dirèctament per conveniència ja que es més ràpid per implementar el prototip si reaprofitem el personatge principal per tenir enemics.

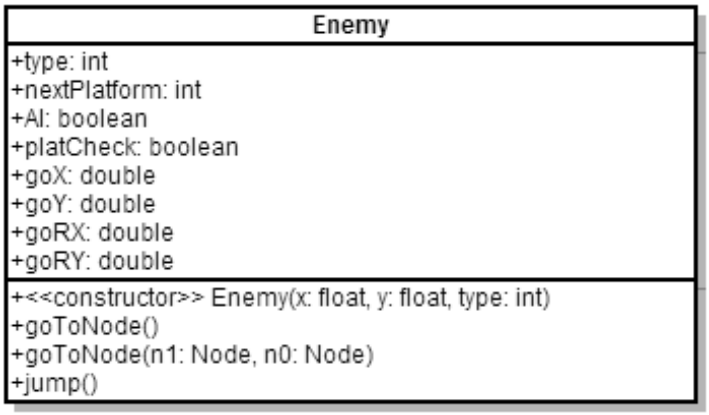


Figura 50: Classe Enemy

Atributs

`+type`: Variable que defineix el tipus.

`+nextPlatform`: Variable que guarda l'identificador de la següent plataforma a la que s'haurà de dirigir.

`+AI`: Variable que ens diu si s'ha de fer córrer l'algorisme per trobar el camí fins al jugador.

`+PlatCheck`: Variable que ens diu si hem de buscar amb quina plataforma anem a col·lisionar.

`+goX`: Variable que mostra a quina posició de l'eix de les “x” haurà d'anar.

`+goY`: Variable que mostra a quina posició de l'eix de les “y” haurà d'anar.

`+goRX`: Variable que mostra el marge d'error que té per posicionar-se horitzontalment

`+goRY`: Variable que mostra el marge d'error que té per posicionar-se verticalment.

Mètodes

`+Enemy(float x, float y, int type)`: Constructor, crea l'enemic a les coordenades x,y amb un tipus determinat.

`+goToNode()`: Dirigeix el personatge cap a una posició coneguda.

```
funcio goToNode()
      si (anarX == 0 && anarY == 0)
```

```
            anarX = enemics[0].posicio.x
            anarY = enemics[0].posicio.y
      fsi
      si (estat != estatAtacat && estat != estatMort)
            si (posicio.y > anarY)
                  si (dinsDelRangHoritzontal(posicio.x, anarX))
                        saltar()
                  fsi
            altrament
                  si (dinsDelRangHoritzontal(posicio.x, anarX) &&
                      dinsDelRangVertical(posicio.y, anarY))
                        saltar()
                  fsi
            fsi
            si (capALaDreta(posicio.x, anarX))
                  velocitat.x = velocitat_moviment
            altrament si (capALEsquerra(posicio.x, anarX))
                  velocitat.x = velocitat_moviment * -1
            altrament
                  si (estat != estatCorrer || vida <= 0)
                        velocitat.x = 0
                  fsi
                  si (estat != estatSaltar && estat != estatCaure)
                        IA = cert
                  fsi
            fsi
      fsi
ffuncio
```

Segons on sigui la posició on ha d'anar es diu al personatge si ha de saltar, incrementar la velocitat horitzontal positiva o negativament, o ambdues.

`+goToNode(Node n1, Node n0)`: Dirigeix el personatge cap a una posició nova.

`+Jump()`: Realitza un salt si es pot.

Multiplicitats

Figura 51: Multiplicitats Enemy

8.6.7 Classe Game

Classe abstracta que implementa l'"ApplicationListener" (7.3.1) i gestiona les pantalles del joc.

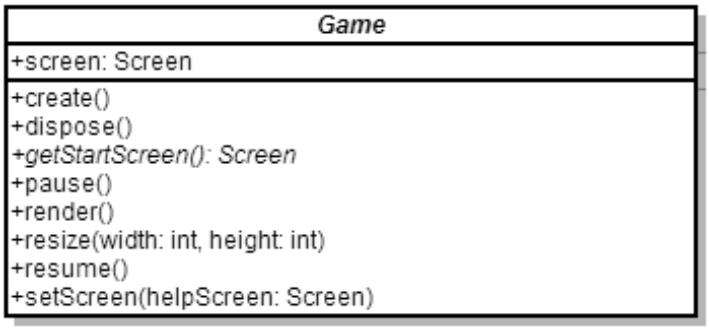


Figura 52: Classe Game

Atributs

`+screen`: Pantalla actual.

Mètodes

`+create()`: Carrega la pantalla d'inici "MainMenuScreen".

`+dispose()`: Crida la funció dispose() de la pantalla actual.

`+getStartScreen()`: Crida la pantalla d'inici.

`+pause()`: Crida la funció pause() de la pantalla actual.

`+render()`: Crida les funcions update(float) i present(float) de la pantalla actual.

`+resume()`: Crida la funció resume() de la pantalla actual.

`+setScreen(Screen helpScreen)`: Para la pantalla actual i la canvia per la nova.

Multiplicitats

Figura 53: Multiplicitats Game

8.6.8 Classe GameObject

Determina un objecte del joc.

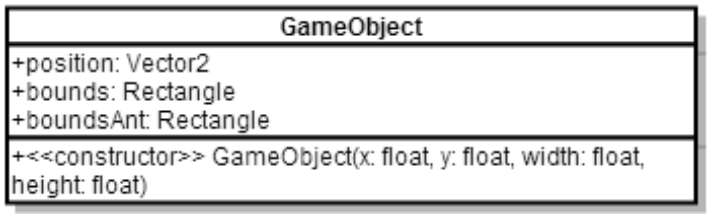


Figura 54: Classe GameObject

Atributs

`+position`: Variable per indicar la posició (x,y).

`+bounds`: Variable per indicar el perímetre i la posició amb un rectangle.

`+boundsAnt`: Variable per indicar el perímetre i la posició anterior amb un rectangle.

Mètodes

`+GameObject(float x, float y, float width, float height)`: Constructor

Multiplicitats

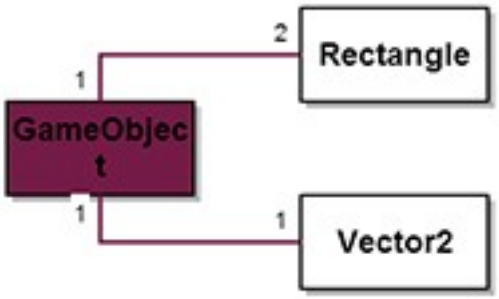


Figura 55: Multiplicitats GameObject

8.6.9 Classe Geometry

Conté mètodes especials per operar amb formes geomètriques

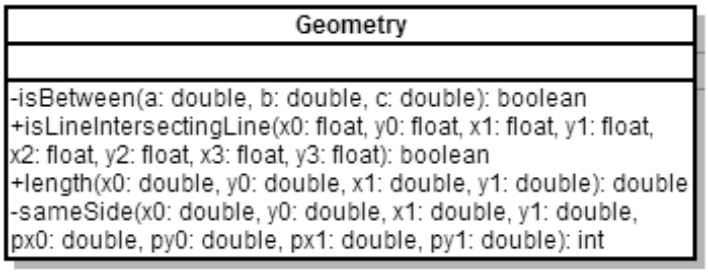


Figura 56: Classe Geometry

Mètodes

`-isBetween(double a, double b, double c)`: Mira si ”c” es troba entre “a” i “b”.

`+isLineIntersectingLine(float x0, float y0, float x1, float y1, float x2, float y2, float x3, float y3)`: Mira si dos segments intersequen.

`+length(double x0, double y0, double x1, double y1)`: Retorna la distància entre dos punts.

`-sameSide(double x0, double y0, double x1, double y1, double px0, double py0, double px1, double py1)`: Mira si dos punts es troben en el mateix costat d'una línia recta.

Multiplicitats (cap)

8.6.10 Classe GameScreen

Extén de “Screen” i és responsable de gestionar la pantalla de joc.

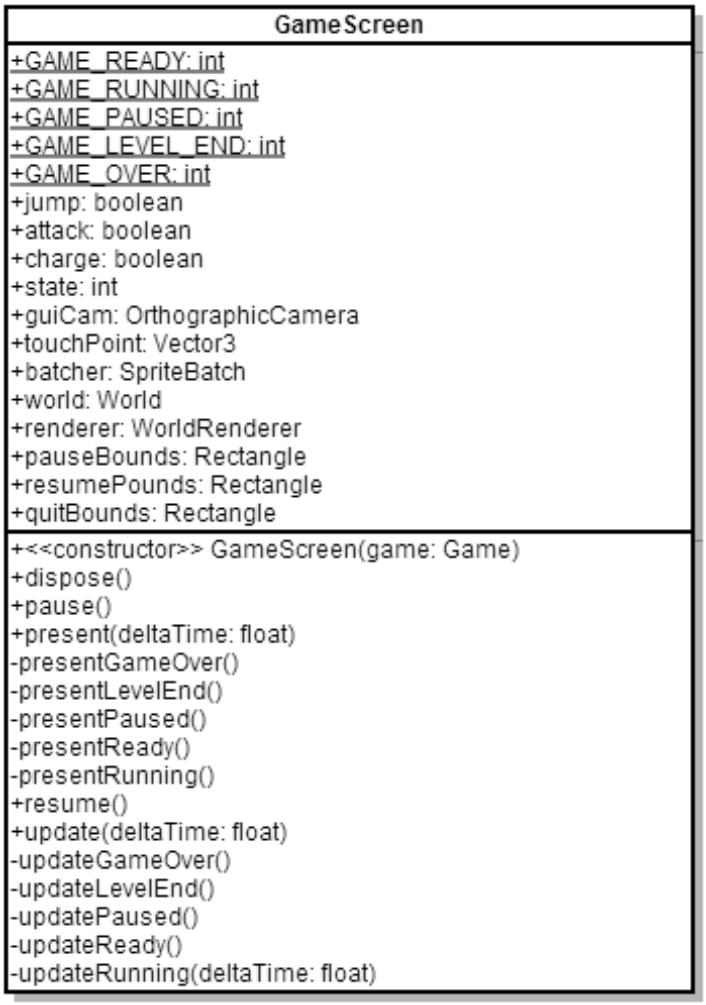


Figura 57: Classe GameScreen

Atributs

`+GAME_READY`: Constant per saber l'estat de la pantalla quan estigui preparada.

`+GAME_RUNNING`: Constant per saber l'estat de la pantalla quan estigui en marxa.

`+GAME_PAUSED`: Constant per saber l'estat de la pantalla quan estigui pausada.

`+GAME_LEVEL_END`: Constant per saber l'estat de la pantalla quan estigui finalitzada.

`+GAME_OVER`: Constant per saber l'estat de la pantalla quan s'hagi perdut.

`+jump`: Variable per saber si es pot saltar o no.

`+attack`: Variable per saber si es pot atacar o no.

`+charge`: Variable per saber si es pot realitzar una càrrega o no.

`+state`: Variable per saber l'estat actual de la pantalla.

`+guiCam`: Càmera de la pantalla.

`+touchPoint`: Punt per saber on es toca la pantalla.

`+batcher`: Búffer de textures.

`+world`: Món de la pantalla.

`+renderer`: Projector dels elements de l'escena.

`+pauseBounds`: Límits de la zona de la pantalla per pausar el joc.

`+resumeBounds`: Límits de la zona de la pantalla per reprendre el joc.

`+quitBounds`: Límits de la zona de la pantalla per sortir del joc.

Mètodes

`+GameScreen(Game game)`: Constructor.

`+pause()`: Canvia l'estat de la pantalla a pausada.

`+present(float deltaTime)`: Dibuixa els elements de l'escena i de la interfície.

`-presentGameOver()`: Dibuixa el menú de quan s'ha perdut.

`-presentLevelEnd()`: Dibuixa el menú de fi del joc.

`-presentPaused()`: Dibuixa el menú de pausa.

`-presentReady()`: Dibuixa el menú d'estar preparat.

`-presentRunning()`: Dibuixa els elements de la interfície.

`+update(float deltaTime)`: Actualitza el temps de la pantalla.

`-updateGameOver()`: S'ocupa de la interacció amb el menú de “game over”.

`-updateLevelEnd()`: S'ocupa de la interacció amb el menú de fi del joc.

`-updatePaused()`: S'ocupa de la interacció amb el menú de pausa.

`-updateReady()`: S'ocupa de la interacció amb el menú d'estar preparat.

`-updateRunning(float deltaTime)`: S'ocupa de la interacció amb el joc.

Multiplicitats

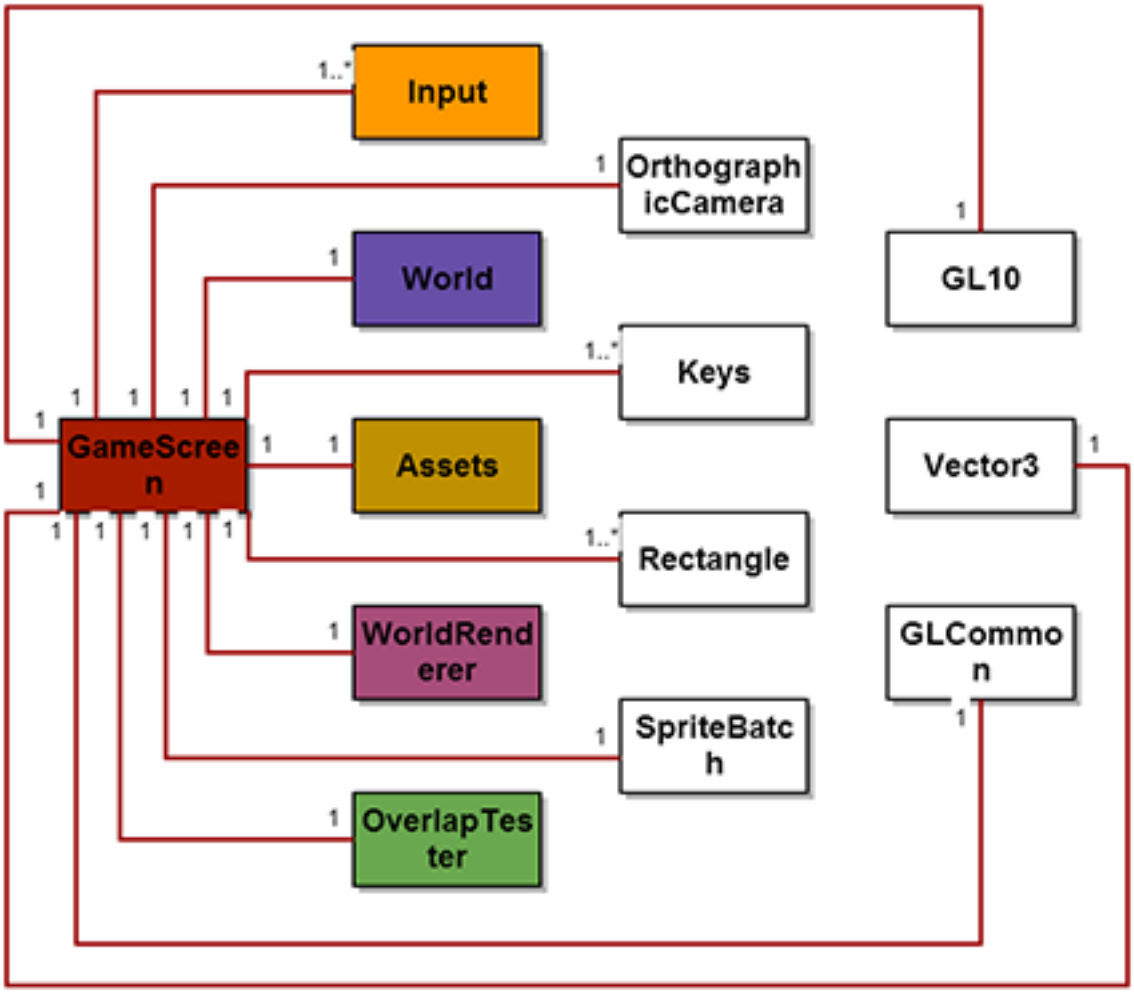


Figura 58: Multiplicitats GameScreen

8.6.11 Classe MainMenuScreen

Extén de “Screen” i és responsable de gestionar la pantalla del menú d'inici.

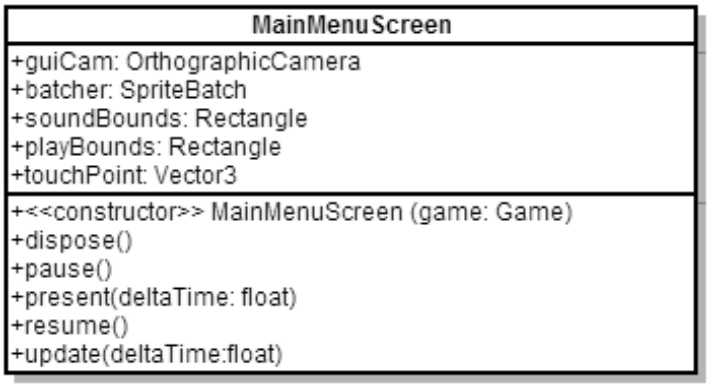


Figura 59: Classe MainMenuScreen

Atributs

`+guiCam`: Càmera de la pantalla.

`+batcher`: Búffer de textures.

`+soundBounds`: Límits de la zona de la pantalla per parar o engegar el so.

`+playBounds`: Límits de la zona de la pantalla per engegar la pantalla de joc.

`+touchPoint`: Punt per saber on es toca la pantalla.

Mètodes

`+MainMenuScreen(Game game)`: Constructor.

`+pause()`: Guarda la configuració actual.

`+present(float deltaTime)`: Dibuixa tots els elements de la pantalla.

Multiplicitats

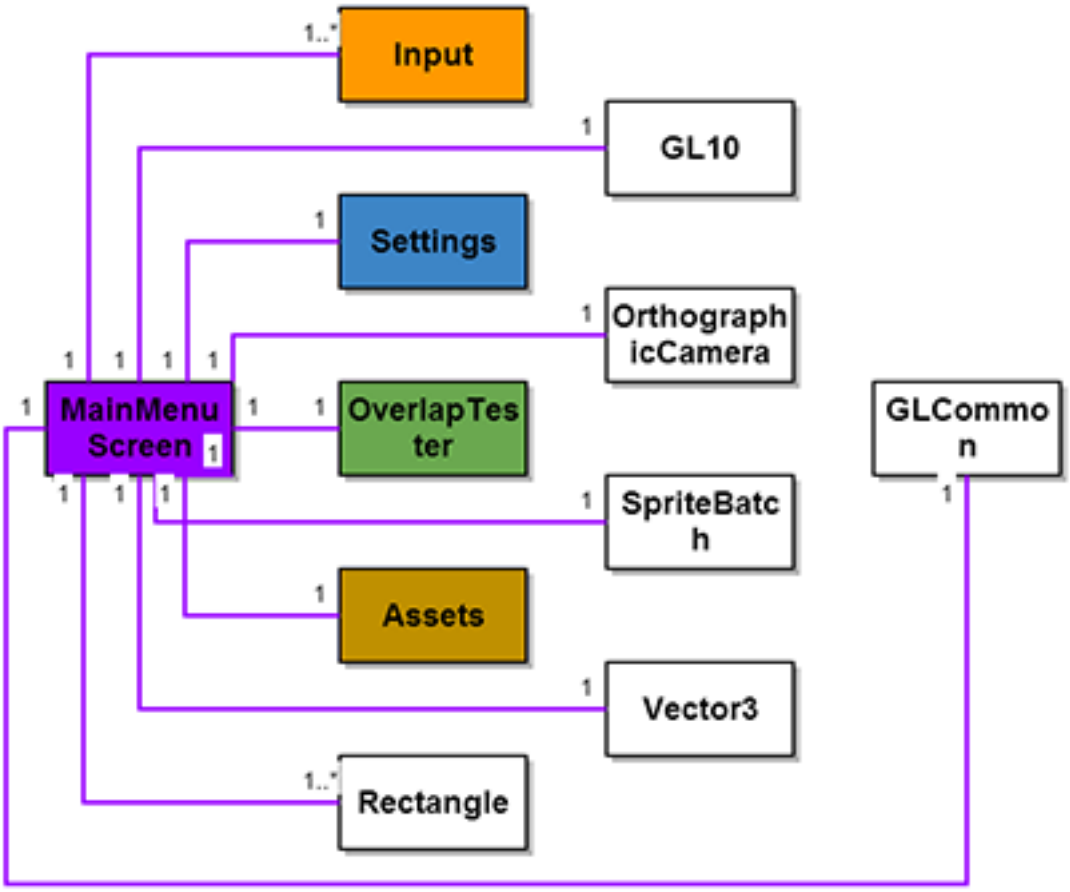


Figura 60: Multiplicitats MainMenuScreen

8.6.12 Classe MonstrumOccidere

Extén de “Game” i s'ocupa de carregar els recursos de l'aplicació

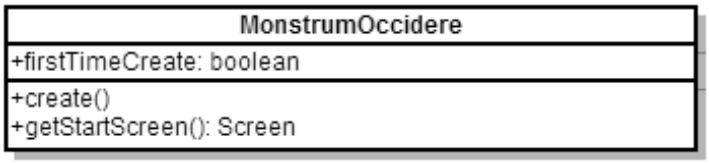


Figura 61: Classe MonstrumOccidere

Atributs

`+firstTimeCreate`: Variable per veure si és la primera vegada que es crea.

Mètodes

`+getStartScreen()`: Crea la pantalla de “MainMenuScreen”.

`+create()`: Carrega les configuracions, tots els recursos que necessitarà l'aplicació i crida el constructor de la classe pare (Game).

Multiplicitats

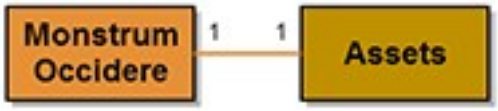


Figura 62: Multiplicitats MonstrumOccidere

8.6.13 Classe MonstrumOccidereDesktop

Crea l'aplicació amb les mides i el nom donats

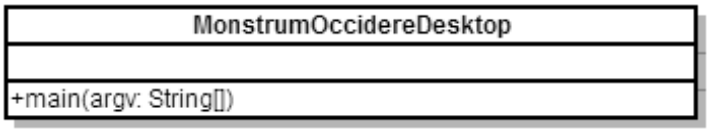


Figura 63: Classe MonstrumOccidereDesktop

Mètodes

`+main(String[] argv)`: Inicialitza l'aplicació.

Multiplicitats

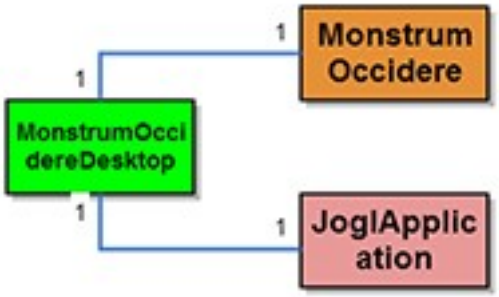


Figura 64: Multiplicitats MonstrumOccidereDesktop

8.6.14 Classe OverlapTester

Conté mètodes especials per determinar les col·lisions.

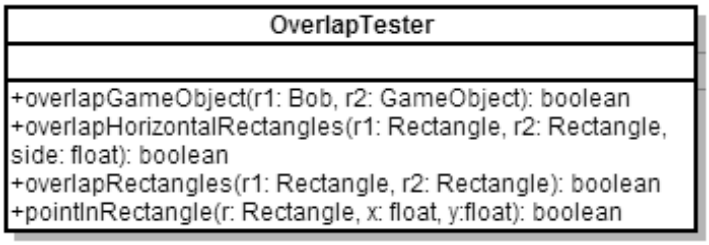


Figura 65: Classe OverlapTester

Mètodes

`+overlapGameObject(Bob r1, GameObject r2)`: Mira si la part inferior del personatge ha travessat la part superior del "GameObject".

```
funcio overlapGameObject(r1: Bob, r2: GameObject)
      si (Geometria.elSegmentIntersecaSegment(
          r1.posicioAnterior.x + r1.posicio.amplada/2, r1.posicioAnterior.y,
          r1.posicio.x + r1.posicio.amplada, r1.posicio.y,
          r2.posicio.x, r2.posicio.y + r2.posicio.altura,
          r2.posicio.x + r2.posicio.amplada, r2.posicio.y + r2.posicio.altura))
            retorna cert
      altrament
            retorna fals
      fsi
ffuncio
```

`+overlapHorizontalRectangles(Rectangle r1, Rectangle r2, float side)`: Mira si dos rectangles intersequen lateralment per un dels costats.

`+overlapRectangles(Rectangle r1, Rectangle r2)`: Mira si dos rectangles intersequen.

`+pointInRectangle(Rectangle r, float x, float y)`: Mira si un punt és dins d'un rectangle.

Multiplicitats

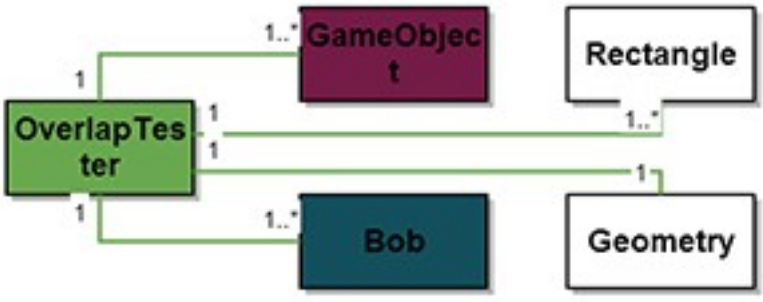


Figura 66: Multiplicitats OverlapTester

8.6.15 Classe Screen

Classe abstracta per crear les diferents pantalles del joc.

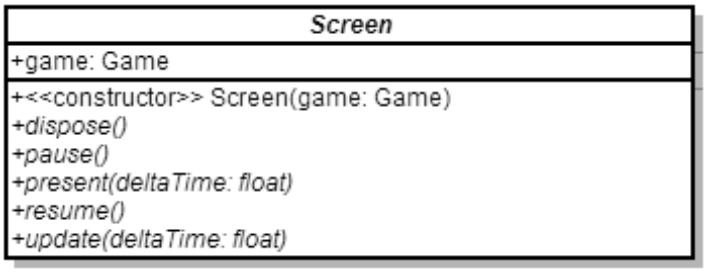


Figura 67: Classe Screen

Atributs

`+game`: Joc al que pertany la pantalla.

Mètodes

`+Screen(Game game)`: Constructor.

Multiplicitats

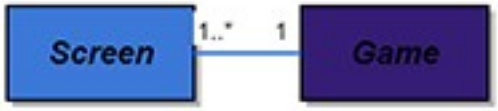


Figura 68: Multiplicitats Screen

8.6.16 Classe Settings

S'encarrega de guardar i carregar les configuracions de l'aplicació.

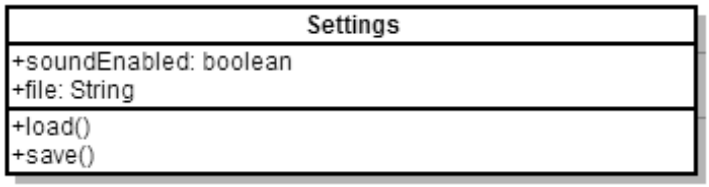


Figura 69: Classe Settings

Atributs

`+soundEnabled`: Variable per veure si el so està habilitat o no.

`+file`: Nom del fitxer que s'utilitza per guardar les configuracions.

Mètodes

`+load()`: Carrega la configuració guardada.

`+save()`: Salva la configuració actual.

Multiplicitats

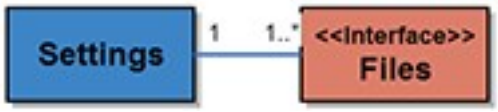


Figura 70: Multiplicitats Settings

8.6.17 Classe Platform

Extén de “DynamicGameObject” i crea i gestiona les plataformes

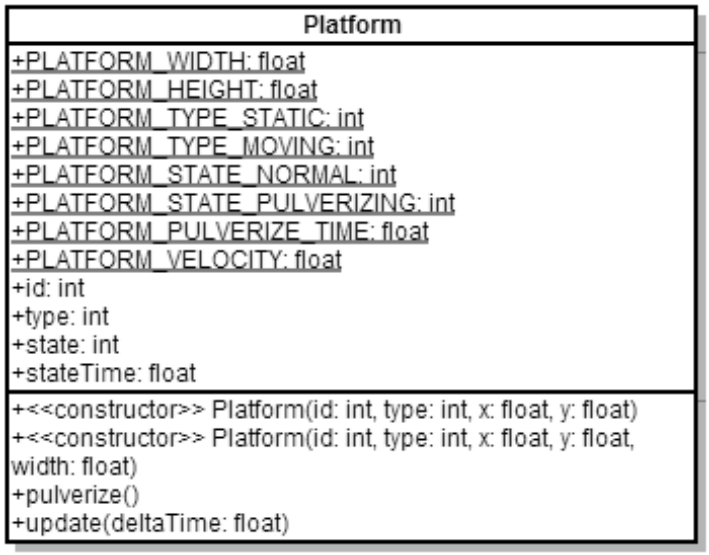


Figura 71: Classe Platform

Atributs

`+PLATFORM_WIDTH`: Constant que indica l'amplada de les plataformes.

`+PLATFORM_HEIGHT`: Constant que indica l'alçada de les plataformes.

`+PLATFORM_TYPE_STATIC`: Constant que indica el tipus de plataforma estàtica.

`+PLATFORM_TYPE_MOVING`: Constant que indica el tipus de plataforma mòbil.

`+id`: Variable identificadora de la plataforma.

`+type`: Variable del tipus de plataforma.

`+state`: Variable de l'estat de la plataforma.

`+stateTime`: Variable del temps d'estat.

Mètodes

`+Platform(int id, int type, float x, float y)`: Constructor de plataforma estàndard.

`+Platform(int id, int type, float x, float y, float width)`: Constructor de plataforma amb amplada donada.

`update(float deltaTime)`: Actualitza l'estat i la posició de la plataforma si es mou.

Multiplicitats (cap)

8.6.18 Classe World

Crea el món del joc amb tots els actors i recursos.

Figura 72: Classe World

Atributs

`+WORLD_WIDTH`: Constant de l'amplada del món.

`+WORLD_HEIGHT`: Constant de l'alçada del món.

`+WORLD_STATE_RUNNING`: Constant de l'estat del món quan el joc es troba en marxa.

`+WORLD_STATE_NEXT_LEVEL`: Constant de l'estat del món quan ha d'anar al següent nivell.

`+WORLD_STATE_GAME_OVER`: Constant de l'estat quan el joc s'ha perdut.

`+gravity`: Constant amb el valor de la gravetat.

`+bob`: Personatge principal.

`+platforms`: Vector de plataformes que formen part del món.

`+walls`: Vector de plataformes utilitzades com a parets.

`+graph`: Graf que utilitzarà la intel·ligència artificial del joc.

`+rand`: Objecte de classe “Random” per events aleatoris.

`+background`: Fons del món.

`+score`: Variable per guardar la puntuació.

`+state`: Variable per guardar l'estat del món.

Mètodes

`+World()`: Constructor.

`+checkAI()`: Diu als enemics a quina posició han d'anar.

El més important d'aquest cas és el fet de moure els personatges eficientment per l'escenari i les plataformes. Així doncs es decideix utilitzar una implementació basada en l'algoritme de cerca **A*** (A-Star), que determina el camí mes curt entre dos nodes d'un graf (Figura 39). És semblant a l'algorisme de Dijkstra, la diferència resideix en que l'A* intenta trobar el camí més òptim a través d'una funció heurística, mentre que Dijkstra explora totes les possibilitats.

```
funcio checkAI()
	actual: Enemic
	it: Iterador<Bob> = enemics.iterador()
	astar: Astar
	cami: Cami
	n1, n0: Node
	mentre (it.hiHagiSeguent()) fer
		actual = (Enemic)it.seguent()
		si (actual.IA)
			astar = nou Astar(graf)
			astar.comutar(nodeMesProper(actual), nodeMesProper(bob))
			cami = astar.camiMesCurt()
			si (cami.mida() > 1) n1 = cami.getNode(1)
			altrament n1 = nul  fsi
			si (cami.mida() > 0) n1 = cami.getNode(0)
			altrament n0 = nul fsi
			actual.vesAlNode(n1,n0)
		altrament
			actual.vesAlNode()
		fsi
	fmentre
ffuncio
```

Per cada enemic executa l'algoritme A* entre el node més proper de l'enemic actual i el node més proper del jugador, per determinar el camí més curt i indicar a quin node del

graf s'ha de dirigir per recórrer el camí.

`-checkCollisions(float deltaTime)`: Comprova totes les col·lisions

`-checkEnemiesPlatformCollisions(float deltaTime)`: Comprova les col·lisions dels enemics amb les plataformes.

`-checkEnemiesWallCollisions(float deltaTime)`: Comprova les col·lisions dels enemics amb les parets.

`-checkPlatformCollisions(float deltaTime)`: Comprova les col·lisions del personatge principal amb les plataformes.

```
funcio checkPlatformCollisions(temps: float)
	tocada: boolea
	plataforma: Plataforma
	si (velocitatVertical() > 0) retorna fsi
	tocada = fals
	si (velocitatVertical() < 0 || ultimaPlataformaTocada() == 0)
		mentre (hi hagi plataformes) fer
			plataforma = plataformes.plataformaSeguent()
			si (posicioVertical() >= plataforma.posicioVertical())
				si (rectangle.sobreposa(plataforma.rectangle))
					velocitatVertical(0)
					ultimaPlataformaTocada(plataforma.id())
					tocada = cert
					hemTocatPlataforma()
					sortir
				fsi
			fsi
		fmentre
		si (no tocada) noHemTocatPlataforma() fsi
	altrament
		plataforma = plataformes.ultimaPlataformaTocada()
		si (rectangle.sobreposa(plataforma.rectangle))
			velocitatVertical(0)
			hemTocatPlataforma()
		altrament
			noHemTocatPlataforma()
		fsi
	fsi
ffuncio
```

Aquest algoritme farà un recorregut per les plataformes, i per cada una cridarà la funció de col·lisió amb la posició actual dels personatges en escena. Si detecta col·lisió quan el personatge es troba descendent, ho reportarà al personatge, que realitzarà el canvi de velocitat a zero, quedant-se sobre la plataforma.

`-checkGameOver():` Comprova que no s'hagi perdut el joc.

`-checkWallCollisions(float deltaTime)`: Comprova les col·lisions amb les parets.

`-closestNode(Bob act)`: Determina quin és el node del graf més proper.

`-generateEnemies()`: Genera els enemics de l'escenari.

`-generateGraph()`: Genera el graf que utilitzarà la intel·ligència artificial dels enemics per trobar el camí més curt fins al personatge principal.

`-generateLevelProves()`: Genera les plataformes i parets de l'escenari.

`+provaCamins()`: Algorisme que comprova tots els camins més curts del graf generat.

`+update(float deltaTime, float accelX)`: Actualitza l'estat del món.

`-updateBob(float deltaTime, float accelX)`: Actualitza el personatge principal.

`-updateEnemies(float deltaTime)`: Actualitza els enemics.

Multiplicitats

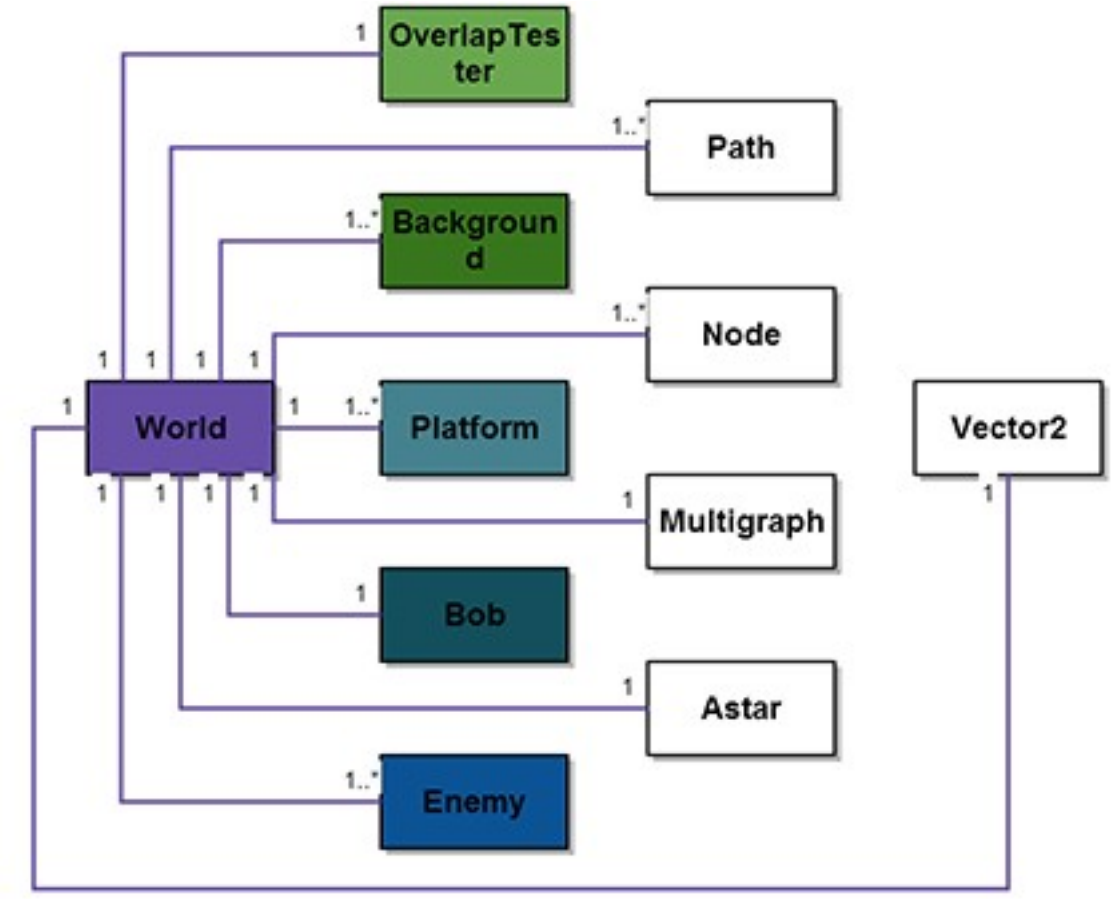


Figura 73: Multiplicitats World

8.6.19 Classe WorldRenderer

S'encarrega d'actualitzar el món i pintar-lo per pantalla.

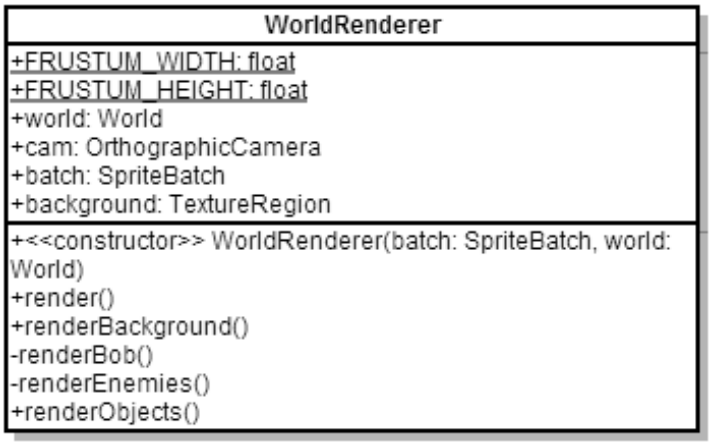


Figura 74: Classe WorldRenderer

Atributs

`+FRUSTUM_WIDTH`: Constant de l'amplada del rang de la càmera.

`+FRUSTUM_HEIGHT`: Constant de l'alçada del rang de la càmera.

`+world`: Món del joc.

`+cam`: Càmera.

`+batch`: Búffer de textures.

`+background`: Textura de fons del món.

Mètodes

`+WorldRenderer(SpriteBatch batch, World world)`: Constructor.

`+render()`: Actualitza la posició de la càmera i mostra per pantalla el fons i els objectes de l'escena.

`+renderBackground()`: Mostra per pantalla el fons de l'escena.

`-renderBob()`: Mostra per pantalla el personatge principal.

`-renderEnemies()`: Mostra per pantalla els enemics.

`+renderObjects()`: Mostra per pantalla els objectes de l'escena.

Multiplicitats

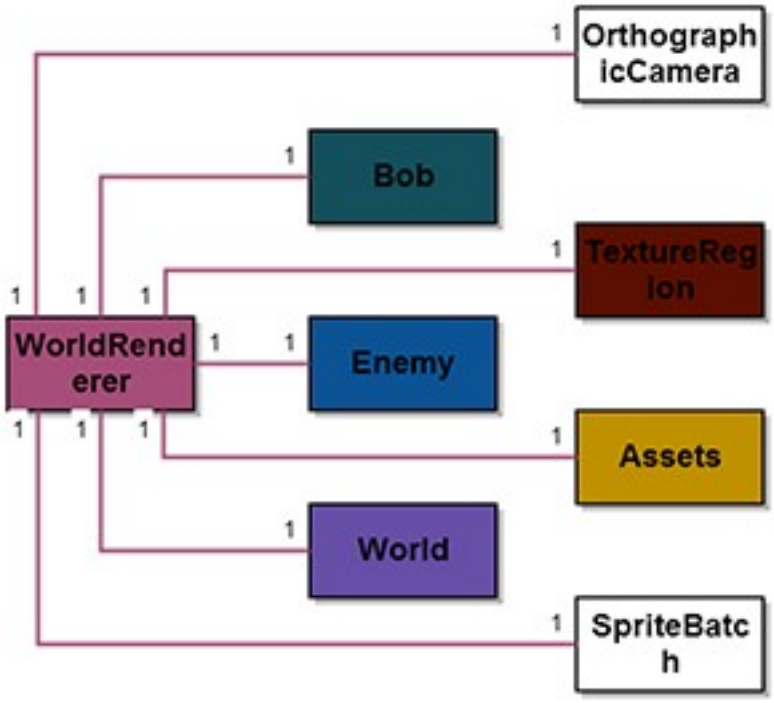


Figura 75: Multiplicitats WorldRenderer

8.7 Diagrama de Seqüència a l'inici de l'aplicació (Android/Libgdx)

Per complementar el que hem explicat sobre l'arquitectura de l'aplicació Android/LibGDX (apartat 7.3) veurem un diagrama de seqüència simplificat (Figura 76) on es mostra l'inici de l'aplicació, la càrrega d'arxius, la creació del segon thread, la creació de les diferents pantalles i com s'interactua amb l'usuari.

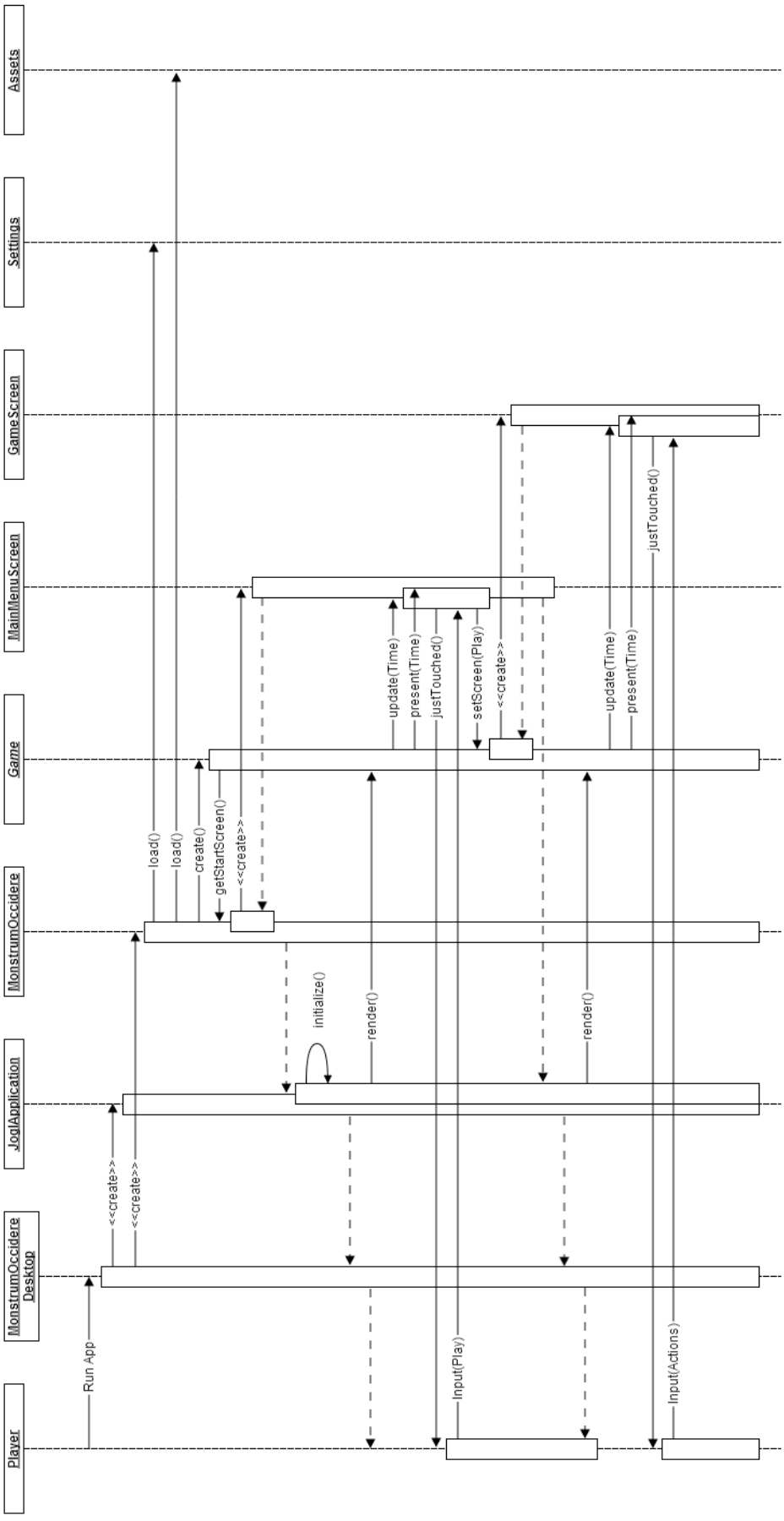


Figura 76: Diagrama de seqüència a l'inici de l'aplicació Android

8.8 Diagrama de classes (Unity)

Com s'ha exposat a l'inici de l'apartat 8.6, donat que Unity és un motor de disseny de videojocs complet, no hem entrat en la modificació de la seva arquitectura. A diferencia del funcionament del projecte que hem realitzat en Android, on s'utilitza **l'herència** com a base, Unity es basa en la **composició d'objecte.**

Tots els elements de Unity son **Game Objects**, als quals se'ls hi afegeixen diferents components per modificar-ne el comportament. Aquests components estan basats en arxius **Javascript**, i tot i que s'utilitzen classes dins d'ells, aquestes al final són molt reduïdes donat que s'utilitzen bàsicament per organitzar grups de variables (veure Figura 111 i Figura 112).

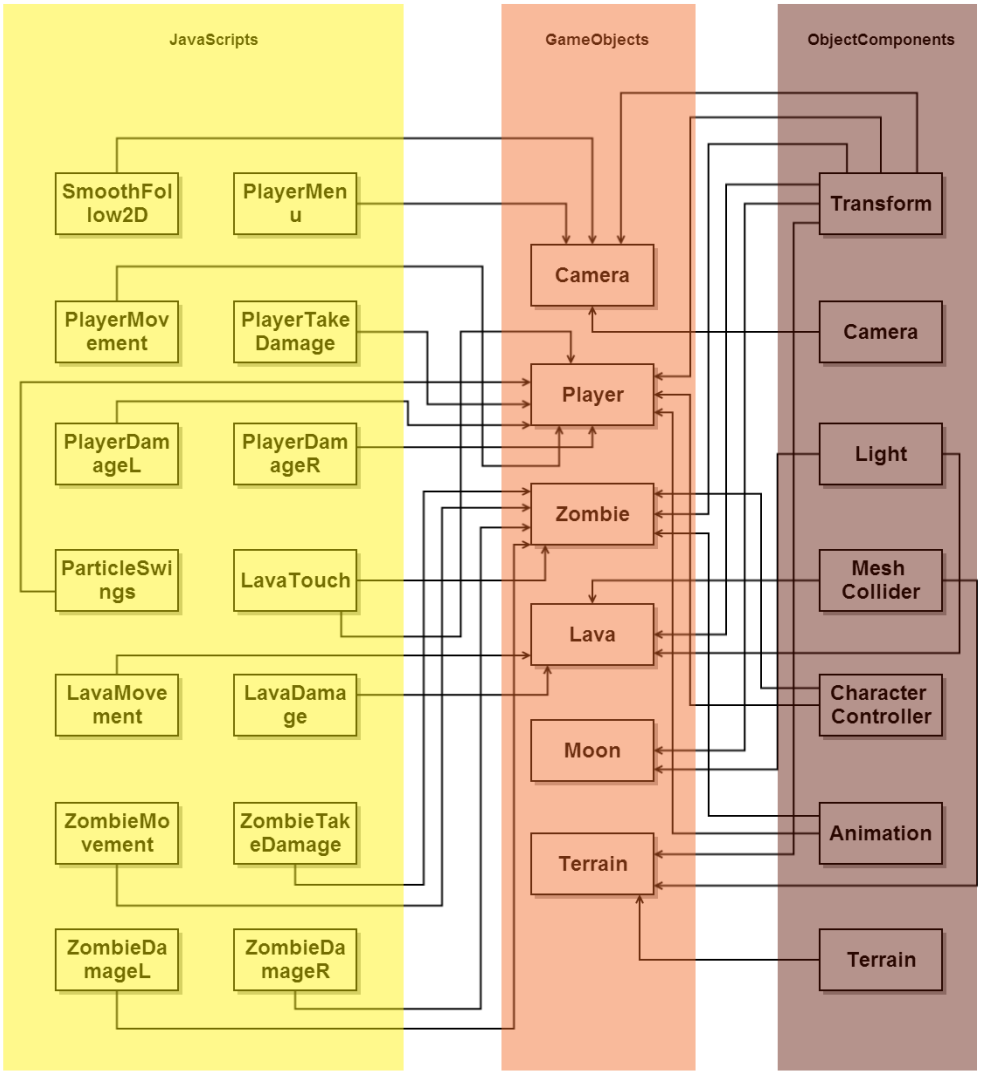


Figura 77: Relacions entre Javascripts, GameObjects i ObjectComponents

El que farem en aquest apartat és un diagrama dels elements del projecte (Game Objects, Scripts, Object Components) que podem identificar dins de l'entorn de desenvolupament i que equivaldria, en aquest cas, al diagrama de classes (Figura 77).

Aixi doncs com podem observar, els requadres vermells son arxius Javascript, els requadres blaus son Object Components i els verds representen els Game Objects.

Farem un repas del contingut dels arxius Javascript, on s'ha realitzat la codificació dels elements.

8.8.1 SmoothFollow2D

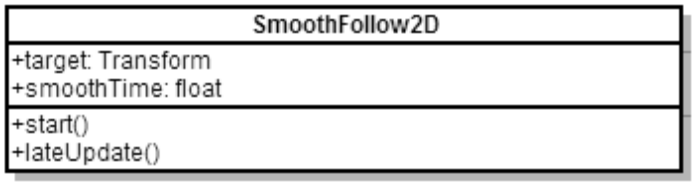


Figura 78: SmoothFollow2D.js

Variables

`+target: Transform //` Posició de l'objectiu al que segueix la càmera.

`+smoothTime: float //` Temps que es tarda en suavitzar el moviment de la càmera.

Funcions

`+start():` Inicialitza la posició de la càmera al iniciar-se l'aplicació.

`+lateUpdate():` Actualitza la posició x,y de la càmera després que el personatge s'hagi mogut.

8.8.2 PlayerMenu

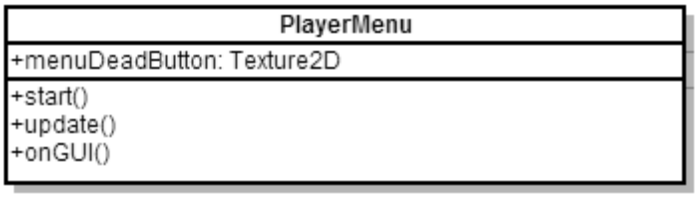


Figura 79: PlayerMenu.js

Variables

`+menuDeadButton: Texture2D //` Textura per definir l'aspecte gràfic del botó.

Funcions

`+start():` Determina l'estat del menú al iniciar-se l'aplicació.

`+update():` Actualitza l'estat del menú mirant si hi ha hagut input.

`+onGUI():` Si s'ha pausat mostra la interfície del menú.

8.8.3 PlayerTakeDamage

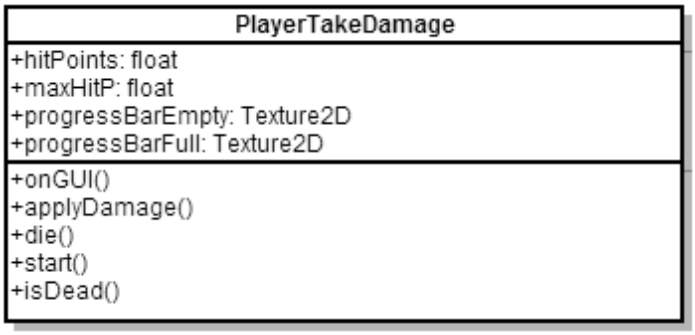


Figura 80: PlayerTakeDamage.js

Variables

`+hitPoints: float //` Nombre de punt de vida.

`+maxHitP: float //` Nombre màxim de punts de vida.

`+progressBarEmpty: Texture2D //` Textura per la part buida de la barra de vida.

`+progressBarFull: Texture2D //` Textura per la part plena de la barra de vida.

Funcions

`+onGUI():` Dibuixa la barra de vida.

`+applyDamage():` Actualitza els punts de vida quan ataquen al personatge.

`+die():` Indica al personatge que mori.

`+start():` Inicialitza la barra de vida a l'inici de l'aplicació.

+isDead(): Retorna si el personatge esta mort o no.

8.8.4 PlayerMovement

Figura 81: PlayerMenu.js

Variables

`+canControl: boolean //` Respon al input?

`+isDead: boolean //` El personatge és mort?

`+movement: controllerMovement //` Conté les variables relacionades amb el moviment:

- `+runSpeed: float //` La velocitat al córrer.
- `+slideFactor: float //` La velocitat quan ens llisquem pels marges.
- `+gravity: float //` La gravetat pel personatge.
- `+maxFallSpeed: float //` Màxima velocitat de caiguda.
- `+speedSmoothing: float //` Com de ràpid canvia el personatge de velocitats.
- `+rotationSmoothing: float //` Com de ràpid el personatge tarda en girar-se.
- `+isMoving: boolean //` Estem prement els botons de direcció del personatge?
- `-direction: Vector3 //` La direcció actual de moviment en x/y.
- `-verticalSpeed: float //` La velocitat vertical actual
- `-speed: float //` La velocitat horitzontal actual
- `-hangTime: float //` El temps que el personatge es troba sospés en l'aire.

`+jump: controllerJumping //` Conté les variables relacionades amb el salt:

- `+enabled: boolean //` Es pot saltar?
- `+doubleJump: boolean //` Es pot fer el doble salt?
- `-landing: boolean //` Estem aterrant?
- `+height: float //` Altura que saltem quan premem el boto i el deixem anar.
- `+extraHeight: float //` Altura extra quan deixem el botó premut.
- `+speedSmoothing: float //` Quan ràpid canvia el personatge de velocitats.
- `+jumpSpeed: float //` Velocitat del personatge quan es troba a l'aire.
- `-repeatTime: float //` Temps que ha de passar abans que podem tornar a saltar.
- `+jumping: boolean //` Estem saltant?
- `-lastButtonTime: float //` Última vegada que hem premut el botó de salt.

-lastGroundedTime: float // Última vegada que hem aterrat.

-groundingTimeout: float // Temps que es tarda en aterrar.

-lastTime: float // Última vegada que s'ha saltat.

-lastStartHeight: float // L'altura des de la que hem saltat.

-touchedCeiling: boolean // Hem tocat sostre?

-buttonReleased: boolean // El botó esta alliberat?

+charge: controllerCharging // Conté les variables relacionades amb les càrregues:

+blocking: boolean // Estem bloquejant?

+rolling: boolean // Estem rodolant?

-rollAnim: boolean // Animació de rodolar activada?

-rollattack: boolean // Atac rodolant actiu?

+charging: boolean // Estem carregant?

+chargeMovingSpeed: float // La velocitat quan carreguem.

-chargeCooldown: float // Interval de temps per a realitzar una altre càrrega.

-chargeTimeCasted: float // Última vegada que hem carregat.

-chargeAnim: boolean // Animació de carrega activada?

+attack: controllerAttacking // Conté les variables relacionades amb els atacs:

+attacking: boolean // Estem atacant?

+attackair: boolean // Estem atacant en l'aire?

-attackTimeCasted: float // Temps de la realització de l'últim atac.

+attackAnim: boolean // Es pot activar una animació d'atac?

+runattack: controllerRunAttacking // Conté les variables relacionades amb els atacs quan es mou:

+runattacking: boolean // Estem atacant i corrent?

-runattackTimeCasted: float // Temps de la realització de l'últim atac, corrent.

+runattackAnim: boolean // Es pot activar una animació d'atac, corrent?

controller: CharacterController // Referencia al controlador del personatge

Com havíem explicat a l'apartat 8.4.2, el propi Unity utilitza aquests controladors de personatges per determinar com intersequen amb l'entorn i entre ells, poguent deshabilitar les col·lisions entre diferents elements per mitjà d'scripts, associats a cada un dels personatges i on també es determina la gravetat.

Tinguent en compte la gravetat i l'Input donat pel jugador calculen la velocitat vertical i horitzontal; després, amb les dades relacionades de les col·lisions es determina la velocitat final. Els controladors de personatge també permeten determinar quina altura màxima poden superar sense provocar una col·lisió, si es desplacen horitzontalment per un terreny abrupte.

Funcions

+updateSmoothMovementDirection(): Actualitza la velocitat i direcció del personatge.

+start(): Configura les animacions a l'inici de l'aplicació.

+animateCharacter(): Conté la lògica per determinar quina animació s'ha de reproduir.

+justBecameUngrounded: Mira si es pot considerar que ja no ens trobem sobre terra.

+applyJumping(): Aplica la lògica de salt quan premem el botó de salt.

```
funcio ApplyJumping ()
      aterrat: boolea

      si (Input.ObtenirBotoPremut("Saltar") && esPotControlar())
            salt.tempsUltimApretat = Temps.tempsActual();
      fsi
      // Evita saltar massa ràpid després de cada salt.
      si (salt.ultimaVegada + salt.tempsPerRepetir > Temps.tempsActual())
            retorna
      fsi
      aterrat = controladorPersonatge.esAterrat();
      // Permet saltar lleugerament després que el personatge abandoni una
      // cornisa sempre i quan no haguem saltat prèviament.
      si (aterrat || justNoEstemAterrats())
            si (aterrat) fer
                  salt.ultimaVegadaAterrat = Temps.tempsActual()
                  si (atac.atacAire)
```

```
                        atac.atacAire = fals
                        salt.aterrant = cert
                  fsi
                  salt.dobleSalt = cert
            fsi
            // Posem un temps de marge per poder prémer el botó lleugerament
            // abans d'aterrar
            si (salt.habilitat && Temps.tempsActual() < salt.tempsUltimApretat
             + salt.tempsMarge)
                  moviment.velocitatVertical =
                   calcularSaltVelocitatVertical(salt.altura)
            fsi
      altrament si (salt.soblesalt && Input.BotoPremut("Saltar") &&
       esPotControlar)
            moviment.velocitatVertical =
             calcularSaltVelocitatVertical(salt.altura)
            salt.dobleSalt = fals
            salt.saltant = cert
      fsi
ffuncio
```

`+applyCharge()`: Aplica la lògica de càrrega quan premem el botó de càrrega.

```
function ApplyCharge () {
      si (carrega.tempsCarregaFeta == 0)
            carrega.tempsCarregaFeta.Temps.tempsActual()
      fsi
      si (Input.ObtenirBotoPremut ("Carrega") && esPotControlar())
            si (Temps.tempsActual() - carrega.tempsCarregaFeta >=
             carrega.TempsRefredamentCarrega)
                  carrega.animacioCarrega = cert
                  si (moviment.enMoviment && NO salt.saltant)
                        moviment.velocitat = carrega.velocitatCarrega
                        moviment.velocitatVertical = carrega.velocitatVertical
                  altrament si (moviment.enMoviment && salt.saltant && NO
                   atac.atacAire)
                        moviment.velocitat = carrega.velocitatSaltCarrega
                        moviment.velocitatVertical = 1
                  fsi
                  carrega.carregant = cert; carrega.tempsCarregaFeta = 0
            fsi
      fsi
      si (moviment.velocitat <= velocitatPararCarrega)
            carrega.carregant = fals
      fsi
      carrega.velocitat = moviment.velocitat
ffuncio
```

`+applyAttack()`: Aplica la lògica d'atac quan es prem el botó d'atac.

`+applyRunAttack()`: Aplica la lògica d'atac mentre el personatge corre i es prem el botó d'atac.

`+applyGravity()`: Aplica la gravetat al personatge.

```
function ApplyGravity () {
	botoSaltar: boolea
	potenciaSaltExtra: boolea

	botoSaltar = Input.getBoto(“Salt”)
	si (NO esPotControlar)
		botoSaltar = fals
	fsi
	si (salt.saltant && NO salt.apex && moviment.velocitatVertical <= 0)
		salt.apex = cert
	fsi
	si (NO botoSaltar)
		salt.botoAlliberat = cert
	fsi
	//Permet més precisió al saltar. Si premem més el botó saltarem més.
	potenciaSaltExtra = salt.saltant && moviment.velocitatVertical > 0 &&
	 botoSaltar && NO salt.botoAlliberat && jo.posicio.y < salt.ultimaPosicioY
	 + salt.alçadaExtra
	si (potenciaSaltExtra)
		retorna
	altrament si (controlador.esTrobaAlTerra)
		moviment.velocitatVertical = -moviment.gravetat *
		 Temps.tempsDeltaSuavitzat
	altrament
		moviment.velocitatVertical -= moviment.gravetat *
		 Temps.tempsDeltaSuavitzat
	fsi
ffuncio
```

`+calculateJumpVerticalSpeed()`: Determina la velocitat de salt en cada un dels punts.

`+didJump()`: Determina l'estat de salt just al moment de saltar.

`+die()`: Fa que el personatge mori.

`+update()`: Actualitza l'estat del personatge

```
funcio Update () {
	ultimaPosicio: Vector3
	despalaçamentActual: Vector3
	direccioMovimentSalt: Vector3

	// Ens asegurem que el personatge es troba en un pla 2D
	transformacio.posicio.z = 802.0
```

```
        actualitzaDireccioMovimentSuau()

        si (controlador.esAterrat)
              Fisiques.IgnorarColisions (Personatge, Zombies, fals)
        altrament fer
              Fisiques.IgnorarColisions (Personatge, Zombies, cert)
        fsi

        animarPersonatge()
        aplicarGravetat()
        aplicarSalt()
        aplicarCarrega()
        aplicarAtac()
        aplicarAtacCorrent()

        //actualitzem la posició segons el moviment actual
        ultimaPosicio = transformacio.posicio
        despalaçamentActual = moviment.direccio * moviment.velocitat +
         Vector3(0.0, moviment.velocitatVertical, 0.0)
        moviment.alertesColisions = controlador.moviment(desplaçamentActual)
        moviment.velocitat = (transformacio.posicio – ultimaPosicio) /
         temps.tempsFrame

        si (moviment.direccio.magnitudQuadrada > 0.01)
              transformacio.rotacio = girar(transformacio.rotacio,
               moviment.direccio, temps.tempsFrame * movimentRotacioSuavitzada)
        fsi
        si (controlador.aterrat && salt.saltant)
              salt.saltant = fals
              direccioMovimentSalt = moviment.direccio * moviment.velocitat
              si (direccioMovimentSalt.magnitudQuadrada > 0.01)
                    moviment.direccio =  direccioMovimentSalt.normalitzat
              fsi
        fsi
ffuncio
```

8.8.5 PlayerDamageL

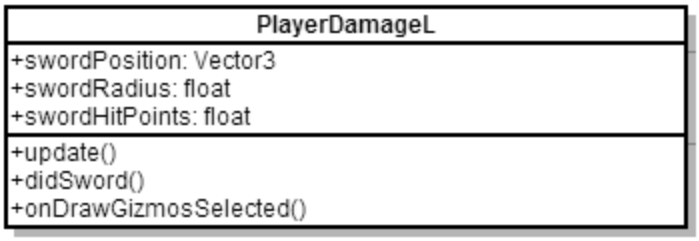


Figura 82: PlayerDamageL.js

Variables

`+swordPosition: Vector3 //` Posició de l'espasa.

`+swordRadius: float //` Radi d'acció de l'espasa.

`+swordHitPoints: float //` Punts de vida que treu al tocar un enemic.

Funcions

`+update():` Actualitza l'estat de l'espasa.

`+didSword():` Mira si quan es fa un atac s'ha tocat algun enemic.

```
funcio didSword () {
	posicio: Vector3
	enemics: GameObject[]
	enemic: Component
	longitudQuadrada: float

	posicio = jo.posicio(Espasa)
	enemics = GameObject.TrobarObjectesAmbEtiqueta("enemic")
	mentre (hi hagi enemics) fer
		enemic = enemics.seguent.getComponent(ZombieTakeDamage)
		si (enemic == nul)
			seguent
		fsi
		longitudQuadrada = (enemic.posicio – posicio).magnitudQuadrada
		si (longitudQuadrada <= radiEspasa*radiEspasa)
			enemic.enviarMissatge("ApplyDamage", puntsTocatEspasa)
		fsi
	fmentre
ffuncio
```

`+onDrawGizmosSelected():` Dibuixa el radi d'acció de l'espasa dins l'editor.

8.8.6 PlayerDamageR

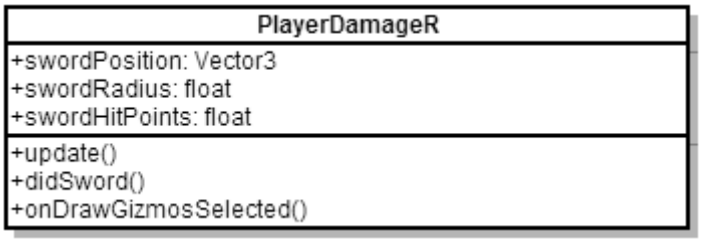


Figura 83: PlayerDamageR.js

Variables

`+swordPosition: Vector3 //` Posició de l'espasa.

`+swordRadius: float //` Radi d'acció de l'espasa.

`+swordHitPoints: float //` Punts de vida que treu al tocar un enemic.

Funcions

`+update():` Actualitza l'estat de l'espasa.

`+didSword():` Mira si quan es fa un atac s'ha tocat algun enemic.

`+onDrawGizmosSelected()`: Dibuixa el radi d'acció de l'espasa dins l'editor.

Es fan dos scripts diferents per cada braç perquè el moment en el que han d'estar actius, per detectar si toquen al personatge, depèn directament de l'animació que hi hagi activa. Aleshores quan una animació utilitza el braç dret per atacar, s'activarà PlayerDamageR; si una animació utilitza el braç esquerre, s'activarà PlayerDamageL; i si fa servir els dos s'activaran els dos scripts. D'aquesta manera evitem col·lisions falses dels braços/espases que es troben dins el rang d'atac però que realment no ataquen.

8.8.7 ParticleSwings

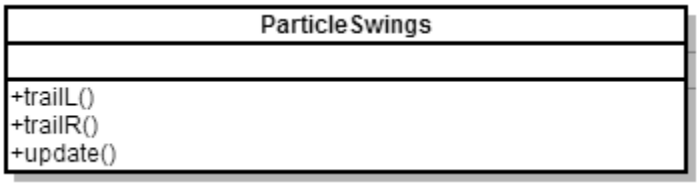


Figura 84: ParticleSwings.js

Funcions

`+trailL():` Dibuixa l'estela de l'espasa esquerra al atacar.

`+trailR():` Dibuixa l'estela de l'espasa dreta al atacar.

`+update():` Actualitza l'estat de les esteles.

8.8.8 LavaTouch

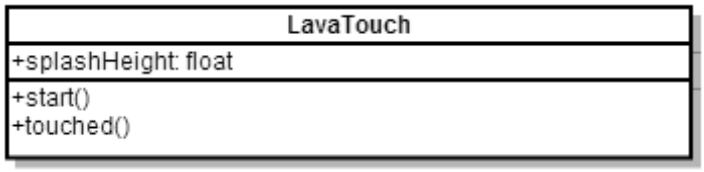


Figura 85: LavaTouch.js

Variables

`+splashHeight: float //` Altura de l'esquitx de lava.

Funcions

`+start():` Inicialitza els esquitxos de lava.

`+touched():` Mira si el personatge o un enemic ha tocat la lava, es crea un esquitx on ha caigut i se li envia un missatge per dir que ha de morir.

8.8.9 LavaMovement

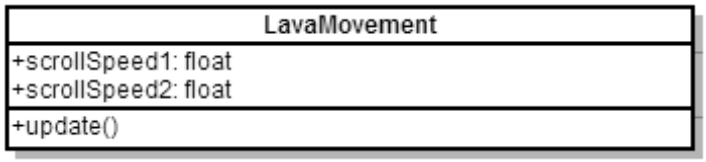


Figura 86: LavaMovement.js

Variables

`+scrollSpeed1: float //` Velocitat de moviment de la primera textura.

`+scrollSpeed2: float //` Velocitat de moviment de la segona textura.

Funcions

`+update():` Actualitza la posició de les textures per crear el moviment.

8.8.10 LavaDamage

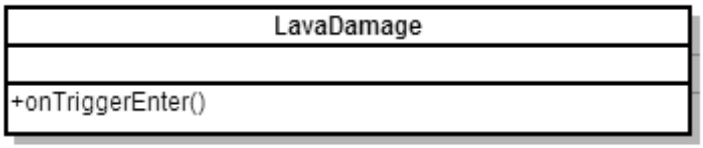


Figura 87: LavaDamage.js

Funcions

`+onTriggerEnter():` Envia un missatge a la lava dient que l'hem tocat.

8.8.11 ZombieMovement

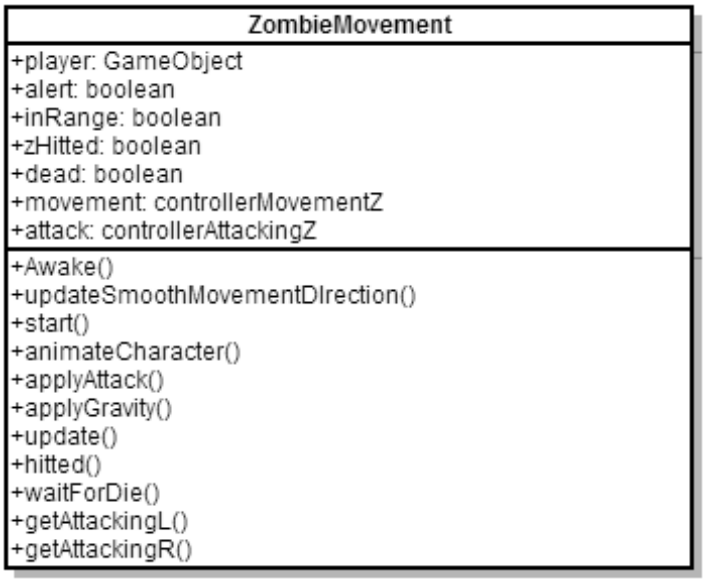


Figura 88: ZombieMovement.js

Variables

`+player: GameObject //` Objectiu al que atacar.

`+alert: boolean //` Hem vist a qui hem d'atacar?

`+inRange: boolean //` Estem suficientment aprop per atacar?

`+zHitted: boolean //` Ens han atacat?

`+dead: boolean //` Estem morts?

`+movement: controllerMovementZ //` Conté les variables relacionades amb el moviment.

`+attack: controllerAttackingZ //` Conté les variables relacionades amb l'atac.

Funcions

`+Awake():` Es posa en moviment.

`+updateSmoothMovementDIrection():` Actualitza la posició i direcció.

`+start():` Inicialitza les animacions.

`+animateCharacter():` Anima segons la lògica descrita.

`+applyAttack():` Aplica la lògica d'atac si es troba dins el rang.

`+applyGravity():` Aplica la gravetat.

`+update():` Actualitza l'estat.

`+hitted():` Actualitza la velocitat i indica que ha rebut un atac.

`+waitForDie():` Controla que toqui al terra abans de morir-se.

`+getAttackingL():` Mira si ataca amb l'esquerra.

`+getAttackingR():` Mira si ataca amb la dreta.

8.8.12 ZombieTakeDamage

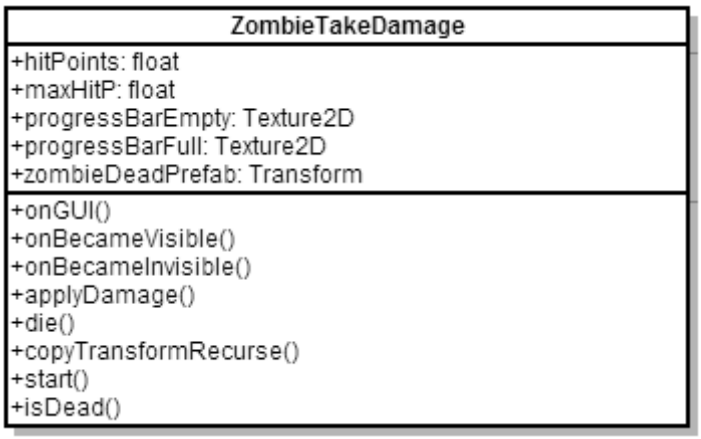


Figura 89: ZombieTakeDamage.js

Variables

`+hitPoints: float //` Nombre de punt de vida.

`+maxHitP: float //` Nombre màxim de punts de vida.

`+progressBarEmpty: Texture2D //` Textura per la part buida de la barra de vida.

`+progressBarFull: Texture2D //` Textura per la part plena de la barra de vida.

`+zombieDeadPrefab: Transform //` Posició on ha mort.

Funcions

`+onGUI():` Dibuixa la barra de vida

`+onBecameVisible():` Indica que s'ha de dibuixar la barra de vida.

`+onBecameInvisible():` Indica que no s'ha de dibuixar la barra de vida.

`+applyDamage():` Actualitza els punts de vida quan ha rebut un atac.

`+die():` Indica que ha de morir.

`+copyTransformRecurse():` Al morir és substituït per una rèplica.

`+start():` Inicialitza la barra de vida a l'inici de l'aplicació.

`+isDead():` Retorna si esta mort o no.

8.8.13 ZombieDamageL

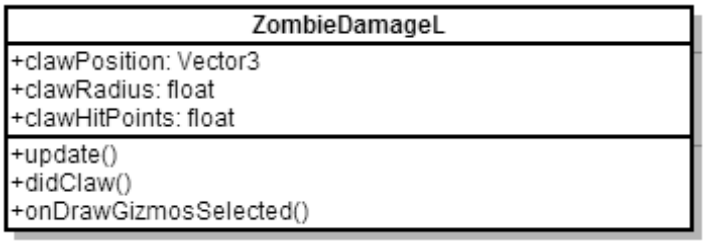


Figura 90: ZombieDamageL.js

Variables

`+clawPosition: Vector3 //` Posició de les urpes.

`+clawRadius: float //` Radi d'acció de les urpes.

`+clawHitPoints: float //` Punts de vida que treu al tocar un enemic.

Funcions

`+update():` Actualitza l'estat de les urpes.

`+didClaw():` Mira si quan es fa un atac s'ha tocat algun enemic.

`+onDrawGizmosSelected():` Dibuixa el radi d'acció de les urpes dins l'editor.

8.8.14 ZombieDamageR

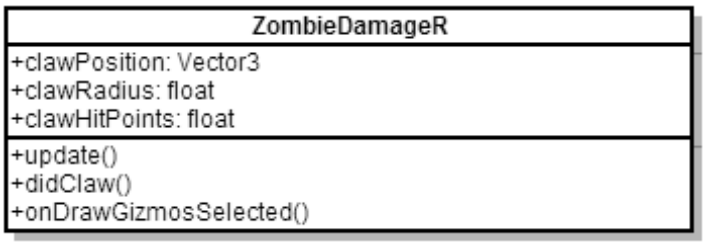


Figura 91: ZombieDamageR.js

Variables

`+clawPosition: Vector3 //` Posició de les urpes.

`+clawRadius: float //` Radi d'acció de les urpes.

`+clawHitPoints: float //` Punts de vida que treu al tocar un enemic.

Funcions

`+update():` Actualitza l'estat de les urpes.

`+didClaw():` Mira si quan es fa un atac s'ha tocat algun enemic.

`+onDrawGizmosSelected():` Dibuixa el radi d'acció de les urpes dins l'editor.

9. Implementació i proves

Aquí exposarem com s'ha implementat cada element per arribar al resultat esperat.

9.1 Android/LibGDX

9.1.1 Menú

Quan es vol engegar l'aplicació es crida automàticament una instància o interfície de JOGL (Java OpenGL) on es determina la classe que implementa l'ApplicationListener (explicat a la secció 7.3.1), el nom i les dimensions de la finestra de l'aplicació.

```
new JoglApplication(new MonstrumOccidere(), "Monstrum Occidere", 480, 320, false);
```

En aquest cas es crida a la classe "MonstrumOccidere" que extén la classe abstracta "Game", la qual implementa la interfície "ApplicationListener". Aleshores, a la creació de l'objecte "MonstrumOccidere", es carreguen les preferències o configuracions, i tots els elements gràfics i de so que requerirà el joc.

```
@Override
    public void create() {
         Settings.load();
         Assets.load();
         super.create();
    }
```

Es crida al create de la classe "Game".

```
@Override
    public void create () {
         screen = getStartScreen();
    }
```

I es carrega la pantalla del menú principal.

```
@Override
    public Screen getStartScreen() {
        return new MainMenuScreen(this);
    }
```

La classe "MainMenuScreen" extén de la classe “Screen”. En la creació de la pantalla es passa un objecte "Game" de manera que es tenen referenciades l'una a l'altre. Aquí instanciarem la càmera que utilitzarem (guiCam), el grup d'sprites (batcher), les

dimensions dels elements (soundBounds i playBounds) i el punt per detectar on es tocarà la pantalla (touchPoint).

```
    public MainMenuScreen (Game game) {
         super(game);
         guiCam = new OrthographicCamera(480, 320);
         guiCam.position.set(480 / 2, 320 / 2, 0);
         batcher = new SpriteBatch();
         soundBounds = new Rectangle(0, 0, 64, 64);
         playBounds = new Rectangle(240 - 150, 200 + 18, 300, 36);
         touchPoint = new Vector3();
    }
```

Quan es crea l'objecte JoglApplication just al principi, en el procés es crida a la funció render() de l'ApplicationListener.

```
@Override
    public void render () {
         screen.update(Gdx.graphics.getDeltaTime());
         screen.present(Gdx.graphics.getDeltaTime());
    }
```

D'aquesta manera ja es dibuixen els elements que volem mostrar al menú, i establim un observador per veure si hi ha interacció amb ells, per tal de reaccionar i cridar als events pertinents.

```
@Override
    public void update (float deltaTime) {
         if (Gdx.input.justTouched()) {
              guiCam.unproject(touchPoint.set(Gdx.input.getX(), Gdx.input.getY(), 0));
              if (OverlapTester.pointInRectangle(playBounds, touchPoint.x,
touchPoint.y)) {
                   game.setScreen(new GameScreen(game));
                   return;
              }
              if (OverlapTester.pointInRectangle(soundBounds, touchPoint.x,
touchPoint.y)) {
                   Settings.soundEnabled = !Settings.soundEnabled;
                   if (Settings.soundEnabled) Assets.music.play();
                   else Assets.music.pause();
              }
         }
    }
```

```
@Override
    public void present (float deltaTime) {
        GLCommon gl = Gdx.gl;
        gl.glClearColor(1, 0, 0, 1);
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        guiCam.update();
        batcher.setProjectionMatrix(guiCam.combined);

        batcher.disableBlending();
        batcher.begin();
        batcher.draw(Assets.backgroundRegion, 0, 0, 480, 320);
        batcher.end();

        batcher.enableBlending();
        batcher.begin();
        batcher.draw(Assets.mainMenu, 240 - 150, (int)(200 - 110 / 2), 300, 110);
        batcher.draw(Settings.soundEnabled ? Assets.soundOn : Assets.soundOff, 0, 0,
64, 64);
        batcher.end();
    }
```

Al final obtenim la pantalla (Figura 92):

Figura 92: Pantalla del menú principal

Quan es toqui la pantalla sobre l'àrea de text "play" es canviarà la pantalla per una nova instància de la classe "GameScreen".

9.1.2 Pantalla de Joc

La pantalla de joc s'inicialitza a partir de la classe "GameScreen" i té una estructura molt semblant a la del menú principal, però més extensa:

```
public GameScreen (Game game) {
      super(game);
      state = GAME_READY;
      guiCam = new OrthographicCamera(480, 320);
      guiCam.position.set(480 / 2, 320 / 2, 0);
      touchPoint = new Vector3();
      batcher = new SpriteBatch();
      world = new World();
      renderer = new WorldRenderer(batcher, world);
      pauseBounds = new Rectangle(480 - 64, 320 - 64, 64, 64);
      resumeBounds = new Rectangle(240 - 96, 160, 192, 36);
      quitBounds = new Rectangle(240 - 96, 160 - 36, 192, 36);
      jump = attack = charge = true;
}
```

En primer lloc tenim els estats del joc, que ens permetran diferenciar i seleccionar els recursos que necessitarem durant la partida i estructurar millor com ha de ser la interacció de l'usuari.

Així doncs, la funció update estarà dividida per cada un dels estats que hem creat.

```
@Override
public void update (float deltaTime) {
      if (deltaTime > 0.1f) deltaTime = 0.1f;

      switch (state) {
      case GAME_READY:
            updateReady();
            break;
      case GAME_RUNNING:
            updateRunning(deltaTime);
            break;
      case GAME_PAUSED:
```

```
            updatePaused();
            break;
        case GAME_LEVEL_END:
            updateLevelEnd();
            break;
        case GAME_OVER:
            updateGameOver();
            break;
        }
    }
```

En el cas de l'estat GAME_RUNNING és essencial saber el temps dins de l'execució ja que és el que ens permetrà tenir el control sobre el que passi al joc; el temps sempre estarà donat per la variable “deltaTime”.

El mateix passarà amb la funció present().

```
@Override
    public void present (float deltaTime) {
        GLCommon gl = Gdx.gl;
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
        gl.glEnable(GL10.GL_TEXTURE_2D);

        renderer.render();

        guiCam.update();
        batcher.setProjectionMatrix(guiCam.combined);
        batcher.enableBlending();
        batcher.begin();
        switch (state) {
        case GAME_READY:
            presentReady();
            break;
        case GAME_RUNNING:
            presentRunning();
            break;
        case GAME_PAUSED:
            presentPaused();
            break;
        case GAME_LEVEL_END:
            presentLevelEnd();
            break;
```

```
        case GAME_OVER:
            presentGameOver();
            break;
        }
        batcher.end();
    }
```

Dins de la funció render(), cridada per un objecte de classe “WorldRenderer”, és on carregarem i escriurem el comportament de tots els elements que hagin de sortir per pantalla i formin part del joc.

```
    public WorldRenderer (SpriteBatch batch, World world) {
        this.world = world;
        this.cam = new OrthographicCamera(FRUSTUM_WIDTH, FRUSTUM_HEIGHT);
        this.cam.position.set(FRUSTUM_WIDTH / 2, FRUSTUM_HEIGHT / 2, 0);
        this.batch = batch;
        this.cache = world.cache;
    }
```

Per crear un objecte d'aquesta classe es necessiten dos elements creats dins de “GameScreen”; el ja mencionat grup d'sprites i el propi món del joc, o el nivell de joc pròpiament dit, la classe “World”.

La classe “World” és la que s'encarrega de generar tots els elements de l'escena:

```
    public World() {
        this.platforms = new Vector<Platform>();
        this.walls = new Vector<Platform>();
        this.graph = new MultiGraph();
        this.rand = new Random();
        this.background = new Background(0,0);

        this.bob = new Bob(11, 8);
        generateLevelProves();
        generateGraph();
        generateEnemies();

        this.score = 0;
        this.state = WORLD_STATE_RUNNING;
    }
```

La classe “Bob” (8.6.4) és el nom de la classe del personatge principal; generateLevelProves() crea les plataformes per les quals “bob” podrà moure's i

col·lisionar; generateGraph() crea el graf que els enemics utilitzaran per moure's pel mapa fins a trobar el personatge; generateEnemies() crea els enemics que hi haurà en escena.

La classe “World” també s'encarrega d'actualitzar els elements i comprovar les col·lisions:

```
public void update(float deltaTime, float accelX) {

     updateBob(deltaTime, accelX);
     updateEnemies(deltaTime);
     checkCollisions(deltaTime);
     checkGameOver();
}

private void checkCollisions(float deltaTime) {
     checkPlatformCollisions(deltaTime);
     checkEnemiesPlatformCollisions(deltaTime);
     checkWallCollisions(deltaTime);
     checkEnemiesWallCollisions(deltaTime);
     checkAI();
}
```

Aleshores ja tenim tots els elements preparats per començar a jugar. Tal com s'especifica a la creació de la “GameScreen”, la pantalla de joc comença amb un estat ”ready”; quan toquem la pantalla començarà l'acció (Figura 93).

Figura 93: Pantalla de Joc "Ready"

9.1.3 Moviment del personatge

En el moment que toquem la pantalla, podrem començar a controlar el nostre personatge.

```
private void updateReady () {
     if (Gdx.input.justTouched()) {
          state = GAME_RUNNING;
     }
}
```

Passarem al ja esmentat “updateRunning” (part de la funció “update”, 9.1.2) on tenim la configuració dels controls per al personatge i on actualitzarem el món/escena. S'ha de remarcar que, per estalviar temps en la implementació, s'utilitza la versió “Desktop” (per ordinador) i per tant el controls corresponen als d'un teclat.

```
private void updateRunning (float deltaTime) {
     if (Gdx.input.justTouched()) {
          guiCam.unproject(touchPoint.set(Gdx.input.getX(), Gdx.input.getY(), 0));

          if (OverlapTester.pointInRectangle(pauseBounds, touchPoint.x,
              touchPoint.y)) {
               state = GAME_PAUSED;
               return;
          }
     }

     float accel = 0;
     if(Gdx.input.isKeyPressed(Keys.DPAD_LEFT)){
          if (world.bob.lastPlatformTouch == -1 || world.bob.sideAnt == 1)
               accel = 5f;
     }
     else if(Gdx.input.isKeyPressed(Keys.DPAD_RIGHT)) {
          if (world.bob.lastPlatformTouch == -1 || world.bob.sideAnt == -1)
               accel = -5f;
     }
     if(Gdx.input.isKeyPressed(Keys.DPAD_UP)) {
          if (jump) {
               world.bob.jump();
               jump = false;
          }
     }
```

```
        else jump = true;

        if (Gdx.input.isKeyPressed(Keys.DPAD_DOWN)) {
             world.bob.finalAttack();
        }
        if(Gdx.input.isKeyPressed(Keys.A)) {
             if (attack) {
                  world.bob.attack();
                  attack = false;
             }
        }
        else attack = true;

        if(Gdx.input.isKeyPressed(Keys.S)) {
             if (charge) {
                  world.bob.charge();
                  charge = false;
             }
        }
        else charge = true;

        if(Gdx.input.isKeyPressed(Keys.C)) world.provaCamins();

        world.update(deltaTime, accel);
        if (world.state == World.WORLD_STATE_NEXT_LEVEL) {
             state = GAME_LEVEL_END;
        }
        if (world.state == World.WORLD_STATE_GAME_OVER) {
             state = GAME_OVER;
        }
    }
```

Per veure com els controls afecten al personatge és important tenir estats. Depenent de l'estat en què es trobi, els controls l'afectaran d'una manera o d'una altre. També ens ajudarà a saber quina animació s'ha de mostrar per pantalla, ja que en farem referència dins del “WorldRenderer” quan toqui dibuixar el personatge per pantalla.

A part de les funcions cridades pels controls, la classe “Bob” en conté d'altres que s'executen quan es crida la funció update() de la classe “World”, que acaben de determinar la posició i l'estat en el que es trobarà el personatge principal.

9.1.4 Detecció de col·lisions del personatge

Per tal d'acabar de determinar en quina posició i estat es troba el nostre personatge principal, és primordial saber si es troba a l'aire o si per altre banda està reposant sobre una plataforma; per això tenim la funció "checkPlatformCollisions".

```
private void checkPlatformCollisions(float deltaTime) {
     if (bob.velocity.y > 0) return;
     boolean hitted = false;
     if (bob.velocity.y < 0 || bob.lastPlatformHit == 0) {
          Iterator<Platform> it = platforms.iterator();
          while (it.hasNext()) {
               Platform platform = it.next();
               if (bob.position.y >= platform.position.y) {
                    if (OverlapTester.overlapRectangles(bob.bounds,
                        platform.bounds) || OverlapTester.overlapGameObject(bob,
                        platform)) {
                         bob.velocity.y = 0;
                         bob.lastPlatformHit = platform.id;
                         hitted = true;
                         bob.hitPlatform();
                         break;
                    }
               }
          }
          if (!hitted) bob.noHitPlatform();
     }
     else {
          Platform platform = platforms.get(bob.lastPlatformHit-1);
          if (OverlapTester.overlapRectangles(bob.bounds, platform.bounds) ||
              OverlapTester.overlapGameObject(bob, platform)) {
               bob.velocity.y = 0;
               bob.hitPlatform();
          }
          else bob.noHitPlatform();
     }
}
```

Es mira l'eix de les y, sobre quina plataforma s'ha col·lisionat. En cas que no estiguem ascendint, comparant els rectangles amb la classe “OverlapTester” de dues maneres diferents: sobreposant rectangles i intersecant segments.

```
public static boolean overlapRectangles(Rectangle r1, Rectangle r2) {
     if (r1.x < r2.x + r2.width &&
          r1.x + r1.width > r2.x &&
          r1.y < r2.y + r2.height &&
          r1.y > r2.y)
          return true;
     else
          return false;
}
```

La funció mira si hi ha intersecció dels rectangles que formen els objectes del personatge principal i la plataforma.

```
public static boolean overlapGameObject(Bob r1, GameObject r2) {
     if (Geometry.isLineIntersectingLine(r1.boundsAnt.x + r1.bounds.width/2,
         r1.boundsAnt.y, r1.bounds.x + r1.bounds.width/2, r1.bounds.y, r2.bounds.x,
         r2.bounds.y + r2.bounds.height, r2.bounds.x + r2.bounds.width, r2.bounds.y
         + r2.bounds.height))
          return true;
     else return false;
}
```

En aquesta altra funció es mira si intersequen dos segments. El primer està format per la posició anterior i actual del personatge, i l'altre per la part superior de la plataforma.

Es segueix un procediment semblant per les interseccions per l'eix de les x, però tinguen en compte el sentit en el que el nostre personatge es mou (checkWallCollisions).

Un cop s'ha determinat si s'ha efectuat la col·lisió o no, es criden les respectives funcions (hitPlatform, noHitPlatform, touchPlatform, noTouchPlatform) per acabar de determinar l'estat en el que s'ha de trobar el personatge.

També s'han de detectar les col·lisions amb els enemics quan els estem atacant.

```
private void checkAttackCollisions() {
     Enemy act; Iterator<Bob> it = enemies.iterator();
     while (it.hasNext()) {
          act = (Enemy)it.next();
          if (act.state != BOB_STATE_DEAD && act.state != BOB_STATE_DEADEND) {
               if (this.position.y + 1.5 > act.position.y && this.position.y - 1.5
                    < act.position.y ) {
                    if (sideAnt < 0) {
```

```
                    if (this.position.x - 2.5 <= act.position.x &&
                        this.position.x > act.position.x) act.hitted(sideAnt);
                }
                else {
                    if (this.position.x + 2.5 >= act.position.x &&
                        this.position.x < act.position.x) act.hitted(sideAnt);
                }
            }
        }
    }
}
```

En aquest cas mirarem simplement si tenim algun enemic dins d'un rang d'atac davant nostre i li aplicarem una funció per dir-li que l'hem tocat (Figura 94).

```
public void hitted(float side) {
    if (health > 0) {
        health -= 10;
        state = BOB_STATE_HITTED;
    }
    else if (health == 0) state = BOB_STATE_DYING;
    velocity.y = BOB_HITTED_VELOCITY;
    velocity.x = BOB_HITTED_VELOCITY * side;
    this.stateTime = 0;
}
```

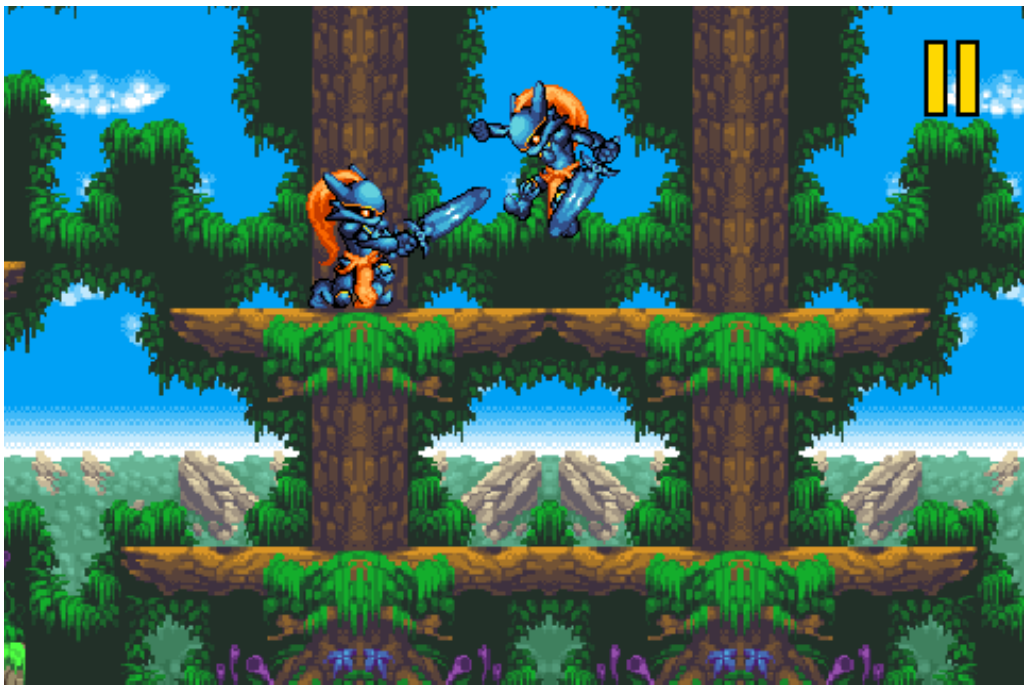


Figura 94: Enemic golpejat per un atac

Com es comenta a l'apartat 8.6.6, per la realització d'aquest prototip hem utilitzat el mateix model que el nostre personatge principal pels enemics, i tenen un comportament gairebé idèntic, fent que la classe “Enemy” estengui directament de “Bob”. En versions futures caldria canviar-ne els colors o incorporar dissenys nous. El que es necessita per controlar-los és la seva IA.

9.1.5 Intel·ligència artificial dels enemics

Tal i com ja hem especificat a l'apartat 8.5, s'ha utilitzat l'algoritme **A*** per moure els enemics des de qualsevol punt de l'escenari fins a la nostra posició (la implementació utilitzada d'aquest algoritme i les classes utilitzades per la creació del graf, són les creades per "graphstream project" `import org.miv.graphstream.algorithm.AStar;`).

Per fer-ho inicialment vàrem realitzar un graf dirigit (Figura 39). Aquest graf ha resultat ser massa complex i poc eficient a l'hora que els enemics puguin construir un camí i per això se n'ha fet una revisió reduint-lo i fent-ne un graf no-dirigit.

Figura 95: Grapf AStar simplificat per plataformes

Recordem que el graf es crea quan es crea el "World" i dins de la funció "checkCollisions" hi ha la funció "checkAI" que serà l'encarregada de dir als enemics el camí que han de seguir per arribar al nostre personatge.

```
public void checkAI() {
     String strEne = "", strBob = "";
     Enemy act;
     Iterator<Bob> it = bob.enemies.iterator();
     while (it.hasNext()) {
          act = (Enemy)it.next();
          if (act.AI) {
               AStar astar = new AStar(graph);
               strEne = closestNode(act); strBob = closestNode(bob);
               try {
                    astar.compute(strEne, strBob);
                    Path path = astar.getShortestPath();
                    Node n1, n0;
                    if (path.size() > 1) n1 = path.getNodePath().get(1);
                    else n1 = null;
                    if (path.size() > 0) n0 = path.getNodePath().get(0);
                    else n0 = null;
                    act.goToNode(n1,n0);
               }
               catch (Exception exec){}
          }
          else act.goToNode();
     }
}
```

El que fem és una iteració per cada enemic i mirem per cada un d'ells quin és el node més proper i quin és el més proper al personatge principal.

```
private String closestNode(Bob act) {
     Node node;
     String nearestNode = "";
     double shortestDistance = 0, distance;
     Iterator<Node> it = (Iterator<Node>) graph.getNodeIterator();
     while (it.hasNext()) {
          node = it.next();
          if ((Double)node.getAttribute("y") - act.position.y <= 2.7) {
               if (shortestDistance == 0) {
                    nearestNode = node.getId();
                    shortestDistance =
                    Geometry.length((Double)node.getAttribute("x"),
                      (Double)node.getAttribute("y"), act.position.x,
```

```
                    act.position.y);
               }
               else {
                    distance = Geometry.Length((Double)node.getAttribute("x"),
                    (Double)node.getAttribute("y"),act.position.x,act.position.y);
                    if (distance < shortestDistance) {
                         shortestDistance = distance;
                         nearestNode = node.getId();
                    }
               }
          }
     }
     return nearestNode;
}
```

Després mirem el camí més curt i li diem a l'enemic d'anar fins al primer node passant-li els dos primers; en cas no tingui segon node significa que el següent pas serà anar fins a la posició del persontge principal (goToNode(n1,n0)).

Un cop l'enemic sap on ha d'anar, no tornarà a renovar el camí fins que no hagi arribat al punt de destí, al que s'ha començat a moure. D'aquesta manera s'aconsegueix reduïr el temps d'execució i es garantitza que no hi haurà comportaments estranys, com per exemple quan nosaltres ens comencem a moure per l'escenari i com a resultat, el camí canvia enmig d'un desplaçament, pel que es produeixen canvis de sentit i/o direcció innecessaris. Per aquest motiu s'ha implementat una altre versió de l'algoritme "goToNode()", sense paràmetres, perquè l'enemic segueixi el camí que ja te indicat.

```
public void goToNode() {
     if (goX == 0 && goRX == 0) {
          goX = enemies.get(0).position.x; goY = enemies.get(0).position.y;
          goRX = 0.5; goRY = 0.2;
     }
     if (state != BOB_STATE_HITTED && state != BOB_STATE_DEAD && state !=
         BOB_STATE_DYING && state != BOB_STATE_DEADEND) {
          if (position.y > goY) {
               if ((goX - position.x > 5 || position.x -goX > 5) && (goX -
                    position.x < 7 || position.x -goX < 7)) jump();
          }
          else {
               if (goY - position.y > goRY && (sideAnt == 1 && goX - position.x <
                    7 || sideAnt == -1 && position.x - goX < 7) && goY-position.y <
```

```
                2.6) jump();
            }

            if (goX - position.x > goRX && (lastPlatformTouch == -1 || sideAnt ==
                -1)) velocity.x = 5f / 10 * Bob.BOB_MOVE_VELOCITY;
            else if (position.x - goX > goRX && (lastPlatformTouch == -1 || sideAnt
                    == 1)) velocity.x = -5f / 10 * Bob.BOB_MOVE_VELOCITY;
            else {
                if (state != BOB_STATE_RUN || health <= 0) velocity.x = 0;
                if (state != BOB_STATE_JUMP && state != BOB_STATE_JUMPING && state !
                    = BOB_STATE_FALL && state != BOB_STATE_FALLING) AI = true;
            }
        }
    }
```

Dins aquest codi tenim els paràmetres que necessiten els enemics per saber com s'han de moure. Bàsicament es codifica el moviment a partir de les limitacions físiques a les que es troben sotmesos; quina altura aconsegueixen quan salten, la distància de salt que poden recórrer en moviment, etc; i deixant uns marges perquè no s'hagin de posicionar exactament sobre el punt al que es dirigeixen sinó sobre un segment. D'aquesta manera evitem possibles canvis d'orientació que es poden produir per culpa de l'acceleració, provocant que no es pugui arribar mai a estar quiet sobre el punt desitjat.

9.1.6 Representació d'animacions i moviment de la càmera

Com hem esmentat a l'apartat 9.1.3, tenim tot un seguit d'estats per als personatges que ens permeten saber quina animació s'ha de mostrar per cadascú. Cal recordar que les animacions es troben dins d'arxius d'imatge en format "png"; veurem com les capturem i les codifiquem per fer-les servir.

El primer que es necessita és una classe que permeti treballar còmodament a la que anomenarem “Animation”. Consta de 2 variables estàtiques per indicar si l'animació serà un bucle o no.

```
public static final int ANIMATION_LOOPING = 0;
public static final int ANIMATION_NONLOOPING = 1;
```

I també conté una taula de frames, representats per la classe “TextureRegion” que proporciona el framework LibGDX, i la duració amb la que s'haurà de mostrar cada un.

```
final TextureRegion[] keyFrames;
final float frameDuration;
```

Una "TextureRegion" és (com el nom ja indica) una part d'una imatge, i els paràmetres que necessita són una textura i les coordenades del rectangle, que formen la regió que volem capturar.

Així doncs, el procediment per obtenir una animació és el d'obtenir la textura/imatge i carregar les parts que en necessitem.

```
public static Texture items = loadTexture("data/items.png");

public static Animation bobJump = new Animation(0.1f,
            new TextureRegion(items, 512, 128, 128, 128),
            new TextureRegion(items, 640, 128, 128, 128),
            new TextureRegion(items, 768, 128, 128, 128),
            new TextureRegion(items, 896, 128, 128, 128));
```

Aquestes operacions es realitzen al principi (com apuntàvem a l'apartat 9.1.1) dins de la funció Assets.load(); i a partir d'aquest moment, quan s'hagi d'utilitzar es farà dins el "WorldRenderer" mirant l'estat del personatge.

```
private void renderBob () {
     TextureRegion keyFrame;
     switch (world.bob.state) {
     case Bob.BOB_STATE_JUMP:
          keyFrame = Assets.bobJump.getKeyFrame(world.bob.stateTime,
           Animation.ANIMATION_NONLOOPING);
          break;
```

Un cop es sap quin frame s'ha de dibuixar per pantalla es mira la direcció per si s'ha de dibuixar en sentir dret o esquerre, i també es mira el factor de correcció que s'ha d'aplicar a l'hora de dibuixar-lo, ja que es pot donar el cas que tinguem diferents mides de frame i per tant no concordaria amb l'anterior (Figura 96).

```
     float wh = 4f;
     float wbpx = 2f;
     float wbpy = 1.2f;
     if (world.bob.state == Bob.BOB_STATE_FINAL_ATTACK) {
          wh = 6.25f;
          wbpx = 3.375f;
     }

     if (side < 0)
```

```
            batch.draw(keyFrame, world.bob.position.x + wbpx, world.bob.position.y -
             wbpy, side * wh, wh);
        else
            batch.draw(keyFrame, world.bob.position.x - wbpx, world.bob.position.y -
             wbpy, side * wh, wh);
```

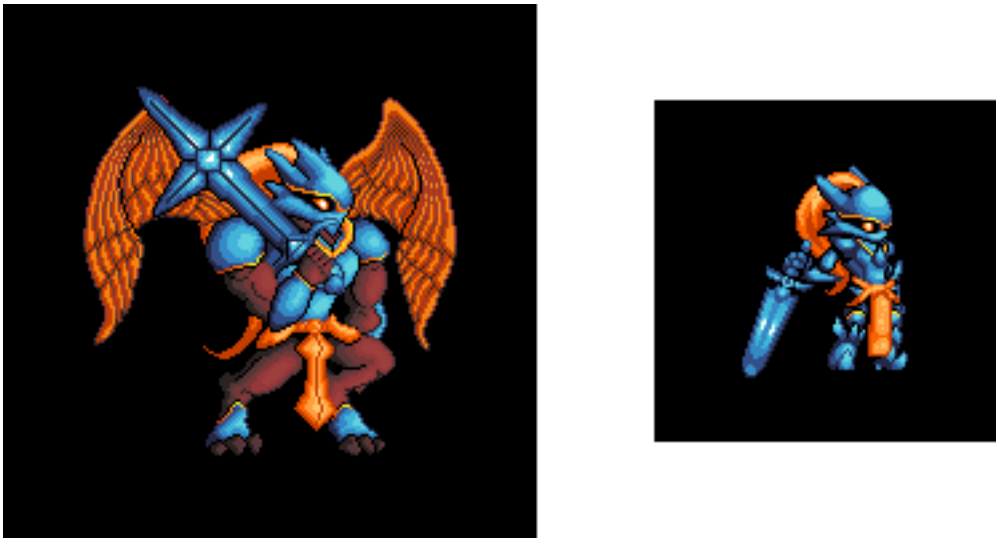


Figura 96: Comparativa mida de frames

Finalment l'únic que ens queda és dir-li a la càmera que segueixi el nostre personatge per l'escenari.

```
//X dibuix background + X set càmera + correcció
if (cam.position.x > world.background.position.x - FRUSTUM_WIDTH / 2.5f +
    FRUSTUM_WIDTH / 2 + 0.5f)
     if (world.bob.position.x < cam.position.x - 3) cam.position.x =
         world.bob.position.x + 3;
//X dibuix background - X set càmera - correcció + W dibuix background
if (cam.position.x < world.background.position.x - FRUSTUM_WIDTH / 2.5f -
    FRUSTUM_WIDTH / 2 - 0.5f + FRUSTUM_WIDTH * 2.8f )
     if (world.bob.position.x > cam.position.x + 3) cam.position.x =
         world.bob.position.x - 3;

if (world.bob.position.y > cam.position.y + 2) cam.position.y =
    world.bob.position.y - 2;

if (cam.position.y > world.background.position.y + FRUSTUM_HEIGHT / 4 + 0.2f)
     if (world.bob.position.y < cam.position.y - 2) cam.position.y =
         world.bob.position.y + 2;
```

9.2 Unity

9.2.1 Menú

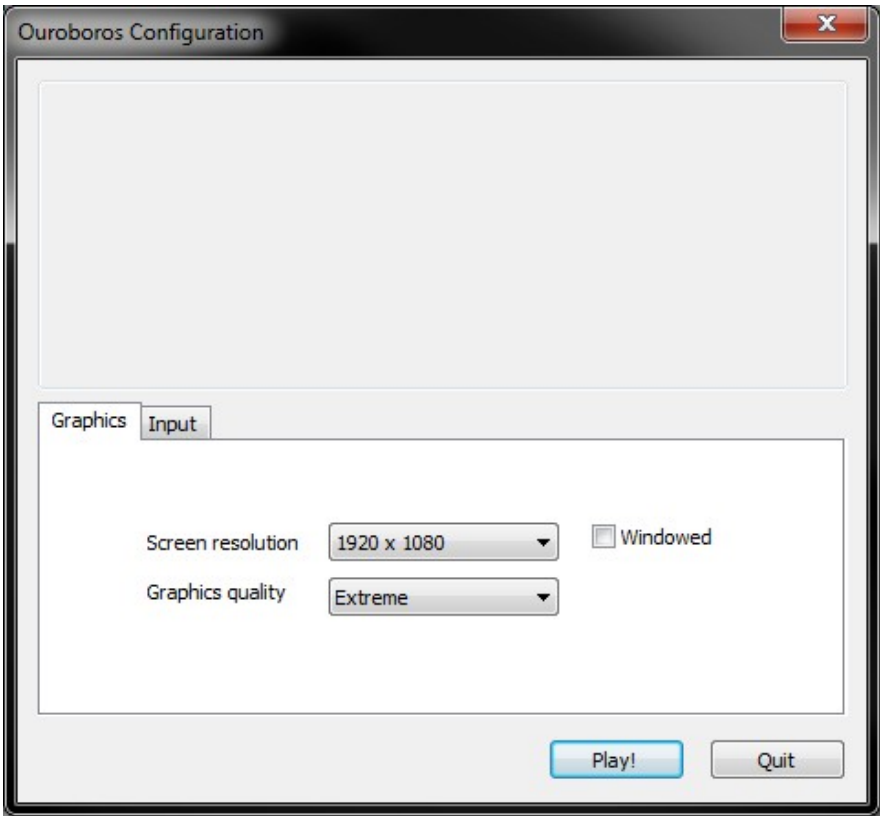


Figura 97: Menú de configuració Unity

Unity ens permet configurar el nostre projecte de manera que quan vulguem crear un executable de la nostra aplicació, mostri un menú on puguem modificar els paràmetres de l'aplicació tals com, per exemple, la resolució de pantalla, la qualitat dels gràfics o els controls de forma molt senzilla (Figura 97).

Quan premem el botó “Play!” anirem a la pantalla de joc. Un cop hi hem entrat, per trobar el menú de joc, del primer que hem de parlar és de la càmera. Unity utilitza les càmeres per representar elements d'interfície (GUI – Graphical User Interface), ja que aquests sempre hauran d'ésser estàtics com si es tractés d'un marc. La càmera doncs, serà un “GameObject” de l'escena sobre el que treballem i conté diferents components, dels quals els més importants són “Transform”, “Camera”, i dos scripts associats “Smooth Follow 2D” i “Player Menu”, en aquest cas.

“Transform” (Figura 98) és un component essencial de qualsevol tipus de “Game Object” i ens diu tota la informació necessària per saber com està posicionat en l'espai.

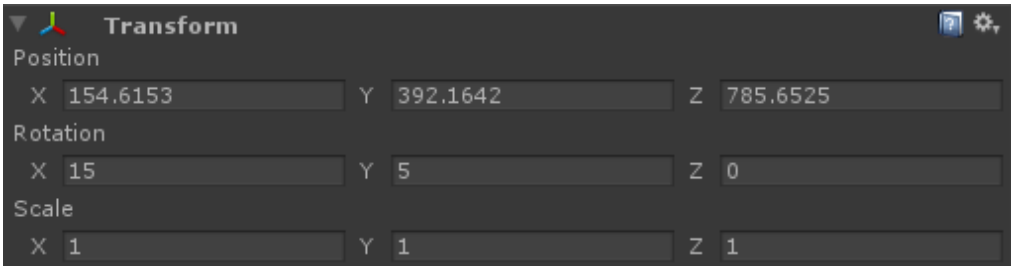


Figura 98: Component Transform d'un Game Object

“Camera” (Figura 99) es un component que mostra tota la informació de què disposen les càmeres i on podem configurar directament els paràmetres.

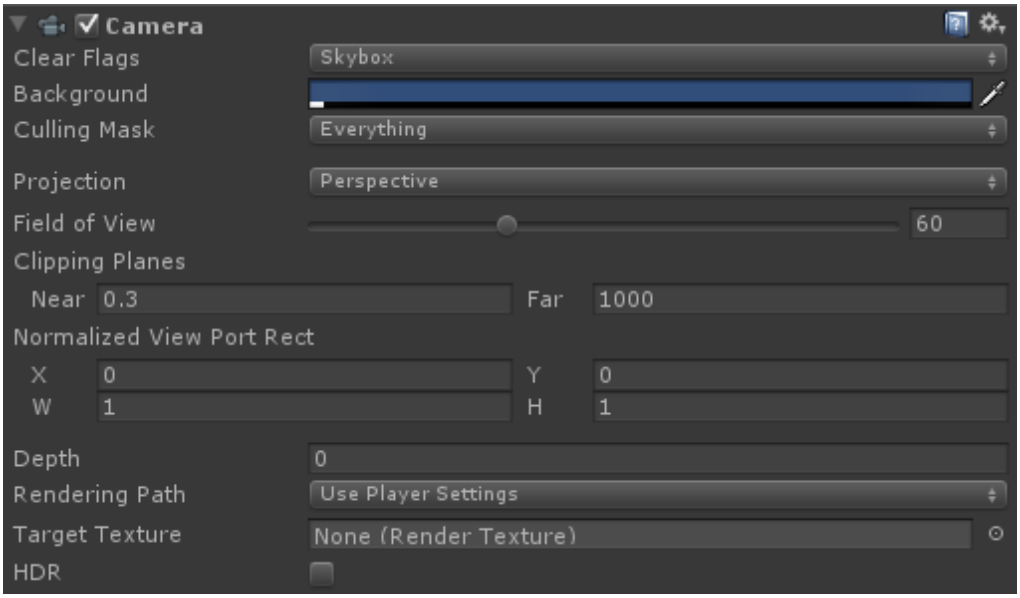


Figura 99: Component Camera

Pel que fa als components dels scripts (Figura 100), contenen les variables públiques que hi ha dins d'ells per poder modificar-los des de l'editor.

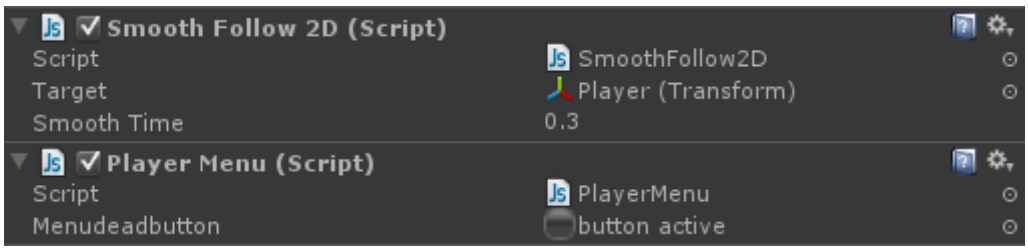


Figura 100: Components generats per Scripts

El primer es centra en moure la càmera suaument quan es mou el personatge principal

(Player) i el segon en mostrar un menú quan es prem una tecla (Esc, Figura 101).

Figura 101: Menú Pantalla de Joc

L'script “PlayerMenu” consta de 3 funcions; “Start()”, “Update()” i “OnGUI()”. La funció “Start()” s'executa sempre en el moment de creació de l'objecte en el que es troba associat, que sol ser al iniciar-se l'escena. Després sempre es cridarà a la funció “Update()”.

```
private var pauseEnabled = false;
var menudeadbutton : Texture2D;

function Start () {
	pauseEnabled = false;
	Screen.showCursor = false;
}
```

La funció “OnGUI()” és específica i fa referència al sistema d'interfícies de Unity. Es crida una vegada per frame.

```
function OnGUI() {
	if (pauseEnabled) {
		//Make a background box
		var size = GUI.skin.GetStyle("Box");
		size.fontSize = 20;
		GUI.contentColor = Color.black;
		GUI.Box (Rect(Screen.width /2 - 110,Screen.height /2 - 100,270,150),
		 "Menu");

		GUI.skin.button.normal.background = menudeadbutton;
		//Make Main Menu button
```

```
            if(GUI.Button(Rect(Screen.width /2 - 100,Screen.height /2 - 60,250,50),
             "Retry")) {
                  Application.LoadLevel("TheGame");
            }
            //Make quit game button
            if (GUI.Button (Rect (Screen.width /2 - 100,Screen.height /2 -10,250,50),
             "Quit Game")) {
                  Application.Quit();
            }
      }
}
```

La funció Update també es crida una vegada per frame.

```
function Update () {
      if(Input.GetKeyDown("escape") || Input.GetButtonDown ("Start")){
            if(pauseEnabled == true){
                  pauseEnabled = false;
                  Screen.showCursor = false;
            }
            else if(pauseEnabled == false){
                  pauseEnabled = true;
                  Screen.showCursor = true;
            }
      }
}
```

9.2.2 Pantalla de Joc

Com hem dit a l'apartat anterior, la càmera és un “GameObject”; ara farem un repàs als objectes més destacats que formen l'escena (Figura 102).

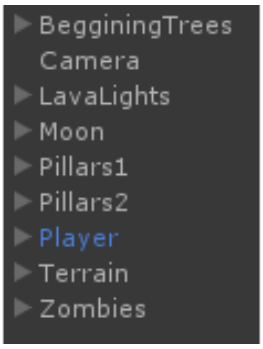


Figura 102: Objectes de l'escena (Unity)

Per començar tenim el terreny sobre el que succeïra tota l'acció; està format a partir d'un objecte “Terrain” al que se li dona relleu amb un component que ja porta incorporat (Figura 103), i posteriorment es pinta amb diferents textures.

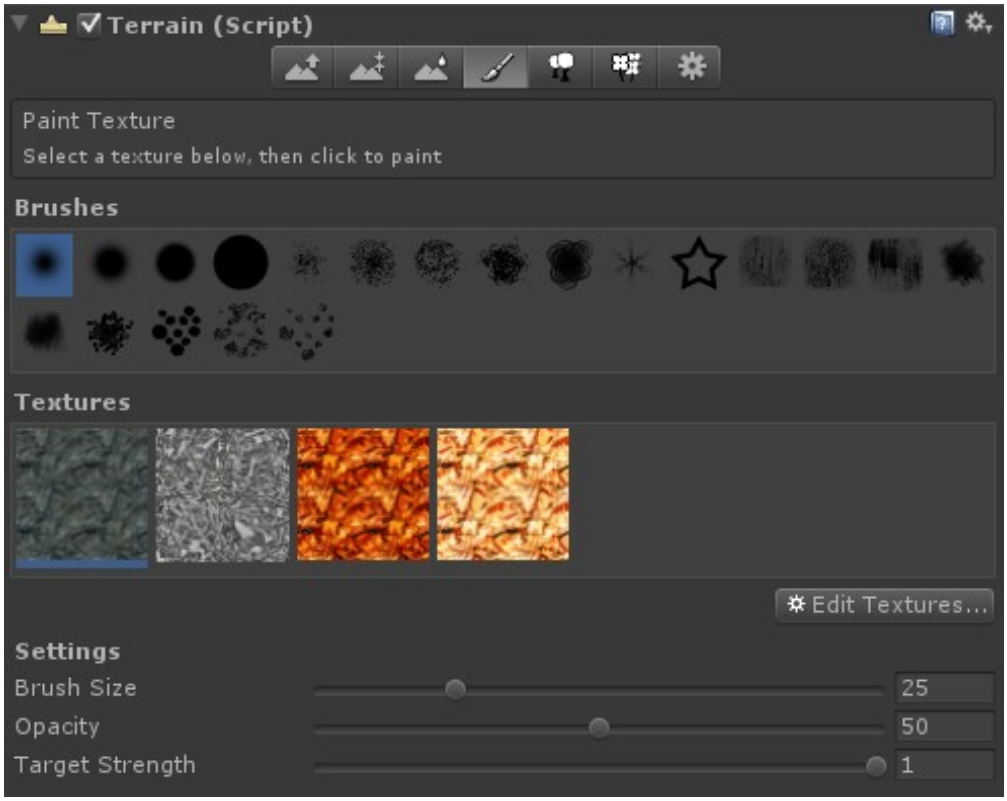


Figura 103: Component editor d'un objecte "Terrain"

En qualsevol moment que es vulgui es pot corregir/modificar el terreny d'acord a les necessitats que vagin sorgint al llarg del camí quan es testeja l'escena. És molt plausible que sorgeixin punts de conflicte, en el recorregut del personatge, que no es puguin superar per aquest, o intersequin amb la vista de la càmera.

També conté un “Terrain Collider” (Figura 104) que és el que impedirà que el nostre personatge, i qualsevol altre objecte al que li afecti la gravetat, el travessi.

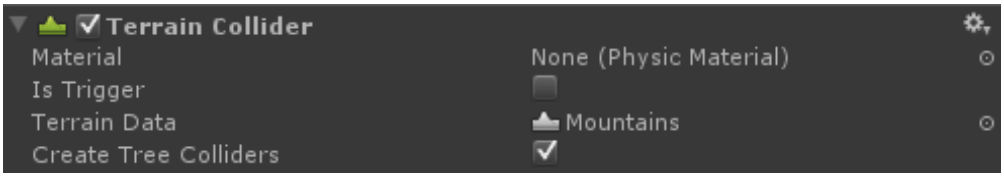


Figura 104: Component "Terrain Collider"

Posteriorment s'ha decorat amb arbres i d'altres elements per complementar el que no es pot fer amb l'editor esmentat anteriorment, com les plataformes enmig de la lava.

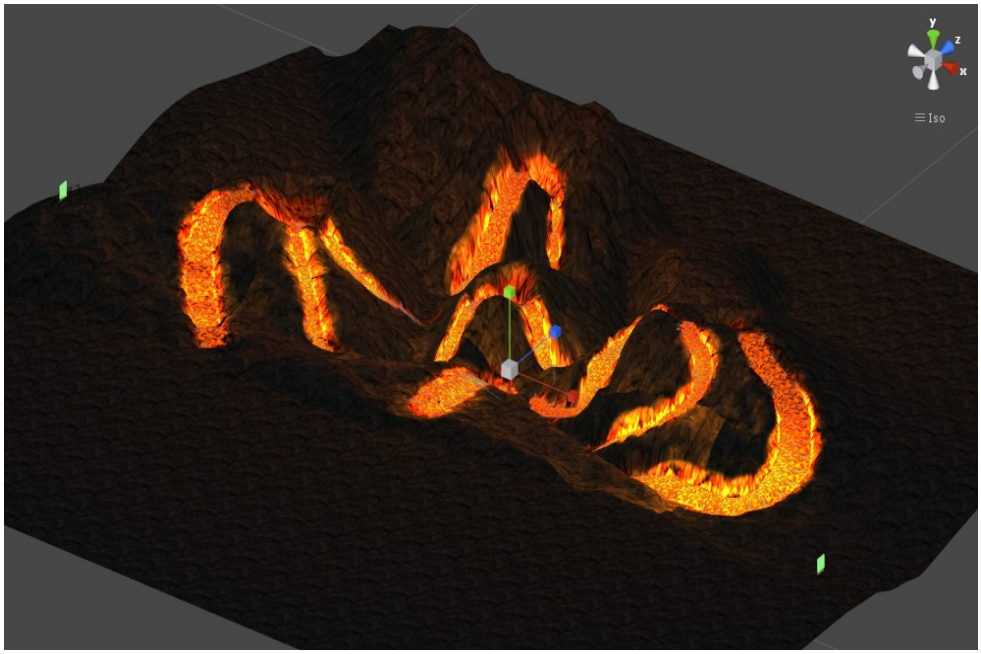


Figura 105: Vista aeria del terreny

També s'ha il·luminat l'escena amb llums de diferents tipologies per aconseguir l'atmosfera desitjada, com la que prové de la lluna o els rius de lava, que consten també d'un efecte (Lava Movement, Figura 106) en el que es fan moure les textures de lava, per tal de simular el seu moviment.

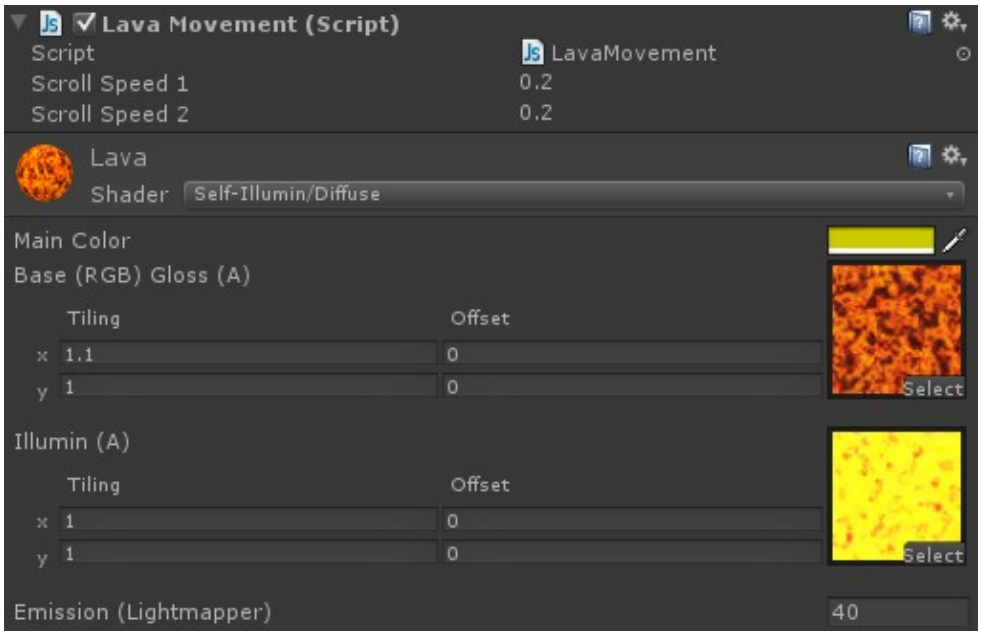


Figura 106: Components per l'aparenaça d'un riu de lava

```
var scrollSpeed1 : float = 0.2;
var scrollSpeed2 : float = 0.2;

function Update () {
    var offset1 : float = Time.time * scrollSpeed1;
    var offset2 : float = Time.time * scrollSpeed2;
    renderer.material.SetTextureOffset ("_MainTex", Vector2(0,offset1));
    renderer.material.SetTextureOffset ("_BumpMap", Vector2(0,offset2));
}
```

Els rius de lava també disposen d'un altre script que envia un missatge a aquells objectes que col·lisionen amb ells; es fa servir bàsicament per saber quan algú ha caigut a la lava.

```
function OnTriggerEnter(other : Collider) {
      var entity = other.gameObject.GetComponent(LavaTouch);
      entity.SendMessage("Touched");
}
```

Per exemple, quan el personatge principal cau dins la lava se li envia el missatge "Touched", que invoca la funció del mateix nom continguda a l'script "LavaTouch" que té associat, i on es detalla el comportament que ha de tenir quan hi cau.

```
function Touched () {
      if (gameObject.name != "Player") {
            lava = GameObject.Find("Zombies/"+gameObject.name+"/LavaSplash");
      }
      var splashPos = Vector3(gameObject.transform.position.x,
       gameObject.transform.position.y + splashHeight,gameObject.transform.position.z);
      var splashRot = Quaternion(0.5, 0.5, 0.5, -0.5);
      Instantiate (lava,splashPos,splashRot);
      if (gameObject.name == "Player") {
            hitted = GetComponent(PlayerTakeDamage);
            hitted.Die();
            Destroy(GameObject.Find("Player/Blob Shadow Projector"));
      }
      else Destroy(gameObject);
}
```

Bàsicament es crida a un objecte del generador de partícules de Unity, que simula una esquitxada de lava, i se li envia un missatge a l'objecte del personatge perquè executi la funció "Die()" indicant que ha de morir.

El personatge principal és un objecte amb un disseny en concret, format per un model en 3D, unes textures i un esquelet. Posteriorment se li han afegit diferents components per fer que aquest model 3D es pugui moure i interactuar amb els elements de l'escena; un "Character Controller", ja esmentat a l'apartat 8.4.2, i un objecte "Animation" (Figura 107).

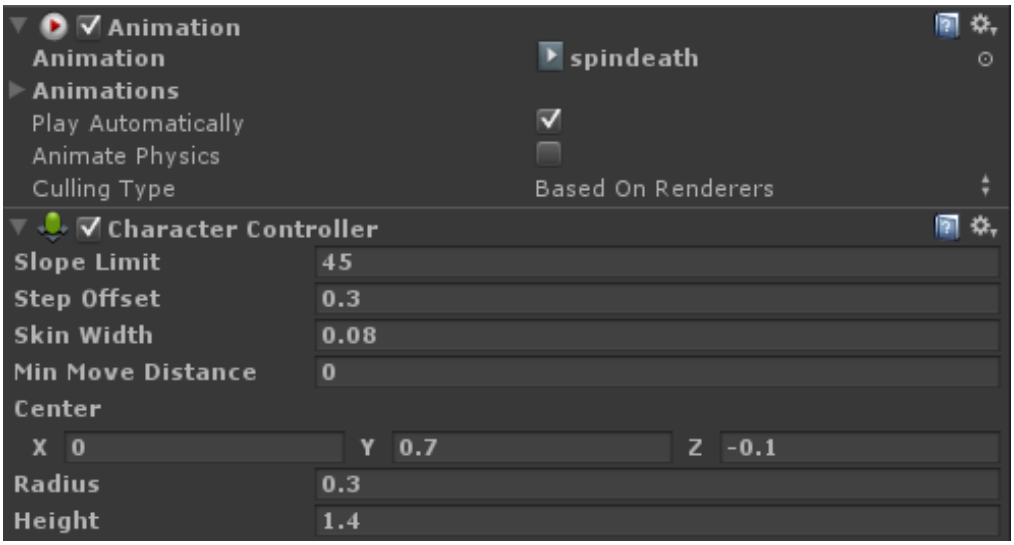


Figura 107: Components Animation i Character Controller

Dins del component "Animation" tenim totes les animacions que el personatge necessita, que prèviament hem importat de 3DsMax al ajustar-les al nostre model modificat pel personatge. Es poden visualitzar dins un petit reproductor al seleccionar-les (Figura 108).

Figura 108: Previsualitzador d'animacions

A part dels components, se li han afegit a l'objecte “Player”, altres objectes com el ja esmentat esquitx de lava, un altre de sang per quan el fereixin, un parell d'espases amb efectes d'il·luminació i partícules, i una aureola que projecta l'ombra.

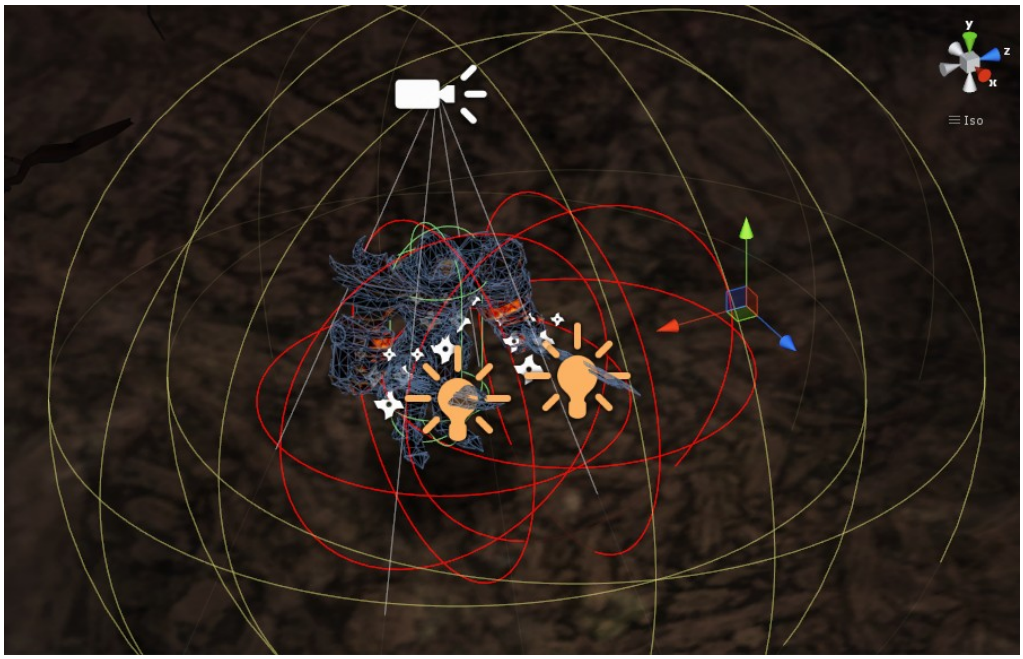


Figura 109: "Player" seleccionat amb tots els seus components

Per acabar amb l'escena, tenim els “Zombies” que seran els enemics als que el “Player” haurà de fer front. També estan dotats d'un “Character Controller” i un “Animation”, i tenen inclosos objectes esmentats al punt anterior com l'esquitx de lava, el de sang i l'ombra.

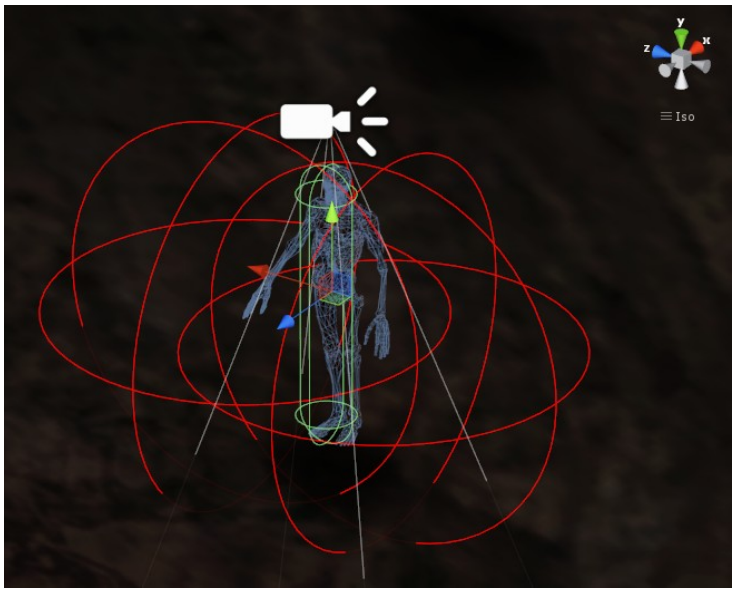


Figura 110: "Zombie" seleccionat

9.2.3 Moviment del personatge

El “Player” ja té associades les animacions que necessita i es manté sobre el terreny i col·lisiona amb els altres elements de l'escenari gràcies al “Character Controller”. Ara falta dir-li com s'ha de moure i quan ha d'utilitzar les animacions. Per fer-ho li associarem un script específic per ell, el “PlayerMovement.js” (Figura 111).

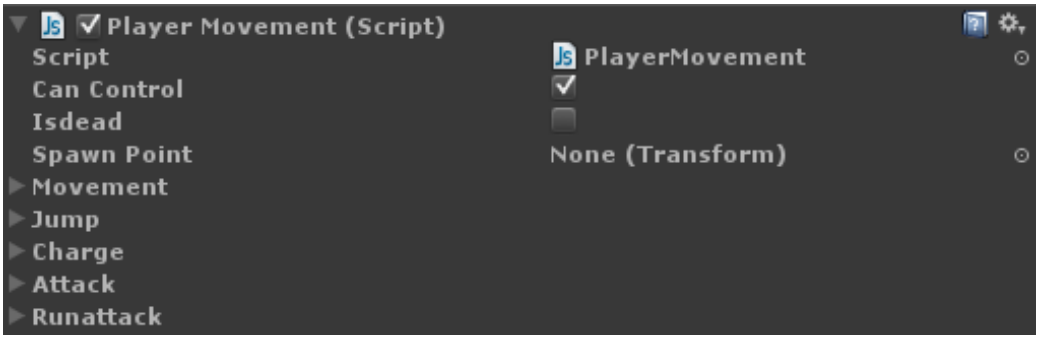


Figura 111: Component principal de l'objecte "Player"

Com es pot observar a la figura anterior, tenim 5 grups de variables; “Movement”, “Jump”, “Charge”, “Attack” i “Runattack”. Cada un d'ells correspon a una classe dins de l'script i sobretot serviran per separar la lògica de cada una de les accions principals i simplificar-ne la codificació i visualització des de l'editor al fer proves, ja que podrem veure en cada moment com van canviant els valors. Per exemple, en el moment que el jugador salta, es pot veure com es marca el booleà “Jumping” (Figura 112).

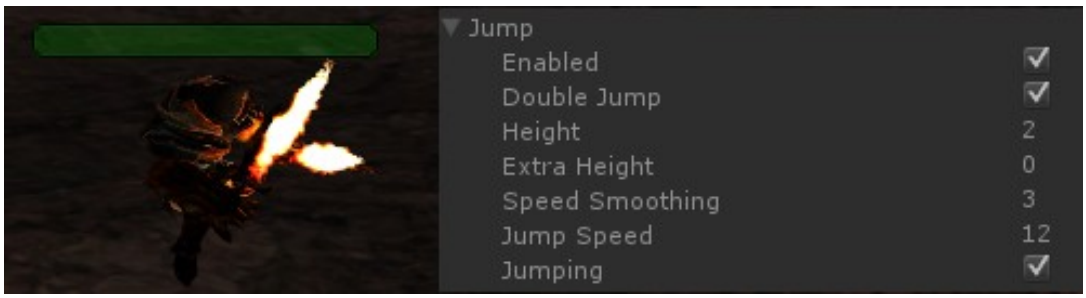


Figura 112: "Player" saltant i vista de les variables

D'ara endavant ens centrarem en explicar el procés pel qual es regeix el moviment del personatge centrant-nos en l'acció de **saltar**. Primer s'inicialitzen les animacions que requereixen un tractament especial dins de la funció “Start()”.

```
animation["jump"].wrapMode = WrapMode.Once;
animation["jump"].speed = 1.5;
animation["jump"].layer = 1;
```

Després es requereix del bucle “Update()” per actualitzar l'estat del personatge a cada frame amb les següents funcions.

```
// Apply movement logic
UpdateSmoothedMovementDirection();
// Apply the correct animation
AnimateCharacter();
// Apply gravity - extra power jump modifies gravity
ApplyGravity ();
// Apply jumping logic
ApplyJumping ();
// Apply charge logic
ApplyCharge ();
// Apply attack logic
ApplyAttack ();
// Apply runattack logic
ApplyRunAttack ();
```

Cada una d'aquestes funcions s'ocupa únicament d'una funció del personatge, és a dir, s'encarreguen de codificar l'entrada que es rep d'un botó, o d'una combinació de botons i que representen una funció específica dins la lògica del personatge.

```
function ApplyJumping () {
	if (Input.GetButtonDown ("Jump") && canControl) jump.lastButtonTime = Time.time;
	if (jump.lastTime + jump.repeatTime > Time.time) return;
	var isGrounded = controller.isGrounded;
	if (isGrounded || JustBecameUngrounded()) {
		if (isGrounded) {
			jump.lastGroundedTime = Time.time;
			if (attack.attackair == true) {
				attack.attackair = false;
				jump.landing = true;
			}
			jump.doubleJump = true;
		}
		if (jump.enabled && Time.time < jump.lastButtonTime + jump.timeout) {
```

```
                movement.verticalSpeed = CalculateJumpVerticalSpeed (jump.height);
                SendMessage ("DidJump", SendMessageOptions.DontRequireReceiver);
            }
        }
        else if (jump.doubleJump && Input.GetButtonDown ("Jump") && canControl) {
            movement.verticalSpeed = CalculateJumpVerticalSpeed (jump.height);
            jump.doubleJump = false;
            jump.jumping = true;
        }
}
```

Al tenir les diferents classes per cada un dels aspectes principals del moviment del personatge, no només permet organitzar més fàcilment la codificació i realitzar proves amb les variables, com ja hem dit abans, sinó que també ens resulta molt més fàcil l'elecció correcta d'animacions. Per exemple, al codi següent podem veure la selecció d'animacions al saltar.

```
        //jump
        else if (jump.jumping && !charge.charging) {
            if (!attack.attackair) {
                animation.CrossFade ("jump");
            }
            else {
                animation.Play ("jumpdouble");
            }
        }
```

9.2.4 Detecció de col·lisions del personatge

Com ja hem apuntat en diverses ocasions, nosaltres no ens encarregarem de la detecció explícita de les col·lisions, és a dir, no hem de crear un algoritme que permeti detectar quan dos objectes (o més) col·lisionen. Dit això, el que sí que haurem de fer és especificar de quina manera gestionem aquestes col·lisions.

Com hem començat a explicar a l'apartat 9.2.2, quan el “Player” cau dins d'un riu de lava, aquest envia una senyal a l'objecte “Player” indicant-li que ha establert contacte. Recordem que això es produeix mitjançant un script que correspon al riu (LavaDamage), i que crida a una funció d'un altre script que correspon al “Player” (LavaTouch); i per tal que això succeeixi s'han d'activar els “Triggers” (disparadors).

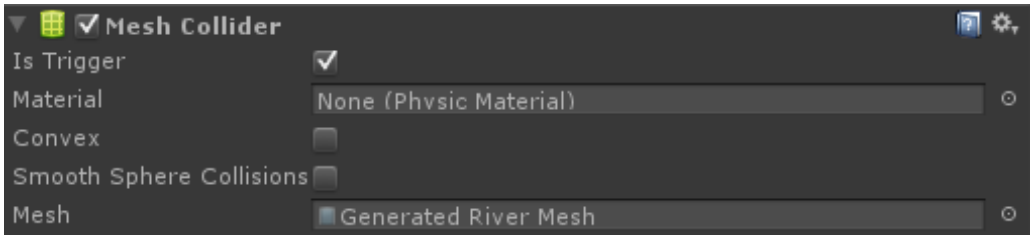


Figura 113: Component "Mesh Collider" d'un riu de lava

Com podem observar (Figura 113), el flag “Is Trigger” es troba activat, el que indica que quan detecti una col·lisió no es comportarà com un cos sòlid, sinó com un disparador d'events sobre les funcions “OnTriggerEnter”, “OnTriggerExit” i “OnTriggerStay”.

Un altre tipus de col·lisions la trobarem a l'hora d'atacar i ferir els enemics. En aquest cas, donat que no tenim un col·lisionador a les armes del personatge principal o als braços dels zombies, els hi em afegit un script a cada espasa o a cada braç. Aquest script mirarà si dins d'un radi d'acció a partir d'un punt determinat (Figura 114), cobrim la distància entre el punt de l'espasa i la posició del zombie; si és així li enviarem un missatge dient que l'hem tocat, si estem realitzant un atac.

Figura 114: Representació del rang d'atac d'una espasa

```
function DidSword() {
	var pos = transform.TransformPoint(swordPosition);
	var enemies : GameObject[] = GameObject.FindGameObjectsWithTag("Enemy");

	for (var go : GameObject in enemies) {
		var enemy = go.GetComponent(ZombieTakeDamage);
```

```
            if (enemy == null) continue;

            var sqrLen = (enemy.transform.position - pos).sqrMagnitude;
            // square the distance we compare with
            if (sqrLen <= swordRadius*swordRadius)
                  enemy.SendMessage("ApplyDamage", swordHitPoints);
      }
}
```

9.2.5 Intel·ligència artificial dels enemics

Donat que en la versió d'Android ja hem implementat l'algoritme per buscar el camí més més curt per un seguit de plataformes, en aquest cas aprofitarem les capacitats de l'escenari en 3D per fer alguna cosa diferent.

El “Player” només es mou a través d'una línia imaginària, aleshores seria normal que els enemics tinguessin un comportament semblant, però el problema sorgiria a l'hora de tenir més d'un enemic intentant acostar-se a nosaltres si aquests es troben a la mateixa línia, s'aglutinarien l'un rere l'altre ja que en aquest cas no es poden solapar com en el cas de l'altra versió.

Així doncs, aprofitant que disposem de molt d'espai al nostre voltant, simplement deixarem que els enemics puguin moure's en qualsevol direcció, el que permetrà obtenir un resultat molt més realista.

Per complementar el que hem fet a l'altre versió, en aquest cas farem que els enemics només es dirigeixin cap a nosaltres quan entrem en el seu radi d'acció (Figura 115), i es podran acostar a nosaltres des de qualsevol punt.

```
function UpdateSmoothedMovementDirection () {
      var h = 0;
      var isdead = isPlayerDead.isDead();
      if (!zhitted ) {
            // Look at which direction the player is
            if (alert) {
                  if (player.transform.position.x < gameObject.transform.position.x &&
                   gameObject.transform.position.x - player.transform.position.x >=
                   0.1) h = -1;
                  else if (player.transform.position.x >
                   gameObject.transform.position.x && player.transform.position.x -
```

```
                gameObject.transform.position.x >= 0.1) h = 1;
        }
        // Calculate range
        if (inrange || isdead) h = 0;
        var dist = Vector3.Distance(player.transform.position,
         gameObject.transform.position);
        if (Mathf.Abs(dist) < 20) alert = true;
        if (Mathf.Abs(dist) < 1.8) inrange = true;
        else inrange = false;
        // Set the movement direction
        movement.isMoving = Mathf.Abs (h) > 0.1;
        if (movement.isMoving) {
              movement.direction = Vector3 (h, 0, player.transform.position.z-
               gameObject.transform.position.z);
              movement.direction.Normalize();
        }
        // Smooth the speed based on the current target direction
        var curSmooth = 0.0;
        // Choose target speed
        var targetSpeed = Mathf.Min (Mathf.Abs(h), 1.0);
        // If the Player is detected we move
        curSmooth = movement.speedSmoothing * Time.smoothDeltaTime;
        if (alert && !inrange) targetSpeed *= movement.runattSpeed;
        else targetSpeed *= 0.0;
        movement.hangTime = 0.0;
        movement.speed = Mathf.Lerp (movement.speed, targetSpeed, curSmooth);
    }
}
```

La funció “Update()” seguirà el mateix patró que la de l'objecte “Player”, però al final s'hi haurà d'afegir la rotació per encarar-se al seu objectiu.

```
    // Set rotation to the move direction
    if (movement.direction.sqrMagnitude > 0.01)
          transform.rotation = Quaternion.Slerp (transform.rotation,
           Quaternion.LookRotation (movement.direction), Time.smoothDeltaTime *
           movement.rotationSmoothing);
```

Figura 115: Entrant al radi d'acció dels enemics

9.2.6 Representació d'animacions i moviment de la càmera

Com ja hem comentat amb anterioritat, les animacions provenen d'un videojoc al que els hi hem extret i posteriorment importat a 3DsMax per a l'edició. Aquestes animacions es basen en un esquelet que representa els ossos humans o al menys les parts més representatives de la seva mobilitat. Per aconseguir el personatge que volíem, s'ha hagut de fer una modificació de l'esquelet (Figura 116).

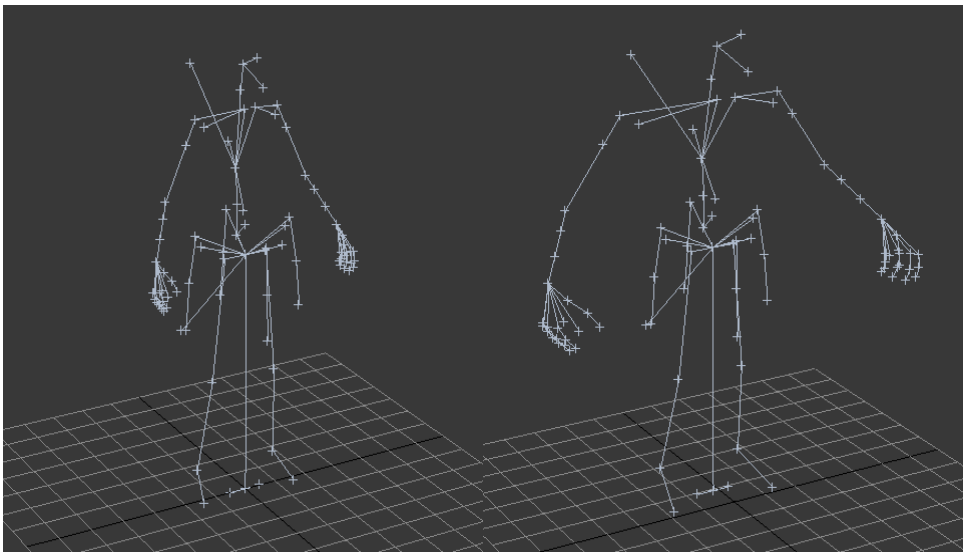


Figura 116: Modificació de l'esquelet

Com a conseqüència d'haver modificat l'esquelet, un cop importem les animacions amb la nova estructura, aquestes no acabaran d'ésser acurades i s'hauran de reajustar.

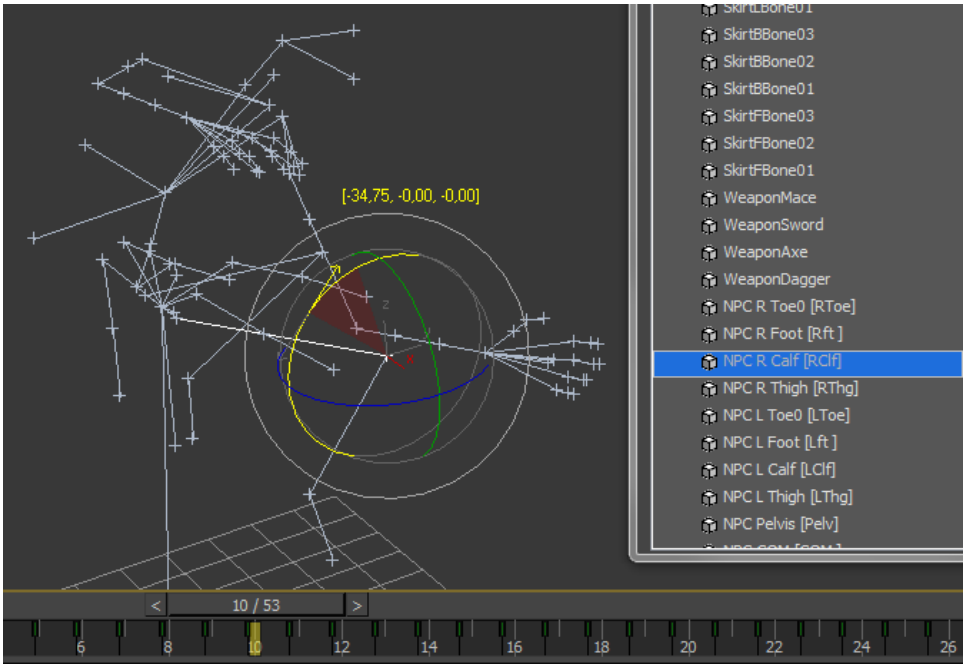


Figura 117: Reajustament d'animació "jump"

Podem observar (Figura 117) com a la part inferior tenim seleccionat el frame a modificar, i a la dreta l'os sobre el que apliquem la rotació, per corregir el moviment.

Un cop tenim l'animació com nosaltres desitgem tenim dues opcions; o bé ho guardem en ".max" i amb un format de nom específic (Player@"nom d'animació"), o escollim l'opció d'exportar a format ".fbx". En qualsevol dels casos, haurem d'importar els fitxers des de l'editor de Unity. Pel primer cas però, la conversió a format "fbx" la realitzarà Unity.

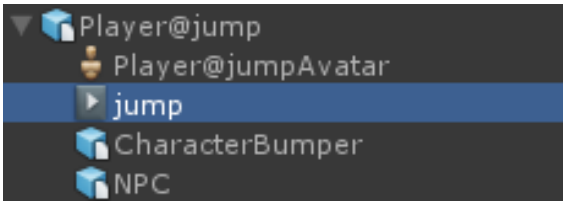


Figura 118: Animació fbx "jump"

Pel que fa al moviment de la càmera, com hem apuntat a l'apartat 9.2.1, es disposa de l'script "SmoothFollow2D", al que li associem l'objecte "Player" per tal que sàpiga què és el que ha de seguir.

```
var target : Transform;
var smoothTime = 0.3;
private var thisTransform : Transform;
private var velocity : Vector2;

function Start() {
	thisTransform = transform;
}

function LateUpdate() {
	thisTransform.position.x = Mathf.SmoothDamp( thisTransform.position.x+0.3,
		target.position.x, velocity.x, smoothTime);
	thisTransform.position.y = Mathf.SmoothDamp( thisTransform.position.y+0.4,
		target.position.y, velocity.y, smoothTime);
}
```

El que fem és ordenar que s'estacioni lleugerament a sobre i a la dreta de la posició del "Player" de manera que tinguem un camp de visó ampli del que tenim per davant, i se li diu que no ho faci a l'instant, sino amb un petit retràs, suaument.

10. Resultats

En aquesta secció s'han realitzat els dos prototips plantejats originalment d'acord amb les especificacions donades als apartats anteriors.

10.1 Prototip Android/LibGDX

Farem un repàs a totes les funcionalitats de la demo.

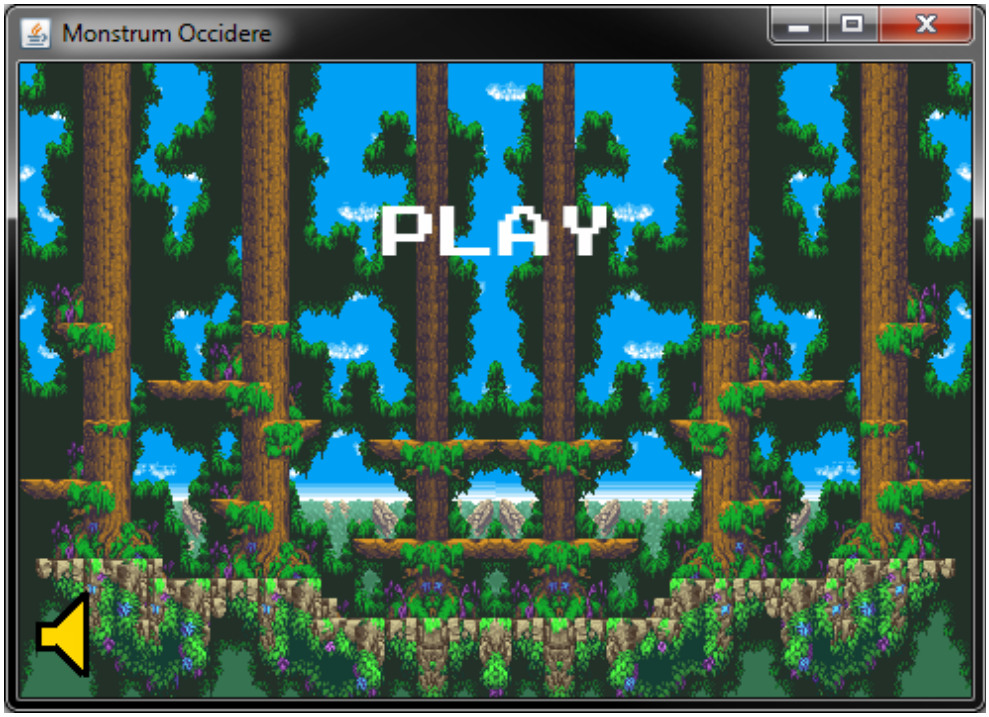


Figura 119: Pantalla d'inici

Pantalla d'inici (Figura 119) amb les opcions de començar la partida i apagar/activar la música (Figura 120).

Figura 120: So

Al Fer click sobre "Play" carregarem la pantalla de joc (Figura 121).

Figura 121: Pantalla de Joc: "Ready?"

Quan estiguem preparats, fent click sobre qualsevol part de la pantalla, començarem a jugar i els enemics dirèctament van a buscar al personatge per atacar-lo (Figura 122).

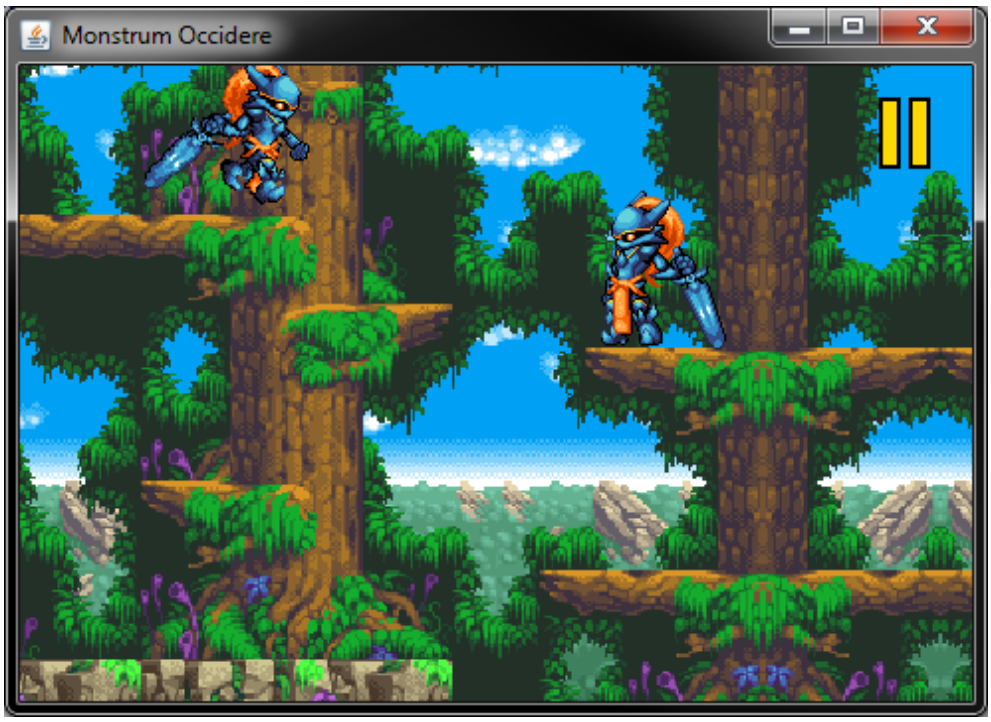


Figura 122: Pantalla de joc, inici

Podem fer click sobre el símbol de pausa per detenir l'acció i sel·leccionar opcions.

Figura 123: Pantalla de joc, menú pausa

Al fer click sobre "resume" (Figura 123) reprenem l'acció. Si ho fem sobre "quit" tornarem a la pantalla d'inici. Un cop represa l'acció podem veure, si canviem la posició del personatge (Figura 124), com els enemics reajusten el seu camí per anar-lo a trobar sempre (Figura 125) .

Figura 124: Pantalla de joc, enemics reajustant el camí (1)

Figura 125: Pantalla de joc, enemics reajustant el camí (2)

En qualsevol moment podem enfrontar i atacar-los (Figura 126).

Figura 126: Pantalla de joc, atacant un enemic

Per últim podem fer acabar el joc si es surt del mapa per qualsevol dels costats de l'escenari. No es podrà tornar ja que no hi ha plataformes i el personatge cau al buid (Figura 127).

Figura 127: Pantalla de joc, game over

Al fer click tornarem a la pantalla d'inici.

10.2 Prototip Unity

Farem un repàs a totes les funcionalitats de la demo.

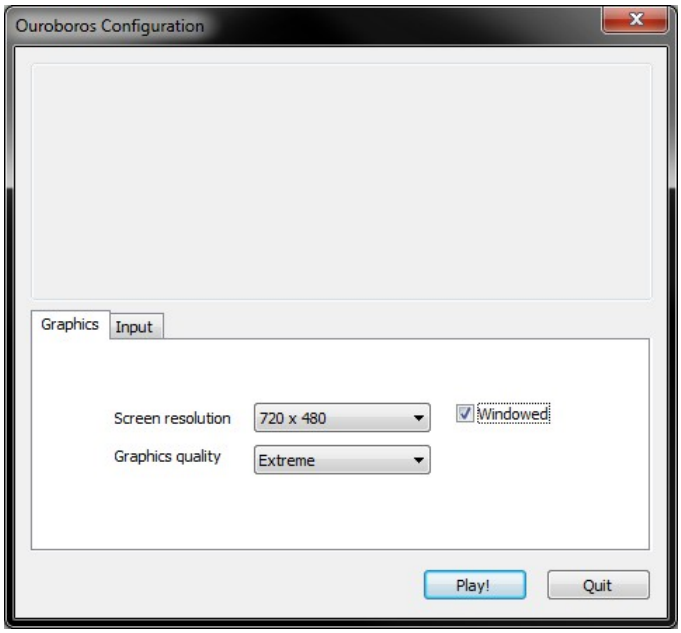


Figura 128: Menú d'arranc, sel·lecció dels gràfics i altres opcions

Amb l'executable del prototip de Unity ens trobem amb el menú de d'arranc (Figura 128), on podem configurar la resolució de pantalla, la qualitat dels gràfics, si es jugarà a pantalla completa o amb una finestra; es canvien els controls dins la pestaña "Input". Quan premem el botó "Play!" arrencarà la demo.

Figura 129: Pantalla de joc, inici

A partir d'aquest moment podem controlar el personatge i moure'l per l'escenari (Figura 129), però també podem obrir la interfície del menú de control, prement una tecla (“Esc”, Figura 130).

Figura 130: Pantalla de joc, menú de control

Aquest menú ens permetrà sortir del joc o tornar a carregar la pantalla d'inici des del començament. Mentre hi ha activat el menú l'acció no s'atura, de fet es pot controlar el personatge amb el menú actiu (Figura 131).

Figura 131: Pantalla de joc, acció ininterrompuda

Continuant ens trobem amb el primer grup d'enemics, als quals podem enfrontar.

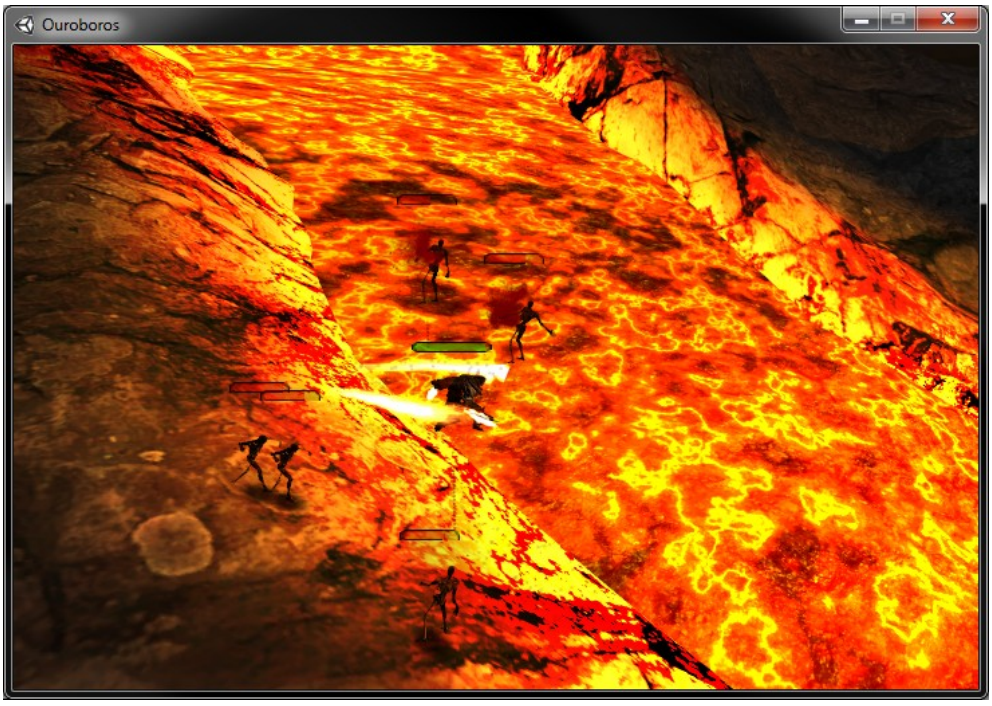


Figura 132: Pantalla de joc, enfrontament contra enemics

Si s'ha sobreviscut a l'enfrontament (Figura 132) podem continuar endavant i creuar el riu de lava. Per fer-ho serà necessària una combinació de moviments per tal de poder arribar a cada una de les plataformes que hi ha enmig del riu. Aquest és un dels elements de "puzzle" que comentàvem dins l'apartat 1.2, i en aquest cas la combinació "Salt + doble Salt + Càrrega" (Figura 133) o "Salt + Càrrega + doble Salt".

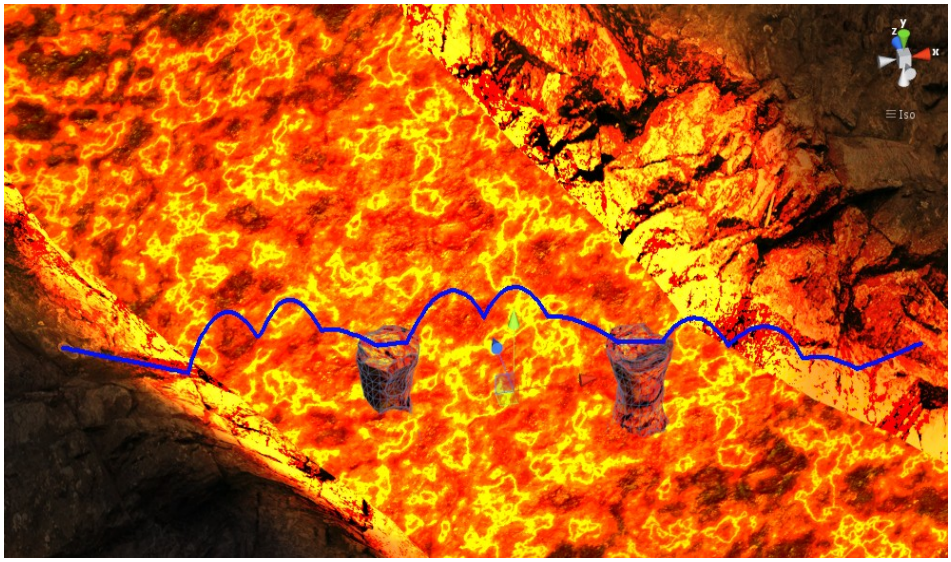


Figura 133: Esquema del recorregut per poder creuar el riu de lava

En cas que caiguem dins la lava o els enemics ens deixin sense vida, es mostra un missatge que indica que hem perdut (Figura 134).

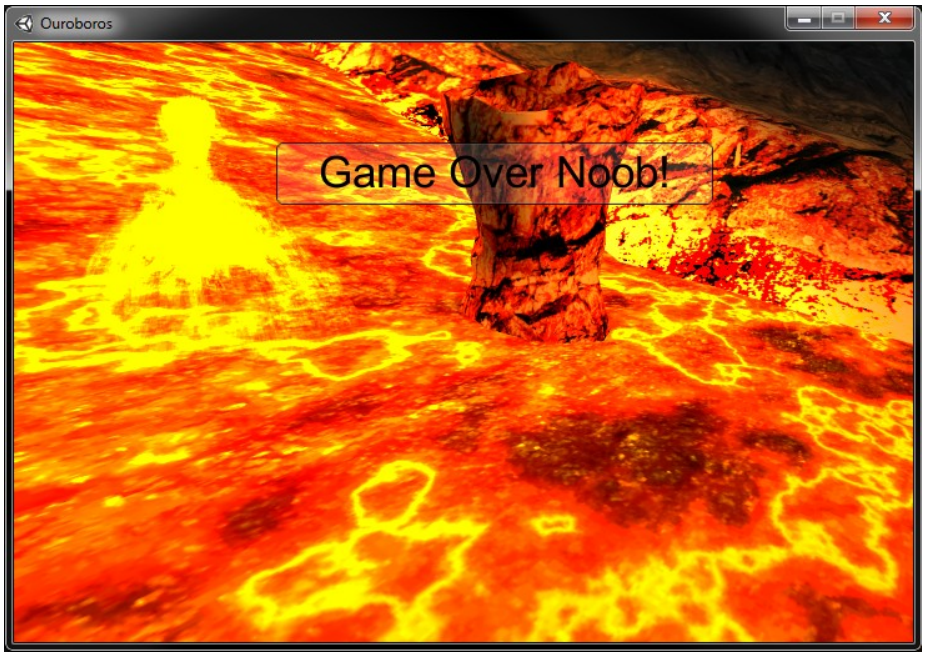


Figura 134: Pantalla de joc, mort per lava

En aquest moment si volem tornar a jugar o sortir farem servir el menú de control (Figura 135).

Figura 135: Pantalla de joc, mort per enemics + menú

11. Conclusions

En general els resultats han estat els esperats en cada una de les versions, i en el procés general de desenvolupament. Anem a desglossar-ne els diferents aspectes.

11.1 Temporització

Donat que aquest desenvolupament es centrava en l'obtenció de prototips i es realitzaven totes les tasques de disseny artístic i conceptual, no es pot saber amb exactitud quantes hores s'han destinat totalment a aquestes parts del projecte, ja que moltes hores destinades a aquestes tasques s'han manifestat fora de l'entorn laboral en “moments d'inspiració”.

Tot i així s'ha acabat amb un temps lleugerament superior a l'esmentat des de l'inici, si es descompten els dies en què no s'ha pogut treballar per períodes de vacances o compromisos personals/laborals que han anat sorgint en el procés. També podem dir que el nombre d'hores, que s'han dedicat en total, és superior al càlcul realitzat dins l'apartat 2.4 i la planificació (Figura 9), sobretot a causa de l'adició de detalls i funcionalitats no plantejades a l'inici.

Hem de destacar que el flux de treball també ha variat respecte al que teníem planificat originalment; s'han solapat tasques que en un principi pensàvem que serien aïllades i s'ha vist que la primera planificació era poc realista en aquest sentit (Apartat 4).

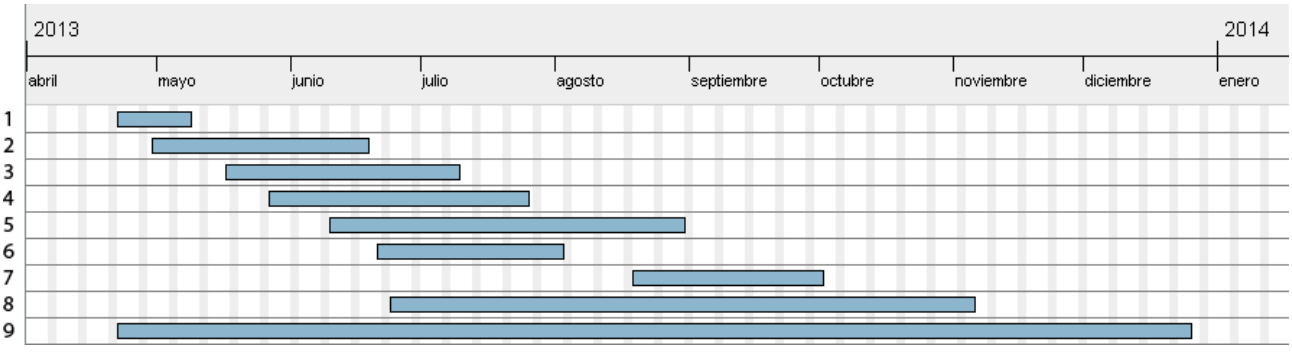


Figura 136: Diagrama de Gantt final

11.2 Canvi realitzat

El canvi més destacat s'ha realitzat dins el prototip d'Android/LibGDX al tenir problemes

amb el motor de col·lisions dels enemics. La codificació d'aquest algoritme no resulta prou eficient per poder mantenir en escena un nombre gaire elevat d'enemics, pel que s'ha optat per generar un escenari en el que només existeix un sol enemic, enfocant aquesta versió a un joc de lluita 1vs1, enlloc de l'esmentat "Beat 'em up" a l'apartat 1.1, però mantinguen la resta de característiques.

11.3 Conclusions

Aquest projecte apuntava a la resolució sobre quina és la millor manera d'aproximar-se a la creació d'un videojoc, en aquest cas en un gènere determinat. Sembla evident pensar que s'ha de desenvolupar una forma de treball estandarditzada per cada pas del procés de creació. Anem a desglossar quin ha d'esser segons l'experiència resultats obtinguts.

11.3.1 Creació de continguts

Quan parlem de creació de continguts ens referim a la capacitat de desenvolupar/obtenir nous recursos, sobretot gràfics. Aquesta capacitat vindrà donada per les habilitats dels integrants de l'equip i la infraestructura d'aquest, incloent-hi el pressupost.

Ja més concretament, en el dos casos que ens ocupen, podem veure com pel prototip d'android no necessitem d'una infraestructura i uns recursos que vagin més enllà del temps que dediquem a generar dissenys nous (sprites) i animacions, ja que no són més que dibuixos. En canvi pel prototip de Unity necessitaríem d'una infraestructura de programari i instal·lacions que suposarien una gran inversió, ja que requeriríem d'una sala de captura de moviment i un programari molt costós.

En cas que es disposés del capital i les instal·lacions necessàries, escolliríem desenvolupar amb Unity i amb models i animacions 3D, ja que és molt més ràpid i senzill crear o modificar animacions que no pas amb un sistema com el de la versió 2D on s'ha de crear absolutament tot a mà.

11.3.2 Codificació de les mecàniques jugables

Les mecàniques jugables fan referència als comportaments que tindran els elements del joc, tant individualment com interactuant entre ells. Aleshores, partint de les base que tota la feina per reproduir aquestes mecàniques que nosaltres dissenyem a una implementació, ha estat realitzada pel nostre compte, el resultat serà molt més acurat que,

si pel contrari, s'utilitzen recursos externs per acabar el desenvolupament.

Com a contrapartida, hi han els riscos que comporten la total implementació d'una solució: Pot resultar errònia o insatisfactòria a causa d'una falta de coneixements o mala planificació, i el temps que es necessita per desenvolupar-la és més alt. El més eficient és, per tant, després d'un estudi previ, buscar implementacions que compleixin amb els requisit de diferents parts i siguin compatibles amb les mecàniques jugables plantejades.

Posant com a exemple els nostres prototips, en el d'Android/LibGDX és ideal poder controlar al 100% el comportament del nostre personatge, però per contrapartida el motor de col·lisions dissenyat per ell no ha resultat esser útil per més d'una instància en el cas dels enemics; per altre banda, Unity ofereix un sistema que compleix amb les especificacions del sistema de col·lisions que havíem pensat, i és totalment robust.

11.3.3 Explotació del treball realitzat

Com a explotació ens referim al profit monetari que es traurà de la venta i/o distribució de l'aplicació desenvolupada. És evident que el millor sistema serà aquell que permetrà fer arribar l'aplicació al màxim nombre de persones possible, i amb més rapidesa.

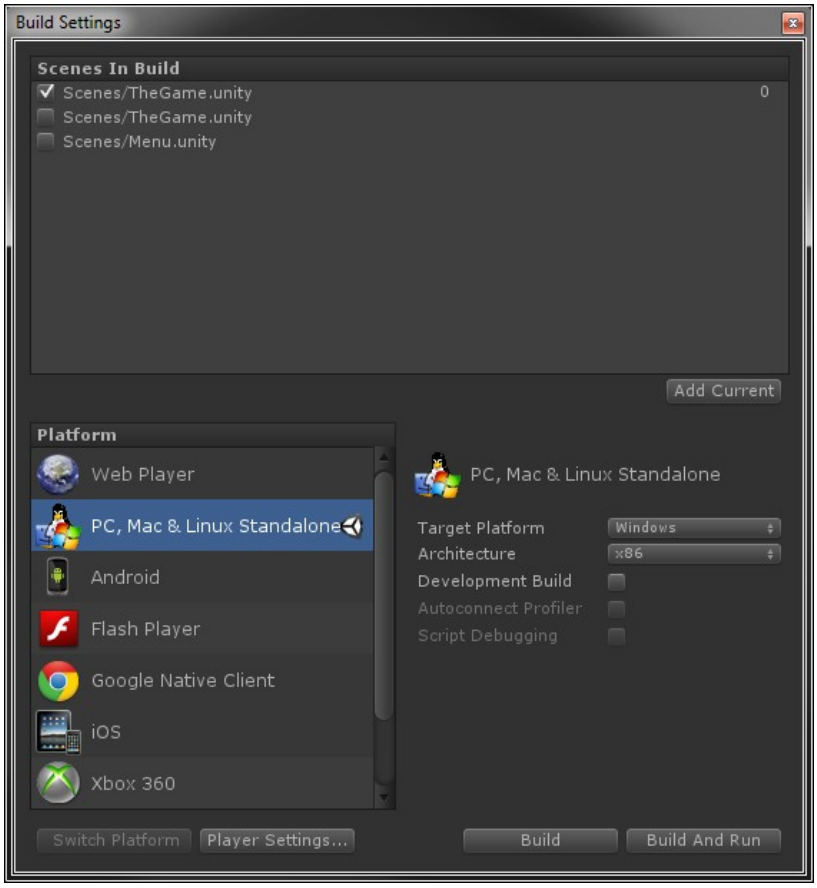


Figura 137: Unity, configuració de creació

En aquest cas, com en el primer, estem supeditats al pressupost o inversió de la que disposem per comprar i/o utilitzar un programari. Si fos així, la millor opció seria Unity, ja que permet exportar (Figura 137) el treball realitzat a molts tipus de plataformes de forma molt senzilla, que no pas en comparació amb Android/LibGDX.

Per tot el que hem explicat, decidim quedar-nos amb la plataforma Unity per ser un sistema compacte en constant actualització, per donar moltes facilitats a l'hora de crear contingut nou, per ser versàtil quan es vol crear un joc per diferents plataformes i per la facilitat de resoldre dubtes i problemes gràcies a la gran comunitat d'usuaris que té al darrere.

12. Treball futur

Des que es va acabar la implementació dels prototips i es va decidir continuar amb Unity el treball que es realitza, com apunta el títol d'aquest treball, és la realització d'un videojoc anomenat “Ouroboros”.

S'ha escrit el guió i tot el transfons de la història, totes les mecàniques jugables (veure annexos), les vies de monetització, s'han establert contactes amb professionals dins dels diferents àmbits de producció (dissenyadors 2D, dissenyadors 3D, tècnics de sistemes MoCap, compositors de música, programadors, fotògrafs, ...) i ja està creat el disseny del personatge principal (Figura 138 i Figura 139). En el moment d'escriure aquestes línies s'està realitzant la selecció de candidats per portar el disseny del personatge a 3D.

La intenció en els darrers mesos és crear els recursos suficients per generar confiança a futurs usuaris, i presentar un projecte per finançar la creació del joc via micromecenatge. Un sistema de finançament en expansió, basat en petites donacions a canvi de recompenses. Després d'estudiar la plataforma i els projectes que s'hi presenten, es vol intentar arribar a l'objectiu de 100.000 USD.

13. Bibliografia

Android

Google Inc.(2013). *Android Developers, Reference*. Recuperat 20 de Desembre 2013 des de http://developer.android.com/reference/android/package-summary.html.

Mario Zechner (2013). *Android/iOS/HTML5/desktop game development framework*. Recuperat 20 de Desembre 2013 des de https://code.google.com/p/libgdx/.

Mario Zechner (2013). *BadLogic Games, Wordpress*. Recuperat 20 de Desembre 2013 des de http://www.badlogicgames.com/wordpress/.

Mario Zechner (2011). *An Introduction to libgdx*. Recuperat 20 de Desembre 2013 des de http://www.youtube.com/watch?v=vLx_72qxK_0.

Mario Zechner (2013). *Desktop/Android/HTML5/iOS Java game development framework*. Recuperat 20 de Desembre 2013 des de https://github.com/libgdx/libgdx.

Antoine Dutot, Yoann Pigné (2013). Org.graphstream.algorithm.Astar (.java) - Class - Source Code View. Recuperat 20 de Desembre 2013 des de http://grepcode.com/file/repo1.maven.org/maven2/org.graphstream/gs-algo/1.0/org/graphstream/algorithm/AStar.java#AStar.aStar%28org.graphstream.graph.Node%2Corg.graphstream.graph.Node%29.

Hannes Drexl (2008). *Pixel Art Tutorial: Optimizing Tilesheets*. Recuperat 20 de Desembre 2013 des de http://www.elitepigs.de/blog/archives/17.

Logan Tanner (2010). *The Pixel Art Tutorial*. Recuperat 20 de Desembre 2013 des de http://www.pixeljoint.com/forum/forum_posts.asp?TID=11299.

Alexander Steiner (2008). *Elk's "How to rotate" Tut*. Recuperat 20 de Desembre 2013 des de http://darkshire.deviantart.com/art/Elk-s-quot-How-To-Rotate-quot-Tut-91120166.

Unity

Unity Technologies (2013). *Unity Documentation. User Manual - Component Reference - Scripting Reference*. Recuperat 20 de Desembre 2013 des de http://unity3d.com/learn/documentation.

Unity Technologies (2013). *The best place for answers about Unity - Unity Answers*.

Recuperat 20 de Desembre 2013 des de http://answers.unity3d.com/index.html.

Six Times Nothing (2010). *Terrain Toolkit - Tutorial*. Recuperat 20 de Desembre 2013 des de http://www.youtube.com/watch?v=YnO9RtarzHE.

TowerJimmy (2012). *Skyrim Modding Tutorial Importing Animations from Skyrim into 3ds Max part 2 English HD*. Recuperat 20 de Desembre 2013 des de http://www.youtube.com/watch?v=OoyP_SWaxnE.

Joe Pikop (2011). *How to get Custom Armor Into Skyrim pt. 1*. Recuperat 20 de Desembre 2013 des de http://www.youtube.com/watch?v=L40qoiuUXL4.

Mike Bauer (2013). *How to Texture a Head Model in 3ds Max*. Recuperat 20 de Desembre 2013 des de http://www.youtube.com/watch?v=_ljpbG6lquQ.

Jixuguo (2013). *Unity3D Engine Architecture*. (Recuperat 20 de Desembre 2013 des de http://blog.csdn.net/jixuguo/article/details/7351307.

14. Annexos

Figura 138: Disseny de les classes del personatge principal

Figura 139: Disseny de personalitats pel personatge principal

Figura 140: Logotip "Ouroboros"

El sistema de joc que s'està creant està basat en els 4 elements: terra, aigua, foc i vent. S'ha agafat cada un dels elements i se n'han fet 2, separant-los en llum i foscor (Figura 141).

	Earth	Watter	Fire	Wind
Light	Nature	River	Aura	Sky
Dark	Land	Abyss	Lightning	Hurricane

Figura 141: Taula d'elements

D'aquí provenen les 8 classes diferents del personatge (Figura 138).

A part de la classe també tindrà un tipus de personalitat (Figura 139) de les 9 disponibles (Figura 142).

		Nice person		
	1.- Lawful Good	2.- Neutral Good	3.- Chaotic Good	
Play by rules	4.- Lawful Neutral	5.- True Neutral	6.- Chaotic Neutral	Doesn't play by rules
	7.- Lawful Evil	8.- Neutral Evil	9.- Chaotic Evil	
		Prick		

Figura 142: Taula de personalitats

Les classes i personalitats del personatge dependran de les 8 habilitats passives, que el jugador hagi elegit de les 64 possibles (Figura 143).

66,66% 33,33%	Light Light	Light Darkness	Darkness Light	Darkness Darkness
Earth Earth	2 / La Protección	15 / La Modestia	23 / La Opresión	52 / El Aquietamiento
Earth Watter	19 / El Acercamiento	7 / El Ejercito	41 / La Limitación	4 / Descubrir
Earth Fire	36 / El Eclipsamiento	24 / El Retorno	22 / La Apariencia	27 / Tragar
Earth Wind	11 / La Abundancia	46 / La Subida	26 / Gran dedicación	18 / La Decadencia
	Light Light	Light Darkness	Darkness Light	Darkness Darkness
Watter Watter	58 / Lo Sereno	47 / La Adversidad	60 / La Restricción	29 / Lo Abismal
Watter Earth	45 / La Unidad	31 / El Influjo	8 / La Alianza	39 / El Impedimento
Watter Fire	49 / La Renovación	17 / El Seguimiento	63 / Cerrar un Ciclo	3 / Acumular
Watter Wind	43 / La Revolución	28 / La Sobrecarga	5 / La Paciencia	48 / El Pozo
	Light Light	Light Darkness	Darkness Light	Darkness Darkness
Fire Fire	30 / El Discernimiento	21 / Eliminar Obstáculos	55 / La Plenitud	51 / La Conmoción
Fire Earth	35 / El Progreso	56 / El Andariego	16 / El Entusiasmo	62 / Importancia de lo Pequeño
Fire Watter	38 / El Antagonismo	64 / El Porvenir	54 / La Concubina	40 / La Liberación
Fire Wind	14 / El Dominio	50 / La Marmita	34 / La Iniciativa	32 / La Constancia
	Light Light	Light Darkness	Darkness Light	Darkness Darkness
Wind Wind	1 / El Movimiento	44 / La Complacencia	9 / Suave Progreso	57 / Lo Penetrante
Wind Earth	12 / El Estancamiento	33 / Eludir	20 / Analizar	53 / El Avance
Wind Fire	13 / La Amistad	25 / La Espontaneidad	37 / El Clan	42 / El Aumento
Wind Watter	10 / El Paso	6 / El Conflicto	61 / Fe Interior	59 / La Disolución

Figura 143: Taula d'habilitats passives

Les 8 habilitats escollides es dipositen dins la “Taula de Chakras” (Figura 144) si el seu tipus coincideix amb els del chakra on la volem posar (no entrarem a especificar els tipus d'habilitats).

La classe resultant serà determinada per la combinació dels 3 elements predominants. En cas d'empat s'agafaran els elements dels 3 primers chakras començant per dalt.

Seat of soul	DEF BUFF	SPD BUFF	ATT DBUFF	SPD DBUFF
	HEAL/+RES	ATT BUFF	DEF DBUFF	CC
Cognitivity	CC	ATT BUFF		
	SPD DBUFF	EXP BUFF		
Clairvoyance	EXP BUFF	HEAL/+RES		
	DEF BUFF	DEF DBUFF		
Influence	ATT DBUFF	SPD DBUFF		
	DEF DBUFF	CC		
Self-worth	HEAL/+RES	ATT DBUFF		
	SPD BUFF	EXP BUFF		
Power	ATT BUFF	DEF DBUFF		
	ATT DBUFF	SPD BUFF		
Aliveness	DEF BUFF	EXP BUFF		
	HEAL/+RES	ATT BUFF		
Survivability	SPD BUFF	SPD DBUFF		
	DEF BUFF	CC		

Figura 144: Taula de Chakras

La personalitat resultant vindrà donada per la quantitat de llum o foscor que tinguin les habilitats que s'hagin escollit.

1.- Lawful Good	24 -> 22 0 -> 2
2.- Neutral Good	21 -> 19 3 -> 5
3.- Chaotic Good	18 -> 16 6 -> 8
4.- Lawful Neutral	15 -> 13 9 -> 11
5.- True Neutral	12 12
6.- Chaotic Neutral	11 -> 9 13 -> 15
7.- Lawful Evil	8 -> 6 16 -> 18
8.- Neutral Evil	5 -> 3 19 -> 21
9.- Chaotic Evil	2 -> 0 22 -> 24

La puntuació per determinar els tipus en els dos casos, es mira tenint en compte que cada habilitat té 3 punts; 2 per l'element superior i 1 per l'inferior (24 en total).

S'ha implementat un algoritme per determinar la quantitat de combinacions vàlides que podem fer amb la taula de chakras. El resultat ha estat de 1.951.186.568 combinacions diferents d'habilitats.

15. Manual d’usuari i/o instal·lació

Ara explicarem els controls dels 2 prototips i els seus mètodes d'instal·lació .

15.1 Manual d'usuari

Els dos prototips tenen controls i funcionalitats lleugerament diferents, anem a especificar-los per cada un.

15.1.1 Android/LibGDX

Moure's: 

Saltar:  Quan ens trobem al terra.

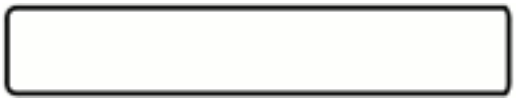
Doble salt:  Quan ens trobem a l'aire. Es recarrega al tocar a terra.

Càrrega:  En qualsevol moment. Es recarrega al moure'ns per una plataforma.

Atacar:  Quan ens trobem quiets, repetidament (5x Combo).

15.1.2 Unity

Moure's: 

Saltar:  Quan ens trobem al terra.

Doble salt:  Quan ens trobem a l'aire. Es recarrega al tocar a terra.

Càrrega:  En moviment. Es recarrega al cap d'un segon i mig.

Voltereta:  En moviment, després de realitzar una càrrega.

Bloqueig:  Quan ens trobem quiets.

Atacar:  En qualsevol moment, repetidament (Quiet: 5x combo, Corrent: 6x combo, Saltant: atac giratori).

15.2 Manual d'instal·lació

Pels dos prototips s'han creat executables per Windows, i no es requereix res més que tenir instal·lat JRE (Java Runtime Environment) pel prototip Android/LibGDX.