Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Enginyeria Informàtica

Títol: Títol: Medieval Wars – Un joc d'estratègia per torns fet en Unity

Document: Memòria

Alumne: Lluís Ramon Armengol Xandri

Tutor: Gustavo Patow
Departament: Informàtica i Matemàtica Aplicada
Àrea: LSI

Convocatòria (mes/any) Setembre/2018

Índex de continguts

1. Introducció, motivacions, propòsit i objectius del projecte

1.1 Introducció

L'industria dels videojocs és, sinó la que més, una de les indústries més importants en el sector de l'entreteniment, juntament amb la de la música i la del cinema. A l'igual que aquestes dos últimes, ofereix una gran quantitat i varietat de títols i gèneres.

Aquest sector va néixer als anys 70, quan Atari Computers va llençar a la venta el considerat primer videojoc comercial, Computer Space, l'any 1971. A partir de llavors, el sector ha anat creixent sense parar: cal destacar la creació de noves plataformes de joc, com les consoles portàtils i les consoles de sobretaula, i l'accés a dispositius com els mòbils i/o tablets. En la Figura 1.1 es pot veure la previsió de ventes de videojocs de cara al 2019:

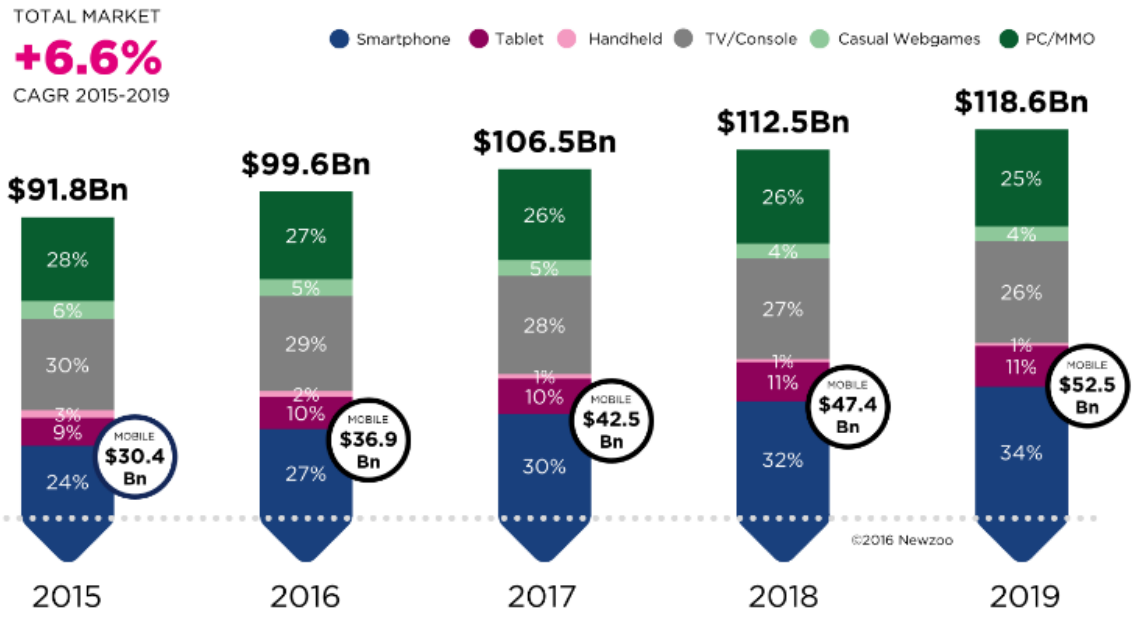


Figura 1.1: Previsió de ventes de videojocs

Com podem observar, es preveu que les ventes seguiran pujant i un factor molt important serà degut als mòbils o tablets, ja que cada cop són més accessibles.

1.2 Motivacions

Desde ben petit, sempre m'han agradat molt els videojocs i hi he dedicat una gran part del meu temps lliure. A mesura que he crescut, m'he anat interessant més i més en aquest sector, ja que no dista molt del món de la creació de software en el qual estic treballant actualment.

Gràcies a les xarxes socials, i mitjans especialitzats he pogut descobrir com ha sigut la evolució de molts títols nous i la gent que hi ha estat treballant.

Un altre punt molt important pel qual he decidit fer un videojoc en el TFG és perquè, desde fa bastant temps, volia crear un videojoc desde 0, i gràcies a aquest projecte puc fer-ho: puc fer el joc que volia i a més puc fer un projecte de fi de grau que m'interessa i em motiva. A més a més, aquest joc serà avaluat i això em servirà per aprendre de què he fet malament i què he fet bé.

1.3 Propòsit i objectius

L'objectiu principal d'aquest projecte és dissenyar i desenvolupar un videojoc d'estratègia per torns en 2D. La idea d'aquest videojoc es basa sobretot en els títols com *Advance Wars* i *Fire Emblem*, propietat de l'empresa japonesa Nintendo. Es desenvoluparà mitjançant el motor de jocs Unity i serà executat en ordinadors amb el sistema operatiu Windows.

1.4 Breu introducció als jocs d'estratègia

Els videojocs d'estratègia són un gènere de videojocs que es basen en l'habilitat de pensar i prendre decisions per aconseguir la victoria. Molt sovint són títols de caire bèlic.

Si sortim del món dels videojocs, un exemple clàssic seria el joc d'escacs. En ambdós casos, els jugadors tindran accés a diferents peces o unitats, i hauran de interaccionar amb elles per obtenir un propòsit. En molts d'aquests casos, s'haurà de tenir en compte quines reaccions desencadenaran les accions, el qual es traduïrà en un arbre de decisions.

Aquest gènere engloba diferents subgèneres, ja que hi han factors que són determinants. Els factors més importants i que marquen una gran diferència són la utilització de torns o temps real i l'ús de l'estratègia i les tàctiques

A continuació descriuré aquest factors, juntament amb títols d'exemple:

- Utilització de torns: el jugador només pot jugar en determinades situacions. Això significa que els jugadors només podran jugar quan sigui el seu torn (els torns es basen en una seqüencia: primer un, després l'altre, ... fins l'últim jugador i llavors tornar al primer).
- Temps real: tots els jugadors juguem a la vegada, al mateix temps.

- Ús de l'element estratègic: quins altres elements o opcions té el jugador per decidir els enfrontaments, i com els pot gestionar.
- Ús de l'element tàctic: fins a quin punt el jugador pot intervenir en els enfrontaments

Els dós primers punts són fàcilment reconeixibles i es diferencien per sí sols, mentre que els dos últims apareixen com una combinació i es classifiquen en funció del seu grau d'importància; un acaba predominant més que l'altre. En els següents exemples explicarem les raons per les quals es classifiquen d'aquesta manera:

Videojoc	Basat en torns	Basat en temps real
Element estratègic	Civilization VI	Age of empires II
Element tàctic	XCOM	Total war WARHAMMER

Com a exemples de jocs d'estratègia basats en temps real, tenim el Age of Empires II i el Total War WARHAMMER, i basat en torns tenim el Civilization VI i el XCOM. Llavors, si els avaluem per l'aplicació de estratègia i tàctiques, el Age of empires II i el Civilization VI destaquen més per l'estratègia, ja que es basen molt en gestionar recursos i construir un imperi (tot i que també tenen tocs tàctics, però no tan notoris com els estratègics), mentre que el XCOM i el Total War WARHAMMER es basen més en com gestionar els enfrontaments o combats (també hi ha l'element estratègic, però està menys desenvolupat i destaca menys que el tàctic).

1.5 Organització del document

Aquesta memòria del treball de fi de grau està estructurada en els següents capítols:

1. Introducció: en aquest apartat s'expliquen breument la història dels videojocs, i les meves motivacions, propòsits i objectius del projecte, així com també es descriuen breument els jocs de estratègia i com s'organitzarà aquest document.
2. Estudi de viabilitat: aquí plantejaré hipotèticament el cost que tindria realitzar aquest projecte, suposant que es dispongués d'un equip.
3. Metodologia: de quina manera s'ha realitzat el projecte.

4. Planificació: relacionat amb l'apartat anterior, aquí explicaré com he gestionat el projecte, desdel principi fins al final i com ho he separat. També es farà una breu comparativa entre el com s'esperava que passés i el que ha acabat passant.
5. Marc de treball i concepts previs: explicació de què és un motor de videojoc, alguns exemples i quin s'ha escollit i el per què.
6. Requisits del sistema: explicació dels requeriments funcionals i no funcionals del sistema.
7. Estudis i decisions: software i llibreries utilitzades.
8. Anàlisi i disseny del sistema: explicació de com és el joc, els elements, les estructures, els casos d'ús, etc. Sense entrar en com s'ha implementat.
9. Implementació i proves: aquí explicarem com s'han implementat les parts del joc, entrant en més detall al motor i al codi.
10. Implantació i resultats: estudi sobre les lleis aplicables, i captures de pantalla comentades d'una partida.
11. Conclusions: discussió crítica dels resultats obtinguts, i també alguns problemes trobats al llarg del projecte.
12. Treball futur: possibles noves millores de cara a millorar el projecte, en un futur.
13. Bibliografia: citacions de pàgines web i llibres utilitzats en aquest projecte.
14. Manual d'usuari i/o instal·lació: aquí fem una breu explicació de com executar el joc, el seu objectiu per guanyar o perdre la partida i informació sobre la jugabilitat.

2. Estudi de viabilitat

Per desenvolupar aquest projecte no són necessaris un gran nombre de recursos, i el cos de l'infraestructura és pràcticament inexistent.

El software que s'ha utilitzat al llarg del projecte és el següent:

- Sistema operatiu Windows 10
- Motor de videojocs Unity (versió gratuïta)
- IDE Visual Studio 2017
- Editor d'imatges GIMP

Referent al hardware, s'ha utilitzat, desde l'inici del projecte, un ordinador amb les següents especificacions:

- Processador AMD Ryzen 5
- Memòria RAM 16 GB
- Gràfica Nvidia 1050 Ti

2.1 Recursos humans

Un videojoc consta de 3 pilars fundamentals: els gràfics, el so i la lògica. Cada un d'ells és indispensable i afecta en la qualitat del producte final. Si algun d'aquests 3 pilars falla, el videojoc es veurà seriament afectat.

Aquest projecte no és una excepció. A causa de que només hi ha una persona en el grup, i que no tinc experiència ni en elements visuals ni en musicals, m'he centrat en la part de disseny i programació (la lògica del joc) i he obtingut els altres dos mitjançant plataformes web que donaven accés de forma gratuïta a aquest elements.

2.2 Avaluació de costos i mitjans

Pel desenvolupament d'aquest videojoc és necessari fer un estudi de viabilitat económica i tècnica. Com que el videojoc presenta pocs problemes legals, no serà necessari un estudi de viabilitat legal.

2.2.1 Estudi de viabilitat tecnològica

En aquets estudi, s'ha de definir les necessitats (tant de hardware, com de software) necessàries per tal de poder realitzar el projecte correctament. En el nostre cas, ja es disposava del hardware i software necessaris, i no s'ha necessitat res més del que ja es disposava.

2.2.2 Estudi de la viabilitat econòmica

En aquest estudi tenim els costos de recursos humans i els de maquinària.

Els costos de recursos humans s'han plantejat de manera hipotètica, suposant que es disposa d'un equip en el qual hi ha una persona per rol. En total, tenim un equip de 8 persones, ocupant cada una un rol diferent:

- Artista: encarregat dels elements visuals. 14 €/h
- Compositor/a: s'encarregarà dels efectes de so i la música. 14 €/h
- Programador/a: implementarà el joc. 17 €/h
- Dissenyador/a: decidirà la idea del joc i com seran els escenaris o nivells. 17 €/h
- Cap de projecte: s'encarregarà de gestionar la feina dels integrants de l'equip. 20 €/h
- Tester o membre de QA: comprovarà que la feina que s'està fent és correcte. 14 €/h
- Productor: s'encaregarà de l'apartat de marketing, de cara a fer conèixer el joc i vendre'l. 16 €/h
- Escriptor: crearà la història i narrativa del joc. 14 €/h

Repertint les tasques amb els treballadors, i les hores necessàries ens surten el següents costs:

Tasques	Encarregat/a	Hores	Cost
Proves i tests	Tester	50	700
Animació i modelats	Artista	60	840

Efectes sonors i música	Compositor	60	840
Memòria	Cap de projecte	85	1700
Disseny del joc	Dissenyador	40	680
Direcció de l'equip	Cap de projecte	20	400
Publicitat i venda	Productor	70	1120
Disseny història	Escriptor	40	560
Recerca i desenvolupament	Programador	90	1530
TOTAL		515	8370

Pel que fa els costos de maquinària, els únics costos són els del desgast de la màquina. Tot el software emprat és gratuït.

Per calcular el desgast, hem suposat que la amortització és d'un 26%, durant uns 8 mesos.

Element	Cost inicial	Cost final
Ordinador	800	138

L'hipotètic cost total per aquest projecte seria de 8508 €

3. Metodologia

Per a la realització del projecte no s'ha seguit una metodologia de treball estàndard, sinó que s'ha triat i usat una metodologia personalitzada adient al projecte a desenvolupar. Els passos d'aquesta metodologia són els següents:

0. Triar el treball a realitzar.

1. Decidir el llenguatge de programació i eines a utilitzar.

2. Aprendre el llenguatge de programació i les eines escollides.

3. Estructurar el treball en parts segons les funcions que s'han de realitzar.

4. Desenvolupar la part corresponent seguint l'ordre de l'estructura del treball.

5. Fer les comprovacions per confirmar que el funcionament és correcte al acabar cada part.

- Si al fer les comprovacions el resultats no és el desitjat, es tornarà al punt 4 per a realitzar els canvis oportuns a la última part desenvolupada o a les anteriors, si així és necessari.

- Si al fer les comprovacions el resultat és el desitjat, es desenvoluparà la següent part tornat al punt 4. Una vegada que s'hagin finalitzar totes les parts amb les respectives comprovacions, s'iniciarà el punt 6.

6. Unir totes les parts desenvolupades i comprovar que el funcionament és correcte.

- Si al fer les comprovacions el resultat no és el desitjat, es tornarà al punt 4 per realitzar els canvis oportuns en la última part desenvolupada o en les anterior, si així és necessari.

- Si al realitzar les comprovacions el resultat és l'esperat, s'iniciarà el punt 7.

7. Generar diferents models d'exemple per a comprovar que el funcionament és el correcte.

- Si al fer les comprovacions el resultat no és el desitjat, es tornarà al punt 4 per a realitzar els canvis oportuns a la última part desenvolupada o en les anteriors si així és convenient.

- Si al realitzar les comprovacions el resultat és l'esperat, s'iniciarà el punt 8.

8. Arrodonir la documentació desenvolupada al llarg del projecte

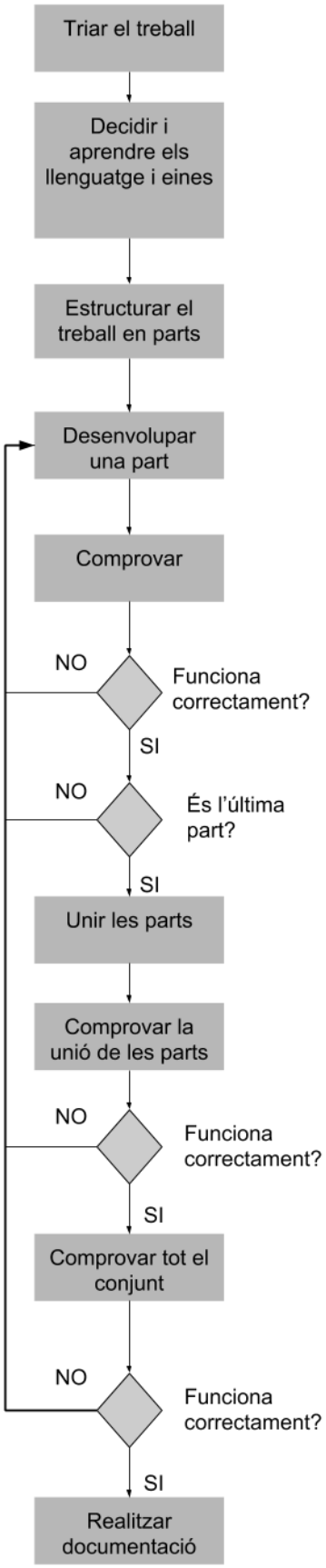

Triar el treball
Decidir i aprendre els llenguatge i eines
Estructurar el treball en parts
Desenvolupar una part
Comprovar
NO
Funciona correctament?
SI
NO
És l'última part?
SI
Unir les parts
Comprovar la unió de les parts
NO
Funciona correctament?
SI
Comprovar tot el conjunt
NO
Funciona correctament?
SI
Realitzar documentació

Figura 3.1: Metodologia

4. Planificació

Aquets projecte va començar a ser elaborat a partir del mes de Febrer de 2018, tot i que mesos abans s'hi van dedicar algunes hores per tal de començar a pensar la idea del joc, i veure alguns exemples per començar a iniciar-se al llenguatge i al motor. Es van realitzar quedades amb el tutor per tal d'anar pulint alguns detalls i alguns dubtes que anaven sorgint.

4.1 Pla de treball

Tal i com he comentat en la metodologia, per tal de realitzar aquest projecte la feina s'ha separat en parts, que exposaré en el següent punt.

4.2 Tasques planificades

Definició del joc

Es decideix de què tracta el joc; condicions per guanyar o perdre, tipus de joc, regles, interaccions del jugador i els elements que hi haurà.

Estudi i proves amb el llenguatge C#

Es comencen a fer petites proves de codi amb el llenguatge C#, de cara a agafar soltura amb el llenguatge i iniciar-se amb el llenguatge.

Estudi i proves amb el motor Unity

S'utilitza un projecte ja fet de Unity per tal de veure i entendre com funciona. En aquest projecte es presenta un joc molt senzill amb el qual es poden fer modificacions i es pot interectuar amb l'editor per tal d'agafar soltura.

Disseny i implementació d'algoritmes de camins i models

En un petit projecte de C#, s'implementa l'algoritme de Dijkstra i els models dels objectes que seran necessaris. Això ho vaig fer així per tal de poder fer-hi proves de manera més ràpida.

Dissenyar i implementar la base del joc amb Unity

Havent vist el joc d'exemple, s'inicia el projecte en Unity, creant inicialment el tauler, amb elements gràfics senzills per tal de veure que s'està fent bé.

Implementar les primeres interaccions de l'usuari

En aquesta tasca es crea el cursor, un element que serveix per mourens pel tauler i que més endevant servirà per interaccionar amb altres elements.

Integració dels algoritmes i models

Amb l'algoritme de camins i els models fets i testejats, s'integren i s'adapten en el projecte de Unity.

Búsqueda i preparació dels elements gràfics

Es cerquen tots els sprites necessaris per animar el joc: les caselles, les seleccions... I si fa falta s'animen, com és el cas de les unitats (les animacions són cadenes de sprites o imatges en successió).

Integració dels elements gràfics

Un cop preparats aquest elements en l'anterior tasca, s'integren en el projecte de Unity i s'assignen als seus respectius elements.

Implementar més interaccions de l'usuari

En aquesta tasca s'afegeixen més opcions per part de l'usuari. Ara es podrà seleccionar a les unitats del jugador i moure-les en el tauler, interaccionar amb les unitats enemigues i obrir el menú de opcions.

Disseny i implementació del combat entre unitats

Es desenvolupa la interacció entre unitats: la pèrdua de punts de vida de les dues unitats i si fa falta, la pèrdua d'alguna de les dues unitats.

Gestió dels torns entre jugadors

S'implementa la lògica del joc per torns: es deshabiliten les accions de l'usuari mentre no sigui el seu torn.

Disseny i implementació de la IA del jugador

Es crea una intel·ligència artificial simple per tal de que el jugador pugui jugar contra un adversari.

Búsqueda i integració de música i efectes de so

Es fa una petita recerca i s'escullen algunes músiques i efectes de so.

Demo i comprovacions finals

Es comprova que la jugabilitat sigui tal i com s'esperava al principi del projecte. Es fan vàries proves més per tal d'assegurar que la qualitat del projecte és òptima i compleix el que es volia.

Documentació

Finalment, s'ha escrit aquest document, documentant el software utilitzat, la metodologia emprada i altres aspectes relacionats amb el projecte.

4.3 Temps estimat i final

El projecte estava planejat per començar a ser desenvolupat a partir del Febrer i per acabar al Juny, tot i que la decicsió de fer un videojoc i començar a definir-lo es va començar uns mesos abans. Tot i això, considerarem l'inici al Febrer, ja que és llavors quan es planteja el projecte i s'hi comença a dedicar més temps.

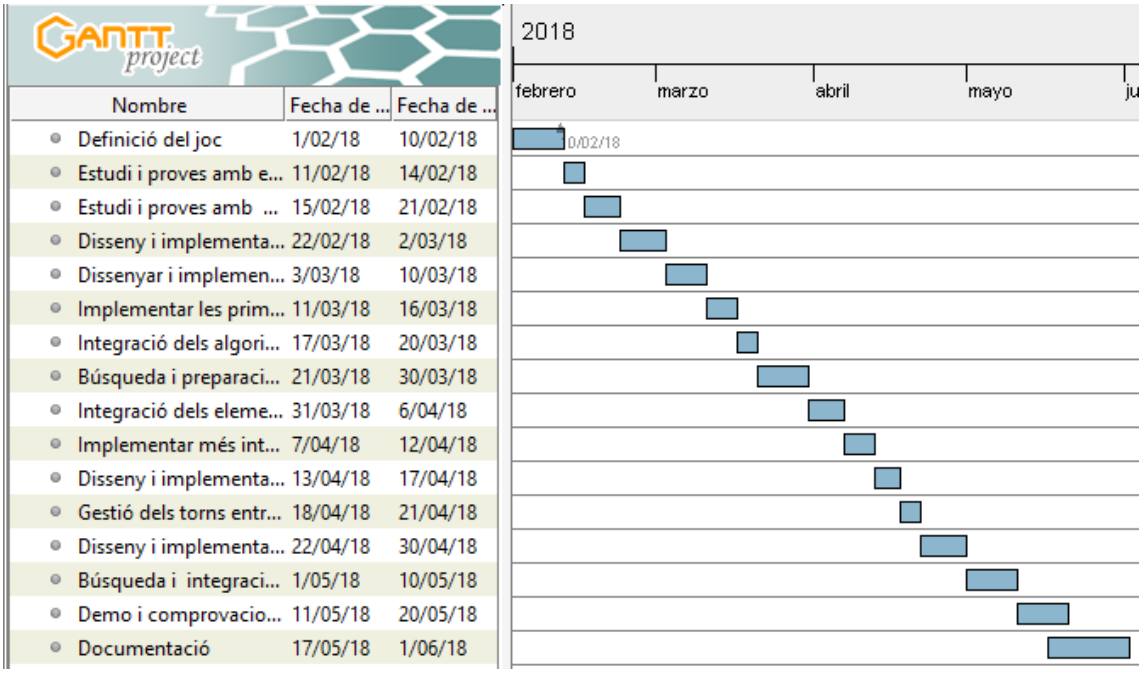


Figura 4.1 : Planificació inicial

Tot i això, el temps final varia molt, a causa de múltiples factors i s'acaba escollint Setembre com la data d'entrega. A continuació es pot veure el temps final:

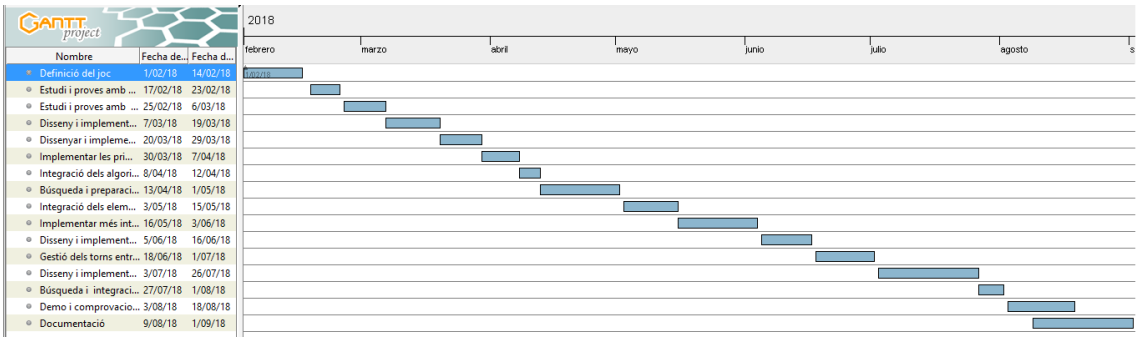


Figura 4.2 : Planificació final

4.4 Resultats esperats

S'espera desenvolupar un videojoc completament funcional, entretingut per l'usuari, amb les idees inicials definides, i fet de tal manera que si es vol afegir més contingut es pugui fer sense moltes complicacions.

5. Marc de treball i conceptes previs

En aquest capítol es respondrà a la pregunta de què és un motor de videojoc, es comentaran alguns motors i les seves característiques i finalment s'escollirà un d'ells i es justificarà les raons per les quals s'ha escollit.

5.1. Introducció als motors de videojocs

El motor de videojoc, també conegut com a Game Engine, és l'element principal per desenvolupar un joc. És el software que utilitzen els desenvolupadors per crear jocs de manera ràpida i eficient, ja que serveix de capa abstracta per tractar les llibreries gràfiques del sistema operatiu. Aquest element, també considerat estructura, agrupa les àrees bàsiques de importació d'art i assets, l'ensamblatge d'aquestes, la il·luminació, el so, els efectes, el motor de física, etc.

Utilitzar un bon motor gràfic permet facilitar el desenvolupament del joc , ja que allibera molta càrrega de feina que ja està feta pel propi motor. També és important destacar que un motor gràfic es pot comercialitzar, i pot generar ingressos al vendre'n llicències obrint així noves oportunitats de mercat.

En la figura 5.1 es poden apreciar els diferents camps que formen part del motor:

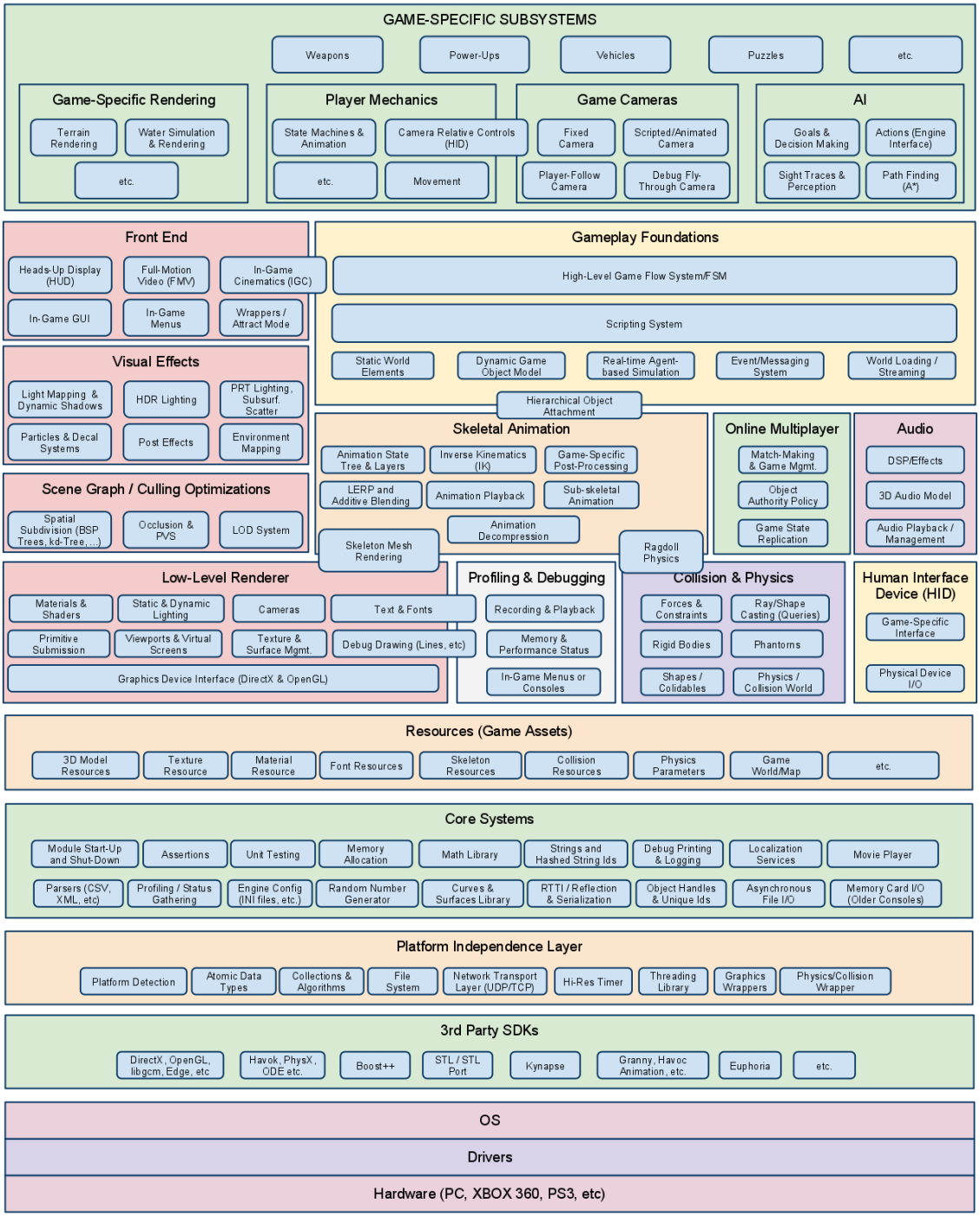


Figura 5.1 : Elements d'un motor gràfic

5.2. Motors de videojocs

Cry engine

Cry Engine és un motor de videojocs dissenyat per l'empresa Crytek, i és conegut per ser utilitzat en els jocs de la franquícia Far Cry o Crysis, entre molts d'altres. Accepta el llenguatge C++ i té suport per les plataformes de Windows, Linux, PlayStation (3 i 4), Wii U i Xbox (360 i One). Està orientat per jocs en 3D.

Exemples que utilitzen aquest motor: Far Cry 4 (Veure Figura 5.2) i Crysis (Veure Figura 5.3)

Figura 5.2 : Gameplay de Far Cry 4

Figura 5.3 : Gameplay de Crysis

Unreal engine

Unreal Engine és un motor de videojocs dissenyat per Epic Games. És un motor molt conegut i molt utilitzat per desenvolupadors, fàcil de fer servir i permet a usuaris sense coneixements poder desenvolupar videojocs, sense necessitat d'escriure codi. Accepta el llenguatge C++ i dóna suport a un gran nombre de plataformes: Windows, macOS, Linux, iOS, Android, Nintendo Switch, PlayStation 4, Xbox One i altres de realitat virtual.

Exemples de jocs que utilitzen aquest motor: Outlast (Veure figura 5.4) i QUBE (veure figura 5.5)

Figura 5.4 : Gameplay de Outlast

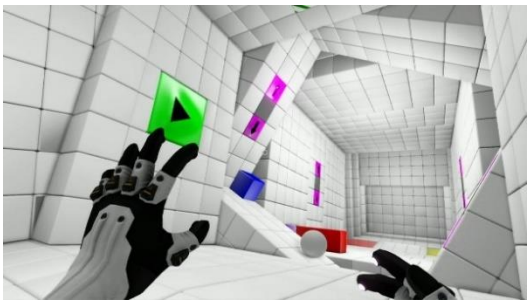


Figura 5.5 : Gameplay de QUBE

Unity

Unity és un motor de videojocs, propietat de Unity Technologies, que permet desenvolupar jocs per moltes plataformes diferents: Windows, Mac, Linux, Xbox, 3DS, PlayStation, Switch, etc. Inicialment acceptava varis llenguatges, com el C#, JavaScript i Boo, però actualment només accepta C#. Està orientat tant per jocs en 3D com per 2D. Té una llicència d'ús gratuïta, tot i que també en té de pagament, amb més serveis disponibles.

Exemples de jocs desenvolupats amb aquest motor: Hollow knight (veure figura 5.6) i Jotun (veure figura 5.7)

Figura 5.6 : Gameplay de Hollow knight

Figura 5.7 : Gameplay de Jotun

5.2. Elecció de motor

Per escollir un motor per aquest projecte, m'he basat en que fós compatible amb 2D, que fós senzill i fàcil d'entendre, i que hi hagués una comunitat amb experiència a darrere. Per aquests motius, he escollit Unity: és de fàcil accés i aprenentatge, la comunitat està molt extesa i hi ha gran varietat de tutorials i/o dubtes resolts. El llenguatge C# és desconegut per mi, però té moltes semblances amb C, C++ i Java. A més, dóna suport per diferents plataformes, un punt que tractarem en el capítol 12, Treballs futurs.

6. Requisits del sistema

A continuació s'explicaran els requeriments funcionals i no funcionals que haura de complir el joc.

6.1. Requeriments funcionals

Els requeriments funcionals són els que determinen què podrà fer i què no podrà fer l'usuari.

- L'usuari jugarà la partida contra una IA. Cada jugador disposarà d'un nombre d'unitats, i cadascún només podrà controlar les seves.
- El jugador podrà moure les unitats en el mapa, i podrà atacar només a les unitats del contrincant.
- Les unitats seràn d'una classe en concret, que en determinarà les estadístiques, com per exemple el moviment i el rang d'atac.
- El mapa serà una matriu de 2 dimensions, i cada cel·la serà d'un determiant tipus de terreny. Aquest terreny aportarà modificadors en les estadístiques de les unitats i també restringirà el moviment.
- El jugador es podrà desplaçar en el mapa utilitzant les fletxes de direcció, la tecla A per seleccionar i la S per cancelar.
- El primer jugador que es quedi sense unitats perdrà la partida, i l'altre guanyarà.

6.2. Requeriments no funcionals

Els requeriments no funcionals, a diferència del funcionals, fan referència a propietats com la fiabilitat, el temps de resposta, l'ús de memòria del dispositiu, la seguretat, etc.

Aquest joc s'ha desenvolupat mitjançant el següent hardware:

- CPU : AMD Ryzen 5 1600 3.2GHZ
- GPU : Nvidia Geforce GTX 1050Ti D5 4GB GDDR5
- Memòria: 16 GB
- SO : Windows 10 Home Edition

Però s'ha pogut executar amb un equip com el següent:

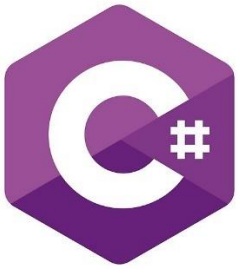
- CPU : Intel Core i5
- GPU : Nvidia Geforce 610
- Memòria: 8 GB
- SO : Windows 7 Home Edition

En quant a seguretat, en cap moment es guarden dades confidencials, i el joc es pot executar en mode usuari. Altres restriccions estarien relacionades amb el motor del joc, en aquest cas Unity.

7. Estudis i decisions

En aquest capítol explicaré tots els programes o software utilitzat en aquest projecte. En el cas del motor Unity, també faré esmena de les llibreries i elements que el composen, ja que pot aportar molta informació i pot facilitar entendre alguns conceptes d'aquesta memòria.

7.1 Llenguatge C#



C Sharp és un llenguatge de programació orientat a objectes desenvolupat per Microsoft. Deriva dels llenguatges C i C++ (utilitzats en nombroses assignatures durant el grau) i té certa semblança amb Java (també utilitzat).

S'ha utilitzat a causa de que Unity només dóna suport a aquest llenguatge, i també per utilitzar un llenguatge que no coneixia. L'aprenentatge no ha estat molt complicat, ja que, com he comentat abans, aquest llenguatge té moltes semblances amb altres llenguatges que ja s'han utilitzat. Tot el codi del projecte està en C#

7.2 Visual Studio

Visual Studio és un entorn integrat de desenvolupament (també conegut com a IDE en anglès: integrated development environment) propietat de Microsoft. Pot ser executat en Windows, Linux i Mac i dóna suport a molts llenguatges de programació.

S'ha utilitzat per programar tot el codi del joc.

7.3 Microsoft Word

Microsoft Word és un programa per processar texts, de la companyia Microsoft. Va ser desenvolupat l'any 1983 i ha esdevingut un dels processadors de text més utilitzat del món. S'ha utilitzat en aquest projecte (es disposava d'una llicència gratuïta com a estudiant) per redactar la memòria i altres documents relacionats amb el projecte.

7.4 Gantt Project

Gantt Project és un programa gratuït (sota la llicènca GPL) utilitzat per generar diagrames de Gantt (gràfiques que mostren la cronologia de les tasques que s'han de fer i el temps que s'hi dedicarà).

S'ha utilitzat per generar les figures 4.1 i 4.2 del capítol 4, Planificació.

7.5 GIMP

GIMP és un programa de tractament d'imatges distribuït sota la llicència GPL.

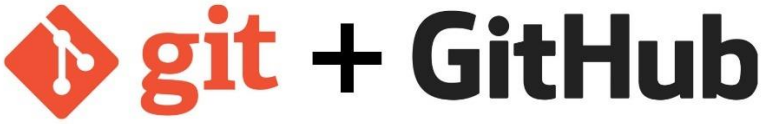
S'ha utilitzat per editar els sprites del projecte, per tal de poder ser tractats després amb Unity.

7.6 Trello

Trello és un programa d'administració de projectes. S'hi pot accedir tant per vía web i també com a aplicació per a mòbil. Destaca per la seva simplesa a l'hora de crear tasques (o cartes) i per mantenir-ne la traça i l'estat.

En aquest projecte s'ha utilitzat per decidir i controlar les tasques que s'havien de fer. Moltes d'aquestes tasques es poden trobar en el capítol 4, Planificació.

7.7 Git + GitHub



Git és un programa de control de versions, molt utilitzat arreu del món en projectes informàtics. Permet accedir a versions passades dels arxius i veure els canvis que s'hi han fet. És realment útil i indispensable si es treballa en grup i s'ha de modificar el projecte simultàniament.

En aquest projecte s'ha utilitzat juntament amb un repositori privat a GitHub (s'ha obtingut gratuïtament per ser estudiant) i amb els objectius de poder desenvolupar sense perdre canvis i tenir sempre accés al codi del pojecte.

7.8 Unity

Tal i com hem comentat en el capítol 5, Marc de treball i conceptes previs, Unity és un motor gràfic per desenvolupar videojocs en 2D o 3D.

L'editor de Unity és molt complet, i ens permet fer moltes accions diferents, ja sigui gestionar les escenes, crear animacions, modifacer sons ... A continuació explicarem com gestiona Unity el joc, i les seves parts.

Els jocs desenvolupats en Unity es divideixen en escenes. Cada escena és com si fós una pàgina, on li assignem diferents elements, ja sigui components de la interfície d'usuari (en anglès UI, *user interface*), il·luminació, el nivell del joc, etc. directament desde l'editor, sense necessitat de programar (inicialment) res i els hi podrem modificar la posició, el text (si en tenen), color, etc.

7.8.1 Entorn visual de l'editor

En la següent figura (Figura 7.1) es pot veure l'entorn visual de l'editor de Unity, i algunes de les parts que ofereix (comentaré les més utilitzades a l'hora de desenvolupar el projecte).

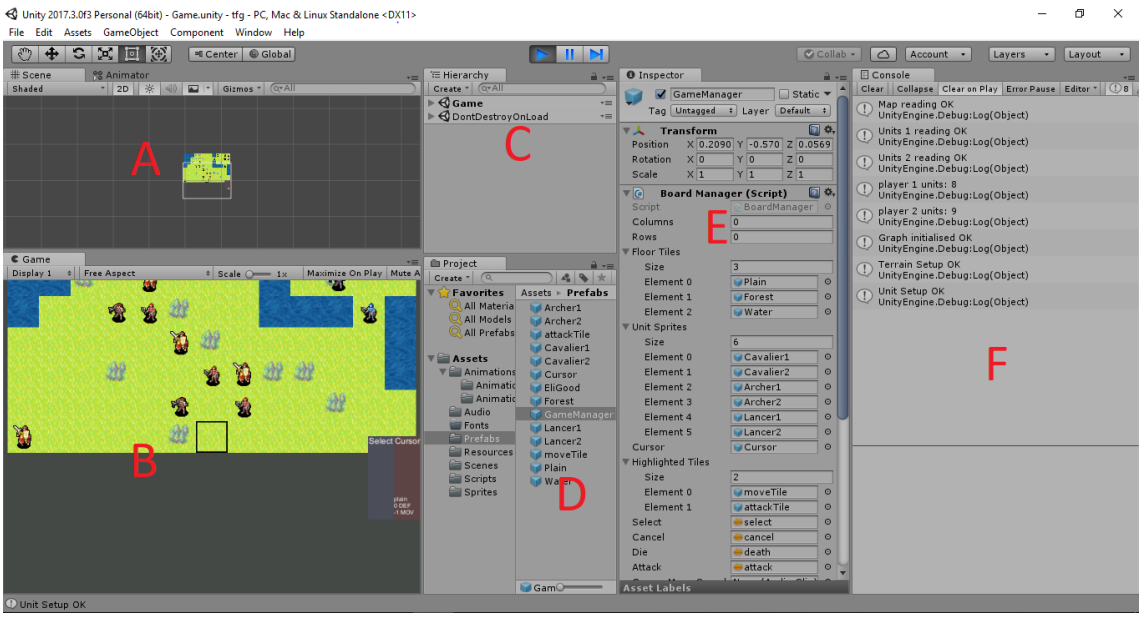


Figura 7.1 Entorn visual de l'editor deUnity

A. Vista actual de l'escena: aquí podem interactuar amb els elements de l'escena, que gestionem en la part C. Es poden editar els elements inclús quan s'està executant l'escena, i en la part B es veuran els canvis en temps real.

B. Vista prèvia de com es veurà el joc: en aquesta part veurem el que es veu en pantalla quan tenim el joc executan-se.

C. Jerarquia de l'escena seleccionada: Aqui veurem l'escena sel·leccionada o les diferents escenes, juntament amb els seus respectius elements. L'escena en negreta és l'escena sel·leccionada.

D. Estrucutra de carpetes del projecte i els seus elements: podem veure com tenim organitzat el joc, i podem sel·leccionar l'arxiu per poder veure les seves característiques i components en la part E

E. Inspector per mostrar les caracteristiques de l'element seleccionat: Aquí podem gestionar els components de l'element (podem afegir-ne, eliminar-ne i modificar-los) així com els seus atributs.

F. Consola de missatges: en aquesta consola apareixen els logs informatius, escrits pel propi Unity o bé també poden ser posats per nosaltres mateixos.

Hi ha un altre element, l'Animador, que serveix per gestionar les animacions dels elements. Podeu veure'l a la Figura 7.2 .

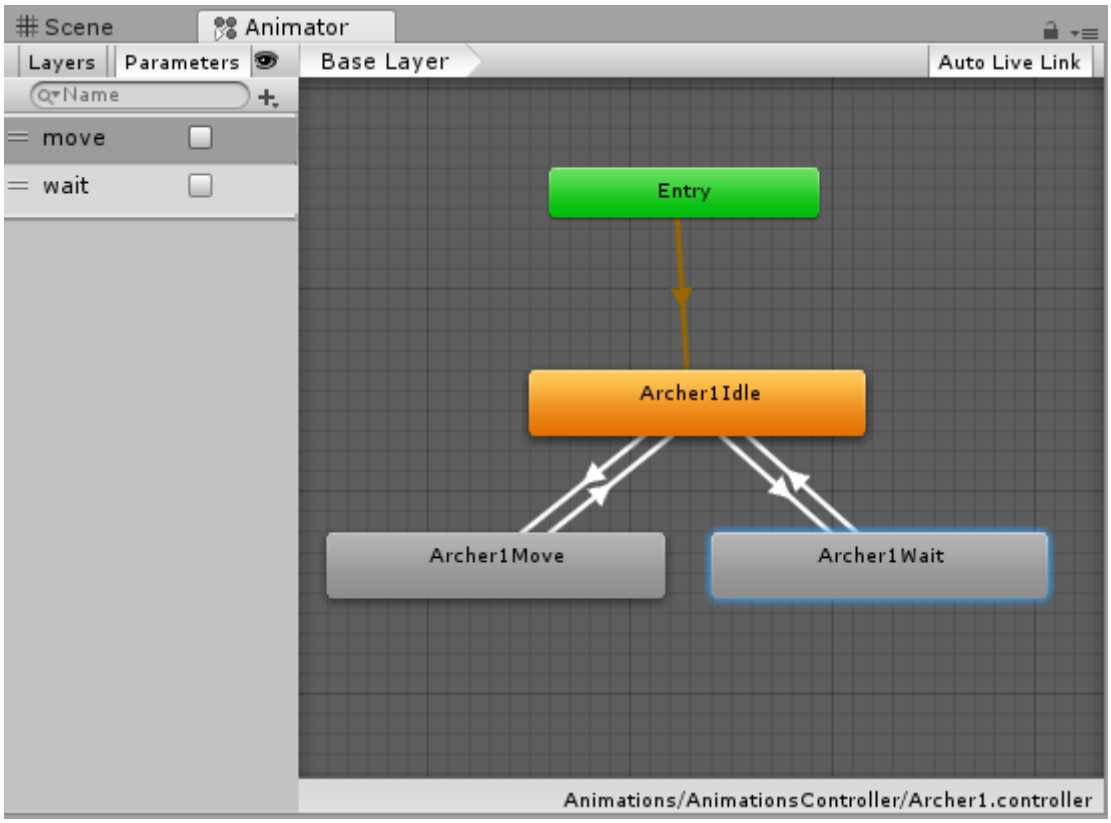


Figura 7.2 Interfície del Animator

En el Animator podem utilitzar variables, ja siguin toggles, booleans, etc. per gestionar quina animació s'ha d'executar.

7.8.2 Components i conceptes rellevants

En aquest apartat definirem breument les components més utilitzades i alguns conceptes per tal de comprendre millor aquest document.

També comentarem els mètodes més utilitzats de les components, que pertanyen a la API de Unity.

Sprite renderer

El sprite renderer és una component que ens permet gestionar el sprite d'aquell element (Veure Figura 7.3). Algunes de les opcions més importants són les d'assignar un sprite i definir-ne la Sorting layer.

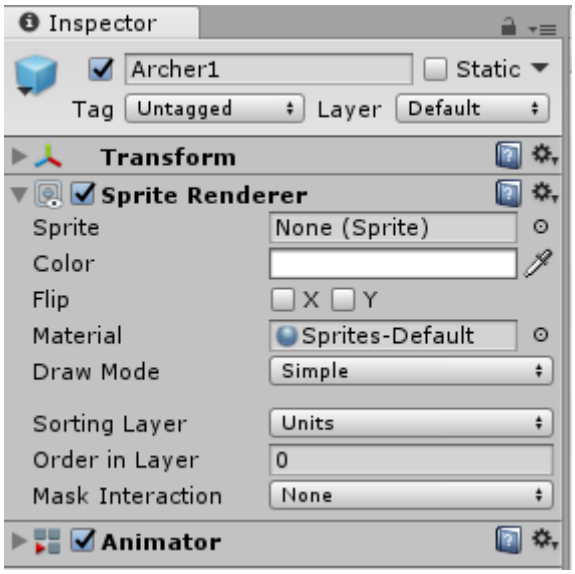


Figura 7.3 Component Sprite Renderer

Input

La classe Input ens serveix per detectar si es prem alguna tecla.

Els mètodes qe hem utilitzat de la API són:

- Bool GetKey (keyCode key): avalua si s'ha polsat la tecla key.
- Bool GetKeyDown(keyCode key): avalua si s'ha polsat la tecla, però només en el primer frame (si es manté polsada, avaluarà fals).

Gameobject

Es considera un GameObject qualsevol element dintre d'una escena. Un GameObject conté components, i quan es guarda s'obté el que es denomina com a *prefab.* És la classe bàsica per totes les entitats de Unity.

Sorting layer

Sorting Layer és una propietat dels elements utilitzada per definir la prioritat de vista en jocs de 2D. S'utilitza per definir si un element ha de quedar per sobre d'un altre o no. Per exemple, en el nostre projecte tenim definides les sorting Layers de la figura 7.4 , i s'interpreta que la primera és la que es queda més en sota i la última és la que està més a sobre.

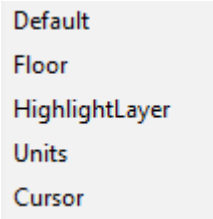


Figura 7.4 Sorting layers del projecte

Transform

Transform és una component que ens permet gestionar la posició de l'element , així com la seva mida (Veure Figura 7.5).

Els mètodes que hem utilitzat de la API són:

- Vector 3 position : retorna la posició de l'objecte
- Void SetParent : permet relacionar el component amb un GameObject

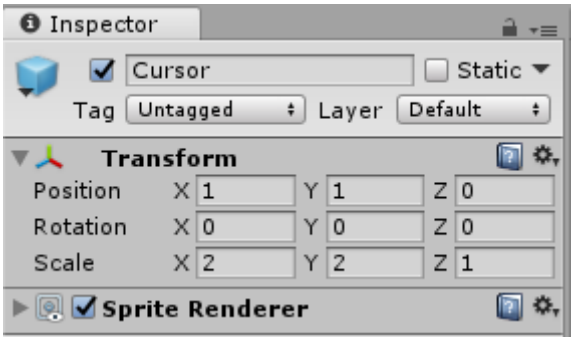


Figura 7.5 Component transform

Animator

Animator és una component que permet gestionar l'animació de l'element. Podem definir un controlador (com el que hem vist abans, en la Figura 7.2) i quan s'ha d'animar (Veure Figura 7.6).

Els mètodes utilitzats de la API són:

- GetBool(String name): obtenir el valor d'una variable de control anomenada name
- SetBool(String name, bool value): fem un set de value a la variable de control name

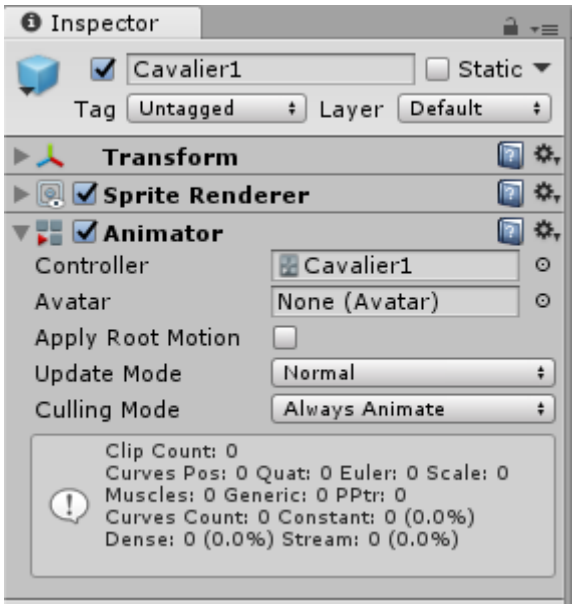


Figura 7.6 Component Animator

Audio source

Audio source és una component per gestioanr un so. S'entén com a so tant els efectes especials com les bandes sonores. En aquest component podem decidir, entre moltes opcions, a si s'ha de reproduïr en bucle, si s'ha d'executar al iniciar-se, etc. I també podem editar-ne les propietats, com el volum (Veure Figura 7.7).

Els mètodes utilitzats de la API són:

- Void Play(): per reproduïr audio.
- Void Stop(): per parar la reproducció.

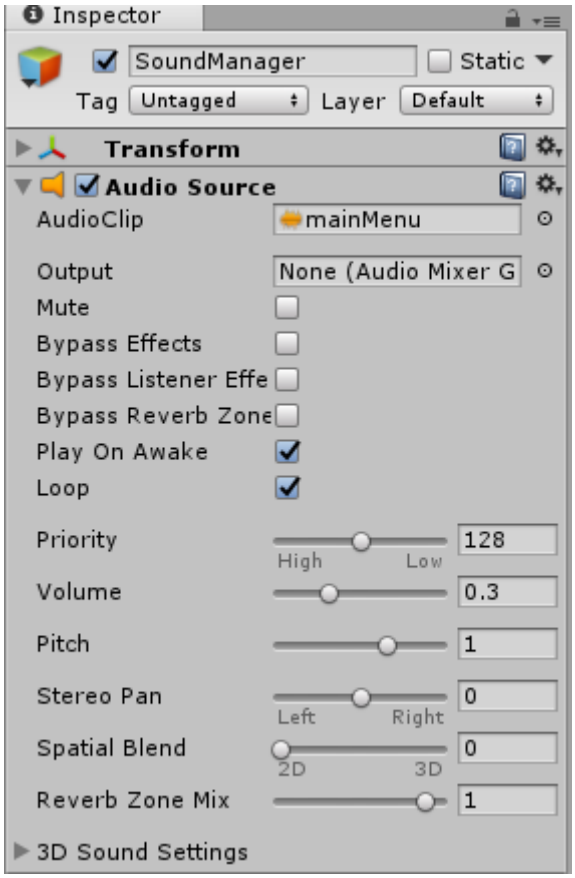


Figura 7.7 Component AudioSource

Canvas i elements

El canvas, o llenç, és l'element bàsic per tal d'afegir elements de la interfície d'usuari. A dintre seu, li podem afegir altres components, com per exemple botons, text, toggles, etc.

En la següent figura, (Figura 7.8) podem veure el canvas del menú principal, que està compost per una imatge (la imatge de fons), un text (el títol) i un parell de botons, que conten text (els botons de Play i Exit).

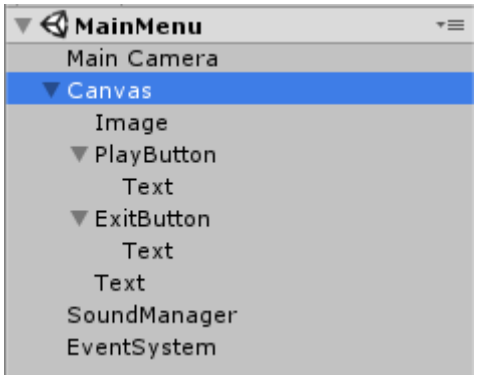


Figura 7.8 Jerarquia del canvas

Scene Manager

La classe SceneManager ens permet interactuar amb les escenes.

El mètode que hem utilitzat de la API és:

- Void LoadScene(int index): carrega l'escena identificada per index.

Cal fer una especial menció als components rigidbody2D i collider2D. Aquestes s'utilitzen per definir la massa i gestionar les col·lisions entre els elements respectivament, però en el nostre cas no ho hem utilitzat degut a que era més senzill gestionar-les mitjançant la matriu del mapa (també resultava més òptim).

7.9 Llibreries utilitzades

UnityEngine

Aquesta és la llibreria principal del motor de Unity. D'aquesta llibreria hem utilitzat altres llibreries, la UnityEngine.IU per gestionar els elements de la interfície d'usuari i la UnityEngine.SceneManagement, que permet gestionar les escenes.

System

La llibreria System és la llibreria bàsica de C#, que engloba les classes i mètodes més bàsics i més utilitzats. D'aquesta llibreria hem utilitzat també la de System.Collections per gestionar col·leccions (llistes i taules) i la de System.Threading per poder veure les accions del jugador enemic.

8. Anàlisi i disseny del sistema

En aquest capítol explicarem com és el joc que s'ha desenvolupat i la temàtica que tracta, sense entrar en detalls en la implementació.

8.1 Descripció general

El videojoc que hem creat es basa en els jocs d'estratègia per torns, amb la càmara centrada a dalt, també coneguda coma *top-down*, la qual es desplaça a mesura que nosaltres ens desplacem pel mapa. Disposarem un grup d'unitats, i haurem d'utilitzar-les juntament amb el mapa per tal d'eliminar totes les unitats del jugador enemic. En les Figures 8.1 i 8.2 es poden veure jocs similars d'aquesta temàtica, i que han influenciat molt aquest projecte:

Figura 8.1 Fire Emblem 6

Figura 8.2 Advance Wars

8.2 Disseny del funcionament

El joc consisteix en utilitzar les unitats aliades (color blau) per tal d'eliminar les unitats enemigues (color vermell). El jugador podrà moure les unitats aliades en el mapa i atacar les enemigues, sempre respectant unes normes:

- Una unitat només pot realitzar un moviment i un atac en cada torn.
- Tant el moviment com l'atac tenen un rang, relacionat amb la classe de la unitat, i es veu afectat pel terreny del mapa.
- El torn del jugador acaba quan ja ha utilitzat totes les seves unitats. En el cas de l'enemic, també.
- Quan una unitat perd tots els seus punts de vida, mor i desapareix del mapa.

A continuació, explicarem els elements que trobem en la partida, mitjançant la figura 8.3.

Figura 8.3 Escenari de la partida

1. En aquest requadre tenim informació dels terrenys i de les unitats. El requadre blau ens indica les propietats de la unitat sel·lecionada (en el nostre cas el llancer) i del terreny que ocupa. El requdre vermell ens mostra les dades de la posició del cursor, en aquest cas el llancer enemic. Els diferents valors que hi apareixen indiquen la classe de la unitat (Lancer) els punts de vida de la unitat (HP), el poder d'atac (ATK), la defensa (DEF), el moviment (MOV) i el rang d'atac (RNG). A sota, segueix la informació del terreny: la defensa adicional que ofereix (DEF) i el cost de moviemnt per accedir-hi (MOV).

2. El mapa. Hi podem trobar les unitats dels dos jugadors i els terrenys.

3. L'area ressaltada. Aquesta àrea pot ser de color blau o vermell. El color blau ens indica les posicions en les que la unitat pot realitzar un moviment, mentre que el vermell ens indica el rang d'atac de la unitat sel·leccionada per realitzar un atac.

4. El cursor. El requadre de color negre indica on estem situats. Podem moure'l per tot el mapa, i utilitzar-lo per moure i/o atacar amb les unitats aliades, o bé per obtenir informació del terreny i les unitats.

Per més informació sobre els controls, unitats o terrenys, veure tema 14, Manual d'usuari i/o instal·lació.

8.3 Identifiació dels actors

En el nostre videojoc, com que no fem distinció d'usuaris, només tindrem un sol actor, el jugador, que interactuarà en tot moment amb el sistema (Veure Figura 8.4).

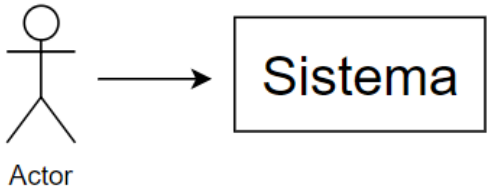


Figura 8.4 Identificació dels actors

8.4 Casos d'ús

En aquest apartat descriurem totes les activitats i comportaments des del punt de vista de l'actor.

En la pantalla del menú principal, l'actor pot decidir si vol jugar o vol sortir del joc (Veure Figura 8.5).

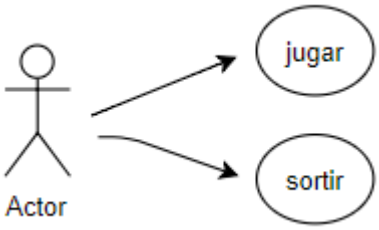


Figura 8.5 Diagrama menú principal

En la pantalla de la partida, el jugador podrà utilitzar el cursor, que li permetrà jugar i el Sistema s'encarregarà de gestionar aquestes accions (Veure Figura 8.6).

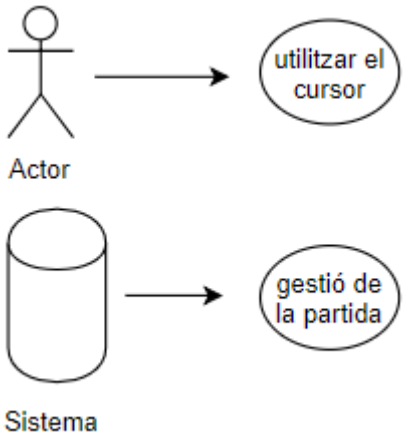


Figura 8.6 Diagrama partida

En la pantalla final, tant si el jugador ha guanyat com si ha perdut, només es podrà sortir del joc (Veure Figura 8.7).

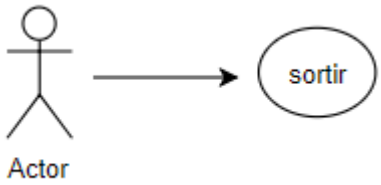


Figura 8.7 DIagrama final de la partida

8.5 Fitxes de casos d'ús

En aquest apartat descriurem més formalment tots els casos d'ús de l'apartat anterior.

Fitxa de cas d'us: Iniciar la partida	
Descripció	En el menú principal, l'actor selecciona 'Play'
Actor	Jugador
Precondició	El joc s'ha iniciat i el menú principal està present
Postcondició	S'ha iniciat la partida
Flux principal	1- Escollir 'Play' en el menú principal
Flux alternatiu	Cap
Observacions	Cap

Fitxa de cas d'us: Sortir del joc	
Descripció	En el menú principal, l'actor selecciona 'Exit'
Actor	Jugador
Precondició	El joc s'ha iniciat i el menú principal està present
Postcondició	El joc s'ha tancat
Flux principal	1- Escollir 'Exit en el menú principal
Flux alternatiu	Cap
Observacions	Cap

Fitxa de cas d'us: Utilitzar el cursor	
Descripció	El jugador utilitza el cursor per interactuar amb el mapa i els seus elements.
Actor	Jugador
Precondició	Partida iniciada
Postcondició	La unitat ja no pot ser utilitzada en aquest torn
Flux principal	1- El jugador mou el cursor per obtenir informació dels terrenys i les unitats. 2- Sel·leccionarà una unitat aliada, i es pintaran de color blau les caselles en les que pot realitzar un moviment. 3- Es sel·lecionarà una casella i la unitat es desplaçarà. Es pintaran de color vermell les caselles en les que pot atacar la unitat. 4- Es sel·leccionarà una casella amb una unitat enemiga per atacar o bé la pròpia casella de la unitat.
Flux alternatiu	Cap
Observacions	Cap

Fitxa de cas d'us: Gestió de la partida	
Descripció	S'inicialitza la partida
Actor	Sistema
Precondició	S'ha iniciat la partida
Postcondició	La partida s'ha iniciat correctament
Flux principal	1- Es carreguen els arxius necessaris i es contrueix la partida
Flux alternatiu	Cap
Observacions	Cap

Fitxa de cas d'us: Sortir del joc des de la pantalla final	
Descripció	El jugador ha guanyat o perdut la partida, i selecciona 'Exit'
Actor	Jugador
Precondició	La partida ha acabat is'ha carrgat la pantalla final
Postcondició	El joc es tanca
Flux principal	1- Escollir 'Exit en la pantalla final
Flux alternatiu	Cap
Observacions	Cap

8.6 Diagrames d'activitat

A continuació es troben els diagrames d'activitat, que són representacions gràfiques dels casos d'ús (en un diagrama hi poden haver múltiples casos).

En el diagrama del menú principal (Veure Figura 8.8) podem veure que es representen els casos d'ús de iniciar partida i sortir del joc, en el diagrama de la pantalla final (Veure Figura 8.9) es representa el cas d'ús de Sortir del joc des de la pantalla final i finalment en la Figura 8.10 podem veure el cas d'ús quan el jugador utilitza el cursor i el sistema ho gestiona.

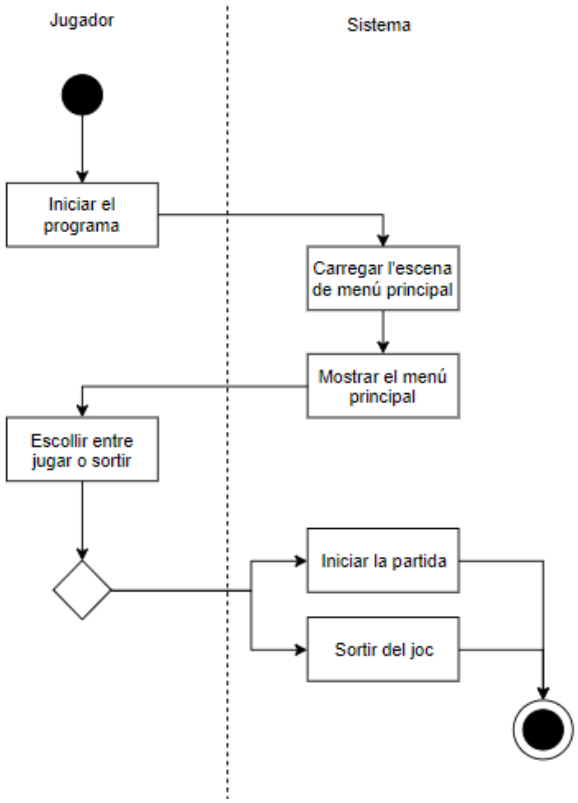


Figura 8.8 Digrama d'acivitat Menu principal

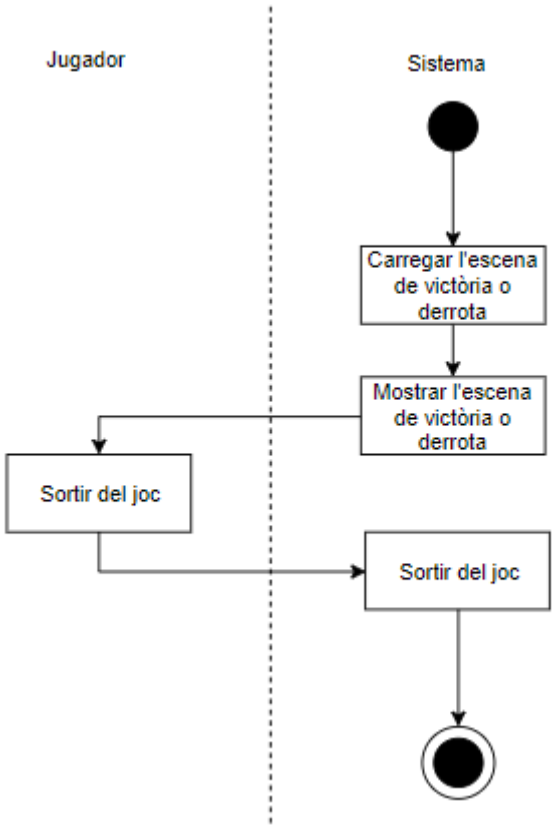


Figura 8.9 Diagrama d'activitat Pantalla final

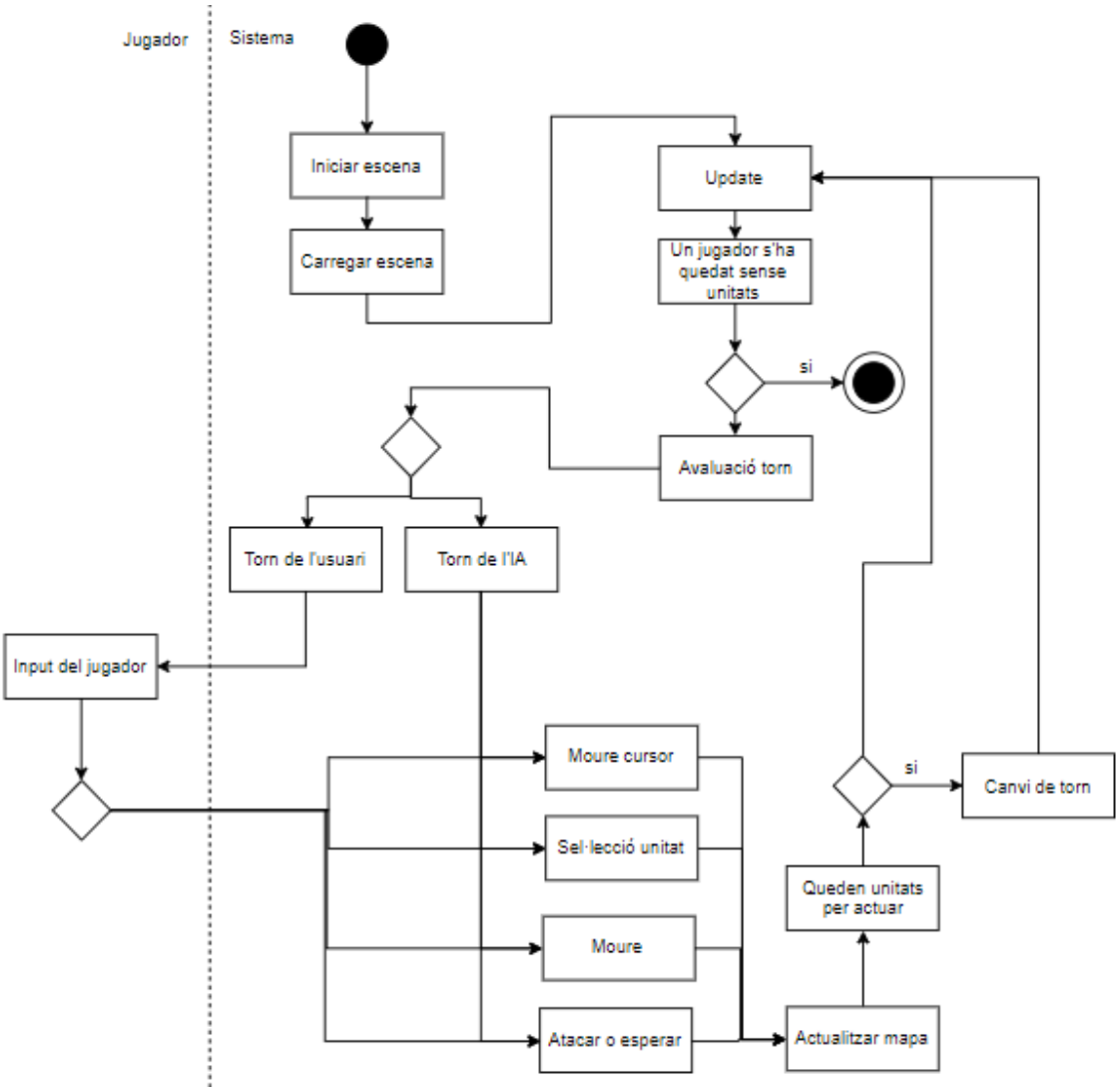


Figura 8.10 Diagrama d'activitat Partida

8.7 Classes i mètodes

En aquest apartat es mostrarà el diagrama de classes del projecte (Veure Figura 8.11) i després s'enumeraran les classes amb els seus atributs i els seus mètodes. Els mètodes que són constructors, setters, getters i que el seu ús és únicament per debugar s'ometran.

En el cas de les classes que no hem implementat, comentarem els mètodes més utilitzats.

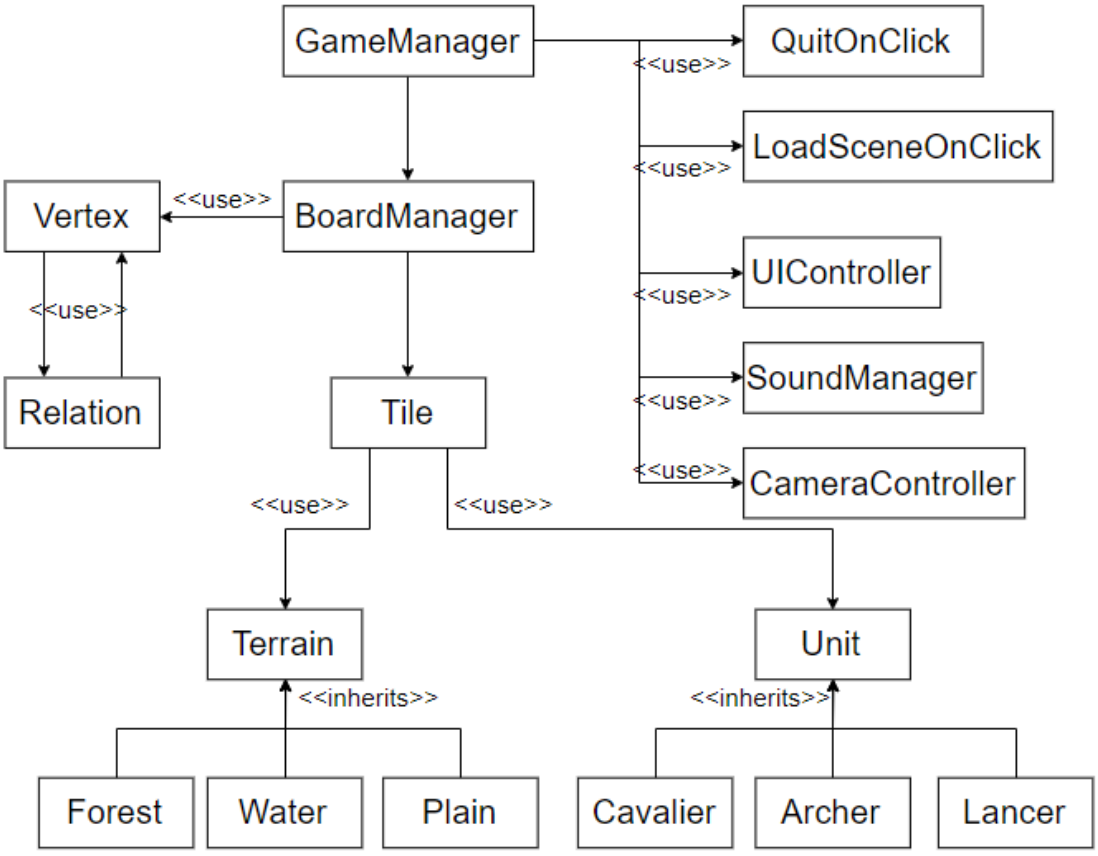


Figura 8.11 Diagrama de classes

Bàsicament, en aquest diagrama de classes d'aquest projecte podríem separar les classes entre les que no tenen cap relació amb Unity (de Vertex, Relation, Tile, Terrain i fills i Unit i fills) i les que sí. Això significa que, si es volgués utilitzar un motor diferent, es podria reutilitzar una gran part del codi.

Classe MonoBehaviour

Aquesta classe pertany a la llibreria de UnityEngine i tots els scripts relacionats amb Unity hereten d'aquesta classe. Les seves funcions més utilitzades són les següents:

- Awake(): mètode que es crida al carregar l'script.
- Start(): mètode que es crida al primer frame de l'escena.
- Update(): mètode que es crida en cada frame.
- LateUpdate(): mètode que es crida cada frame però sempre l'últim.

Classes Water, Plain i Forest

Aquestes 3 classes hereten de la classe Terrain. Només tenen un únic mètode, que és el constructor.

Classes Cavalier, Lancer i Archer

Aquestes 3 classes hereten de la classe Unit. Només tenen un únic mètode, que és el constructor.

Classe Terrain

La classe abstracta Terrain representa un tipus de terreny, i es la classe de la qual hereten les classes Water, Plain i Forest.

Els seus aributs són:

- Int Id : identificador del tipus de terreny.
- String Type: tipus de terreny.
- Int BonusDef: valor de defensa que aplica el terreny.
- Int MovRed: valor que determina quant costa passar per aquest terreny.
- Bool Heal: booleà que indica si aquest terreny cura o no.

Classe Unit

La classe abstarcta Unit representa una classe de unitat o soldat. D'aquesta classe hereten les classes Lancer, Archer i Cavalier.

Els seus atributs són:

- Int NextId: pròxim identificador únic.
- GameObject Sprite: sprite assignat a aquesta unitat.
- Int Id: identificador de la unitat.
- String class: classe de la unitat.
- Tile Position: posició de la unitat en el tauler.
- Int maxHP: punts de vida màxim de la unitat
- Int currentHP: punts de vida actuals de la unitat.
- Int ATK: poder o atac de la unitat.
- Int DEF: defensa o resistència de la unitat
- Int MOV: moviment o distància que pot recórrer la unitat.
- Int RNG: rang o distància d'atac de la unitat.
- Int Owner: propietari de la unitat.
- List<String> Advantage: llista de classes les quals aquesta unitat té bonificació quan s'enfronten.

Els seus mètodes són:

- Void recoverHP(): la unitat recupera punts de vida.
- Void loseHP(int dmg): la unitat perd dmg punts de vida.
- Bool isDead(): avalua si la unitat ja no té punts de vida.
- Void move(Tile t): actualitza la posició de la unitat a la tile T.
- Void wait(): s'accedeix al controlador de l'animació de la unitat i es posa en 'espera'.
- Bool canAct(): s'accedeix al controlador de l'animació de la unitat i s'avalua si està en 'espera'.
- Void move(): s'accedeix al controlador de l'animació de la unitat i es posa en 'moviment'.
- Void idle(): s'accedeix al controlador de l'animació de la unitat i es posa en 'apunt'.

- Int damageDealtTo(String enemy): avalua i retorna el dany que causarà aquesta unitat a la unitat enemy.

Classe Tile

La classe Tile conté la posició en el tauler, el Terrain i la Unit.

Els seus atributs són:

- Int X: component X del tauler.
- Int Y: component Y del tauler.
- Unit Unit: unitat que es troba en aquesta Tile.
- Terrain Terrain: terreny d'aquesta tile.

Classe Vertex

La classe Vertex representa un node en el tauler i s'utilitza per construïr l'esquelet de l'algoritme de camí mínim Dijkstra.

Els seus atributs són:

- Int X: component X del tauler.
- Int Y: component Y del tauler.
- List<Relation> nearBy: llista de Relation d'aquest Vertex amb altres.
- Int minDist: distància minima per arribar a aquest Vertex.
- Vertex previous: Vertex antecessor per arribar a aquest Vertex.

Classe Relation

La classe Relation representa una relació entre un Vertex i el seu cost per passar-hi.

Els seus atributs són:

- Vertex vertex: Vertex de la Relation.
- Int cost: cost o valor per passar per vertex.

Classe UIController

La classe UIController s'encarrega de gestionar la interfície d'usuari que apareix durant el transcurs de la partida.

Els seus atributs són:

- Text textElement: referència a l'element de text de la interfície que serà actualitzat.
- GameManager gameManager: referència al gameManager de la partida.
- String option: indica el contingut text que ha d'aparèixer.
- Tile tileCursor: Tile en la qual es troba el cursor.
- Tile tileSelected: Tile en la que es troba la unitat sel·leccionada.

El seus mètodes és:

- LateUpdate(): actualitza el text textElement amb el contingut que marca option.

Classe QuitOnClick

La classe QuitOnClick s'encarrega de tancar el joc.

El seu mètode és:

- Void Quit(): tanca l'aplicació.

Classe LoadSceneOnClick

La classe LoadSceneOnClick s'encarrega de carregar una escena.

El seu mètode és:

- LoadByIndex(int sceneIndex): carrega l'escena identificada per sceneIndex.

Classe CameraController

La classe CameraController s'encarrega d'actualitzar la posició de la càmara durant la partida.

El seu atribut és:

- GameManager gameManager: referència al GameManager de la partida.

El seu mètode és:

- LateUpdate(): actualitza la posició de la càmara a la del cursor.

Classe SoundManager

La classe SoundManager s'encarrega de gestionar la música i efectes de so durant el transcurs de la partida.

Els seus atributs són:

- AudioSource efxSource: audio que fa referència a un efecte de so.
- AudioSource musicSource: audio que fa referència a una música.
- SoundManager instance: instància a aquest objecte.

Els seus mètodes són:

- Awake(): crea una instància nova o destrueix-la si ja existia.
- PlaySfx(AudioClip clip): reproduexi l'efecte de so clip.
- PlauMusic(AudioClip clip): reprodueix la música clip.

Classe GameManager

La classe GameManager s'encarrega de gestionar tota la partida: els torns, les accions de l'usuari i les condicions de victòria o derrota.

Els seus atributs són:

- BoardManager boardScript: tauler en el que transcorre la partida.
- Bool playersTurn: indica si és el torn del jugador o no.
- String stage: string que s'utilitza per saber com interpretar l'input de l'usuari.
- Bool victory: indica si l'usuari ha guanyat la partida o no.
- Bool stillPlaying: indica si s'ha acabat la partida o encara no.

- Float time: mesura de temps per filtrar alguns inputs de l'usuari.
- AudioClip playerTurnMusic: referència a l'arxiu de so del torn del jugador.
- AudioClip enemyTurnMusic: referència a l'arxiu de so del torn del jugador enemic.
- AudioClip victoryMusic: referència a l'arxiu de so de victròria.
- AudioClip gameOverMusic: referència a l'arxiu de so de derrota.

Els seus mètodes són:

- Start(): carrega el mapa i inicia la partida.
- InitGame(): crea l'escena del mapa i i inicialitza el cursor.
- EndGame(): comprova si s'ha acabat la partida i canvia d'escena a la corresponent.
- Update(): comprova si la partida pot acabar i gestiona el canvi de torn.
- readInput(): segons el valor de stage, interpreta l'input de l'usuari.
- readInputInIdleMap(): llegeix l'input de l'usuari quan no té cap unitat sel·leccionada.
- readInputMoveUnit(): llegeix l'input de l'usuari quan ha sel·leccionat una unitat.
- readInputAttackUnit(): llegeix l'input de l'usuari quan ha desplaçat la unitat sel·leccionada.

Classe BoardManager

La classe boardManager s'encarrega de gestionar el mapa; la seva inicialització, modificació i també de la intel·ligència artificial del jugador enemic.

En aquest classe s'utilitza una estructura anomenada Cell, que conté els següents atributs:

- Tile tile: casella a la que fa referència.
- GameObject[] highlightLayers: per ressaltar una casella.
- Vertex vertex: s'utilitza per els càlculs de camins mínims

Els seus atributs són:

- int Columns: nombre de columnes del mapa.

- Int Rows: nombre de files del mapa
- Cell[,] Map: estructura del tauler
- GameObject[] floorTiles:
- GameObject[] unitSprites:
- List<Unit> player1Units: llista d'unitats del jugador 1 (usuari).
- List<Unit> player2Units: llista d'unitats del jugador 2 (enemic).
- GameObject cursor: cursor que s'utilitza per desplaçar-se pel tauler.
- GameObject[] highlightedTiles: llista de diferents tipus de ressaltament de caselles.
- Transform boardHolder: component de posició en l'escena.
- AudioClip select: referència a l'efecte de so quan es fa una acció de sel·lecció.
- AudioClip cancel: referència a l'efecte de so quan es cancel·la una acció.
- AudioClip die: referència a l'efecte de so quan una unitat és eliminada.
- AudioClip attack: referència a l'efecte de so quan es fa una acció d'atac.
- AudioClip cursorMoveSound: referència a l'efecte de so quan es mou el cursor.
- Unit unitSelected: referència a la unitat sel·leccionada.
- Tile startPosition: posició inicial de la unitat sel·leccionada.
- TextAsset mapFile: referència a l'arxiu de text que conté el mapa.
- TextAsset enemyUnits: referència a l'arxiu de text que conté les unitats enemigues i la seva posició.
- TextAsset allyUnits: referència a l'arxiu de text que conté les unitats aliades i la seva posició.
- Random random: llavor per generar nombres aleatoris.

Els seus mètodes són:

- Int terrainFactory(Tile t): retorn l'identifiacdor del tipus de Tile t.
- Terrain tileFactory(int t): retorna el tipus de Terrain identificat per t.

- Int unitFactory(Unit u): retorna l'identificador de la classe de u.
- Unit ClassFactory(int t): retorna una nova unitat de la classe identificada per t.
- Void initialiseCursor(): inicialitza l'atribut cursor i lliga'l a boardHolder.
- Void SetupScene(): llegeix els arxius de text i prepara el mapa i l'estructura per l'algoritme de camins mínim.
- Void readMapFile(): llegeix l'arxiu de text mapFile i prepara el mapa amb els terrenys.
- Void readUnitsFiles(): llegeix els arxius de text de enemyUnits i allyUnits i crea les unitats en el mapa.
- Void terrainSetup(): aplica els sprites als terrenys.
- Void unitSetup(): aplica els sprites a les unitats.
- initialiseGraph(): inicialitza el graf per calcular camins mínims.
- initialisePath(): inicialitza els vèrtexs per calcular camins mínims.
- Void moveCursor(int x, int y): mou el cursor en la direcció (x,y).
- Void moveCursorInsideArea(int x, int y): mou el cursor en la direcció (x,y) dintre de l'area ressaltada.
- moveCursor(Vector2 v): mou el cursor a la casella v.
- Void setUnit(Unit unit, Vector2 pos): mou la unitat unit a la casella pos.
- Int moveUnit(Unit unit, Vector2 ori, Vector2 dest): mou la unitat Unit desde la casella ori a la casella dest i retorna si és possible o no.
- Int moveSelectedUnit(): mou la unitat unitSelected a la posició del cursor.
- Void returnSelectedUnitToStartingTile(): mou la unitat unitSelected a la casella startPosition, juntament amb el cursor.
- selectTile(string action): sel·lecciona la unitat que es troba en la casella del cursor.
- attackOrWait(): si en la casella del cursor hi ha una unitat enemiga, realitza un atac de la unitat unitSelecte. Altrament, si en la casella del cursor hi ha la mateix unitat unitSelected, acaba el torn de la unitat.

- Void combat(Unit enemy): realitza un enfrontament entre la unitat unitSelected i la unitat enemy.
- Void freeUnitSelected(): allibera la unitat unitSelected.
- Void endUnitTurn(): acaba el torn de la unitat unitSelected i allibérala.
- Void enemyTurn(): gestiona el torn del jugador enemic mitjançant la següent intel·ligència artificial:

```
unitatSeleccionada = següentUnitatDisponible();
si (hiHaEnemicsDintreDelRang())
        victima = buscaLaMillorVictima();
        posicio = buscaLaMillorPosicioPerAtacar(victima);
        mouLaUnitat(posicio);
        ataca(victima);
altrament
        mouLaUnitatEnUnaPosicioALAtzar();
fsi
acabaElTornDeLaUnitat();
```

- Void startTurn(int player): inicialitza el torn del jugador player.
- Unit nextUnit Available(int player): retorna la primer unitat que pot actuar del jugador player.
- Void clearHighlightedTiles(): esborra el ressaltat de totes les caselles.
- Void endPlayerTurn(): acaba el torn del jugador.
- moveUnitToRandomPosition(List<Cell> list): mou la unitat unitSelected a una posició vàlida dintre de la llista list.
- Unit evaluateEnemiesAndReturnBest(HashSet<Cell> set): dels enemics de set, retorna la millor opció per que s'hi enfronti la unitat unitSelected.
- Tile searchBestCellToAttack(Tile objective, List<Cell> zone): sabent que hi ha una unitat enemiga a objective, de la llista zone es tria la més avantatjosa.

- Tile bestPosition(List<Cell> list): retorna la millor casella de la llista list per moure-hi la unitat unitSelected.
- Bool enemyDies(Unit u): avalua si en un enfrontament la unitat unitSelected eliminaria la unitat enemiga u.
- Int trueDamage(Unit u): retorna el dany final que causarà la unitat unitSelected a la unitat u (tinguent en compte els terrenys).
- Int trueCounterAttack(Unit u): retorna el danyde contratac final que rebrà la unitat unitSelected de la unitat u (tinguent en compte els terrenys).
- HashSet<Cell> enemisInRange(List<Cell> movement): retorna totes les caselles en les quals hi hagi enemics i que estiguin dintre del rang de moviment i atac de la unitat unitSelected.
- Void addNeighbours(int x, int y): afegeix al vertex situat a (x,y) les relacions entre els vertexs veïns (4 direccions).
- Void paintHighlightedTiles(Unit u, String s): ressalta, segons s, les caselles dintre del rang de la unitat u.
- Bool validPosition(Vector2 v): avalua si la posició o casella v està dintre del mapa.
- Bool validPositionInArea(Vector2 v): avalua si la posició o casella v està dintre del mapa i dintre de la area ressaltada.
- List<Cell> getCellsInRange(Unit u, String action): retorna les cel·les que es troben dintre del rang de la unitat u en funció de action.
- List<Cell> getCellsInAttackRange(int range, Tile pos): retorna les cel·les que es troben dintre del rang d'atac de la unitat situada a pos.
- List<Cell> getCellsInMovementRange(int range, Tile pos): retorna les cel·les que es troben dintre del rang de moviment de la unitat situada a pos.
- Bool isBetterWay(Vertex start, Relation nextTo, int limit): avalua si el camí actual és millor que el calculat anteriorment.
- Void deleteUnitFromPlayer(Unit u): elimina la unitat u.

- Bool canCounterattack(Unit enemy): avalua si la unitat unitSelected està dintre el rang d'atac de la unitat enemy.

9. Implementació i proves

En aquest apartat explicarem com s'han implementat les diferents parts del joc, i comentarem algunes parts del codi. Tot i això, aquest projecte serà entregat juntament amb el codi font, per tant es tindrà accés a tot el codi.

9.1 Escena Menú principal i final

S'ha pogut dissenyar sense pràcticament cap línia de codi. Aquestes escenes contenen un canvas, pels elements visuals, elements d'audio i botons que tenen un petit script, per tal de canviar d'escena.

```
    public void LoadByIndex(int sceneIndex)
    {
        SceneManager.LoadScene(sceneIndex);
    }
```

Les escenes són:

0 – Main menu

1 – Partida

2 – Victòria

3 – Derrota

9.2 Escena de la partida

Aquesta és l'escena en la que s'ha dedicat més temps, i és la més complexa de totes.

9.2.1 Interfície d'usuari

Utilitzem el mètode LateUpdate perquè es cridi després del Update(), i utilitzem la variable de control option per saber quines dades hem d'obtenir.

Aquest script forma part, com a component, de cada element de text.

```
        // LateUpdate is called after Update each frame
        void LateUpdate()
        {
            if (option == "selectedUnit")
            {
                if (gameManager.boardScript.unitSelected != null)
                {
                    tileSelected =
gameManager.boardScript.Map[gameManager.boardScript.unitSelected.getPosition()
.getX(), gameManager.boardScript.unitSelected.getPosition().getY()].tile;
                    setSelectedUnitInfo();
                }
                else
                {
                    this.textElement.text = "";
                }
            }
            else if (option == "selectedTerrain")
            {
                if (gameManager.boardScript.unitSelected != null)
                {
                    tileSelected =
gameManager.boardScript.Map[gameManager.boardScript.unitSelected.getPosition()
.getX(), gameManager.boardScript.unitSelected.getPosition().getY()].tile;
                    setSelectedTerrainInfo();
                }
                else
                {
                    this.textElement.text = "";
                }
            }
            else if (option == "cursorUnit")
            {
                tileCursor =
gameManager.boardScript.Map[(int)gameManager.boardScript.cursor.transform.posi
tion.x, (int)gameManager.boardScript.cursor.transform.position.y].tile;
                if (tileCursor.getUnit() != null)
                {
                    setCursorUnitInfo();
                }
                else
                {
                    this.textElement.text = "";
                }
            }
            else if (option == "cursorTerrain")
            {
                tileCursor =
gameManager.boardScript.Map[(int)gameManager.boardScript.cursor.transform.posi
tion.x, (int)gameManager.boardScript.cursor.transform.position.y].tile;
                setCursorTerrainInfo();
            }
            else
            {
                Debug.Log("Wrong option");
            }
        }
```

9.2.2 Camera

La camera és un element que es troba en cada escena. En l'escena principal i les finals és fixa, però durant la partida sempre es centra en el cursor, per tal de poder veure tot el mapa.

```
    // LateUpdate is called after Update each frame
    void LateUpdate()
    {
        transform.position = new
Vector3(gameManager.boardScript.cursor.transform.position.x,
gameManager.boardScript.cursor.transform.position.y, -10);
    }
```

9.2.3 Mapa

La gestió del mapa és la part més important del joc. A continuació detallarem les parts més importants i com s'han implementat, aportant talls de codi.

Construcció del tauler

Per construir i iniciar la classe BoardManager, ho hem separat en 3 parts: la primera tracta de llegir 3 documents de texts, després inicialitzar el graf i finalment animar els components amb sprites. A continuació veurem aquestes parts més en profunditat.

En el següent tall de codi podem trobar el mètode SetupScene(), que inicialitza el tauler:

```
    public void SetupScene()
    {
        readMapFile();
        readUnitsFiles();

        initialiseGraph();

        terrainSetup();
        unitSetup();
    }
```

I a continuació adjuntarem les funcions i comentarem com és que es van fer així.

L'arxiu del mapa, és un document de text que conté una matriu de valors entre el 0 i el 2. Aquests valors representen un terreny en concret, i ja tenim un mètode Factory que interpreta el el valor i retorna el terreny. Es va decidir fer així de cara a poder fer més proves, i permetre en un futur utilitzar diferents mapes, únicament guardant petits documents de text.

Inicialment s'utilitzava un FileReader, però ens vam trobar que quan es construïa l'executable, l'estructura del projecte canviava i no es podia trobar l'arxiu. Per aquest motiu, es va canviar el FileReader per un TextAsset, i simplement accedint al seu atribut text ja tenim el document llegit. Per últim, comentar que en aquuest mètode creem el tauler o mapa, i en cada cel·la assignem el terreny, el quadre de ressaltament pel moviment o l'atac i finalment el vertex.

El codi és el següent:

```
public void readMapFile()
    {
        String[] lines = this.mapFile.text.Split('\n');
        this.Columns = lines[1].Split(' ').Length;
        this.Rows = lines.Length;
        this.Map = new Cell[Columns, Rows];

        for (int y = 0; y < Rows; y++)
        {
            String[] linia = lines[Rows - 1 - y].Split(' ');    // Unity starts
the matrix (0,0) at left down instead of left up
            for (int x = 0; x < Columns; x++)
            {
                Map[x, y].tile = new Tile(x, y, tileFactory(int.Parse(linia[x])));

                Map[x, y].highlightLayers = new GameObject[2];
                Map[x, y].highlightLayers[0] = Instantiate(highlightedTiles[0],
new Vector2(x, y), Quaternion.identity);
                Map[x, y].highlightLayers[1] = Instantiate(highlightedTiles[1],
new Vector2(x, y), Quaternion.identity);
                Map[x, y].highlightLayers[0].SetActive(false);
                Map[x, y].highlightLayers[1].SetActive(false);

                Map[x, y].vertex = new Vertex(x, y);
            }
        }
        Debug.Log("Map reading OK");
    }
```

Per llegir les unitats, es fa servir exactament el mateix mètode i motiu. En el seu cas, les untiats es representaran per els valors de l'1 al 3 i després de crear-les se'ls hi assigna el propietari. A continuació es pot veure un tall del mètode readUnitsFile() per llegir les unitats del jugador 1. Per les de ljugador 2 ,l'enemic, es fa el mateix.

```
    public void readUnitsFiles()
    {
        String[] lines = this.allyUnits.text.Split('\n');

        this.player1Units = new List<Unit>();

        for (int y = 0; y < Rows; y++)
        {
            String[] linia = lines[Rows - 1 - y].Split(' ');    // Unity starts the
matrix (0,0) at left down instead of left up
            for (int x = 0; x < Columns; x++)
            {
                if (int.Parse(linia[x]) != 0)
                {
                    Map[x, y].tile.setUnit(ClassFactory(int.Parse(linia[x])));
                    Map[x, y].tile.getUnit().setOwner(1);
                    Map[x, y].tile.getUnit().setPosition(Map[x, y].tile);
                    this.player1Units.Add(Map[x, y].tile.getUnit());
                }
            }
        }
```

El mètode de initialiseGraph() crea les relacions entre els vèrtexs, i prepara tot el necessari. Quan creem les relacions, només enim en comptes les 4 posicions més pròximes (esquerra, dreta, amunt i avall).

Els mètodes de validPosition() i similars comproven que no es surti del mapa.

```
    void initialiseGraph()
    {
        for (int y = 0; y < Rows; y++)
        {
            for (int x = 0; x < Columns; x++)
            {
                addNeighbours(x, y);
            }
        }
        Debug.Log("Graph initialised OK");
    }


private void addNeighbours(int x, int y)
    {
        if (validPosition(new Vector2(x - 1, y)))
        {
            this.Map[x, y].vertex.addRelation(new Relation(this.Map[x - 1,
y].vertex, this.Map[x - 1, y].tile.getMoveRed()));
        }
        if (validPosition(new Vector2(x + 1, y)))
        {
            this.Map[x, y].vertex.addRelation(new Relation(this.Map[x + 1,
y].vertex, this.Map[x + 1, y].tile.getMoveRed()));
        }
        if (validPosition(new Vector2(x, y + 1)))
        {
            this.Map[x, y].vertex.addRelation(new Relation(this.Map[x, y +
1].vertex, this.Map[x, y + 1].tile.getMoveRed()));
        }
        if (validPosition(new Vector2(x, y - 1)))
        {
            this.Map[x, y].vertex.addRelation(new Relation(this.Map[x, y -
1].vertex, this.Map[x, y - 1].tile.getMoveRed()));
        }
    }
```

Finalment, només resta animar el terreny i les unitats. S'ha separat d'aquesta manera perquè sigui més entenedor i per separar els processos que es fan. En aquest procés es carrega el sprite corresponent al terreny i a la unitat (segons el propietari) i s'anima.

```
    // Initialise the grid and add the terrain
    void terrainSetup()
    {
        boardHolder = new GameObject("Board").transform;

        for (int y = 0; y < Rows; y++)
        {
            for (int x = 0; x < Columns; x++)
            {
                GameObject toInstantiate = floorTiles[terrainFactory(Map[x,
y].tile)];
                GameObject instance = Instantiate(toInstantiate, new Vector2(x,
y), Quaternion.identity) as GameObject;
                instance.transform.SetParent(boardHolder);
            }
        }
        Debug.Log("Terrain Setup OK");
    }

    // Put the units in the board
    void unitSetup()
    {
        foreach (Unit u in player1Units)
        {
            GameObject toInstantiate = unitSprites[unitFactory(u)];
            u.setSprite(Instantiate(toInstantiate, new
Vector2(u.getPosition().getX(), u.getPosition().getY()), Quaternion.identity)
as GameObject);
            u.getSprite().transform.SetParent(boardHolder);
        }

        foreach (Unit u in player2Units)
        {
            GameObject toInstantiate = unitSprites[unitFactory(u)];
            u.setSprite(Instantiate(toInstantiate, new
Vector2(u.getPosition().getX(), u.getPosition().getY()), Quaternion.identity)
as GameObject);
            u.getSprite().transform.SetParent(boardHolder);
        }
        Debug.Log("Unit Setup OK");
    }
```

Acció moure

Per moure una unitat, primer és necessari sel·leccionarla. Després, es calcular el camí mínim, acotant fins el moviment de la pròpia unitat, i totes les caselles accessibles es ressaltaran de color blau.

```
   // action domain is {"move", "attack"}
    public int selectTile(String action)
    {
        Tile focus = Map[(int)cursor.transform.position.x,
(int)cursor.transform.position.y].tile;
        if (focus.Unit != null)
        {
            if (focus.Unit.getOwner() == 1 && focus.Unit.canAct())
            {
                paintHighlightedTiles(focus.Unit, action);
                if (this.unitSelected == null)
                {
                    SoundManager.instance.PlaySfx(this.select);
                    this.unitSelected = focus.Unit;
                    this.startPosition = focus;
                }
                return 1;
            }
        }
        return 0;
    }
```

Per ressaltar la casella, només cal posar com a actiu l'element, Es va pensar en fer-ho mitjançant sorting layers, però d'aquesta manera és més senzilla i òptima.

```
    // Paint the cell list obtained by the unit u and the option s
    private void paintHighlightedTiles(Unit u, String s)
    {
        List<Cell> positions = getCellsInRange(u, s);

        int type;
        if (s == "move") { type = 0; u.move(); }
        else if (s == "attack") { type = 1; u.idle(); }
        else { Debug.Log("Not a valid type"); return; }

        foreach (Cell cell in positions)
        {
            Map[cell.tile.getX(),
cell.tile.getY()].highlightLayers[type].SetActive(true);
        }
    }
```

```
    // Return a list containing all the cells in range of the unit and the
action
    private List<Cell> getCellsInRange(Unit u, String action)
    {
        int range = 0;
        List<Cell> positions = new List<Cell>();
        if (action == "attack")
        {
            range = u.getRange();
            positions = getCellsInAttackRange(range, u.getPosition());
        }
        else if (action == "move")
        {
            range = u.getMovement();
            positions = getCellsInMovementRange(range, u.getPosition());
        }
        else
            Debug.Log("Not a valid action");

        return positions;

    }
```

```
// Return a list containg the cells accessible by pos
    private List<Cell> getCellsInMovementRange(int range, Tile pos)
    {
        List<Cell> positions = new List<Cell>();
        LinkedList<Vertex> list = new LinkedList<Vertex>();
        initialisePath();
        Map[pos.getX(), pos.getY()].vertex.setMinDist(0);
        list.AddFirst(Map[pos.getX(), pos.getY()].vertex);
        positions.Add(Map[pos.getX(), pos.getY()]);
        while (list.First != null)
        {
            Vertex act = list.First.Value;
            list.RemoveFirst();

            foreach (Relation r in act.getNearBy())
            {
                if (isBetterWay(act, r, range))
                {
                    r.getVertex().setMinDist(act.getMinDist() + r.getCost());
                    r.getVertex().setPrevious(act);
                    list.AddLast(r.getVertex());
                    positions.Add(Map[r.getVertex().getX(),
r.getVertex().getY()]);
                }
            }
        }
        return positions;
    }
```

```
    // unit is the id of the unit; ori i s the initial position and desti
is the final position
    public int moveUnit(Unit unit, Vector2 ori, Vector2 dest)
    {
        if (!validPosition(dest))
        {
            Debug.Log("Invalid position");
            return 0;
        }

        // if tile is free or the unit is there, can move
        if (Map[(int)dest.x, (int)dest.y].tile.Unit != null &&
Map[(int)dest.x, (int)dest.y].tile.Unit != unit)
        {
            Debug.Log("Tile " + dest.x + " " + dest.y + " is occupied");
            return 0;
        }
        else
        {
            this.Map[(int)ori.x, (int)ori.y].tile.Unit = null;
            this.Map[(int)dest.x, (int)dest.y].tile.Unit = unit;
            this.Map[(int)dest.x,
(int)dest.y].tile.Unit.move(Map[(int)dest.x, (int)dest.y].tile);
            Debug.Log("unit " + unit + " in the position " + (int)ori.x + "
" + (int)ori.y + " moved to the position " + (int)dest.x + " " +
(int)dest.y);
            return 1;
        }
    }
```

Acció combat

Després de moure, en aquest cas no utilitzarem cap camí mínim, ja que el rang d'atac és fixe.

El combat es desenvolupa de la següent manera: la unitat sel·leccionada ataca a l'enemic, i si aquest pot contratacar, ho farà. Sempre es comprovarà si alguna de les dues unitats ha perdut tots els punts de vida, i en conseqüència s'eliminarà, tant de la llista d'unitats del jugador i del tauler.

Altrament, si es torna a sel·leccionar la mateix unitat, no s'atacarà i acabarà el torn.

```
// Return a list containg all the cells in a radius of range and pos in the
center
    private List<Cell> getCellsInAttackRange(int range, Tile pos)
    {
        List<Cell> positions = new List<Cell>();
        for (int i = 0; i <= range; i++)
        {
            if ((pos.getY() - range + i) >= 0)
            {
                positions.Add(Map[pos.getX(), pos.getY() - range + i]);
                for (int j = 1; j <= i; j++)
                {
                    if ((pos.getX() + j) < this.Columns)
positions.Add(Map[pos.getX() + j, pos.getY() - range + i]);
                    if ((pos.getX() - j) >= 0) positions.Add(Map[pos.getX()
- j, pos.getY() - range + i]);
                }
            }

            if ((pos.getY() + range - i) < this.Rows && i < range)
            {
                positions.Add(Map[pos.getX(), pos.getY() + range - i]);
                for (int j = 1; j <= i; j++)
                {
                    if ((pos.getX() + j) < this.Columns)
positions.Add(Map[pos.getX() + j, pos.getY() + range - i]);
                    if ((pos.getX() - j) >= 0) positions.Add(Map[pos.getX()
- j, pos.getY() + range - i]);
                }
            }
        }

        return positions;
    }
```

```
    public int attackOrWait()
    {
        Tile focus = Map[(int)cursor.transform.position.x,
(int)cursor.transform.position.y].tile;

        // attack enemy unit
        if (focus.Unit != null && focus.Unit.getOwner() !=
this.unitSelected.getOwner())
        {
            // this.unitSelected attack focus.Unit
            combat(focus.Unit);
            return 1;
        }
        // wait
        else if (focus.Unit != null && focus.Unit == this.unitSelected)
        {
            return 1;
        }
        return 0;
    }
```

```
    public void combat(Unit enemy)
    {
        // unitSelected attacks enemy
        SoundManager.instance.PlaySfx(this.attack);
        enemy.loseHP(trueDamage(enemy));

        if (enemy.isDead())
        {
            deleteUnitFromPlayer(enemy);
        }
        else if (canCounterattack(enemy))
        {
            SoundManager.instance.PlaySfx(this.attack);
            this.unitSelected.loseHP(trueCounterattack(enemy));
            if (this.unitSelected.isDead())
            {
                deleteUnitFromPlayer(this.unitSelected);
            }
        }
    }
```

```
    private void deleteUnitFromPlayer(Unit u)
    {
        if (u.getOwner() == 1)
        {
            SoundManager.instance.PlaySfx(this.die);
            this.Map[u.getPosition().getX(),
u.getPosition().getY()].tile.Unit = null;
            this.player1Units.Remove(u);
            Destroy(u.getSprite());
            u = null;
        }
        else if (u.getOwner() == 2)
        {
            SoundManager.instance.PlaySfx(this.die);
            this.Map[u.getPosition().getX(),
u.getPosition().getY()].tile.Unit = null;
            this.player2Units.Remove(u);
            Destroy(u.getSprite());
            u = null;
        }
        else
        {
            Debug.Log("Unknown player");
        }
    }
```

IA

En el capítol anterior vam presentar el pseudocodi de la IA. Aqui exposarem el seu funcionament.

Inicialment, sel·leccionem la primera unitat disponible del jugador, i calculem el seu rang de moviment i el rang total (moviment + atac) i avaluem si tenim unitats enemigues dintre del rang total. En cas que no en tingui cap, desplacem la unitat a una casella aleatòri dintre del rang de moviment; altrament escollim el millor enfrontament possible (tant la victima com la posició) i realitzem l'enfrontament.

Hem utilitzat Sleeps per tal de que el torn enemic duri un pèl més i perquè el jugador pugui veure què està fent l'enemic.

```
    public void enemyTurn()
    {
        this.unitSelected = nextUnitAvailable(2);
        this.startPosition = unitSelected.getPosition();

        moveCursor(new Vector2(this.startPosition.getX(), this.startPosition.getY()));
        System.Threading.Thread.Sleep(500);

        List<Cell> movementZone = getCellsInRange(this.unitSelected, "move");
        HashSet<Cell> attackZone = enemiesInRange(movementZone);

        if (attackZone.Count != 0)
        {
            Unit victim = evaluateEnemiesAndReturnBest(attackZone);
            Tile bestSpot = searchBestCellToAttack(victim.getPosition(), movementZone);
            moveCursor(new Vector2(bestSpot.getX(), bestSpot.getY()));
            System.Threading.Thread.Sleep(500);
            moveUnit(this.unitSelected, new Vector2(this.startPosition.getX(),
this.startPosition.getY()), new Vector2(bestSpot.getX(), bestSpot.getY()));
            combat(victim);
        }
        else
        {
            moveUnitToRandomPosition(movementZone);

        endUnitTurn();
    }
```

```
    private void moveUnitToRandomPosition(List<Cell> list)
    {

        int r = this.random.Next(list.Count);
        int tries = 3;
        while (list[r].tile.getUnit() != null && tries > 0)
        {
            r = this.random.Next(list.Count);
            tries--;
        }

        System.Threading.Thread.Sleep(500);
        if (tries == 0)
        {
            moveCursor(new Vector2(this.startPosition.getX(),
this.startPosition.getY()));
            moveUnit(this.unitSelected, new Vector2(this.startPosition.getX(),
this.startPosition.getY()), new Vector2(this.startPosition.getX(),
this.startPosition.getY()));
        }
        else
        {
            moveCursor(new Vector2(list[r].tile.getX(), list[r].tile.getY()));
            moveUnit(this.unitSelected, new Vector2(this.startPosition.getX(),
this.startPosition.getY()), new Vector2(list[r].tile.getX(),
list[r].tile.getY()));
        }
    }



    private HashSet<Cell> enemiesInRange(List<Cell> movement)
    {
        List<Cell> aux;
        HashSet<Cell> enemiesInRange = new HashSet<Cell>();
        foreach (Cell cell in movement)
        {
            if (cell.tile.getUnit() == null)    // if this unit can move there
            {
                aux = getCellsInAttackRange(this.unitSelected.getRange(),
cell.tile);
                foreach (Cell possibleEnemy in aux)
                {
                    //if we find an enemy, we add it to the list
                    if (possibleEnemy.tile.getUnit() != null &&
possibleEnemy.tile.getUnit().getOwner() != 2)
                    {
                        enemiesInRange.Add(possibleEnemy);
                    }
                }
            }
        }
        return enemiesInRange;
    }
```

Per escollir l'enemic, ho fem en funció de si el podem eliminar, sinó al que li causem més dany, i si i ha un empat, el que ens faci menys dany. Llavors, busquem la posició desde on atacar que ens aporti més defensa.

```
    private Unit evaluateEnemiesAndReturnBest(HashSet<Cell> set)
     {
         Unit victim = null;
         int dmg = 0;
         int ca = 10;
         foreach (Cell c in set)
         {
             if (enemyDies(c.tile.getUnit()))
                 return c.tile.getUnit();
             else if (dmg < trueDamage(c.tile.getUnit()))
             {
                 victim = c.tile.getUnit();
                 dmg = trueDamage(c.tile.getUnit());
                 ca = trueCounterattack(c.tile.getUnit());
             }
             else if (dmg == trueDamage(c.tile.getUnit()) && ca >
trueCounterattack(c.tile.getUnit()))
             {
                 victim = c.tile.getUnit();
                 ca = trueCounterattack(c.tile.getUnit());
             }
         }
         return victim;
     }

     private Tile searchBestCellToAttack(Tile objective, List<Cell> zone)
     {
         List<Cell> candidates = new List<Cell>();
         List<Cell> attackRange = getCellsInAttackRange(this.unitSelected.getRange(),
objective);
         foreach (Cell c in zone)
         {
             if (attackRange.Contains(c) && c.tile.getUnit() == null)
                 candidates.Add(c);
         }
         return bestPosition(candidates);
     }

     private Tile bestPosition(List<Cell> list)
     {
         Tile bestSpot = null;
         int def = -2;
         foreach (Cell c in list)
         {
             if (c.tile.getTerrainDef() == 1)
                 return c.tile;
             else if (c.tile.getTerrainDef() > def)
             {
                 bestSpot = c.tile;
                 def = c.tile.getTerrainDef();
             }
         }
         return bestSpot;
     }
```

Camí mínim

Per tal de mostrar i calcular el rang de moviment, utilitzem l'algoritme de Dijkstra. Per això, preparem la seva estructura mitjançant les classes Vertex i Relation, i en la classe BoardManager, que és on tenim el tauler, l'apliquem. Ja hem vist en la construcció del tauler com s'inicialitzava. A continuació, veurem el codi dels mètodes relacionats amb el camí.

```
    // Return a list containg the cells accessible by pos
    private List<Cell> getCellsInMovementRange(int range, Tile pos)
    {
        List<Cell> positions = new List<Cell>();
        LinkedList<Vertex> list = new LinkedList<Vertex>();
        initialisePath();
        Map[pos.getX(), pos.getY()].vertex.setMinDist(0);
        list.AddFirst(Map[pos.getX(), pos.getY()].vertex);
        positions.Add(Map[pos.getX(), pos.getY()]);
        while (list.First != null)
        {
            Vertex act = list.First.Value;
            list.RemoveFirst();
            foreach (Relation r in act.getNearBy())
            {
                if (isBetterWay(act, r, range))
                {
                    r.getVertex().setMinDist(act.getMinDist() + r.getCost());
                    r.getVertex().setPrevious(act);
                    list.AddLast(r.getVertex());
                    positions.Add(Map[r.getVertex().getX(),
r.getVertex().getY()]);
                }
            }
        }
        return positions;
    }
```

```
    void initialisePath()
    {
        for (int y = 0; y < Rows; y++)
        {
            for (int x = 0; x < Columns; x++)
            {
                Map[x, y].vertex.setMinDist(int.MaxValue);
            }
        }
    }

       // Evaluate if the path start->nextTo is cheaper than the previous value
    private bool isBetterWay(Vertex start, Relation nextTo, int limit)
    {
        int oldPathCost = nextTo.getVertex().getMinDist();
        int newPathCost = start.getMinDist() + nextTo.getCost();
        return (newPathCost <= limit && newPathCost < oldPathCost);
    }
```

9.2.4 Gestió de torns

Per gestionar els torns dels jugadors, s'ha fet de la següent manera: inicialment, comença la partida amb el torn del jugador o usuari. Quan en aquest ja no li queden més unitats per utilitzar, canvia a l'altre jugador i es va seguint aquesta seqüència fins acabar la partida.

Per fer això possible, en la classe GameManager utilitzem la variable de control Bool playersTurn: quan sigui true el jugador podrà moure i fer servir el cursor, però quan es posi a false no es llegiran els inputs.

Ho podeu veure en el mètode Update() a continuació:

```
        // Update is called once per frame
    void Update()
    {
        if (boardScript.player1Units.Count == 0)
        {
            this.stillPlaying = false;
            this.victory = false;
            enabled = false;
            EndGame();
        }

        else if (boardScript.player2Units.Count == 0)
        {
            this.stillPlaying = false;
            this.victory = true;
            enabled = false;
            EndGame();
        }
        else if (this.playersTurn && this.stillPlaying)
        {
            time++;
            readInput();
            if (boardScript.nextUnitAvailable(1) == null)
            {
                this.playersTurn = false;
                boardScript.startTurn(2);
                SoundManager.instance.PlayMusic(this.enemyTurnMusic);
            }
        }

        else if (!this.playersTurn && this.stillPlaying)
        {
            boardScript.enemyTurn();
            if (boardScript.nextUnitAvailable(2) == null)
            {
                this.playersTurn = true;
                boardScript.startTurn(1);
                SoundManager.instance.PlayMusic(this.playerTurnMusic);
            }

        }
    }
```

9.2.5 Controls

Per poder capturar l'input del jugador, hem utilitzat un parell de mètodes de la classe Input: getKeyDown i getKey (són mencionats en el capítol 7: Estudis i decisions). Aquests mètodes es criden al mètodes de readInput() i similars de la classe GameManager.

Utilitzem getKeyDown per les tecles A,S i Backspace per només capturar-la un cop i evitar que es detecti múltiples cops. En canvi, per les fletxes de direcció utilitzem getKey perquè ens interessa que al mantenir premuda la tecla ens ho segueixi detectant. Pel cas de getKey, també apliquem un filtre de temps per evitar capturar masses cops el mateix input (per defecte es captura en cada crida del mètode Update, en cada frame). Totes aquestes decisions s'han pres de cara a que els controls siguin agradables i no es facin pesats pel jugador.

Ja que la mateixa tecla pot tenir funcionalitats diferents, s'ha utilitzat una variable de control, anomenada stage, que en funció del seu valor el input llegit varia.

A continuació, adjunto un fragment de codi de la classe GameManager, on es podran veure els mètodes de readInput() i similars:

```
    void readInput()
    {
        if (stage == "defaultMap") { readInputInIdleMap(); }
        else if (stage == "moveUnitMap") { readInputMoveUnit(); }
        else if (stage == "attackUnitMap") { readInputAttackUnit(); }
    }

    // when nothing is selected
    void readInputInIdleMap()
    {
        if (Input.GetKeyDown(KeyCode.A))
        {
            int result = boardScript.selectTile("move");
            if (result == 1) { stage = "moveUnitMap"; }
        }

        if (Input.GetKeyDown(KeyCode.Backspace))
        {
            boardScript.endPlayerTurn();
        }

        else if (time > 4)
        {
            if (Input.GetKey(KeyCode.UpArrow))
                boardScript.moveCursor(0, 1);
            else if (Input.GetKey(KeyCode.DownArrow))
                boardScript.moveCursor(0, -1);
            else if (Input.GetKey(KeyCode.LeftArrow))
                boardScript.moveCursor(-1, 0);
            else if (Input.GetKey(KeyCode.RightArrow))
                boardScript.moveCursor(1, 0);

            time = 0;
        }
    }

    //when a unit is selected
    void readInputMoveUnit()
    {
        if (Input.GetKeyDown(KeyCode.A))
        {
            int result = boardScript.moveSelectedUnit();
            if (result == 1)
            {
                boardScript.clearHighlightedTiles();
                boardScript.selectTile("attack");
                stage = "attackUnitMap";
            }
        }
        else if (Input.GetKeyDown(KeyCode.S))
        {
            //cancel move attempt and de-seelct unit
            boardScript.clearHighlightedTiles();
            boardScript.freeUnitSelected();
            stage = "defaultMap";
        }

        else if (time > 4)
        {
            if (Input.GetKey(KeyCode.UpArrow))
                boardScript.moveCursorInsideArea(0, 1);
            else if (Input.GetKey(KeyCode.DownArrow))
                boardScript.moveCursorInsideArea(0, -1);
            else if (Input.GetKey(KeyCode.LeftArrow))
                boardScript.moveCursorInsideArea(-1, 0);
            else if (Input.GetKey(KeyCode.RightArrow))
                boardScript.moveCursorInsideArea(1, 0);
```

```
            time = 0;
        }
    }

    void readInputAttackUnit()
    {
        if (Input.GetKeyDown(KeyCode.A))
        {
            int result = boardScript.attackOrWait();
            if (result == 1)
            {
                boardScript.clearHighlightedTiles();
                boardScript.endUnitTurn();
                stage = "defaultMap";
            }
        }
        else if (Input.GetKeyDown(KeyCode.S))
        {
            Debug.Log("cancel");
            boardScript.clearHighlightedTiles();
            boardScript.returnSelectedUnitToStartingTile();
            boardScript.selectTile("move");
            stage = "moveUnitMap";
        }

        else if (time > 4)
        {
            if (Input.GetKey(KeyCode.UpArrow))
                boardScript.moveCursorInsideArea(0, 1);
            else if (Input.GetKey(KeyCode.DownArrow))
                boardScript.moveCursorInsideArea(0, -1);
            else if (Input.GetKey(KeyCode.LeftArrow))
                boardScript.moveCursorInsideArea(-1, 0);
            else if (Input.GetKey(KeyCode.RightArrow))
                boardScript.moveCursorInsideArea(1, 0);

            time = 0;
        }
    }
```

9.2.6 Victòria o derrota

Per tal de comprovar si la partida ha acabat, comprovem si algun dels dos jugadors s'ha quedat sense unitats. Si és el jugador usuari, llavors carregarem l'escena de derrota; altrament, la de victòria. Per fer això utilitzem les variables de control stillPlaying, que retorna cert o fals si encara s'està jugant la partida i la de victory, que avalua si el jugador ha guanyat a la IA o no.

A continuació es pot veure el mètode EndGame, que es crida en el mètode Update (adjunt en el apartat 9.2.4).

```
void EndGame()
    {
        if (!this.stillPlaying && this.victory)
        {
            Debug.Log("victory");
            SoundManager.instance.PlayMusic(this.victoryMusic);
            SceneManager.LoadScene(3);
        }
        else if (!this.stillPlaying && !this.victory)
        {
            Debug.Log("defeat");
            SoundManager.instance.PlayMusic(this.gameOverMusic);
            SceneManager.LoadScene(2);
        }
    }
```

9.3 Proves realitzades

Per tal de detectar el major nombre d'errors possible i sol·lucionar-los, s'han fet moltes proves executant el joc, i també s'ha utilitzat la comanda Debug.Log() per imprimir a la consola missatges i valors de variables.

10. Implantació i resultats

En aquest capítol veurem quines lleis tenen efecte sobre el nostre joc, i si presenten algun problema o no i mostrarem, mitjançant imatges, el transcurs d'una partida.

10.1 Legislació i normativa vigent

En termes de seguretat, el joc no presenta cap problema, ja que pot ser executat en mode usuari (no es necessita ser administrador ni cap mode en especial).

Com que no es guarden cap tipus de dades, ni tampoc dades de caràcter personal, no s'ha tingut en compte la llei orgànica de protecció de dades de caràcter personal (LOPD).

Segons la llei de serveis de la societat de la informació i comerç electrònic, la LSSICE, el joc no constitueix una activitat econòmica en cap dels sentits.

Per tant, el videojoc no presenta cap problema amb la legislació.

10.2 Captures de pantalla del joc

A l'executar el joc, ens apareixerà el menú principal. En aquest menú, podrem decidir si jugar o bé tancar el joc (Figura 10.1)

Figura 10.1 Menú principal

Si escollim jugar, es carregarà el mapa i començarà la partida, sent el primer torn per l'usuari, que disposarà de les unitats de color blau (Veure Figura 10.2).

Figura 10.2 Inici de la partida

Ara, sel·leccionarem l'arquer aliat, per veure a quines caselles es pot desplaçar (Veure Figura 10.3).

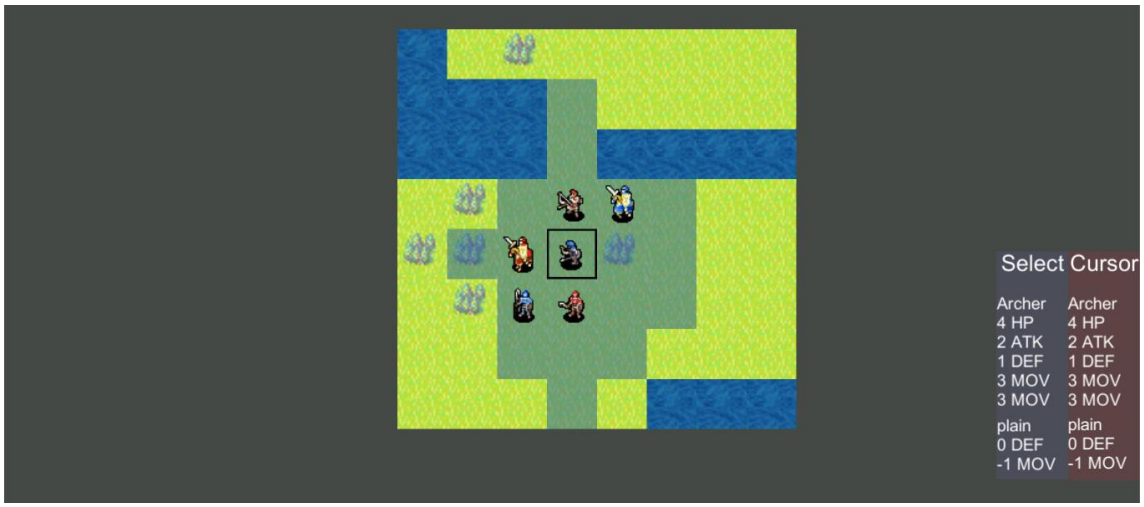


Figura 10.3 Sel·lecció de l'arquer

El desplacem dos caselles cap a la dreta i comprovem el rang d'atac que té (Veure Figura 10.4).

Figura 10.4 Moviment realitzat i inici de la fase d'atac

Com que el llancer enemic estava dins del rang, l'hem pogut atacar. Aquest no ha pogut contratacar perquè no estem dintre el seu rang, i li hem causat 4 punts de mal a causa de que tenim avantatge. Actualment li queda 1 punt de vida i el torn del nostre arquer s'ha acabat (Veure Figura 10.5).

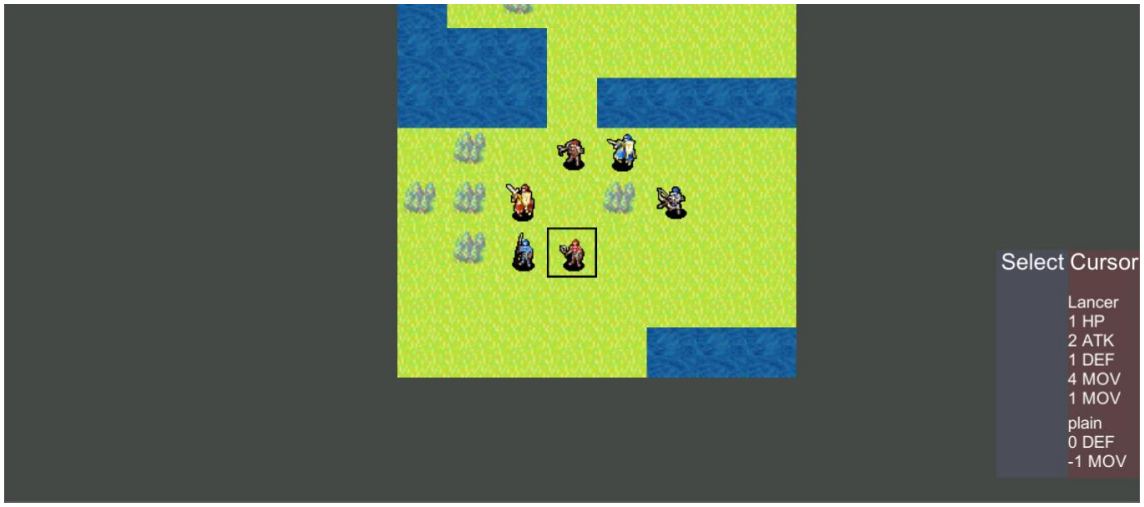


Figura 10.5 Atac realitzat i torn de l'arquer acabat

Sel·leccionarem el llancer, i ens disposarem a atacar el llancer enemic (Veure Figura 10.6).

Figura 10.6 Moviment del llancer

Atacarem el llancer enemic, i aquest no podrà contraatacar a causa que, amb el nostre atac, l'eliminarem (Veure Figura 10.7).

Figura 10.7 Fase d'atac del llancer

Un cop realitzat l'atac, com que el llancer enemic ha perdut tots els punts de vida, s'elimina i ja no pot ser utilitzat més (Veure Figura 10.8).

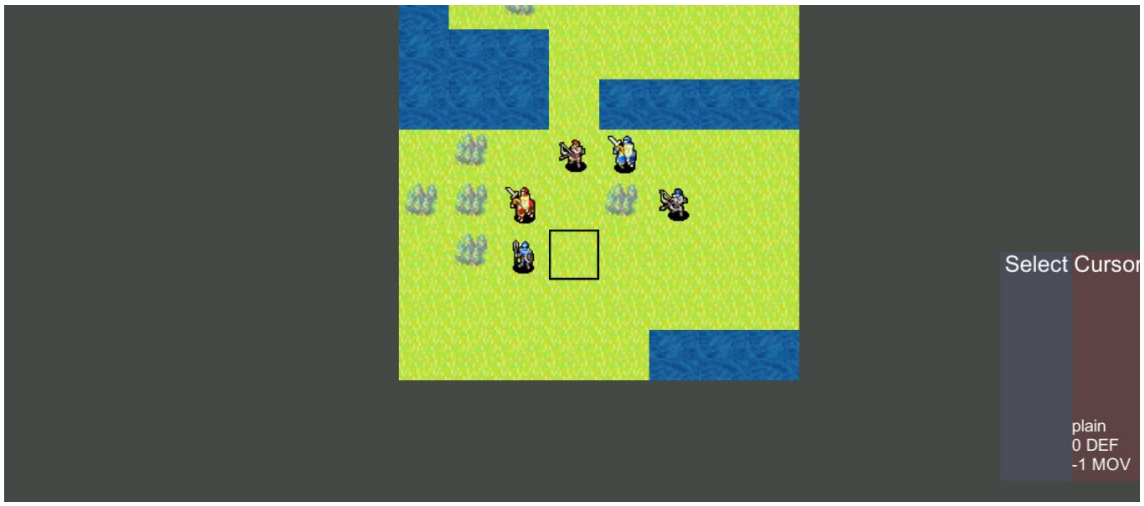


Figura 10.8 Eliminació del llancer enemic

Per acabar el nostre torn, amb el cavaller atacarem a l'arquer enemic, i l'eliminarem (Veure Figura 10.9).

Figura 10.9 Elminiació de l'arquer enemic

Ara és el torn de l'enemic. El cavaller enemic es mourà al bosc de dos caselles a la seva dreta i atacarà al arquer aliat, eliminant-lo (Veure Figura 10.10).

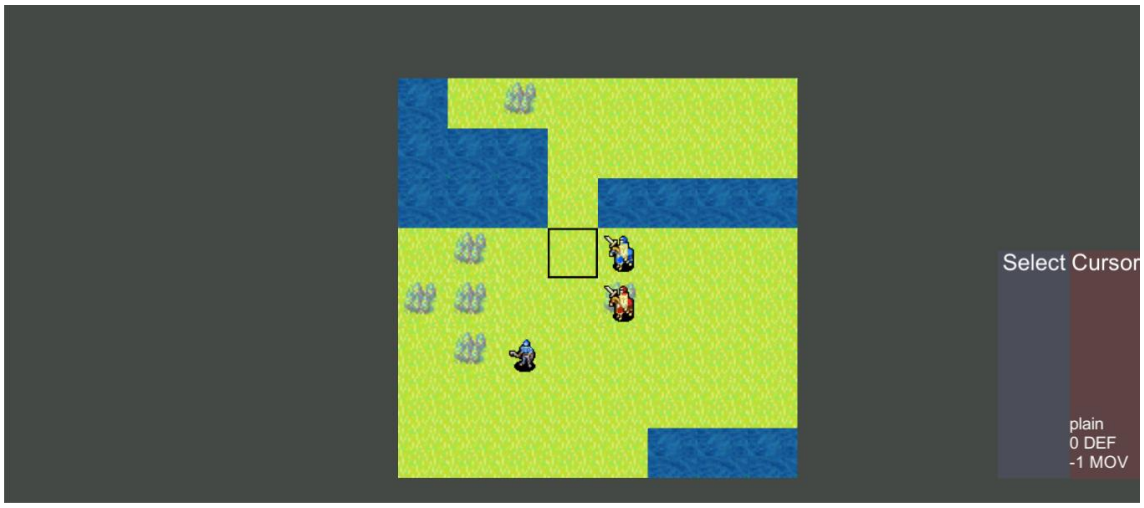


Figura 10.10 Eliminació de l'arquer aliat

Un altre cop en el nostre torn, atacarem al cavaller amb el llancer, però a causa de que està en un bosc, li causarem menys dany i en aquest torn no serà possible eliminar-lo (Veure Figura 10.11).

Figura 10.11 Atac del llancer contra el cavaller

Finalment, el jugador aconsegueix eliminar el cavaller enemic i aconsegueix la victòria (Veure Figura 10.12).

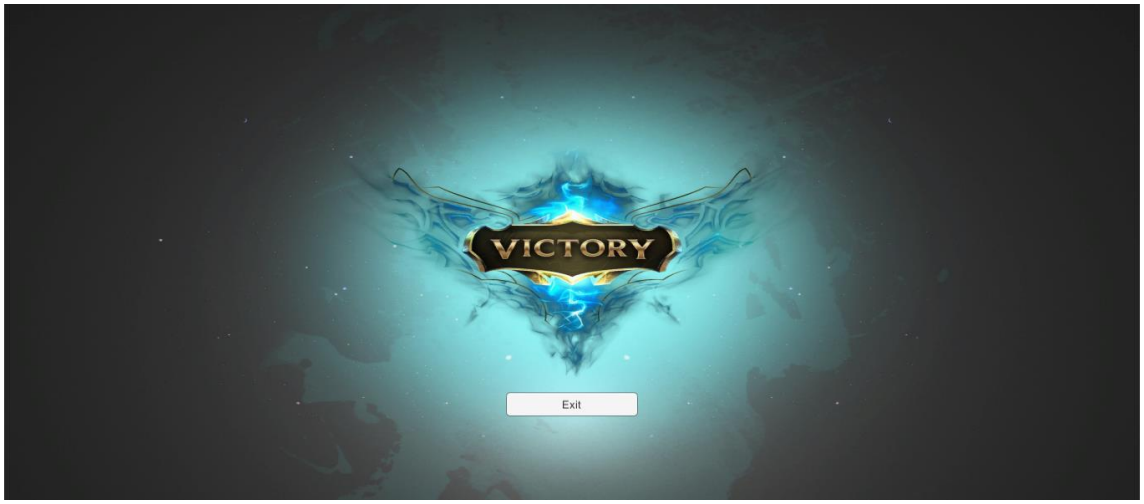


Figura 10.12 Victòria

Altrament, si l'enemic acaba eliminant les unitats del jugador, será la derrota del jugador (Veure Figura 10.13).

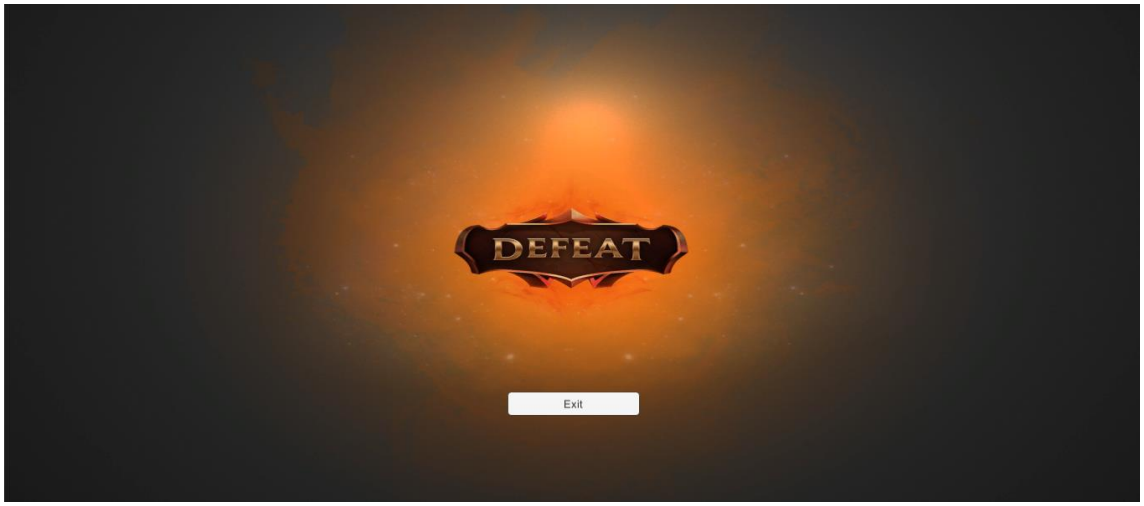


Figura 10.13 Derrota

11. Conclusions

En realitzar aquest projecte, s'han après molts conceptes que no es coneixien abans:

- utilitzar el motor de Unity i algunes de les seves funcionalitats,
- el llenguatge de C#, que no ha resultat excessivament complicat ja que s'estava familiaritzat amb altres que s'hi assemblaven.
- tot el procés de desenvolupament d'un videojoc.

I també s'ha après i reflexionat sobre altres temes, alguns dels quals han anat bé i s'ha obtingut un resultat favorable, i altres que no tant. Es poden englobar en els dos punts següents:

El disseny del videojoc: es va dedicar el temps suficient, al principi de tot del projecte, a definir còm seria i de què tractaria el joc. Això va permetre que, en les següents etapes del desenvolupament, no s'hagués de canviar res d'aquest disseny o idea i es va mantenir tal i com s'esperava fins al final.

La gestió del temps: aquest projecte ha estat el més gran i complert que he fet per part meva, i m'ha ensenyat que és necessari, al principi de tot, estudiar quin és el temps que s'hi podrà dedicar i aleshores ser constant i aprofitar aquest temps al màxim. En aquest cas, vaig sobreestimar el temps que hi podria dedicar, i això va resultar en haver d'anar a la convocatòria de Setembre, en comptes de la de Juny (veure diagrames de Gantt, al capítol 4, Planificació, sub capítol 3, Temps estimat i final) i a no poder fer totes les funcionalitats que voldria, les quals algunes estan esmentades en el capítol 12, Treball futur.

Com a punt final per aquest capítol que tracta les conclusions, comentaré que estic satisfet amb el resultat del projecte, i que tot i que hagi fallat en algun punt, a partir d'aquest moment ja en sóc conscient, i en futurs projectes ho tindré més en compte i evitaré que torni a passar.

12. Treball futur

Aquest joc es va plantejar inicialment amb uns requeriments funcionals bàsics, però molts dels aspectes desenvolupats s'han fet de cara a que es puguin ampliar o afegir-ne de nous. A continuació descriure els més interessants o importants, els quals millorarien la jugabilitat i farien la partida més entretinguda:

- Afegir més classes: actualment n'hi ha 3, i actuen com el típic joc de pedra-paper-estisora. El codi ja està preparat, i només caldria afegir un model i els sprites corresponents, així com modificar els mètodes de factory.
- Nous terrenys: a l'igual que les unitats, es poden afegir nous terrenys per tal d'obtenir escenaris o mapes més ben definits i també per millorar l'element estratègic.
- Noves mecàniques de joc, com per exemple sobreviure X torns o bé arribar a la casella X, i també canvis climàtics, com per exemple la pluja o la boira, o el dia i la nit.
- Afegir el concepte de diners o moneda al joc, i permetre al jugador invertir-los en noves unitats, o millores en les unitats (més moviment, més atac, etc.). S'implementaria una classe nova, Building, que es trobaria dintre de una casella i en ella es podrien seleccionar les unitats a crear.
- També es podria permetre al jugador canviar el mapejat de les tecles (per defecte són les que hi ha esmentades, però es podria permetre canviar-les), o inclús utilitzar pads.
- Permetre executar el joc en diferents plataformes.
- Millorar la IA existent, inclús fer-ne múltiples en funció de l'objectiu del mapa (si el jugador ha de defensar una posició, l'enemic seria més agressiu, i viceversa).

13. Bibliografia

13.1. Pàgines web

Microsoft, Guía de C#, https://docs.microsoft.com/es-es/dotnet/csharp/, 2018.

Unity Technologies, Tutorials, https://unity3d.com/es/learn/tutorials, 2018.

Unity Technologies, Unity – Manuals, https://docs.unity3d.com/Manual/, 2018.

The VG Resource (comunitat web), The spriters resource, https://www.spriters-resource.com/

The VG Resource (comunitat web), The sounds resource, https://www.sounds-resource.com/

Wikimedia Foundation, Wikipedia: The free encyclopedia, https://en.wikipedia.org/, 2001

13.2 Llibres

Schell, Jesse. (2018). *The Art of Game Design*. Pittsburgh: Elsevier Inc.

14. Manual d'usuari i/o instal·lació

14.1 Instal·lació i execució

El videojoc és un executable de Windows (arxiu .exe) i no requereix cap mena d'instal·lació. Tan sols cal executar l'arxiu i apareixeran les opcions gràfiques, i si es vol o no executar en pantalla completa. Un cop s'han confirmat aquestes opcions, només restarà prémer Play! I el joc s'executarà.

14.2 Controls

Per poder jugar, serà necessari un ratolí i un teclat. Els inputs necessaris apareixen resaltats en les figures 14.1 i 14.2:

Figura 14.1 Inputs del teclat

Figura 14.2 Inputs del ratolí

El ratolí s'utilitzarà en les escenes inicials i finals, per iniciar la partida o bé per tancar el joc.

Un cop a dins la partida, només serà necessari el teclat, utilitzant les tecles següents:

A : serveix per confirmar accions i sel·leccionar la nostra unitat que es troba a la casella on hi ha el cursor.

S : serveix per cancel·lar accions. Quan es cancel·la una acció, es torna a l'acció anterior, fins arribar a la sel·lecció d'unitats.

Backspace: serveix per acabar el torn del jugador (s'acaba automàticament si s'han mogt totes les unitats).

Fletxes de direcció : permeten desplaçar el cursor pel tauler

14.3 Objectiu de la partida

La partida acabarà quan un dels dos jugadors es quedi sense unitats.

14.4 Consells bàsics

Cada classe té avantatge (provoca més dany) quan s'enfronta a una altra classe. En la figura 14.3 es pot veure-les:

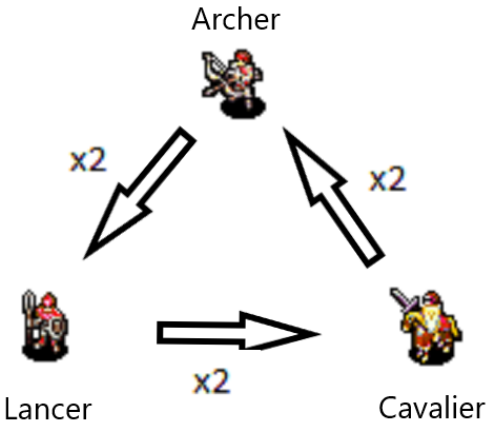


Figura 14.3 Avantatges entre classe

Les estadístiques de les unitats són les següents:

Classe	Archer	Cavalier	Lancer
Sprite			
HP	4	5	5
ATK	2	2	2
DEF	1	1	1
MOV	3	6	4
RNG	3	1	1

Cada terreny aplica un modificador de Defensa a la unitat que s'hi troba, i també té un cost de Moviment per passar-hi. En la figura 14.4 es poden veure els diferents tipus de terreny:

Figura 14.4 Terrenys