Universitat de Girona
Escola Politècnica Superior

Grau en Enginyeria Informàtica

PROJECTE FINAL DE GRAU

Desenvolupament d'un joc modern de tipus Tower Defense

Autor:
Marc Sayrach Baró

Tutors:
Gustavo Patow

MEMÒRIA

Convocatòria:
Juny 2023

Departament:
IMAE

Universitat de Girona
Escola Politècnica Superior

TREBALL FINAL DE GRAU

MTD:
Desenvolupament d'un joc modern de tipus Tower Defense

Autor:
Marc Sayrach Baró

Juny 2023

Grau en Enginyeria Informàtica

Tutor:
Gustavo Patow

– ÍNDEX -

– 1. ÍNDEX DE FIGURES -

CAPÍTOL 1

– 1. INTRODUCCIÓ -

Es defineix com a joc l'activitat estructurada que realitza un o més jugadors utilitzant la seva imaginació o alguna eina, per crear una situació amb un nombre determinat de regles amb la finalitat de proporcionar entreteniment o diversió. Dintre dels jocs podem identificar els videojocs.

Els videojocs s'han convertit en els últims anys en una de les indústries d'entreteniment més importants del món, a causa del seu creixement exponencial de consumidors en les últimes dècades, arribant a facturar 207.000 milions d'euros l'any 2022 arreu del món.

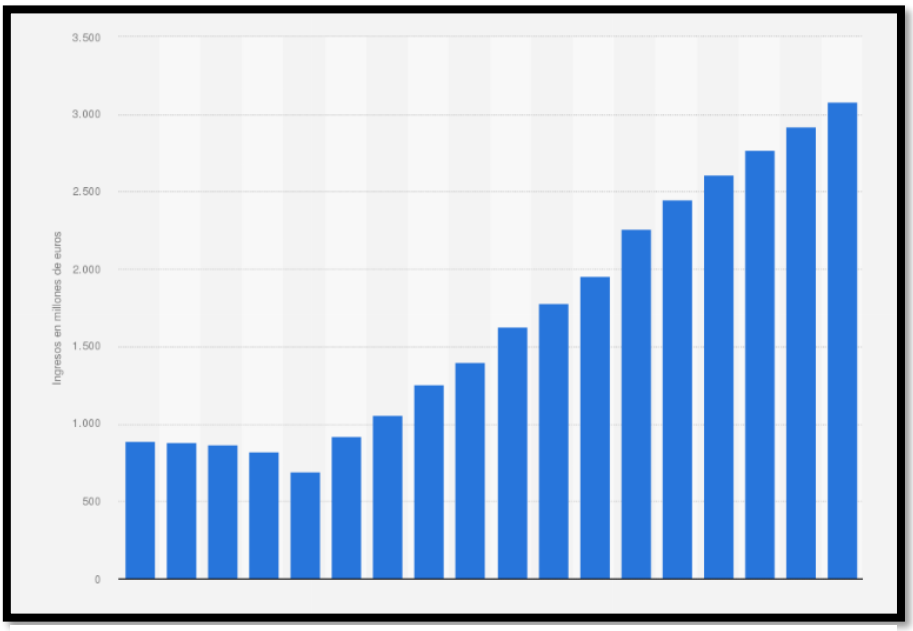

Figura 1.1: Facturació anual d'España en el mercat dels videojocs

A causa d'aquest èxit, els videojocs han anat evolucionant i adaptant als diferents públics, creant molts modes diferents, des dels clàssics com els de plataformes fins als actuals com els battle royale. Això ha provocat que grans empreses busquin invertir en aquest sector, ja sigui creant videojocs de més pressupost i més ambiciosos (triple A) o creant infraestructures al voltant d'ells, com els eSports (esports electrònics) o Steam (plataforma de distribució de videojocs), entre altres.

Figura 1.2: Final del eSport més important del moment

Cal destacar que cada cop és més difícil innovar, ja que hi ha propostes de videojocs més originals i complexes, sobretot pels petits desenvolupadors anomenats *indie*. Malauradament, de mica en mica a causa d'aquesta evolució, s'ha anat deixant en l'oblit els videojocs clàssics a la recerca de fórmules noves quan, sovint, les antigues encara tenen molt per aportar. Un d'aquests jocs clàssics és el Tower Defense, on el jugador ha de construir torres en llocs predeterminats per evitar que els enemics arribin al final del camí.

En aquest projecte es busca recuperar aquesta categoria i fer-la més dinàmica per crear una versió més moderna que s'adapti a les noves generacions.

– 1.1 MOTIVACIONS PERSONALS -

En el transcurs de la carrera he après sobre moltes tecnologies diferents, i esperava en aquest projecte poder continuar aprenent-ne de noves. Si bé abans de començar, ja tenia entès que no serien difícils d'aprendre amb la base obtinguda aquests anys, sí que m'enriquiria d'algunes que en són completament diferents.
Per altra banda, de ben petit sempre he estat un aficionat als videojocs. No tinc record d'aprendre a jugar, vaig començar molt abans de ser-ne conscient. Puc associar cada etapa de la meva vida amb un videojoc, que sempre han estat un aspecte important de mi.

Els videojocs m'han ajudat a connectar més amb els amics, amb la parella, amb la família, amb la gent en general. Així com un partit de futbol pot unir a un grup de gent, doncs això han significat per a mi els videojocs. Per aquest fet, he decidit en fer del meu últim treball de carrera un videojoc. Aprenent en el camí com es gestiona i desenvolupa, i esperar que em serveixi com un inici de portfoli per poder treballar algun dia en alguna empresa de videojocs per continuar agafant experiència.

Per aquest fet, em feia molta il·lusió poder participar en el desenvolupament d'un videojoc. Aprendre de primera mà cada etapa del desenvolupament i poder finalment aportar el meu granet de sorra en aquest sector, creant una variació innovadora dels Tower Defense.

– 1.2 PROPÒSIT -

El propòsit d'aquest treball de final de carrera d'Enginyeria Informàtica és la creació d'una demo d'un videojoc 2D de temàtica Tower Defense amb el motor de videojocs Unity. Amb la intenció de diferenciar-lo de la resta de jocs d'aquesta temàtica agafant aspectes dels RPG. Fent que es controli un personatge al mateix temps que es construeixen torres a qualsevol punt del mapa. A més, es volen agafar dels jocs de tipus roguelike, fent que a cada partida es generin diferent els enemics.

– 1.3 OBJECTIUS -

Per poder dur a terme el propòsit del projecte, s'han de complir els següents objectius:

- Planificació del joc.
- Estudi de les alternatives a les eines finalment seleccionades.
- Utilitzar i treure profit de les característiques d'Unity.
- Dominar un dels llenguatges de programació que suporta Unity, C#.
- Cercar sprites que concordin amb la idea del projecte.
- Crear animacions amb els *sprites*.
- Cerca de material de domini públic per afegir-los al projecte.
- Crear la mecànica de moviment i d'habilitats del jugador.
- Crear la mecànica d'habilitats.

- Crear una diversitat d'enemics amb tècniques de *Behavior Trees (IA).*
- Crear la mecànica de les torres per a què el personatge les instanciï.
- Crear un sistema de comptador per a les monedes i els enemics derrotats.
- Crear un sistema de menús.
- Crear un *mini mapa*.
- Cercar sons que siguin adients i implementar-los.
- Crear un sistema de regulació de so.
- Treballar amb eines de gestions de projectes.
- Documentar el projecte.

CAPÍTOL 2

– 2. VIABILITAT -

Hardware

Per a desenvolupar aquest projecte no són necessaris un gran nombre de recursos i el cost d'infraestructura ha estat adquirir un ordinador de taula de 1600 euros amb les següents característiques (Taula 2.1).

Sistema operatiu	Windows 11
Processador	AMD Ryzen 5 5600
Memòria	16 GB
Gràfica	NVIDIA Geforce GTX 3060

Taula 2.1 Característiques del segon maquinari

És recomanat utilitzar maquinàries de més rendiment per facilitar el desenvolupament del projecte. Els primers mesos es va usar un ordinador d'una generació anterior (Taula2.2), però ja que el desenvolupament d'un videojoc utilitza programari pesat, el que es va notar sobretot en la memòria, perquè si es te poca és probable que el joc ocupi gran part d'ella.

Sistema operatiu	Windows 10
Processador	Intel Core i7
Memòria	8 GB
Gràfica	NVIDIA Geforce GTX 1050

Taula 2.2 Característiques del primer maquinari

Software

Tots els programaris utilitzats en el projecte tenen una versió gratuïta i una versió de pagament. Per aquest cas s'han utilitzat els gratuïts. Si en un futur es comercialitza el videojoc, Unity ens faria pagar una llicència professional en el cas que facturéssim més de 100000$ anuals. En resum, la inversió inicial d'aquest projecte seria de 1600 euros.

– 2.1 VIABILITAT LEGAL -

Tots els recursos externs que s'han inclòs en el projecte compleixen els drets legals i de propietat intel·lectual, ja que s'han obtingut de fonts d'ús lliure, donant crèdit a l'autor corresponent.

– 2.2 RECURSOS HUMANS -

La producció d'un videojoc sol estar associada a un equip que està format per diversos perfils tècnics, els quals es comuniquen per assegurar la coherència en el procés de creació i optimitzar els resultats finals en termes de temps d'entrega, difusió, entre altres aspectes. Entre els perfils tècnics més importants, es poden esmentar:

- El programador, que implementa el disseny i el funcionament del joc.
- El compositor musical, que crea els efectes de so i la música del joc.
- El dissenyador, que es responsabilitza de desenvolupar el concepte del joc, el guió, la jugabilitat, les mecàniques de joc i el disseny d'escenaris i nivells.
- El dissenyador gràfic, que s'encarrega dels elements visuals del joc i defineix l'estil gràfic.

Els perfils de compositor musical i dissenyador gràfic no s'han pogut desenvolupar de zero a causa de la falta de coneixements i temps. S'agraeix a totes les persones que han compartit els seus dissenys gràfics, sons i músiques en plataformes web amb llicència Creative Commons, ja que han fet possible el desenvolupament d'aquest projecte fins a l' etapa final. Els gràfics en gran part han estat extrets de [*FoozleCC*, s. d.].

– 2.3 VIABILITAT ECONÒMICA -

S'ha tingut en compte que una empresa amb gent experimentada no trigaria més de dos mesos en fer el desenvolupament d'un projecte com aquest. Si tenim en compte que les jornades laborals són de 40 hores mensuals, hem de calcular el cost per un total de 80 hores per cada perfil tècnic.

S'han extret els sous mostrats a la taula 2.3 dels següents enllaços:

- Programador de videojocs: [*¿Cuánto Cobra un Programador de Videojuegos? (Sueldo 2023) | Jobted.es*, s. d.].
- Compositor musical de videojocs : [*Sueldo: Compositor en España en 2023*, s. d.]
- Dissenyador de videojocs: [Tokio School, 2023]
- Dissenyador gràfic de videojocs: [*Sueldo: 2D Artist (Mayo, 2023)*, s. d.]

Tots aquests sous estan pensats en el que cobraria de mitjana un treballador a Espanya.

	Salari anual
Programador	32.100 €
Compositor musical	22.000 €
Dissenyador	30.000 €
Dissenyador gràfic	30.000 €
Cost total = 114.100 anual → dos mesos: 19.016 €	

Taula 2.3 Salaris perfil tècnic

Suposant que tots els treballadors tenen la mateixa maquinària esmentada en la taula 2.1, llavors tindríem un cost tecnològic de 1600*4 = 6.400 €.
Per tant, el cost total d'aquest projecte seria aproximadament de 25.416 €.

CAPÍTOL 3

– 3. METODOLOGIA -

En el desenvolupament de tot projecte, és crucial decidir quina metodologia s'utilitzarà per dur-lo a terme. Per tant, és important que tinguem coneixement de les més usades avui en dia. [*Richter, L. (s. d.)*].

– 3.1 METODOLOGIES ESTÀNDARDS -

Metodologia Agile

Les metodologies Agile són un enfocament iteratiu en la gestió de projectes que utilitza cicles curts de desenvolupament, anomenats "sprints", per obtenir informació dels usuaris sobre el que necessiten a continuació. Això permet als equips de projecte elaborar millores contínues al llarg del procés, en lloc de fer-les només a la fi del projecte.
La metodologia Agile posa èmfasi en la creativitat i la flexibilitat, en lloc de seguir un calendari rígid. Això és particularment útil en projectes complexos de desenvolupament. A més, l'enfocament Agile ofereix oportunitats per als usuaris i les parts interessades per aportar comentaris durant el procés i participar en la presa de decisions sobre com el projecte ha de continuar o quins canvis són necessaris amb urgència.

Metodologia Scrum

Scrum és una metodologia de gestió de projectes àgil que comparteix principis amb altres metodologies d'aquesta categoria. Aquesta metodologia es basa en la utilització de sprints curts per crear un cicle de projecte. Scrum és utilitzada sobretot per organitzar equips i tasques, especialment en casos en què és necessari desenvolupar productes de manera ràpida o quan els equips han de col·laborar de manera estreta en períodes de temps curts, com ara en el context de startups.

Metodologia Kanban

Kanban és una metodologia que ajuda als membres de l'equip a mantenir una estructura senzilla i concentrar-se en les tasques més necessàries i importants. A través de l'ús de targetes que representen cada etapa d'un procés, Kanban permet visualitzar els fluxos de treball i el progrés dels projectes d'una manera clara i senzilla. Aquestes targetes es col·loquen en un tauler, que és com un mur on es van

movent a mesura que es va completant cada etapa. Així, els membres de l'equip poden identificar ràpidament quines tasques estan avançant bé i quines estan estancades, per així poder solucionar els problemes més eficientment i mantenir el projecte en el bon camí.

Figura 3.1 Metodologia Kanban

Metodologia Waterfall

També anomenada metodologia en cascada, és una gestió de projectes que es considera tradicional, la qual implica acordar el propòsit del projecte i planificar-lo de forma exhaustiva.
Aquesta tècnica consisteix a dividir el projecte en diferents processos que s'executen de forma successiva fins que s'assoleixen els objectius de cada fase o de tot el projecte en general. Amb la planificació en cascada, els responsables del projecte poden controlar amb detall cada fase.
Tanmateix, aquesta metodologia deixa poc espai per a possibles canvis si sorgeixen situacions imprevistes. Aquesta tècnica és comuna en el sector del desenvolupament industrial, la construcció i el programari.

– 3.2 METODOLOGIA ESCOLLIDA -

Un cop fet l'estudi, podem veure que hi ha una gran diversitat de metodologies diferents. Finalment, s'ha decidit seguir una metodologia personalitzada proposada pel nostre tutor, ja que cap s'adaptava a les necessitats del projecte.

Els passos de la metodologia són els següents:

- Decidir quin projecte es portarà a terme.

- Adquirir el coneixement i les eines necessàries per dur a terme el projecte.
- Descompondre i organitzar el projecte en diferents mòduls funcionals per facilitar-ne la realització per separat.
- Seleccionar una part del projecte que encara no s'ha realitzat i desenvolupar-la seguint l'ordre dels mòduls definits al punt 3.
- Provar si la part feta funciona correctament:
  - Si hi ha problemes, tornar al punt 4 per corregir els errors.
  - Si funciona correctament, comprovar si era la darrera part a dur a terme. Si és així, passar al punt 6, si no tornar al punt 4 per seleccionar la següent part.
- Integrar totes les parts del projecte en un sol element i comprovar-ne el funcionament:
  - Si no funciona correctament, tornar al punt 4 per analitzar i corregir les parts que no funcionen.
  - Si funciona correctament, passar al punt 7.
- Generar diferents models d'exemple a partir de la combinació de les parts i comprovar-ne el correcte funcionament:
  - Si els models no es comporten com s'esperava, tornar al punt 4 per analitzar i corregir les parts que no funcionen.
  - Si s'obté el resultat desitjat, passar al punt final 8.
- Revisar i millorar la documentació portada a cap durant el projecte i elaborar la documentació final.

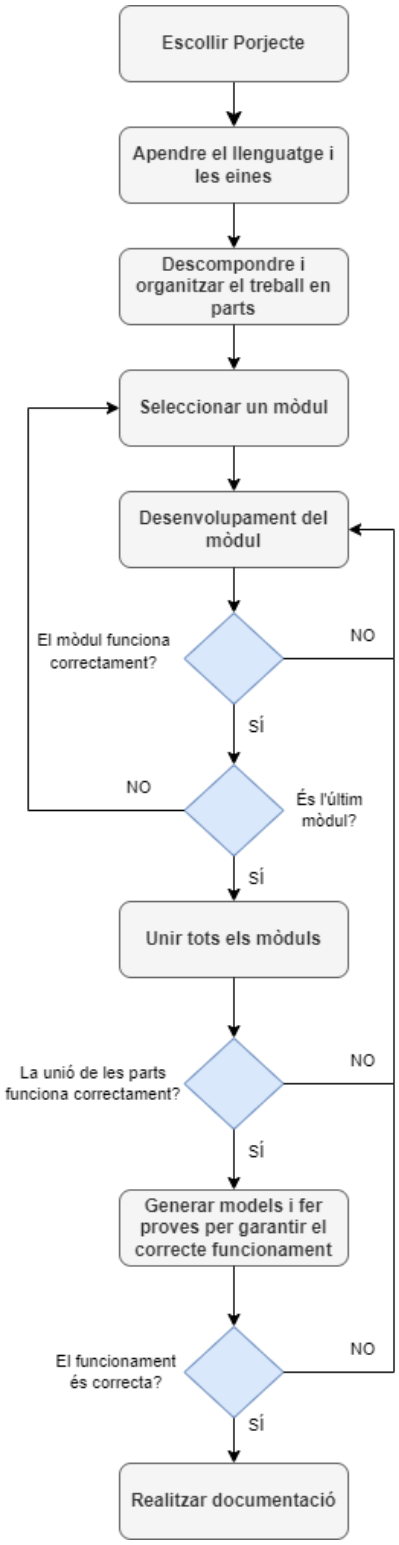

Figura 3.2 Diagrama del flux de la metodologia

CAPÍTOL 4

– 4. PLANIFICACIÓ -

La planificació és una etapa essencial per a qualsevol projecte, ja que ens ajuda a establir la prioritat adequada de les diferents tasques i a tenir un millor control del temps per aconseguir l'èxit en el desenvolupament. En el meu cas, la planificació ha estat crucial, ja que havia de conciliar el projecte amb el darrer any de la carrera i sobretot amb l'ocupació professional.

En aquesta secció, analitzarem com vam organitzar la planificació al principi del projecte i la compararem amb la planificació final.

– 4.1 PLANIFICACIÓ INICIAL -

El treball va començar l'estiu del 2022, quan es va decidir que el treball final consistiria en desenvolupar un videojoc, encara que aquest inici era únicament cercar i pensar en què podria consistir el videojoc. En la pràctica, el projecte no va començar fins a mitjans de setembre, quan el tutor va acceptar portar el seguiment del projecte.

Vam decidir dividir les tasques del projecte en cinc apartats:

- Plantejament i estudi
- Disseny
- Implementació
- Integració i testeig
- Documentació

– 4. 2 TASQUES PLANIFICADES -

Planificació del videojoc

En aquesta primera tasca es va definir l'objectiu del videojoc, les característiques principals, els diferents elements i les regles principals.

Estudi dels diferents motors

Es van estudiar les diverses opcions de motor de videojocs que hi ha en el mercat per a utilitzar en el projecte. Finalment es va escollir el motor Unity.

Estudi del llenguatge C#

En aquesta etapa s'ha realitzat una investigació sobre el funcionament del motor de videojocs Unity i el llenguatge de programació C# que s'utilitza per al desenvolupament. Aquest aprenentatge s'ha aconseguit mitjançant cursos en línia, vídeos informatius disponibles en plataformes com YouTube, pàgines web d'aprenentatge, i la documentació oficial d'aquests programes i llenguatges.

Cerca d'elements gràfics

Es cerquen tots els sprites necessaris per formar el joc: personatge, enemics, torres, interfaç, mapa, etc. Un cop acabada la cerca, es creen les animacions i tilemaps a partir dels sprites i/o elements gràfics.

Disseny del Sistema del jugador

En aquesta secció s'ha establert el sistema de control del personatge del videojoc, juntament amb els algoritmes necessaris per dur-lo a terme. També es va fer un aproximat de la física (velocitat, abast d'atac, mida, etc).

Disseny dels enemics i torres

Aquest apartat es va dissenyar les diferents característiques que tindrien els enemics i les torres del videojoc. També es van definir quins comportaments tindria cada enemic, és a dir la IA.

Disseny del control de càmera

En aquesta etapa es va definir com funcionaria la càmera del videojoc. Es va decidir per una càmera estàtica que mostres tot el mapa en una vista zenital.

Figura 4.1 Videojoc amb vista zenital

Disseny del videojoc

En aquesta fase es dissenya l'esquema principal del videojoc, és a dir les diverses pantalles i la interfície d'usuari de cadascuna.

- Pantalla 1: Tutorial.
- Pantalla 2: Defensar la porta dels enemics.
- Pantalla 3: Lluitar contra el cap final.

També es va definir el sistema d'aparició dels enemics, el qual es va decidir en un format aleatori. D'aquesta manera el videojoc agafa una mica l'escènica dels roguelike.

Disseny dels menús

Aquí es va decidir com serien els diferents menús que hi hauria en el videojoc, el menú d'inici, de pausa, de mort i de victòria.

Implementació del sistema del jugador

En aquesta secció es va dur a terme la implementació del sistema de control del personatge principal, així com el sistema de col·lisions que tindria amb els altres elements del videojoc.

Implementació dels enemics i torres

S'implementen les diferents torres i enemics, juntament amb les diverses intel·ligències artificials de cada enemic. Depenent de l'enemic és més o menys complexa.

Implementació del control de càmera

En aquesta tasca es va implementar la càmera de forma estàtica i que enfoques a tot el mapa de manera zenital. Una vegada estava en funcionament, vam veure que no acabava d'encaixar amb la idea del projecte. Es va decidir fer una càmera zenital amb moviment que seguís al personatge. Aquest canvi va portar també la decisió d'implementar un mini mapa, i tots els elements que l'acompanyen.

Implementació del videojoc

En aquesta secció vàrem implementar els diferents elements de la interfície d'usuari del jugador:

- Barra de vida
- Comptador de monedes
- Icona menú de pausa
- Cooldown de les habilitats
- Comptador d'enemics derrotats
- Generador d'enemics aleatori

Implementació dels menús

En aquest apartat es van desenvolupar tots els menús del videojoc, així com els diferents texts de la primera pantalla, el tutorial.

Cerca de efectes sonors

Cerca dels elements de so a utilitzar pels diferents elements del videojoc: efectes de so i banda sonora.

Verificació i proves

Es realitzen proves per garantir el correcte funcionament dels diversos elements del joc, i es revisa el projecte per assegurar-se que compleix amb tots els requisits establerts en la fase de plantejament de joc i disseny.

Creació d'una demo del videojoc

Després de realitzar totes les proves necessàries, es va crear una demo del videojoc que inclou tots els elements desenvolupats fins a aquest punt.

Documentació

Pel que fa a la documentació, aquesta és una tasca contínua que s'ha de dur a terme durant tot el projecte i que es presenta al final d'aquest. Per això, la seva durada abasta des del començament fins al final del projecte.

– 4. 3 TEMPS ESTIMAT I REAL -

El projecte estava planejat per començar a inicis del mes d'octubre i tenir-lo finalitzat el 2 de juny. Seguidament, s'adjuntaran dues figures, on es mostra el temps estimat per cada tasca (figura 4.2) i el temps real (figura 4.3).

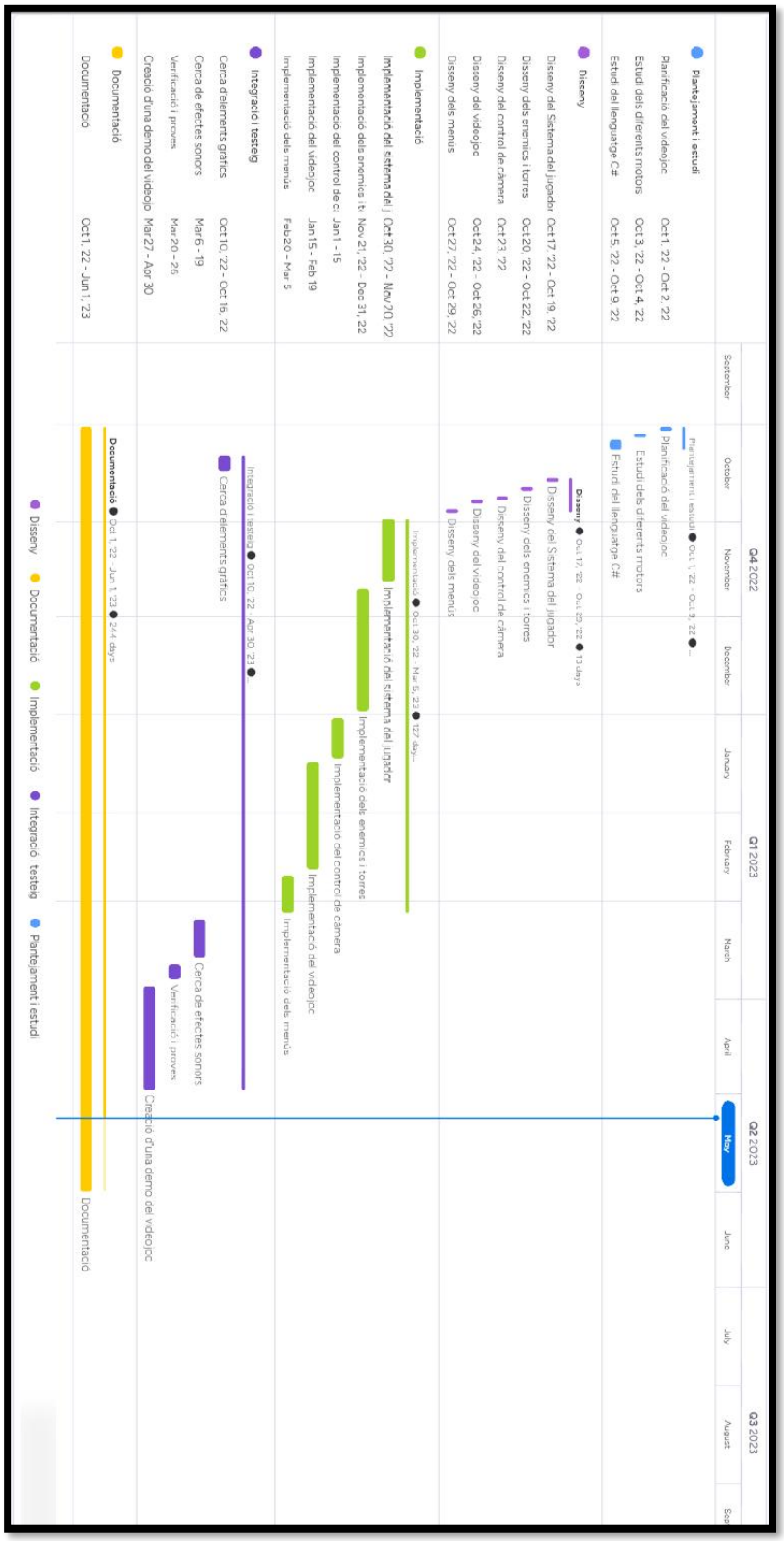

Figura 4.2 Diagrama de Gantt: Temps estimat

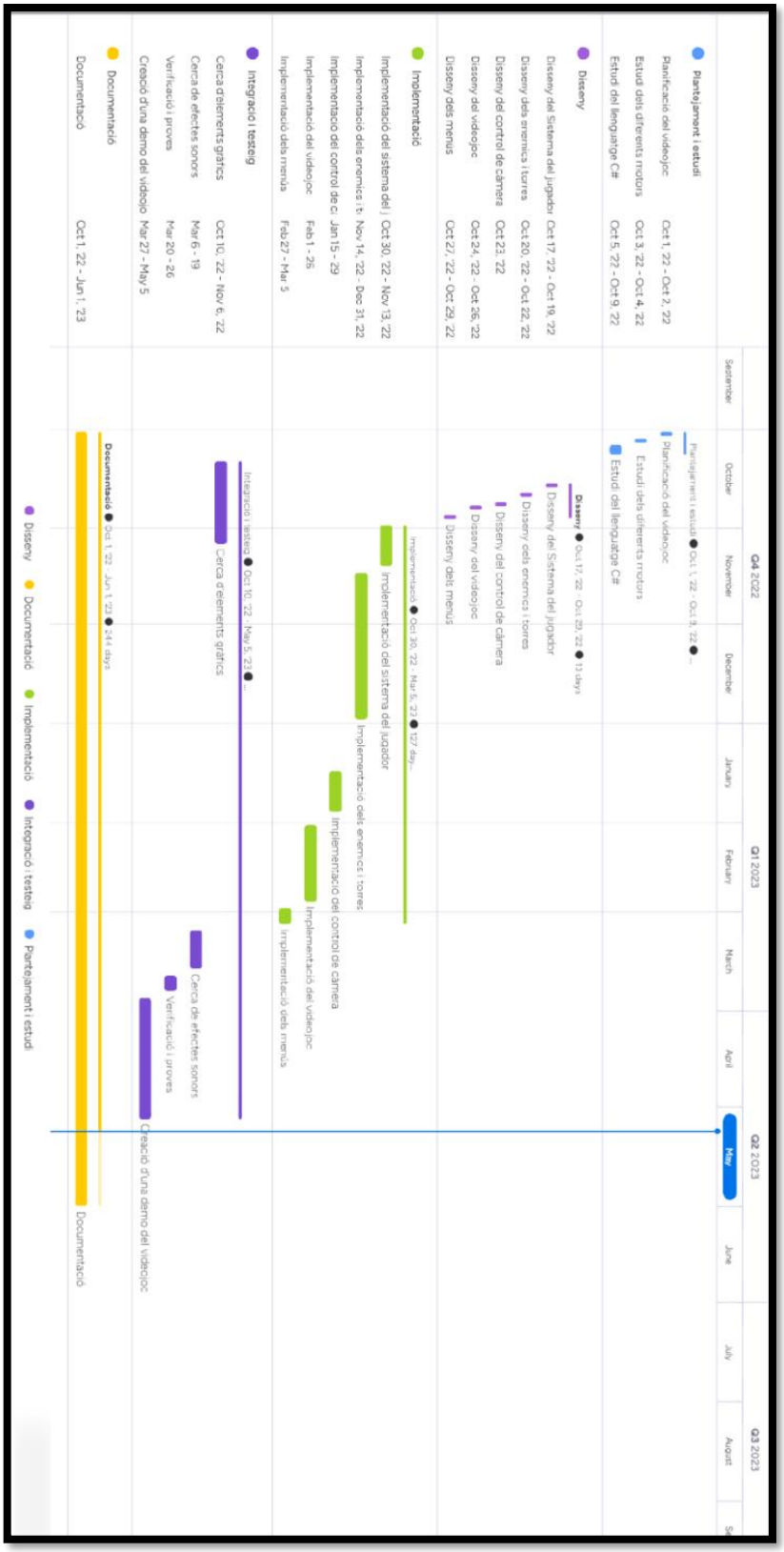

Figura 4.3 Diagrama de Gantt: Temps real

– 4. 4 RESULTATS ESPERATS -

S’espera poder crear una demo totalment funcional del que podria ser un primer pas de l’evolució dels videojocs clàssics Tower Defense. També s’espera que compleixi amb totes les premisses i expectatives presentades al principi del projecte.

CAPÍTOL 5

- 5. MARC DE TREBALL I CONCEPTES PREVIS -

El grau de complexitat en el desenvolupament d'un videojoc pot variar des de relativament senzill fins a molt complicat, depenent de si és necessari crear la infraestructura i l'entorn de desenvolupament juntament amb el joc, o si es fa servir un entorn dissenyat específicament per a aquesta tasca.

El mercat del desenvolupament de videojocs ofereix una gran varietat d'eines d'entorn de desenvolupament, conegudes com a motors de videojocs.

Aquests motors són la principal eina utilitzada per als desenvolupadors de videojocs i permeten la creació, disseny i representació dels jocs, amb moltes funcionalitats com ara la renderització, física, col·lisió, so, lògica de codi, animació, intel·ligència artificial, xarxa i transmissió en temps real, entre d'altres.

El motor de videojoc està format per diversos motors més petits, entre els quals es troben el motor gràfic i el motor físic:

- El motor gràfic es dedica a la part visual del joc, generant imatges sintètiques amb informació espacial i visual.

- El motor físic, per la seva banda, s'encarrega de simular les lleis de la física per recrear accions realistes, fent servir variables com la gravetat, la massa, la fricció, la força i la flexibilitat.

El principal objectiu d'un motor de videojocs és facilitar la tasca dels desenvolupadors, fent que puguin abstreure's de la implementació a baix nivell. Això és essencial per al desenvolupament de videojocs multiplataforma, ja que l'abstracció del maquinari permet als jocs funcionar en diferents dispositius.

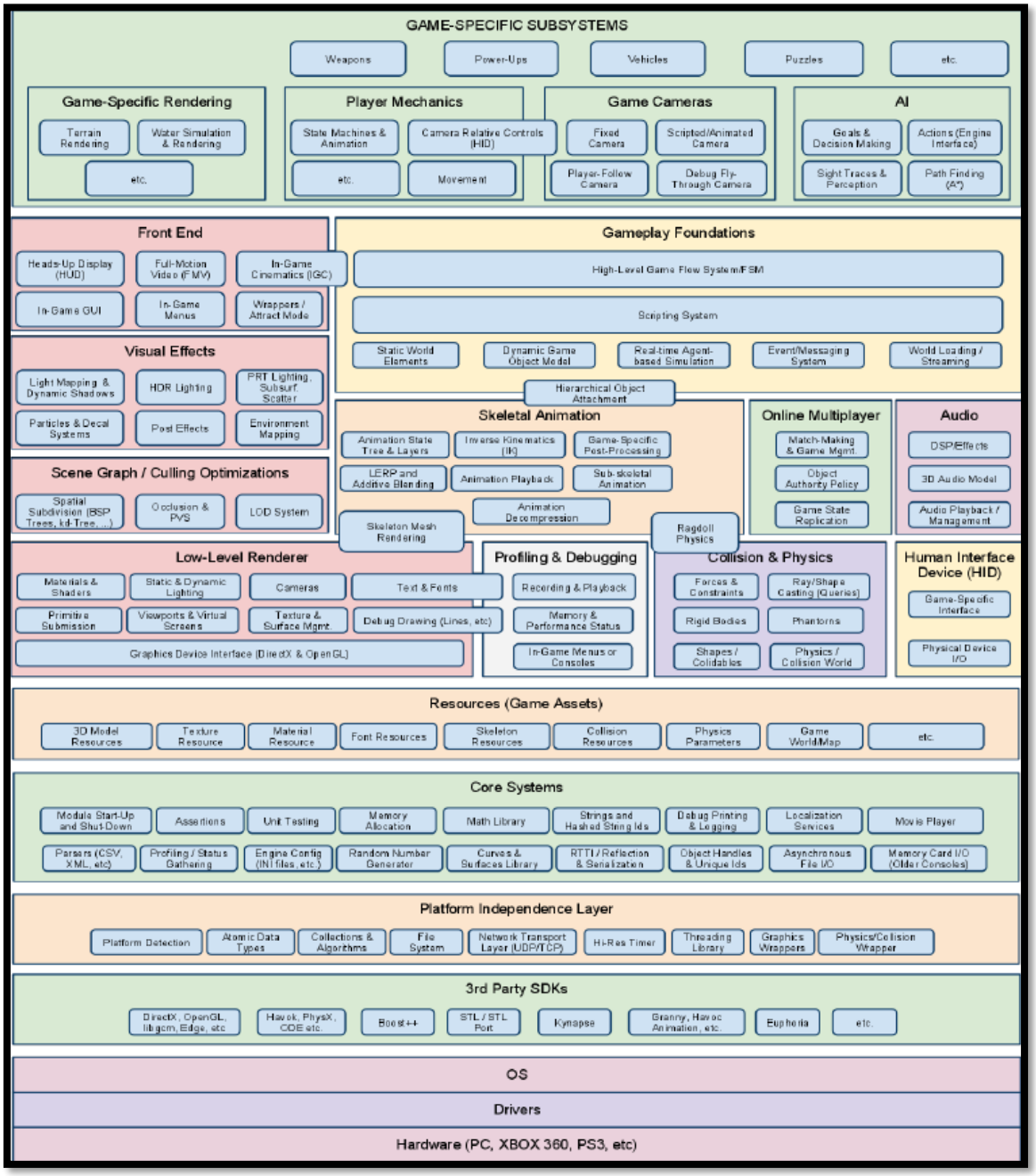

Figura 5.1 Esquema d'un motor de videojocs.

- 5.1 INTRODUCCIÓ MOTORS DE VIDEOJOCS -

Unity

Unity és un motor gràfic àmpliament utilitzat en el desenvolupament de videojocs, però també es pot utilitzar per a altres projectes que requereixen gràfics, com aplicacions, escenaris, arquitectura i realitat augmentada. El programari proporciona una àmplia gamma de funcions, com la renderització de gràfics 2D i 3D, la simulació de lleis físiques, les animacions, els sons, la intel·ligència artificial, la programació o el scripting.

Els avantatges d'Unity són nombroses. És fàcil d'emprar, amb una gran quantitat d'eines a la disposició de l'usuari. També disposa d'una botiga d'actius (Asset Store) on es poden adquirir actius i recursos de forma ràpida i senzilla, tant de pagament com gratuïts, el que ajuda a estalviar temps i diners.

A més a més, Unity és multiplataforma, el que significa que els projectes creats es poden publicar en diferents sistemes operatius, consoles i dispositius. Un altre avantatge és que el programari és completament gratuït per als usuaris, amb la possibilitat de pagar un petit percentatge si es ven el projecte per una quantitat determinada.

No obstant això, també hi ha desavantatges. Els projectes en Unity poden arribar a ser molt grans, la qual cosa ocupa molt d'espai en el disc dur. Això pot provocar problemes a l'hora de moure els fitxers, i si es volen moure molts fitxers, el procés pot trigar molt de temps. A més a més, el rendiment d'Unity pot ser un problema, ja que pot arribar a sobrecarregar el processador i fer que l'ordinador se sobreescalfi.

Finalment, la gestió de les versions pot ser una mica complexa, pel fet que alguns canvis poden provocar errors o fer que alguna cosa deixi de funcionar.

Figura 5.2 Logo de Unity.

Unreal Engine

Unreal Engine és un motor de joc desenvolupat per la companyia Epic Games. Actualment, la versió més recent és la 5, i és un dels motors de joc més populars i utilitzats per usuaris i empreses. Funciona amb codi C++ i proporciona totes les eines necessàries per construir jocs o simulacions, així com per editar vídeo, àudio i renderitzar animacions.

Entre els avantatges de l'Unreal Engine, destaquen la capacitat per crear entorns en realitat virtual, la adaptabilitat a diferents plataformes (sistemes operatius, consoles, tauletes i telèfons), i les eines per crear escenaris en 3D amb gran detall. A més, igual que Unity, el preu és gratuït.

No obstant això, també té alguns desavantatges, com la gran quantitat d'espai que ocupen els projectes en el disc dur, la falta d'eines per crear jocs 2D i una comunitat més petita en comparació amb Unity.

Pel que fa als jocs famosos creats amb aquest motor, trobem títols com Fortnite, Gears of War, Bioshock Infinite i Mortal Kombat 11, entre altres.

Figura 5.3 Logo de Unreal Engine.

Game Maker

Game Maker és un motor de videojocs desenvolupat per YoYo Games. Aquesta eina permet crear jocs de manera senzilla sense la necessitat de tenir coneixements avançats de programació. El programa ofereix una interfície gràfica fàcil d'utilitzar que permet crear jocs en 2D i 3D mitjançant una varietat d'eines que inclouen la

creació d'escenaris, la importació de gràfics i sons, la configuració de físiques i comportaments d'objectes, la creació de lògica de joc, i la publicació del joc en múltiples plataformes.

Entre els avantatges de GameMaker es troben la interfície d'usuari amigable i la capacitat per crear jocs en 2D i 3D, amb suport per a diferents plataformes com ara Windows, macOS, Linux, Android, iOS, PlayStation, Xbox, Nintendo Switch, entre altres. A més, GameMaker és conegut per ser una eina molt ràpida i eficaç per al desenvolupament de jocs.

No obstant això, GameMaker també té algunes limitacions, com ara la falta de suport per a jocs en línia i la necessitat d'adquirir llicències per a la publicació dels jocs en algunes plataformes. També hi ha altres motors de videojocs amb més funcionalitats i opcions avançades.

Figura 5.4 Logo de Game Maker.

Godot

Godot és un motor de jocs lliure i de codi obert, que ofereix una ampla gamma de funcions per ajudar en el desenvolupament de jocs. El motor té una interfície gràfica d'usuari (GUI) i un llenguatge de programació pròpiament dit, GDScript, que es basa en Python. També admet altres llenguatges de programació com C #, C ++ i Visual Script.

Godot té una estructura de nodes, on els nodes són objectes que es poden afegir i organitzar per crear un joc. També hi ha molts recursos predefinits, com ara animacions, efectes de partícules i models 3D, que poden ajudar a accelerar el desenvolupament del joc.

El motor també té moltes funcions avançades, com ara el suport per a la creació de jocs de realitat virtual i augmentada, la física 2D i 3D, la simulació de fluids i el suport per a la xarxa. També es pot compilar per a diferents plataformes, com ara Windows, Mac, Linux, Android, iOS i consoles de jocs com la Nintendo Switch.

En resum, Godot és un motor de jocs amb moltes funcions i una interfície fàcil d'utilitzar, que permet als desenvolupadors de jocs crear jocs per a una àmplia varietat de plataformes.

Figura 5.5 Logo de Godot.

– 5. 2 MOTOR ESCOLLIT -

Figura 5.6 Logo de Unity.

El motor escollit per desenvolupar aquest projecte ha estat Unity pels següents motius:

- Gran comunitat de desenvolupadors: Unity té una gran comunitat de desenvolupadors que poden proporcionar suport, recursos i consells.

- Asset Store: Unity té una botiga d'assets amb una gran varietat de recursos, des de gràfics i models fins a música i efectes de so, que pots utilitzar en el teu joc.
- L'editor Unity, ofereix eines molt bones per desenvolupar videojocs 2D, com és el cas d'aquest projecte.
- Facilitat de programació: Unity fa servir el llenguatge de programació C#, que és relativament fàcil d'aprendre i té una gran quantitat de recursos i documentació per ajudar a programar els jocs.
- Conté una gran varietat de tutorials per començar a familiaritzar-se amb el motor.

CAPÍTOL 6

– 6. REQUISITS DEL SISTEMA -

En aquest capítol, establim els criteris necessaris que l'aplicació ha de satisfer. Aquests criteris inclouen tant els objectius que ha de complir com les funcions que ha de realitzar. Així mateix, es dividiran els criteris en dues grans categories: els requisits funcionals, que es refereixen a les tasques que el sistema ha de fer; i els requisits no funcionals, que inclouen restriccions en els recursos disponibles, seguretat, interfícies externes, la forma de desenvolupament del projecte, entre d'altres.

– 6.1 PERFIL DE L'USUARI -

Aquest videojoc no distingeix els usuaris de cap manera, és a dir, tots els jugadors tenen els mateixos privilegis. Per tant, hi haurà un actor que representarà tots els usuaris que vulguin fer servir el sistema.

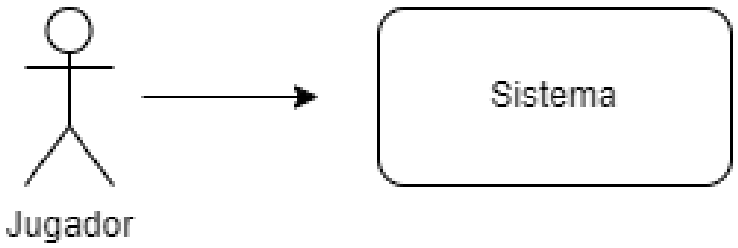

Figura 6.1 Interacció del jugador amb el sistema

– 6.2 REQUERIMENTS FUNCIONALS -

Els requeriments funcionals del videojoc són els següents:

- El jugador haurà de ser capaç de desplaçar-se i atacar amb les tecles del teclat i el ratolí.

- El jugador tindrà d'una interfície gràfica que li digui la situació actual i li proporcioni un minimapa de la seva posició i la dels enemics.

- En el moment d'iniciar el joc, el jugador tindrà la possibilitat de canviar la configuració de so per ajustar el volum de la música i dels efectes sonors segons les seves preferències.

- El jugador podrà instanciar diverses torres a qualsevol part del mapa.

- El jugador podrà derrotar als enemics del mapa i obtenir monedes d'ells.

- El jugador ha de poder veure la configuració de mapatge de tecles dintre del joc i en la pantalla d'inici.

- L'usuari ha de tenir la capacitat de sortir del joc en qualsevol moment, tant des del menú d'inici com des del menú de pausa del joc

- El jugador ha de poder veure la configuració de les tecles dintre del videojoc i en la pantalla d'inici.

– 6.3 REQUERIMENTS NO FUNCIONALS -

Els requisits no funcionals són aquells que descriuen les característiques i qualitats que el sistema ha de tenir en compte durant el disseny, com ara els recursos necessaris per al correcte funcionament i els nivells de seguretat necessaris per al sistema.

En aquest cas, el programa només serà utilitzat en mode d'usuari, de manera que no són necessàries mesures de seguretat addicionals per controlar l'accés al programa. A més, no es guardarà cap dada confidencial, de manera que no es necessita cap classe de protecció de dades.

Pel que fa a les restriccions de plataforma, el nostre videojoc s'ha creat amb un motor de videojocs, de manera que les restriccions de plataforma seran les que s'apliquin al motor.

Els requisits tècnics utilitzats per l'aplicació han estat majorment amb la maquinària mostrada a la taula 2.2, amb un rendiment molt satisfactori. Aquestes són les característiques:

- Sistema operatiu: Widnows 11
- Processador: AMD Ryzen 5 5600
- Memòria: 16 GB
- Gràfica: NVIDIA Geforce GTX 3060

Els requisits mínims per poder executar el videojoc, seran mínims del motor amb el qual ha estat desenvolupat, Unity:

- Sistema operatiu: Windows 7 SP1+, macOS 10.12+, Ubuntu 16.04, 18.04, i CentOS 7
- Processador: amb suport SSE2
- Memòria: 8GB
- Gràfica: DX10 (shader 4.0)

Cal destacar que els requisits poden variar depenent del tipus de projecte que es vulgui desenvolupar amb Unity. Si es planeja crear un joc 2D senzill, és possible que es pugui treballar amb especificacions més baixes. En canvi, si es vol treballar en projectes més grans i complexos, es necessitaran especificacions més altes per garantir un rendiment òptim del programari.

CAPÍTOL 7

– 7. ESTUDIS I DECISIONS -

En aquesta secció, detallarem les diferents llibreries i programes que s'han utilitzat durant la creació del projecte, així com el sistema operatiu en el qual s'ha desenvolupat el videojoc. Finalment, explicarem les eines que s'han usat per a la documentació del projecte.

– 7.1 SISTEMA OPERATIU -

El sistema operatiu utilitzat durant el desenvolupament del projecte ha estat el Windows 11 Home amb arquitectura x64.

Figura 7.1 Logo Windows 11

– 7.2 LLENGUATGE DE PROGRAMACIÓ -

El llenguatge de programació escollit és C#.

C# és un llenguatge de programació de propòsit general desenvolupat per Microsoft a finals dels anys 90 com a part de la plataforma .NET. Es tracta d'un llenguatge orientat a objectes que permet als programadors desenvolupar aplicacions per a diferents plataformes com ara Windows, MacOS, iOS, Android, Xbox, entre d'altres.

El motiu de seleccionar C# com a llenguatge de programació per al projecte és que, dels tres llenguatges suportats per Unity (Boo, UnityScript i C#), actualment C# és l'únic que és rellevant dins de l'ecosistema de desenvolupament de videojocs en Unity. De fet, els altres dos estan desaconsellats.

– 7.3 VISUAL STUDIO 2022 -

Visual Studio és un entorn de desenvolupament integrat (IDE, per les seves sigles en anglès) creat per Microsoft. Es tracta d'una eina de programació potent i completa que ofereix una ampla gamma de funcionalitats per al desenvolupament de software en diversos llenguatges de programació, incloent-hi C#.

Visual Studio ofereix moltes funcions per als programadors, com ara la depuració de codi, la creació d'interfícies gràfiques d'usuari, l'edició de codi en temps real i la integració amb moltes eines i serveis externs. A més, Visual Studio té una gran comunitat de desenvolupadors que crea extensions i plugins per a l'IDE que permeten estendre les seves funcions i personalitzar-lo segons les necessitats dels desenvolupadors.

S'ha optat per utilitzar aquesta eina com a principal entorn de programació del projecte per la simplicitat i l'estreta relació amb el motor de jocs Unity. Això permet una transició fluida entre ambdues eines sense preocupar-se per possibles problemes de compatibilitat o interrupcions innecessàries.

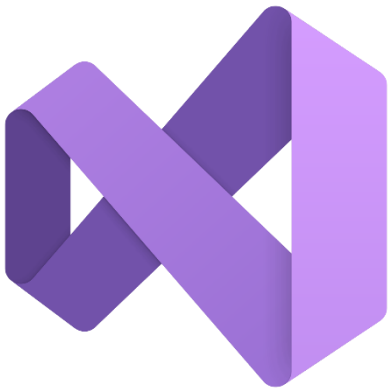

Figura 7.2 Logo Visual Studio

Finalment, cal remarcar que existeix una versió gratuïta anomenada Visual Studio Community, que és la que s'ha usat pel desenvolupament d'aquest projecte.

– 7.4 UNITY -

Unity és l'element més important del desenvolupament d'aquest projecte, ja que ha estat el centre de la creació. Per saber-ne més de Unity, consultar els apartats 5.1 i 5.2, allà s'ha detallat les funcions i capacitats de l'eina en qüestió.

Ara ens centrarem en la distribució de l'editor d'Unity, les escenes, la física, la definició d'un objecte, la utilització de l'element Canvas, el sistema de Tiles, la càmera, entre d'altres.

-7.4.1 EDITOR -

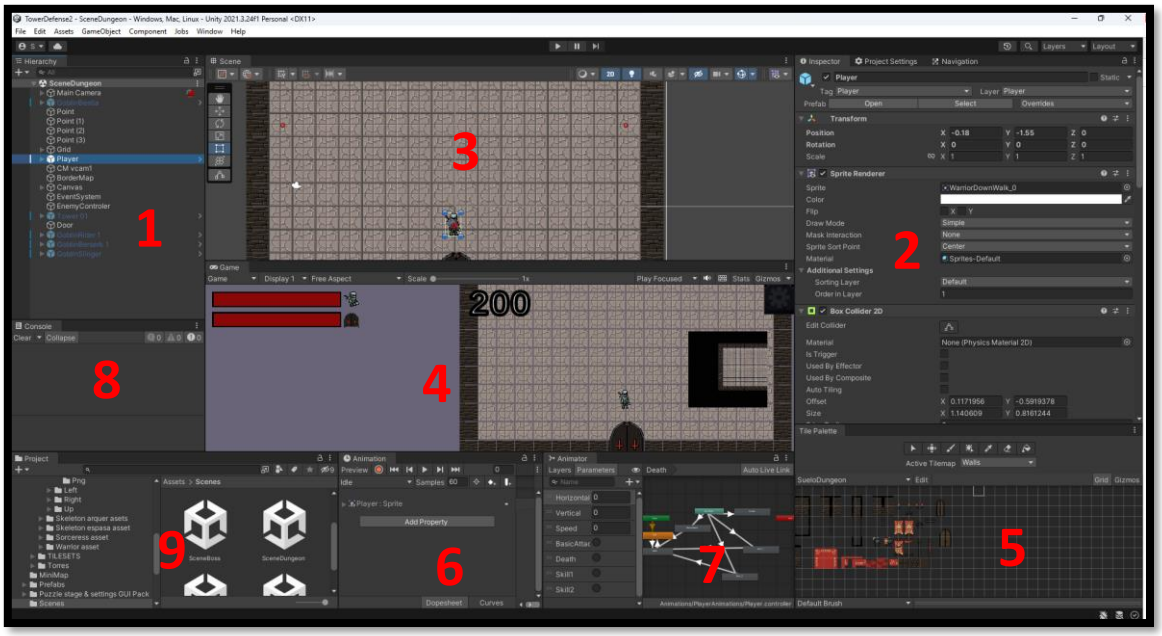

Figura 7.3 Editor d'Unity

L'editor d'Unity està compost per tantes finestres diferents que fa impossible esmentar-les totes en aquest apartat. Així que s'ha decidit anomenar les més importants o les més utilitzades pel desenvolupament d'aquest projecte.

1. Hirerachy

Aquesta finestra ens mostra tots els GameObjects, així com la jerarquia que puguin tenir aquests, de l'escena on estem treballant.

2. Inspector

L'Inspector és una finestra que es pot veure a l'Editor de Unity i que mostra les propietats dels objectes seleccionats a la jerarquia. Aquesta finestra és una eina fonamental per al desenvolupament de jocs en Unity, ja que permet als desenvolupadors i dissenyadors ajustar les propietats dels objectes i components del joc en temps real. És aquí on podrem associar els scripts creats en C# a GameObjects.

3. Scene

La Scene en Unity és l'àrea on es visualitzen i es manipulen els objectes 3D i 2D dels videojocs. És un entorn de treball en el qual els desenvolupadors poden construir i editar les diferents parts del joc, com ara la configuració dels nivells, la posició dels personatges, els elements interactius, els menús, etc.

4. Game

Mentre s'esta creant un videojoc, en la Scene es pot anar editant i modificant l'escena, però poden haver-hi moltes coses que, després en la jugabilitat, no es vegin. Aquesta pestanya mostra el que veurà el jugador.

Quan has acabat d'editar pots executar el joc i testejar a través d'aquesta pestanya.

5. Tilemap

Aquesta és una eina que es va haver d'instal·lar en forma de plug-in pel desenvolupament del projecte, ja que no ve per defecte amb Unity. L'eina permet tenir diferents paletes de Tiles. Els Tiles són especials perquè es poden posar sobre components Tilemap, que fa de llenç. D'aquesta manera podem pintar sobre l'escena quadrat a quadrat i dissenyar el contingut de l'escena de manera molt ràpida i senzilla.

6. Animation

La majoria d'elements del videojoc són sprites, és a dir que són un conjunt d'imatges que juntes formen un moviment. Per ajuntar aquestes imatges i crear una animació (d'atac, caminar,etc.) es va utilitzar aquest component, al qual permet unir imatges

i modificar la velocitat així com crear esdeveniments en diferents punts de l'animació.

7. Animator

Aquesta eina proporciona una manera de crear i controlar animacions en el joc a través de la definició de diferents estats i transicions entre ells. Els desenvolupadors poden usar aquesta eina per a configurar el comportament dels personatges o altres elements interactius del joc, i connectar les animacions d'una manera més eficient i eficaç.

8. Console

Mostra els missatges, avisos i errors que es produeixen durant l'execució del joc. Aquesta finestra és una eina fonamental per al desenvolupament de jocs en Unity, ja que ajuda als desenvolupadors a depurar el codi del joc i a identificar possibles problemes que puguin sorgir durant l'execució.

9. Project

Aquesta finestra mostra tots els elements emprats en el projecte separats per carpetes.

-7.4.2 CONCEPTES IMPORTANTS -

Per poder seguir sense problemes els següents capítols de la memòria és important tenir coneixement dels diferents conceptes que s'explicaran a continuació.

GameObject

És l'element bàsic utilitzat per a construir el món virtual d'un joc. Es pot considerar com una entitat o objecte que representa una part del joc, com ara un personatge, un objecte 2D/3D, un terreny o un punt de llum.

És a dir, es tracta de qualsevol element que hi hagi en una escena i aquests es defineixen pels diferents Components i les seves configuracions.

Prefab

Si es vol fer servir un GameObject en més d'una escena, es pot guardar en el disc per usar-lo en una altra escena, així no s'ha de crear el mateix GameObject un altre cop. Aquest objecte rep el nom de Prefab.

Un altre avantatge, a part d'optimitzar l'eficiència i el temps, és que cada còpia de cada Prefab que es fa servir a les diferents escenes, pot tenir diferents valors (vida, velocitat, atac, etc.). Donant molta flexibilitat a l'hora de crear videojocs.

Component

Els components agrupen tots els atributs que un GameObject pot tenir. Poden ser components que ja venen per defecte a Unity com d'animació, so, elements 2D/3D, etc, o elements creats pel desenvolupador, en forma de scripts. Seguidament en aquest apartat es veuran alguns dels components més usats en el projecte.

- **Transform**

  És l'únic component que és obligatori que tinguin tots els GameObjects, ja que indica quina és la posició, rotació i escala en l'escena que estem treballant.

- **Collider**

  Un Collider és un component d'Unity que permet saber quan dos objectes estan en contacte, o se superposen en l'espai. Existeixen dues classes de Colliders: els que avisen quan dos objectes han entrat en contacte, i els que no deixen passar a través d'ells. També poden ser els dos al mateix temps.

  Aquests es poden modificar i fer que agafin, per exemple, la silueta d'un arbre, o les cames del personatge, etc, facilitant així el realisme de les col·lisions en un videojoc.

- **Rigidbody**

  El Rigidbody és un element d'Unity que s'usa per fer que un element sigui afectat per la configuració de la física del motor. Aquest component en permet conferir una massa, fricció i més característiques físiques als elements. També permet restringir per quin dels eixos del món l'element es podrà desplaçar i sobre quins eixos podrà girar.

Per exemple en aquest projecte, al ser un videojoc 2D en vista zenital, s'ha desactivat l'eix de les Z en tots els elements. Normalment, els Rigidbody van associats amb almenys un Collider.

Tag

Els diferents GameObjects poden tenir associats una etiqueta. D'aquesta manera és més fàcil poder identificar els objectes dintre l'escena. Per exemple es pot tenir un tag "Enemic" i un altre "Jugador", així a l'hora de programar és més fàcil diferenciar si l'element trobat és un enemic o un jugador.

Layer

Seguint la mateixa idea dels Tags, de poder dividir els objectes en grups/etiquetes. El Layer el que fa és dividir els GameObjects en diverses capes. Això permet decidir què renderitza en l'escena, o per evitar certs colliders segons correspongui.

Layer Collision Matrix

Com s'ha comentat en el punt anterior, es pot evitar la col·lisió entre dos objectes gràcies al component Layer. Per fer-ho, s'usa la Layer Collision Matrix que te Unity. Com es pot veure a la Figura 7.4, entre Player i Bullet no hi ha col·lisions, mentre que entre Player i Enemy si que n'hi ha.

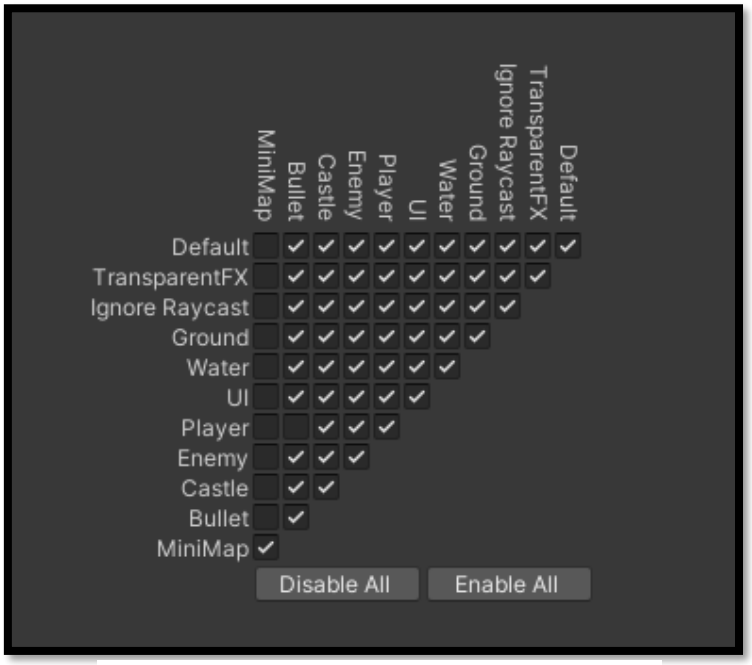

Figura 7.4 Matriu de col·lisions de capes

Camera

Dispositiu que fa la funció d'una càmera de vídeo en l'escenari, i des de la que el jugador veurà el videojoc. Aquest element és el punt d'unió entre el jugador i el videojoc, ja que serà a través de la càmera des d'on el jugador podrà veure i interactuar amb el videojoc.

Cine Machine

Un altre component que no ve per defecte a Unity i que, per tant, s'ha hagut d'instal·lar.

Es pot utilitzar per a crear seqüències de càmeres cinemàtiques en jocs i altres projectes interactius. Aquest paquet d'eines permet als desenvolupadors crear una varietat d'efectes de càmera, transicions, moviments i altres efectes visuals per a les seves escenes, tot sense necessitat de programar cada efecte de forma individual.Ha estat molt útil, ja que ha estalviat la feina d'implementar una lògica a la càmera per tal que segueixi al jugador i sempre mostri el contingut de les escenes sense sortir-se dels límits.

Audio Source

Component que ens permet reproduir àudio a través d'un GameObject.

Audio Listener

Component que generalment es troba en el GameObject que fa de càmera, però no és obligatori. Permet sentir els sons generats pels diferents Audio Source.

Canvas

Component molt important, ja que permet crear tot el sistema d'interfícies gràfiques (barra de vida, mini mapa, comptadors de moneda, etc.) així com el sistema de menús del projecte.

Text Mesh Pro

Un component que no ve per defecte a Unity i que, per tant, s'ha d'instal·lar. Permet als desenvolupadors crear textos altament personalitzables i de gran qualitat per als jocs i altres projectes interactius. Aquesta eina proporciona una gran varietat de funcionalitats per a la creació de textos, com ara fonts personalitzables, ombres, efectes de transparència, efectes de dissolució i animació de text, entre altres.

MonoBehaviour

MonoBehaviour és una classe base de C# que s'utilitza per als scripts que s'adjunten als GameObjects. Aquesta classe proporciona una sèrie de mètodes i variables que permeten als scripts interactuar amb el GameObject i l'entorn, així com amb altres components i sistemes de la Unity.

MonoBehaviour també proporciona una sèrie de mètodes específics que es criden en moments determinats durant el cicle de vida del GameObject:

- Awake: Aquesta funció s'executa una vegada quan el GameObject s'inicialitza. Es fa servir per configurar les referències dels components, les variables o altres elements que necessitin ser inicialitzats abans de començar el joc. Awake() s'executa abans que qualsevol altre mètode del script, per la qual cosa és útil per a la inicialització de les variables que altres mètodes del script podrien necessitar.

- Start: Aquesta funció s'executa una sola vegada al començament del joc. S'usa per inicialitzar les variables o altres elements que no necessiten ser inicialitzats en el moment de la creació del GameObject, sinó que requereixen ser-ho en un moment posterior del joc. Start() s'executa immediatament després de la funció Awake().

- Update: Aquesta funció s'executa en cada fotograma del joc. Es fa servir per actualitzar el comportament del GameObject a cada fotograma. Això pot incloure la modificació de la posició, rotació, velocitat, acceleració, col·lisions, interaccions amb l'usuari, i altres elements que puguin canviar al llarg del joc. És important tenir en compte que Update() s'executa en cada fotograma, la qual cosa pot afectar al rendiment del joc si es fan operacions pesades o innecessàries en aquest mètode.

- OnDestroy: Aquesta funció s'executa quan el GameObject és destruït o es desactiva. S'utilitza per realitzar qualsevol tasca de neteja o alliberament de recursos que siguin necessàries quan el GameObject ja no estigui en ús. Això pot incloure la destrucció de referències, la cancel·lació de subscripcions, la desactivació de comportaments i altres elements que ja no siguin necessaris. És important tenir en compte que OnDestroy() no s'executa quan el joc es tanca, sinó només quan es destrueix o es desactiva el GameObject.

Scene Manager

És una classe d'Unity que fa un seguiment de les escenes del joc, permetent passar d'una a l'altre. Permet eliminar-les, carregar-les o descarregar-les en temps d'execució. Això proporciona una gran flexibilitat per al desenvolupament del joc, ja que permet canviar el contingut de les escenes del joc sense haver de sortir de l'aplicació.

– 7.5 BEHAVIOR TREES -

Els Behavior Trees són una tècnica popular en el desenvolupament de videojocs per a la intel·ligència artificial dels personatges no jugadors (PNJ). Aquesta tècnica es basa en la teoria de grafs per representar les decisions i accions que prenen els PNJ. En aquesta estructura, cada node representa una decisió o una acció que es pot prendre, i les arestes connecten els nodes per indicar el flux de la lògica del comportament. Així, els comportaments de les IA es modelen com un arbre que s'anomena Behavior Tree.

Un exemple molt simplificat el podem veure a la Figura 7.5, on tenim una condició "És visible X element?" i una acció "Patrullar". Si l'element X és visible llavors patrullarà, altrament es quedarà esperant.

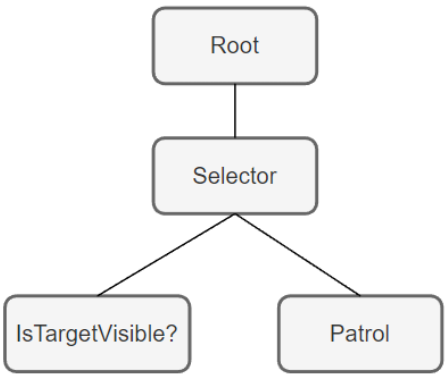

Figura 7.5 Exemple Behavior Tree

Els Behavior Trees tenen diversos avantatges en el desenvolupament de videojocs:

- Fàcil de comprendre i depurar: Els Behavior Trees són fàcils de llegir i comprendre, ja que l'estructura dels arbres reflecteix directament la lògica

del comportament dels PNJ. Això facilita la tasca de depuració dels errors i la detecció de comportaments inesperats.

- Adaptabilitat: Els Behavior Trees es poden adaptar fàcilment a diferents situacions del joc i a diferents tipus de PNJ. Per exemple, es poden canviar les prioritats dels comportaments, afegir o eliminar nodes, o modificar les condicions de les accions en temps d'execució. Això proporciona una gran flexibilitat per al desenvolupament dels comportaments de les intel·ligències artificials.

- Escalabilitat: Els Behavior Trees són escalables, el que significa que es poden crear arbres de comportament complexos per a un gran nombre de PNJ sense afectar el rendiment del joc. Això és important per als jocs que tenen moltes IA interactuant amb el jugador alhora.

- Separació de tasques: Els Behavior Trees permeten als desenvolupadors separar les tasques relacionades amb la intel·ligència artificial dels PNJ de la resta del joc, cosa que simplifica el desenvolupament i el manteniment del joc.

S'ha escollit aquesta arquitectura d'IA, pels avantatges esmentades anteriorment, però també perquè es volia aprendre una manera diferent de fer les intel·ligències artificials a com s'ha vist a la carrera.

– 7.6 DRAW.IO -

Draw.io és una eina de disseny gràfic en línia i gratuïta que permet crear diagrames, diagrames de flux de dades, diagrames de flux de processos, mapes mentals, organigrames, diagrames de xarxes, diagrames UML i altres tipus de diagrames. Es pot utilitzar per a la planificació de projectes, la documentació, la creació de presentacions, entre altres finalitats.

S'ha fet servir per fer tots els diagrames necessaris per a la documentació del projecte.

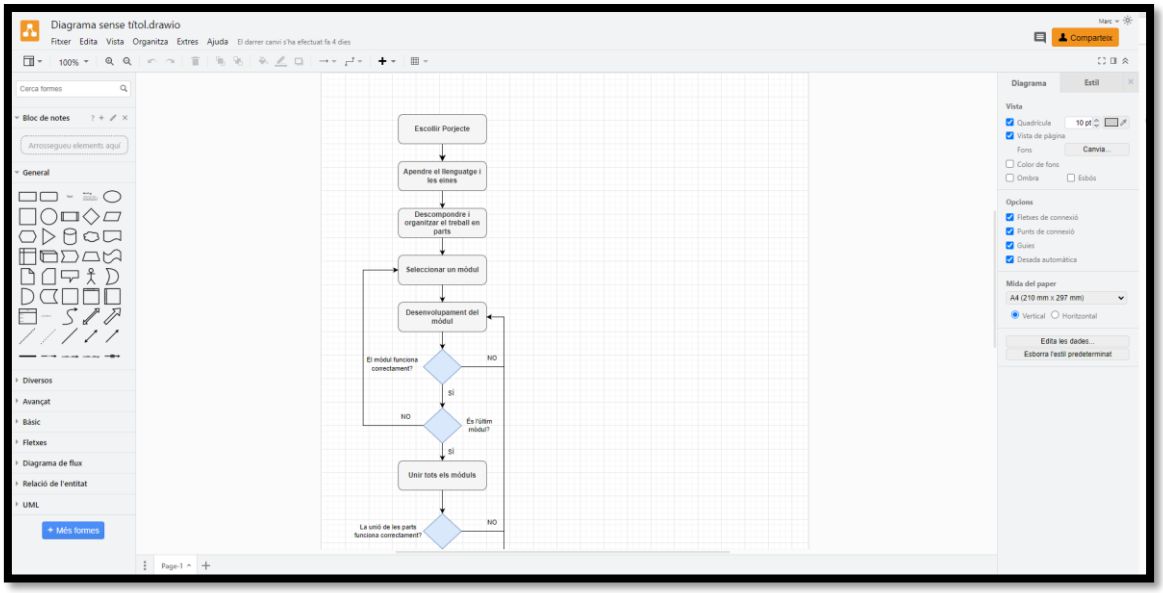

Figura 7.6 Creació de la figura 3.2 amb draw.io

– 7.7 MICROSOFT WORD -

Probablement un dels softwares més famosos del món per l'edició de textos. La raó per la qual es va decidir utilitzar-lo per dur a terme la redacció d'aquest document, és perquè la universitat ens proporciona una llicència legal i al ser tan complet, es va triar aquest software.

Figura 7.7 Logo de Microsoft Word

– 7.8 MONDAY.COM -

Monday.com és una plataforma de gestió de projectes i col·laboració en línia que permet als equips organitzar i gestionar les tasques, projectes i processos en un sol lloc. Amb Monday.com, els usuaris poden crear taulers personalitzables per visualitzar i gestionar les tasques, així com col·laborar amb altres membres de l'equip en temps real.

És de pagament, però té un període gratuït de 30 dies. Per tant, es van fer dues comptes, un per la Figura 4.2 a l'inici del desenvolupament del projecte i l'altre per la Figura 4.3 pel final del desenvolupament.

– 7.9 GITHUB -

Figura 7.8 Logo de GitHub

A l'inici del projecte no es va utilitzar cap mena de repositori, ni guardat de seguretat. A inicis d'any va haver-hi un contratemps que va desembocar amb la pèrdua de gran part del projecte, per aquesta raó la Figura 4.3, es pot veure on les primeres setmanes de Gener no hi ha cap tasca assignada, a causa que es van dedicar les dues setmanes a refer la feina perduda.

Els desenvolupadors poden usar GitHub per emmagatzemar i compartir el codi font, col·laborar en projectes amb altres desenvolupadors, contribuir al codi d'altres persones i fer un seguiment de les versions del mateix projecte. GitHub proporciona eines de gestió de versions de codi font i una plataforma de col·laboració que permet als desenvolupadors treballar junts en el mateix codi de manera eficient.

Per evitar una altra pèrdua es va decidir guardar el projecte a Github i anar fent pujades de noves versions, per evitar més problemes.

CAPÍTOL 8

– 8. ANÀLISI I DISSENY DEL SISTEMA -

En aquest capítol s'aporten les solucions implementades per a donar vida al projecte, abordant els requisits funcionals detallats en el Capítol 6. Aquí, s'expliquen de manera conceptual les bases del videojoc, el que ens permet descriure com s'ha abordat cada problema i com s'ha trobat una solució per a cadascun d'ells.

– 8.1 DESCRIPCIÓ GENERAL -

Els videojocs de "Tower Defense" són un gènere popular de jocs d'estratègia que es concentren en la defensa d'una àrea contra hordes d'enemics que s'apropen en ones successives. El jugador ha de construir i col·locar torres de defensa en punts estratègics per evitar que els enemics arribin a la seva destinació. Al tractar-se d'un videojoc que és una combinació de jocs d'estratègia i jocs de puzzle, requereix que el jugador faci una planificació, estratègia i pensament crític per aconseguir superar els nivells i derrotar els enemics. Un Tower Defense s'acaba quan l'usuari s'ha passat totes les pantalles del videojoc.

En la Figura 8.1 podem veure un exemple de Tower Defense, on s'ha d'evitar que els enemics arribin al final del camí.

En el projecte s'ha buscat innovar la fórmula clàssica i mostrar una demo del que podria ser una evolució dels Tower Defense. El jugador haurà de defensar una porta dels enemics. Per fer-ho, controlarà un personatge amb el qual haurà d'anar caminant pel mapa i atacant als enemics amb les seves habilitats. Al tractar-se d'un Tower Defense, podrà construir torres en qualsevol punt del mapa per ajudar-se. Un cop derroti a tots els enemics passarà a la pantalla final on haurà de guanyar a un Boss (jefe final). Tots els enemics han estat generats amb intel·ligència artificial, concretament amb Behavior Trees.

Per diferenciar-nos encara més dels diferents Tower Defense, les onades d'enemics no estaran pre-programades, d'aquesta manera cada cop que es jugui, els enemics apareixeran en diferent ordre. Això farà que rejugar el videojoc sigui una experiència diferent, agafant una mica l'essència dels videojocs *roguelike*.

Figura 8.1 Exemple Tower Defense clàssic

– 8.2 PERSONATGE PRINCIPAL -

El personatge principal (Figura 8.2) és l'element més important del projecte, ja que és la forma amb la qual el jugador interactua amb el videojoc. Aquest és un cavaller medieval al qual el jugador controlarà al llarg de les tres pantalles. Les accions que pot realitzar són les següents: atacar a un enemic amb l'espasa, congelar i fer mal en àrea, esquivar (tornar-se intocable), construir diverses torres i caminar lliurement pel mapa.

Figura 8.2 Personatge principal

– 8.3 LA PORTA -

L'objectiu del jugador és protegir la porta, per evitar que els enemics ataquin als habitants del castell. Aquesta porta (Figura 8.3) té certa quantitat de vida, que si

arriba a zero, es perd la partida. En tot moment es podrà veure la vida restant de la porta.

Figura 8.3 Element a protegir

– 8.4 ENEMICS -

Els enemics volen destruir la porta que protegeix el jugador, fent així que aquest perdi la partida. Per tant, l'objectiu serà eliminar tots els enemics. Quan s'elimina un enemic, aquest ens donarà monedes per poder construir torres. Hi ha 4 tipus d'enemics bàsics i 1 que serà el jefe final del videojoc. Tots estan controlats per arbres de comportament.

Goblin Berserk

És l'enemic que fa menys mal de tots, però també el més ràpid. Prioritza el jugador sobre tots els elements del mapa, pel que, si s'apropa molt al jugador, el perseguirà i l'atacarà fins que es torni a allunyar. La següent prioritat seran les torres i per acabar la porta. En conseqüència, el jugador haurà de ser ràpid i cridar la seva atenció perquè no s'apropi a la porta. Veure Figura 8.4.

Figura 8.4 Enemic goblin berserk

Goblin Beast

Aquest és el més lent de tots, però també el que fa més malt i el que té més vida. Té dues habilitats, utilitzarà una per atacar al jugador i l'altre per atacar a les torres. Aquesta última les trencarà d'un sol cop. Les prioritats d'atac són: primer les torres, seguit del jugador i finalment la porta. El jugador haurà d'intentar anar-lo congelant per evitar que ataqui a les torres. Veure Figura 8.5.

Figura 8.5 Enemic goblin beast

Goblin Slinger

És de tots els enemics el que té l'arbre de comportament més senzill, però això no vol dir el més fàcil de derrotar. Aquest enemic ignora completament el jugador i les torres, és a dir, va directa cap a la porta. Una altra particularitat que té respecte als altres, és que aquest enemic ataca a distància, per tant, no li fa falta arribar a la porta per començar a fer-li mal. Veure Figura 8.6.

Figura 8.6 Enemic goblin slinger

Goblin Rider

L'últim dels enemics de la segona pantalla. Té atac, velocitat i vida equilibrades. A més, ignora completament les torres. Per tant, el jugador haurà d'apropar-se per cridar la seva atenció i desviar-lo de la porta. Si el jugador es col·loca damunt seu l'empentarà cap amunt allunyant-lo de la porta. Veure Figura 8.7.

Figura 8.7 Enemic goblin rider

Skeleton King

Finalment el jefe final del videojoc. Aquest enemic té tres habilitats diferents. La primera és un atac cos a cos que fa força mal, el qual el jugador haurà d'intentar esquivar movent-se o amb l'habilitat de fer-se invulnerable. El segon atac, que dispara un projectil, s'activa quan el jugador s'allunya molt d'ell. Per acabar, un cop el jugador li hagi tret la meitat de la vida, aquest utilitzarà la tercera habilitat, que provocarà una pluja de foc pel mapa, que no pararà fins que acabi el combat. Veure Figures 8.7, 8.8 i 8.9.

Figura 8.7 Jefe final

Figura 8.8 Projectil Jefe final

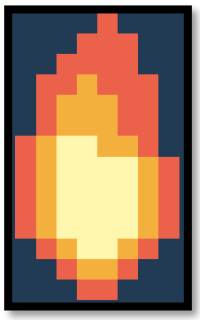

Figura 8.9 Bola de foc

– 8.5 TORRES -

Les torres són els elements que ens ajudaran a fer front a les onades d'enemics. En aquest videojoc s'han desenvolupat tres torres diferents. Totes les torres tenen el mateix funcionament, ataquen al primer enemic que veuen, d'aquesta manera, si el jugador vol que una torre determinada ataqui a un altre enemic, primer haurà de matar al que l'estructura estigui atacant.

Les torres es componen de tres components:

- La base: és l'element que té associada la vida de la torre i el collider2D, per tant, rebrà el mal dels enemics.
- L'arma: situada damunt de la torre, aquesta s'encarrega de buscar l'enemic i invocar el projectil quan estigui dintre el rang. Anirà rotant, seguint la posició de l'enemic seleccionat.
- El projectil, és l'element que atacarà directament a l'enemic. Un cop invocat, perseguirà a l'enemic que li han assignat, fent que els projectils invocats per les torres encertin sempre l'objectiu. Un cop interaccioni amb l'enemic, el GameObject serà destruït.

Ara es mostraran els components de cada torre.

Torre d'arquers

Aquesta torre ataca amb fletxes ràpides als enemics. Fan poc malt, però la torre té una cadència de tir força elevada. Veure Figures 8.10, 8.11, 8.12 i 8.13.

Figura 8.10 Base torre arquers

Figura 8.11 Arma torre arquers

Figura 8.12 Projectil torre arquers

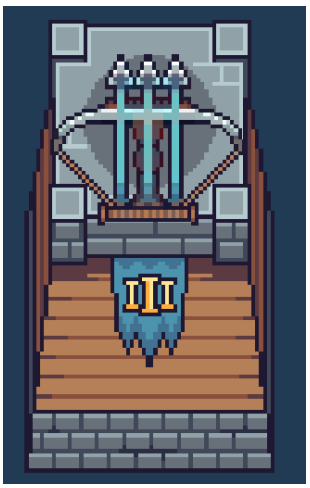

Figura 8.13 Torre arquers

Torre de verí

La segona torre de l'arsenal farà més mal als enemics quan tinguin més vida, és a dir, fa un 10% de la vida màxima de l'enemic. Per tant, en deu dispars qualsevol enemic morirà. Un cop impacti un projectil d'aquesta torre, els enemics es quedaran amb una tonalitat verdosa, per ajudar al jugador a indicar ràpidament quin és l'enemic que està essent atacat per aquesta torre. Veure Figures 8.14, 8.15, 8.16 i 8.17.

Figura 8.14 Base torre de verí

Figura 8.15 Arma torre de verí

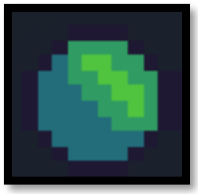

Figura 8.16 Projectil torre de verí

Figura 8.17 Torre de verí

Torre màgica

Finalment, aquesta ataca als enemics cos a cos, per tant, no té projectil. Aquesta torre és una arma de doble fil, quan el primer enemic se li apropi, aquesta torre atacarà en àrea i ja no pararà mai fins que sigui destruïda. Aquest atac màgic en àrea, farà mal tant a enemics com al jugador. Veure Figures 8.18, 8.19 i 8.20.

Figura 8.18 Base torre màgica

Figura 8.19 Arma torre màgica

Figura 8.20 Torre màgica

– 8.6 INTERFÍCIES D'USUARI -

En el videojoc es poden trobar un seguit d'interfícies d'usuari, en total de 7 menús diferents.

Menú principal

Menú inicial que es trobarà l'usuari a l'obrir el joc i que permetrà anar al menú d'opcions (botó OPTIONS) i de controls (botó CONTROLS). El botó PLAY portarà al jugador a la primera pantalla (la del tutorial), el botó QUIT tancarà el videojoc. Veure Figura 8.21.

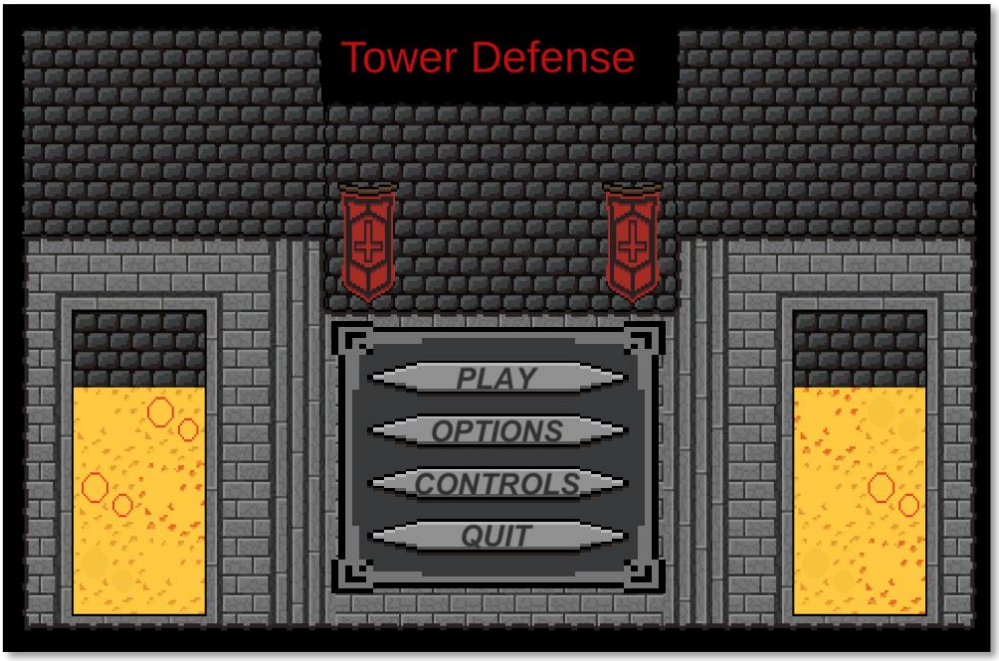

Figura 8.21 Menú principal

Menú d'opcions

El menú d'opcions permetrà a l'usuari poder regular el volum des de la barra de progressió vermella. El jugador també podrà decidir si jugar el joc en pantalla completa o no des del botó Full Screen. Per tornar al menú principal haurà de prémer el botó BACK. Veure Figura 8.22.

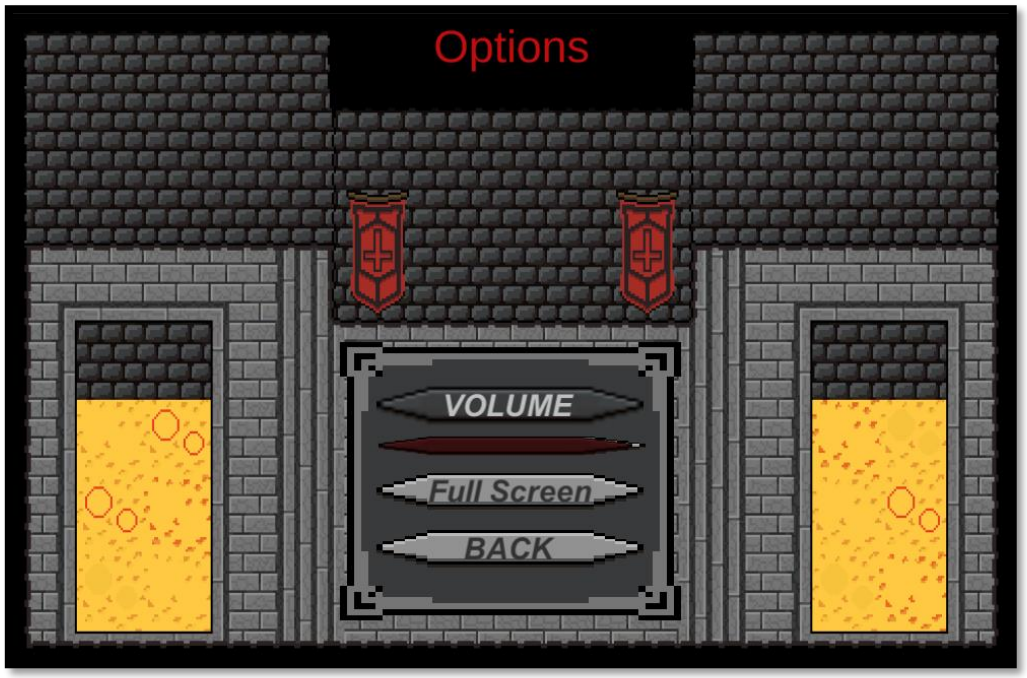

Figura 8.22 Menú d'opcions

Menú de controls

Aquest menú s'hi pot accedir des de qualsevol pantalla. Mostra al jugador la configuració de tecles, en cas que ho necessiti consultar. Cal remarcar que en la primera pantalla es fa un tutorial per familiaritzar-se amb aquestes configuracions. Veure Figura 8.23.

Figura 8.23 Menú de controls

Menú pausa

En qualsevol moment de la partida el jugador podrà parar el joc i accedir a un menú per poder fer diferents accions (veure Figura 8.24). Per accedir aquest menú es podrà fer de dues maneres: premen la tecla ESC del teclat o clicant el botó virtual situat a dalt a l'esquerra de la pantalla (veure Figura 8.25). Les diferents accions que pot fer el jugador en aquest menú són les següents:

- RESUME: Tanca el menú pausa i retorna al joc en l'instant que estava. També pot tornar a prémer la tecla ESC per sortir del menú.
- RESTART: Reinicia la pantalla actual en la qual es troba el jugador.
- CONTROLS: Accedeix al menú controls de la Figura 8.23.
- QUIT: Tanca el videojoc.

Figura 8.24 Menú pausa

Interfície de joc

És la interfície gràfica que utilitza el jugador al llarg de la partida, on es pot veure reflectit l'estat actual del jugador i els elements que l'envolten.

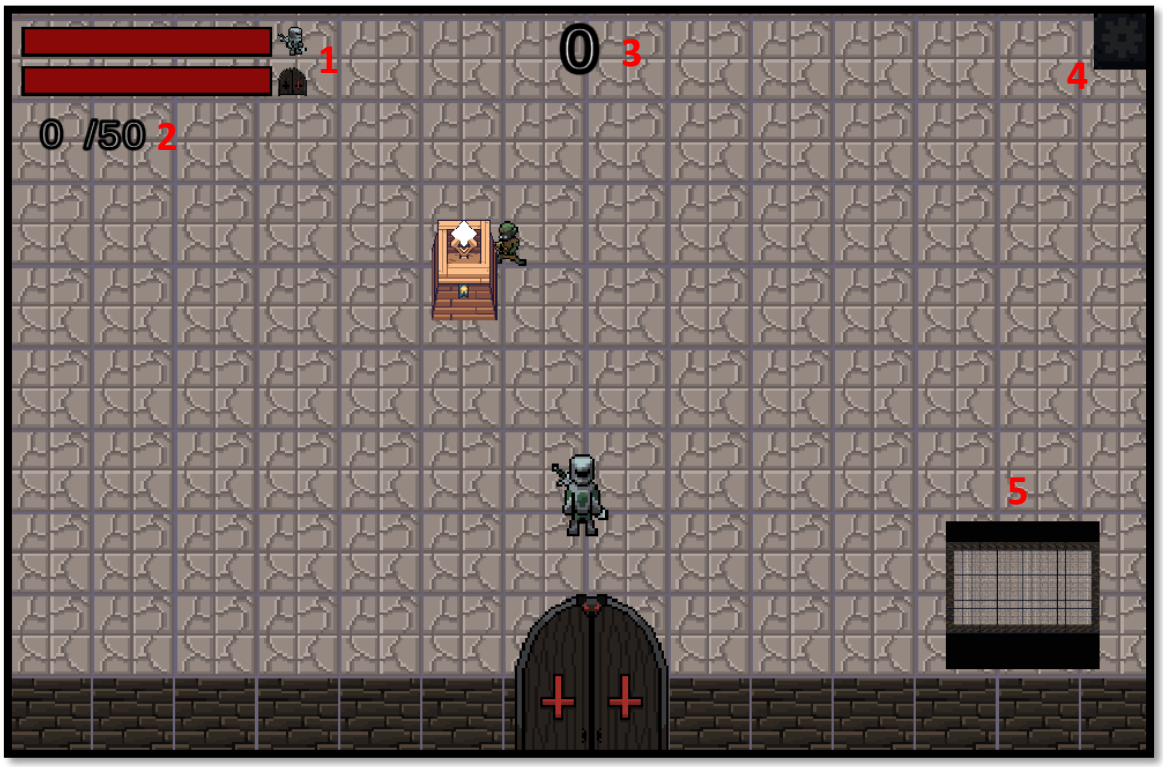

Figura 8.25 Menú de joc

A la Figura 8.25 es pot observar diferents elements:

1. Dues barres vermelles que corresponen a la vida del jugador i de la porta.
2. Un comptador (en aquest cas 0/50) que indica que el jugador ha matat 0 dels 50 enemics necessaris per passar de pantalla.
3. Les torres costen recursos construir-les, aquí es mostra el nombre de recursos que té el jugador.
4. Botó per accedir al menú de pausa.
5. Minimapa on es mostra la posició actual del jugador i els enemics.

Alguns elements poden variar depenent de la pantalla en la qual es trobi, per exemple en la pantalla del jefe final no hi haurà un comptador d'enemics derrotats.

Menú de mort i de victòria

Si els enemics maten al jugador o a la porta, es mostrarà un menú que donarà l'opció de sortir del joc, reiniciar la pantalla o tornar al menú principal. Per altra banda, si el jugador guanya es mostrarà un menú semblant al de mort però en lloc de reiniciar,

passar a la següent pantalla. Si es tracta de l'última pantalla únicament podrà tornar al menú principal o sortir del joc. Veure Figures 8.26 i 8.27.

Figura 8.26 Menú de mort

Figura 8.27 Menú de victòria

– 8.7 CASOS D'ÚS -

En aquesta secció establirem les diferents activitats i conductes que es poden dur a terme en el videojoc mitjançant els casos d'ús. Un cas d'ús és una explicació il·lustrada dels passos que es realitzen entre un actor (en aquest cas el jugador) i el sistema com a resposta a un esdeveniment provocat per l'actor. Són descripcions de les funcionalitats del sistema independentment de la implementació.

A continuació, s'ha creat un diagrama de casos d'ús per a cada menú disponible en el joc, ja que en cada menú les accions que el jugador pot dur a terme són diferents.

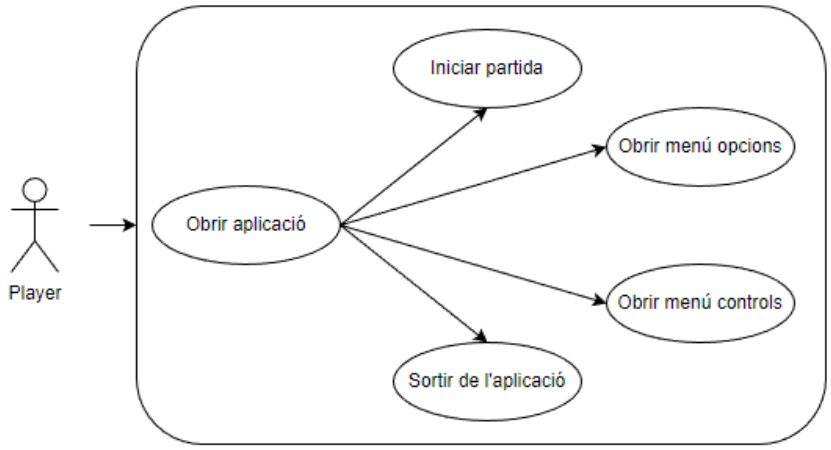

Figura 8.28 Diagrama de casos d'ús del menú d'inici

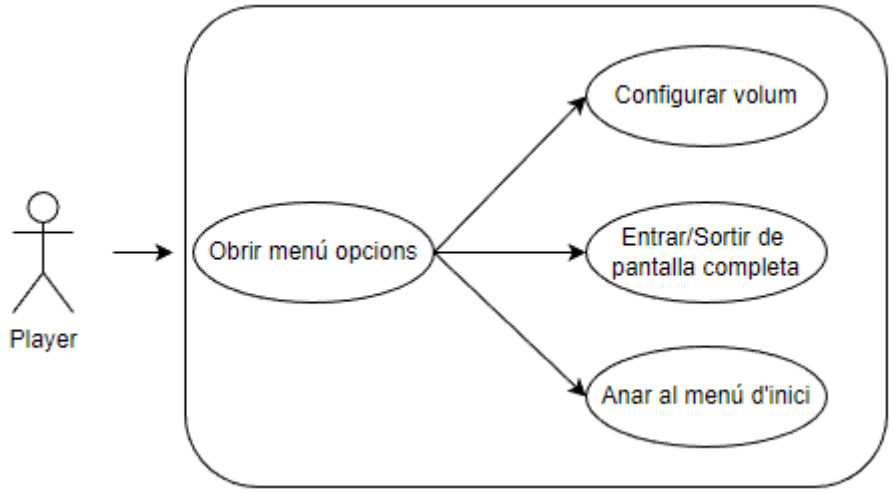

Figura 8.29 Diagrama de casos d'ús del menú d'opcions

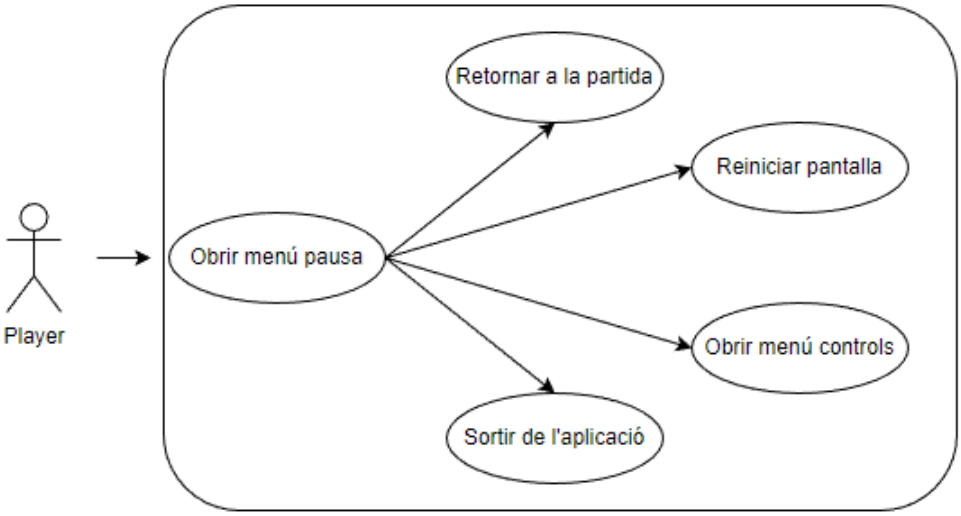

Figura 8.30 Diagrama de casos d'ús del menú de pausa

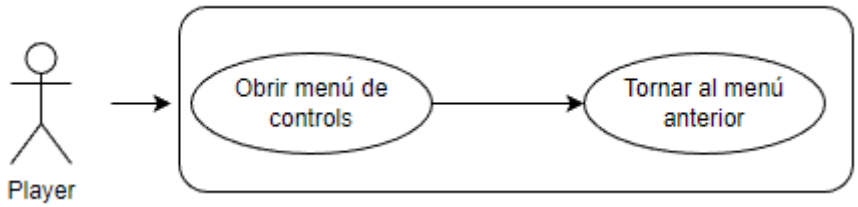

Figura 8.31 Diagrama de casos d'ús del menú de controls

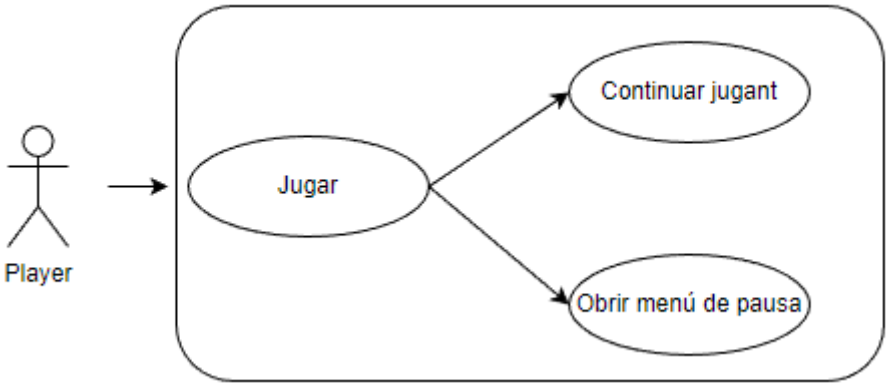

Figura 8.32 Diagrama de casos d'ús dintre del joc

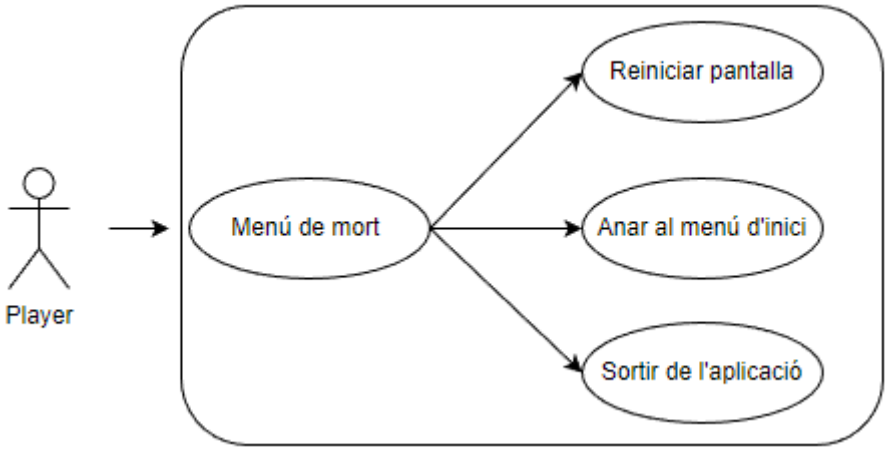

Figura 8.33 Diagrama de casos d'ús del menú de mort

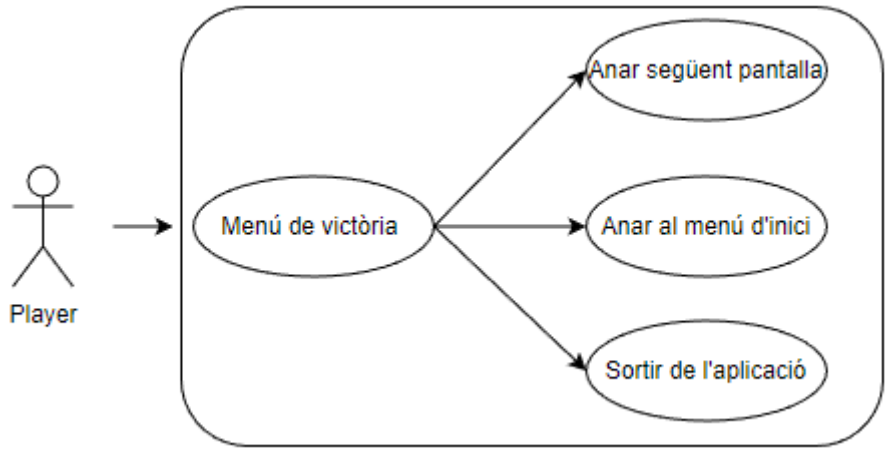

Figura 8.33 Diagrama de casos d'ús del menú de victòria

– 8.8 DIAGRAMES D'ACTIVITATS -

Els diagrames d'activitats són una eina valuosa per il·lustrar la seqüència d'activitats en un procés. En aquesta secció, es presentarà de manera general alguns dels processos essencials per la jugabilitat. A través d'aquests diagrames, podrem entendre millor com s'organitzen i interrelacionen les diverses activitats per proporcionar una jugabilitat coherent i fluida.

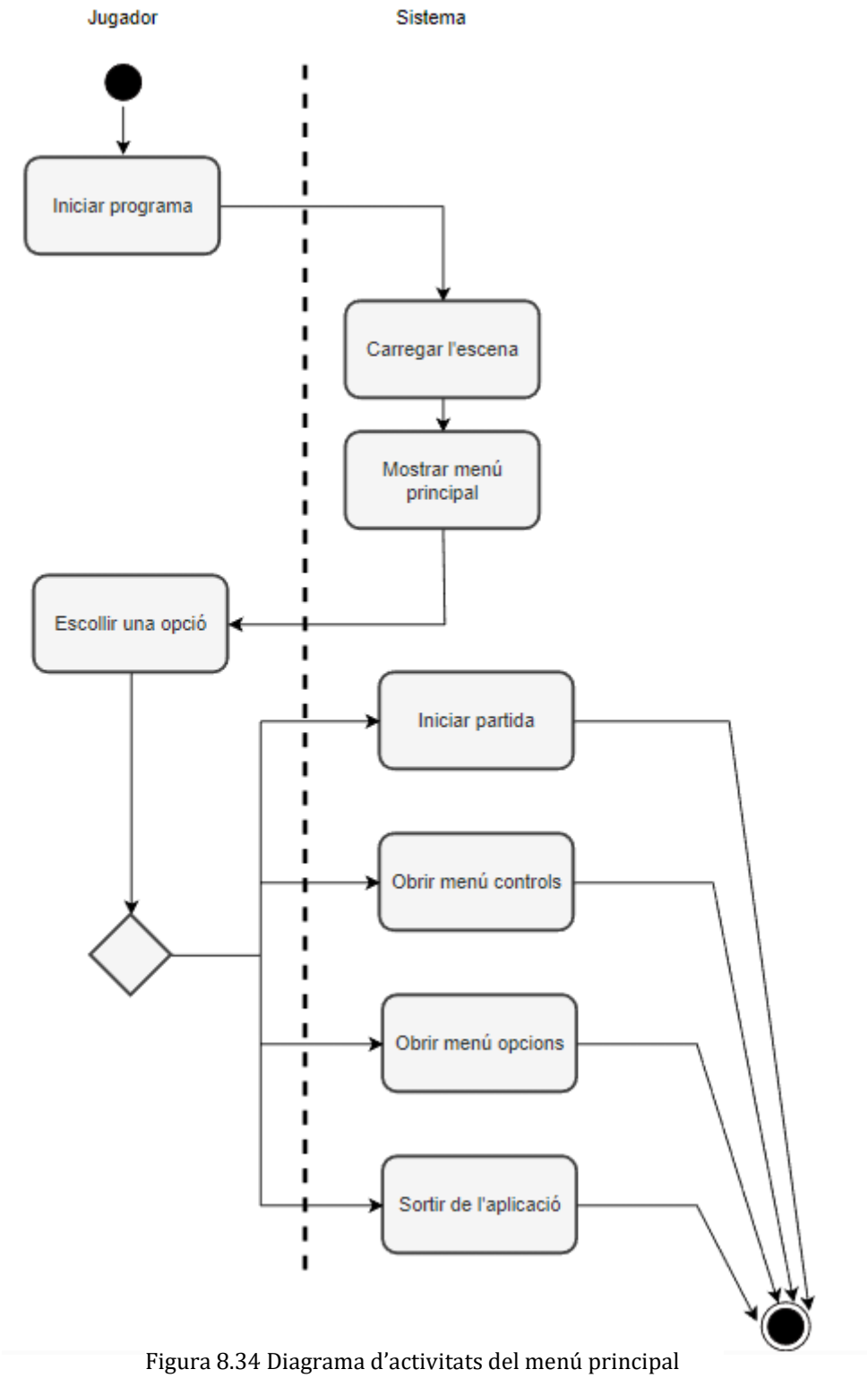

Figura 8.34 Diagrama d'activitats del menú principal

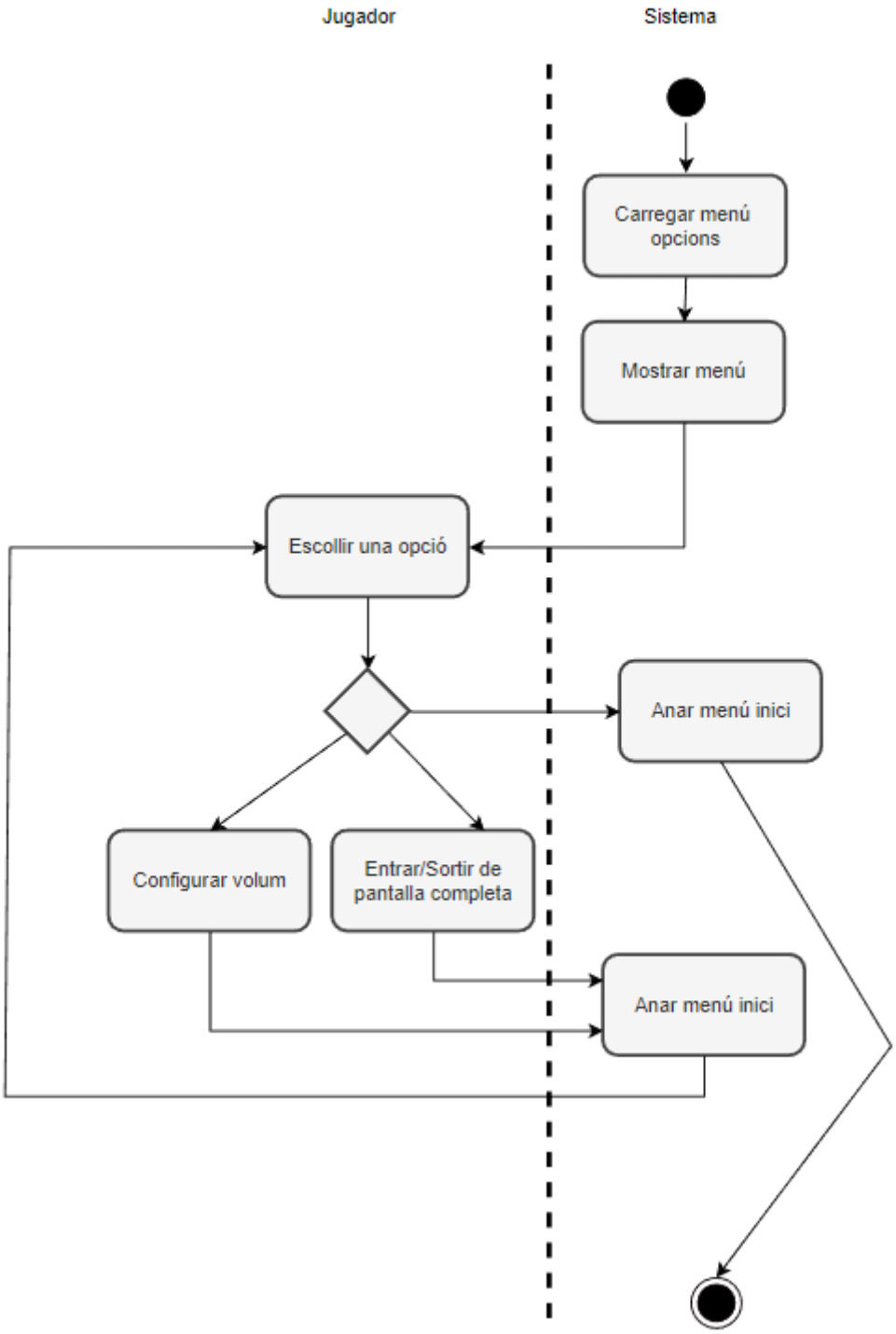

Figura 8.35 Diagrama d'activitats del menú d'opcions

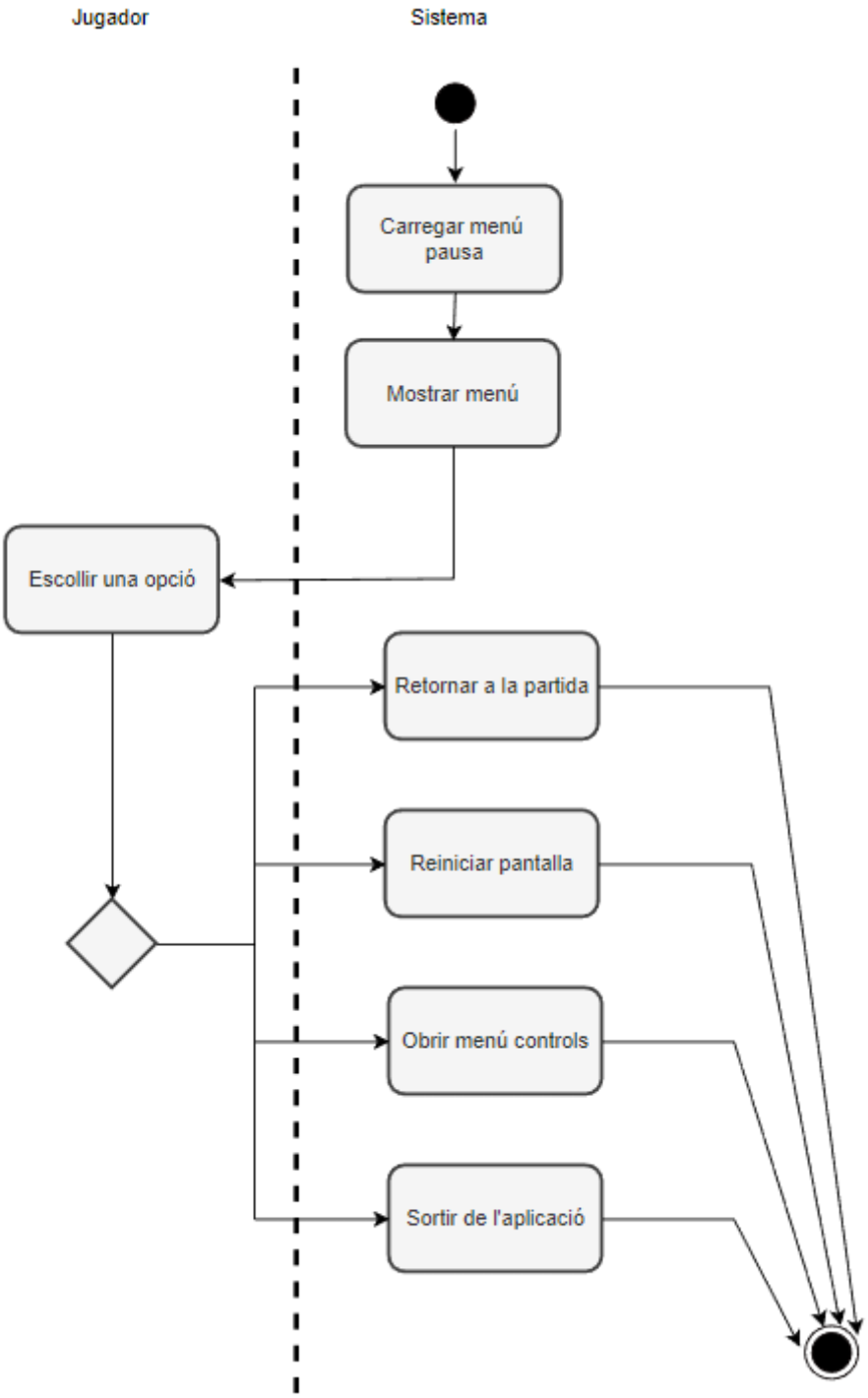

Figura 8.36 Diagrama d'activitats del menú de pausa

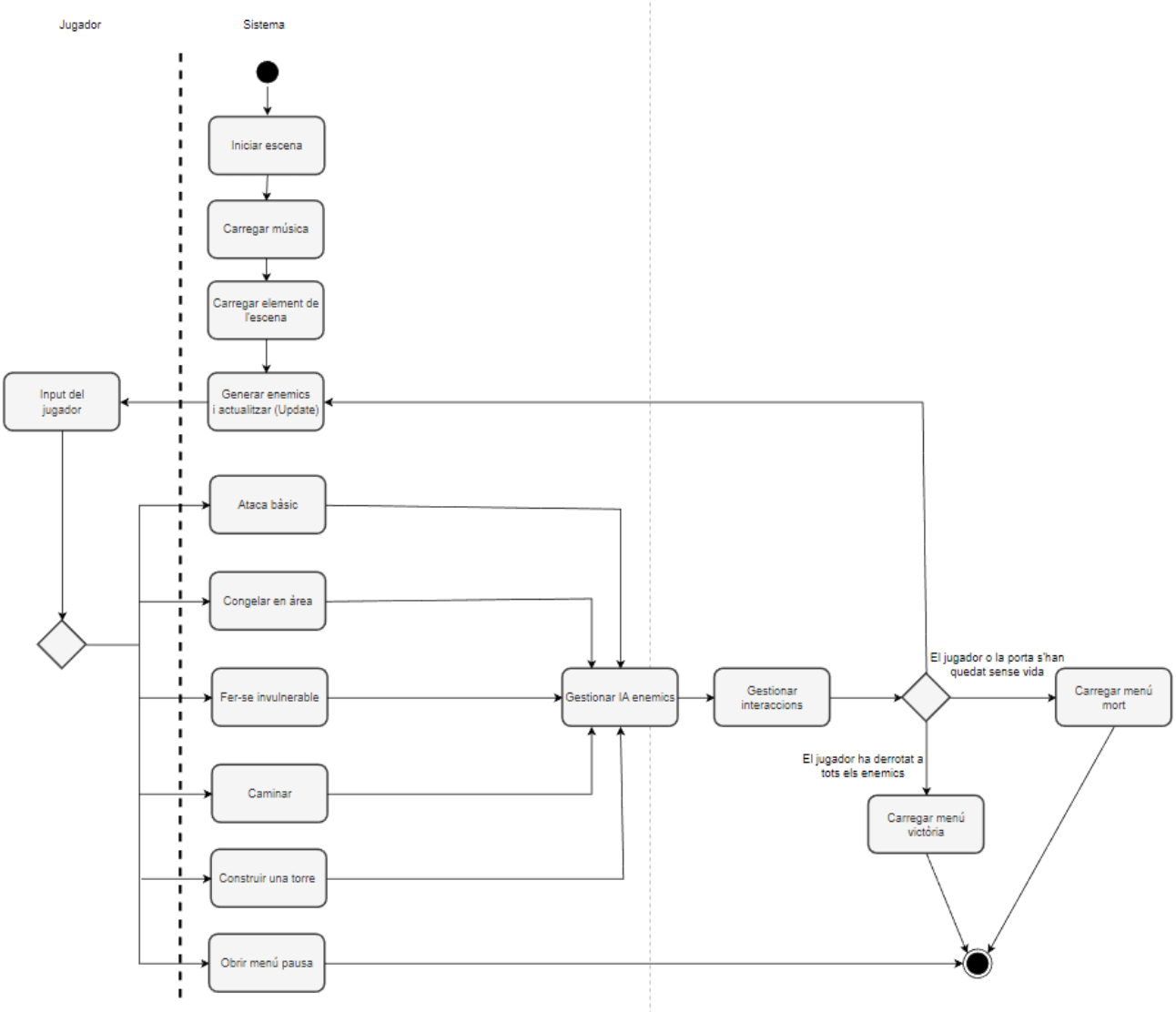

Figura 8.37 Diagrama d'activitats del videojoc

– 8.9 MODEL DE CLASES -

En aquest apartat s'exposaran les parts més importants del disseny del model de classes del sistema, acompanyats de les màquines d'estats d'animacions quan calgui. Seguidament, es pot veure el model de classes del projecte, després s'anirà mostrant les classes utilitzades per cada part, així com els mètodes corresponents. Veure Figura 8.38.

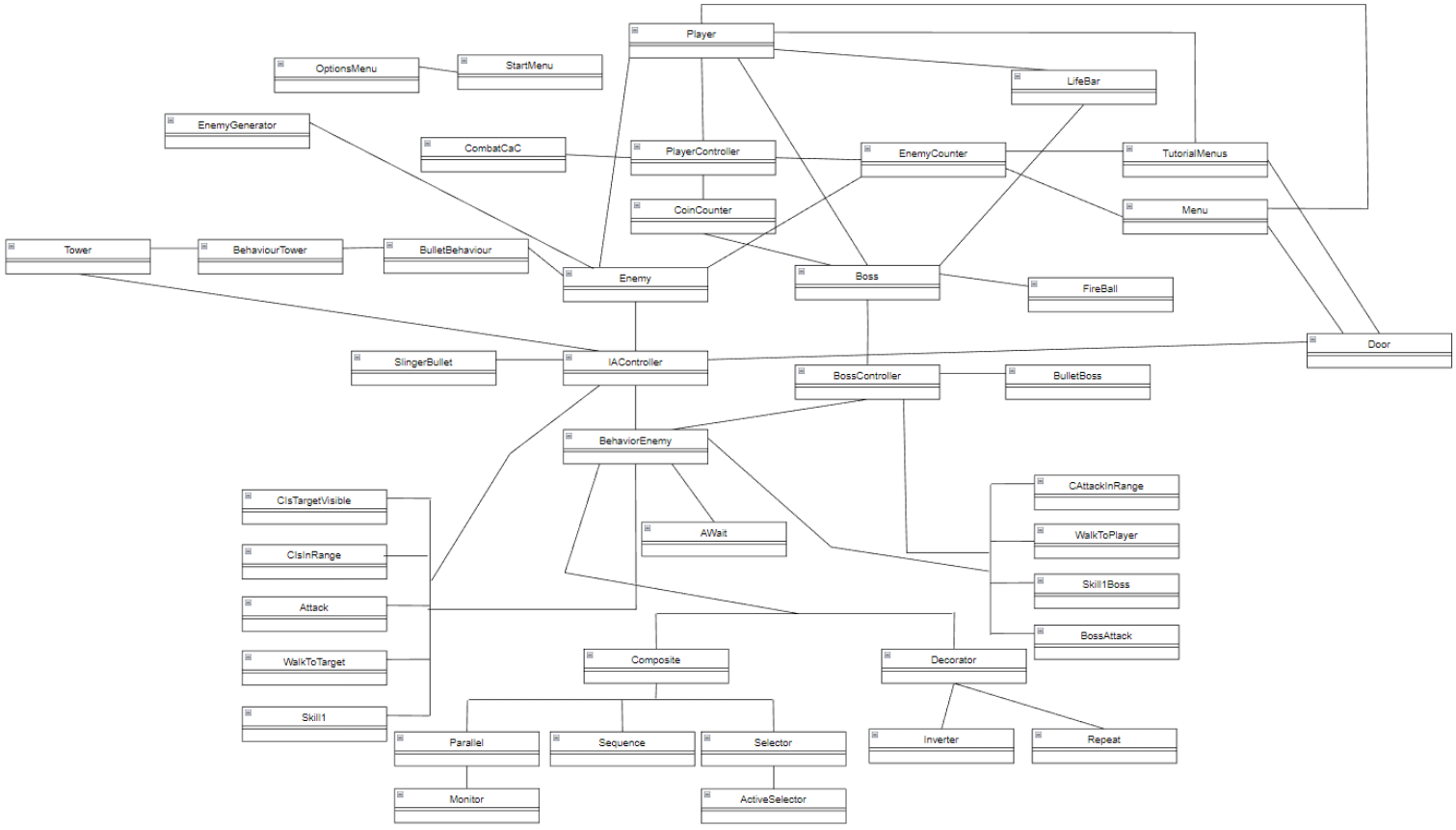

Figura 8.38 Model de classes del projecte

Personatge principal

Les classes que componen la lògica del personatge principal són les següents:

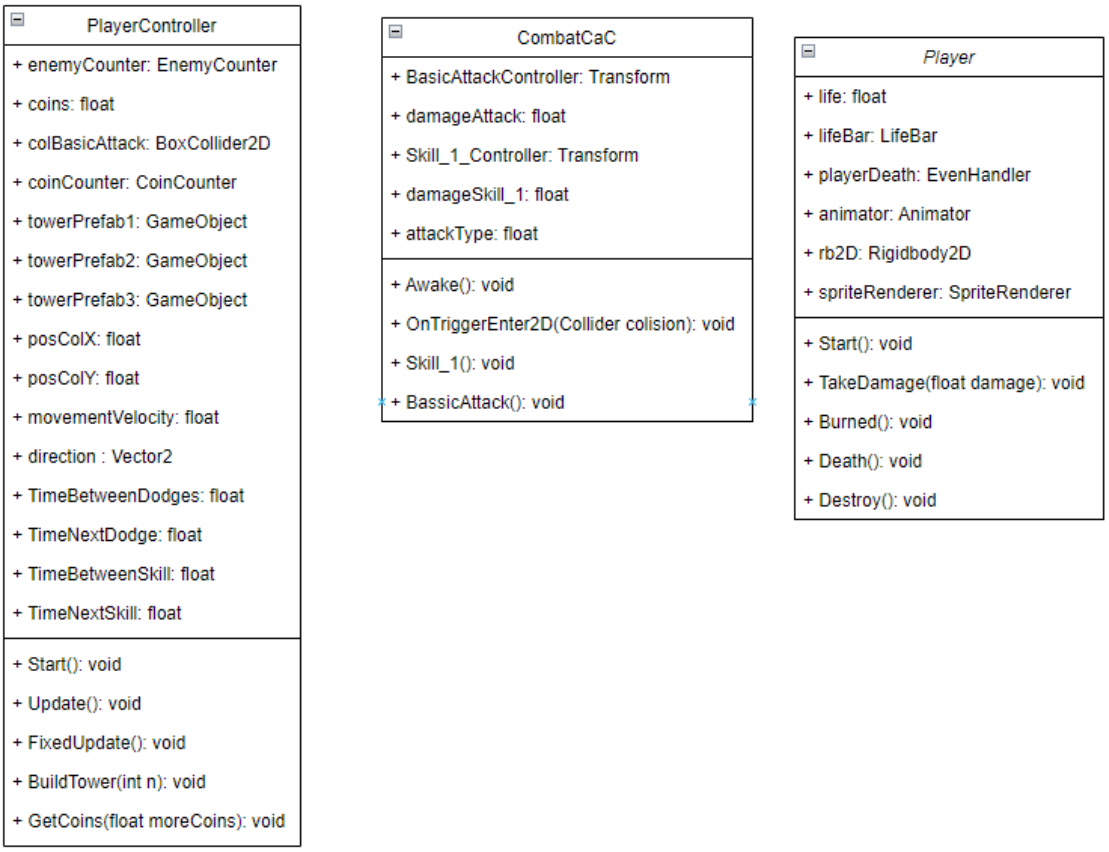

Figura 8.38 Classes del personatge principal

Player: Classe que serveix per guardar informació referent a l'estat del personatge, així com configuracions per defecte.

- Start(): Obté el component Animator i RigidBody. També inicialitza la barra de vida.
- TakeDamage(float damage): Resta la vida del jugador amb la quantitat entrant "damage". Si el resultat de la vida és igual o més baix que zero, llavors inicia el procés de morir.
- Burned(): Canvia de color el spriteRenderer del jugador, per donar l'efecte de cremat.
- Death(): Envia l'esdeveniment que el jugador ha mort.
- Destroy(): Destrueix el gameObject.

PlayerController: Responsable de la gestió dels recursos i d'interpretar els inputs del jugador: caminar, atacar, construir torre.

- Start(): Obté el component Animator i RigidBody

- Update(): Interpreta els Inputs del jugador i posiciona el BoxCollider2D de l'atac bàsic segons la direcció del jugador.
- FixedUpdate(): Mou el jugador segons els inputs entrats.
- BuildTower(int n): Instància una torre, l'int n ens indica quina de les tres torres vol instanciar el jugador.
- GetCoins(float moreCoins): Suma "moreCoins" a la variable de monedes per construir torres.

CombatCac: Classe que s'encarrega de la col·lisió de l'atac del jugador. Aquesta classe s'encarrega de la col·lisió tant de l'atac bàsic com de l'habilitat de congelar en àrea.

- Awake(): Obté el component BoxCollider2D.
- OnTriggerEnter2D(Collider2D colision): Comprova si l'objecte amb el qual ha col·lisionat, té el tag Enemy o Boss. En cas afirmatiu fa mal a l'enemic.
- Skill_1(): Fa que el OnTriggerEnter tingui en compte el collider de l'habilitat de congelar i el mal de l'habilitat.
- BassicAttack(): El mateix que la funció anterior però per l'atac bàsic.

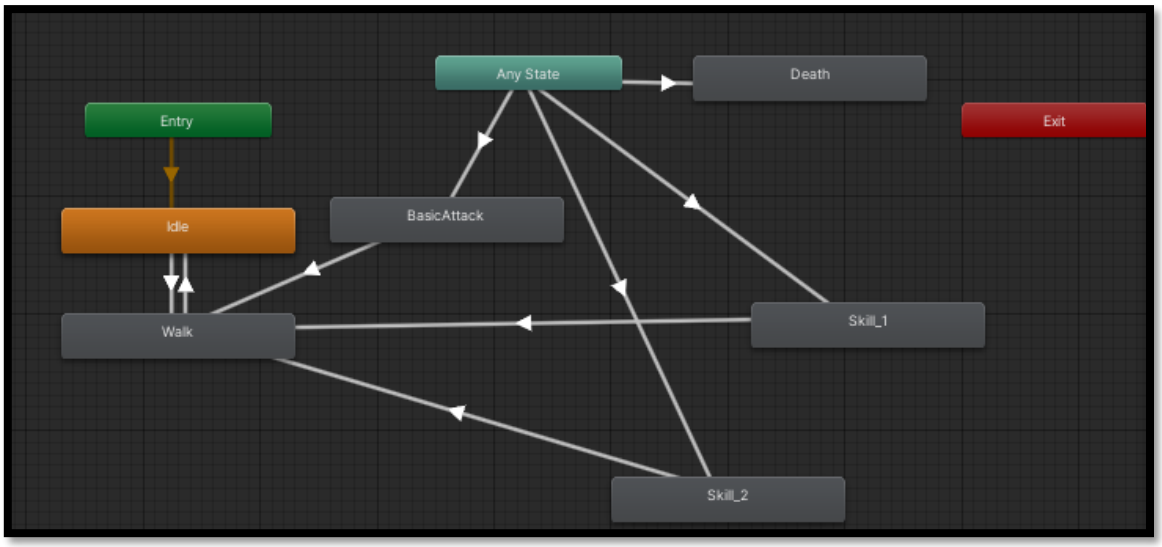

Figura 8.39 Màquina d'estats d'animacions del personatge principal

A la Figura 8.39, podem observar la màquina d'estat d'animacions del personatge principal del component Animator d'Unity. Aquest component ha estat utilitzat per fer gestionar quina animació mostrem en cada moment del videojoc, tant pel jugador com pels enemics, torres, projectils, etc. Cada estat d'aquesta màquina representa una animació o arbre d'animacions (Figura 9.14), i les connexions són variables que podem definir des del codi.

Torres

Totes les torres utilitzen la classe "Tower" com a classe base, però la classe BehaviourTower i BulletBehaviour són diferents per cadascuna. Ara es mostrarà les classes de la torre d'arquers.

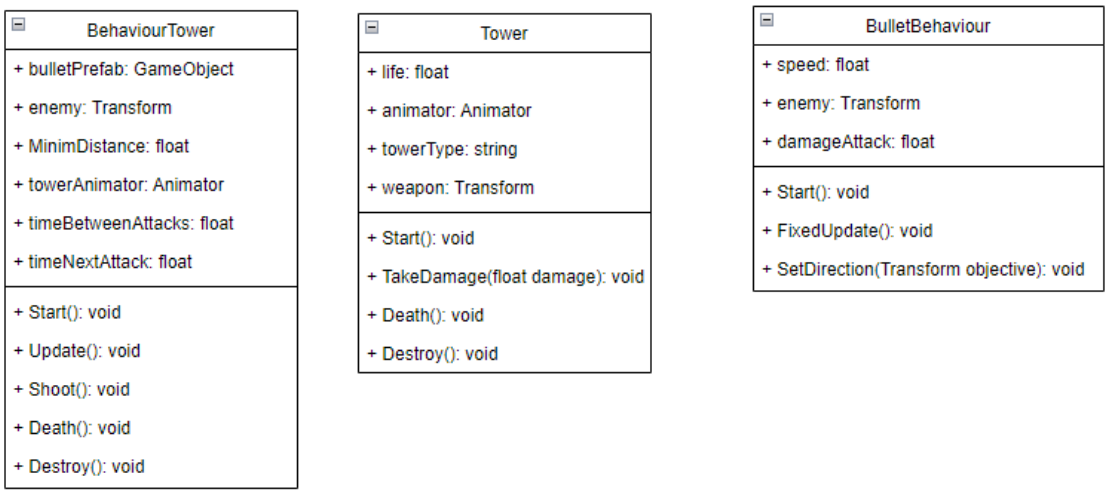

Figura 8.40 Classes de la torre d'arquers

Tower: Classe que serveix per guardar informació referent a l'estat de la torre, així com configuracions per defecte.

- Start(): Inicialització de components i variables.
- TakeDamage(float damage): Resta la vida de la torre amb la quantitat entrant "damage". Si el resultat de la vida és igual o més baix que zero, llavors inicia el procés de morir.
- Death(): Inicia el procés de l'animació de morir de la torre i crida al procés de morir de l'arma de la torre.
- Destroy(): Destrueix el gameObject.

BehaviourTower: Classe associada a l'arma de la torre (veure Figura 8.11), s'encarrega d'escollir a quin enemic atacar i instanciar una bala quan sigui oportú.

- Start(): Inicialització de components i variables.
- Update(): Busca si hi ha algun enemic, en cas afirmatiu comprova si està dintre el rang de dispar per atacar-lo.
- Shoot(): Instància un projectil en direcció a l'enemic seleccionat.
- Death(): Inicia l'animació de destrucció de l'arma de la torre.
- Destroy(): Destrueix el gameObject.

BulletBehaviour: Aquesta classe pertany al projectil que dispari la torre en qüestió, fa que el GameObject associat es dirigeixi cap al seu objectiu, un cop arriba li fa mal i es destrueix.

- Start(): Inicialització de components i variables.
- FixedUpdate(): Mou el projectil cap a la direcció on es troba l'enemic, un cop arriba, li fa mal i es destrueix el GameObject del projectil.
- SetDirection(Transform objective): Seteja el valor de l'objectiu del projectil.

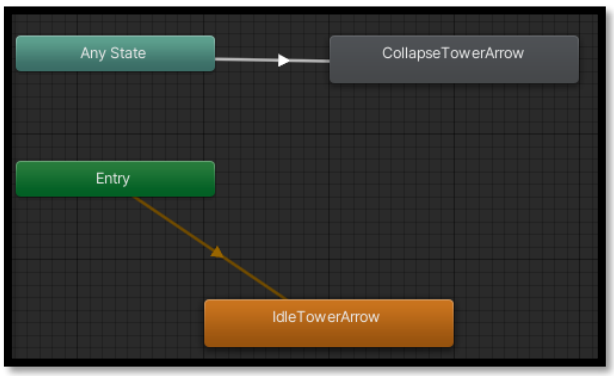

Figura 8.41 Màquina d'estats d'animacions de la torre d'arquers

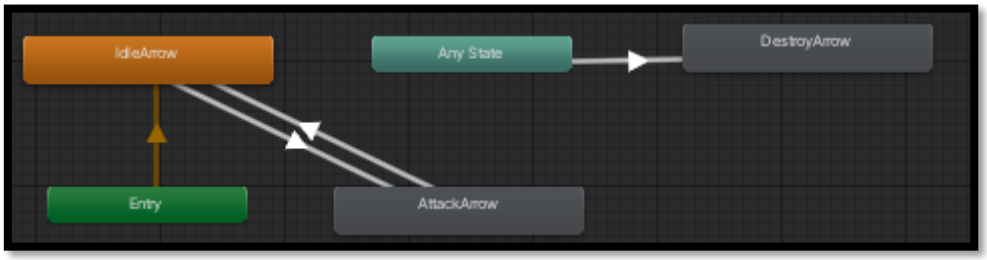

Figura 8.42 Màquina d'estats d'animacions de l'arma

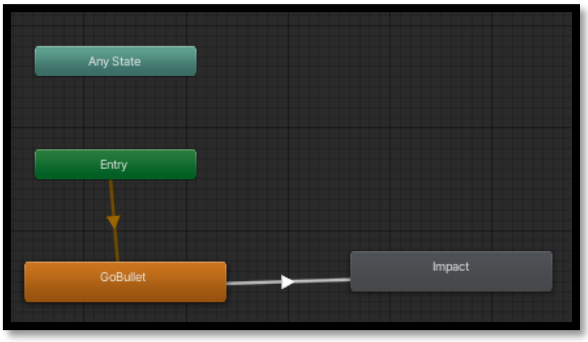

Figura 8.43 Màquina d'estats d'animacions del projectil

A les figures 8.41, 8.42 i 8.43 es pot veure les màquines d'estat de les animacions que componen la torre.

Enemics

Els enemics es divideixen en dos tipus, els bàsics i el jefe final. Els dos grups estan fets amb intel·ligència artificial, però utilitzen classes diferents.

Tots els enemics bàsics usen la classe IAController per gestionar l'arbre de comportament que tenen associat. El jefe final utilitza la classe BossController.

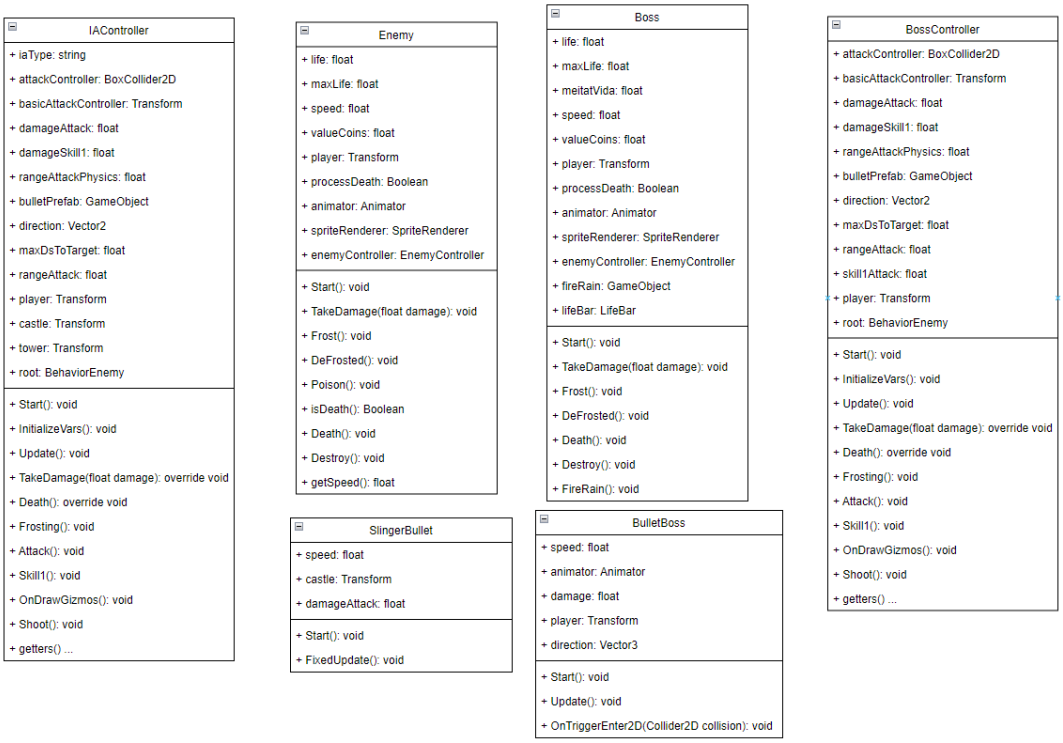

Figura 8.44 Classes dels enemics

Enemy: Classe que serveix per guardar informació referent a l'estat de l'enemic, així com configuracions per defecte.

- Start(): Inicialització de components i variables.
- TakeDamage(float damage): Resta la vida de l'enemic amb la quantitat entrant "damage". Si el resultat de la vida és igual o més baix que zero, llavors inicia el procés de morir.
- Frost(): Canvia el color del sprite de l'enemic i activa l'animació Frost. Si l'atac de congelar mata l'enemic, llavors no es congela, mor directament.
- DeFrosted(): Descongela l'enemic.

- Poison(): Es calcula el 10% de la vida màxima de l'enemic i es resta a la vida actual. També canvia el color del sprite a l'enemic.
- Death(): Inica l'animació de morir i envia al jugador les monedes que guanya per haver-lo derrotat.
- Destroy(): Destrueix el GameObject.

IAController: S'encarrega de gestionar el comportament de l'enemic utilitzant una intel·ligència artificial, concretament Behavior Trees.

- Start(): Crida la funció InitializeVars().
- InitialitzeVars(): Inicialitza els components i variables. També genera l'arbre de comportament.
- Update(): Obté la posició horitzontal i vertical respecte el jugador. Si existeix una torre, la selecciona com a objectiu i avalua i executa l'arbre de comportament.
- Death(): crida la funció Death d'enemy.
- Attack(): Gestiona la col·lisió de l'atac bàsic de l'enemic amb el jugador, les torres i la porta.
- Skill1():Gestiona la col·lisió de la primera habilitat de l'enemic amb el jugador, les torres i la porta.
- OnDrawGizmos(): Dibuixa en l'editor d'Unity un cercle que indica l'àrea de l'atac bàsic de l'enemic.
- Shoot(): Instànica un projectil.

SlingerBullet: És la classe associada al projectil que dispara l'enemic Goblin Slinger (veure apartat 8.4), aquesta s'encarrega que el projectil vagi directament cap a la porta, li fa mal i es destrueix.

- Start(): Inicialitza components i variables.
- FixedUpdate(): Desplaça el projectil cap a l'objectiu, en aquest cas la porta. Quan arriba a la posició de la porta, li aplica mal i destrueix el gameObject de la bala.

Boss: Classe que serveix per guardar informació referent a l'estat del jefe final, així com configuracions per defecte.

- Start(): Inicialització de components i variables.
- TakeDamage(float damage): Resta la vida de l'enemic amb la quantitat entrant "damage". Si el resultat de la vida és igual o més baix que la meitat, llavors s'activa l'animació de la segona habilitat del jefe final. Si el resultat de la vida és igual o més baix que zero, llavors inicia el procés de morir.

- Frost(): Canvia el color del sprite de l'enemic i activa l'animació Frost. Si l'atac de congelar mata al enemic, llavors no es congela, mor directament.
- DeFrosted(): Descongela l'enemic.
- Death(): Inica l'animació de morir.
- Destroy(): Destrueix el GameObject
- FireRain(): Inicia el generador de boles de foc.

BossController: Fa el mateix que la classe IAController però pel jefe de la pantalla final.

- Start(): Crida la funció InitializeVars().
- InitialitzeVars(): Inicialitza els components i variables. També genera l'arbre de comportament.
- Update(): Obté la posició horitzontal i vertical respecte el jugador, també avalua i executa l'arbre de comportament.
- Death(): crida la funció Death de Boss.
- Attack(): Gestiona la col·lisió de l'atac bàsic del jefe final amb el jugador.
- Skill1():Gestiona la col·lisió de la primera habilitat del jefe final amb el jugador.
- OnDrawGizmos(): Dibuixa en l'editor d'Unity un cercle que indica l'àrea de l'atac bàsic del jefe final.
- Shoot(): Instànica un projectil.

BulletBoss: Classe associada al projectil que dispara el jefe final. Fa que el projectil associat vagi a la posició que estava el jugador al ser invocat.

- Start(): Inicialitza components i variables. Inicialitza la direcció de la bala cap a la posició del jugador en el moment d'instanciar-se la bala.
- Update(): Mou la bala cap a la posició que té com a objectiu.
- OnTriggerEnter2D(Collider2D colision): Gestiona la col·lisió de la bala amb el jugador.

A les Figures 8.45, 8.46 i 8.47 es poden observar les màquines d'estats de les animacions dels diferents enemics del videojoc.

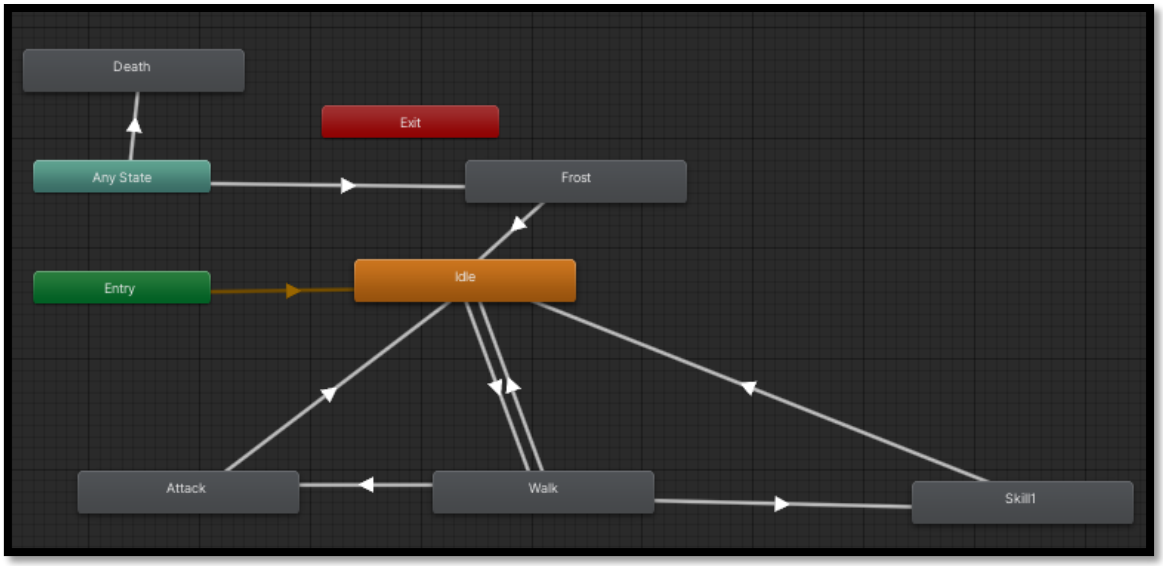

Figura 8.45 Màquina d'estats d'animacions del Goblin Beast

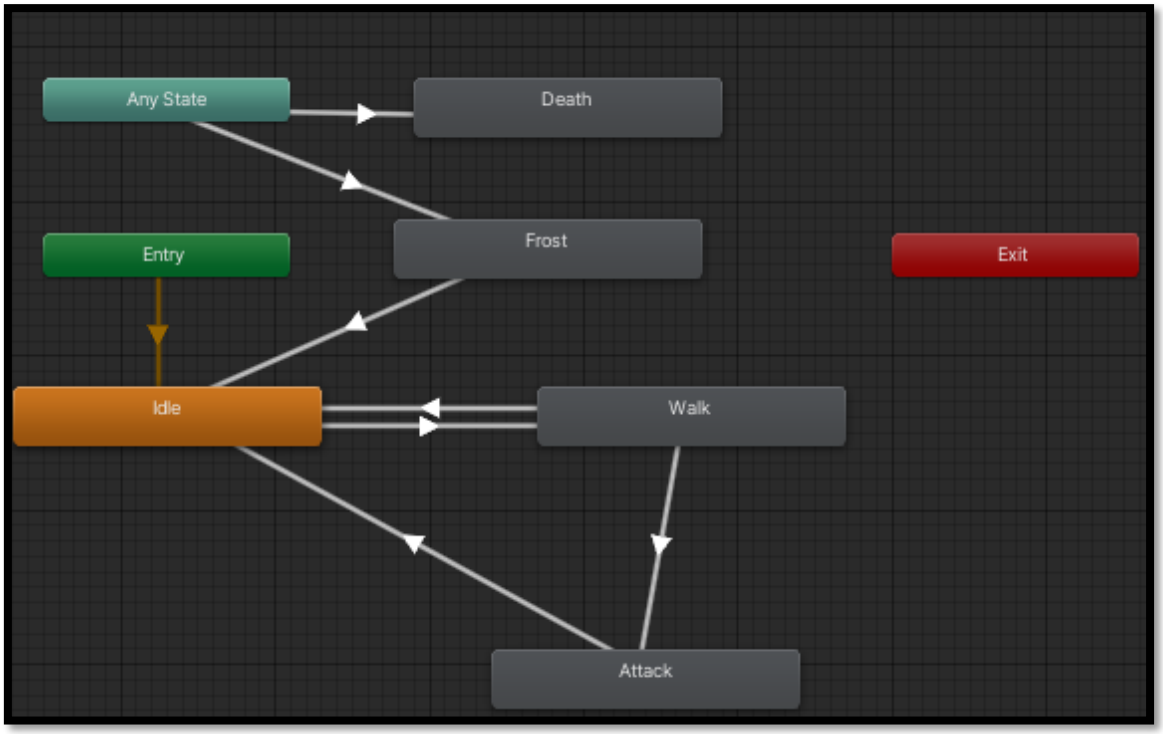

Figura 8.46 Màquina d'estats d'animacions del Goblin Rider/Berserk i Slinger

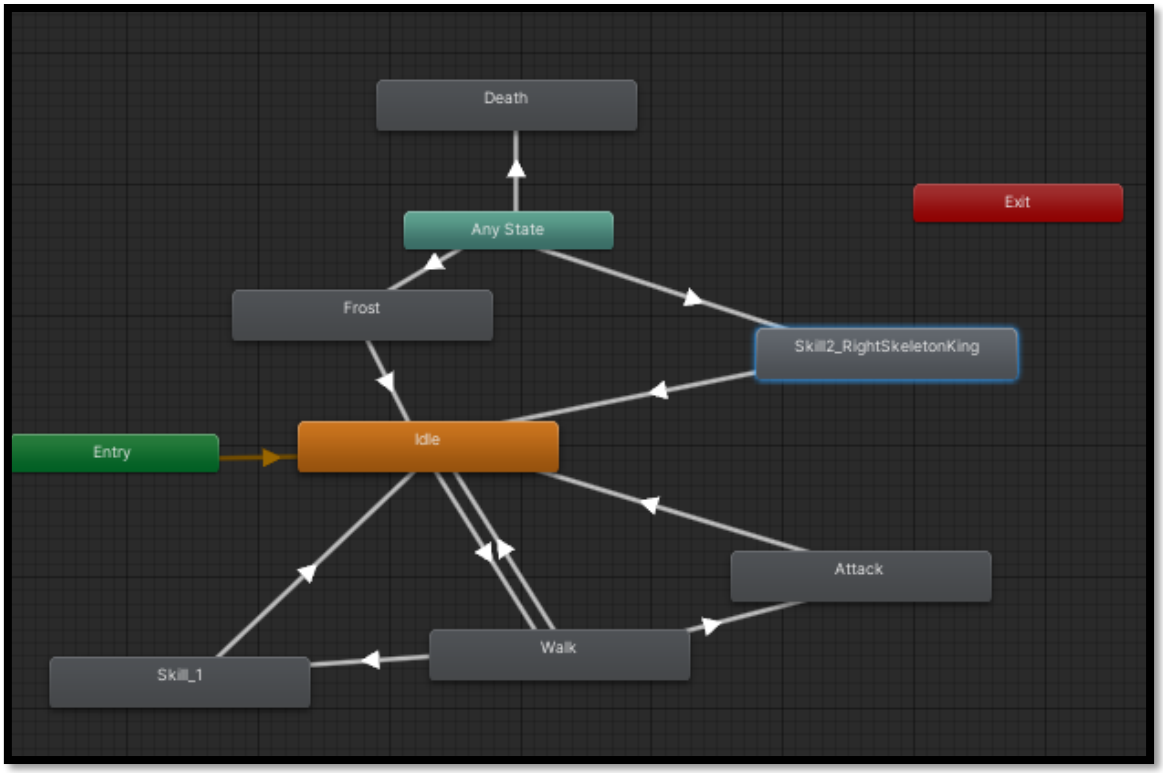

Figura 8.47 Màquina d'estats d'animacions del jefe final

Behavior Trees

Els behavior trees és la intel·ligència artificial escollida per donar vida als enemics d'aquest projecte. Un arbre de comportament en intel·ligència artificial és una estructura de dades jeràrquica que s'utilitza per modelar el comportament dels agents intel·ligents, com ara personatges no jugables (NPCs) en videojocs o agents virtuals en simulacions. Aquesta estructura organitza els comportaments en forma d'un arbre, on cada node representa una acció o una condició que l'agent pot prendre en un determinat context, explicar a l'apartat 7.5.

Seguidament, es mostrarà el diagrama de classes que s'ha fet servir per fer composicions de les accions i condicions.

S'ha après molt d'aquesta pàgina web sobre el funcionament dels behavior trees : [*Pêcheux, M. (2022, 4 enero)*].

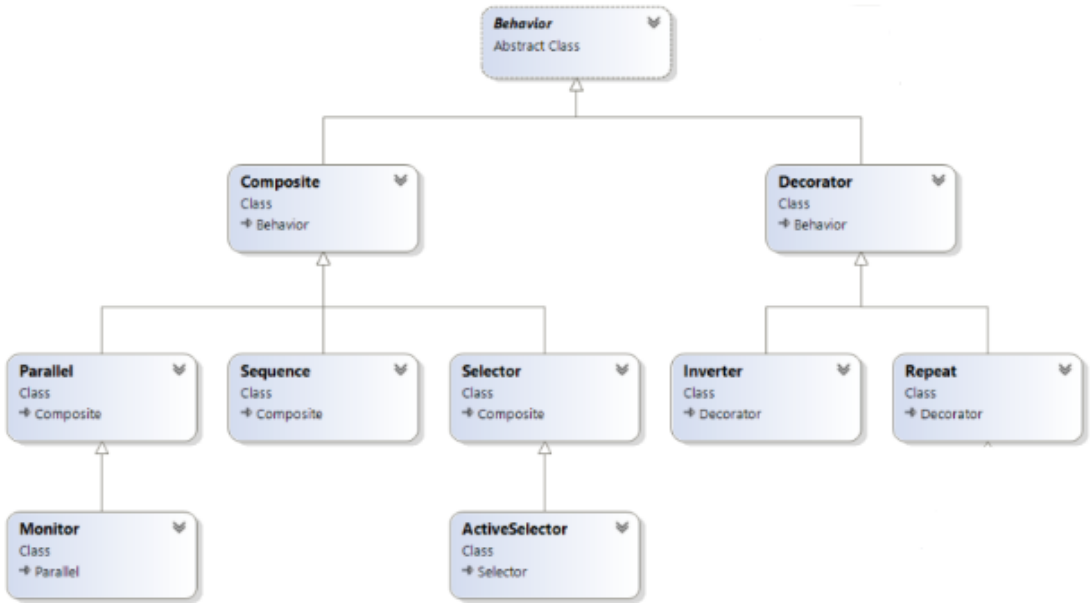

Figura 8.48 Diagrama de classes del behaviour tree

Behavior: Classe principal que gestionarà tots els comportaments (accions i condicions) de l'arbre de comportament.

Un comportament té un estat, el qual pot prendre qualsevol dels valors següents.

- Invalid: resultat inesperat d'actualitzar el comportament.
- Success: el comportament s'ha actualitzat amb èxit.
- Failure: el comportament s'ha actualitzat sense èxit.
- Running: el comportament se segueix executant.
- Avorteig: comportament avortat.

A la classe principal també s'han declarat alguns mètodes: Reset(), Abort(), IsTerminated(), IsRunning(), GetStatus(), per ajudar-nos a obtenir i modificar l'estat actual del comportament d'acord amb les nostres necessitats.

- Reset(): Reinicia i posa l'estat en invàlid.
- Abort(): Finalitza l'estat i posa l'estat en avorteig.
- IsTerminated(): Retorna un booleà indicant si l'estat a acabat o no.
- IsRunning(): Retorna un booleà si l'estat encara està corrent.
- GetStatus(): Obté l'estat actual.
- Tick(): L'objectiu del mètode Tick() és avançar en l'avaluació i execució de l'arbre de comportament.

Composite: Component que serveix per emmagatzemar diversos comportaments. Ara s'especialitzarà en comportaments més específics com ho són Sequence, Selector i Parallel. D'aquesta manera el NPC podrà utilitzar més d'una condició o acció.

Sequence: Executa comportaments en seqüencia, quan acaba de fer una acció passa a la següent.

Selector: Aquest component selecciona un dels comportaments i l'executa.

ActiveSelector: A vegades el Selector anterior pot seleccionar en algunes ocasions el comportament que no volem, ja que aquesta tria es fa de forma 'passiva'. Per tant, aquest component el que fa és forçar la selecció perquè sempre executi el comportament que volem.

Parallel: Executa comportaments simultàniament. S'ha programat de dues formes, que executi tots els comportaments quan almenys un retorni èxit, o per altra banda quan tots retornin èxit.

Monitor: Sempre declararem primer una condició, aquesta serà l'encarregada de monitorar el comportament que cal executar, quan la condició arribi a fallar l'acció es deixa d'executar.

Decorator: Aquesta classe envolta un node existent i li afegeix funcionalitats o restriccions addicionals. En aquest programa s'ha creat el Inverter i el Repeat.

Inverter: Específic per a condicions, inverteix l'estat de tornada. És a dir, citant la nostra condició (És l'objectiu visible?), en posar-la dins del Inverter la pregunta canviaria per (És l'objectiu NO visible?). D'aquesta manera estalvies haver de fer dues condicions.

Repeat: S'utilitza per repetir un comportament el número de vegades que desitgem.

Ara es mostrarà com han quedat els arbres de comportament dels diferents enemics i s'explicarà les condicions i accions que contenen:

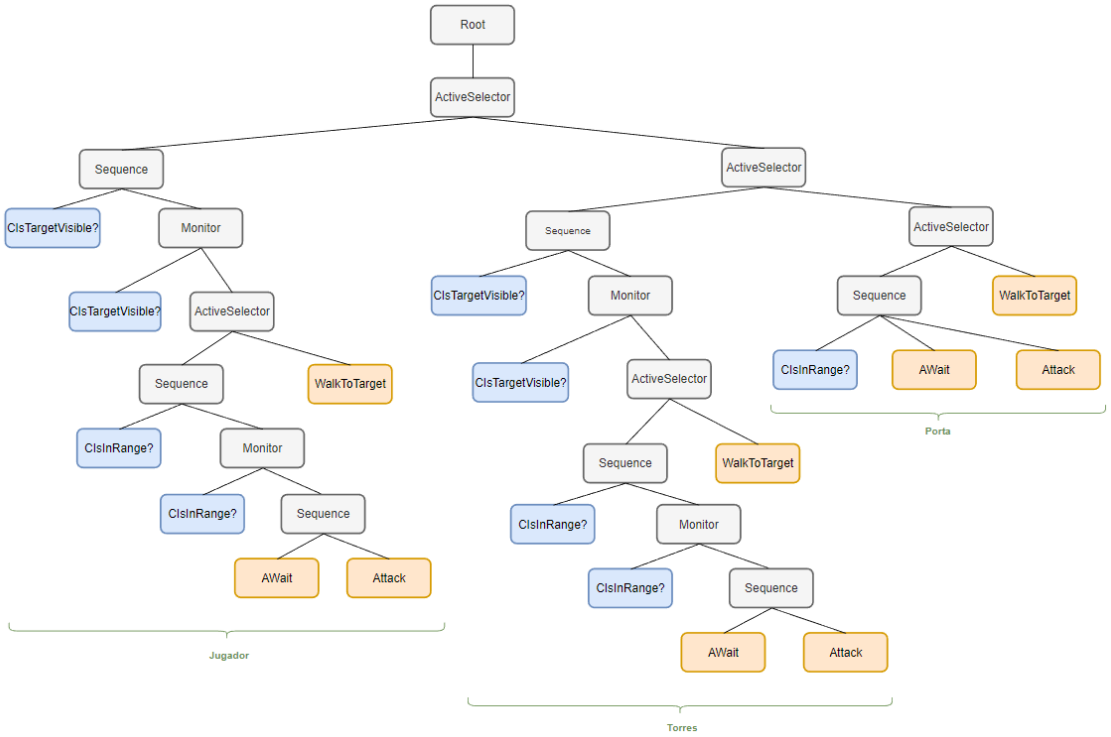

Figura 8.49 Behavior Tree Goblin Berserk

En la Figura 8.49, es pot observar l'arbre de comportament del goblin berserk. Aquest arbre té dues grans branques. La de l'esquerra és per gestionar el comportament que té l'enemic quan el jugador entra en el rang de visió. Per altra banda, la branca de la dreta està dividida amb dues branques: una per gestionar el comportament de l'enemic quan una torre entra en el rang de visió, i l'altre per anar cap a la porta i atacar quan estigui dintre el rang de visió.

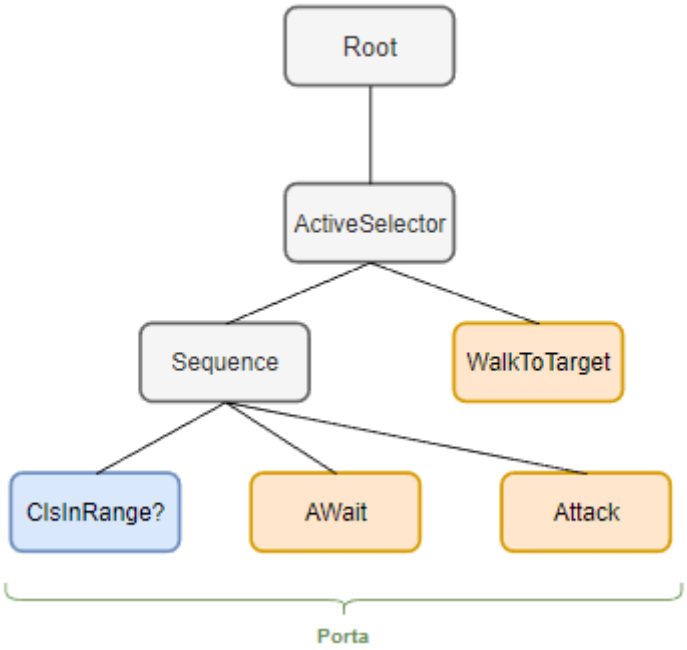

Figura 8.50 Behavior Tree Goblin Slinger

En la Figura 8.50 es pot veure l'arbre de comportament del goblin slinger. Aquest arbre comprova si la porta està dintre el rang de visió, per seguidament preparar-se i atacar. Si la porta no està dintre el rang d'atac, continuarà caminant cap a la porta.

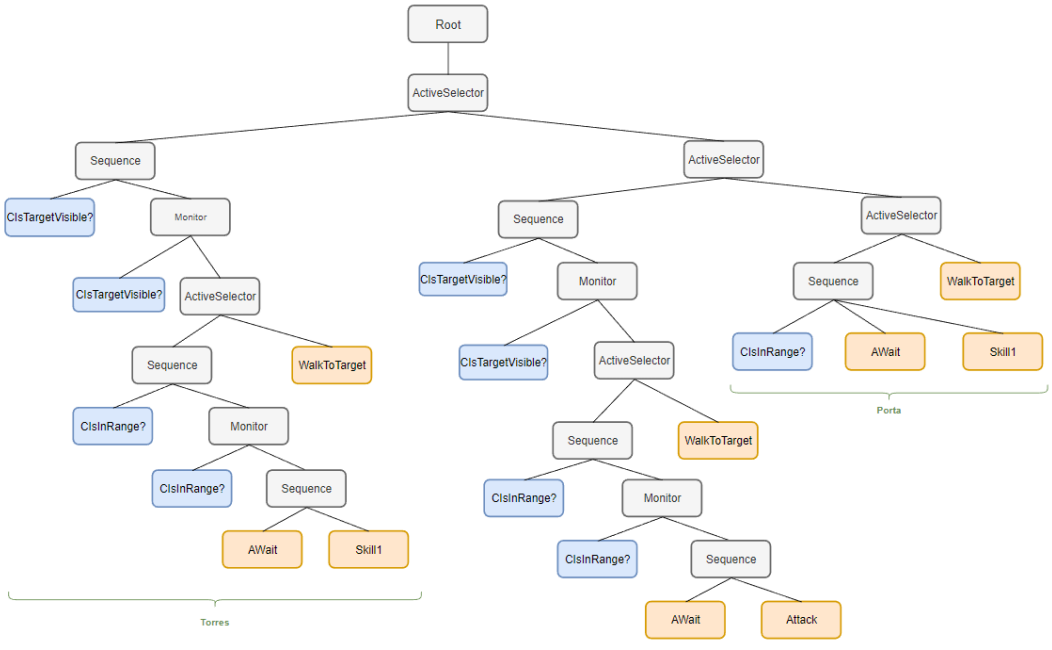

Figura 8.51 Behavior Tree Goblin Beast

A la Figura 8.51 s'observa l'arbre de comportament del goblin beast. És semblant al del goblin berserk, però prioritzant a les torres (branca esquerra) i posteriorment al jugador i per acabar la porta. Una altra diferència és que si l'enemic ataca a les torres o la porta, es crida a la classe Skill1, en lloc d'Attack.

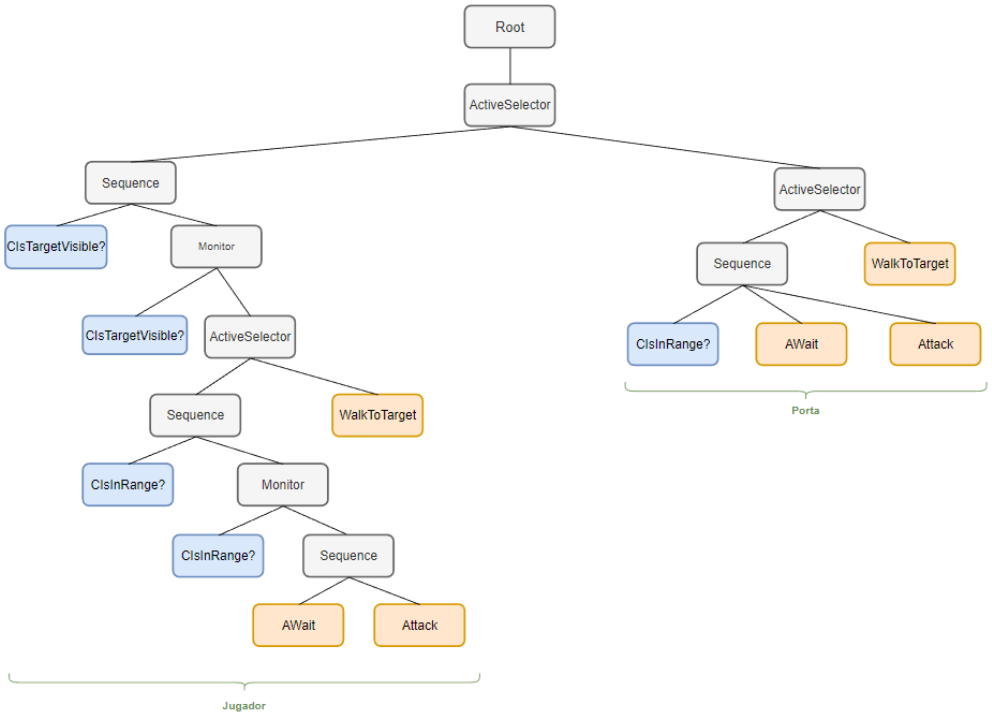

Figura 8.52 Behavior Tree Goblin Rider

A la Figura 8.52 es pot veure l'arbre de comportament del goblin rider. Aquest enemic té dues branques: la primera gestiona el comportament de l'enemic quan el jugador entra al rang de visió, la segona gestiona l'atac cap a la porta.

A la Figura 8.53 es pot observar l'arbre de comportament del jefe final del videojoc. La branca de la dreta camina cap al jugador, en cas que aquest es trobi molt lluny. Si el jugador s'apropa, llavors poden passar dues coses: que entri dins el rang de visió de l'atac bàsic o que no s'apropi prou i provoqui que el jefe el dispari.

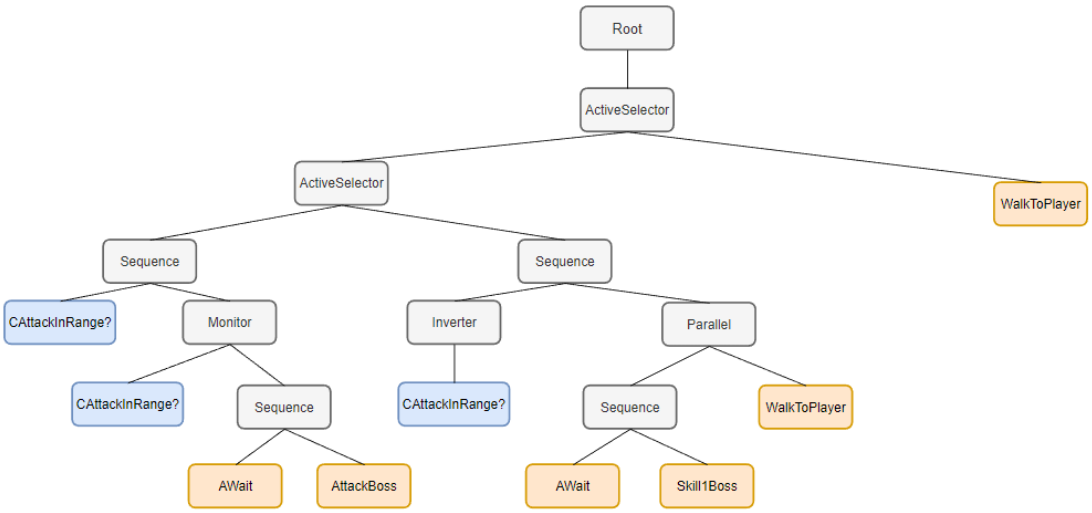

Figura 8.53 Behavior Tree Jefe final

CIsTargetVisible: És una condició que retorna èxit si l'element que te com a objectiu entra dintre el rang de visió, altrament fals.
CIsInRange: Una condició que retorna èxit si l'element que te com a objectiu entra en el rang d'atac.
Attack: Acció que s'encarrega d'orientar el collider2D d'atac de l'enemic i fer que aquest ataqui.
WalkToTarget: Acció que provoca que l'enemic es dirigeixi cap a la posició de l'element que té com a objectiu.
Skill1: Acció que s'encarrega de la gestió de la primera habilitat dels enemics, en aquest cas la del goblin beast.
AWait: Acció que atura el comportament actual de l'enemic durant uns segons determinats.
CAttackInRange: Una condició que retorna èxit si l'element que té com a objectiu el jefe final entra en el rang d'atac.
Skill1Boss: Acció que s'encarrega de la gestió de l'habilitat 1 del jefe final.
WalkToPlayer: Acció que provoca que el jefe final camini cap al jugador.
AttackBoss: Acció que s'encarrega d'orientar el collider2D d'atac del jefe final i fer que aquest ataqui.

Classes interfície del jugador

Classes que s'han creat per controlar la interfície del videojoc, menús, barres de vida, generació d'enemics, comptador d'enemics i comptador de monedes, etc..

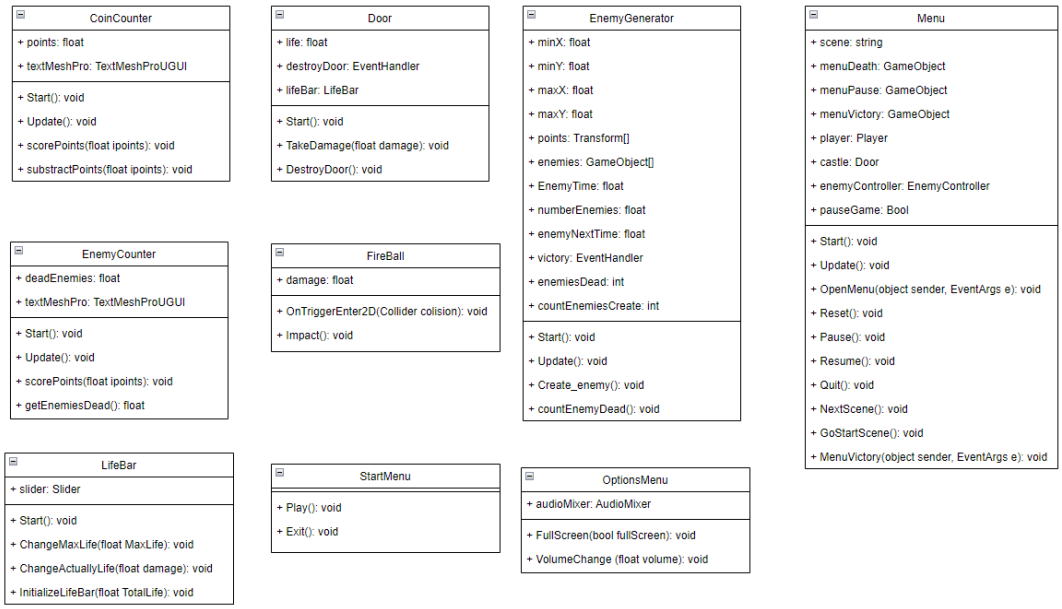

Figura 8.54 Classes control d'interfície del jugador

CoinCounter: Classe que gestiona els recursos que té el jugador per construir torres.

- Start(): Obté el component TextMeshPro.
- Update(): Actualitza el comptador amb la variable points.
- scorePoints(float ipoints): Suma a la variable points el valor d'ipoints.
- substractPoints(float ipoints): Resta a la variable points el valor d'ipoints.

EnemyCounter: Classe que augmenta una unitat el comptador d'enemics derrotats per l'enemic.

- Start(): Inicialitza variables i components.
- Update(): Comprova si ja ha derrotat a tots els enemics. En cas afirmatiu crida un esdeveniment per obrir el menú de victòria. Si encara no ha derrotat a tots els enemics, actualitza el comptador amb el valor de la variable deadEnemies.
- scorePoints(float ipoints): Suma a la variable deadEnemies el valor d'ipoints.
- getEnemiesDead(): Retorna el nombre d'enemics derrotats.
- countEnemyDead(): Augmenta en un els enemics derrotats.

LifeBar: Aquesta classe gestiona les barres de vida, tant la del jugador, la de la porta o la del jefe final.

- Start(): Obté el component Slider.
- ChangeMaxLife(float MaxLife): Inicialitza la vida màxima del component Slider.
- ChangeActuallyLife(float lifeDamage): Modifica el valor de la barra de vida perquè sigui igual a lifeDamage.
- InitializeLifeBar(float TotalLife): Crida a les dues funcions anteriors.

Door: Classe que serveix per guardar informació referent a l'estat del personatge.

- Start(): Inicialitza la barra de vida.
- TakeDamage(float damage): Resta la vida de la porta amb la quantitat entrant "damage". Si el resultat de la vida és igual o més baix que zero, llavors inicia el procés de morir.
- Destroy(): Destrueix el gameObject.

FireBall: Classe associada a les boles de foc que surten quan el jefe final perd la meitat de la vida. Gestiona la interacció de les boles de foc amb el jugador.

- OnTriggerEnter2D(Collider colision): Gestiona la col·lisió entre el jugador i la bala. Li aplica mal i l'estat de cremat al jugador en cas de col·lisió.
- Impact(): Destrueix el gameObject.

StartMenu: Classe que controla les opcions del menú principal (veure Figura 8.21).

- Play(): Ens redirigeix a l'escena del tutorial.
- Exit(): Surt de l'aplicació.

OptionsMenu: Classe que controla les opcions del menú d'opcions (veure Figura 8.22).

- FullScreen(bool fullScreen): Canvia la configuració de l'aplicació entre pantalla completa a pantalla no completa.
- VolumeChange(float volume): Modifica el volum del videojoc amb la variable volume entrada per paràmetre.

Menu: Classe que controla els diferents menús que apareixen durant la partida: menú pausa, menú victòria i derrota.

- Start(): Inicialitza variables, components i esdeveniments.
- Update(): Si el jugador prem la tecla Esc del teclat obre/tanca el menú de pausa.
- OpenMenu(object sender, EventArgs e): Para el temps de joc i obre el menú de mort.
- Reset(): Activa el temps de joc que ha estat pausat prèviament i reinicia l'escena actual.
- Pause(): Pausa el temps de joc i obre el menú de pausa.
- Resume(): Tanca el menú de pausa i activa el temps de joc.
- Quit(): Surt de l'aplicació.
- NextScene(): Redirigeix al jugador a la següent escena.
- GoStartScene(): Redirigeix al jugador a la pantalla d'inici.
- MenuVictory(object sender, EventArgs e): Para el temps de joc i obre el menú de victòria.

TutorialMenus: Gestiona tots els texts i menús que s'obriran durant la primera pantalla del videojoc, el tutorial.

- Start(): Inicialitza variables, components i esdeveniments.
- Update(): Si el jugador prem la tecla Esc del teclat obre/tanca el menú de pausa. També comprova quants enemics ha derrotat el jugador per obrir els següents texts del tutorial.
- OpenMenu(object sender, EventArgs e): Para el temps de joc i obre el menú de mort.
- Reset(): Activa el temps de joc que ha estat pausat prèviament i reinicia l'escena actual.
- Pause(): Pausa el temps de joc i obre el menú de pausa.
- Resume(): Tanca el menú de pausa i activa el temps de joc.
- Quit(): Surt de l'aplicació.
- NextScene(): Redirigeix al jugador a la següent escena.
- activeMenu(float duration, string type): Activa texts del tutorial després d'una duració entrada per paràmetre.

EnemyGenerator: Classe que te associada quatre punts en el mapa, i fa aparèixer els enemics aleatòriament dintre de l'area que conformen els quatre punts.

- Start(): Inicialitza variables.
- Update(): Si el temps d'aparició d'un enemic és igual a zero, llavors crida a la funció create_enemy(). Cada x enemics creats redueix en el temps d'aparició dels enemics.

- Create_enemy(): Crea un enemic aleatòriament de la llista d'enemics en un punt aleatori en l'àrea que conformen els quatre punts inicialitzats en el mètode Start().

CAPÍTOL 9

– 9. IMPLEMENTACIÓ I PROVES -

En aquest capítol s'explicarà com s'ha implementat cada una de les parts del projecte.

– 9.1 MENÚ PRINCIPAL, D'OPCIONS I DE CONTROLS -

Aquests dos menús formen la primera escena que veurà el jugador quan inicia el videojoc. El menú principal conté un canvas que escala amb la resolució de la pantalla i on es col·loca els dos scripts, Figures 9.3 i 9.4, així com tots els elements d'UI.

Dintre del canvas hi ha un element que conté tots els elements de la interfície gràfica. Aquest element és un "panel", el qual es tracta de la imatge de fons que hi ha mentre el jugador escull l'opció del menú que vol fer. També conté tres elements de text, per canviar de títol en funció del menú que es vulgui obrir i tres elements que dintre conté els botons i els respectius texts. Veure Figura 9.1.

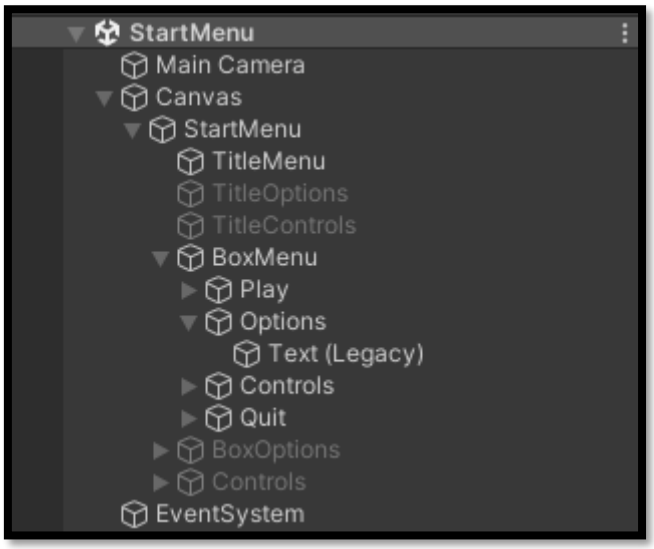

Figura 9.1 Hierarchy de la primera escena

Aquests menús tenen diferents botons que han de dur a terme una acció implementada en scripts. Per enllaçar la implementació amb el botó, s'utilitza la funció OnClick() del component Button d'Unity (Figura 9.2).

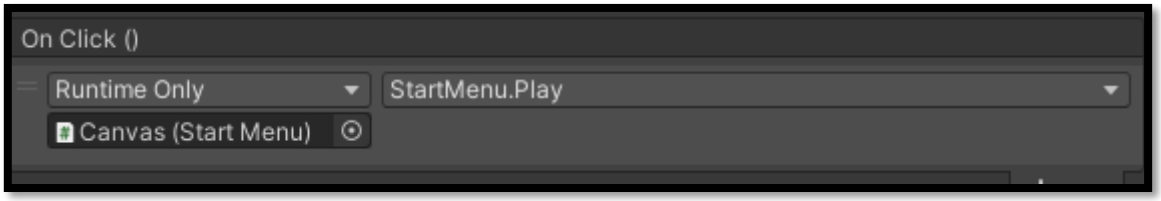

Figura 9.2 Funció OnClick() d'Unity

Inicialment, el menú inicial i el d'opcions estaven implementats en el mateix script, ja que el disseny del joc estava pensat perquè aquests dos menús estiguessin únicament a la pantalla inicial. Però més endavant es va decidir separar-los en dos scripts diferents la implementació de cada menú, per si en un futur el menú d'opcions es volgués utilitzar en una altra escena.

```
1  public class OptionsMenu : MonoBehaviour
2  {
3      [SerializeField] private AudioMixer audioMixer;
4
5      public void FullScreen (bool fullScreen)
6      {
7          Screen.fullScreen = fullScreen;
8      }
9
10     public void VolumeChange(float volume)
11     {
12         audioMixer.SetFloat("Volume", volume);
13     }
14 }
```

Figura 9.3 Script OptionsMenu

El menú d'opcions consta d'un botó que si es prem, el joc entra en mode pantalla completa i per sortir de la pantalla completa s'ha de tornar a prémer el botó. Per fer això s'utilitza la funció FullScreen, que s'associa amb el botó utilitzant la funció per defecte d'Unity OnValueChanged(Boolean), que canvia el valor del booleà actual. També conté una barra de progrés que podem arrestar, quant més plena el volum serà més fort.

```
1 public class StartMenu : MonoBehaviour
2 {
3     public void Play()
4     {
5         SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);
6     }
7     public void Exit()
8     {
9         Application.Quit();
10    }
11 }
```

Figura 9.4 Script StartMenu

Per altra banda el menú d'inici pot mostrar els dos menús mencionats anteriorment, iniciar el joc o sortir de l'aplicació. Per iniciar el joc i sortir de l'aplicació s'utilitzen les dues funcions de la Figura 9.4, utilitzant l'OnClick() d'Unity mencionat anteriorment.

– 9.2 MENÚ DE PAUSA, VICTÒRIA I MORT-

Aquests tres menús són els que poden sortir mentre l'usuari està jugant la partida. Estan formats per GameObjects dintre d'un canvas. Els diferents botons dels menús es comuniquen amb la part lògica amb l'OnClick() explicat a l'apartat anterior.

Quan el jugador mort o la porta és destruïda es llança un esdeveniment per indicar-li al script Menu, que pari el joc i mostri el menú de mort. De la mateixa manera si el jugador acaba amb els enemics, es llança un esdeveniment per indicar que mostri el menú de victòria.

A la Figura 9.4 podem veure com a la funció Start() a la línia 5 i 6 està observant l'esdeveniment del jugador i porta que s'activarà quan un dels dos sigui destruït. Si això passa es cridarà la funció OpenMenu, que pararà el joc (línia 16) i mostrarà el menuDeath.
Per saber si ha eliminat a tots els enemics i mostrar el menú de victòria, escolta l'esdeveniment de la classe enemyCounter (encarregada de generar enemics) i crida la funció MenuVictory que fa el mateix que el OpenMenu, però amb el menú de victòria. A la línia 7 hi ha un if() que diferencia l'escena, ja que aquest mateix menú s'utilitza a totes les escenes i a la pantalla del jefe final no hi ha cap GameObject que generi enemics i que tingui associat el script enemyCounter.

```
private void Start()
    {
        player = GameObject.FindGameObjectWithTag("Player").GetComponent<Player>();
        castle = GameObject.FindGameObjectWithTag("Castle").GetComponent<Door>();
        player.playerDeath += OpenMenu;
        castle.destroyDoor += OpenMenu;
        if (Equals(scene, "dungeon"))
        {
            enemyCounter.victory += MenuVictory;
        }
    }
private void OpenMenu(object sender, EventArgs e)
    {
        pauseGame = true;
        Time.timeScale = 0f;
        menuDeath.SetActive(true);
    }
public void MenuVictory(object sender, EventArgs e)
    {
        pauseGame = true;
        Time.timeScale = 0f;
        menuVictory.SetActive(true);
    }
```

Figura 9.4 Lògica del menú victòria i mort de la classe Menu

Per cridar el menú de pausa, es pot clicar la tecla Esc del teclat, un cop premuda es crida el menú igual que amb els dos anteriors.

```
private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Escape))
        {
            if (pauseGame)
            {
                Resume();
            }
            else
            {
                Pause();
            }
        }
    }

public void Pause()
    {
        pauseGame = true;
        Time.timeScale = 0f;
        menuPause.SetActive(true);
    }
    public void Resume()
    {
        pauseGame = false;
        Time.timeScale = 1f;
        menuPause.SetActive(false);
    }
```

Figura 9.5 Lògica del menú de pausa de la classe Menu

– 9.3 TORRES -

Les torres són elements que pot invocar el jugador durant les pantalles en les quals ha de defensar la porta de les onades dels enemics. Per tant, aquestes torres s'han hagut de convertir en prefabs, per a poder instanciar-les quan el jugador ho necessiti.

Ara es veurà com s'ha implementat l'estructura i lògica de les torres. Com podem veure a l'apartat 8.5, les torres les podem dividir en 3 parts: la base, l'arma i el projectil que disparen. Seguidament, es mostrarà com han estat implementades la torre d'arquers i la de verí, ja que el funcionament és força semblant.

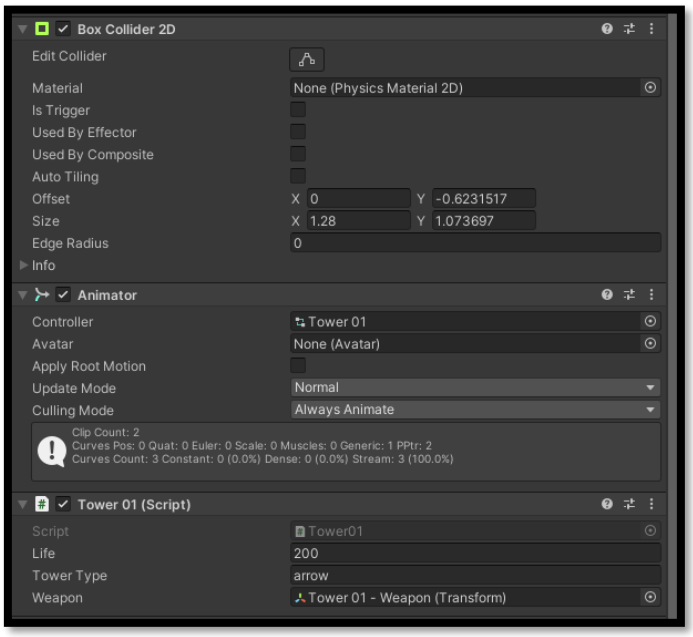

Figura 9.6 Inspector de la base de la torre

Les torres estan compostes per dos GameObjects, un fa de base i l'altre de l'arma. En la Figura 9.6 podem veure part de l'inspector de la base, on els components importants són el BoxCollider2D, ja que s'encarrega de fer la col·lisió de la torre perquè pugui rebre mal dels enemics. El segon component és l'animator, que únicament s'utilitza per a quan és destruïda (Figura 8.41). Per acabar, conté un script per poder setejar la vida inicial de la torre, un string per diferenciar quina torre és i un GameObject que serà l'arma.

```
1  void Start()
2      {
3          animator = GetComponent<Animator>();
4      }
5
6      public void TakeDamage(float damage)
7      {
8
9          life -= damage;
10         if (life <= 0)
11         {
12             Death();
13         }
14     }
15
16     private void Death()
17     {
18         animator.SetTrigger("Death");
19         if (Equals(towerType, "arrow"))
20         {
21             weapon.transform.GetComponent<BehaviourTower>().Death();
22         }
23         else if (Equals(towerType, "poison"))
24         {
25             weapon.transform.GetComponent<BehaviourPoisonTower>().Death();
26         }
27         else if(Equals(towerType, "cac"))
28         {
29             weapon.transform.GetComponent<BehaviourCacTower>().Death();
30         }
31     }
32     private void Destruir()
33     {
34         Destroy(gameObject);
35     }
```

Figura 9.7 Script Tower

Es pot observar que és un script força senzill, en el qual té una funció per anar baixant la vida de la torre (línia 6). Una altra funció, Death() que gestiona l'animació de morir de la torre i crida a la funció Death() del script de l'arma incorporada a la torre. Per diferenciar entre torres, ja que totes comparteixen aquesta classe per la base, s'utilitza un string.

Finalment, un recurs molt usat en aquest projecte, és cridar a funcions des del component Animation (explicat en l'apartat 7.4.1). Per veure com s'ha fet això es por observar la Figura 9.8. El component Animation, permet entre altres coses, afegir un esdeveniment en un instant de temps de l'animació. En aquest cas s'ha posat que en

l'últim instant de l'animació CollapseTower de la torre, es cridi la funció de la línia 32, Destruir() per eliminar el GameObject de la torre de la partida.

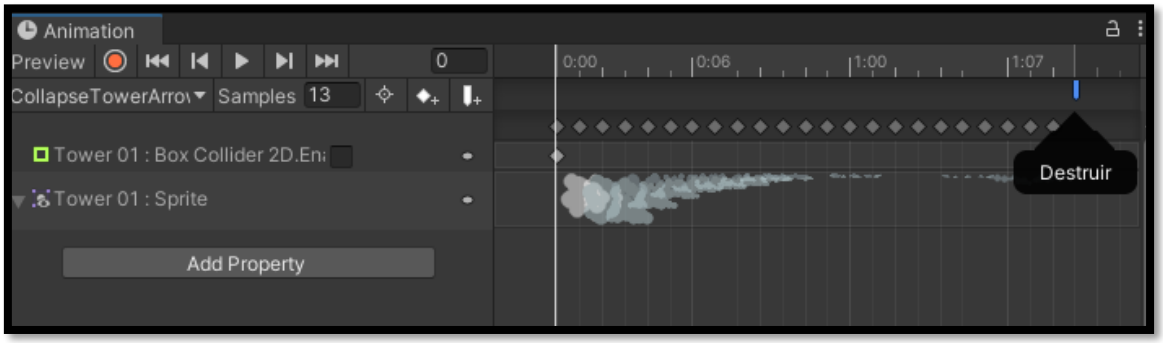

Figura 9.8 Animació de la base de la torre destruint-se

El segon component de la torre, l'arma, té associat un animator per fer l'animació d'atac i de mort. També té associat el l'script de la Figura 9.9.

```
1  void Start()
2      {
3          animator = GetComponent<Animator>();
4      }
5
6  void Update()
7      {
8          if (GameObject.FindGameObjectWithTag("Enemy"))
9          {
10             enemy = GameObject.FindGameObjectWithTag("Enemy").GetComponent<Transform>();
11         }
12         if (TimeNextAttack > 0)
13         {
14             TimeNextAttack -= Time.deltaTime;
15         }
16         if (enemy != null &&Vector2.Distance(transform.position, enemy.position) < MinimDistance)
17         {
18             float radianAngle = Mathf.Atan2(enemy.position.y - transform.position.y, enemy.position.x - transform.position.x);
19             float degreeAngle = (180 / Mathf.PI) * radianAngle - 90;
20             transform.rotation = Quaternion.Euler(0, 0, degreeAngle);
21
22             if(TimeNexAttack == 0){
23               towerAnimator.SetBool("Attack",true);
24               TimeNextAttack = TimeBetweenAttacks;
25             }
26         }
27         else
28         {
29             towerAnimator.SetBool("Attack",false);
30         }
31     }
32     private void Shoot()
33     {
34         GameObject bullet = Instantiate(bulletPrefab, transform.position, transform.rotation);
35         bullet.GetComponent<BulletMovement>().SetDirection(enemy);
36     }
37
38     public void Death()
39     {
40         towerAnimator.SetTrigger("Death");
41     }
42     private void Destruir()
43     {
44         Destroy(gameObject);
45     }
```

Figura 9.9 Script BehaviourTower

El funcionament de la destrucció del component funciona igual que la base, es crida a l'animació i des del component Animation es crida a la funció Destruir(). Les parts interessant són les funcions Update() i Shoo().

Les torres busquen a un enemic (línia 10), es redueix el temps de coldown de l'atac de la torre (línia 12 i 14), i llavors si es compleix que hi ha un enemic i que aquest està dintre el camp de visió (línia 16), llavors es mou l'arma de la torre perquè segueixi el moviment de l'enemic, és a dir que giri sobre el seu eix seguint a l'enemic (línia 18,19 i 20). Per acabar si coldown de l'habilitat està en 0 s'ataca i es torna a posar en coldown l'habilitat.

Quan una torre ataca, únicament es crida l'animació (línia 23). És aquesta que s'encarrega de cridar la funció Shoot(), a partir d'un esdeveniment, que instanciarà un projectil i li associarà el component que ha de seguir, en aquest cas l'enemic. Això es fa així, ja que els enemics no han de poder esquivar els projectils d'una torre.

El script del projectil té únicament dues funcions, un FixedUpdate(), que va movent la bala fins a l'enemic i un cop arriba fa li fa mal i es destrueix. Hi ha una funció anomenada SetDirection ,que és la que crida l'arma a l'hora d'invocar l'objecte, al qual li seteja l'objectiu del projectil.

Per acabar la secció de les torres, cal mencionar que la torre de màgia no instància cap objecte. El GamObject que fa d'arma, té associat un GameObject desactivat amb l' script de la Figura 9.10.

```
1  public void ContactExplosion()
2      {
3          Collider2D[] objects = Physics2D.OverlapCircleAll(transform.position, rangeAttack);
4          foreach (Collider2D colision in objects)
5          {
6              if (colision.CompareTag("Enemy"))
7              {
8                  colision.GetComponent<Enemy>().TakeDamage(damageAttack);
9              }
10             else if (colision.CompareTag("Player"))
11             {
12                 colision.GetComponent<Player>().TakeDamage(damageAttack);
13             }
14         }
15     }
16
17     private void OnDrawGizmos()
18     {
19         Gizmos.color = Color.yellow;
20         Gizmos.DrawWireSphere(transform.position, rangeAttack);
21     }
```

Figura 9.10 Funcions importants del Script Explosion

En aquestes funcions podem veure que s'utilitza un gestor de col·lisions, que fa una àrea circular al voltant de l'objecte associat. Com aquesta torre també pot fer mal al jugador, la col·lisió de l'àrea generada comprova si està fent contacte amb un Enemy o Player. Finalment, la funció OnDrawGizmos() dibuixa l'àrea d'aquesta col·lisió a l'editor d'Unity perquè sigui més fàcil dissenyar el joc (veure Figura 9.11).

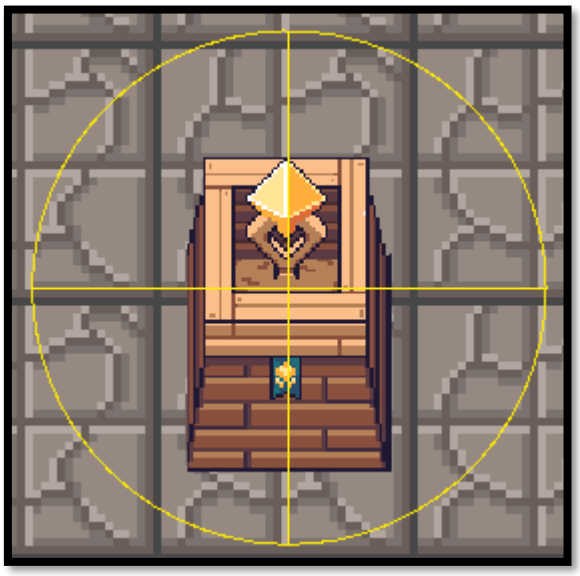

Figura 9.11 Àrea de col·lisió dibuixada per OnDrawGizmos()

– 9.4 JUGADOR -

El jugador, com s'ha mencionat en l'apartat 8.9, té associat 3 scripts on estan implementades totes les funcionalitat. El primer, el script Player, gestiona la vida de l'enemic amb la funció TakeDamage().

També gestiona l'efecte de cremat del jugador quan una bola de foc, de la pantalla final del joc, impacta amb el jugador. Per fer-ho canvia el color del sprite del jugador i al cap de dos segons amb la funció Invoke(), cridem a la funció DeBurned() per tornar al color original del sprite del jugador.

La funció Death es crida des de l'Animation del jugador, i crida l'esdeveniment que està observant la classe Menu, vista a la Figura 9.4.

```
1  public void TakeDamage(float damage)
2      {
3          life -= damage;
4          lifeBar.ChangeActuallyLife(life);
5          if(life <= 0)
6          {
7              rb2D.constraints = RigidbodyConstraints2D.FreezeAll;
8              animator.SetTrigger("Death");
9              Physics2D.IgnoreLayerCollision(LayerMask.NameToLayer("Player"), LayerMask.NameToLayer("Enemy"), true);
10         }
11     }
12     public virtual void Burned()
13     {
14         spriteRenderer.color = new Color(0.7f, 0.3f, 0.3f, 1);
15         Invoke("DeBurned", 2f);
16     }
17     private void DeBurned()
18     {
19         spriteRenderer.color = new Color(1, 1, 1, 1);
20     }
21     public void Death()
22     {
23         playerDeath?.Invoke(this, EventArgs.Empty);
24     }
```

Figura 9.12 Funcions script Player

El personatge principal té l'atac bàsic i una habilitat en àrea que congela. Per implementar això s'han associat dos GameObjects al jugador, un per cada atac, que únicament tenen un BoxCollider2D. Des del script PlayerController es gestiona tots els inputs del jugador i la posició del BoxCollider2D de l'atac bàsic, ja que ataca en la direcció que està mirant el jugador.

Si s'observa la Figura 9.13, es veu la funció Update(), on es comença guardant els inputs horitzontal i vertical i s'acaba creant una direcció, que és cap a la que es dirigeix el jugador. Al ser un joc píxel art amb vista zenital, el jugador ha de tenir una animació per cada direcció (a dalt, a baix, esquerra i dreta), per gestionar això es crea al component animator un Blend Tree (veure Figura 9.14) que gestiona quina animació mostra en funció d'uns valors (Horizontal i Vertical). Tant el jugador com els enemics tenen un Blend Tree per cada animació (atacar, morir, caminar, etc..).

Si es continua veient la Figura 9.13 veiem que de la línia 12 a la 15 es redirigeix el BoxCollider2D depenent de la direcció del jugador. Finalment, la resta del codi del Update(), serveix per configurar les tecles amb les diferents accions del jugador.

```
private void Update()
    {
        float horizontal = Input.GetAxisRaw("Horizontal");
        float vertical = Input.GetAxisRaw("Vertical");
        direccion = new Vector2(horizontal, vertical).normalized;

        playerAnimator.SetFloat("Horizontal", horizontal);
        playerAnimator.SetFloat("Vertical", vertical);
        playerAnimator.SetFloat("Speed", direccion.sqrMagnitude);


        if (horizontal > 0) { colBasicAttack.offset = new Vector2(posColX, posColY); }
        else if (horizontal < 0) { colBasicAttack.offset = new Vector2((float)(-2), posColY); }
        else if (vertical < 0) { colBasicAttack.offset = new Vector2((float)(-1), (float)(-0.5)); }
        else if (vertical > 0) { colBasicAttack.offset = new Vector2((float)(-1), (float)(0.5)); }

        if(TimeNextDodge > 0)
        {
            TimeNextDodge -= Time.deltaTime;
        }

        if(TimeBetweenSkill > 0)
        {
            TimeNextSkill -= Time.deltaTime;
        }

        if (Input.GetMouseButtonDown(0))
        {
            playerAnimator.SetTrigger("BasicAttack");
            TimeNextDodge = TimeBetweenDodges;
        }
        if (Input.GetKey(KeyCode.Alpha1) && coins > 3)
        {
            BuildTower(1);
            coins -= 4;
            coinCounter.substractPoints(2);
        }
        if (Input.GetKey(KeyCode.Alpha2) && coins > 5)
        {
            BuildTower(2);
            coins -= 6;
            coinCounter.substractPoints(4);
        }
        if (Input.GetKey(KeyCode.F) && TimeNextSkill <= 0)
        {
            playerAnimator.SetTrigger("Skill1");
            TimeNextSkill = TimeBetweenSkill;
        }
        if (Input.GetKey(KeyCode.Space) && TimeNextDodge <= 0)
        {
            playerAnimator.SetTrigger("Skill2");
            TimeNextDodge = TimeBetweenDodges;
        }
    }
```

Figura 9.13 Update() script PlayerController

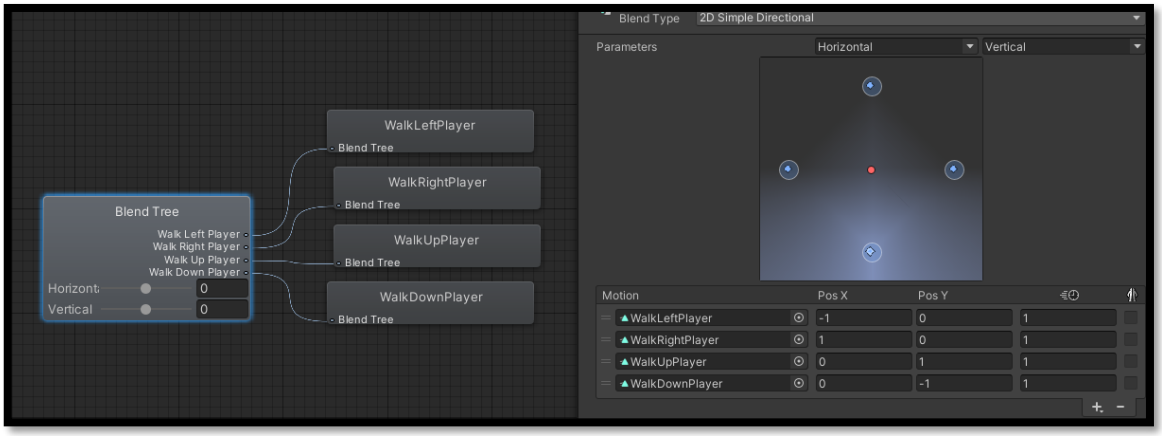

Figura 9.14 Blend Tree de l'animació caminar del jugador

Els BoxCollider2D dels atacs del jugador, sempre estan desactivats. Quan ataca, l'Animation activa els activa en el frame corresponent al moment de l'impacte, és el script CombatCaC que s'encarrega de gestionar aquesta col·lisió, veure figura 9.15.

```
1  private void OnTriggerEnter2D(Collider2D colision)
2      {
3          if(colision.CompareTag("Enemy"))
4          {
5              if(attackType == 0)
6              {
7                  colision.transform.GetComponent<Enemy>().TakeDamage(damageAttack);
8              }
9              else if(attackType == 1)
10             {
11                 colision.transform.GetComponent<Enemy>().TakeDamage(damageSkill_1);
12                 colision.transform.GetComponent<Enemy>().Frost();
13                 colision.transform.GetComponent<IAController>().Frosting();
14             }
15         }
16         else if (colision.CompareTag("Boss"))
17         {
18             if (attackType == 0)
19             {
20                 colision.transform.GetComponent<Boss>().TakeDamage(damageAttack);
21             }
22             else if (attackType == 1)
23             {
24                 colision.transform.GetComponent<Boss>().TakeDamage(damageSkill_1);
25                 colision.transform.GetComponent<Boss>().Frost();
26                 colision.transform.GetComponent<BossController>().Frosting();
27             }
28         }
29     }
```

Figura 9.15 Funció OnTriggerEnter2D del script CombatCaC

– 9.5 GENERACIÓ ENEMICS -

Els enemics en aquest Tower Defense, s'han de poder generar de forma automàtica, per fer això s'han creat quatre punts, visibles únicament en l'editor d'Unity i invisibles durant el joc. Aquests quatre punts formen una àrea (Figura 9.16) on els enemics apareixeran.

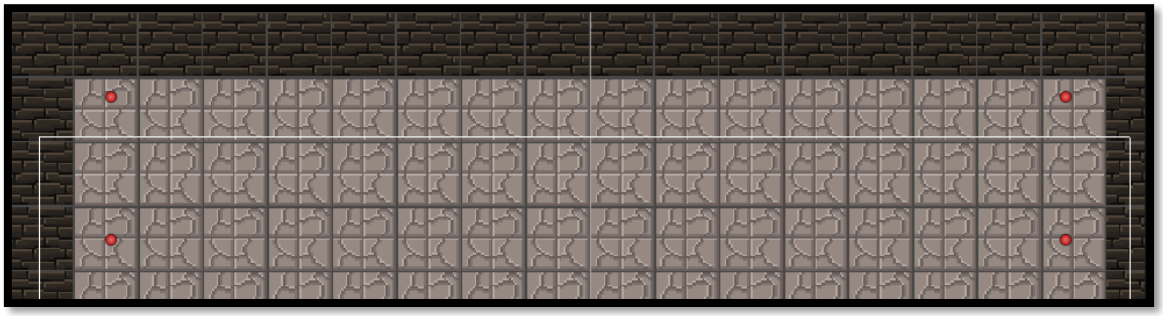

Figura 9.16 Punts limitadors de l'àrea on es generen els enemics

Seguidament, s'ha creat un GameObject on li afegim el script EnemyController, que s'encarregarà de generar aleatòriament els enemics. Per fer això necessitem passar-li els punts que conformen l'àrea i la llista d'enemics que poden aparèixer. Veure figura 9.17.

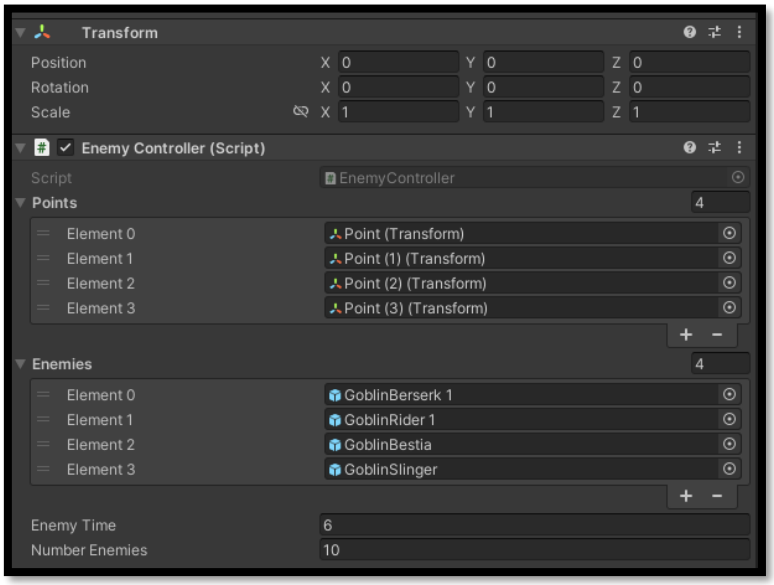

Figura 9.17 Inspector del GameObject encarregat de generar enemics

A la Figura 9.18 es pot veure el funcionament d'aquest script. A cada Update() es comprova que els enemics eliminats no siguin igual als enemics necessaris per guanyar, en cas contrari es genera un enemic nou si és que no s'han generat tots els enemics de la pantalla.

Per crear un enemic es crida a la funció Create_enemy(), que genera un aleatori dintre de la llista d'enemics que li hem passat a la Figura 9.17.

```
void Start()
    {
        maxX = points.Max(point => point.position.x);
        minX = points.Min(point => point.position.x);
        maxY = points.Max(point => point.position.y);
        minY = points.Min(point => point.position.y);
        countEnemiesCreate = 0;
    }


    void Update()
    {
        enemyNextTime += Time.deltaTime;

        if (enemyNextTime >= EnemyTime && countEnemiesCreate < numberEnemies)
        {
            enemyNextTime = 0;
            Create_enemy();
            countEnemiesCreate++;
            if(countEnemiesCreate == 15 || countEnemiesCreate == 30 || countEnemiesCreate == 40)
            {
                EnemyTime--;
            }
        }
    }

    private void Create_enemy()
    {
        int numberEnemy = Random.Range(0, enemies.Length);
        Vector2 randomPosition = new Vector2(Random.Range(minX,maxX), Random.Range(minY,maxY));

        Instantiate(enemies[numberEnemy], randomPosition, Quaternion.identity);
    }
```

Figura 9.18 Lògica creació aleatòria dels enemics

– 9.6 ENEMICS -

Ara s'explicarà com s'ha implementat els enemics de la segona pantalla, és a dir tots els enemics exceptuant al jefe final. Tots ells comparteixen dos scripts. El primer és la classe Enemy, la qual gestiona l'estat de l'enemic durant el transcurs de la seva existència en la partida. S'encarrega de rebre el mal amb la funció TakeDamage (float damage) el qual baixa la vida de l'enemic, i si aquesta arriba a 0, inicia el procés de morir. També s'encarrega de l'estat de congelat i d'enverinat. Per congelar-lo, es modifica el color de l'sprite enemic amb el component spriteRenderer.color, i es crida l'animació "Frost" per simular que l'enemic es queda quiet. Es fa servir la funció Invoke() per cridar una última funció al cap de X segons, que descongelarà l'enemic.

La funció Poison() s'encarrega de calcular el 10% de la vida màxima de l'enemic i restar-li a la vida actual. També canvia de color el sprite de l'enemic per notar ràpidament qui enemic està essent objectiu de la torre de verí.

Per acabar, la funció Death(), gestiona la mort de l'enemic i cal destacar que avisa a la classe enemyCounter perquè sumi un valor al comptador d'enemics morts, i també envia al script PlayerController el valor de monedes, com a compensació per derrotar-lo i que el jugador pugui construir més torres.

```
public virtual void TakeDamage(float damage)
    {
        life -= damage;
        if (life <= 0)
        {
            Death();
        }
    }

    public virtual void Frost()
    {
        spriteRenderer.color = new Color(0, 1, 1, 1);
        if(life > 0)
        {
            animator.SetBool("Frost", true);
            Invoke("DeFrosted", 5f);
        }

    }
    private void DeFrosted()
    {
        spriteRenderer.color = new Color(1, 1, 1, 1);
        animator.SetBool("Frost", false);
        if(life <= 0)
        {
            Destroy(gameObject);
        }
    }
    public void Poison()
    {
        spriteRenderer.color = new Color (0, 1, 0, 1);
        float damage = (float)(maxLife * 0.1);
        TakeDamage(damage);
    }
    public virtual void Death()
    {
        processDeath = true;
        animator.SetTrigger("Death");
        enemyCounter.countEnemyDead();
        player.GetComponent<PlayerControllert>().GetCoins(valueCoins);
    }
```

Figura 9.19 Part de la lògica del script Enemy

La segona classe que comparteixen tots els enemics és la IAController, que gestiona els diferents comportaments dels enemics. Aquesta classe és bastant extensa, ja que està ple de getters perquè les diferents accions i funcions dels arbres de comportament utilitzin. S'ha cregut adient no ensenyar en aquesta memòria tots els getters existents. Per altra banda, la part important d'aquest script és la creació del Behavior Tree. Seguidament, es mostrarà un seguit d'imatges que són la implementació dels arbres mostrats en les Figures 8.45, 8.46, 8.47 i 8.48.

```
root = new ActiveSelector(
                    new Sequence(
                        new CIsTargetVisible(this, "player"),
                        new Monitor(
                            new CIsTargetVisible(this, "player"),
                            new ActiveSelector(
                                new Sequence(
                                    new CIsInRange(this, "player"),
                                    new Monitor(
                                        new CIsInRange(this, "player"),
                                        new Sequence(
                                          new AWait(0.5f),
                                          new Attack(this, "player", 2)
                                        )
                                    )
                                ),
                                new WalkToTarget(this, "player")
                            )

                       )
                    ),
                    new ActiveSelector(
                        new Sequence(
                            new CIsTargetVisible(this, "tower"),
                            new Monitor(
                                new CIsTargetVisible(this, "tower"),
                                new ActiveSelector(
                                    new Sequence(
                                        new CIsInRange(this, "tower"),
                                        new Sequence(
                                          new AWait(0.5f),
                                          new Attack(this, "player", 2)
                                        )
                                    ),
                                    new WalkToTarget(this, "tower")
                                )
                            )

                       ),
                        new ActiveSelector(
                                new Sequence(
                                    new CIsInRange(this, "castle"),
                                    new Sequence(
                                          new AWait(0.5f),
                                          new Attack(this, "player", 2)
                                     )
                                ),
                                 new WalkToTarget(this, "castle")
                         )
                    )
                );
```

Figura 9.20 Lògica del Behavior Tree del Goblin Berserk

A la Figura 9.20 es pot veure la implementació de l'arbre de comportament del goblin berserk. Es destaquen tres blocs de codi, el primer gestiona el comportament del goblin amb el jugador, el segon amb la torre i per acabar amb el castell. Veient clarament la prioritat d'atac de cada objectiu.

```
root = new ActiveSelector(
                    new Sequence(
                            new CIsTargetVisible(this, "tower"),
                            new Monitor(
                                new CIsTargetVisible(this, "tower"),
                                new ActiveSelector(
                                    new Sequence(
                                        new CIsInRange(this, "tower"),
                                        new Monitor(
                                          new CIsInRange(this, "tower"),
                                          new Sequence(
                                            new AWait(0.5f),
                                            new Skill1(this, "tower")
                                          )
                                        )
                                    ),
                                    new WalkToTarget(this, "tower")
                                )
                            )

                    ),
                    new ActiveSelector(
                     new Sequence(
                        new CIsTargetVisible(this, "player"),
                        new Monitor(
                            new CIsTargetVisible(this, "player"),
                            new ActiveSelector(
                                new Sequence(
                                    new CIsInRange(this, "player"),
                                    new Monitor(
                                      new CIsInRange(this, "player"),
                                      new Sequence(
                                        new AWait(0.5f),
                                        new Attack(this, "player",2)
                                      )
                                    )
                                ),
                                new WalkToTarget(this, "player")
                            )

                       )
                    ),
                        new ActiveSelector(
                                new Sequence(
                                    new CIsInRange(this, "castle"),
                                    new AWait(0.5f),
                                    new Skill1(this, "castle")
                                ),
                                 new WalkToTarget(this, "castle")
                        )
                    )
                );
```

Figura 9.21 Lògica del Behavior Tree del Goblin Beast

A la Figura 9.21 es pot observar un arbre de comportament molt semblant al de la 9.20. Però amb les prioritats invertides. També podem observar que quan ataca a la torre o a la porta utilitza la classe Skill1. En aquests arbres podem veure com sempre li passem per referència l'element a qui li està fent target i la classe IAController per a què pugui accedir als getters que necessitin cada acció i condició.

```
root = new ActiveSelector(
                        new Sequence(
                        new CIsTargetVisible(this, "player"),
                        new Monitor(
                            new CIsTargetVisible(this, "player"),
                            new ActiveSelector(
                                new Sequence(
                                    new CIsInRange(this, "player"),
                                    new Monitor(
                                      new CIsInRange(this, "player"),
                                      new Sequence(
                                        new AWait(0.5f),
                                        new Attack(this, "player", 2)
                                      )
                                    )
                                ),
                                new WalkToTarget(this, "player")
                            )

                       )
                    ),
                        new ActiveSelector(
                                new Sequence(
                                    new CIsInRange(this, "castle"),
                                    new AWait(0.5f),
                                    new Attack(this, "castle", 2)
                                ),
                                 new WalkToTarget(this, "castle")
                        )
            );
```

Figura 9.22 Lògica del Behavior Tree del Goblin Rider

A la Figura 9.23 es pot observar l'arbre de comportament del goblin rider. Es veu com únicament té en compte el jugador i la porta. En tots els arbres de comportament en lloc de porta posa "castle", ja que a la majoria de tower defense es defensa un castell en lloc d'una porta.

```
root = new ActiveSelector(
               new Sequence(
                  new CIsInRange(this, "castle"),
                  new Await(0.5f),
                  new Attack(this, "castle", 4)
               ),
               new WalkToTarget(this, "castle")
         );
```

Figura 9.23 Lògica del Behavior Tree del Goblin Slinger

A la Figura 9.23 es veu un arbre de comportament força senzill, el qual fa que l'enemic camini sempre directament cap a la torre, si entra al rang de visió llavors es para i ataca.

El goblin slinger utilitza, a l'atacar, l'animation. Té un esdeveniment associat que crida la funció Shoot() del script Enemy. Aquesta funció invoca un prefab, una bala, que té com a objectiu la porta. El script del projectil és senzillament un FixedUpdate() que va movent la bala fins a la posició de la porta.

La implementació del jefe final és molt semblant, té un script BossController, que conté un seguit de getters per les accions i condicions de l'arbre de comportament del Boss. Ara es mostrarà com ha quedat implementat l'arbre de comportament de la Figura 8.49. També té el script Enemy comentat a l'inici d'aquest apartat. Veure Figura 9.23.

```
new ActiveSelector (
                    new ActiveSelector(
                           new Sequence(
                                new CAttackInRange(this, _RangeAttack),
                                new Monitor(
                                  new CAttackInRange(this, _RangeAttack),
                                  new Sequence(
                                    new AWait(1f),
                                    new AttackBoss(this, 4)
                                  )
                                )
                           ),
                           new Sequence (
                               new Inverter (new CAttackInRange(this, _Skill1Attack)),
                               new Parallel (Parallel.Policy.RequireAll,Parallel.Policy.RequireAll,
                                             new Sequence(
                                               new AWait(1f),
                                               new Skill1Boss(this, 4)
                                             ),
                                             new WalkToPlayer(this)
                               )
                           )
                     ),
                    new WalkToPlayer(this)
              );
```

Figura 9.23 Lògica del Behavior Tree del Jefe final

– 9.7 BEHAVIOR TREES -

Els behavior trees estan compostos per accions, condicions i classes que ajuden a poder fer composicions de les dues anteriors (Figura 8.44). Ara es mostrarà la implementació de cada una de les classes que componen aquests arbres de comportament.

Inicialment, la IA dels enemics estava implementada amb molts condicionals diferents (if()), es va decidir canviar-ho a Behavior trees, per fer el codi més escalable.

BehaviourEnemy

Classe principal que gestionarà tots els comportaments (accions i condicions) de l'arbre de comportament. Un comportament té un estat, el qual pot prendre qualsevol dels valors següents.

- Invalid: resultat inesperat d'actualitzar el comportament.
- Success: el comportament s'ha actualitzat amb èxit.
- Failure: el comportament s'ha actualitzat sense èxit.
- Running: el comportament se segueix executant.
- Avorteig: comportament avortat.

A la classe principal també s'han declarat alguns mètodes: Reset(), Abort(), IsTerminated(), IsRunning(), GetStatus(), per ajudar-nos a obtenir i modificar l'estat actual del comportament d'acord amb les nostres necessitats. Veure Figura 9.23.

```
public enum Status
    {
        Invalid,
        Success,
        Failure,
        Running,
        Aborted
    }
public abstract class BehaviorEnemy
    {
        private Status status;

        public abstract Status Update();

        public virtual void OnInitialize() { }
        public virtual void OnTerminate(Status status) { }

        public BehaviorEnemy()
        {
            status = Status.Invalid;
        }

        public Status Tick()
        {
            if (status != Status.Running)
                OnInitialize();
            status = Update();

            if (status != Status.Running)
                OnTerminate(status);

            return status;
        }

        public virtual void Reset()
        {
            status = Status.Invalid;
        }

        public virtual void Abort()
        {
            OnTerminate(Status.Aborted);
            status = Status.Aborted;
        }

        public bool IsTerminated()
        {
            return status == Status.Success
                || status == Status.Failure;
        }

        public bool IsRunning()
        {
            return status == Status.Running;
        }

        public Status GetStatus()
        {
            return status;
        }
    }

}
```

Figura 9.23 Lògica de la classe BehaviourEnemy

Composite

Component que serveix per emmagatzemar diversos comportaments en una llista. Aquest component mai es cridarà com a tal en un arbre de comportament, sinó que es crearan altres components que agafaran aquest com a pare per descriure un comportament més específic. Veure Figura 9.24.

```
protected List<BehaviorEnemy> children;

        protected Composite(BehaviorEnemy behavior,
            params BehaviorEnemy[] behaviors)
        {
            children = new List<BehaviorEnemy>();

            if (behavior != null)
                children.Add(behavior);

            if (behaviors != null)
            {
                for (int i = 0; i < behaviors.Length; i++)
                    children.Add(behaviors[i]);
            }
        }

        public override void Reset()
        {
            base.Reset();

            for (int i = 0; i < children.Count; i++)
                children[i].Reset();
        }

        public override void Abort()
        {
            base.Abort();

            for (int i = 0; i < children.Count; i++)
                children[i].Abort();
        }
```

Figura 9.24 Lògica de la classe Composite

Sequence

La classe executa comportaments en seqüència. Per tant, per emmagatzemar comportaments, utilitzarà com a classe pare el Composite. Podem veure a la lògica de la Figura 9.25, que s'inicialitza amb el primer component de la llista de comportaments, i en cada Update() gestiona la llista de comportaments.

```
public Sequence(BehaviorEnemy behavior,params BehaviorEnemy[] behaviors)
            : base(behavior, behaviors) { }

        public override void OnInitialize()
        {
            index = 0;
            currentChild = children[index];
        }

        public override Status Update()
        {
            while (true)
            {
                Status s = currentChild.Tick();

                if (s != Status.Success)
                    return s;

                if (++index == children.Count)
                    return Status.Success;

                currentChild = children[index];
            }
        }
```

Figura 9.25 Lògica de la classe Sequence

Selector

Una altra classe que hereta del Composite. A diferència del Sequence, aquest no va executant els comportaments en ordre, sinó que selecciona un dels diferents comportaments i l'executa. Veure Figura 9.26.

```
public Selector(BehaviorEnemy behavior,params BehaviorEnemy[] behaviors)
            : base(behavior, behaviors) { }

        public override void OnInitialize()
        {
            index = 0;
            currentChild = children[index];
        }

        public override Status Update()
        {
            while (true)
            {
                Status s = currentChild.Tick();

                if (s != Status.Failure)
                    return s;

                if (++index == children.Count)
                    return Status.Failure;

                currentChild = children[index];
            }
        }
```

Figura 9.26 Lògica de la classe Selector

Aquesta implementació servia si utilitzàvem un Selector amb una condició seguida d'una acció. Doncs si la condició es complia executava sempre l'acció, i si no sempre executava la condició. Però a l'anar complicant l'arquitectura de l'IA, va començar a fer comportaments que no es volien: ignorar el jugador quan entra al camp de visió, objectiu dintre del rang d'atac i no atacar, etc. Això era a causa de la tria passiva del Selector que s'havia implementat. Per arreglar-lo es va crear una classe filla que fos un selector més específic, la classe ActiveSelector.

ActiveSelector

```
public class ActiveSelector : Selector
    {
        public ActiveSelector(BehaviorEnemy behavior,params BehaviorEnemy[] behaviors)
            : base(behavior, behaviors) { }

        public override void OnInitialize()
        {
            index = children.Count - 1;
            currentChild = children[index];
        }

        public override Status Update()
        {
            int previous = index;

            base.OnInitialize();
            Status result = base.Update();

            if (previous != children.Count && index != previous)
                children[previous].Abort();

            return result;
        }
    }
```

Figura 9.26 Lògica de la classe ActiveSelector

Parallel

Executa comportaments simultàniament. Té un constructor per paràmetres que se li indica, que executi tots els comportaments quan almenys un retorni èxit, o per altra banda, quan tots retornin èxit.

En el mètode Upate() es pot observar com va anotant cada comportament de la llista si ha retornat èxit o fallit, per més endavant comparar-ho amb tots els comportaments, si han de donar èxit o no. Veure Figura 9.27.

```
public enum Policy
        {
            RequireOne,
            RequireAll
        }

        protected Policy successPolicy;
        protected Policy failurePolicy;

        protected Parallel(Policy forSuccess, Policy forFailure)
            : this(forSuccess, forFailure, null, null) { }

        public Parallel(Policy forSuccess, Policy forFailure,
            BehaviorEnemy behavior, params BehaviorEnemy[] behaviors)
            : base(behavior, behaviors)
        {
            successPolicy = forSuccess;
            failurePolicy = forFailure;
        }

        public override Status Update()
        {
            int successCount = 0;
            int failureCount = 0;

            for (int i = 0; i < children.Count; i++)
            {
                BehaviorEnemy b = children[i];

                if (!b.IsTerminated())
                    b.Tick();

                if (b.GetStatus() == Status.Success)
                {
                    ++successCount;
                    if (successPolicy == Policy.RequireOne)
                        return Status.Success;
                }

                if (b.GetStatus() == Status.Failure)
                {
                    ++failureCount;
                    if (failurePolicy == Policy.RequireOne)
                        return Status.Failure;
                }
            }

            if (failurePolicy == Policy.RequireAll
                && failureCount == children.Count)
                return Status.Failure;

            if (successPolicy == Policy.RequireAll
                && successCount == children.Count)
                return Status.Success;

            return Status.Running;
        }

```

Figura 9.27 Lògica de la classe Parallel

Monitor

El monitor és una classe filla de Parallel. Hi ha situacions en les quals volem que s'executi una acció sempre que una condició doni èxit, i quan doni fallit que l'acció es deixi d'executar. Veure Figura 9.28.

```
public class Monitor : Parallel
    {

        public Monitor(BehaviorEnemy condition, BehaviorEnemy action)
        : base(Policy.RequireOne, Policy.RequireOne)
        {
              children.Add(new ToMonitoringMode(condition));
              children.Add(action);
        }

        private class ToMonitoringMode : Decorator
        {
            public ToMonitoringMode(BehaviorEnemy child)
            : base(child) { }

            public override Status Update()
            {
                Status childStatus = child.Update();

                if (childStatus == Status.Success)
                    return Status.Running;
                else
                    return Status.Failure;
            }
        }

    }
```

Figura 9.28 Lògica de la classe Monitor

En el codi es pot veure com en la funció Update(), comprova si l'estat de la condició retorna èxit (Success), en aquest cas l'estat continua corrent. En cas contrari l'estat de l'acció passa a estat fallit.

Decorator

Aquesta classe envolta un node existent i li afegeix funcionalitats o restriccions addicionals. Aquesta classe igual que el Composite, no s'utilitza directament en l'arbre de comportament, sinó que s'ha creat classes filles que donen funcionalitats específiques. Veure Figura 9.29.

```
protected BehaviorEnemy child;

        public Decorator(BehaviorEnemy child)
        {
            this.child = child;
        }

        public override void Reset()
        {
            base.Reset();
            child.Reset();
        }

        public override void Abort()
        {
            base.Abort();
            child.Abort();
        }
    }
```

Figura 9.29 Lògica de la classe Decorator

Inverter

Per estalviar haver d'implementar altres condicions que comprovin el mateix, però amb el resultat invertit, es va decidir implementar aquesta classe que canvia el resultat d'una condició. D'aquesta manera el disseny de classes queda més net, a més de repetir codi innecessàriament. Veure Figura 9.30.

```
1  public class Inverter : Decorator
2     {
3         public Inverter(BehaviorEnemy child)
4             : base(child) { }
5
6         public override Status Update()
7         {
8             Status childStatus = child.Update();
9
10            if (childStatus == Status.Success)
11                return Status.Failure;
12            else
13                return Status.Success;
14        }
15    }
```

Figura 9.30 Lògica de la classe Inverter

Repeat

Aquesta és una classe que a l'inici del desenvolupament de l'arquitectura de l'IA, es va implementar, per fer que cada enemic executes l'acció Attack, un número de vegades diferent. Un cop implementat, el resultat no acabava d'encaixar amb la idea que es tenia del projecte. Per tant, es va decidir que els enemics ataquessin únicament una vegada i fessin una pausa, amb l'acció AWait, entre atac i atac.

Tot i així es mostrarà el codi d'aquesta classe, ja que si el projecte es continués desenvolupant probablement s'utilitzaria. Veure Figura 9.31.

```
public class Repeat : Decorator
   {
       protected int limit;
       protected int counter;

       public Repeat(int limit, BehaviorEnemy child)
           : base(child)
       {
           this.limit = limit;
       }

       public override void OnInitialize()
       {
           counter = 0;
       }

       public override Status Update()
       {
           while (true)
           {
               child.Tick();

               if (child.GetStatus() == Status.Running)  return Status.Running;
               if (child.GetStatus() == Status.Failure)  return Status.Failure;
               if (++counter == limit) return Status.Success;
           }
       }
   }
```

Figura 9.31 Lògica de la classe Repeat

Ara es mostrarà les implementacions de les accions i condicions usades en els arbres de decisió dels enemics.

CIsTargetVisible

Els enemics, un cop apareixen, van directe cap a la porta a destruir-la. Però pel camí poden trobar-se amb torres o el jugador. Alguns enemics donen prioritat a atacar al jugador o torres per davant de la porta. Per tant, es va implementar aquesta condició per comprovar quan un element entrat per paràmetre, entrava dins el seu rang de visió. Veure Figura 9.32.

```
public CIsTargetVisible(IAController controller,string t)
    {

        this.controller = controller;
        this.t = t;
    }
    public override Status Update()
    {
        if (Equals(t, "tower"))
        {
            this.target = controller._towerr;
        }
        else if (Equals(t, "player"))
        {
            this.target = controller._Player;
        }
        else if (Equals(t, "castle"))
        {
            this.target = controller._Castle;
        }

        if (target != null && Vector2.Distance(controller.transform.position, target.position) < controller._MaxDstToTarget)
        {
            return Status.Success;
        }
        else
        {
            return Status.Failure;
        }
    }
```

Figura 9.32 Lògica de la classe CIsTargetVisible

A la majoria d'accions i condicions se li passa la IAController, per accedir als getters de la classe. Inicialment, es va fer un CIsTargetVisible per cada element (torre ,jugador ,porta). Per no repetir codi innecessàriament, en les condicions també se li passa un string t, per diferenciar a quin element li estem fent target.

WalkToTarget

Acció que va molts cops acompanyada de la condició vista prèviament. Quan un element entra al camp de visió el que fa l'enemic inicialment és dirigir-se cap a ell. Aquesta funció fa que l'enemic camini cap a la direcció de l'enemic i el segueixi sempre que estigui provoca que l'enemic es dirigeixi cap a la posició de l'element que té com a objectiu. Retornarà fallit si l'element desapareix, per exemple si es dirigeix cap a una torre que ha destruït un altre enemic. Veure Figura 9.33.

```
public override Status Update()
        {
            if (Equals(t, "tower"))
            {
                this.target = controller._towerr;
            }
            else if (Equals(t, "player"))
            {
                this.target = controller._Player;
            }
            else if (Equals(t, "castle"))
            {
                this.target = controller._Castle;
            }

            if (target != null)
            {
                controller._animator.SetFloat("Horizontal", target.position.x - controller.transform.position.x);
                controller._animator.SetFloat("Vertical", target.position.y - controller.transform.position.y);
                controller._animator.SetFloat("Speed", controller._direction.sqrMagnitude);

                controller.transform.position =
                  Vector2.MoveTowards(controller.transform.position, target.position, controller._speed * Time.deltaTime);

            return Status.Running;
            }
            else
            {
                return Status.Failure;
            }

        }
```

Figura 9.33 Lògica de la classe WalkToTarget

CIsInRange

Aquesta condició que retorna èxit si l'element que té com a objectiu entra en el rang d'atac, altrament fallit. Veure Figura 9.34.

```
public CIsInRange(IAController controller, string t)
    {
        this.controller = controller;
        this.t = t;
    }
    public override Status Update()
    {
        if (Equals(t, "tower"))
        {
            this.target = controller._towerr;
        }
        else if (Equals(t, "player"))
        {
            this.target = controller._Player;
        }
        else if (Equals(t, "castle"))
        {
            this.target = controller._Castle;
        }

        if (target != null && Vector2.Distance(controller.transform.position, target.position) < controller._RangeAttack)
        {
            return Status.Success;
        }
        else
        {
            return Status.Failure;
        }
    }
```

Figura 9.34 Lògica de la classe CIsInRange

Attack

El script Attack (Figura 9.35) , primer comprova si el coldown de l'habilitat de l'enemic està en 0, en cas contrari col·loca aquest component GameObject que gestiona les col·lisions cap a la direcció en la qual mira l'enemic. Seguidament, inicia l'animació de l'atac. En l'Animation de l'animació d'atac, hi ha col·locat un esdeveniment que activa la funció de la Figura 9.36.

```
public Attack(IAController controller, string t, float timeBetweenAttacks)
    {
        this.controller = controller;
        this.typeTarget = t;
        this.timeBetweenAttacks = timeBetweenAttacks;
    }

    public override void OnInitialize()
    {
        timeNextAttack = 0;
    }
    public override void OnTerminate(Status status) { }

    public override Status Update()
    {

            if (timeNextAttack > 0)
            {
                timeNextAttack -= Time.deltaTime;
            }
            if (Equals(typeTarget, "tower"))
            {
                this.target = controller._towerr;
            }
            else if (Equals(typeTarget, "player"))
            {

                this.target = controller._Player;
            }
            else if (Equals(typeTarget, "castle"))
            {
                this.target = controller._Castle;
            }
        if(timeNextAttack <= 0)
        {

                float horizontal = target.position.x - controller._Me.position.x;
                float vertical = target.position.y - controller._Me.position.y;

                Vector3 temp = new Vector3(1, 0, 0);
                string pos = "right";

                if (horizontal > 0 & Math.Abs(horizontal) >= Math.Abs(vertical) & !Equals(pos, "right")) {
                  temp = new Vector3(1, 0, 0);
                }
                else if (horizontal < 0 & Math.Abs(horizontal) >= Math.Abs(vertical) & !Equals(pos, "left")) {
                  temp = new Vector3(-1, 0, 0); pos = "left";
                }
                else if (vertical < 0 & Math.Abs(horizontal) < Math.Abs(vertical) & !Equals(pos, "down")) {
                  temp = new Vector3(0, -1, 0); pos = "down";
                }
                else if (vertical > 0 & Math.Abs(horizontal) < Math.Abs(vertical) & !Equals(pos, "up")) {
                  temp = new Vector3(0, 1, 0); pos = "up";
                }

                controller._attac.transform.localPosition = temp;

                controller._animator.SetFloat("Horizontal", target.position.x - controller.transform.position.x);
                controller._animator.SetFloat("Vertical", target.position.y - controller.transform.position.y);
                controller._animator.SetFloat("Speed", controller._direction.sqrMagnitude);

                controller._animator.SetTrigger("Attack");

                timeNextAttack = timeBetweenAttacks;
            }


            return Status.Running;
    }
```

Figura 9.35 Lògica de la classe Attack

Els enemics tenen associat un GameObject, que fa de contenidor per la funció OverlapCircleAll, que s'utilitza per trobar tots els colliders 2D que se superposen amb un cercle especificat a l'espai del joc.

```
public void Attack()
   {
       Collider2D[] objects = Physics2D.OverlapCircleAll(atac.position, rangeAttackPhysics);
       foreach (Collider2D colision in objects)
       {
           if (colision.CompareTag("Player"))
           {
               colision.GetComponent<Player>().TakeDamage(damageAttack);
           }
           else if (colision.CompareTag("Tower"))
           {
                   colision.GetComponent<Tower01>().TakeDamage(damageAttack);
           }
           else if (colision.CompareTag("Castle"))
           {
               colision.GetComponent<Door>().TakeDamage(damageAttack);
           }
       }
   }
```

Figura 9.36 Funció Attack() de la classe IAController

Skill1

Aquesta acció es comporta de manera semblant a Attack, amb la diferència que crida una animació diferent, que aquesta conté un esdeveniment que crida una funció semblant a la 9.36.

AWait

Acció que atura el comportament actual de l'enemic durant uns segons determinats.

```
public AWait(float timeToWait)
    {
        this.timeToWait = timeToWait;
    }

    public override void OnInitialize()
    {
        timer = 0f;
    }

    public override Status Update()
    {
        timer += Time.deltaTime;

        if (timer >= timeToWait)
            return Status.Success;
        else
            return Status.Running;
    }
```

Figura 9.37 Lògica de la classe AWait

Es va decidir implementar aquesta classe per regular el temps entre atac i atac dels enemics, per així ajustar la dificultat i les característiques de cada enemic.

A la pantalla del jefe final, no es poden invocar torres ni tampoc li interessa destruir la porta. És un contra un. Per tant, moltes de les variables implementades en la classe IAController, no eren necessàries. Es va decidir crear un BossController, que funciones de manera molt semblant, repetint codi a vegades, però amb la intenció de fer el codi més net i ordenat.

Al fer això es va presentar un problema, algunes de les accions i condicions, demanaven com a paràmetre, la classe IAController, per accedir als getters. Es van presentar tres solucions:

- Adaptar les accions i condicions perquè acceptessin les dues classes.
- Crear accions i condicions noves pel jefe final, repetint codi en alguns casos.
- Crear una classe pare de la IAController i BossController, que contingués els getters necessaris.

La primera opció va quedar descartada de seguida, ja que deixava el codi molt brut.

La tercera era la que semblava la millor, però al començar a implementar-la van sorgir més errors dels previstos, i per tant una pèrdua de temps de desenvolupament imprevista. Llavors es va optar per implementar la segona opció i si quedava temps al final del projecte s'implementaria la tercera opció, finalment ha quedat pel treball futur.

A continuació es comentaran les funcions i accions creades pel jefe final.

WalkToPlayer

Aquesta acció fa que el Boss, camini cap al jugador. A diferència del WalkToTarget, no una fallida, ja que si el jugador mort es pausa la partida i s'obre el menú de mort. Veure Figura 9.38.

```
public override Status Update()
    {
        controller._animator.SetFloat("Horizontal", controller._PlayerPos.x - controller.transform.position.x);
        controller._animator.SetFloat("Vertical", controller._PlayerPos.y - controller.transform.position.y);
        controller._animator.SetFloat("Speed", controller._direction.sqrMagnitude);

        controller.transform.position = Vector2.MoveTowards(controller.transform.position, controller._PlayerPos, controller._speed *
 Time.deltaTime);

            return Status.Running;

    }
```

Figura 9.38 Lògica de la classe WalkToPlayer

CAttackInRange

Condició molt semblant a CIsInRange, amb la diferència que no fem les comprovacions prèvies per saber si l'element objectiu és una torre, el jugador o la porta.

Skill1Boss

Aquesta acció comprova que el coldown de l'habilitat del jefe estigui a 0. En cas afirmatiu inicia l'animació de la primera habilitat. Aquesta te un esdeveniment en el Animation que crida la funció Shoot() de l'script BossController, que instancia el projectil del jefe final.

A diferència dels altres projectils, aquest ha de girar sobre el seu eix, depenent de quina direcció ha d'anar. Per això es va crear una animació amb Blend Tree, que tingués quatre sprites diferents, un per cada direcció. Quan l'objecte s'instanciï, en el Start() de la classe BulletBoss se li passi la posició Horizontal i Vertical en la que ha d'anar. Veure Figures 9.39 i 9.40.

```
public override void OnInitialize()
    {
        timeNextAttack = 0;
    }
    public override void OnTerminate(Status status) { }

    public override Status Update()
    {

        if (timeNextAttack <= 0)
        {

            controller._animator.SetFloat("Horizontal", target.position.x - controller.transform.position.x);
            controller._animator.SetFloat("Vertical", target.position.y - controller.transform.position.y);
            controller._animator.SetFloat("Speed", controller._direction.sqrMagnitude);

            controller._animator.SetTrigger("Skill1");

            timeNextAttack = timeBetweenAttacks;
        }
        else
        {
            timeNextAttack -= Time.deltaTime;
        }

        return Status.Running;
    }
```

Figura 9.39 Lògica de la classe Skill1Boss

```
void Start()
    {
        animator = GetComponent<Animator>();
        player = GameObject.FindGameObjectWithTag("Player").GetComponent<Transform>();
        direction = player.position;
        animator.SetFloat("Horizontal", player.position.x - transform.position.x);
        animator.SetFloat("Vertical", player.position.y - transform.position.y);
    }


    void Update()
    {
        if(transform.position == direction)
        {
            Destroy(gameObject);
        }
        else
        {
            transform.position = Vector2.MoveTowards(transform.position, direction, speed * Time.deltaTime);
        }
    }
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            collision.GetComponent<Player>().TakeDamage(damage);
            Destroy(gameObject);
        }
    }
```

Figura 9.40 Lògica de la classe BulletBoss

– 9.8 PORTA -

La porta és un element que únicament rep mal. La seva importància recau en el fet que s'ha de protegir per evitar que sigui destruïda. Per tant, la implementació és molt senzilla: un script que contingui una funció que rebi el mal i el resti de la variable vida. I un esdeveniment per quan la porta sigui destruïda que cridi al script Menu i obri el menú de mort.

– 9.9 BARRA DE VIDA -

La barra de vida és un element dintre del canvas, que té associat tres GameObjects dintre seu: el marc, el color del farcit de la barra i la imatge per diferenciar de qui és la barra de vida. També té associat un script (Figura 9.41) que gestiona la barra de vida.

```
1  public class LifeBar : MonoBehaviour
2  {
3      private Slider slider;
4      void Start()
5      {
6          slider = GetComponent<Slider>();
7      }
8
9      public void ChangeMaxLife(float MaxLife)
10     {
11         slider.maxValue = MaxLife;
12     }
13     public void ChangeActuallyLife(float damage)
14     {
15         slider.value = damage;
16     }
17     public void InitializeLifeBar(float TotalLife)
18     {
19         ChangeMaxLife(TotalLife);
20         ChangeActuallyLife(TotalLife);
21     }
22 }
```

Figura 9.41 Classe LifeBar

El GameObject de la barra de vida té un component Slider que facilita el funcionament i ens estalvia haver d'implementar molta lògica. Veure Figura 9.42.

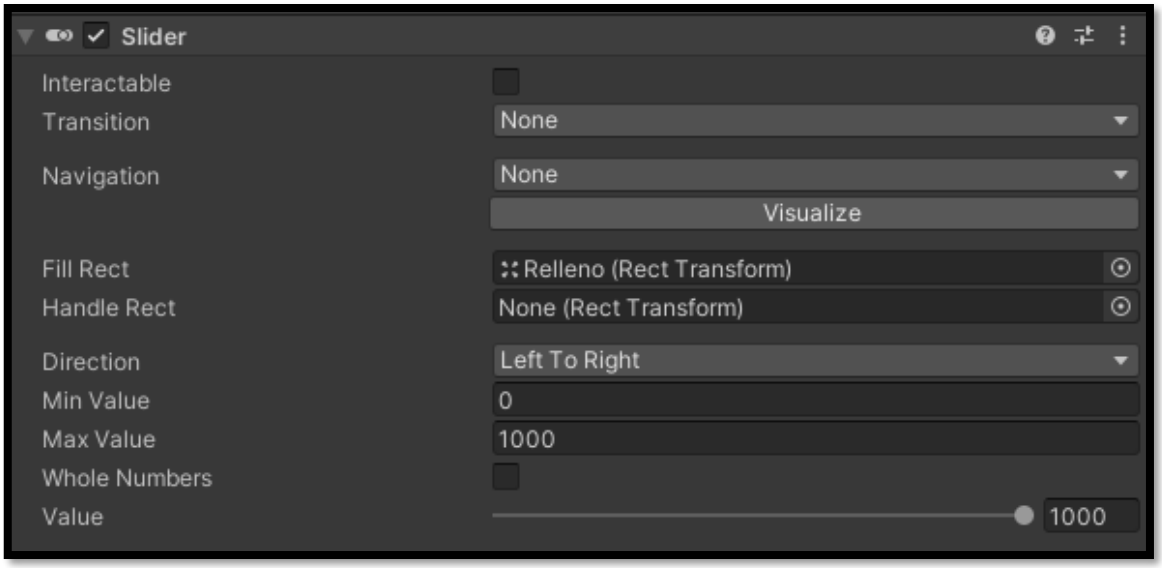

Figura 9.42 Component Slider

– 9.10 CONTADOR D'ENEMICS I DE MONEDES -

Els dos comptadors estan implementats de manera molt semblant. Tots dos tenen un GameObject al canvas que te un component de TextMeshPro, per editar el text dels comptadors. També tenen un script cadascun per anar augmentant el número del comptador. En el cas de les monedes també poden disminuir al construir una torre. El script d'enemics té un esdeveniment per a quan el comptador arriba al límit, notificar l'script Menu que pot obrir el menú victòria. Veure Figures 9.43 i 9.44.

```
    void Start()
    {
        textMeshPro = GetComponent<TextMeshProUGUI>();
        enemiesDead = 0;
    }

    void Update()
    {
        if (enemiesDead == numberEnemies)
        {
            victory?.Invoke(this, EventArgs.Empty);
        }

        textMeshPro.text = deadEenemies.ToString("0");
    }

    public void scorePoints(float ipoints)
    {
        deadEenemies += ipoints;
    }

    public float getEnemiesDead()
    {
        return deadEenemies;
    }

    public void countEnemyDead()
    {
        enemiesDead++;
    }
```

Figura 9.43 Classe EnemyCounter

```
    void Start()
    {
        textMeshPro = GetComponent<TextMeshProUGUI>();
    }
    void Update()
    {
        textMeshPro.text = points.ToString("0");
    }

    public void scorePoints(float ipoints)
    {
        points += ipoints;
    }
    public void substractPoints(float ipoints)
    {
        points -= ipoints;
    }
```

Figura 9.44 Classe CoinCounter

– 9.11 TILEMAP -

Els Tilemaps en Unity són una característica que permet representar i gestionar mosaics o "tiles" en un joc o escena. Cada "tile" és una peça gràfica que pot ser part d'un patró més gran. Aquesta tècnica s'ha utilitzat per a la creació dels mapes de cada pantalla.

El mapa s'ha dividit en dos components: el terra i les parets laterals que delimiten la pantalla. S'ha fet així per aplicar un Tilemap Collider2D a tots els tiles que conformen el component de paret. A la Figura 9.45 es pot veure amb verd els tiles de les parets, la resta són els tiles del terra.

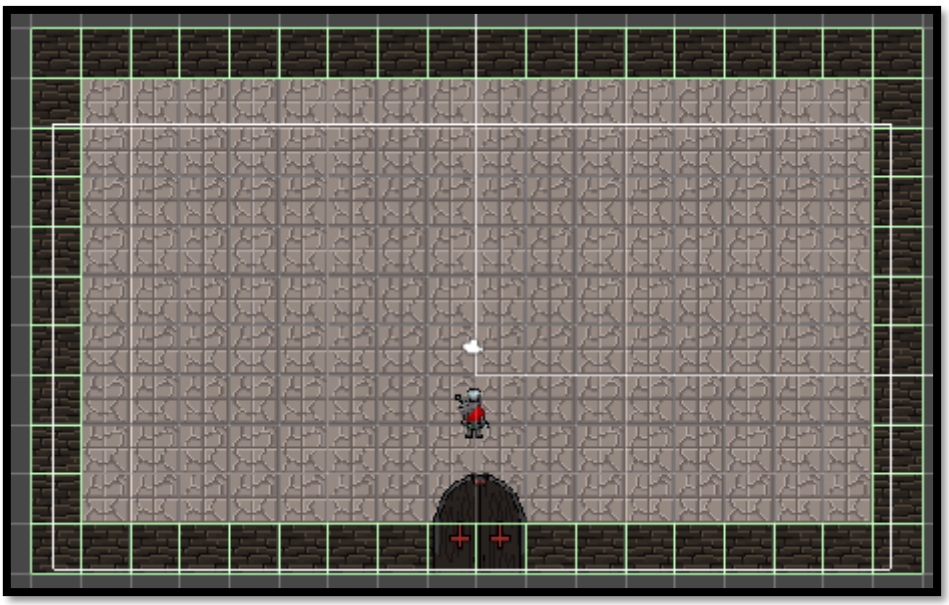

Figura 9.45 GameObject que conté els tiles de les parets

– 9.12 CÀMERA I MINIMAPA –

La càmera d'aquest joc és amb vista zenital, és a dir que es veu tot com si el jugador estigués situat al cel. Una decisió que es va prendre a la meitat del projecte va ser que la càmera no enfoques tot el mapa i fos estàtica, si no que fos dinàmica i es mogués amb el moviment del personatge.

Per fer això es va generar una càmera virtual, amb el component Cinemachine, que s'encarrega de gestionar el seguiment del personatge, estalviant així molta lògica a

implementar. Però amb això es genera un altre problema, la càmera pot enfocar fora dels límits del mapa.

Per solucionar-ho, es va crear un gameObject amb un polygonCollider2D i es va resseguir tot el perímetre del mapa. Després s'afegeix el component Cinemachine Confiner a la càmera virtual, i li passem el gameObject com a límit en el nou component. A la figura 9.46 es pot observar una línia groga que correspon a l'objecte creat per fer de límit del mapa, i a l'esquerra a l'Inspector podem observar els components Cinemachine.

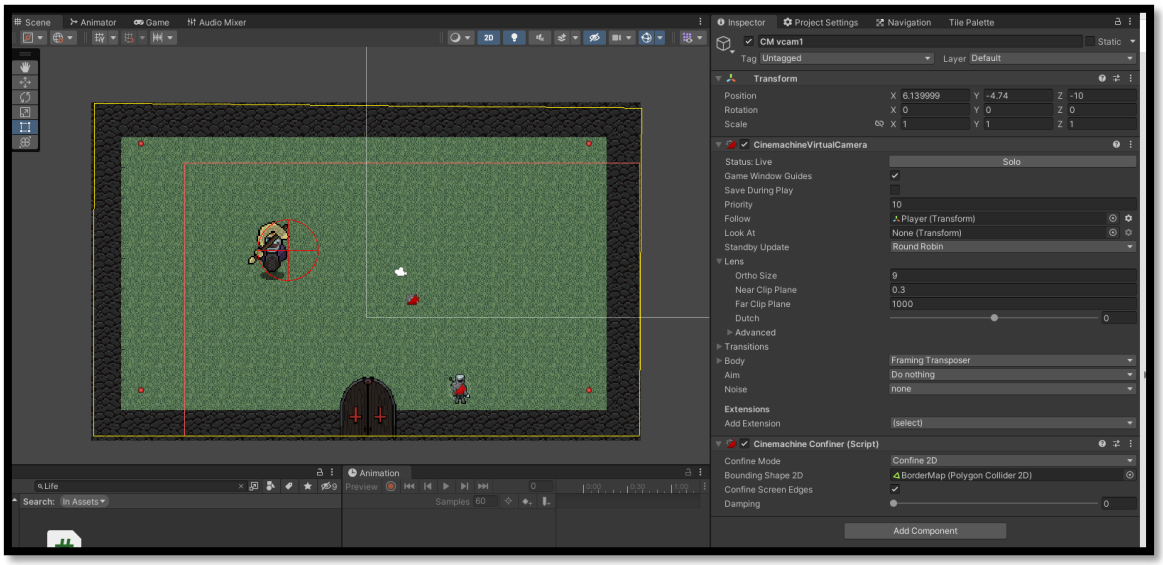

Figura 9.46 Límits del mapa i components Cinemachine

A l'aplicar aquest canvi de càmera, el jugador ja no pot veure tot el mapa, per tant, per ajudar-lo a fer front als enemics, es va decidir implementar un minimapa.

Es va crear un altre càmera pel minimapa que agafaria coma referència a la càmera principal, però disminuint el zoom per a què el jugador pugui veure molt més tros de mapa. Finalment, a cada enemic se li va crear un gameObject nou, que fos un punt Vermell (al jugador un punt blau) i es va posar en un layer (capa) que només la càmera del miniMapa pugui veure. D'aquesta manera en el minimapa els enemics sortiran representats amb un punt Vermell i el jugador amb un punt blau. Per gestionar el que pot veure una càmera o l'altre, existeix un component anomenat Culling Mask (Figura 9.47).

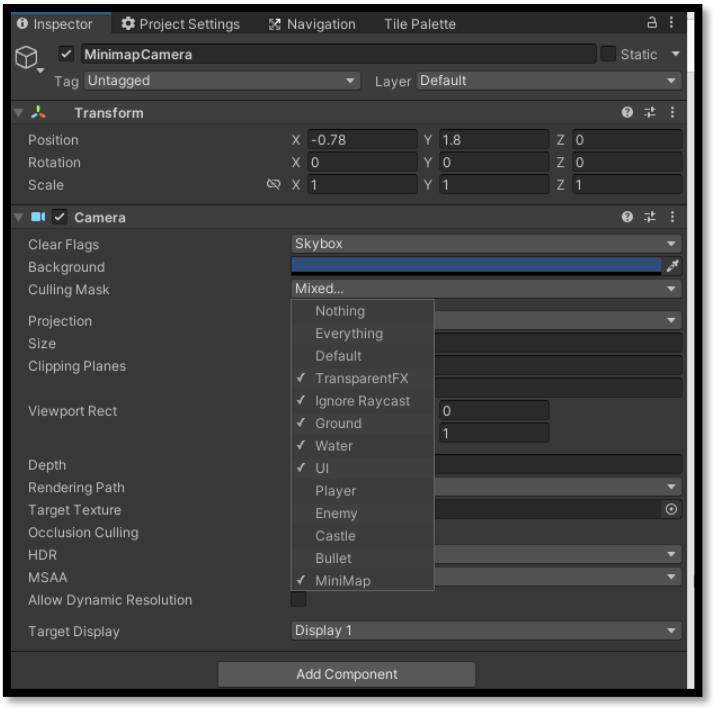

Figura 9.47 Culling Mask de la càmera del minimapa

– 9.13 MÚSICA I SONS -

Per reproduir les diferents músiques i sons, es va utilitzar el component audioSource. Aquest component permet reproduir sons i gestionar-ho des del codi.

Les músiques que sonen mentre estem jugant estan incorporades a la càmera, ja que segueix al jugador en tot moment.

A la Figura 9.48 podem veure un exemple del codi de la Figura 9.15 al qual fa referència a la gestió de l'atac del personatge principal, però aquest cop quan ataca podem veure que a la línia 6, 10, 20 i 25 executa un so depenent de l'atac que estigui fent.

```
if(colision.CompareTag("Enemy"))
        {
            if(attackType == 0)
            {
                audioSourceAttack.Play();
                colision.transform.GetComponent<Enemy>().TakeDamage(damageAttack);
            }
            else if(attackType == 1)
            {
                audioSourceSkill1.Play();
                colision.transform.GetComponent<Enemy>().TakeDamage(damageSkill_1);
                colision.transform.GetComponent<Enemy>().Frost();
                colision.transform.GetComponent<IAController>().Frosting();
            }
        }
        else if (colision.CompareTag("Boss"))
        {
            if (attackType == 0)
            {
                audioSourceAttack.Play();
                colision.transform.GetComponent<Boss>().TakeDamage(damageAttack);
            }
            else if (attackType == 1)
            {
                audioSourceSkill1.Play();
                colision.transform.GetComponent<Boss>().TakeDamage(damageSkill_1);
                colision.transform.GetComponent<Boss>().Frost();
                colision.transform.GetComponent<BossController>().Frosting();
            }
        }
```

Figura 9.48 Funció OnTriggerEnter2D del script CombatCaC amb sons implementats

CAPÍTOL 10

– 10. IMPLANTACIÓ I RESULTATS -

En aquest capítol es mostrarà captures de pantalles del prototip final d'aquest projecte, juntament amb les restriccions legals.

– 10.1 LEGISLACIÓ I NORMATIVA VIGENT -

No s'ha tingut en consideració la normativa de la Llei Orgànica de Protecció de Dades de Caràcter Personal (LOPD), ja que el sistema no tracta en cap moment cap mena de dades relacionades amb l'usuari.

Des del punt de vista de la seguretat, no es requereix cap forma de control d'accés al programa, ja que es tracta d'una aplicació on els usuaris que hi accedeixen només tenen un rol específic.

Pel que fa a la Llei de Serveis de la Societat de la Informació i Comerç Electrònic (LSSICE), el projecte no constitueix en cap sentit una activitat econòmica.

– 10.2 CAPTURES DE PANTALLA-

A continuació es mostrarà diverses captures de pantalla del videojoc:

La Figura 10.1 correspon a la primera escena del joc, el menú principal. Des d'aquí el jugador pot anar al menú d'opcions, el menú de controls, iniciar la partida o sortir de l'aplicació.

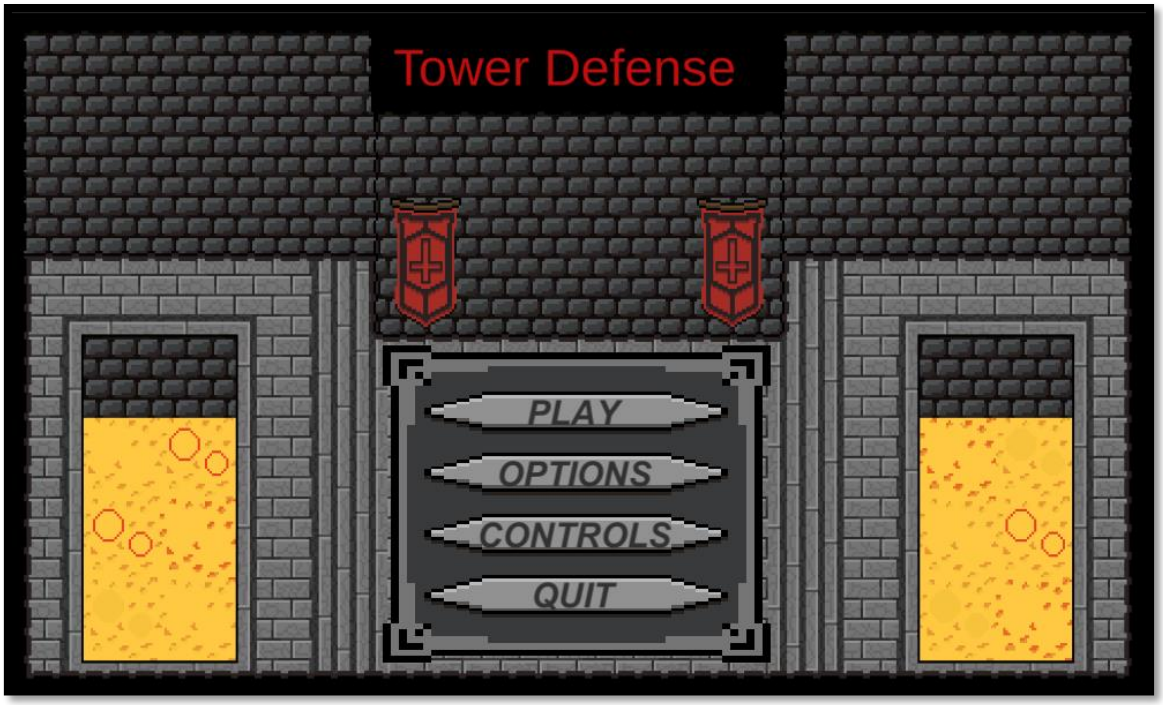

Figura 10.1 Menú principal

A la Figura 10.2 es pot veure el menú d'opcions des del qual es pot regular el volum i sortir/entrar de la pantalla completa.

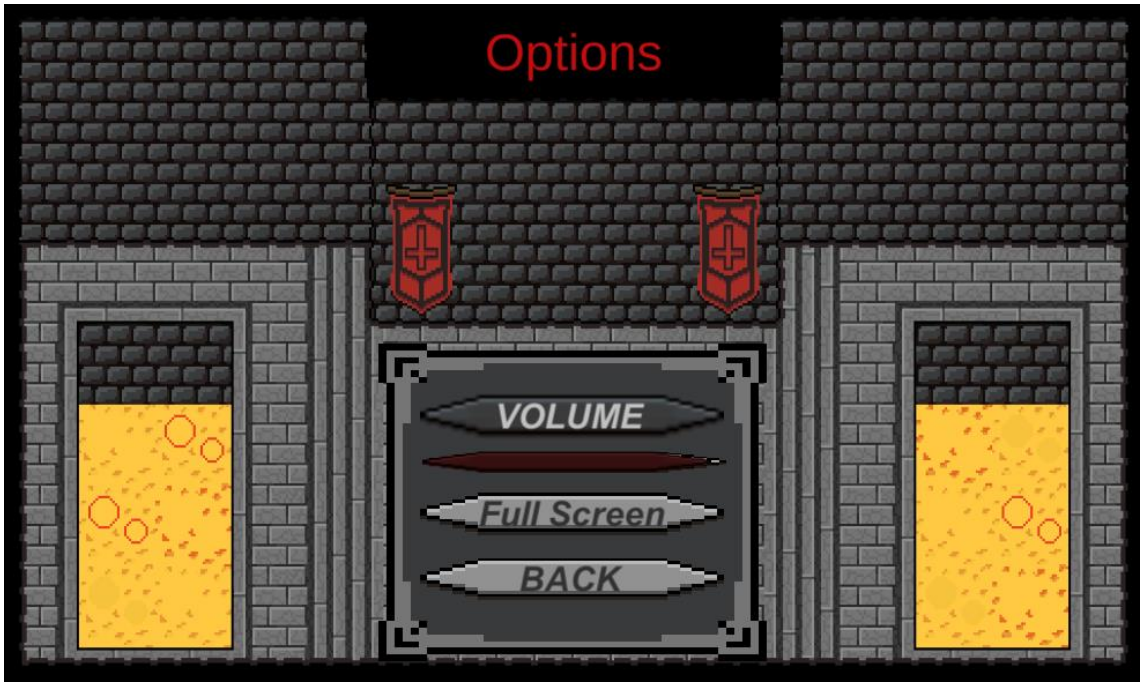

Figura 10.2 Menú d'opcions

La Figura 10.2 és el menú de controls. És únicament visual, per tant, l'única acció que pot fer el jugador és tornar enrere al menú principal. Aquest menú es pot accedir des de qualsevol pantalla.

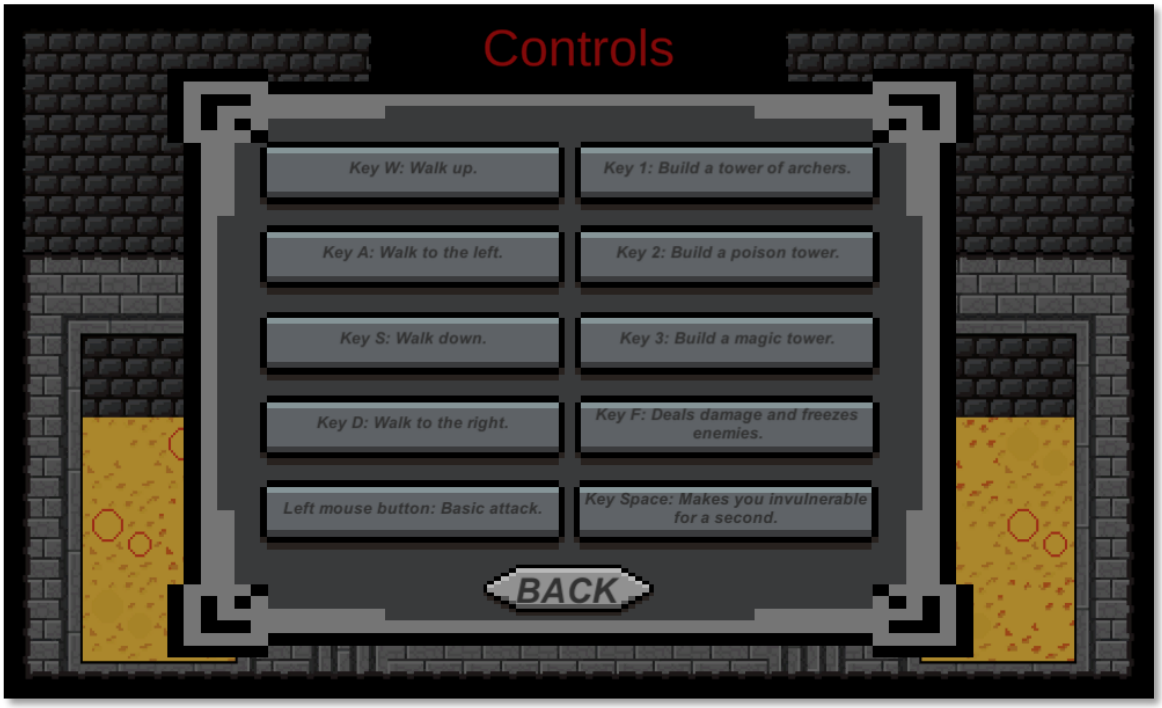

Figura 10.3 Menú de controls

La Figura 10.4 mostra l'inici del tutorial amb el primer missatge, deixant-nos escollir si fer el tutorial, saltar-lo o sortir del joc.

Figura 10.4 Inici del tutorial

La Figura 10.5 mostra el personatge a la segona pantalla. Podem veure que ha de derrotar a 50 enemics, ja que a sota de les barres de vida hi ha un comptador que posa 0/50. A dalt veiem el nombre de monedes que té per construir torres.

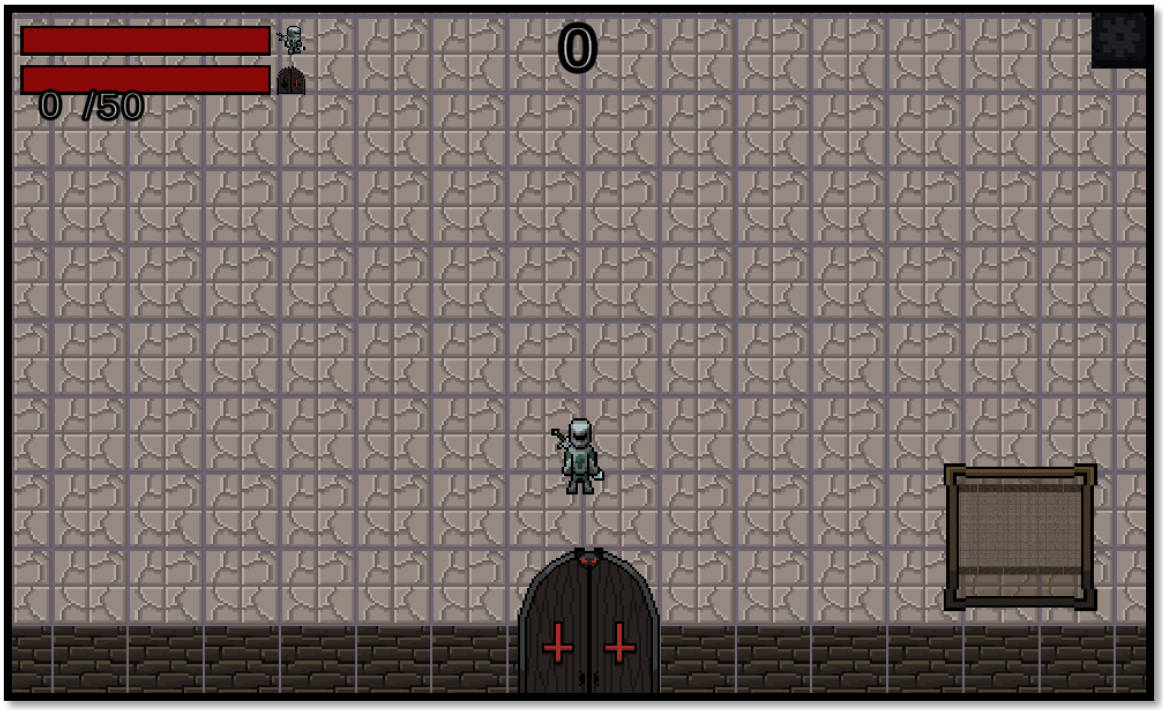

Figura 10.5 Inici pantalla defensar la porta

A la Figura 10.6 es pot observar com el joc ha avançat i el jugador ha construït diferents torres per fer front als enemics.

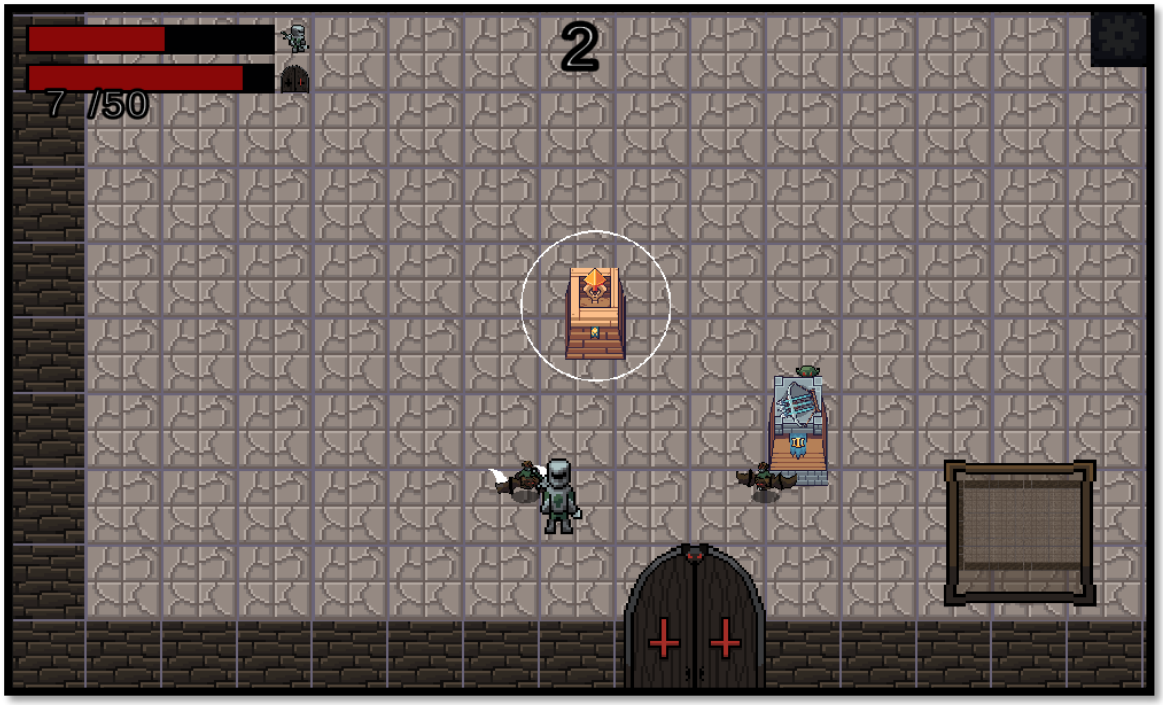

Figura 10.6 Torres ajudant a defensar

A la Figura 10.7 el jugador està atacant amb l'atac bàsic. A la Figura 10.8 amb l'habilitat de congelar en àrea, i es pot veure dos enemics congelats. A la Figura 10.9 esquivant l'atac d'un enemic.

Figura 10.7 Atac bàsic jugador

Figura 10.8 Habilitat congelar jugador

Figura 10.9 Habilitat esquivar jugador

A la Figura 10.10 es pot veure el menú de pausa que el jugador pot obrir en qualsevol moment prement el botó Esc del teclat o el botó de dalt a la dreta de la pantalla.

Figura 10.10 Menú de pausa

La Figura 10.11 mostra el menú de mort amb les opcions de reiniciar pantalla, tornar al menú d'inici o sortir de l'aplicació. La Figura 10.12 mostra el menú de victòria que és igual al de mort, però substituint el primer botó i fent que avancis a la pantalla final contra el jefe final.

Figura 10.11 Menú de mort

Figura 10.12 Menú de victòria

A la Figura 10.13 es mostrà una captura de pantalla que fa referència a l'inici del combat entre el jefe final i el jugador. Es pot observar que ja no hi ha ni comptador de monedes ni d'enemics. Tampoc la porta i la barra de vida, ja que és un combat un contra un. En canvi, s'ha afegit la barra de vida del jefe final.

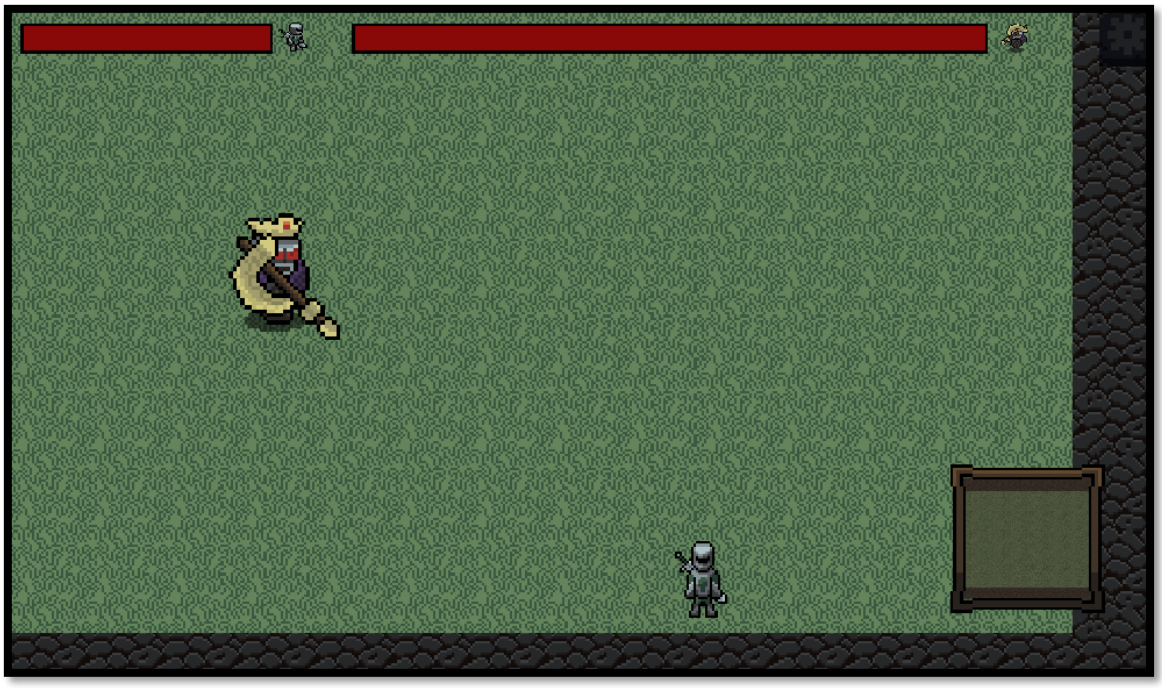

Figura 10.13 Inici de la pantalla final

La Figura 10.14 mostra al jefe final disparant un projectil, ja que estem allunyats d'ell.

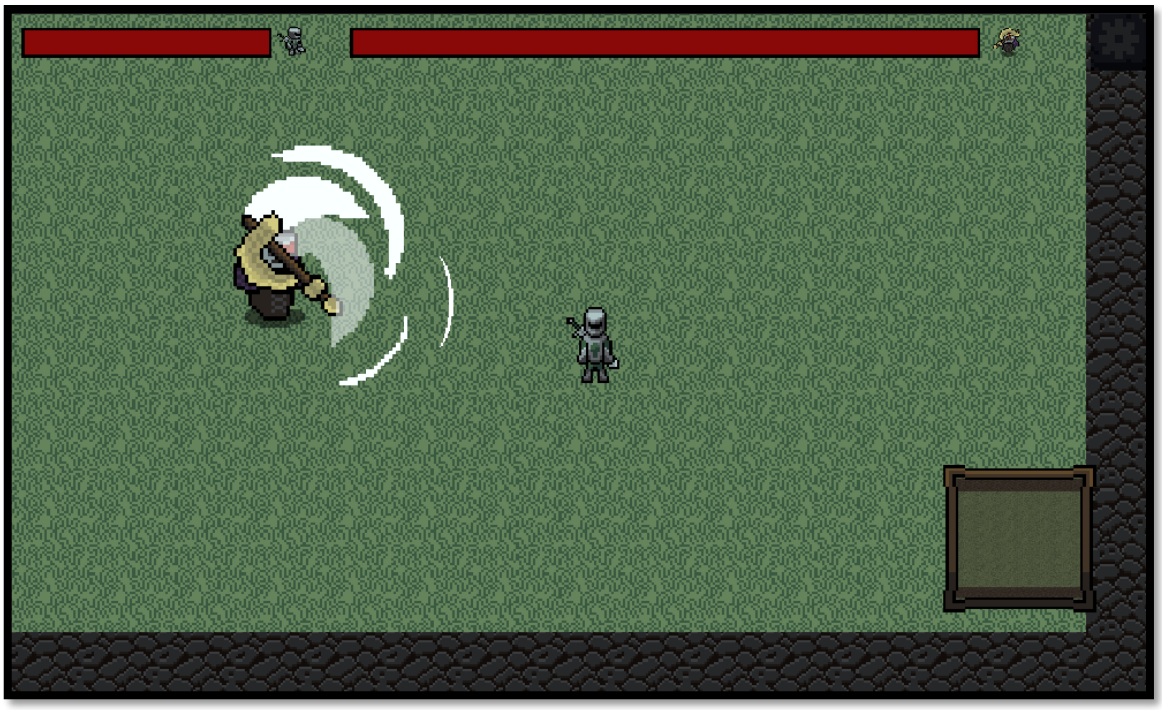

Figura 10.14 Jefe final tirant l'habilitat 1

A la Figura 10.15 es pot observar al jefe final fent l'atac bàsic al jugador.

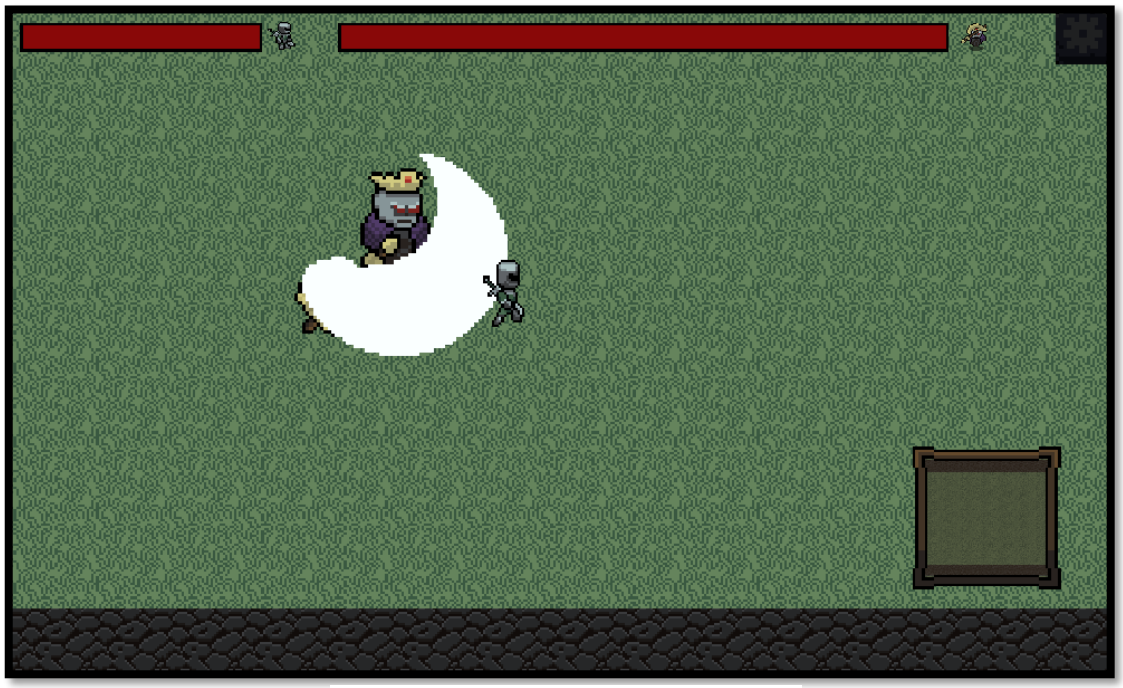

Figura 10.15 Jefe final fent l'atac bàsic

Per acabar amb les habilitats del Boss, a la Figura 10.16 es pot veure com fa servir l'habilitat que genera boles de foc. Aquesta habilitat s'activa quan la vida baixa de la meitat.

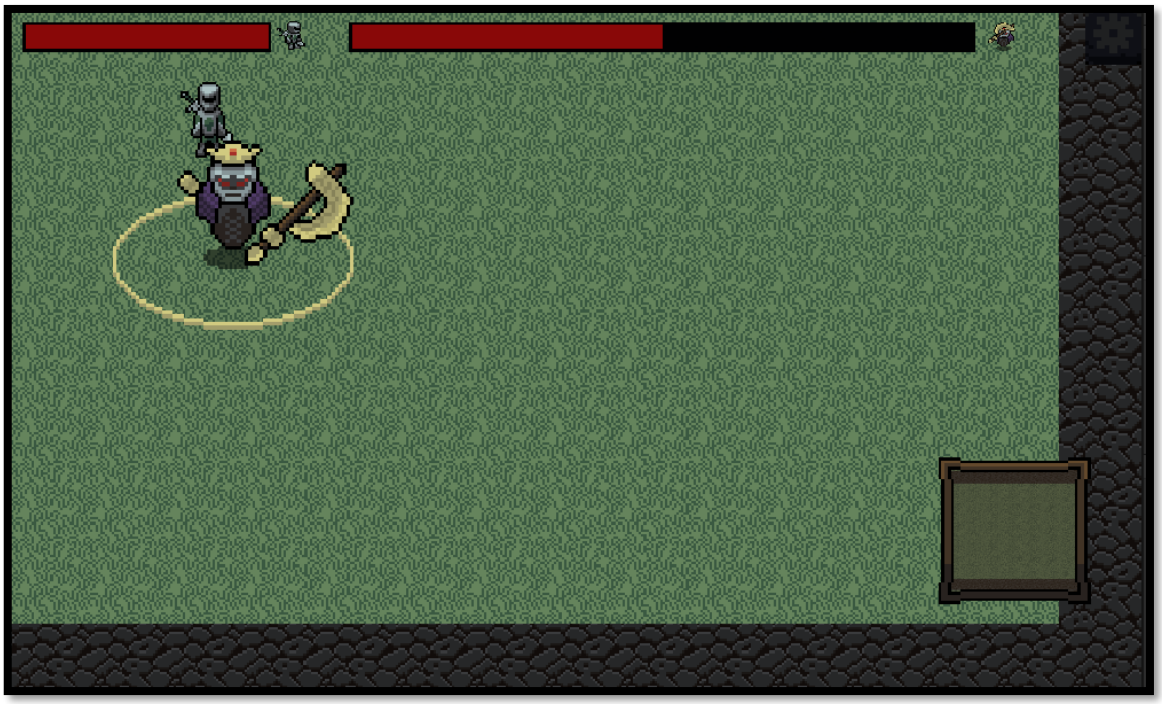

Figura 10.16 Jefe tirant l'habilitat 2

A la Figura 10.17 es pot veure les boles de foc que es generen pel mapa quan el jefe final perd la meitat de la vida.

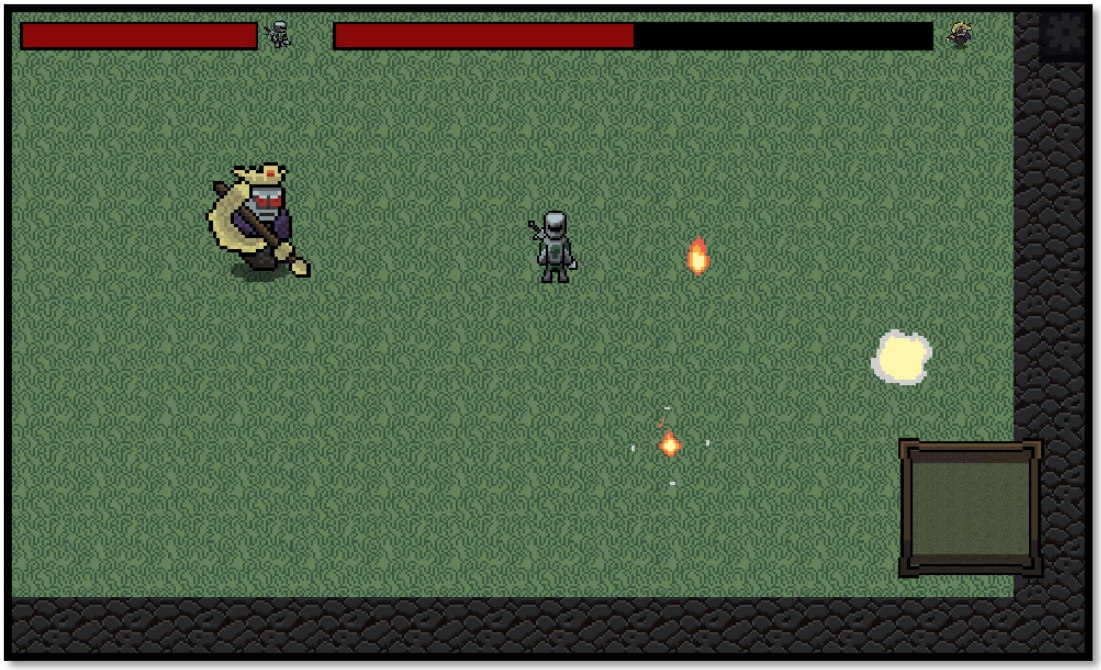

Figura 10.17 Boles de foc

CAPÍTOL 11

– 11. CONCLUSIONS -

Durant el curs d'aquest projecte, s'ha aconseguit importants avenços en el desenvolupament de videojocs utilitzant la plataforma Unity. S'ha adquirit un coneixement profund sobre les funcionalitats de Unity i après a fer servir l'editor per crear escenes, implementar física i col·lisions, gestionar assets i desenvolupar funcionalitats de joc com ara l'IA i els controls de l'usuari.

Tanmateix, des del principi el temps era limitat i tot i tenir una bona planificació, la implementació de noves funcionalitats a vegades incorporen problemes, i aquests problemes modifiquin el temps estimat per funcionalitat.

Una de les principals lliçons que s'han après és la importància d'utilitzar un sistema d'emmagatzematge extern al projecte, per evitar poder tenir alguna pèrdua o la totalitat del projecte. També a fer tot el codi per a què sigui escalable. Si no, a mitjans del desenvolupament s'havia de tornar a implementar algunes funcionalitats per a què encaixin en el projecte, com va passar amb l'IA del jefe final (Apartat 9.7).

Al ser el projecte més gran que fins ara m'he enfrontat sol, s'ha après a dividir el projecte en etapes, realitzar iteracions i millorar gradualment el joc en cada fase. Això ha permès ajustar, corregir i optimitzar el joc per oferir una experiència de joc més àmplia i enriquidora.

La resolució de problemes ha estat una part integral del desenvolupament del videojoc. S'ha abordat reptes tècnics i de disseny de manera sistemàtica, identificant les causes dels problemes i proposant solucions efectives. Això ha potenciat les habilitats de resolució de problemes, fomentant la perseverança i la creativitat davant dels obstacles que sorgeixen durant el desenvolupament del videojoc.

En resum, aquest projecte de desenvolupament d'un videojoc amb Unity ha estat una experiència enriquidora en la qual s'han adquirit coneixements tècnics, resolt problemes i creat una experiència de joc diferent de les clàssiques. Tot i això, no s'han pogut desenvolupar moltes de les idees que es tenien a l'inici a causa de la falta de temps i falta de coneixement al principi del projecte.

CAPÍTOL 12

– 12. TREBALL FUTUR -

Aquest projecte és més ambiciós del que s'ha pogut mostrar en aquest prototip. El punt principal d'aquest projecte és donar un nou enfocament al gènere Tower Defense i proporcionar un producte diferent. Ara es comentarà la feina que faltaria per fer o millorar.

Per començar, com ja s'ha comentat en aquesta memòria, s'hauria de millorar el codi que gestiona les intel·ligències artificials, unint el IAController i el BossController sota una classe pare que es comuniqui amb les accions i funcions de l'arbre de comportament.

Una idea que es volia desenvolupar en aquest projecte, era augmentar el factor de roguelike. Quan el jugador es passés la pantalla dos, sortiria un menú que deixa triar al jugador entre continuar defensant o contraatacar. En cas de continuar defensant, hi hauria una altra pantalla amb un jefe final, juntament amb enemics. Si es volgués contraatacar, llavors el jugador, sense ajuda de les torres, hauria d'arribar cap a la base enemiga abans que s'acabés el temps, esquivant obstacles i derrotant enemics pel camí. Si aconseguís arribar a temps s'enfrontaria ell sol contra un jefe (pantalla final del joc).

Òbviament per poder desenvolupar aquesta idea i que augmentés el factor roguelike, s'haurien de crear nous enemics i nous jefes. Així cada cop que es comencés una nova partida, al decidir defensar o contraatacar, el jugador no sabrà contra quin jefe lluitarà.

Per fer el joc més atractiu, també s'haurien d'implementar nous personatges per jugar. S'havia pensat amb una maga i un arquer. Així com noves torres. Aquest nou contingut es desbloquejaria a l'eliminar els diferents jefes del joc.

Cal remarcar que aquest videojoc s'ha fet amb gràfics i música extrets per internet. Totes tenen permís per a poder ser comercialitzat sempre que se'ls mencionin. Però, si el joc es volgués comercialitzar, s'hauria de crear nous assets, per donar estil propi al videojoc.

Finalment, fer algunes animacions i crear una història per donar vida al videojoc.

– 13. BIBLIOGRAFIA -

Stack Overflow - Where Developers Learn, Share, & Build Careers. (s. d.). Stack Overflow. *https://stackoverflow.com/*

Pêcheux, M. (2022, 4 enero). How to create a simple behaviour tree in Unity/C# - Geek Culture - Medium. *Medium*. https://medium.com/geekculture/how-to-create-a-simple-behaviour-tree-in-unity-c-3964c84c060e

Technologies, U. (s. d.). *Solutions*. Unity. https://unity.com/solutions

Technologies, U. (s. d.). *Unity - Scripting API*: https://docs.unity3d.com/ScriptReference/

¿Cuánto Cobra un Programador de Videojuegos? (Sueldo 2023) | Jobted.es. (s. f.). https://www.jobted.es/salario/programador-videojuegos

Tokio School. (2023, 10 mayo). *Descubre cuánto cobra un diseñador de videojuegos: ¿cuál es su salario?* https://www.tokioschool.com/noticias/sueldo-disenador-videojuegos-cuanto-cobra/

Sueldo: Compositor en España en 2023. (s. f.). Glassdoor. https://www.glassdoor.es/Sueldos/compositor-sueldo-SRCH_KO0,10.htm

Sueldo: 2D Artist (Mayo, 2023). (s. f.). Glassdoor. https://www.glassdoor.es/Sueldos/2d-artist-sueldo-SRCH_KO0,9.htm

Richter, L. (s. d.). Capítulo 3: Metodologías de gestión de proyectos. *Smartsheet*. https://es.smartsheet.com/project-management-guide/methodologies

FoozleCC. (s. d.). itch.io. https://foozlecc.itch.io/

CAPÍTOL 14

– 14. ANNEXOS -

L'annex d'aquesta memòria és el mateix projecte sencer d'Unity que està adjunt amb la documentació. En el projecte s'hi poden trobar tots els Assets que s'han buscat i utilitzat durant el desenvolupament del projecte. Inclou sprites, animacions, prefabs, etc.

Els directoris del projecte han estat organitzats perquè es puguin identificar fàcilment els diferents recursos utilitzats:

- Els prefabs es troben a la carpeta: Assets/Prefabs
- Els fitxers de codi es localitzen a la carpeta: Assets/Scripts
- Les músiques i sons estan a la carpeta: Assets/Sounds
- Les animacions es troben a la carpeta: Assets/Animations
- Els assets utilitzats pels enemics, personatges i mapa es localitzen a la carpeta: Assets/Lucifer
- Els diferents Tilemaps generats estan a la carpeta: Assets/Tilemap
- Les escenes del joc es troben a la carpeta: Assets/Scenes
- Els elements d'interfície gràfica (barra de vida, botons, etc.) estan a la carpeta: Assets/ALL_UI

CAPÍTOL 15

– 15. MANUAL D'USUARI -

Per executar el videojoc cal obrir la carpeta del projecte anomenada "TowerDefense" i executar el fitxer "TowerDefenseRPG", que té la icona d'un ull. Seguidament, el joc s'iniciarà mostrant el logotip d'Unity i posteriorment el menú principal.

Controls

Amb el ratolí s'ha de clicar una de les opcions del menú, si es clica el botó "PLAY", es començarà la partida.

Per jugar s'utilitza el teclat i el ratolí.

-Moviment:

- Tecla W: desplaçament cap a dalt.
- Tecla A: desplaçament cap a l'esquerra.
- Tecla S: desplaçament cap avall.
- Tecla D: desplaçament cap a la dreta.

-Atacs:

- Botó esquerre del ratolí: atac bàsic.
- Tecla F: congelar i fer mal en àrea.
- Tecla espai: esquivar.

-Construir torres:

- Tecla 1: construir torre d'arquers.
- Tecla 2: construir torre màgica.
- Tecla 3: construir torre de verí.

Objectiu

L’objectiu del videojoc és defensar la porta de l’onada d’enemics, derrotant-los amb els atacs i les torres. Finalment matar al jefe final.

Menú de configuracions

Es pot accedir al menú de configuracions des del menú inicial. Permet configurar el volum i sortir/entrar de pantalla completa