Universitat de Girona
Escola Politècnica Superior

Grau en Enginyeria Informàtica

PROJECTE FINAL DE GRAU

Desenvolupament amb Unity d'un videojoc tipus Metroidvania amb elements Roguelike

Autor:
Santiago Jimenez Garcia

Tutors:
Gustavo Patow
Gonzalo Besuievsky

MEMÒRIA

Convocatòria:
Setembre 2022

Departament:
Informàtica, Matemàtica Aplicada i Estadístic

Índex

1. Introducció

Des de dels anys 70, quan l'empresa *Atari* va comercialitzar el videojoc *Computer Space*, l'industria dels videojocs no ha fet més que créixer. Avui en dia es pot considerar una de les industries més grans i importants del sector de l'entreteniment. Això, en part, es a causa de la gran varietat de tipus o gèneres que els videojocs poden assolir. Des d'aventures explorant un nou mon amb màgia i monstres a competicions d'estratègia entre dos jugadors o simuladors de carreres de cotxes, existeixen videojocs per a tots els gustos.

El confinament va beneficiar al mercat dels videojocs de manera considerable. L'accessibilitat i capacitat de jugar des de casa i connectar-se amb altres persones els van popularitzar els videojocs encara més durant aquesta època. Observar la Figura 1.

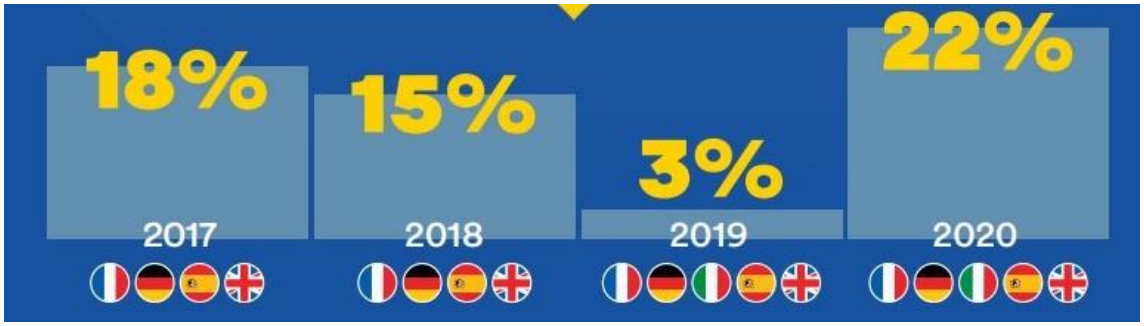


Figura 1. Increment de beneficis del mercat de videojocs d'Europa

Els grans líders del mercat de videojocs són les companyies anomenades de manera informal Triple A, les quals són les productores i distribuïdores de les franquícies de videojocs més famoses i amb desenvolupament de alt cost. Tot i ser les empreses amb més beneficis dins l'industria, als últims anys s'han popularitzat els jocs de companyies o desenvolupadors independents, també anomenats *indie*.

1.1 Gèneres

Existeixen una gran quantitat de gèneres dins del mon dels videojocs, però per aquest projecte ens fixarem específicament en dos: *Metroidvania* i *Roguelike*.

- **Metroidvania**

El gènere Metroidvania incorpora plataformes i combat i és caracteritza per progressió no lineal, grans mapes, la seva exploració i limitacions dissenyades pel desenvolupador, que poden ser superades per l'obtenció de noves habilitats dins del joc. El nom és la combinació dels noms de les famoses franquícies *Metroid* i *Castlevania* (Figura 2), les quals van popularitzar aquests conceptes.

Figura 2: "CASTLEVANIA" (NES, 1986)

- **Roguelike**

Els videojocs Roguelike utilitzen la generació procedural per remodelar el mapa cada vegada que el jugador mor i torna al principi. D'aquesta manera crea una experiència diferent cada vegada que es comença una altra partida i incrementa la rejugabilitat del joc. La diferència entre Roguelike i *Roguelite* és que al segon el jugador per tot el que havia aconseguit al morir, mentre que al primer conserva alguns recursos per la següent "run". Un bon exemple d'aquest gènere seria el videojoc *The Binding of Isaac* (Figura 3).

Figura 3: "The Binding of Isaac: Rebirth"

1.2 Motivacions

Tot i que els dos gèneres mencionats anteriorment no són els més jugats, tenen un lloc especial dins la comunitat de videojocs, així que sempre haurà demanda de jocs d'aquest dos tipus . Dit això, no es pot trobar cap videojoc que combini els dos gèneres de manera satisfactòria.

En aquest projecte es vol aprofitar del forat en el mercat i crear un videojoc que combini el millor dels dos mons.

1.3 Propòsit

El propòsit d'aquest projecte és utilitzar la rejugabilitat dels jocs Roguelike junt amb la sensació d'exploració i progrés dels jocs Metroidvania per crear un videojoc 2D que proporcioni una experiència única i revolucionaria.

1.4 Objectius

Per complir el propòsit, caldrà assolir els següents objectius:

1. Planificar els elements principals del joc.
2. Planificar els poders a aconseguir i les mecàniques corresponents.
3. Implementació de les accions possibles pel jugador i la interacció amb l'escenari.
4. Estudiar i implementar la generació procedural del mapa amb obstacles per limitar l'accés a certes zones temporalment.
5. Estudiar i implementar IAs pels enemics.
6. Crear un sistema de menús.
7. Verificar i provar del codi implementat.

1.5 Estructura del document

Aquesta memòria de Projecte de Fi de Grau esta estructurat en 14 apartats diferents.

1. Introducció, motivacions, propòsit i objectius
   S'expliquen els motius per la realització del projecte, els propòsits i els objectius a assolir per completar-lo, a demés d'un petit resum del joc. S'afegeix també la organització del document de la memòria.

2. Estudi de la viabilitat
   Es justifiquen els paràmetres necessaris per al desenvolupament del projecte.

3. Metodologia
   S'explica la metodologia utilitzada al llarg del projecte i els motius per utilitzar-la.

4. Planificació
   Es parla sobre l'estratègia seguida per desenvolupar i assolir els objectius marcats del projecte plantejat, junt amb la temporització seguida al llarg d'aquest.

5. Marc de treball
   En aquest apartat es troba una descripció general dels diferents aspectes del desenvolupament del projecte, amb el fi de facilitar la lectura dels següents capítols. S'expliquen tant les accions més significatives preses durant les etapes inicials del disseny del projecte, com els passos d'estudi i aprenentatge dels conceptes utilitzats durant el desenvolupament.

6. Requisits del sistema
   S'especifiquen els requisits funcionals i no funcionals del programari, els quals recullen els objectius de l'aplicació i les funcionalitats a obtenir. Aquest apartat ajuda a entendre el que envolta el sistema informàtic que es vol construir.

7. Estudi i decisions
   S'explica de les eines implementades per dur a terme el projecte. També s'expliquen els motius que han portat a la utilització d'aquestes eines i amb quina finalitat han estat utilitzades.

8. Anàlisi i disseny del sistema

Es proporciona la investigació realitzada per resoldre els problemes trobats durant el desenvolupament del projecte, com poden ser les especificacions i requeriments. També s'aprofundeix en els detalls més importants del projecte junt amb les classes i mètodes.

9. Implementació i proves

Conté el procés de construcció de l'aplicació, amb les classes i mètodes implementats més importats a l'hora de comprendre l'aplicació. A diferència del capítol anterior, en aquest apartat es tracta el software realitzat amb detall.

10. Resultats

Es mostren proves d'execució de l'aplicació junt a imatges del joc en marxa, les interfícies inicials, imatges de la partida i altres aspectes a destacar del projecte.

11. Conclusions

S'exposen les conclusions del projecte i una crítica dels resultats obtinguts.

12. Treball futur

Es descriuen totes les possibles ampliacions i millores que es podrien realitzar al sistema per tal de millorar-ho.

13. Bibliografia

Conté les diferents referències utilitzades per a desenvolupar el projecte.

14. Manual d'usuari

Conté un manual d'usuari per la utilització del videojoc.

2. Estudi de viabilitat

Els recursos necessaris per a desenvolupar aquest projecte són mínims. El cost d'infraestructura és pràcticament inexistent.

El material tecnològic utilitzat ha estat un portàtil de la marca DELL Inspiron 16.

Pel que fa a software, el material utilitzat ha sigut:

- Windows 11
- Motor de videojocs multi plataforma Unity (Figura 4)
- Editor Visual Studio Code

Com ja es contava amb aquests materials prèviament a l'inici del projecte, no ha suposat cap cost addicional. La versió de Unity utilitzada era gratuïta.

Figura 4: Unity logo

2.1 Recursos humans

Tot videojoc esta format de 3 elements: gràfics, so i programació. Aquests components són fonamentals al crear un videojoc decent. Seria necessari disposar d'un equip format per un analista, programador, dissenyador gràfic, compositor i cap de projecte per treballar al 100%. Idealment, es disposarien de més membres dedicats a les tasques artístiques a causa de la gran quantitat de temps necessària per complir aquests apartats del videojoc de manera satisfactòria. En el cas d'aquest projecte, tots els rols seran realitzats per una sola persona.

2.2 Viabilitat legal

Tots els recursos externs que s'han inclòs al projecte han estat obtinguts de fonts d'us lliure i, sempre que així s'especifiqui, es dona crèdit a l'autor corresponent.

Tot l'apartat gràfic emparat s'ha descarregat de fonts oficials.

2.3 Inversió inicial

Aquest projecte no requereix cap inversió inicial. Dit això, si es volgués publicar aquest videojoc a la plataforma *Steam* per guanyar un benefici, caldria una inversió de 91,40 euros, a demés que Steam es queda un 30% dels beneficis que aquest joc genera.

A més, si s'excedissin els 100000$ anuals al comercialitzar el joc, seria necessari obtenir la llicència professional de Unity, que costaria uns 1868 euros anuals.

2.4 Avaluació de costos mitjans

Abans de començar el desenvolupament del projecte, cal estudiar la viabilitat des de diferents punts de vista: econòmic i tècnic

2.4.1 Estudi de la viabilitat tecnològica

En el cas d'aquest projecte, ja es disposava del hardware necessari pel desenvolupament. Donat això i el fet que el hardware utilitzat es troba en la gama mitjana d'ordinadors actuals, no ha estat necessari realitzar un estudi extensiu.

Portàtil DELL Inspiron 16:	876,80 €
Desgast al projecte:	97,42 €
Percentatge dedicat al projecte:	(Preu Portàtil / Desgast) * 100 = 11 %

2.4.2 Estudi de la viabilitat econòmica

Aquest estudi es pot dividir en dues parts.

- Costos de recursos humans:

  Si partim del cas que disposem d'un equip complet per desenvolupar el projecte com s'ha definit al apartat 2.1, necessitaríem primer definir els salaris de cada empleat segons la funció.

Professional	Sou (euros/hora)
Cap de projecte	20
Analista	18
Programador	15
Dissenyador gràfic	15
Compositor	22

Com aquest és un cas hipotètic, cal teoritzar el nombre d'hores que cada membre invertiria en la creació del videojoc.

Tasca	Membre	Hores
Investigació	Analista	40
Disseny del concepte del joc	Cap de projecte	12
Implementació de l'IA	Programador	30
Implementació de l'algorisme de generació procedural	Programador	65
Implementació dels sistemes de control	Programador	25
Creació de models i animacions	Dissenyador gràfic	110
Creació del disseny dels mapes	Dissenyador gràfic	40
Creació de la banda sonora	Compositor	25
Creació dels efectes de so	Compositor	15
Memòria	Cap de projecte	80
Total		502

El resultat final dels costos de recursos humans són:

- Cap de Projecte: 1.840 €
- Analista: 720 €
- Programador: 1.800 €
- Dissenyador gràfic: 2.250 €
- Compositor: 880 €
- Total: 7.490 €

- Costos de maquinària

En el cas dels costos de maquinària, tot el software utilitzat durant el procés era gratuït, però en el cas hipotètic que es volgués comercialitzar el videojoc resultant, seria necessària la llicència pro de Unity, a demés de les llicències per als diferents software necessaris com Adobe Photoshop o Adobe Audition.

Per calcular el cost total de les subscripcions mensuals, s'assumeix que s'ha trigat 3 mesos a completar el videojoc.

Software	Cost mensual	Cost Total
Llicència digital Windows 10		11,90 €
Unity Pro	137,11 €	411,33 €
Adobe Photoshop	24,19 €	72,57 €
Adobe Audition	24,19 €	72, 57 €
Total		568,37 €

Partint de les estimacions anteriors, podem aproximar el preu total del projecte a 8.058,37 €.

3. Metodologia

Tot i que existeixen una gran quantitat de metodologies diferents, cap d'elles s'adaptava completament a les necessitats del projecte. Per tant, s'ha escollit una metodologia personalitzada, proposada pels tutors del treball, la qual segueix els següents passos:

1. Escollir el projecte a realitzar.

2. Escollir les eines que es faran servir i aprendre a utilitzar-les.

3. Dividir i estructurar el treball en diferents mòduls i tasques per facilitar la seva repartició i desenvolupament.

4. Escollir una de les tasques del punt anterior encara no realitzada i desenvolupar-la.

5. Comprovar si la part desenvolupada funciona correctament.

    - **No funciona:** Es revisen les errades de la tasca i es torna al punt 4 per arreglar-les.
    - **Funciona:** En cas que la part realitzada fos l'última, es passa al punt 6, altrament es torna al punt 4 i s'escull una nova part a realitzar.

6. S'uneixen tots els mòduls del projecte i es comprova el seu funcionament.

    - **No funciona:** S'analitza el treball per determinar quines parts donen problemes i tornar al punt 4 amb aquestes per arreglar-les.
    - **Funciona:** Es passa el punt 7.

7. Generar diferents models d'exemple a partir de la unió del punt 6 i es comprova el bon funcionament.

    - **Comprovacions no són les esperades:** S'analitza el treball per determinar quines parts donen problemes i tornar al punt 4 amb aquestes per arreglar-les.
    - **Comprovacions són les esperades:** Es passa al punt final.

8. Redactar la documentació del projecte.

Veure figura 5.

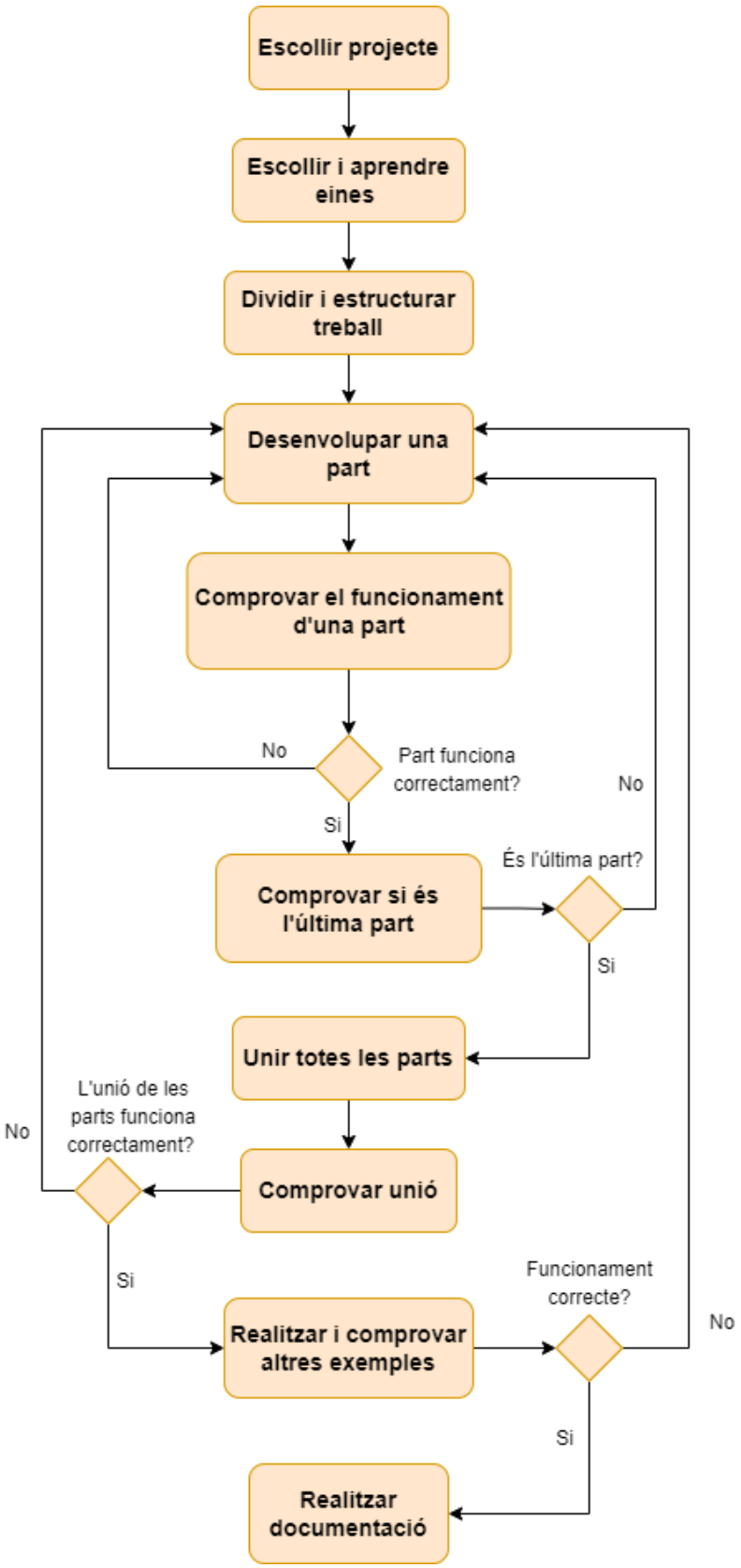


Figura 5: Diagrama de flux de la metodologia

4. Planificació

En qualsevol projecte és important definir la planificació per poder assolir tots els objectius dins del termini atorgat. Ens permet valorar el temps necessari de cada part del treball a realitzar.

Tot i no ser una data exacta, es pot considerar l'1 de febrer com l'inici del projecte. S'havien realitzat reunions i xerrades amb els tutors anteriorment, però el treball pràctic no es va començar fins a vores d'aquesta data. Es considerarà el dia de lliurament del TFG, el 2 de Juny, com a data final del projecte.

4.1 Pla de treball

El següent pla de treball ha consistit a separar el treball a realitzar del projecte en tasques generals i dividir aquestes en tasques més específiques.

4.2 Planificació de tasques

4.2.1 Plantejament del joc

- **Planificació del joc**

  En aquesta etapa es va definir el tipus de joc a desenvolupar. La idea general, el gènere, si seria 2D o 3D i les regles principals.

  Inicialment es va proposar un joc del estil Metroidvania per preferències personals però, com es va determinar que no seria una bona mesura de les habilitats i els coneixements obtinguts durant la carrera d'Informàtica, es va modificar per implementar conceptes de jocs del estil Roguelike, el més important sent la generació procedural del mapa.

- **Estudi del motor gràfic Unity2D i el llenguatge C#**

Aquesta etapa es va dedicar a l'estudi i la comprensió de les eines disponibles per realitzar el projecte. Es va escollir Unity2D i C# a causa de la seva accessibilitat i familiaritat, a demés de la gran quantitat de recursos disponibles per facilitar l'aprenentatge.

- **Estudi de l'API de Unity2D**

En aquesta etapa es va estudiar l'API de Unity per tal de entendre el funcionament de la programació de videojocs i habituar-se a l'entorn.

4.2.2 Disseny

- **Disseny del sistema de moviment del personatge**

En aquesta secció es va definir com es mouria el personatge que el jugador controlarà dins del joc i els algoritmes necessaris per implementar-ho, a demés de valors com la velocitat i l'alçada del salt.

- **Disseny de les interaccions entre els elements**

En aquesta etapa es van definir les interaccions entre els diferents elements del videojoc. Aquestes interaccions es poden dividir en: interacció entre jugador i escenari, interacció entre enemic i escenari, interacció entre jugador i enemic. Es va decidir que els enemics no operarien individualment i no interactuarien amb altres enemics, amb l'excepció del "Boss".

- **Disseny del sistema de combat**

En aquest apartat es va definir i dissenyar el sistema de combat del personatge, el qual inclou el sistema que controla els punts de salut del jugador i dels enemics.

- **Disseny de la IA**

En aquesta tasca es varen definir les Intel·ligències Artificials dels diferents enemics i les particularitats de cada una. Es va arribar a la conclusió que s'utilitzarien Behavior Trees per programar les diferents IA. La demo del videojoc conte dues IA per enemics comuns i una IA més complexa per al "Boss".

- **Disseny de l'algorisme procedural de generació del mapa**

En aquesta secció es va definir i dissenyar l'algorisme que s'encarregaria de generar de forma procedural el mapa de cada nivell. Aquest algorisme havia de complir amb el directiu de que sempre existís un camí des del començament del nivell fins a l'última sala. També tenia la possibilitat de crear zones amb obstacles que només serien possible superar una vegada obtinguda l'habilitat d'aquella zona. En aquest pas també es van dissenyar la aparició d'enemics dins del mapa.

- **Disseny de la interfície d'usuari**

En aquesta tasca es van definir i dissenyar tots els elements necessaris per a la interfície d'usuari a demés de la seva posició a la pantalla. També es va dissenyar les interaccions de la interfície amb l'usuari.

4.2.3 Implementació

- **Implementació del sistema de moviment del personatge**

En aquesta tasca es van implementar els algorismes que controlen el moviment del personatge. El jugador es desplaça mitjançant l'aplicació de vectors de força, enlloc de variar directament la posició. D'aquesta manera, el control del personatge es mou amb més naturalitat.

- **Implementació de la interacció entre els elements**

En aquesta etapa es van implementar les interaccions entre els diferents elements del joc. Com s'ha comentat a l'apartat de disseny, els enemics no interactuen entre ells, excepte el "Boss", que pot invocar a un tipus d'enemic per a que l'ajudi a la lluita contra el jugador.

- **Implementació del sistema de combat**

En questa secció es va implementar el sistema de combat del personatge junt amb els algorismes que controlen els punts de salut del jugador i els enemics i com interactuen aquest dos sistemes entre ells.

- **Implementació de la IA**

En aquest apartat es van implementar les diferents Intel·ligències Artificials dels enemics utilitzant el model de Behavior Tree. Encara que Unity suporta llibreries d'aquest model, després d'investigar-les es va arribar a la conclusió que no oferien suficient llibertat. Per tant, les classes i funcions necessàries per al videojoc es van implementar manualment.

- **Implementació de l'algorisme procedural de generació del mapa**

En aquesta etapa es va implementar l'algorisme procedural de generació de mapa i enemics, per tal que complissin amb les condicions proporcionades a la secció de disseny.

- **Implementació de la interfície d'usuari**

En aquesta secció es van implementar els diferents elements de la interfície d'usuari junt amb les interaccions amb el jugador.

4.2.4 Integració i testeig

- **Cerca de modelats i animacions**

Una vegada fetes les proves de funcionament, es va cercar els modelats i animacions que s'utilitzarien per al jugador, els enemics i l'escenari. Es va decidir utilitzar pixelart com a estil gràfic del videojoc.

- **Cerca de banda sonora i efectes de so**

En aquesta etapa es van buscar els efectes de so a fer servir per als diferents elements del videojoc i la banda sonora.

- **Integració dels algorismes amb els modelats**

En aquesta secció es va unir els diferents algorismes i accions del jugador i els enemics amb els modelats o animacions corresponents

- **Verificació del funcionament final del joc**

En aquest apartat es van realitzar proves exhaustives per comprovar el correcte funcionament de la unió de tots els components de joc. També es comprova que el resultat final assoleix els reptes plantejats a l'apartat de plantejament del joc i disseny.

- **Creació de la demo del videojoc**

Un cop testejat el videojoc, es va crear un demo que compta amb tots els elements realitzats.

- **Documentació**

Com aquesta és una tasca constant, la duració és la duració del projecte.

4.3 Temps estimat i definitiu

L'estudi de temps esperat es veu reflectit en la figura 6, mentre que el resultat un cop acabat el projecte es veu reflectit en la figura 7.

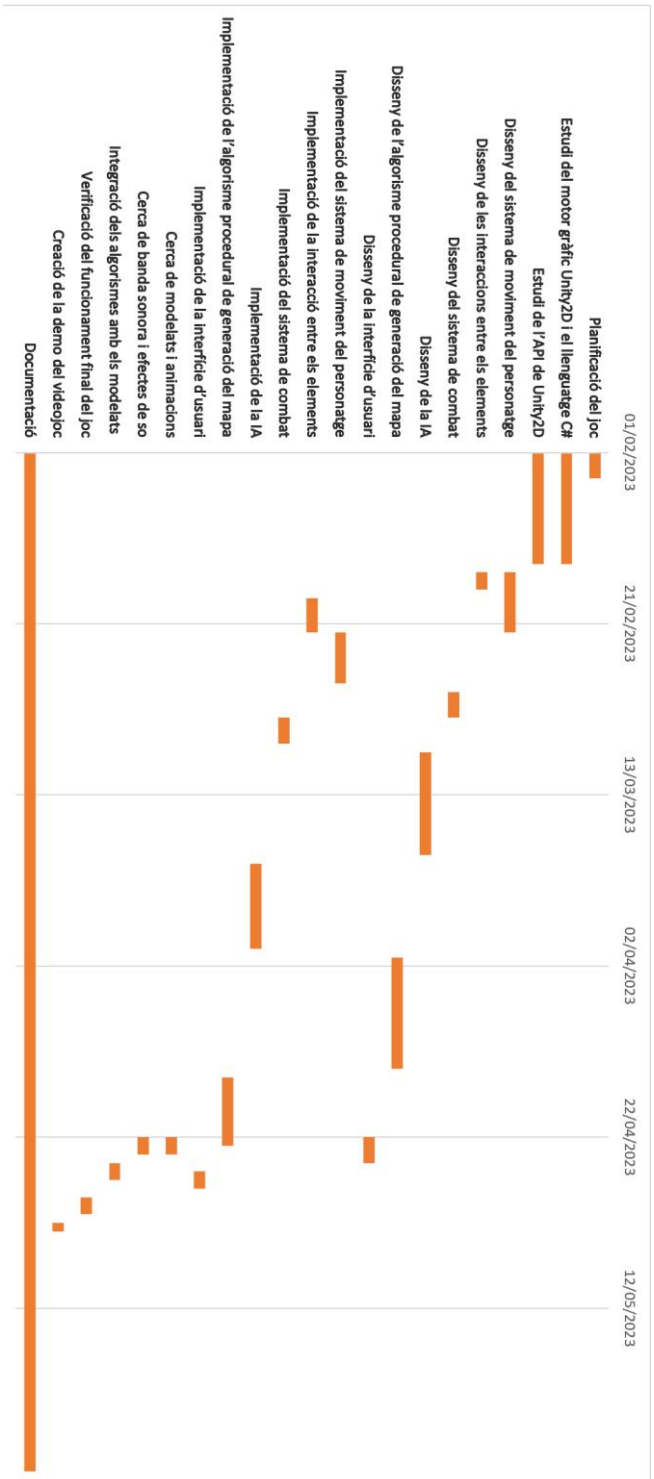


Figura 6: Diagrama Gantt del temps real

Figura 7: Diagrama Gantt del temps esperat

En el cas de l'aprenentatge, es va trigar menys temps al planificat a causa de la gran quantitat de recursos i documentació de Unity a la meva disposició, els quals faciliten aquesta tasca. El disseny de la IA del monstre "Boss" va ser més difícil del previst i, per tant, va causar que es trigués més a completar la tasca "Disseny de la IA". El mateix va passar a l'hora d'implementar l'algorisme de generació procedural del mapa, ja que sorgien problemes inesperats quan s'executava. Aquests costos de temps addicionals van causar que tasques com "Implementació de la interfície d'usuari" s'acceleressin per poder complir els objectius. Per sort, cercar modelats, animacions i efectes de so i integrar-los al joc va ser una tasca més ràpida de l'esperat.

4.3 Resultat esperat

El resultat esperat és un videojoc completament funcional amb possibilitat de créixer i millorar i que, en un futur, es pugui llançar al mercat i ser capaç de competir a causa de la combinació innovadora de dos gèneres estimats per la comunitat de videojocs.

També s'espera haver creat un joc entretingut que compleixi amb les expectatives presentades al principi del projecte.

5. Marc de treball i conceptes previs

En aquest apartat s'aprofundeix en la feina realitzada durant l'etapa de preparació i estudi del projecte. Es parla dels aspectes generals, eines i decisions bàsiques que s'han pres durant el desenvolupament.

5.1 Introducció als motors de videojocs

En la gran majoria de casos, quan es desenvolupa un videojoc comercial s'utilitza un motor de videojoc com a eina principal. Al parlar d'un motor de videojoc es refereix al programari que permet la creació, disseny i representació d'un videojoc, i que avarca un gran nombre de funcionalitats. Entre elles, la renderització, simulació física, col·lisió, sistema operatiu i l'animació.

La funció principal del motor és facilitar el procés de desenvolupament d'un videojoc. Amb aquesta eina els desenvolupadors poden deixar de preocupar-se de tocar aspectes de baix nivell, com per exemple les llibreries gràfiques del sistema operatiu.

Els motors de videojocs d'avui en dia són més complexos que mai. Estan formats per motors més petits que s'encarreguen de diferents aspectes del joc, com el motor gràfic o el motor físic. Veure la Figura 8.

Existeixen una quantitat considerable de motors de videojoc disponibles al públic, cascun amb les seves fortaleses i debilitats. Per això, és molt important que abans de començar el desenvolupament d'un videojoc, s'esculli el motor de videojoc que millor s'adapti al que es necessiti.

Els criteris de selecció més importants per aquest projecte són:

- Corba de dificultat
- Comoditat d'ús
- Documentació del motor
- Comunitat
- Motor gràfic
- Motor de física

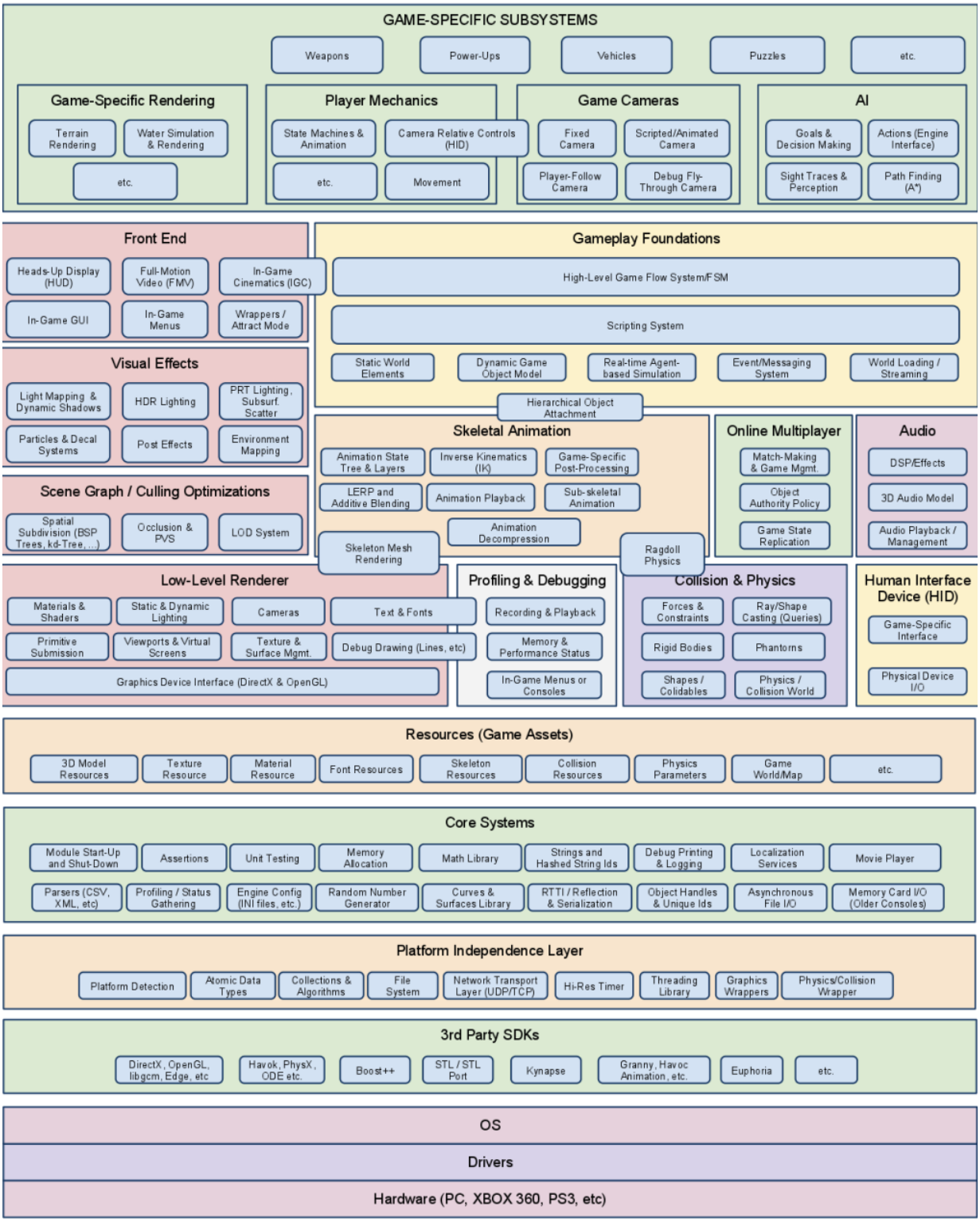


Figura 8: Esquema d'un motor de videojocs

5.2 Exemples de motors de videojocs

- **OpenGL**

Figura 9: Logo de OpenGL

OpenGL és una API multillenguatge i multiplataforma orientada a programar aplicacions i jocs de 3 dimensions. **Open G**raphics **L**ibrary va ser desenvolupada per Silicon Graphics Inc i és una API simple i estable.

Aquest motor és suportat en molts videojocs actuals i competeix directament amb Direct3D de Microsoft.

- **Unreal Engine**

Figura 10: Logo de Unreal Engine

Unreal Engine és un motor desenvolupat per Epic Games i és un dels més utilitzats degut a la seva facilitat d'ús i a la seva manera de programar jocs, la qual no requereix escriure cap línia de codi.

És un motor gratuït que permet la comercialització de jocs creats amb ell mentre no es passin dels 3000$ per quadrimestre.

Alguns dels videojocs que utilitzen Unreal Engine com a motor de videojoc són: *Fortnite*, *Final fantasy 7 Remake*, *Batman Arkham Knight, Dragon Ball FighterZ* i *Kingdom Hearths 3*.

Figura 11: Jocs que utilitzen Unreal Engine

- **Unity**

Figura 12: Logo de Unity

Unity és un motor de videojocs creat per Unity Technologies. Es tracte de un motor multiplataforma utilitzat majoritàriament per empreses independents a causa de la gran quantitat de contingut gratuït i l'activa comunitat. Treballa en Windows i en Mac OS X i els jocs creats utilitzant aquest programari poden ser executats en Windows, Mac, Xbox 360, PlayStation 3, Wii, iPad, iPhone i Android.

Al igual que Unreal Engine, permet comercialitzar els videojocs creats de manera gratuïta mentre no es passi de cert benefici anual.

Tot i ser popular en empreses independents, també ha estat utilitzat per grans empreses com *Blizzard*. El gran joc de cartes digital *HerthStone* va ser creat amb el motor Unity. Veure Figura 13.

Figura 13: HearthStone

5.3 Motor triat

El motor de videojocs triat per la realització del projecte ha estat Unity. La seva baixa corba d'aprenentatge, l'amplia comunitat activa i la gran quantitat de documentació disponible són qualitats que fan d'aquest motor l'opció més adient pel projecte a realitzar. A més, Unity és un motor que s'utilitza al mon laboral per crear jocs i aplicacions per un gran nombre de plataformes.

Tot i que Unreal Engine supera a Unity en certs aspectes com motor de renderització o gestió de memòria, aquests no eren estrictament necessaris per la creació d'un petit videojoc 2D amb estil pixelart.

6. Requisits del sistema

En aquest capítol s'especifiquen els requeriments de l'aplicació. Aquests es poden dividir en dos apartats, requeriments funcionals i no funcionals

6.1 Requeriments funcionals

Aquests requeriments fan referència a les funcionalitats que el sistema he de dur a terme.

- El jugador haurà de ser capaç de desplaçar-se, saltar utilitzant les tecles del teclat i atacar utilitzant del ratolí.

- El jugador podrà derrotar als enemics del mapa.

- El jugador, una vegada derrotat un "Boss", rebrà una habilitat que canviarà la manera de desplaçar-se.

- Els enemics hauran de poder perseguir el jugador a través del mapa.

- Els enemics podran atacar al jugador i, en cas de derrotar-lo, el jugador tornarà al principi del joc.

- El generador de mapa sempre crearà el mapa amb, com a mínim, un camí de principi a final i contindrà elements generats de forma procedural i elements construïts prèviament.

- Un dels elements procedurals seran obstacles que el jugador només podrà superar una vegada obtinguin l'habilitat corresponent.

- Els diferents elements del videojoc podran col·lisionar entre ells.

6.1 Requeriments no funcionals

Aquests requeriments fan referència a les qualitats i característiques de la maquinària i que s'han de tenir en compte a l'hora de dissenyar el videojoc, com ara els recursos necessaris per a un bon funcionament i la seguretat necessària del sistema.

Donat que el programa només pot ser usat en mode usuari, no es necessiten mesures de seguretat addicionals per controlar l'accés al programa, i al no guardar cap tipus de dades confidencials, no comptem amb cap classe de protecció de dades. Pel que fa a restriccions de plataforma, al haver realitzat el videojoc amb un motor de videojocs, les restriccions de plataforma seran les restriccions que tingui el motor.

Pel que fa als requeriments tècnics, la aplicació ha estat testejada en un ordinador amb les següents prestacions donant un rendiment acceptable:

- **CPU:** 11th Gen Intel(R) Core(TM) i7-11800H
- **GPU:** NVIDIA GeForce RTX 3060 Laptop
- **RAM:** 16 GB
- **SO:** Windows 11 Home

D'altra banda els requeriments mínims per poder executar aquest joc seran els requeriments mínims dol motor Unity, que són els següents:

- Sistema operatiu: Windows XP SP2+, Mac OS X 10.8+, Ubuntu 12.04+, SteamOS+.
- Targeta gràfica: DX9 (model de shader 3.0) o DX11 amb capacitats de funcions de nivell 9.3.
- CPU: Compatible amb el conjunt d'instruccions SSE2.

7. Estudi i presa de decisions

En aquest apartat es descriuran les llibreries, els programés utilitzats al llarg del projecte i el sistema operatiu sobre el que s'ha implementat el videojoc. També es donarà una descripció específica de l'ús que se li han donat a cascuna de les eines utilitzades durant el projecte.

7.1 Sistema Operatiu

Els sistema operatiu utilitzat al llarg de la realització del projecte ha estat el Windows 11 Home edition x64.

7.2 Software utilitzat

A continuació es descriuran breument cada un dels softwares utilitzats i quin paper complien en el desenvolupament.

7.2.1 C#

Al llarg del projecte s'ha utilitzat el llenguatge de programació C#, el qual esta orientat a objectes i va ser desenvolupat per Microsoft com a part de .NET. Tot hi que va derivar dels llenguatges C i C++, també conté similituds amb Java i diferents millores d'altres llenguatges de programació.

És un dels 3 llenguatges suportats per Unity, els altres dos sent Boo i JavaScript. S'ha escollit C# pel desenvolupament del projecte perquè és actualment el llenguatge estàndard per al desenvolupament de videojocs a les últimes versions de Unity i perquè al llarg de la carrera s'ha fet un gran ús de llenguatges semblants.

7.2.2 Visual Studio Code

S'ha escollit Visual Studio Code per desenvolupar el codi en C#. És un entorn desenvolupat per Microsoft i permet desenvolupar aplicacions en una gran quantitat de llenguatges de programació diferents gràcies a les extensions gratuïtes de les que disposa.

Tot i que Unity recomana utilitzar Visual Studio, s'ha escollit Visual Studio Code, que ha diferencia de Visual Studio no és un IDE si no un editor de text orientat a codi, perquè és un programa molt flexible i conta de moltes ajudes i eines per facilitar la implementació de codi, a demés de ser un software gratuït.

7.2.3 Unity

En els punts 5.2 i 5.3 ja s'ha explicat que és i per quines raons s'ha escollit. En aquest capítol s'explicarà les seves parts i com està distribuït.

Unity és l'element més important en el desenvolupament del projecte, ja que es creen totes les escenes i elements del videojocs dins de l'editor.

Els videojocs en Unity es divideixen en escenes, les quals són pantalles per les que el jugador es pot desplaçar i poden ser des de menús a nivells complets. El dissenyador pot crear i manipular els elements del joc dins d'aquesta escena.

- Entorn de treball de Unity

L'entorn de treball de Unity és còmode i intuïtiu i permet moure i ordenar les finestres com l'usuari vulgui. Conta de un gran nombre d'eines a utilitzar però en aquest apartat es parlarà de 6 en específic que caldrà conèixer per al desenvolupament del videojoc. Veure Figura 14.

1. **Visió d'escena**
   Finestra des de la que podem interactuar amb l'escenari en 3D o 2D i modificar, afegir, o treure els elements que conté la nostra escena actual. Aquesta finestra és l'eina més utilitzada ja que és la que utilitzem per construir els nivells i menús.

2. Visió de joc

Finestra que mostra en tot moment com és veurà el joc quan aquest s'estigui executant. Des d'aquesta finestra es realitzaran les proves d'execució del joc.

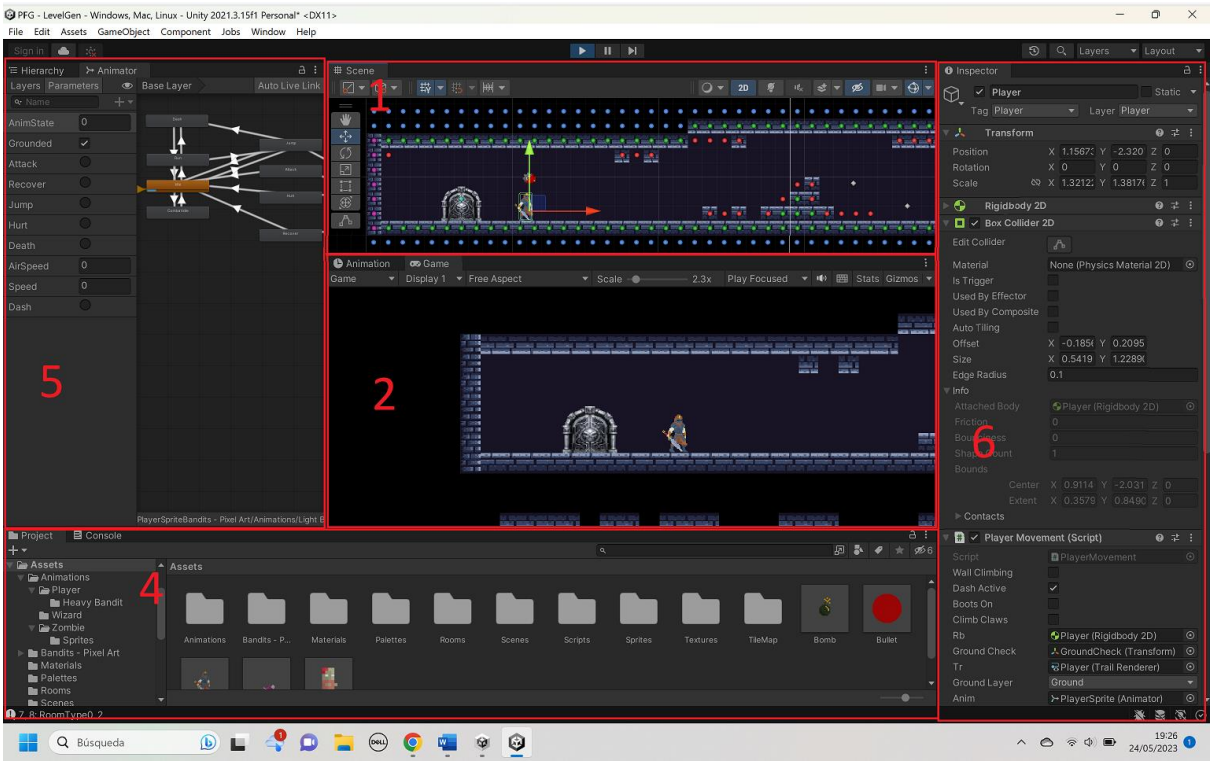


Figura 14: Entorn de treball de Unity

3. Jerarquia de l'escena

Aquest índex mostra tots els GameObjects de l'escena actual a l'igual que la relació jeràrquica que puguin tenir. A la Figura 22 podem veure com actualment tenim seleccionat l'element OptionsMenu de la jerarquia.

4. Jerarquia de fitxers

Aquest índex permet accedir a tots els elements utilitzats en el projecte i mostra sempre a la part esquerra l'arbre de directoris actual junt al contingut de la carpeta en la que ens situem.

5. **Animator**
Eina que permet la creació d'un animationControler, que és l'element responsable de gestionar les animacions dels elements. Des d'aquesta eina s'han creat i estructurat tots els arbres d'animació del projecte.

6. **Inspector**
L'inspector es tracta d'un apartat que permet visualitzar tots els Components que un element de l'escena té relacionats. Aquests Components poden ser qualsevol classe d'atribut com ara la posició de l'objecte o un script associat.

- Conceptes importants

Per poder comprendre els següents capítols de la memòria, és important conèixer alguns dels conceptes més importants de Unity.

1. **Textura**
Una textura és una imatge que s'ha projectat a un modelat per tal de donar-li una experiència visual, o que s'ha utilitzat com un element d'una interfície gràfica.

2. **Material**
Els materials són els elements que controlen l'aparença visual dels nostres GameObjects i normalment estan compostos per una textura.

3. **GameObject**
Els GameObjects es tracten de tots els elements que es dipositen en una escena i que es poden desplaçar, escalar i eliminar. Estan formats per diferents Components, o atributs, que li otorguen característiques especials.

Els GameObjects poden ser configurats i guardats a disc per tal de poder utilitzar-los de forma repetida sense haver de inicialitzar-los cada cop, aquests elements guardats són anomenats Prefabs. En aquest cas els Prefabs són un element molt important del programa, ja que al crear el mapa de

forma procedural és necessari tenir totes les peces que el poden formar el mapa guardades i configurades a memòria.

4. Component

Els components agrupen tots els atributs que un GameObjects podria arribar a tenir des de elements 2D, llum, so, animacions, ... L'únic Component que és obligatori tenir per a tots els GameObjects és el Component Transform, el qual dona la posició, rotació i escala del GameObject.

5. Transform

Component d'un GameObject que diu la posició, rotació i escala.

6. Collider

El collider és un component de Unity que permet saber quan dos objectes amb un component collider es posen en contacte dins l'escena.

Depenent de si es selecciona l'opció "IsTrigger" dins del Inspector d'aquest component, l'objecte pot tornar-se intangible però enviar un senyal cada vegada que un altre collider es superposa en l'espai o, en cas de que "IsTrigger" estigués desactivat, detectar i crear col·lisions entre objectes.

7. RigidBody

El Rigidbody és un component de Unity que s'utilitza per fer que un element sigui afectat per les físiques del joc. Aquest component permet conferir una massa, i una fricció als elements, i restringir per quin dels eixos del mon l'element es podrà desplaçar i sobre quins eixos podrà rotar . Normalment els Rigidbodys van associats amb els colliders, ja que, per a que un element en moviment pugui interaccionar amb un collider, necessita tenir un rigidbody associat.

8. 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.

9. Mesh filtre

Component que permet agafar un modelat en 3D o 2D dels Assets i passar-lo a través del MeshRenderer per poder visualitzar-lo a pantalla.

10. Mesh renderer

Component que agafa un modelat de punts d'un component MeshFilter i el renderitza en la posició transform del GameObject amb el que estigui associat.

11. Canvas

Component de Unity que permet la creació de interfícies d'usuari i de menús en el videojoc.

12. Sound source

Component de Unity que permet reproduir un clip d'àudio des de un punt del joc. Aquest element és capaç de reproduir l'àudio de tal forma que el volum disminueixi en funció de la distancia a la que es trobi la càmera del punt de so.

13. Tag

Es poden dividir els gameObjects utilitzant unes etiquetes anomenades Tag. Aquestes etiquetes serveixen per poder agrupar, identificar i buscar els diferents objectes dins del videojoc. Un exemple seria l'etiqueta "Enemy", el qual ens diu que l'element en qüestió és un enemic, o l'etiqueta "Player", el qual ens diu que l'element trobat és el personatge controlat pel jugador.

14. Layer

A l'igual que els Tags, les Layers són divisions que podem aplicar als GamoObjects. Les Layers divideixen els GameObjects en capes per decidir renderitzar només una part de l'escena, o be per ignorar certs colliders a l'hora de realitzar els nostres RayCasts.

15. Animator

Component utilitzat per tal de comunicar un GameObjects amb l'animationControler. Els AnimationControlers són uns diagrames de transacció d'animacions que permeten a un GameObject realitzar una animació si les condicions necessàries es compleixen.

- API de Unity

Unity ofereix una amplia API de treball a través de la qual, utilitzant Scripts, podem interactuar amb els GameObjects i els seus components. A continuació s'explicaran els elements de l'API necessaris per entendre el funcionament de l'aplicació.

1. GameObject

Classe principal de totes les entitats de Unity

Variable	Retorn	Descripció
layer	int	Retorna la Layer en la que es troba el GameObject
tag	String	Retorna el Tag al qual pertany el GameObject
transform	Transform	Retorna el component transform del GameObject
name	String	Retorna el nom del GameObject

Mètode	Retorn	Descripció
GetComponent(type t)	type	Retorna el component de tipus p adherit al GameObjects, si no en te cap retorna null.
GetComponentChildren(type t)	type	Retorna el primer component de tipus t trobat en els fills del GameObject fent una cerca depth first pels fills. Si no en troba cap retorna null
GetComponentParent(type t)	type	Retorna el component de tipus p adherit al pare GameObjects, si no en te cap retorna null.
Destroy(object o)	void	Remou l'objecte o de l'escena
DontDestoryOnLoad(object o)	void	Fa o evita que l'objecte sigui destruït al canviar d'escena
Instantiate(Object original, Vector3 position, Quaternion rotation);	void	Retorna un clon de l'objecte original a la posició position. amb una rotació igual a rotation.

2. Monobehavior

Monobehavior és la classe de la qual deriva cada script de Unity. Els components de Unity tenen un cicle de de vida intern i Monobehavior exposa aquest cicle per mètodes que podem sobreescriure. Veure Figura 15.

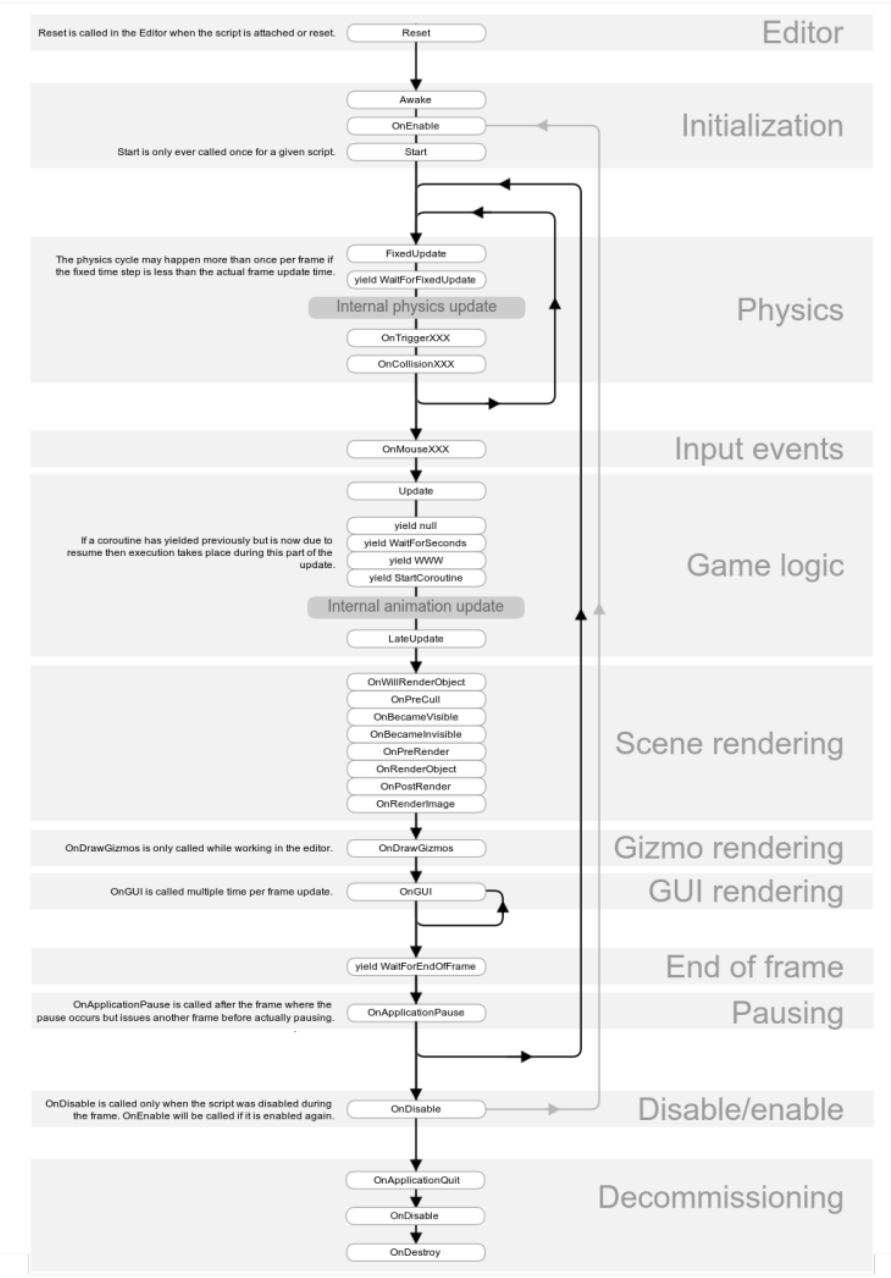


Figura 15: Cicle de vida d'un component

Mètode	Retorn	Descripció
Awake ()	void	Funció que es crida al carregar la instància de l'Script.
Start()	void	Funció que es crida en el primer frame posterior a la creació de l'Script.
Update()	void	Funció que es crida cada frame durant l'execució.
FixedUpdate()	void	Funció cridada cada frame de framerate de l'aplicació. Utilitzada per gestionar els moviments dels objectes en temps de framerate real.
OnTriggerEnter(Collider collider)	void	Funció que es crida quan un element col·lisiona amb un trigger.
OnTriggerExit(Collider collider)	void	Funció que es crida quan un element deixa de col·lisionar amb un trigger
OnTriggerStay(Collider collider)	void	Funció que es crida cada frame que un element passa tocant un trigger.

3. Transform

Classe associada amb el component transform d'un GameObject i que permet modificar dit Componen.

Variable	Retorn	Descripció
localScale	Vector3	Retorna l'escala del GameObject en els eixos x, y ,z.
position	Vector3	Retorna la posició actual respecte l'eix del mon.
rotation	Quanternion	Retorna la rotació de l'objecte respecte el mon.
right	Vector3	Retorna un vector que apunta a la part positiva de l'eix x del GameObject.

Mètode	Retorn	Descripció
CompareTag(string name)	bool	Retorna true si la tag del objecte és igual a name, false altrament.
GetComponent<type>()	type	Retorna el Component type del objecte.

4. Collider

Classe de la que provenen tots els colliders primitius de Unity junt al MeshCollider. Aquests Collider poden ser “IsTrigger” colliders o colliders normals, però en els dos casos criden als mètodes OnTriggerEnter(), OnTriggerExit() i OnTriggerStay(). Les funcions i variables utilitzades són les heretades de la classe GameObject.

5. Input

Interfície de Unity utilitzada per consultar les entrades del sistema.

Mètode	Retorn	Descripció
GetAxis(string axisName)	float	Retorna el valor de l'eix amb nom axisName.
GetKey (string name)	bool	Retorna cert si la tecla amb nom name està presa.
GetMouseButtonDown (int button)	bool	Retorna cert quan el boto del ratolí numero button es pressiona.

6. SceneManager

Interfície de Unity que ens permet gestionar les escenes del videojoc.

Mètode	Retorn	Descripció
LoadScene (int sceneIndex)	void	Carrega la escena numero sceneIndex

7. AudioSource

Classe associada a un Componenet AudioSource que permet canviar les seves característiques a través de ell.

Variable	Retorn	Descripció
volume	float	Variable que controla el volum de l'AudioSource.

Mètode	Retorn	Descripció
Play()	void	Dona inici a la reproducció del clic assignat a l'AudioSource.
Stop()	void	Para la reproducció del clic assignat a l'AudioSource.

8. Animator

Classe associada a un Componenet Animator que permet canviar les seves característique.

Mètode	Retorn	Descripció
SetInteger(string name, int value)	void	Canvia el valor de la variable entera de control name amb el valor value.
SetBool (string name, bool value)	void	Canvia el valor de la variable booleana de control name amb el valor value.
SetTrigger(string name)	void	Activa la variable trigger de control name.

9. RigidBody

Classe associada a un Componenet Rigidbody que permet canviar les seves característiques.

Variable	Retorn	Descripció
velocity	Vector3	Vector que marca la direcció i velocitat del Rigidbody.

10. LayerMask

Classe que utilitzem per tal de filtrar diferents tipus de colliders a l'hora de realitzar interaccions com comprovar que el personatge esta tocant el terra. Aquesta classe és utilitzada únicament com a filtre i no se'n crida cap funció.

- Llibreries utilitzades

  - **UnityEngine:** Llibreria principal del motor Unity.

  - **UnityEngine.SceneMangement:** Llibreria que conté les funcionalitats necessàries per desplaçar-nos entre les escenes del videojoc.

  - **UnityEngine.UI:** Llibreria que conté les funcionalitats necessàries per interactuar amb els elements de la interfície gràfica del videojoc.

  - **System:** Llibreria fonamental de C# que conté les classes més utilitzades i bàsiques que descriuen els valors i referències de tipus de data més comuns.

  - **System.Collections:** Llibreria que conté interfícies i classes que defineixen col·leccions de dades com ara llistes, cues, taules i diccionaris.

  - **System.Text.RegularExpressions:** Llibreria que conté interfícies i classes orientades a la creació i comparació d'expressions regulars.

  - **TMPro:** Libreria que conté interfícies i classes orientades a la creació i edició de textos.

7.2.4 Microsoft Word

Microsoft Word és un software d'edició de text comercialitzat per Microsoft i desenvolupat per Richard Brodie en 1983 sota el sistema operatiu DOS.

S'ha escollit aquest software a l'hora d'escriure la memòria perquè es disposa d'una llicència legal i la familiaritat amb ell.

7.2.5 Gantt Project

Gantt Software és una aplicació gratuïta per realitzar diagrames de Gantt, i és l'aplicació utilitzada per generar els diagrames del projecte.

Un diagrama de Gantt és una gràfica que conté la cronologia de totes les tasques previstes en un projecte, junt a la seva duració i l'ordre en el qual es realitzaran. Aquests diagrames no indiquen les relacions entre les diferents activitats sinó que simplement especifiquen en quin ordre s'han de fer i quines tasques no poden començar fins que unes altres no hagin acabat.

8. Anàlisi i disseny del sistema

En aquest capítol es descriurà com és i quina temàtica tindrà el videojoc a desenvolupar. També s'explicarà, de forma conceptual, sense entrar en la implementació, els elements més importants per poder entendre el videojoc.

8.1 Descripció general

El videojoc creat és un videojoc 2D d'acció i plataformes de l'estil Roguelike amb elements de videojocs de l'estil Metroidvania. El jugador comença la partida dins d'un calabós, el qual està separat per plantes i controla a un mercenari sense nom que vol aconseguir les riqueses situades a la planta més baixa d'aquest calabós. Per aconseguir-ho, ha de derrotar a tots els enemics i superar tots els obstacles que es trobi en el seu camí.

Més endavant s'explica quines són les característiques que marquen aquest videojoc com a un Roguelike amb elements de Metroidvania. Uns bons exemples dels dos estils de videojocs són Hades com a Roguelike i Hollow Knight com a Metroidvania. Veure figures 16 i 17.

Figura 16: Hades

Figura 17: Hollow Knight

8.1 Disseny del funcionament

El videojoc està situat en un calabós, el qual està dividit per nivells, i cada nivell és genera de forma procedural cada vegada que el jugador hi entra. En cas de que fos derrotat, el jugador tornaria al principi del calabós, només retenint el coneixement guanyat durant la partida anterior i algunes millores aconseguides al derrotar als monstres més poderosos, que es trobaran a cada certs nombre de nivells.

Fins aquí el jocs es pot considerar un Roguelike, però cada millora que el jugador aconsegueix obre noves possibilitats d'exploració a cada nivell. Totes les plantes del calabós tenen la possibilitat de contenir zones delimitades per obstacles que no es podran superar fins que el jugador obtingui la millora corresponent. Aquestes són característiques dels videojocs de l'estil Metroidvania.

En la demo del videojoc presentada per aquest projecte, el jugador haurà de completar dos nivells i enfrontar-se a un "Boss" al tercer nivell, on aconseguirà l'habilitat de "Dash". Aquesta habilitat permetrà al jugador travessar als enemics, els seus atacs i certes parets marcades per un color diferent. D'aquesta manera, no tan sols guanya la capacitat de explorar noves zones, si no que també obté més opcions a l'hora de moure's i enfrontar-se als enemics.

Una vegada derrotat el "Boss", el jugador tornarà al menú d'inici, on tindrà l'opció de crear una nova partida o seguir jugant els nivells de la demo retenint la millora aconseguida.

8.1.1 Millores

Tot i que el jugador només té accés a la primera millora dins la demo, hi hauran tres millores al joc complet:

- Dash

  Aquesta és la primera millora que el jugador pot obtenir i, com s'ha explicat anteriorment, permet al personatge travessar certes parets a demés d'enemics i els seus atacs. Durant el combat s'ha d'utilitzar amb seny, ja que una vegada activada, no es pot tornar ha activar fins a que no hagin passat uns segons.

- Botes reforçades

Aquesta és la segona millora que el jugador pot obtenir. Amb aquestes botes, es podrà caminar per zones amb obstacles perillosos, com espines gegants o punxes de ferro, sense prendre cap mal.

- Urpes d'escalar

Aquesta és l'última millora que el jugador pot obtenir. Amb les urpes, el jugador podrà escalar qualsevol paret, donant accés a més maneres de moure's verticalment per tot el mapa sense necessitat de plataformes.

8.1.1 Personatge principal

El personatge principal és l'element més clau del videojoc ja que, a través d'ell, el jugador interactuarà amb l'entorn.

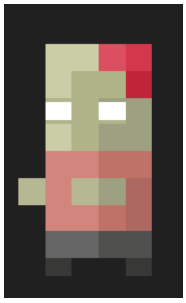
El personatge es tracta d'un mercenari armat amb una espasa i la determinació d'arribar al final del calabós. Veure Figura 18.

Sense cap millora, les accions que pot realitzar aquest personatge són moure's de dreta a esquerra, saltar i atacar amb l'espasa.

Figura 18: Mercenari sense nom

8.1.2 Enemics

- Zombi



El zombi és un dels enemics més comuns del joc. L'única manera d'atacar que tenen és xocant contra el personatge, però una vegada s'adonen de que el jugador està a prop, no deixaran de perseguir-lo fins que siguin derrotats. Tot i això, no són massa llestos i una vegada comencen a perseguir, perden noció de qualsevol obstacle o de si tenen terra sota els seus peus. Veure Figura 19.

Figura 19: Zombi

- Mag

El mag és un enemic més espavilat. El seu mitjà d'atac principal són uns míssils de màgia que llencen en línia recta. Prefereixen lluitar a distància, per tant, si detecta que el jugador està massa a prop, intentarà allunyar-se abans de preparar el següent atac. La seva màgia travessa obstacles, però tarda uns moments en llançar-se i tenen poca vida, així que poden ser derrotats amb un parell de cops d'espasa mentre el jugador pugui esquivar els seus atacs.

Figura 20: Mag

- Nigromant

El nigromant és un dels "Boss" d'aquest videojoc i al que s'enfrontarà el jugador al arribar al final de la demo. És la forma més poderosa d'un zombi i, per tant, utilitza el mateix model gràfic. Tot i això, és més gran, té molta més vida i té un parell d'habilitats més que els zombis normals. El nigromant és capaç de llençar bombes màgiques al jugador que exploten al contacte i una vegada ha perdut la meitat de la seva vida, pot invocar zombis per a que persegueixin i ataquin al jugador, a demés de llençar les bombés amb més freqüència. Una vegada derrotat per primera vegada, el jugador guanya la millora del "Dash".

8.1.3 Interfícies d'usuari

- Menú principal

Aquest és el menú que el jugador es troba quan inicia el joc i al que torna quan completa la demo o és derrotat. El botó "Play" porta al menú d'inici de joc, el botó "Volume" porta al menú de so i el botó "Quit" tanca l'aplicació. Veure Figura 21.

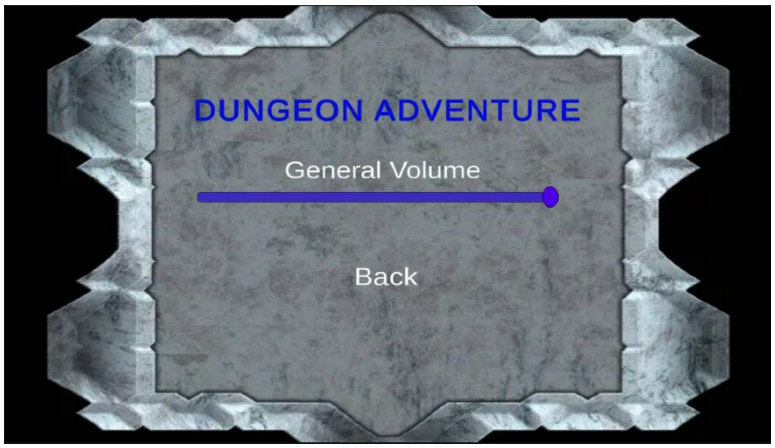
IFigura 21: Menú principal

- Menú d'inici de joc

Figura 22: Menú d'inici de joc

Aquest és el menú des del que el jugador pot començar la partida. Té l'opció de començar una partida nova o continuar una guardada, on el personatge retindrà les millores obtingudes. En cas de que no hi hagués cap partida guardada, el botó "Load" estaria desactivat. Es pot tornar al menú principal amb el botó "Back".

- Menú de so



El jugador pot accedir a aquest menú des del menú principal i aquí pot regular el volum general del videojoc. Es pot tornar al menú principal amb el botó “Back”.

Figura 23: Menú de so

- Interfície de joc

Figura 24: Interfície de joc

Com el jugador no té molts recursos dels que preocupar-se, la interfície del joc és bastant simple. Només conté el personatge principal i una barra de vida a la cantonada superior esquerra. En cas d’enfrontar-se a un “Boss” la seva barra de vida es mostraria a la part inferior de la pantalla.

Com que l’objectiu es trobar un camí fins a la planta més baixa, la càmera està centrada una mica per sobre del jugador per mantenir les plantes per sota del jugador misterioses fins que no es trobi una manera de baixar. Veure Figura 24.

- Menú de pausa

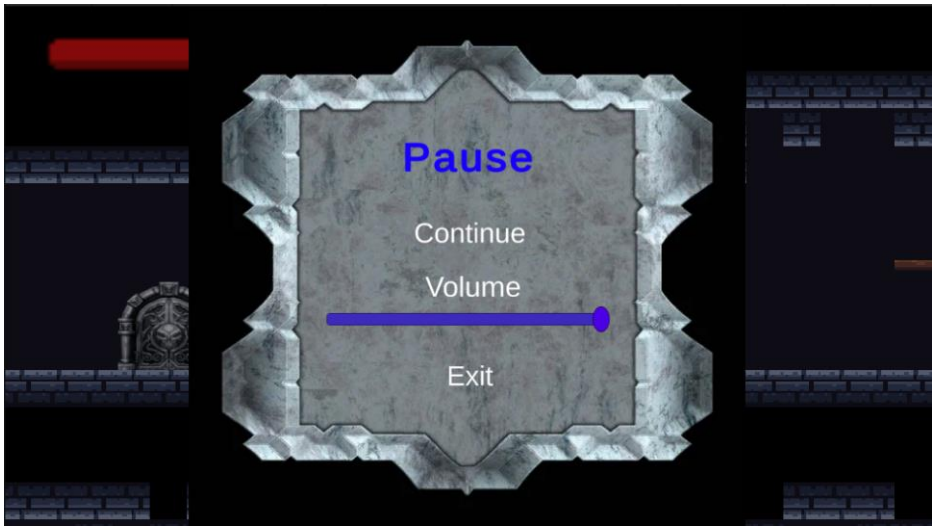


Figura 25: Menú de pausa

El jugador pot accedir a aquest menú durant la partida pressionant el botó d'escape del teclat. Des d'aquí es pot continuar la partida clicant el botó "Continue" o tornant a pressionar el botó "esc" del teclat, ajustar el volum general del videojoc i tornar al menú principal amb el botó "Exit". Veure Figura 25.

8.2 Identificació dels actors

En aquest capítol s'identifiquen els actors principals de l'aplicació. Un actor és una entitat que interactua amb l'aplicació assolint un rol o estat concret.

Com el videojoc no disposa de cap classe per diferenciar entre usuaris ni de cap manteniment, poden assumir que només existeix un actor, el jugador, que interactuarà amb el sistema.

8.3 Casos d'ús

Un cas d'ús és una descripció dels passos que es realitzen entre un actor i el sistema com a resposta d'un esdeveniment causat pet l'actor. En aquest apartat s'utilitzen casos d'ús per definir les activitats i comportaments que es poden realitzar dins del videojoc.

A continuació es poden veure els diagrames creats per cada una de les interfícies que el jugador es pot trobar.

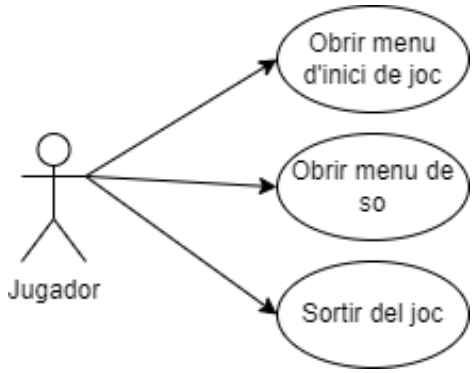


Figura 26: Diagrama de casos d'ús del menú principal

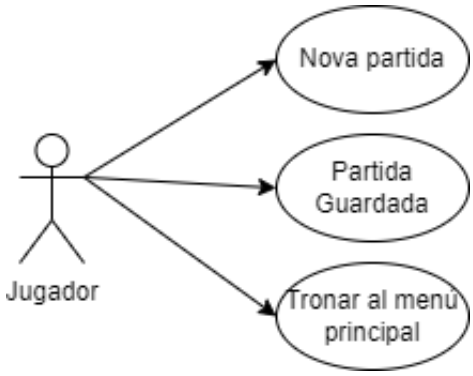


Figura 27: Diagrama de casos d'ús del menú d'inici de joc

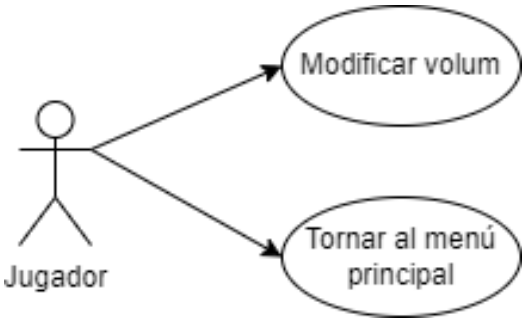


Figura 28: Diagrama de casos d'ús del menú de so

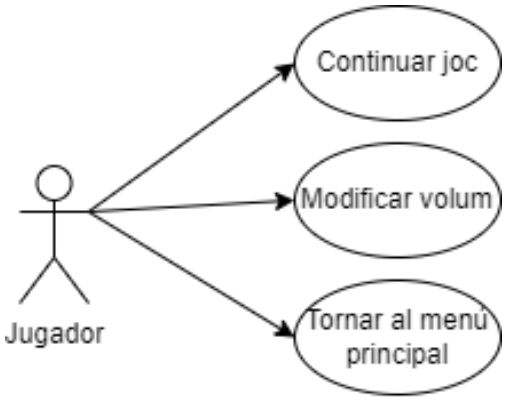


Figura 29: Diagrama de casos d'ús del menú de pausa

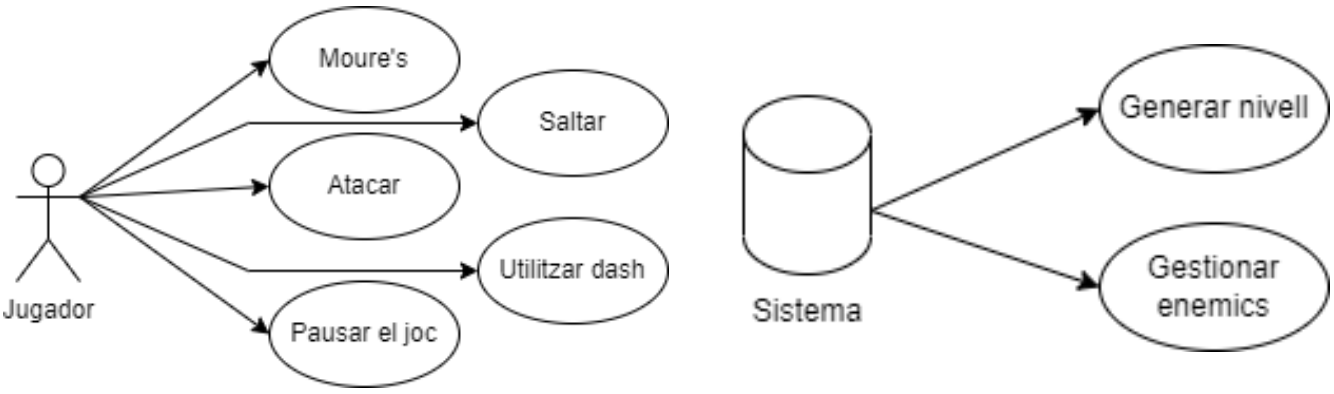


Figura 30: Diagrama de casos d'ús de la interfície de joc

8.3.1 Fitxes de casos d'ús

En aquest apartat s'utilitzen fitxes de cas d'ús per explicar cada cas d'ús, l'actor que el realitza, les precondicions, fluxos principals i les observacions trobades sobre aquest.

Fitxa de cas d'ús: Obrir menú d'inici de joc	
Descripció	El jugador selecciona l'opció d'obrir el menú d'inici de joc del menú principal
Actor	Jugador
Precondició	S'ha iniciat el programa
Flux principal	1- Escollir l'opció Play del menú principal.
Flux alternatiu	Cap
Postcondició	S'ha executat l'opció escollida
Observacions	Cap

Fitxa de cas d'ús: Obrir menú de so	
Descripció	El jugador selecciona l'opció d'obrir el menú de so del menú principal
Actor	Jugador
Precondició	S'ha iniciat el programa
Flux principal	1- Escollir l'opció Volume del menú principal
Flux alternatiu	Cap
Postcondició	S'ha executat l'opció escollida
Observacions	Cap

Fitxa de cas d'ús: Sortir del joc	
Descripció	El jugador selecciona l'opció de sortir del menú principal
Actor	Jugador
Precondició	S'ha iniciat el programa
Flux principal	1- Escollir l'opció Quit del menú principal.
Flux alternatiu	Cap
Postcondició	S'ha executat l'opció escollida
Observacions	Cap

Fitxa de cas d'ús: Nova partida	
Descripció	El jugador selecciona l'opció de començar una partida nova del menú d'inici de joc
Actor	Jugador
Precondició	S'ha accedit al menú d'inici de joc
Flux principal	1- Escollir l'opció New Game del menú d'inici de joc.
Flux alternatiu	Cap
Postcondició	S'ha executat l'opció escollida
Observacions	Cap

Fitxa de cas d'ús: Partida guardada	
Descripció	El jugador selecciona l'opció de continuar una partida guardada del menú d'inici de joc
Actor	Jugador
Precondició	S'ha accedit al menú d'inici de joc i existeix una partida guardada
Flux principal	1- Escollir l'opció Load del menú d'inici de joc.
Flux alternatiu	Cap
Postcondició	S'ha executat l'opció escollida
Observacions	Cap

Fitxa de cas d'ús: Tornar al menú principal	
Descripció	El jugador torna al menú principal i, en cas de realitzar-se des de la interfície del joc, es guarda la partida
Actor	Jugador
Precondició	S'ha accedit al menú d'inici de joc o el menú so o el menú de pausa
Flux principal	1- Escollir l'opció Back o Exit del menú d'inici de joc o del menú so o del menú de pausa
Flux alternatiu	Cap
Postcondició	S'ha executat l'opció escollida
Observacions	Cap

Fitxa de cas d'ús: Modificar volum	
Descripció	El jugador modifica el valor de so del videojoc
Actor	Jugador
Precondició	S'ha accedit al menú de so
Flux principal	1- Desplaçar indicadors de volum del menú de so. 2- Guardar el nou valor de so al sistema

Flux alternatiu	Cap
Postcondició	La musica es reprodueix amb els nous valors de so
Observacions	Cap

Fitxa de cas d'ús: Moure's	
Descripció	El jugador es desplaça pel mapa
Actor	Jugador
Precondició	S'ha iniciat una partida
Flux principal	1- Prémer una de les tecles de moviment 2- Desplaçar el personatge cap a la direcció indicada
Flux alternatiu	Cap
Postcondició	El personatge s'ha desplaçat a la direcció desitjada
Observacions	Cap

Fitxa de cas d'ús: Saltar	
Descripció	El jugador salta
Actor	Jugador
Precondició	S'ha iniciat una partida
Flux principal	1- Prémer la tecla de saltar 2- El personatge salta
Flux alternatiu	Cap
Postcondició	El personatge ha saltat
Observacions	Cap

Fitxa de cas d'ús: Atacar	
Descripció	El jugador realitza un atac
Actor	Jugador
Precondició	S'ha iniciat una partida
Flux principal	1- Clicar el botó d'atacar. 2- El personatge realitza un atac i activa l'efecte sonor d'atacar 3- Si hi ha un o més enemics en contacte amb el jugador aquests reben dany
Flux alternatiu	Cap
Postcondició	El jugador ha realitzat un atac.
Observacions	Cap

Fitxa de cas d'ús: Utilitzar dash	
Descripció	El jugador utilitza l'habilitat de dash
Actor	Jugador
Precondició	S'ha iniciat una partida i s'ha aconseguit la millora de dash
Flux principal	1- Prémer la tecla de dash 2- El personatge realitza un dash
Flux alternatiu	Cap
Postcondició	El jugador ha realitzat un dash.
Observacions	Cap

Fitxa de cas d'ús: Pausar el joc	
Descripció	El jugador pausa la partida i accedeix al menú de pausa
Actor	Jugador
Precondició	S'ha iniciat una partida
Flux principal	1- Prémer la tecla de pausa
Flux alternatiu	Cap
Postcondició	La partida queda pausada i s'accedeix al menú de pausa
Observacions	Cap

Fitxa de cas d'ús: Continuar	
Descripció	El jugador resumeix la partida
Actor	Jugador
Precondició	S'ha accedit al menú de so
Flux principal	1- Clicar l'opció Continue del menú de pausa o prémer la tecla de pausa
Flux alternatiu	Cap
Postcondició	S'ha executat l'opció escollida
Observacions	Cap

Fitxa de cas d'ús: Generar nivell	
Descripció	Es genera el nivell de joc
Actor	Sistema
Precondició	S'ha iniciat una partida
Flux principal	1- Generació d'una llavor aleatòria 2- Creació d'un camí de principi a fi del nivell 3- Generar un mapa de les habitacions i passadissos de l'escenari en funció a la llavor escollida i del camí creat 4- Afegir les habitacions pre-creades a l'escenari 5- Afegir les escales per canviar de nivell 6- Generar els enemics a l'escenari en funció de les habitacions generades
Flux alternatiu	Cap
Postcondició	Nivell generat en funció d'una llavor aleatoria
Observacions	Cap

Fitxa de cas d'ús: Generar enemics	
Descripció	Es gestionen i inicialitzen els algorismes d'intel·ligència artificial dels enemics
Actor	Sistema
Precondició	S'ha creat un nivell de joc amb el seu escenari
Flux principal	1- Inicialitzar les IAs dels enemics amb els valors per defecte 2- Eliminar els enemics quan aquests arribin a 0 de vida.
Flux alternatiu	Cap
Postcondició	Els enemics es gestionen al llarg del nivell
Observacions	Cap

8.4 Diagrames d'activitat

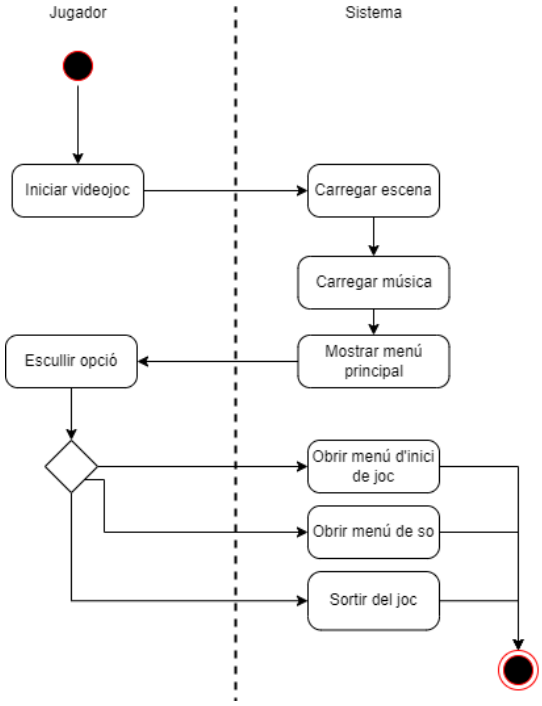


Figura 31: Diagrama d'activitat del menú principal

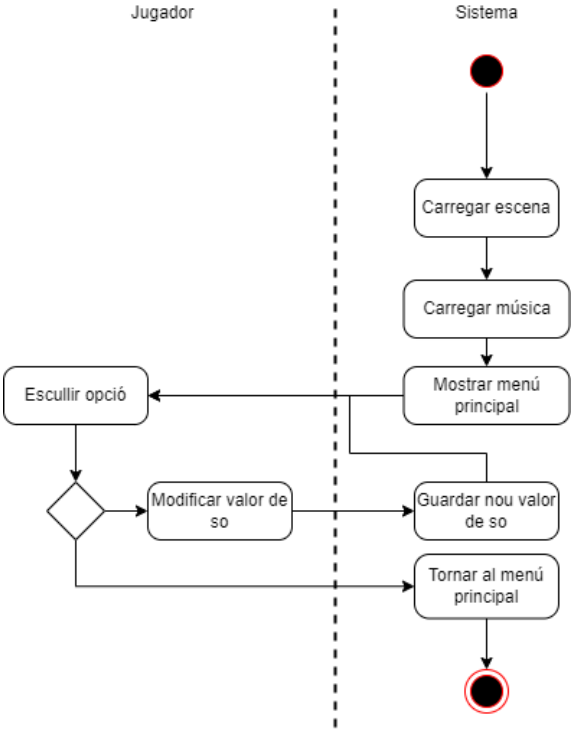


Figura 32: Diagrama d'activitat del menú de so

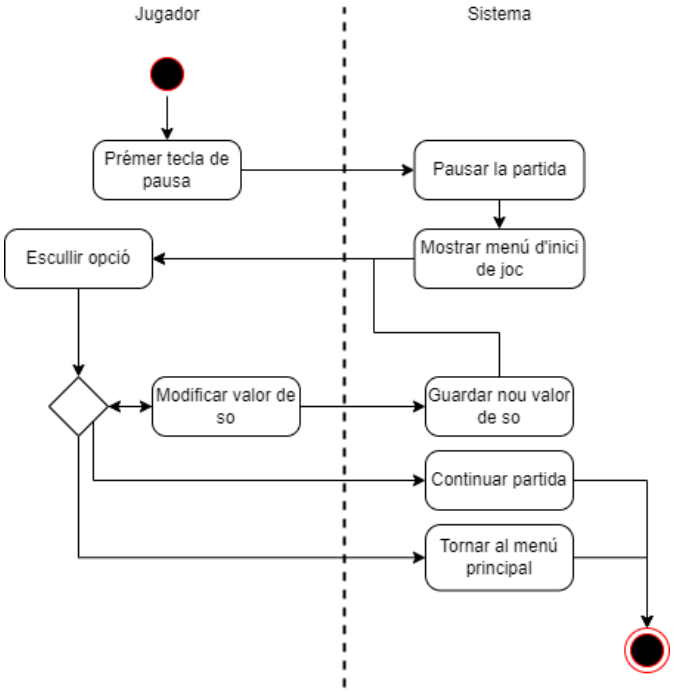


Figura 32: Diagrama d'activitat del menú de pausa

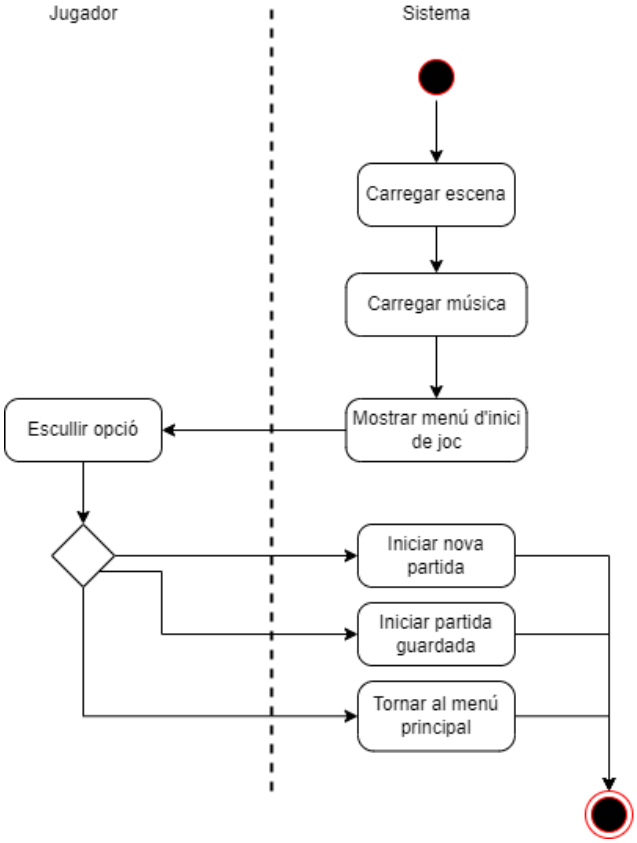


Figura 32: Diagrama d'activitat del menú d'inici de joc

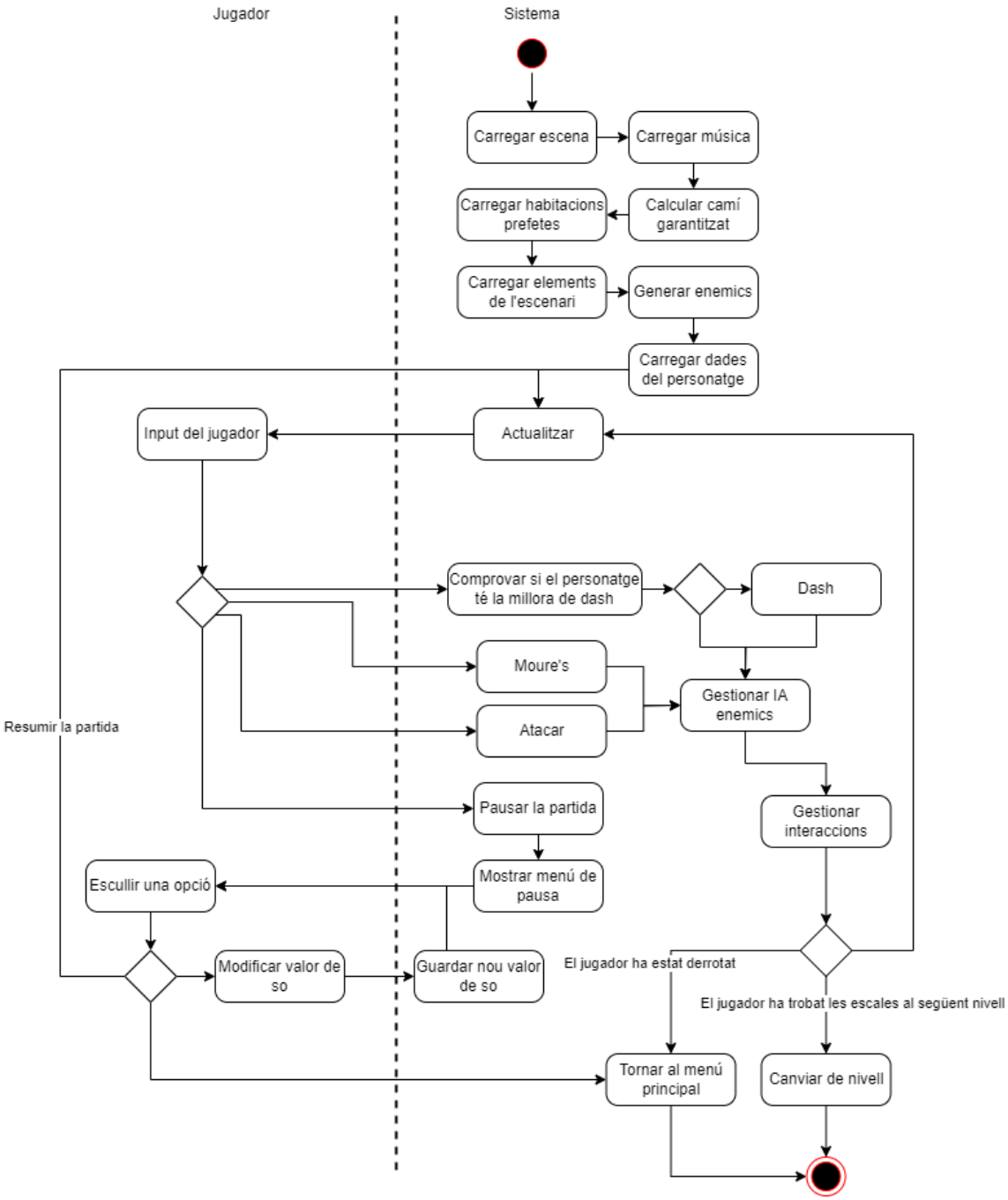


Figura 35: Diagrama d'activitat de la interfície del joc

8.5 Classes i mètodes

En aquest capítol es mostraran les classes que formen el videojoc, les seves funcions i atributs i les relacions entre elles. Com que el diagrama de relacions ha quedat massa gran com per mostrar-ho tot junt, es dividirà en els subgrups: personatge principal, behavior tree, enemics, generador de nivell i UI.

8.5.1 Personatge principal

Aquest grup controla les funcionalitats del jugador. Conté tres classes exclusives del personatge i una que comparteix amb qualsevol element amb punts de salut.

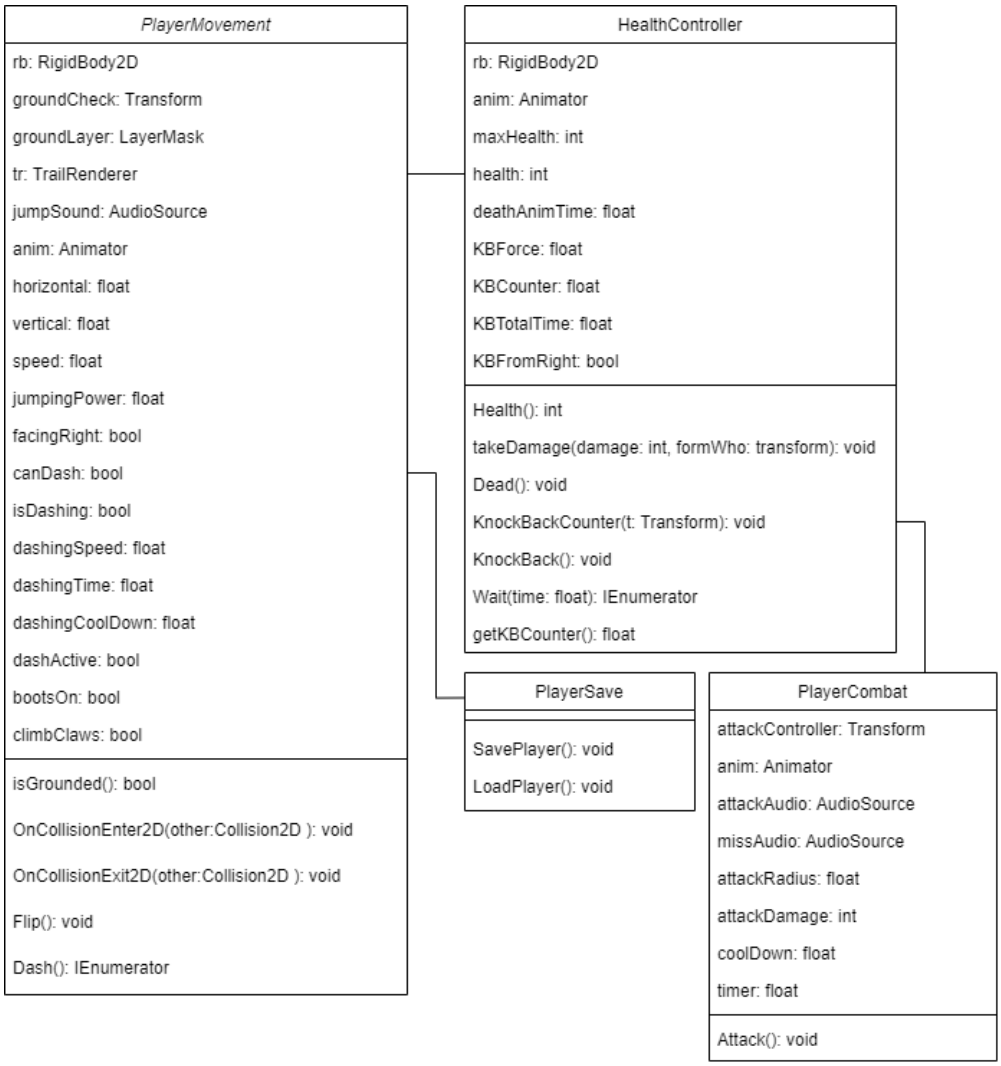


Figura 36: Diagrama de classes del personatge principal

- PlayerMovement

Aquesta classe implementa totes les opcions de moviment que el jugador pot realitzar amb el personatge principal. També controla la utilització de les habilitats obtingudes.

- **Atributs**

  - **rb:** Component RigidBody del personatge. S'utilitza per moure al personatge mitjançant l'aplicació de vectors de força.

  - **groundCheck:** Objecte situat als peus del personatge per detectar si està tocant a terra.

  - **groundLayer:** Component Layer que pertany al terra i les plataformes.

  - **tr:** Component TrailRenderer per indicar si el personatge està fent un dash.

  - **jumpSound:** Component AudioSource per reproduir el so que fa el personatge al saltar.

  - **anim:** Component Animator del personatge per reproduir les animacions de cada acció.

  - **horizontal:** Valor direccional horitzontal del Input del jugador.

  - **vertical:** Valor direccional vertical del Input del jugador.

  - **speed:** Velocitat del personatge al moure's.

  - **jumpingPower:** Velocitat en la que el jugador salta.

  - **facingRight:** Indica si el personatge està mirant a la dreta o no.

  - **canDash:** Indica si el personatge pot realitzar un dash.

  - **isDashing:** Indica si el personatge està realitzant un dash.

  - **dashingSpeed:** Velocitat del dash.

- **dashingCooldown:** Temps a esperar abans de realitzar un altre dash consecutiu.

- **dashActive:** Indica si el personatge ha aconseguit l'habilitat del dash.

- **bootsOn:** Indica si el personatge ha aconseguit la millora de les botes.

- **climbClwas:** Indica si el personatge ha aconseguit la millora de les urpes d'escalar.

- **Mètodes**

  - **IsGrounded():** Indica si el personatge està tocant a terra.

  - **OnCollisionEnter2d(Collision2D other):** Detecta quan el personatge està en contacte amb una paret escalable (en cas de que tingui la millora d'escalar).

  - **OnCollisionExit2d(Collision2D other):** Detecta quan el personatge deixa d'estar en contacte amb una paret escalable (en cas de que tingui la millora d'escalar).

  - **Flip():** Gira el personatge quan es detecta que canvia de sentit.

  - **Dash():** Inicia el dash i controla el temps d'espera necessari per realitzar-lo.

- PlayerCombat

En aquesta classe es controla com lluita el personatge

- **Atributs**

  - **attackController:** Objecte que indica la posició dels atacs en respecte al personatge.

  - **anim:** Component Animator del personatge per reproduir les animacions de cada acció.

- **attackAudio:** Component AudioSource per reproduir el so que fa el personatge al realitzar un atac.

- **missAudio:** Component AudioSource per reproduir el so que fa el personatge al realitzar un atac però no donar-li a res.

- **attackRadius:** Rang d'attack del personatge.

- **attackDamage:** Dany d'atac del personatge.

- **coolDown:** Temps abans de poder realitzar una altre atac consecutiu.

- **timer:** Valor que es compara amb coolDown per saber quan es pot tornar a atacar.

- **Mètodes**

  - **Attack():** El personatge realitza un atac amb l'espasa i tots els enemics dins del rang prenen mal.

- PlayerSave

Aquesta classe gestiona el guardat i carregat de les dades del personatge que es retenen a través d'una partida. Com tot es guarda en PlayerPrefs, la classe no conté cap atribut

- **Mètodes**

  - **SavePlayer():** Es guarden les millores i els punts de vida del personatge.

  - **LoadPlayer():** Si hi han dades guardades, es carreguen al personatge.

- HealthController

Aquesta és una classe que tots els elements del joc amb punts de vida implementen. Controla la salut i el “knock back” que reben els enemics i el jugador al ser atacats.

- **Atributs**
  - **maxHealth:** Els punts de vida màxims.
  - **health:** Els punts de vida actuals.
  - **deathAnimTime:** El temps per reproduir l’animació de morir abans de ser destruït.
  - **KBForce:** La força de l’empeny al rebre un atac
  - **KBCounter:** Contador de temps que es passa en estat “knock back” sense poder moure’s o atacar.
  - **KBTotalTime:** Temps que es passa en estat “knock back”.
  - **KBFromright:** Indica si l’atac s’ha rebut de la esquerra o no.
  - **rb:** Component RigidBody de l’individu.
  - **anim:** Component Animator per per reproduir les animacions de cada acció.

- **Mètodes**
  - **Health():** Retorna els punts de vida actuals.
  - **takeDamage(int damage, Transform formWho):** Disminueix els punts de vida i indica de quin costat s’ha rebut l’atac.
  - **Dead():** Espera a que es reprodueixi l’animació de mort i destrueix l’objecte o, en cas de tractar-se del personatge, retorna al menú d’inici guardant les millores aconseguides.

- **KnockBackCounter(Transform t):** Reseteja el contador KBCouter i guarda cap a quina direcció ha estat empès.

- **KnockBack():** Aplica l'empeny al RigidBody cap a la direcció adient.

- **Wait(float time):** S'espera el temps entrat.

- **getKBCounter():** Es retorna el comptador de "knock back".

8.5.2 Behavior Tree

Tot i que existeixen llibreries suportades per Unity per fer servir la metodologia de Behavior Trees, després d'investigar-les em van semblar massa restrictives a l'hora de realitzar tasques més complexes.

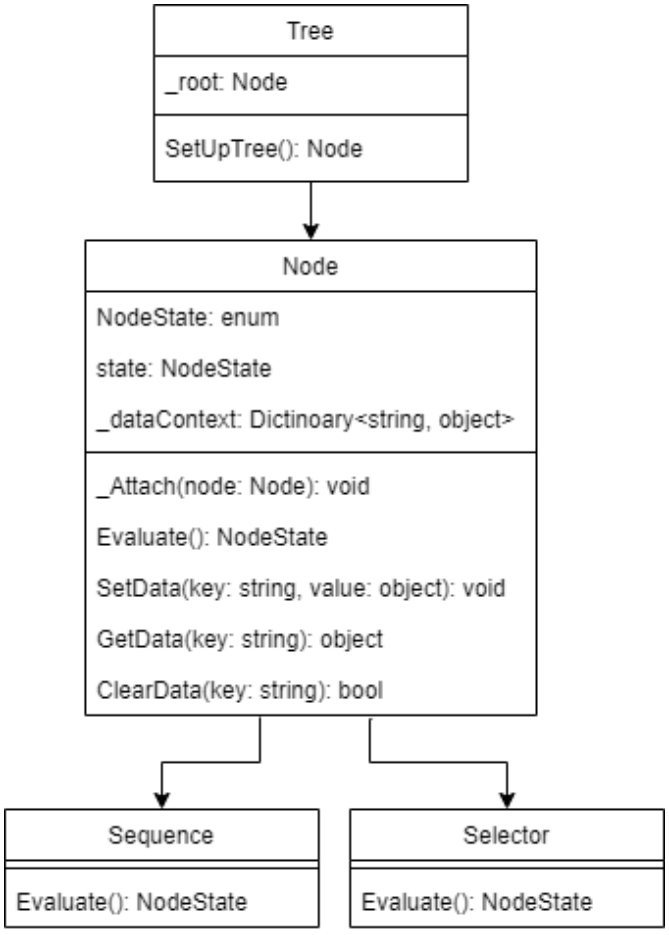


Figura 37: Diagrama de classes del Behavior Tree

- Tree

Aquesta és la classe abstracta a partir de la qual es crea tot l'arbre.

- **Atributs**
  - **_root:** El node arrel del arbre
- **Mètodes**
  - **SetUpTree():** Mètode abstracte. Construeix el Behavior Tree.

- Node

La classe Node té la funció de gestionar les fulles del arbre.

- **Atributs**
  - **NodeState:** Enum que conté els valors: RUNNING, SUCCESS, FAILURE
  - **state:** Indica si un Node està en execució, ha sigut un èxit o ha fallat
  - **dataContext:** Diccionari on es guarden tots els objectes que farà servir per realitzar les tasques.
- **Mètodes**
  - **_Attach(Node node):** Afegeix el node rebut com a paràmetre com a fill del node actual.
  - **Evaluate():** Mètode virtual. Avalua el node per determinar en quin estat es troba.
  - **GetData(key string):** Retorna l'objecte del diccionari relacionat amb la clau passada com a paràmetre.
  - **ClearData(key string):** Si existeix un objecte relacionat amb "key" al diccionari, s'elimina i retorna cert, fals altrament.

- Sequence

Aquesta classe avalua una llista de nodes de manera que tallarà l'avaluació de la llista si es troba un node fallat o en execució.

- Selector

Aquesta classe avalua una llista de nodes de manera que només tallarà l'avaluació de la llista si es troba un node amb èxit o en execució.

8.5.3 Enemics

Tot i ser un subgrup del total de classes, el diagrama resultant segueix sent massa extens. Per facilitar la compressió, s'ha tronat a dividir en grups diferents.

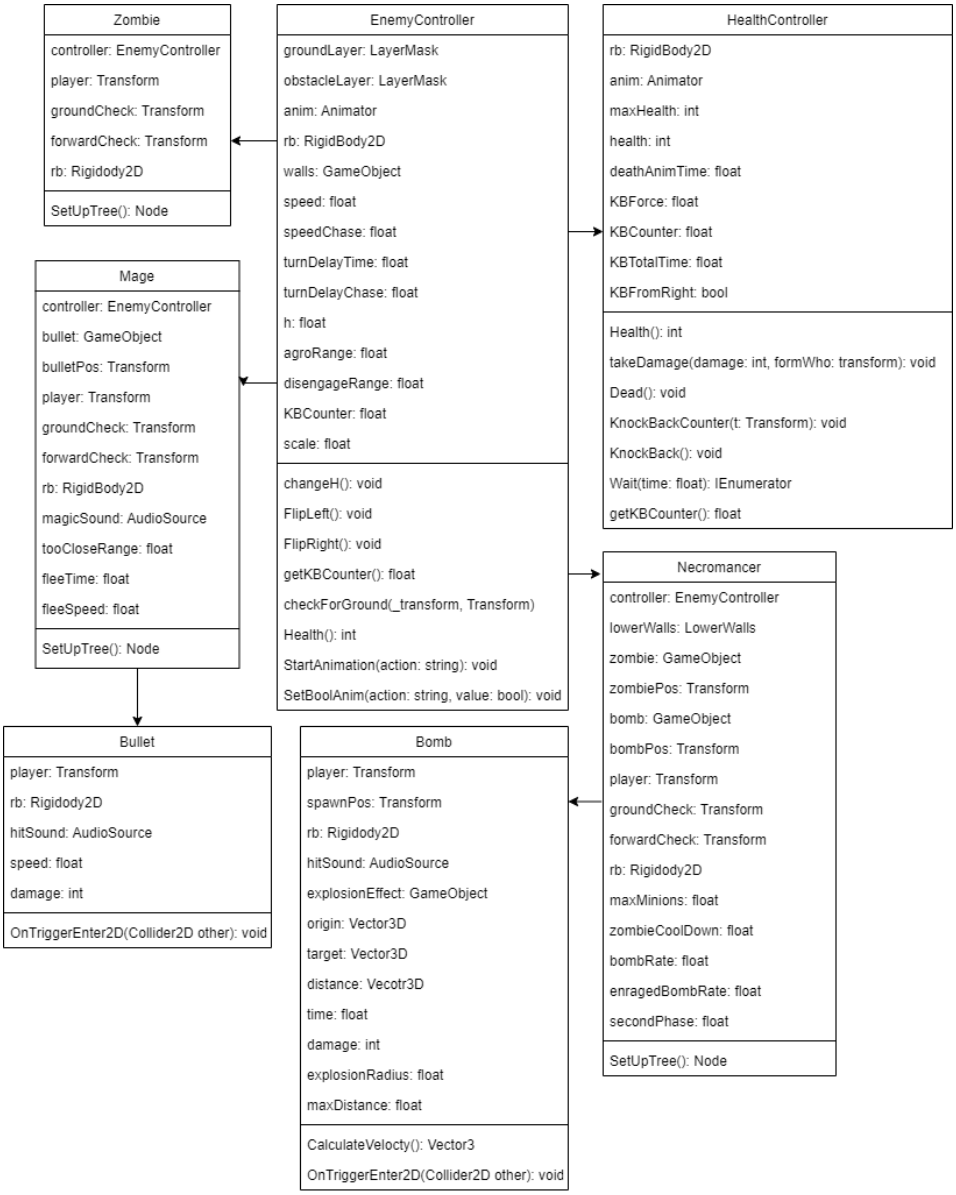


Figura 38: Diagrama de classes dels enemics

A demés de les classes que es poden observar a la Figura 38, també cal parlar de les tasques i comprovacions que s'utilitzen per estructurar les IAs de cada tipus d'enemic. Cal dir que aquests tipus de classes són compartides per tots els enemics i, per tant, només es parlarà d'elles la primera vegada que apareixin.

Per representar l'arbre creat a cada enemic, s'utilitzarà un diagrama on □ seran selectors i ◇ seran seqüencies.

8.5.3.1 Zombie

En aquest subgrup es parlarà de la classe Zombie i de les tasques i comprovacions que aquesta utilitza.

- Zombie

- **Atributs**

  - **controller:** Component que fa referència a la classe EnemyController.

  - **player:** Posició del personatge principal.

  - **rb:** Component RigidBody de l'enemic. S'utilitza per moure al zombi mitjançant l'aplicació de vectors de força.

  - **groundCheck:** Objecte situat als peus del zombi per detectar si està tocant a terra.

  - **forwardCheck:** Objecte situat davant del zombi per detectar si xocarà contra algun obstacle.

- **Mètodes**

  - **SetUpTree:** Construeix el Behavior Tree. Veure Figura 38.

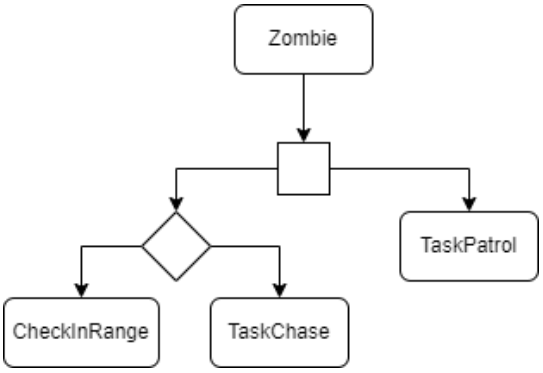


Figura 38: Diagrama del Behavior Tree de Zombie

- TaskPatrol

- **Atributs**

  - **_controller:** Component que fa referència a la classe EnemyController.

  - **_rb:** Component RigidBody que s'utilitza per moure a l'enemic mitjançant l'aplicació de vectors de força.

  - **_groundCheck:** Objecte situat als peus de l'enemic per detectar si està tocant a terra.

  - **_forwardCheck:** Objecte situat davant de l'enemic per detectar si xocarà contra algun obstacle.

  - **_turnDelay:** El temps que triga en donar la volta.

- **Mètodes**

  - **Evaluate():** S'executa la tasca de patrullar evitant obstacles. Sempre retorna l'estat SUCCESS.

- CheckInRange

- **Atributs**

  - **_controller:** Component que fa referència a la classe EnemyController.

  - **_player:** Posició del personatge principal.

  - **_transform:** Posició de l'enemic.

- **Mètodes**

  - **Evaluate():** Retorna estat SUCCESS si el jugador es troba dins del seu rang, retorna FAILURE altrament

- TaskChase

- **Atributs**

  - **_controller:** Component que fa referència a la classe EnemyController.

  - **_rb:** Component RigidBody de l'enemic. S'utilitza per moure al zombi mitjançant l'aplicació de vectors de força.

  - **_groundCheck:** Objecte situat als peus del zombi per detectar si està tocant a terra.

  - **_forwardCheck:** Objecte situat davant del zombi per detectar si xocarà contra algun obstacle.

  - **_turnDelay:** Comptador pel temps que triga en donar la volta.

  - **_delayTotal:** Temps total que triga en donar la volta.

- **Mètodes**

  - **Evaluate():** S'executa la tasca de perseguir al jugador. En cas que el jugador surti del rang d'atenció o que el zombi sigui derrotat retornarà FAILURE, altrament retornarà RUNNING.

8.5.3.2 Mage

Aquesta classe no només té relacionades les seves tasques i comprovacions, també té relació amb la classe Bullet.

- Mage

- **Atributs**

  - **controller:** Component que fa referència a la classe EnemyController.

- **bullet:** GameObject que fe referència al prefab d'un projectil

- **bulletPos:** Posició des d'on es llença el projectil

- **player:** Objecte que fa referència a la posició del personatge principal.

- **rb:** Component RigidBody de l'enemic. S'utilitza per moure al zombi mitjançant l'aplicació de vectors de força.

- **groundCheck:** Objecte situat als peus del mag per detectar si està tocant a terra.

- **forwardCheck:** Objecte situat davant del mag per detectar si xocarà contra algun obstacle.

- **magicSound:** Component AudioSource per reproduir el so del llançament del míssil màgic.

- **tooCloseRange**: Rang al que el mag considera que el jugador està massa a prop.

- **fleeTime:** Temps que fuig abans de tornar a disparar.

- **fleeSpeed:** Velocitat a la que fuig.

- **Mètodes**

  - **SetUpTree():** Construeix el Behavior Tree. Veure Figura 39.

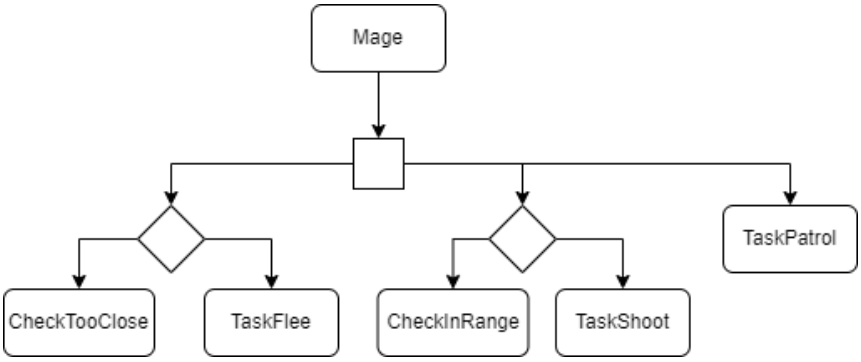


Figura 39: Diagrama del Behavior Tree de Mage

- Bullet

- **Atributs**

  - **player:** Posició del jugador.

  - **rb:** Component Rigidbody2D del projectil.

  - **hitSound:** Component AudioSource per reproduir el so del impacte del projectil en cas de que acerti.

  - **speed:** Velocitat del projectil.

  - **damage:** Dany del projectil.

- **Mètodes**

  - **OnTriggerEnter2D(Collider2D other):** Si entra en contacte amb un objecte amb el Tag igual a “Player”, reprodueix el so d’impacte i danya al personatge principal.

- CheckTooClose

- **Atributs**

  - **_controller:** Component que fa referència a la classe EnemyController.

  - **_transform:** Posició del mag.

  - **_forwardCheck:** Objecte situat davant del mag per detectar si xocarà contra algun obstacle.

  - **tooCloseRange:** Rang al que el mag considera que el jugador està massa a prop.

  - **timer:** Comptador que controla quanta estona es passa fugint

  - **totalTime:** Temps que fuig abans de tornar a disparar.

- **Mètodes**

  - **Evaluate():** Comprova si el jugador suficientment a prop com per fugir. En cas de que no pugui fugir a causa d'algun obstacle , no trobi el jugador dins del rang o es passi del temps assignat per fugir, retornarà FAILURE, en cas contrari retornarà SUCCESS.

- TaskFlee

- **Atributs**

  - **_controller:** Component que fa referència a la classe EnemyController.

  - **_rb:** Component RigidBody de l'enemic. S'utilitza per moure al zombi mitjançant l'aplicació de vectors de força.

  - **_groundCheck:** Objecte situat als peus del zombi per detectar si està tocant a terra.

  - **_forwardCheck:** Objecte situat davant del zombi per detectar si xocarà contra algun obstacle.

  - **_transform:** Posició del mag

  - **h:** Sentit cap a on encara el mag.

  - **fleeSpeed:** Velocitat a la que el mag fuig.

- **Mètodes**

  - **Evaluate():** Realitza la tasca de llençar el projectil màgic a la posició del jugador. En cas que li donin un cop, el jugador es situï fora de rang o es trobi amb algun obstacle retornarà l'estat FAILURE, altrament retornarà l'estat RUNNING.

- TaskShoot

- **Atributs**

  - **_controller:** Component que fa referència a la classe EnemyController.

  - **_transform:** Posició de l'enemic.

  - **_bullet:** Prefab del projectil a llençar.

  - **_bulletPos:** Posició des d'on es llençarà el projectil.

  - **_magicSound:** Component AudioSource per reproduir el so del l'acció de disparar.

  - **timer:** Comptador que controla la freqüència de dispar.

  - **tooClose:** Rang al que el mag considera que el jugador està massa a prop.

- **Mètodes**

  - **Evaluate():** Realitza la tasca de llençar el projectil després d'un temps de càrrega. En cas de que el jugador surti fora de rang retornarà FAILURE, en cas de que el temps de càrrega no s'hagi completat retornarà RUNNING i en cas de que cap de les dos situacions anteriors succeeixin, retornarà SUCCESS.

8.5.3.3 Necromancer

Al igual que en el cas del mag, el nigromant té una classe addicional relacionada a ell anomenada Bomb.

- Necromancer

- **Atributs**

  - **controller:** Component que fa referència a la classe EnemyController.

  - **lowerWalls:** Component que fa referència a la classe LowerWalls i s'utilitza per tancar l'habitació del nigromant quan el jugador hi entra.

  - **zombie:** Prefab del zombi.

  - **zombiePos:** Posició des d'on s'invocaran al zombi.

  - **bomb:** Prefab de la bomba.

  - **bombos:** Posició des d'on es llençaran les bombes.

  - **player:** Objecte que fa referència a la posició del personatge principal.

  - **rb:** Component RigidBody de l'enemic. S'utilitza per moure al nigromant mitjançant l'aplicació de vectors de força.

  - **groundCheck:** Objecte situat als peus del nigromant per detectar si està tocant a terra.

  - **forwardCheck:** Objecte situat davant del nigromant per detectar si xocarà contra algun obstacle.

  - **maxMinions:** Nombre màxim de zombis que el nigromant pot tenir invocats en tot moment.

  - **zombieCooldown:** Freqüència en la que s'invoquen els zombis.

  - **bombRate:** Freqüència en la que es llencen les bombes.

- **enragedBombRate:** Freqüència en la que es llencen les bombes quan el nigromant entra en la segona fase de la lluita.

- **secondPhase:** Punts de vida en els que el nigromant entra en la segona fase.

- **Mètodes**

  - **SetUpTree():** Construeix el Behavior Tree. Veure Figura 40.

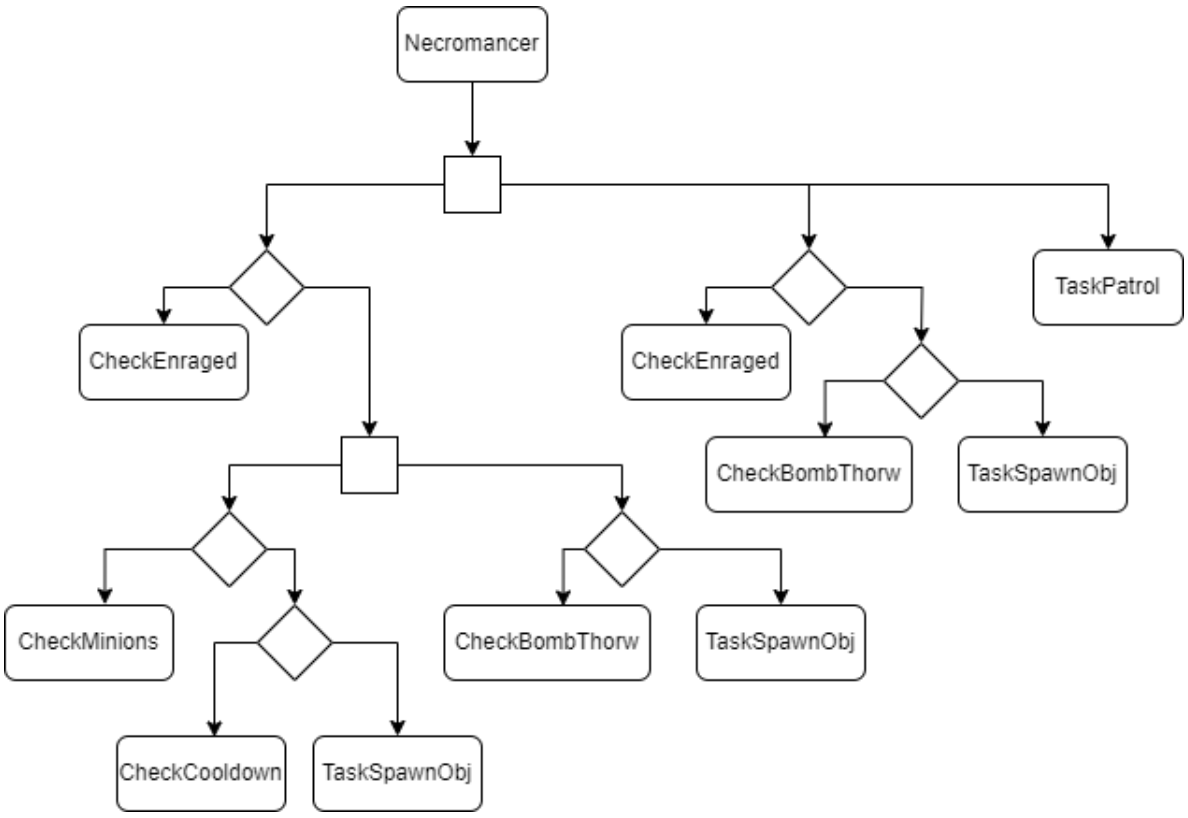


Figura 40: Diagrama del Behavior Tree de Necromancer

- Bomb

- **Atributs**

  - **player:** Posició del personatge principal.

  - **rb:** Component RigidBody2D de la bomba.

  - **spawnPos:** Posició des d'on es llençarà la bomba.

- **hitSound:** Component AudioSource per reproduir el so de la bomba explotant.

- **explosionEffect:** Prefab de l'animació de la bomba explotant.

- **origin:** Vector3 on es guarda la posició des d'on es llença la bomba.

- **target:** Vector3 on es guarda la posició del jugador.

- **distance:** Vector3 on es guarda la distància des de origin fins a target o la distància màxima en cas de que target estigui fora de rang.

- **time:** Temps que passa la bomba a l'aire.

- **damage:** Dany de les bombes.

- **explosionRadius:** Rang de l'explosió de la bomba.

- **maxDistànce:** La distància màxima que poden viatjar la bomba.

- **Mètodes**

  - **CalculateVelocity():** Calcula la velocitat en la que ha de viatjar la bomba per arribar a la posició target en un temps determinat.

- CheckEnraged

- **Atributs**

  - **_controller:** Component que fa referència a la classe EnemyController.

  - **_enrageThreshold:** Punts de vida a partir dels quals el nigromant entra a la segona fase.

  - **_enraged:** Indica si el nigromant ha entrat a la segona fase.

- **Mètodes**

  - **Evaluate():** Si se li ha passat com a paràmetre enraged com a cert, aleshores retornarà SUCCESS si es troba en la segona fase. Altrament, si se li ha passat com a paràmetre enraged com a fals, retornarà SUCCESS si no es troba en la segona fase.

- CheckMinions

- **Atributs**

  - **_maxMinions:** Nombre màxim de zombis,

  - **numMinions:** Nombre de zombis invocats que encara segueixen vius.

  - **minions:** Llista de GameObjects existents dins l'escena amb el Tag igual a "Enemy".

- **Mètodes**

  - **Evaluate():** Retornarà SUCCESS si pot invocar un zombi, altrament retornarà FAILURE.

- CheckCooldown

- **Atributs**

  - **_timer:** Comptador que controla el temps de cooldown.

  - **_totalTime:** Temps que ha de passar abans de poder invocar un altre zombi.

- **Mètodes**

  - **Evaluate():** Retornarà SUCCESS si pot invocar un zombi, altrament retornarà FAILURE.

- CheckBombThrow

- **Atributs**

  - **_transform:** Posició del nigromant

  - **_player:** Posició del personatge principal

  - **_timer:** Comptador que controla el temps de cooldown.

  - **_totalTime:** Temps que ha de passar abans de poder llençar una altra bomba.

  - **_range:** Rang en el que el nigromant pot llençar una bomba.

- **Mètodes**

  - **Evaluate():** Retornarà SUCCESS si el nigromant pot llençar una bomba, altrament retornarà FAILURE.

- TaskSpawnObject

Aquesta classe pot invocar zombis o bombes depenent del objecte que se li passi per paràmetre.

- **Atributs**

  - **_obj:** Prefab del objecte a generar.

  - **_objPos:** Posició des on generar l'objecte.

- **Mètodes**

  - **Evaluate():** Realitza la tasca de generar l'objecte passat per paràmetre. Sempre retornarà SUCCESS.

Per últim, la classe que tots els enemics comparteixen és la que gestiona la col·lisió amb el jugador i com aquesta col·lisió danya al personatge principal.

- EnemyDamage

- **Atributs**

  - **Player:** Posició del jugador.

  - **attackSound:** Component AudioSource per reproduir el so de la co·lisió.

  - **damage:** El dany que un enemic fa al jugador al xocar.

  - **timer:** Comptador de temps.

  - **coolDown:** Temps que ha de transcorre abans de poder tornar a fer mal al jugador.

8.5.4 Generació del mapa

Aquest subgrup conté les classes que s'encarreguen de generar cada nivell de manera que sempre hi hagi com a mínim un camí fins al final i sigui possible trobar-se zones que es poden desbloquejar una vegada s'obtingui la millora corresponent.

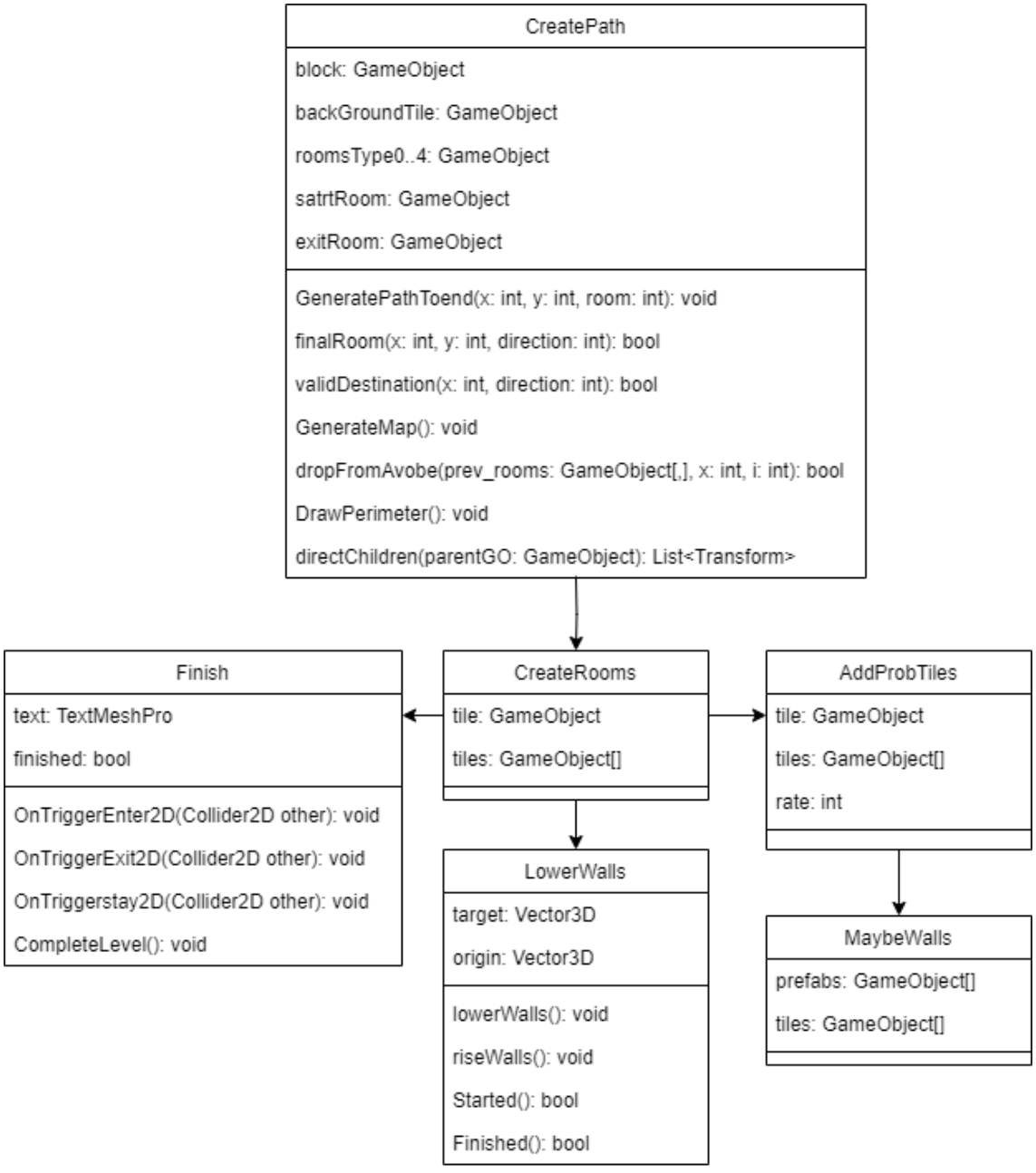


Figura 41: Diagrama de classes de la generació de mapa

- CreatePath

Aquesta classe crea una matriu aleatòria segons l'alçada i llargada que entrem com paràmetres. Cada casella conté una numero i cada numero indica els moviments a fer per viatjar de principi a fi: 1 i 2 són moviment cap a l'esquerra; 3 i 4, moviment cap a la dreta ; 5, moviment cap a baix i 0 vol dir que per aquella casella el camí no passa. Cada número és generat aleatòriament i s'augmenta en cas de que no sigui possible continuar.

Una vegada s'ha creat un camí aleatori amb aquests valors, es recorre la matriu de dalt a baix i, segons el valor que es trobi en cada casella, generarà una sala d'un tipus o d'un altre. Per exemple: si es troba una casella amb el valor 5, generarà una habitació de tipus 2, ja que totes les prefabs d'aquest tipus garanteixen una entrada per la dreta, una per l'esquerra i una per a baix.

D'aquesta manera, forçant el tipus d'habitació generada, obtenim un nivell aleatori amb, com a mínim, una ruta fins a la victòria assegurada.

- **Atributs**

  - **block:** Prefabs de blocks utilitzats per dibuixar el perímetre del mapa.

  - **backGroundTile:** Prefab del block utilitzat per dibuixar el backgorund.

  - **roomsType0..4:** Conjunt de Prefabs de grups d'habitacions de diferents tipus. Tot i que es diferencien per quines sortides tenen garantides , totes les habitacions tenen la possibilitat de tenir qualsevol sortida:

    - **Type0:** Només garanteixen una sortida cap a l'esquerra o dreta
    - **Type1:** Garanteixen sortides cap a la dreta i esquerra
    - **Type2:** Garanteixen sortides cap a la dreta, esquerra i a baix
    - **Type3:** Garanteix sortides cap a la dreta, esquerra i a dalt.
    - **Type4:** Garanteix sortides cap a tots els costats

  - **startRoom:** Prefab del conjunt de possibles habitacions d'entrada al nivell.

  - **exitRoom:** Prefab del conjunt de possibles habitacions de sortida al nivell.

  - **lenght:** Llargada de la matriu.

  - **height:** Alçada de la matriu.

- **roomDimensions:** La llargada i alçada d'una habitació.

- **level:** Matriu on es guarda el camí

- **rnd:** Generador de nombres aleatoris.

- **Mètodes**

  - **GeneratePathToEnd(int x, int y, int room):** Omple la matriu level amb valors de 0 a 5 per crear un camí des de la sala d'entrada fins la de sortida.

  - **finalRoom(int x, int direction):** Si s'intenta moure's cap a baix quan es troba en la planta més baixa, es considerarà aquella habitació la de sortida.

  - **validDestination(int x, int y, int direction):** Retorna cert si la direcció en la que es vol moure és vàlida, fals altrament.

  - **GenerateMap():** Seguint els valors de la matriu level, es genera el nivell amb les prefabs de les habitacions.

  - **dropFromAvobe(GameObject[,] prev_rooms, int x, int y):** Retorna cert si a sobre de l'habitació actual hi ha una amb una entrada per a baix, fals altrament.

  - **DrawPerimeter():** Dibuixa el perímetre del nivell amb el prefab de block.

  - **directChildren(GameObject parentGO):** Retorna una llista amb els fills directes de parentGO.

- CreateRooms

La manera en que generem habitacions amb sortides garantides és amb els blocs estàtics i els dinàmics.

En aquesta classe es generen els blocs estàtics (els garantits) d'una habitació .

- **Atributs**

  - **tile:** Prefab del bloc que es vol generar.

  - **tiles:** Llista de posicions on generar els blocs estàtics.

- AddProbTiles

En aquesta classe es generen els blocs dinàmics (els possibles) d'un habitació de manera aleatòria.

- **Atributs**
  - **tile:** Prefab del bloc que es vol generar.
  - **tiles:** Llista de posicions on generar els blocs estàtics (els garantits).
  - **rate:** La freqüència en que apareixeran blocs aleatoris.

- MaybeWalls

En aquesta classe es generen els blocs les possibles parets que es pot trobar el jugador i que no podrà superar fins que no obtingui el dash.

- **Atributs**
  - **prefabs:** Prefab del bloc que es vol generar.
  - **tiles:** Llista de posicions on generar les parets de forma aleatòria.

- LowerWalls

Aquesta classe detecta quan el jugador entra en una habitació amb un "Boss" i el tanca dins d'aquella habitació baixant les parets dels costats fins que derroti a l'enemic.

- **Atributs**
  - **origin:** Posició original de les parets baixades
  - **target:** Posició a la qual es mouran les parets per tal de tancar el jugador.
- **Mètodes**
  - **lowerWalls():** Baixa les parets.

- **riseWalls():** Puja les parets.

- **Started():** Detecta si el jugador ha entrat a l'habitació.

- **Finished():** Detecta si el jugador ha derrotat al "Boss".

- **areLowered():** Indica si les parets estan baixades.

- Finish

Aquesta classe mostra al jugador quina tecla clicar per passar de nivell una vegada davant la porta de sortida i crida a la classe SceneLoader per passar d'escena .

- **Atributs**

  - **text:** Text que es mostrarà per sobre de la porta una vegada el jugador s'hi apropi.

  - **finifshed:** Inidica si el jugador ha completat el nivell.

- **Mètodes**

  - **CompleteLevel():** Canvia l'estat de finished i, en cas que que es trobés en l'últim nivell, torna al menú principal.

8.5.5 Interfícies d'usuari

Les classes d'aquest subgrup gestionen les funcionalitats dels menús a demés de les transicions entre escenes.

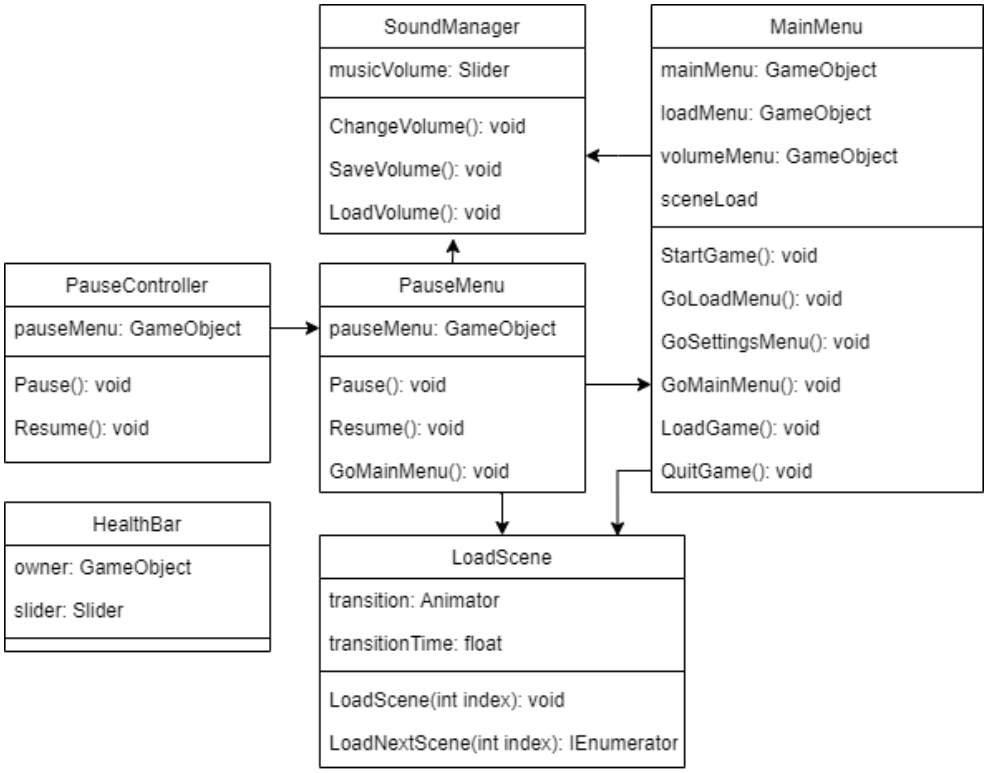


Figura 41: Diagrama de classes de les interfícies d'usuari

- MainMenu

El menú principal, el d'inici de joc i el de so estan en la mateixa escena. Aquesta classe conté les funcions per navegar entre els tres menús.

- **Atributs**

  - **mainMenu:** GameObject del menú principal.

  - **loadMenu:** GameObject del menú d'inici de joc.

  - **volumeMenu:** GameObject del menú de so.

- **Mètodes**
  - **StarttGame():** Inicia una partida nova.
  - **GoLoadMenu():** Navega al menú d'inici de joc.
  - **GoSettingsMenu():** Navega al menú de so.
  - **GoMainMenu():** Navega al menú principal.
  - **LoadGame():** Carrega l'última partida guardada.
  - **QuitGame():** Tanca l'aplicació.

- SoundManger

- **Atributs**
  - **musicVolume:** Slider amb el valor de so desitjat.
- **Mètodes**
  - **ChangeVolume():** Modifica el volum general del videojoc.
  - **SaveVolume():** Guarda el volum que el jugador selecciona.
  - **LoadVolume():** Carrega el volum guardat a la partida.

- PauseController

Aquesta classe detecta quan el jugador clica la tecla de pausa, pausa la partida mitjançant Time.timeScale i mostra el menú de pausa

- **Atributs**
  - **pauseMenu:** GameObject del menú de pausa.

- **Mètodes**
  - **Pause():** Pausa la partida i mostra el menú de pausa.
  - **Resume():** Deixa de mostrar el menú de pausa i reprèn la partida.

- PauseMenu

- **Atributs**
  - **pauseMenu:** GameObject del menú de pausa.
- **Mètodes**
  - **Pause():** Pausa la partida.
  - **Resume():** Reprèn la partida.
  - **GoMainMenu():** Guarda el progrés del jugador i torna al menú principal

- SceneLoader

- **Atributs**
  - **transition:** Component Animator de l'animació de transició entre escenes.
  - **transitionTime:** Temps que dura la transició.
- **Mètodes**
  - **LoadScene(int index):** Carrega l'escena amb l'índex passat com paràmetre.
  - **LoadNextScene(int index):** Reprodueix l'animació de transició, espera el temps necessari i carrega l'escena amb l'índex passat com paràmetre.

- SceneLoader

Aquesta classe controla la barra de vida del jugador i dels “Boss” monstres.

- **Atributs**
    - **owner:** GameObject de qui pertany la barra de vida.
    - **slider:** Quantitat de punts de salut dins de la barra.

8.5.6 Diagrama de classe final

Una vegada s'ajunten tots els subgrups, el diagrama final és el següent. Veure Figura 42

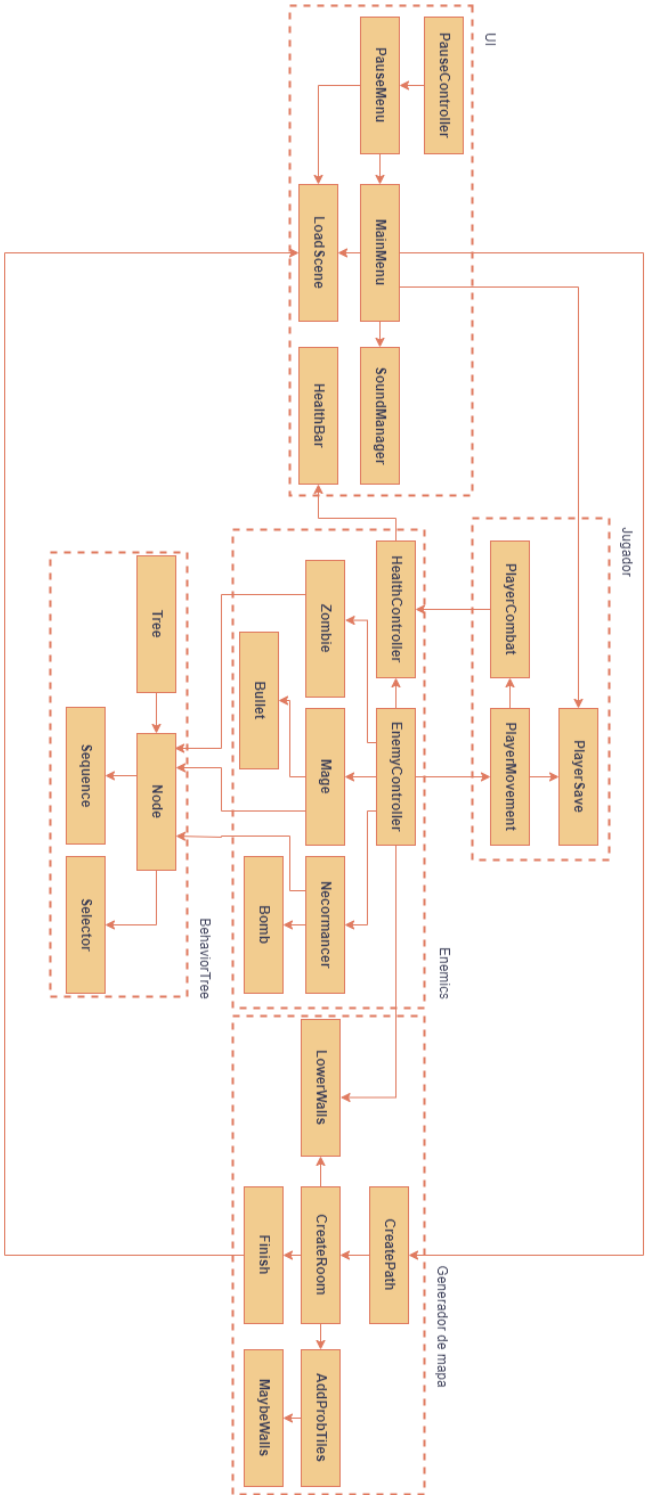


Figura 42: Diagrama de classe del videojoc

9. Implementació i proves

9.1 Menús

Amb l'excepció del menú de pausa, tots els menús estan implementats en la mateixa escena i, per navegar entre menús, s'activen i desactiven els GameObjects que contenen el conjunt de botons da cada un. Aquesta decisió es va prendre a causa de que la imatge de fons de tots els menús és igual i, a demés, la música de menú es reiniciava si es canviava d'escena.

9.1.1 Menú principal

El menú principal conté tres botons que estan lligats a les següents funcions:

- "Play" -> GoLoadMenu()
- "Volume" -> GoSettingsMenu()
- "Quit" -> Quit()

```
public void GoLoadMenu()
{
    loadMenu.SetActive(true);
    mainMenu.SetActive(false);

    if(PlayerPrefs.HasKey("dash"))
        loadMenu.transform.GetChild(1).GetComponent<Button>().interactable
           = true;
    else
        loadMenu.transform.GetChild(1).GetComponent<Button>().interactable
           = false;
}

public void GoSettingsMenu()
{
    volumeMenu.SetActive(true);
    mainMenu.SetActive(false);
}

public void QuitGame()
{
    Application.Quit();
}
```

En el cas del botó per navegar al menú d'inici de joc, es busca si existeixen dades guardades i, en cas de que no les trobi, es torna inactiu.

9.1.2 Menú d'inici de joc

El menú d'inici de joc conté tres botons que estan lligats a les següents funcions:

- "New Game" -> StartGame()
- "Load" -> LoadGame()
- "Back" -> GoMainMenu()

```
public void StartGame()
{
    PlayerPrefs.DeleteKey("dash");
    PlayerPrefs.DeleteKey("claws");
    PlayerPrefs.DeleteKey("boots");
    PlayerPrefs.DeleteKey("health");
    sceneLoad.LoadScene(1);
}

public void LoadGame()
{
    PlayerPrefs.DeleteKey("health");
    sceneLoad.LoadScene(1);
}

    public void GoMainMenu()
{
    mainMenu.SetActive(true);
    volumeMenu.SetActive(false);
    loadMenu.SetActive(false);
}
```

Com es pot veure, al començar una nova partida, s'esborren les dades guardades abans de carregar l'escena del primer nivell. En el cas de continuar, s'esborren la vida guardada per evitar començar la partida sense la vida al màxim o sense punts de vida, depenent de com es va sortir de la partida.

9.1.2 Menú de so

El menú de so conté un control lliscant per modificar el volum del joc i un botó per tornar al menú principal. Mentre que el botó comparteix la funció *GoMainMenu* de l'apartat anterior, el lliscador està lligat a la següent classe:

```
public class SoundManager : MonoBehaviour
{

    [SerializeField] Slider musicVolume;

    void Start()
    {
        if(!PlayerPrefs.HasKey("musicVolume"))
        {
            PlayerPrefs.SetFloat("musicVolume", 1);
            LoadVolume();
        }
        else
        {
            LoadVolume();
        }
    }

    public void ChangeVolume()
    {
        AudioListener.volume = musicVolume.value;
        SaveVolume();
    }

    private void SaveVolume()
    {
        PlayerPrefs.SetFloat("musicVolume", musicVolume.value);
    }

    private void LoadVolume()
    {
        musicVolume.value = PlayerPrefs.GetFloat("musicVolume");
    }
}
```

El lliscador està lligat a la funció *ChangeVolume*, la qual canvia el volum general del videojoc i crida la funció *SaveVolume* per guardar aquest canvi.

9.1.3 Menú de pausa

Aquesta interfície utilitza dues classes. Una, per les funcionalitats del menú i l'altra, per controlar l'aturada i la represa de la partida a partir dels inputs del teclat.

La classe *PauseController* inicialitza el menú de pausa com a inactiu i només l'activa si el jugador clica la tecla Escape i una vegada ha pausat el joc mitjançant la variable global Time.timeScale.

La funció *Resume* és pel cas que el jugador surti del menú de pausa prement de nou la tecla Escape.

```
public class PauseController : MonoBehaviour
{

    [SerializeField] private GameObject pauseMenu;

    void Start()
    {
        pauseMenu.SetActive(false);
    }

    void Update()
    {
        if(!pauseMenu.activeSelf &&
           Input.GetKeyDown(KeyCode.Escape))
        {
            Pause();
        }
        else if(pauseMenu.activeSelf &&
           Input.GetKeyDown(KeyCode.Escape)) Resume();
    }

    public void Pause()
    {
        Time.timeScale = 0f;
        pauseMenu.SetActive(true);
    }

    public void Resume()
    {
        pauseMenu.SetActive(false);
        Time.timeScale = 1f;
    }
}
```

La classe *PauseMenu* conté les funcions lligades a les diferents funcionalitats del menú de pausa, amb excepció de la modificació del volum del joc, la qual utilitza la classe *SoundManger* vista en el punt 9.1.2. En aquest menú, l'opció de tornar al menú principal ve acompanyada amb la funcionalitat de guardar les dades del personatge.

```
public class PauseMenu : MonoBehaviour
{
    [SerializeField] private GameObject pauseMenu;

    public void Resume()
    {
        pauseMenu.SetActive(false);
        Time.timeScale = 1f;
    }

    public void GoMainMenu()
    {
        Time.timeScale = 1f;
        GameObject.Find("DataSave").GetComponent<PlayerSave>().SavePlayer();
        GameObject.Find("SceneLoader").GetComponent<SceneLoader>().LoadScene(0);
    }
}
```

9.2 Personatge Principal

Aquest capítol es pot dividir en tres apartats: Moviment, combat i gestió dades del personatge.

9.2.1 Moviment

La majoria d'operacions que controlen el moviment del jugador és realitzen dins de les funcions *Update* i *FixedUpdatede* la classe *PlayerMovement*, ja que depenen dels inputs del jugador. En tot moment es comprova que el jugador no estigui en un estat en el que no té permès controlar el moviment del personatge, com en mig d'un dash, quan està sortint del nivell o quan està en estat de "Knock Back". A demés, aquesta classe també s'encarrega de, en cas de que en tingui, carregar les dades guardades del personatge al inicialitzar-se.

```
void Start()
{
    GameObject.Find("DataSave").GetComponent<PlayerSave>().LoadPlayer();
}

void Update()
{

    if(isDashing ||
       GameObject.Find("ExitDoor").GetComponent<Finish>().finished)
       return;

    if(Input.GetButtonDown("Jump") && (isGrounded() &&
       !wallClimbing) || (!isGrounded() && wallClimbing))
    {
        jumpSound.Play();
        rb.velocity = new Vector2(rb.velocity.x, jumpingPower);
        anim.SetTrigger("Jump");
    }

    horizontal = Input.GetAxisRaw("Horizontal");
    if(climbClaws) vertical = Input.GetAxisRaw("Vertical");


    if(facingRight && horizontal < 0f || !facingRight && horizontal
       > 0f)
    {
        Flip();
    }

    if(Input.GetKeyDown(KeyCode.LeftShift) && dashActive && canDash
       && horizontal != 0)
    {
        anim.SetTrigger("Dash");
        StartCoroutine(Dash());
    }

    anim.SetFloat("Speed", Mathf.Abs(horizontal));
    anim.SetFloat("AirSpeed", rb.velocity.y);
    anim.SetBool("Grounded", isGrounded());

}
```

```
private void FixedUpdate()
{
    if(isDashing) return;

    if(transform.GetComponent<HealthController>().KBCounter <= 0)
    {
        rb.velocity = new Vector2(horizontal * speed, rb.velocity.y);
        if(climbClaws && wallClimbing) rb.velocity = new
           Vector2(rb.velocity.x, vertical * climbSpeed);
    }
}
```

També informa els paràmetres de l'Animator dels valors de la velocitat horitzontal i vertical, a demés d'informar si el personatge està tocant a terra. La funció *isGrounded* també s'utilitza per evitar que el jugador pugui saltar infinitament al aire. També es detecta en quina direcció es mou el personatge per girar-lo quan cal.

```
private bool isGrounded()
{
    return Physics2D.OverlapCircle(groundCheck.position, 0.2f,
       groundLayer);
}

private void Flip()
{
    facingRight = !facingRight;
    Vector3 localScale = transform.localScale;
    localScale.x *= -1f;
    transform.localScale = localScale;
}
```

En aquesta classe també es realitza l'execució del dash i es detecta quan el jugador entra en contacte amb una paret escalable.

```
private void OnCollisionEnter2D(Collision2D other)
{
    if(other.gameObject.tag.Equals("ClimbWall") && climbClaws) wallClimbing = true;
}

private void OnCollisionExit2D(Collision2D other)
{
    if(other.gameObject.tag.Equals("ClimbWall")) wallClimbing = false;
}

private IEnumerator Dash()
{
    canDash = false;
    isDashing = true;
    float originalGravity = rb.gravityScale;
    rb.gravityScale = 0f;

    Physics2D.IgnoreLayerCollision(7, 8, true);
    Physics2D.IgnoreLayerCollision(7, 10, true);
    rb.velocity = new Vector2(horizontal * dashingSpeed, 0);
    tr.emitting = true;

    yield return new WaitForSeconds(dashingTime);

    Physics2D.IgnoreLayerCollision(7, 8, false);
    Physics2D.IgnoreLayerCollision(7, 10, false);
    tr.emitting = false;
    isDashing = false;
    rb.gravityScale = originalGravity;

    yield return new WaitForSeconds(dashingCoolDown);
    canDash = true;
}
```

La funció *Dash* dona la propulsió establerta al jugador, desactiva la gravetat del personatge, activa el component Trail Renderer per indicar que s'està executant el dash i fa que el personatge es torni intangible amb certes layers durant uns moments.

9.2.2 Combat

La classe *PlayerCombat* controla els atacs del personatge principal, mentre que la classe *HealthController* controla la vida tant com del personatge com dels enemics.

```
public class PlayerCombat : MonoBehaviour
{

    [SerializeField] private Transform attackController;
    [SerializeField] private Animator anim;
    [SerializeField] private AudioSource attackAudio;
    [SerializeField] private AudioSource missAudio;

    public float attackRadius;
    public int attackDamage;
    public float cooldown;
    private float timer = 0;

    void Update()
    {

        if(Input.GetButtonDown("Fire1") && timer >= cooldown)
        {
            timer = 0;
            Attack();
        }

        timer += Time.deltaTime;

    }

    public void Attack()
    {
        anim.SetTrigger("Attack");
        Collider2D[] objects =
            Physics2D.OverlapCircleAll(attackController.position, attackRadius);
        bool enemy = false;

        foreach(Collider2D collision in objects)
        {
            if(collision.CompareTag("Enemy"))
            {
                enemy = true;
                collision.transform.GetComponent<HealthController>().takeDamage(attackDamage,
                    transform);
            }
        }

        if(enemy) attackAudio.Play();
        else missAudio.Play();
    }
}
```

La classe *PlayerCombat* detecta quan el jugador clicar el botó esquerre del ratolí i, en cas de que hagi passat una certa quantitat de temps des de l'últim atac executat, crida a la funció *Attack*, la qual s'encarrega de detectar si hi havia algun enemic dins del rang de l'atac i, en cas positiu, els envia el dany realitzat i des d'on ha vingut l'atac. A demés, s'encarrega de reproduir el so d'un cop en cas de que hagi detectat un enemic, o el so de fallada en cas contrari. A demés de controlar els punts de vida, la classe *HealthController* també gestiona carregar la vida guardada a l'inici, l'empeny que genera un atac i el cas de ser derrotat.

```
public class HealthController : MonoBehaviour
{
    public int maxHealth = 100;
    public int health = 100;
    public float deathAnimTime;

    public float KBForce;
    public float KBCounter;
    public float KBTotalTime;
    public bool KBFromRight;

    [SerializeField] private Rigidbody2D rb;
    public Animator anim;

    void Start()
    {
        if(!PlayerPrefs.HasKey("health"))
        {
            health = maxHealth;
            PlayerPrefs.SetInt("health", health);
        }
    }

    void FixedUpdate(){
        if(KBCounter > 0) KnockBack();
    }

    public void Dead()
    {
        rb.constraints = RigidbodyConstraints2D.FreezePosition;
        GetComponent<BoxCollider2D>().enabled = false;
        if(gameObject.name == "Necromancer")
        {
            GameObject.Find("Player").GetComponent<PlayerMovement>().dashActive =
                true;
        }

        if(transform.CompareTag("Player"))
        {
            GameObject.Find("DataSave").GetComponent<PlayerSave>().SavePlayer();
            PlayerPrefs.DeleteKey("health");
            StartCoroutine(Wait(deathAnimTime));
            GameObject.Find("SceneLoader").GetComponent<SceneLoader>().LoadScene(0);
        }
        else Destroy(this.gameObject, deathAnimTime);
    }

    public int Health()
    {
        return health;
    }
```

```
    public void takeDamage(int damage, Transform fromWho)
    {
        if(!GameObject.Find("ExitDoor").GetComponent<Finish>().finished)
        {
            health -= damage;
            if(health > 0)
            {
                anim.SetTrigger("Hurt");
                KnockBackCounter(fromWho);
            }
            else {
                anim.SetTrigger("Death");
                Dead();
            }
        }

    }

    private void KnockBackCounter(Transform t)
    {
        KBCounter = KBTotalTime;
        if(t.position.x <= transform.position.x) KBFromRight = true;
        else KBFromRight = false;
    }

    private void KnockBack()
    {
        if(KBFromRight)
        {
            rb.velocity = new Vector2(KBForce, rb.velocity.y);
        }
        else
        {
            rb.velocity = new Vector2(-KBForce, rb.velocity.y);
        }

        KBCounter -= Time.deltaTime;
    }

    private IEnumerator Wait(float time)
    {
        yield return new WaitForSeconds(time);
    }

    public float getKBCounter()
    {
        return KBCounter;
    }

}
```

En cas de ser derrotat, es procedirà a congelar la posició per evitar que l'individu es mogui i desactiva el component BoxCollider2D per evitar interaccions mentre es reprodueix l'animació de mort. En cas de que l'individu derrotat sigui el nigromant, s'atorgarà al personatge principal la millora del dash. En cas de que l'individu sigui el personatge, enlloc de destruir l'objecte es guardaran les dades esborrant la vida per a que no es comenci la següent escena amb zero punts de vida i crida la funció per carregar l'escena del menú principal.

Al parlar de controlar els punts de vida, també podem parlar de la implementació que permet al jugador visualitzar els canvis realitzats als punts de vida. La classe *HealthBar* controla la funcionalitat de la barra de salut.

```
public class HealthBarManager : MonoBehaviour
{
    [SerializeField] private GameObject owner;
    public Slider slider;

    void Start()
    {
        slider.maxValue = owner.GetComponent<HealthController>().maxHealth;
    }


    void Update()
    {
        slider.value = owner.GetComponent<HealthController>().health;
    }
}
```

9.2.3 Gestió de dades del personatge

Per realitzar la tasca de guardar i carregar les dades del personatge principal s'utilitza la classe *PlayerSave*.

```
public class PlayerSave : MonoBehaviour
{

    public void SavePlayer()
    {
        GameObject player = GameObject.Find("Player");
        PlayerMovement move = player.GetComponent<PlayerMovement>();
        PlayerPrefs.SetInt("dash", move.dashActive?1:0);
        PlayerPrefs.SetInt("claws", move.climbClaws?1:0);
        PlayerPrefs.SetInt("boots", move.bootsOn?1:0);
        PlayerPrefs.SetInt("health",
            player.GetComponent<HealthController>().Health());
    }

    public void LoadPlayer()
    {
        GameObject player = GameObject.Find("Player");
        if(PlayerPrefs.HasKey("dash"))
        {
            player.GetComponent<PlayerMovement>().dashActive =
                PlayerPrefs.GetInt("dash")==1?true:false;
            player.GetComponent<PlayerMovement>().climbClaws =
                PlayerPrefs.GetInt("claws")==1?true:false;
            player.GetComponent<PlayerMovement>().bootsOn =
                PlayerPrefs.GetInt("boots")==1?true:false;
            player.GetComponent<HealthController>().health =
                PlayerPrefs.GetInt("health");
        }

    }
}
```

Aquesta classe utilitza *PlayerPrefs* per guardar les dades entre escenes, ja que a l'utilitzar una prefab del personatge principal, qualsevol canvi realitzat durant l'escena s'esborraria al passar a una altra escena.

Al ser un videojoc amb elements Roguelike, el personatge només reté les millores al guardar la partida. Es guarda la vida per a que el personatge no comenci cada nivell amb la vida plena sense tenir en compte el mal que ha pres en nivells anteriors.

9.3 Generació del mapa

9.3.1 Generador de camí

Abans de poder generar els blocs i les habitacions que composaran el nivell, cal generar un camí garantit de principi a fi. La classe *CreatePath* s'encarrega d'aquesta tasca.Les funcions principals són *GeneratePathToEnd* i *GenerateMap*.

```
private void GeneratePathToEnd(int x, int y, int room)
{
    int movement;
    if(x==0 && y==0) movement = rnd.Next(1,6);
    else movement = rnd.Next(1,7);

    while(!validDestination(x, y, movement))
    {
        movement++;
    }

    if(finalRoom(x, movement))
    {
        level[x,y] = 4;
    }
    else
    {
        level[x,y] = room;

        switch(movement)
        {
            case 1:
            case 2:
            case 3:
                GeneratePathToEnd(x, y-1, 1);
                break;
            case 4:
            case 5:
                GeneratePathToEnd(x, y+1, 1);
                break;
            default:
                level[x,y] = 2;
                GeneratePathToEnd(x+1, y, 3);
                break;
        }

    }
}
```

GenreatPathToEnd és una funció recursiva que utilitza un algorisme inspirat en el que s'utilitza al videojoc d'estil Roguelike anomenat *Spelunky*. Es basa en separar el mapa en cel·les (cada cel·la és una habitació) i tractar-lo com una matriu amb valors de 0 a 4. Cada número indica la direcció (esquerra, dreta o a baix) de la següent cel·la que forma part del camí. En el cas d'aquesta funció, els valor 1 indica esquerra o dreta, el valor 2 indica a baix, el valor 3 és per al caso que la cel·la superior tingui el valor 2 i el valor 4 indica que s'ha arribat al fi del camí.

Amb l'excepció de la cel·la d'inici, es genera un valor per indicar a quina direcció es vol moure que, en cas de no ser vàlid, s'augmenta:

- 1,2,3: esquerra
- 4,5: dreta
- 6: a baix

Aquest mètode es pot utilitzar perquè el jugador sempre pot anar cap a baix. En el cas de que es trobi en la planta més baixa la cel·la d'anar cap a baix s'assignarà com l'habitació de fi de camí.

Per comprovar que el moviment a realitzar és vàlid s'utilitzen un parell de funcions:

```
private bool finalRoom(int x, int direction)
{
    return direction > 5 && x == height-1;
}


private bool validDestination(int x, int y, int direction)
{
    switch(direction)
    {
        case 1:
        case 2:
        case 3:
            return y > 0 && level[x, y-1] == 0;
        case 4:
        case 5:
            return y < length-1 && level[x, y+1] == 0;
        default:
            return true;
    }
}
```

Una vegada creat el camí, es passarà a assignar el tipus d'habitació que li correspondrà a cada cel·la. Per tal de generar el nivell de manera que es respecti el camí creat però doni més opcions per explorar al jugador, s'han creat set tipus de conjunts d'habitacions prefabricades. Cada habitació conté blocs estàtics, que sempre es generaran amb l'habitació, i blocs dinàmics, que tenen la probabilitat de ser generats amb l'habitació. Distribuint aquests dos tipus de blocs es pot controlar quines sortides (esquerra, dreta, a baix i a dalt) estaran obertes de forma garantida i quins obstacles tenen la possibilitat d'aparèixer.

Primer es parlarà de l'assignació de tipus d'habitació per cada cel·la. Cada tipus es diferencia per les sortides que té garantides:

- Tipus 0: Cap sortida.
- Tipus 1: Sortides a l'esquerra i dreta.
- Tipus 2: Sortides a l'esquerra, dreta i a baix.
- Tipus 3: Sortides a l'esquerra, dreta i a dalt.
- Tipus 4: Totes les sortides.

Amb excepció de les habitacions de tipus d'entrada, des d'on el jugador comença el nivell i sempre estan situades a la cantonada esquerra superior del mapa, i les de tipus sortida, on estan situades les portes per passar al següent nivell.

```
private void GenerateMap()
{
    DrawPerimeter();
    GameObject rooms;
    GameObject[,] prev_rooms = new GameObject[height, length];
    int r;

    for(int i = 0; i < height; i++)
    {
        for(int j = 0; j < length; j++)
        {

            switch(level[i,j])
            {
                case 0:
                    if(dropFromAvobe(prev_rooms, i, j)) r = 3;
                    else if(j == length-1) r = 1;
                    else if(j == 0 || (j > 0 && prev_rooms[i,j-1].name ==
                        "NoRoom"))  r = 2;
                    else if(prev_rooms[i,j-1].name == "RoomType0_2")
                    {
                        int[] room = {0,7};
                        r = room[rnd.Next(0,2)];
                    }
                    else if(i < height-1 && level[i+1,j] == 0)
                    {
                        int[] room = {0,0,0,0,0,0,5,6,4,4,4,4,4};
                        r = room[rnd.Next(0,14)];
                    }
                    else
                    {
                        int[] room = {0,0,0,0,0,5,6};
                        r = room[rnd.Next(0,7)];
                    }
                    rooms = roomsType0;
                    break;

                case 2:
                    if(dropFromAvobe(prev_rooms, i, j))
                    {
                        r = rnd.Next(0,3);
                        rooms = roomsType4;
                    }
                    else
                    {
                        r = rnd.Next(0,4);
                        rooms = roomsType2;
                    }

                    break;
                case 3:
                    r = rnd.Next(0,3);
```

```
                    rooms = roomsType3;

                    break;
                case 4:
                    if(dropFromAvobe(prev_rooms, i, j)) r = 1;
                    else r = 0;
                    rooms = exitRoom;
                    break;
                default:
                    if(dropFromAvobe(prev_rooms, i, j)) r = 5;
                    else if(i == height-1) r = rnd.Next(0,4);
                    else
                    {
                        int[] choices = {0,1,2,3,4,4,4};
                        r = choices[rnd.Next(0,7)];
                    }
                    rooms = roomsType1;

                    break;
            }
            if(i == 0 && j == 0)
            {
                r = 0;
                rooms = startRoom;
            }


            Transform[] directChildren =  (from directChild in rooms.
                                GetComponentsInChildren<Transform>()
                                where directChild.transform.parent ==
                                    rooms.transform
                                select directChild).ToArray();

            prev_rooms[i,j] = directChildren[r].gameObject;

            Instantiate(directChildren[r].gameObject, new
                Vector3(j*roomDimensions.x, -i*roomDimensions.y, 0),
                Quaternion.identity);
        }
    }
}
```

A les cel·les amb valor 0 se'ls assigna les habitacions de tipus 0, amb casos on es pot forçar quin prefab d'habitació de tipus 0 depenent de la posició i de les habitacions anteriors. Per exemple ,es pot forçar una habitació amb paret a un costat o l'altre si es troba a la primera o última columna de la matriu.

En cas de que l'habitació de dalt tingui sortida cap a baix, es força que la prefab de l'habitació tingui una sortida cap a dalt per poder connectar-les i així crear camins falsos o alternatius.

Les cel·les amb valor 2 forcen l'assignació d'habitacions del tipus 2 o 4, depenent de si l'habitació de sobre té sortida cap a baix o no.

Les cel·les amb valor 3 forcen l'assignació d'habitacions del tipus 3. Aquest tipus d'habitació també es pot forçar en el cas que es trobi a l'última fila de la matriu i l'habitació superior tingui sortida cap a baix.

En cas de que es trobi en la cel·la d'inici o de fi, se'ls assignarà l'habitació del tipus corresponent.

En el cas de que la cel·la contingui el valor 1 o no es compleixin les condicions per l'assignació de certes habitacions, s'assignarà un habitació del tipus 1.

Per facilitar la comprovació d'habitacions anteriors o superior, es crea una segona matriu i es crida la funció *dropFromAvobe*. Finalment, es selecciona un prefab d'habitació de dins del conjunt assignat de forma aleatòria utilitzant la funció *directChildren*.

```
private bool dropFromAvobe(GameObject[,] prev_rooms, int x, int y)
{
    return (x > 0 && (prev_rooms[x-1,y].name == "RoomType0_5" ||
            prev_rooms[x-1,y].name == "RoomType1_5" ||
            prev_rooms[x-1,y].transform.parent.gameObject.name == "RoomType4" ||
            prev_rooms[x-1,y].transform.parent.gameObject.name == "RoomType2"));
}

public List<Transform> directChildren(GameObject parentGO)
{
    List<Transform> directChildren = new List<Transform>();
    foreach(Transform go in parentGO.transform){  // This will only find direct
        children
            Transform c = go.gameObject.GetComponent<Transform>();
            directChildren.Add(c);
    }
    return directChildren;
}
```

9.3.2 Generador d'habitacions

La classe *CateRoom* s'encarrega de generar tots els blocs estàtics d'una habitació quan s'instancia el prefab.

```
public class CreateRooms : MonoBehaviour
{

    public GameObject tile;

    public GameObject[] tiles;

    void Start()
    {
        tiles = new GameObject[transform.childCount];
        for (int i = 0; i < transform.childCount; i++)
        {
            tiles[i] = transform.GetChild(i).gameObject;
        }

        foreach(GameObject obj in tiles)
        {
            Instantiate(tile, obj.transform.position, Quaternion.identity);
        }
    }

}
```

La classe *AddProbTiles* s'encarrega de generar de forma aleatòria els diferents obstacles dinàmics d'una habitació amb la probabilitat desitjada quan s'instancia el prefab.

```
public class AddProbTiles : MonoBehaviour
{
    public GameObject tile;
    public int rate = 5;

    public GameObject[] tiles;

    void Start()
    {
        tiles = new GameObject[transform.childCount];

        for (int i = 0; i < transform.childCount; i++)
        {
            tiles[i] = transform.GetChild(i).gameObject;
        }

        System.Random rnd = new System.Random();

        foreach(GameObject obj in tiles)
        {
            int chance = rnd.Next(0,10);
            if(chance < rate)
            {
                Instantiate(tile, obj.transform.position, Quaternion.identity);
            }
        }
    }
}
```

La classe *MaybeWalls* s'encarrega de generar de manera aleatòria les parets especials formades per blocs dinàmics d'una habitació quan s'instancia la prefab.

```
public class MaybeWalls : MonoBehaviour
{

    public GameObject[] prefabs;

    private GameObject[] tiles;

    void Start()
    {
        Transform[] directChildren = (from directChild in transform.
                                      GetComponentsInChildren<Transform>()
                                      where directChild.transform.parent == transform
                                      select directChild).ToArray();

        System.Random rnd = new System.Random();

        int x = rnd.Next(0,2);

        tiles = new GameObject[directChildren[x].childCount];

        for (int i = 0; i < directChildren[x].childCount; i++)
        {
            tiles[i] = directChildren[x].GetChild(i).gameObject;
        }
        foreach(GameObject obj in tiles)
        {
            Instantiate(prefabs[x], obj.transform.position, Quaternion.identity);
        }
    }
}
```

En el cas de que es tracti del nivell del "boss", hi hauran parets amagades a cada sortida de l'habitació on es lluita per evitar que el jugador continuï abans de derrotar l'enemic. Per controlar aquestes parets s'utilitza la classe *LowerWalls*, que detecta quan el jugador entra en l'habitació i quan derrota al "boss" per pujar i baixar aquestes parets.

```
public class LowerWalls : MonoBehaviour
{

    private Vector3 target;
    private Vector3 origin;

    void Start()
    {
        origin = transform.position;
        target = new Vector3 (transform.position.x, transform.position.y - 3,
            transform.position.z);

    }

    // Update is called once per frame
    void Update()
    {
        if(Started()) lowerWalls();
        if(Finished()) riseWalls();
    }
```

```
    private void lowerWalls()
    {
        transform.position = target;
    }

    private void riseWalls()
    {
        transform.position = origin;
    }

    public bool Started()
    {
        return GameObject.Find("Player").transform.position.x > -10;
    }

    public bool Finished()
    {
        return GameObject.Find("Necromancer") == null;
    }

    public bool areLowered()
    {
        return transform.position == target;
    }
}
```

Per últim, la classe *Finish* controla la interacció entre el jugador i la porta de sortida del nivell. Mostra un text per indicar al jugador com passar al següent nivell i crida la funció per carregar la següent escena a demés de guardar les dades del personatge principal.

```
public class Finish : MonoBehaviour
{
    public TextMeshPro text;
    public bool finished;

    void Start()
    {
        text.text = "";
        finished = false;
    }

    private void OnTriggerEnter2D(Collider2D other)
    {
        if(other.gameObject.tag == "Player")
        {
            text.text = "[E]";
        }
    }
    private void OnTriggerExit2D(Collider2D other)
    {
        if(other.gameObject.tag == "Player")
        {
            text.text = "";
        }
    }

    private void OnTriggerStay2D(Collider2D other)
    {
        if(other.gameObject.tag == "Player" && Input.GetKey(KeyCode.E))
        {
            GameObject.Find("DataSave").GetComponent<PlayerSave>().SavePlayer();
            CompleteLevel();
        }
    }
```

```
    private void CompleteLevel()
    {
        if(SceneManager.GetActiveScene().buildIndex == 3)
        {
            if(PlayerPrefs.HasKey("health"))PlayerPrefs.DeleteKey("health");
            GameObject.Find("SceneLoader").GetComponent<SceneLoader>().LoadScene(0);
        }
        else
            finished = true;
    }
}
```

9.4 Transició entre escenes

Per controlar la navegació entre escenes i controlar l'animació de transició entre elles, s'utilitza la classe *SceneLoader*, la qual s'ha cridat en varies funcions anteriors.

```
public class SceneLoader : MonoBehaviour
{
    public Animator transition;
    public float transitionTime;

    void FixedUpdate()
    {
        if(SceneManager.GetActiveScene().buildIndex > 0 &&
            GameObject.Find("ExitDoor").GetComponent<Finish>().finished)
        {
            StartCoroutine(LoadNextScene(SceneManager.GetActiveScene().buildIndex
                + 1));
        }
    }

    public void LoadScene(int index)
    {
        StartCoroutine(LoadNextScene(index));
    }

    IEnumerator LoadNextScene(int index)
    {
        transition.SetTrigger("Start");

        yield return new WaitForSeconds(transitionTime);

        SceneManager.LoadScene(index);
    }
}
```

La classe detecta si s'ha acabat el nivell actual i executa la transició de escena en cas positiu. També conta amb la funció publica *SceneLoader* per a que altres classes puguin utilitzar *SceneLoader*.

9.5 Behavior Tree

Per implementar la metodologia Behavior Tree s'ha creat les classes *Tree*, *Node, Selector* i *Sequence*. Per una part, la classe *Tree* és una classe abstracta on s'inicialitza l'arbre i es guarda el node arrel. Fa la funció de punt d'interacció entre les classe dels diferents tipus d'enemics i els arbres de comportament.

```
namespace BehaviorTree
{
    public abstract class Tree : MonoBehaviour
    {

        private Node _root = null;

        protected virtual void Start()
        {
            _root = SetUpTree();
        }

        private void Update()
        {
            if (_root != null)
                _root.Evaluate();
        }

        protected abstract Node SetUpTree();

    }
}
```

Per l'altra part, la classe *Node* gestiona els nodes de l'arbre i les derivades *Selector* i *Sequence* s'encarreguen d'avaluar els nodes o grups de nodes.

```
namespace BehaviorTree
{
    public enum NodeState
    {
        RUNNING,
        SUCCESS,
        FAILURE
    }

    public class Node
    {
        protected NodeState state;

        public Node parent;
        protected List<Node> children = new List<Node>();

        private Dictionary<string, object> _dataContext = new Dictionary<string,
            object>();

        public Node()
        {
            parent = null;
        }
        public Node(List<Node> children)
        {
            foreach (Node child in children)
                _Attach(child);
        }
```

```
        private void _Attach(Node node)
        {
            node.parent = this;
            children.Add(node);
        }

        public virtual NodeState Evaluate() => NodeState.FAILURE;

        public void SetData(string key, object value)
        {
            _dataContext[key] = value;
        }

        public object GetData(string key)
        {
            object value = null;
            if (_dataContext.TryGetValue(key, out value))
                return value;

            Node node = parent;
            while (node != null)
            {
                value = node.GetData(key);
                if (value != null)
                    return value;
                node = node.parent;
            }
            return null;
        }

        public bool ClearData(string key)
        {
            if (_dataContext.ContainsKey(key))
            {
                _dataContext.Remove(key);
                return true;
            }

            Node node = parent;
            while (node != null)
            {
                bool cleared = node.ClearData(key);
                if (cleared)
                    return true;
                node = node.parent;
            }
            return false;
        }
    }

}
```

Les funcions *_Attach*, *GetData*, *SetData* i *ClearData* afegeixen, consulten, modifiquen i esborren dades del diccionari d'objectes *_dataContext* respectivament. Això va bé per no haver de fer la mateixa comprovació varies vegades. Per exemple, una vegada el personatge entra al rang d'atenció de l'enemic, es guarda l'objecte Player al diccionari i si surt del rang, s'elimini el registre. D'aquesta manera, si es troba el registre relacionat amb l'objecte Player dins del diccionari, no caldrà comprovar de nou.

L'enum *NodeState* defineix els possibles estats d'un node.

La classe *Selector* avalua un grup de nodes de manera que només sortirà del procés si es troba un node en estat d'èxit o execució o arriba al final de la llista.

```
namespace BehaviorTree
{
    public class Selector : Node
    {
        public Selector() : base() { }
        public Selector(List<Node> children) : base(children) { }

        public override NodeState Evaluate()
        {
            foreach (Node node in children)
            {
                switch (node.Evaluate())
                {
                    case NodeState.FAILURE:
                        continue;
                    case NodeState.SUCCESS:
                        state = NodeState.SUCCESS;
                        return state;
                    case NodeState.RUNNING:
                        state = NodeState.RUNNING;
                        return state;
                    default:
                        continue;
                }
            }

            state = NodeState.FAILURE;
            return state;
        }
    }
}
```

La classe *Sequence* avalua un grup de nodes de manera que només sortirà del procés si es troba un node en estat de fallida o execució o arriba al final de la llista.

```
namespace BehaviorTree
{
    public class Sequence : Node
    {
        public Sequence() : base() { }
        public Sequence(List<Node> children) : base(children) { }

        public override NodeState Evaluate()
        {
            bool anyChildIsRunning = false;

            foreach (Node node in children)
            {
                switch (node.Evaluate())
                {
                    case NodeState.FAILURE:
                        state = NodeState.FAILURE;
                        return state;
                    case NodeState.SUCCESS:
                        continue;
                    case NodeState.RUNNING:
                        anyChildIsRunning = true;
                        continue;
                    default:
                        state = NodeState.SUCCESS;
                        return state;
                }
            }

            state = anyChildIsRunning ? NodeState.RUNNING : NodeState.SUCCESS;
            return state;
        }
    }
}
```

9.6 Enemics

Per facilitar la comprensió, aquest apartat es dividirà en diferents subapartats. La primera es parlarà sobra la implementació de les classes base de cada tipus de monstre per veure com construeixen cada arbre de comportament. A la segona part es parlarà de la implementació de les diferents tasques i comprovacions que comparteixen per poder definir el comportament de les IAs.

Com les classes dels enemics deriven de *Tree* i només implementen la funció *SetUpTree*, només es mostrarà la implementació d'aquesta funció da cada enemic.

9.6.1 Zombi

Els enemics de tipus zombi estan controlats per la classe *Zombie*.

```
protected override Node SetUpTree()
{
    player = GameObject.FindGameObjectWithTag("Player").transform;

    Node root = new Selector(new List<Node>
    {
        new Sequence(new List<Node>
        {
            new CheckInRange(transform, player, controller),
            new TaskChase(transform, rb, controller, 0),
        }),
        new TaskPatrol(groundCheck, forwardCheck, rb, controller),
    });
    return root;
}
```

Com es pot veure, l'arbre de comportament utilitza la comprovació *CheckInRange* i les tasques *TaskChase* i *TaskPatrol*.

El zombi exhibeix el següent comportament:

- En cas de que el jugador entri al seu rang d'atenció: el perseguirà
- Altrament: patrullarà la zona.

L'ordre dels d'aquests punts és important ja que el selector els avalua de dalt a baix i pararà quan es trobi el primer que retorni SUCCESS.

9.6.2 Mag

Els enemics de mag zombi estan controlats per la classe *Mage* i fan servir la classe *Bullet* per realitzar els atacs a distància.

```
protected override Node SetUpTree()
{
    player = GameObject.FindGameObjectWithTag("Player").transform;

    Node root = new Selector(new List<Node>
    {
        new Sequence(new List<Node>
        {
            new CheckTooClose(transform, controller, forwardCheck, tooCloseRange,
                fleeTime),
            new TaskFlee(transform, groundCheck, forwardCheck, rb, controller,
                fleeSpeed),
        }),
        new Sequence(new List<Node>
        {
            new CheckInRange(transform, player, controller),
            new TaskShoot(transform, bullet, bulletPos, controller, tooCloseRange,
                magicSound),
        }),

        new TaskPatrol(groundCheck, forwardCheck, rb, controller),
    });

    return root;
}
```

Com es pot veure, el seu arbre de comportament utilitza les comprovacions *CheckTooClose* i *CheckInRange* i les tasques *TaskFlee, TaskShoot* i *TaskPatrol*.

El mag exhibeix el següent comportament:

- En cas de que el jugador s'apropi massa: realitzarà la tasca de fugir.
- En cas de que el jugador entri al seu rang d'atac: dispararà un projectil màgic.
- Altrament: patrullarà

La classe *Bullet* controla el comportament del projectil màgic que el mag pot invocar.

```
public class Bullet : MonoBehaviour
{

    private GameObject player;

    public float speed = 20f;
    public int damage;
    private Rigidbody2D rb;
    [SerializeField] private AudioSource hitSound;

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        player = GameObject.FindGameObjectWithTag("Player");

        Vector3 direction = player.transform.position - transform.position;
        rb.velocity = new Vector2(direction.x, direction.y).normalized * speed;
    }
```

```
    void OnTriggerEnter2D(Collider2D other)
    {
        if(other.CompareTag("Player"))
        {
            hitSound.Play();
            player.GetComponent<HealthController>().takeDamage(damage, transform);
            Destroy(gameObject);
        }
    }
}
```

Calcula la trajectòria a la que es mourà al inicialitzar-se i detecta quan xoca contra el personatge principal per fer-li mal.

9.6.2 Nigromant

L'enemic de tipus Nigormant és controlat per la classe *Necromancer* i fa servir la classe *Bomb* per realitzar els seus atacs a distància.

```
protected override Node SetUpTree()
{

    Node root = new Selector(new List<Node>
    {

        new Sequence(new List<Node>
        {
            new CheckEnraged(controller, secondPhase, true),
            new Selector(new List<Node>
            {
                new Sequence(new List<Node>
                {
                    new CheckMinions(maxMinions),
                    new Sequence(new List<Node>
                    {
                        new CheckCooldown(zombieCooldown),
                        new TaskSpawnObj(zombie, zombiePos),
                    }),
                }),
                new Sequence(new List<Node>
                {
                    new CheckBombThrow(transform, player, enragedBombRate,
                        controller.agroRange),
                    new TaskSpawnObj(bomb, bombPos),
                }),
            }),
        }),

        new Sequence(new List<Node>
        {
            new CheckEnraged(controller, secondPhase, false),
            new Sequence(new List<Node>
            {
                new CheckBombThrow(transform, player, bombRate,
                    controller.agroRange),
                new TaskSpawnObj(bomb, bombPos),
            })
        }),

        new TaskPatrol(groundCheck, forwardCheck, rb, controller),
    });
    return root;

}
```

Com es pot veure, el seu arbre de comportament utilitza les comprovacions *CheckEnraged, CheckMinions, CheckCooldown* i *CheckBombThrow* i les tasques *TaskSpawnObj* i *TaskPatrol*.

El mag exhibeix el següent comportament:

- En cas que hagi entrat en la segona fase:
  - En cas que no es passi del nombre de minions màxim:
    - En cas de que hagi passat suficient temps des de l'última invocació: Invocarà un zombi.
  - Altrament:
    - En cas de que pugui llençar una bomba: llençarà bombes amb freqüència més ràpida
- Altrament:
  - En cas de que pugui llençar una bomba: llençarà bombes amb freqüència més lenta
- Sempre patrullarà

La classe *Bomb* controla el comportament de les bombes que el nigromant llença.

```
public class Bomb : MonoBehaviour
{

    private GameObject player;
    private GameObject spawnPos;

    [SerializeField] private AudioSource hitSound;
    [SerializeField] private GameObject explosionEffect;

    public float time = 7f;
    public int damage;
    public float explosionRadius = 0.5f;
    public float maxDistance;

    private Vector3 origin;
    private Vector3 target;
    private Vector3 distance;

    private Rigidbody2D rb;

    // Start is called before the first frame update
    void Start()
    {
        player = GameObject.FindGameObjectWithTag("Player");
        spawnPos = GameObject.FindGameObjectWithTag("BombSpawnPos");

        rb = GetComponent<Rigidbody2D>();

        origin = spawnPos.transform.position;
        target = player.transform.position;

        distance = target - origin;

        if(Mathf.Abs(distance.x) > maxDistance)
        {
            if(distance.x < 0) distance.x = -maxDistance;
            else distance.x = maxDistance;
        }

        Vector3 vo = CalculateVeliocity();
        rb.velocity = vo;
    }
```

```
    Vector3 CalculateVeliocity()
    {
        Vector3 distanceXz = distance;
        distanceXz.y = 0f;

        float sY = distance.y;
        float sXz = distanceXz.magnitude;

        float Vxz = sXz / time;
        float Vy = (sY / time) + (0.5f * Mathf.Abs(Physics.gravity.y) * time);

        Vector3 result = distanceXz.normalized;
        result *= Vxz;
        result.y = Vy;

        return result;
    }

    void OnTriggerEnter2D(Collider2D other)
    {
        if(!other.CompareTag("Enemy"))
        {
            Collider2D[] objects = Physics2D.OverlapCircleAll(transform.position,
                explosionRadius);

            foreach(Collider2D collision in objects)
            {
                hitSound.Play();
                if(collision.CompareTag("Player"))
                {
                    player.GetComponent<HealthController>().takeDamage(damage,
                        transform);
                }
            }

            rb.constraints = RigidbodyConstraints2D.FreezePosition;
            Instantiate(explosionEffect, transform.position, Quaternion.identity);
            Destroy(gameObject,0.5f);
        }
    }
}
```

Utilitza la funció *CalculateVelocity* per calcular la velocitat que s'ha d'aplicar per a que la bomba faci una paràbola des del punt d'origen fins al final, que pot ser la posició del personatge principal o el rang màxim en direcció al personatge. També detecta quan xoca contra el personatge o el mapa i fa dany en cas de que el jugador estigués dins del rang de l'explosió. A més reprodueix el prefab d'una animació per la explosió quan fa contacte.

9.6.3 Comprovacions

***CheckInRange*:** Si no troba l'objecte jugador al diccionari, comprova si la posició del jugador entrada per paràmetre està dins del rang d'atenció i, en cas positiu, la guardarà al diccionari i retorna SUCCESS, altrament retorna FAILURE. En cas de que trobi l'objecte del personatge principal al diccionari, no comprovarà res i retorna SUCCESS.

```
public class CheckInRange : Node
{
    private EnemyController _controller;
    private Transform _player;
    private Transform _transform;

    public CheckInRange(Transform transform, Transform player, EnemyController
        enemy)
    {
        _transform = transform;
        _player = player;
        _controller = enemy;
    }

    public override NodeState Evaluate()
    {
        Transform p = (Transform)GetData("player");
        if(p == null)
        {

            if(Vector2.Distance(_transform.position, _player.position) <=
                _controller.agroRange)
            {
                parent.parent.SetData("player", _player);

                state = NodeState.SUCCESS;
                return state;
            }

            state = NodeState.FAILURE;
            return state;
        }
        state = NodeState.SUCCESS;
        return state;
    }

}
```

***CheckTooClose*:** Si troba l'objecte jugador al diccionari, el personatge principal es troba dins del rang que es considera massa a prop i no s'ha acabat el temps que triga en fugir, es retorna SUCCESS. Altrament, o en cas que no pugui fugir a causa d'algun obstacle, retorna FAILURE.

```
public class CheckTooClose : Node
{
    private EnemyController _controller;
    private Transform _transform;
    private Transform _forwardCheck;

    private float tooCloseRange;
    private float timer = 0;
    private float totalTime;

    public CheckTooClose(Transform transform, EnemyController enemy, Transform
        forwardCheck, float range, float fleeTime)
    {
        _transform = transform;
        _forwardCheck = forwardCheck;
        _controller = enemy;
        tooCloseRange = range;
        totalTime = fleeTime;
    }

    public override NodeState Evaluate()
    {

        if(_controller.checkForGound(_forwardCheck))
        {
            state = NodeState.FAILURE;
            return state;
        }

        Transform p = (Transform)GetData("player");
        if(p == null)
        {
            state = NodeState.FAILURE;
            return state;
        }

        if(Vector2.Distance(_transform.position, p.position) <= tooCloseRange)
        {
            timer += Time.deltaTime;
            if(timer <= totalTime)
            {
                state = NodeState.SUCCESS;
                return state;
            }

            if(timer > totalTime+2) timer = 0;
            state = NodeState.FAILURE;
            return state;

        }

        timer = 0;
        state = NodeState.FAILURE;
        return state;
    }
}
```

***CheckMinions*:** Troba tots els objectes de l'escena amb el tag "Enemy" i, en cas que trobi un número menor al nombre màxim de minions, retorna SUCCESS. Altrament, retorna FAILURE.

```
public class CheckMinions : Node
{
    private float _maxMinions;
    public float numMinions = 0;
    private GameObject [] minions;

    public CheckMinions(float maxMinions)
    {
        _maxMinions = maxMinions;

    }

    public override NodeState Evaluate()
    {

        minions = GameObject.FindGameObjectsWithTag("Enemy");

        foreach(GameObject minion in minions)
        {
            numMinions++;
        }

        if(numMinions <= _maxMinions)
        {
            numMinions = 0;
            state = NodeState.SUCCESS;
            return state;
        }
        numMinions = 0;

        state = NodeState.FAILURE;
        return state;
    }
}
```

***CheckCooldown*:** Comprova si el temps des de l'última invocació d'un zombi és menor al temps de recàrrega de l'habilitat d'invocació i, en cas positiu, retorna SUCCESS. Altrament, retorna FAILURE.

```
public class CheckCooldown : Node
{
    private float _timer = 0;
    private float _totalTime;

    public CheckCooldown(float cooldown)
    {
        _totalTime = cooldown;
    }

    public override NodeState Evaluate()
    {
        if(_timer >= _totalTime)
        {
            _timer = 0;
            state = NodeState.SUCCESS;
            return state;
        }

        _timer += Time.deltaTime;
        state = NodeState.FAILURE;
        return state;
    }
}
```

CkeckBombThrow: Guarda la posició del jugador al diccionari en cas de que no la tingui i comprova si el jugador està dins del rang d'atenció. En cas positiu, i si el temps que ha passat des de l'última bomba llençada és menor al temps de recàrrega de les bombes, retorna SUCCESS. Altrament, retorna FAILURE.

```
public class CheckBombThrow : Node
{
    private Transform _transform;
    private Transform _player;
    private float _timer;
    private float _totalTime;
    private float _range;

    public CheckBombThrow(Transform transform, Transform player, float timer,
        float range)
    {
        _transform = transform;
        _player = player;
        _totalTime = timer;
        _range = range;
    }

    public override NodeState Evaluate()
    {
        Transform p = (Transform)GetData("player");
        if(p == null)
        {
            parent.parent.SetData("player", _player);
            _timer = 0;
        }

        if(_timer >= _totalTime && (Vector2.Distance(_transform.position,
            p.position) <= _range))
        {
            _timer = 0;
            state = NodeState.SUCCESS;
            return state;
        }
        _timer += Time.deltaTime;
        state = NodeState.FAILURE;
        return state;
    }
}
```

9.6.4 Tasques

***TaskPatrol*:** En cas que l'enemic no es trobi cap obstacle i no surti de la plataforma on es troba, camina cap endavant amb la velocitat entrada com paràmetre. Si no pot avançar i el temps passat és superior al temps de "delay" entrat, dona la volta. Sempre retorna SUCCESS, ja que al ser l'últim node del arbre, és la tasca que l'enemic realitzarà en cas de que no es compleixin les condicions per realitzar les altres tasques.

```
public class TaskPatrol : Node
{

    public EnemyController _controller;
    private Transform _groundCheck;
    private Transform _forwardCheck;
    private Rigidbody2D _rb;

    private float _turnDelay = 1;

    public TaskPatrol(Transform groundCheck, Transform forwardCheck, Rigidbody2D
        rb, EnemyController controller)
    {
        _groundCheck = groundCheck;
        _forwardCheck = forwardCheck;
        _rb = rb;
        _controller = controller;
    }

    public override NodeState Evaluate()
    {
        if(_controller.checkForGound(_groundCheck) &&
            !_controller.checkForGound(_forwardCheck))
        {
            _rb.velocity = new Vector2(_controller.speed * _controller.h,
                _rb.velocity.y);
        }
        else
        {
            if(_turnDelay <= 0)
            {
                _controller.changeH();
                _rb.velocity = new Vector2(_controller.speed * _controller.h,
                    _rb.velocity.y);
                _turnDelay = _controller.turnDelayTime;
            }
            _turnDelay -= Time.deltaTime;
        }

        state = NodeState.SUCCESS;
        return state;
    }
}
```

***TaskChase*:** L'enemic es dirigeix cap a la posició del jugador, la qual obté del diccionari, girant amb "delay" per seguir al personatge principal i xocar contra ell. S'utilitza el valor de direcció horitzontal "h" per comparar-lo amb la direcció a "EnemyController" i així poder crear un temps de "delay" constant. Mentre no sigui empès per un atac o el jugador surti de rang, retorna RUNNING, altrament, retorna FAILURE. En el cas de que el jugador surti de rang, també s'esborra el registre del jugador del diccionari per a que *CheckInRange* no es pensi que el jugador està dins del rang d'atenció.

```
public class TaskChase : Node
{

    private EnemyController _controller;
    private Transform _transform;
    private Rigidbody2D _rb;

    private float _turnDelay;
    private float _delayTotal = 0.4f;
    private float h;

    public TaskChase(Transform transform, Rigidbody2D rb,
       EnemyController controller, float turnDelay)
    {
        _transform = transform;
        _rb = rb;
        _controller = controller;
        _turnDelay = turnDelay;
    }

    public override NodeState Evaluate()
    {
        if(_controller.KBCounter > 0) {
            state = NodeState.FAILURE;
            return state;
        }

        Transform player = (Transform)GetData("player");

        if(Vector2.Distance(_transform.position, player.position) >
           _controller.disengageRange)
        {
            parent.parent.ClearData("player");
            state = NodeState.FAILURE;
            return state;
        }
        else if(_controller.Health() < 0)
        {
            _rb.velocity = new Vector2(0,_rb.velocity.y);
            state = NodeState.FAILURE;
            return state;
        }
        if(_transform.position.x <= player.position.x) h = 1;
        else h = -1;

        if(h != _controller.h)
        {
            if(_turnDelay <= 0)
            {
                _turnDelay = _delayTotal;
                _controller.changeH();
            }
            _turnDelay -= 0.01f;
        }

        _rb.velocity = new Vector2(_controller.speedChase *
           _controller.h, _rb.velocity.y);

        state = NodeState.RUNNING;
        return state;
    }
}
```

TaskFlee: Mentre l'enemic no sigui empès per un atac, el jugador no surti de rang i l'enemic no es trobi cap obstacle, l'enemic es mou per allunyar-se del jugador i retorna RUNNING. Altrament, retorna FAILURE.

```
public class TaskFlee : Node
{
    private EnemyController _controller;
    private Transform _transform;
    private Transform _groundCheck;
    private Transform _forwardCheck;
    private Rigidbody2D _rb;

    private float h;
    private float fleeSpeed;

    public TaskFlee(Transform transform, Transform groundCheck, Transform
        forwardCheck, Rigidbody2D rb, EnemyController enemy, float speed)
    {
        _transform = transform;
        _groundCheck = groundCheck;
        _forwardCheck = forwardCheck;
        _rb = rb;
        _controller = enemy;
        fleeSpeed = speed;
    }
    public override NodeState Evaluate()
    {

        if(_controller.KBCounter > 0) {
            state = NodeState.FAILURE;
            return state;
        }

        Transform player = (Transform)GetData("player");

        if(Vector2.Distance(_transform.position, player.position) >
            _controller.disengageRange)
        {
            parent.parent.ClearData("player");
            state = NodeState.FAILURE;
            return state;
        }
        if(_transform.position.x <= player.position.x) h = -1;
        else h = 1;

        if(h != _controller.h) _controller.changeH();
        if(_controller.checkForGound(_groundCheck) &&
            !_controller.checkForGound(_forwardCheck))
        {
            _rb.velocity = new Vector2(fleeSpeed * h, _rb.velocity.y);
            state = NodeState.RUNNING;
            return state;
        }

        state = NodeState.FAILURE;
        return state;
    }
}
```

***TaskShoot*:** Mentre la posició del jugador, obtinguda del diccionari, no estigui fora de rang, l'enemic no sigui empès per una atac i hagi passat suficient temps de càrrega, crea una instància del projectil i retorna SUCCESS. Mentre el temps passat sigui menor al temps de càrrega del atac, retorna RUNNING. Altrament, retorna FAILURE.

```
public class TaskShoot : Node
{
    private EnemyController _controller;
    private Transform _transform;
    private GameObject _bullet;
    private Transform _bulletPos;
    private AudioSource _magicSound;

    private float timer = 0;
    private float tooClose;

    public TaskShoot(Transform transform, GameObject bullet, Transform bulletPos,
        EnemyController enemy, float range, AudioSource magicSound)
    {
        _transform = transform;
        _bullet = bullet;
        _bulletPos = bulletPos;
        _controller = enemy;
        tooClose = range;
        _magicSound = magicSound;
    }

    public override NodeState Evaluate()
    {

        Transform player = (Transform)GetData("player");
        if(_transform.position.x > player.position.x) _controller.FlipLeft();
        else _controller.FlipRight();
        _controller.SetBoolAnim("ShootReady", true);

        if(Vector2.Distance(_transform.position, player.position) >
            _controller.disengageRange || _controller.getKBCounter() > 0)
        {
            if(_controller.getKBCounter() > 0) _controller.StartAnimation("Hurt");
            else _controller.SetBoolAnim("ShootReady", false);
            timer = 0;
            parent.parent.ClearData("player");
            state = NodeState.FAILURE;
            return state;
        }
        timer += 0.01f;
        if(timer > 3.5f)
        {
            _controller.StartAnimation("Shoot");
            _magicSound.Play();
            MonoBehaviour.Instantiate(_bullet, _bulletPos.position,
                _bulletPos.rotation);
            timer = 0;
            state = NodeState.SUCCESS;
            return state;
        }
        state = NodeState.RUNNING;
        return state;
    }
}
```

***TaskSpawnObj*:** Aquesta tasca crea una instància del GameObject entrat com a paràmetre a la posició desitjada. Sempre retorna SUCCESS.

```
public class TaskSpawnObj : Node
{
    private GameObject _obj;
    private Transform _objPos;

    public TaskSpawnObj(GameObject obj, Transform objPos)
    {
        _obj = obj;
        _objPos = objPos;
    }

    public override NodeState Evaluate()
    {
        GameObject newObj = MonoBehaviour.Instantiate(_obj, _objPos.position,
            Quaternion.identity);

        state = NodeState.SUCCESS;
        return state;
    }
}
```

10. Implantació i resultats

10.1 Legislació i normativa vigent

El projecte desenvolupat no presenta cap problema des del punt de vista legislatiu. No s'ha tingut en compte la llei orgànica de protecció de dades de caràcter personal (LOPD) ja que el sistema en cap moment tracta cap tipus de dades relatives a l'usuari.

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

Sobre la llei de serveis de la societat de la informació i comerç electrònic (LSSICE), el projecte no constitueix una activitat econòmica en cap dels sentits.

10.2 Captures de pantalla

En aquest apartat es mostraran captures de pantalla realitzades durant el funcionament del videojoc.

Figura 43: Menú principal

En la Figura 43 es pot observar el menú principal del joc. En aquest cas es tenia el ratolí sobre la opció "Volume" i per això surt ressaltada.

A continuació es mostren els menús que es poden accedir des del menú principal.

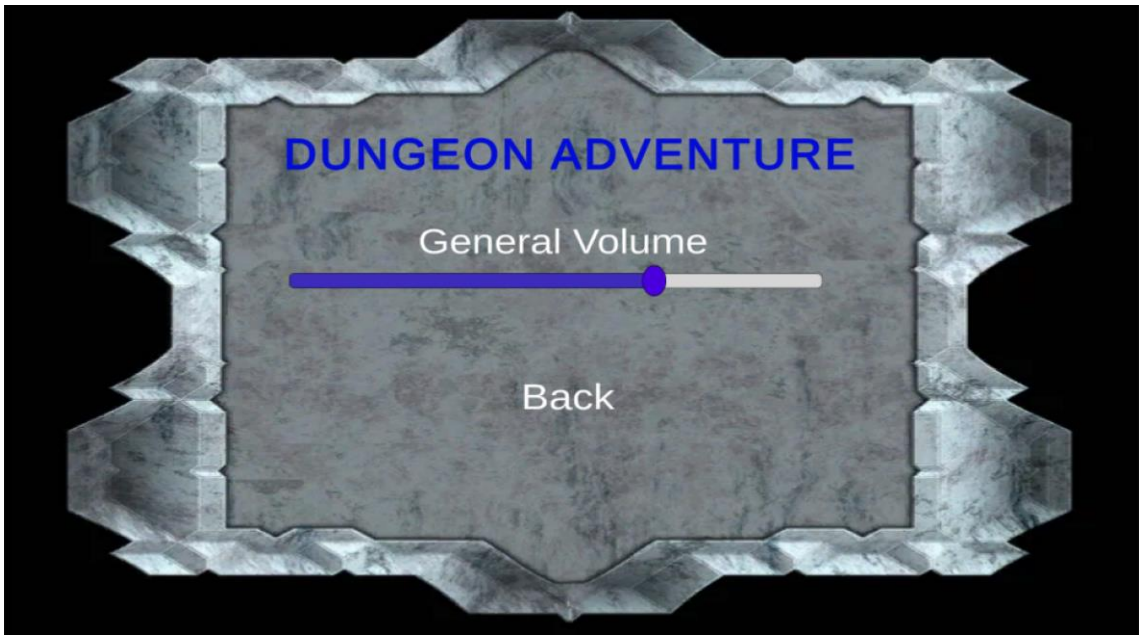


Figura 44: Menú de so

Figura 45: Menú d'inici de joc

A la Figura 45 es pot veure que, al no tenir cap partida guardada, l'opció "Load" no està disponible.

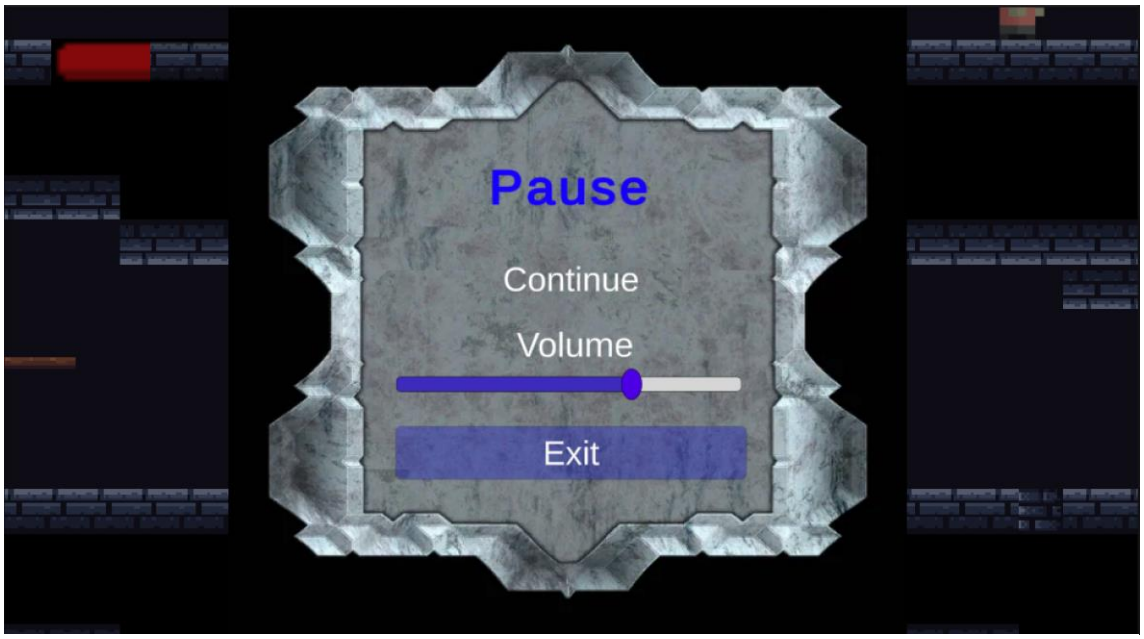


Figura 46: Menú de pausa

Fins aquí són totes les captures de pantalla dels menús. Tot seguit es mostren les captures realitzades durant una partida.


Figura 47: Sala inicial

El personatge apareix al costat una porta tancada, simbolitzant que no té cap altra opció que continuar cap al fons del calabós. Veure Figura 47.

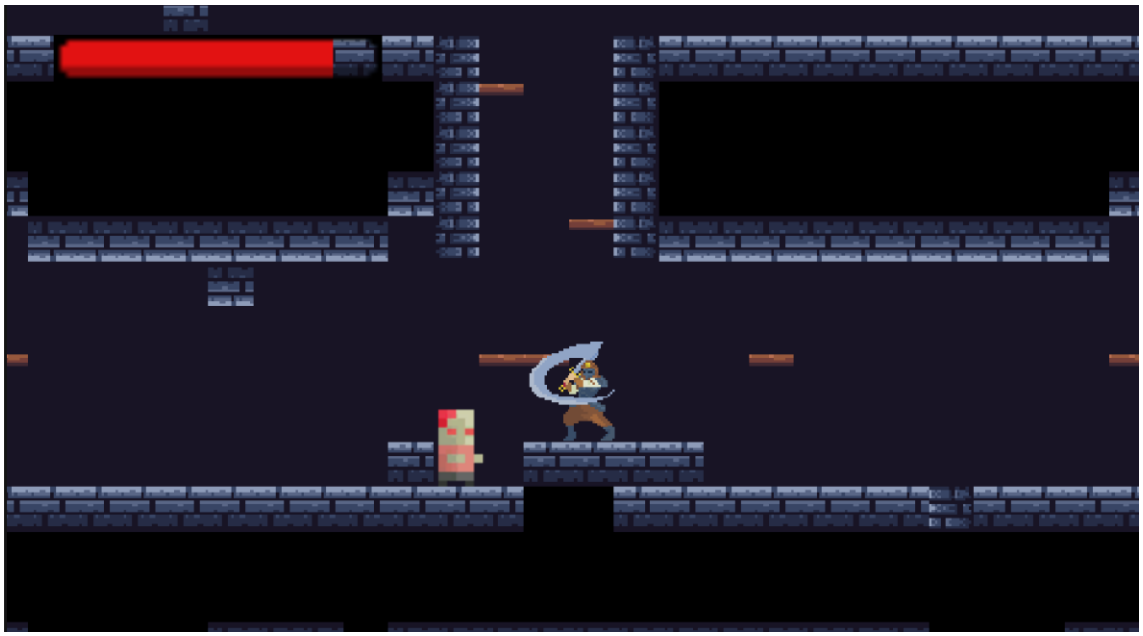


Figura 48: Lluita contra el primer enemic

En la Figura 48 es pot veure com el personatge ataca i com al zombi se li tornen els ulls vermells durant uns instants per comunicar al jugador que l'enemic ha pres mal. També es pot observar com el jugador pot baixar i pujar entre les plantes del nivell.

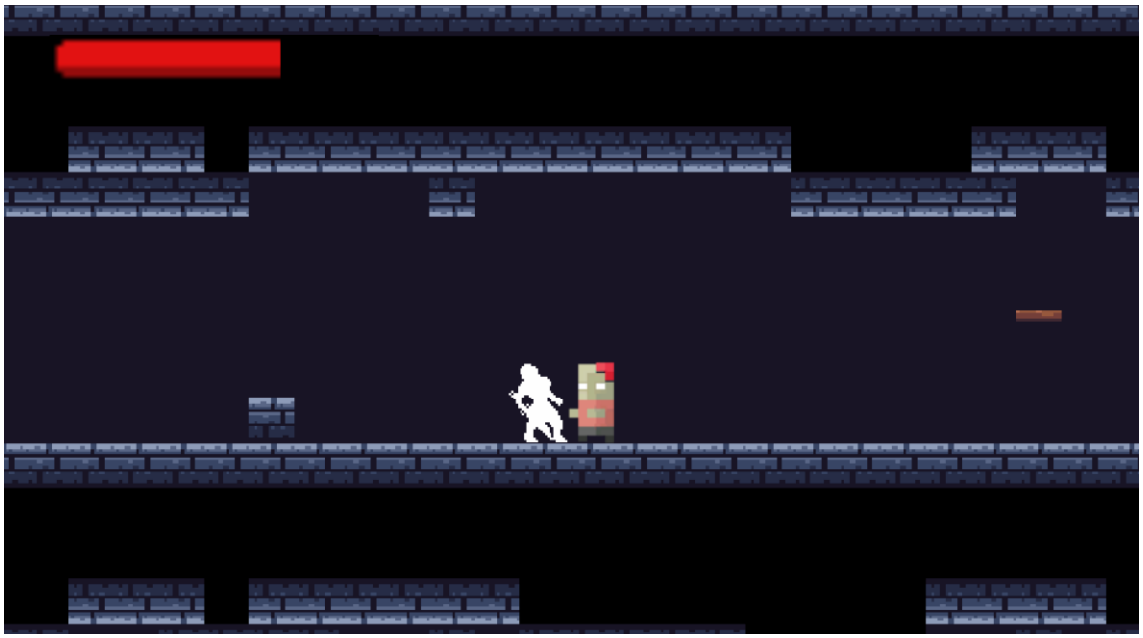


Figura 49: El personatge pren mal

Quan el personatge principal pren mal, reprodueix una animació per comunicar-ho visualment al jugador, a més de ser empès una curta distància cap enrere. Veure Figura 49.

Figura 50: Atac del mag

En la Figura 50 es pot observar com el mag realitza una animació al llençar el projectil màgic. També es pot veure que el personatge ha pres mal per la barra de vida.

Figura 51: Habitació de sortida

En la Figura 51, el jugador ha trobat la sortida que portarà al següent nivell. Una vegada s'apropi, apareixerà un text per comunicar amb quina tecla del teclat es pot interactuar amb aquesta porta. Veure Figura 52.

Figura 52: Pista per interactuar amb la porta

Al explorar el segon nivell, el jugador es podrà trobar amb parets que de moment li barren el pas. Veure Figura 53.

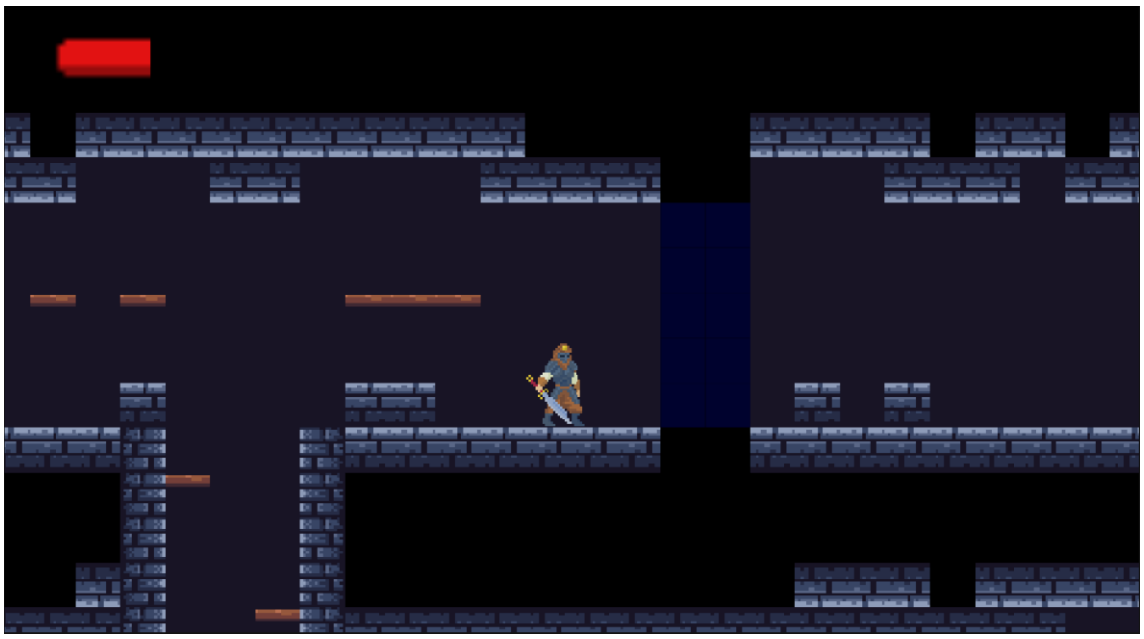


Figura 53: Paret especial

Una vegada el jugador superi el segon nivell, entrarà al nivell del “Boss”, el qual no farà res fins que el jugador entri a l’habitació. A diferència dels altres nivells, els que contenen un “Boss” no són generats proceduralment. Veure Figura 54. Una vegada entri, les parets es tancaran i la lluita començarà. Veure Figura 55.

Figura 54: Entrada a l’habitació del nigromant

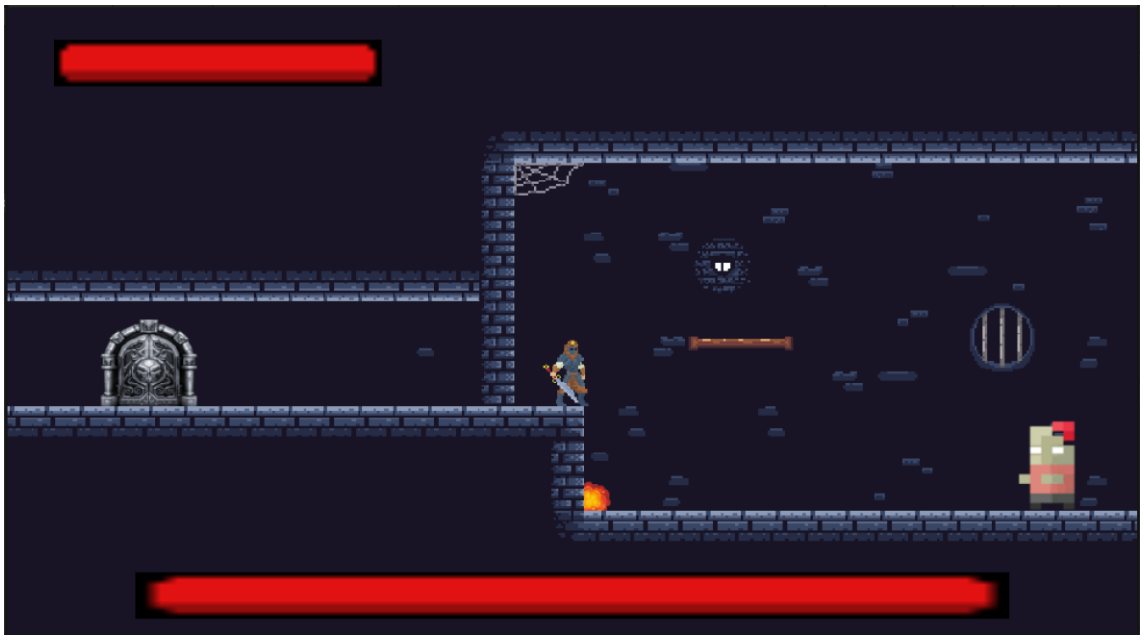


Figura 55: Començament de la lluita

Figura 56: Invocació de zombis

Com es pot veure en la Figura 56, una vegada el nigromant tingui menys de la meitat de la seva vida total, podrà invocar a zombis per a que l'ajudin. Una vegada derrotat, el jugador podrà avançar fins trobar-se que la porta al següent nivell està bloquejada per una de les parets especials que s'ha trobat anteriorment. Veure Figura 57.

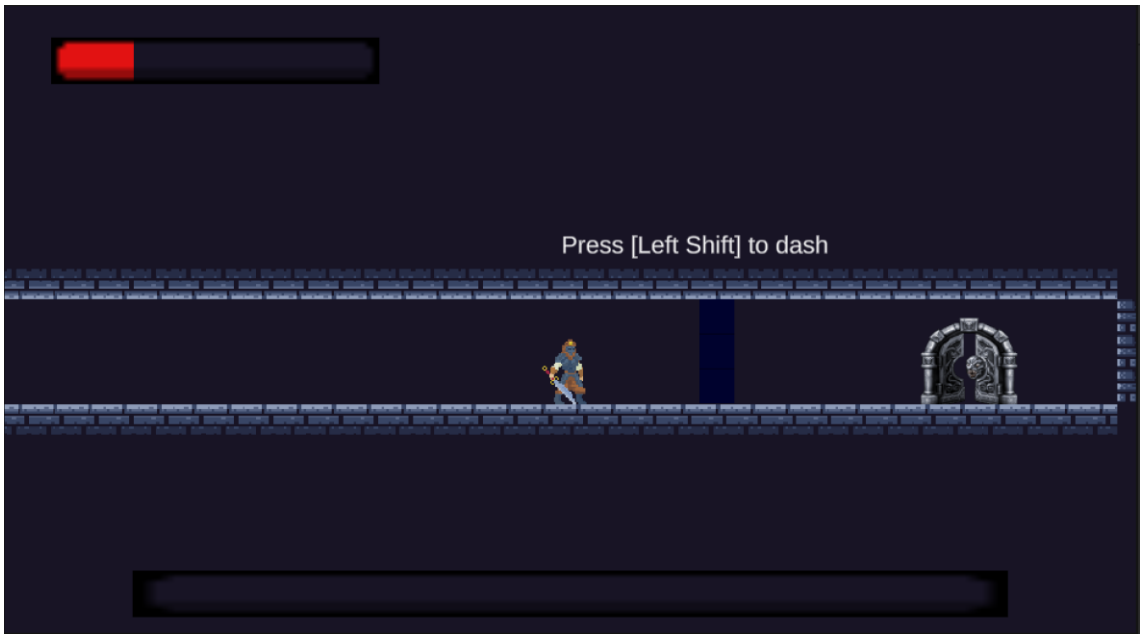


Figura 57: Obstacle abans de continuar

Per poder continuar, haurà d'utilitzar la nova habilitat que ha aconseguit al derrotar el nigromant i, seguint les instruccions escrites en la paret del calabós, travessar la paret i continuar l'aventura. Veure Figura 58.

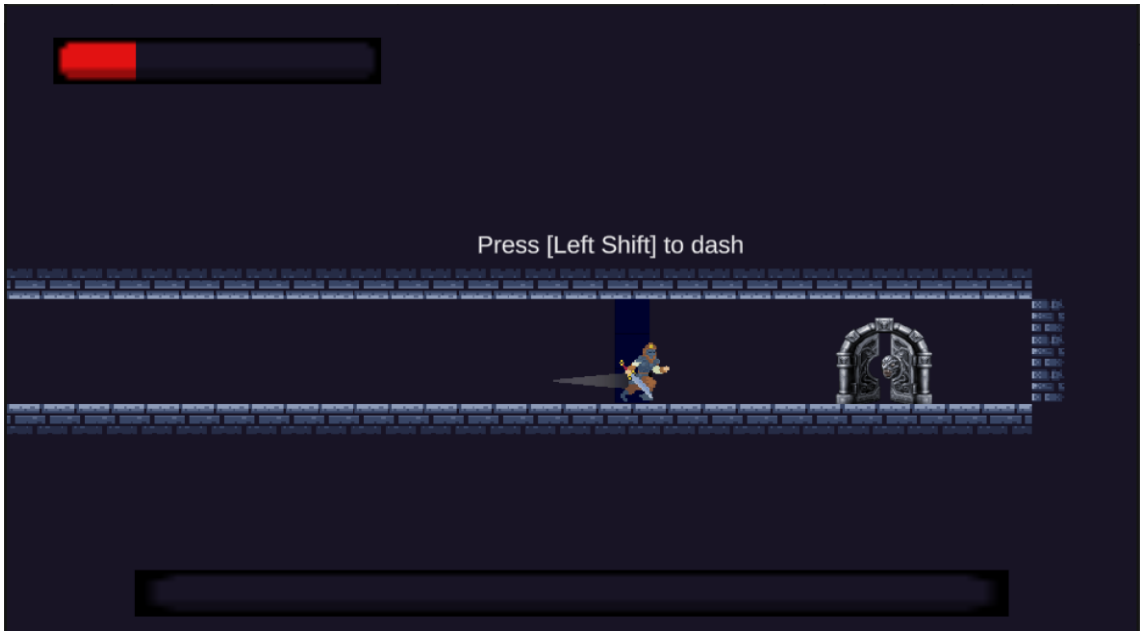


Figura 58: Travessar la paret especial amb la nova habilitat

11. Conclusions

Aquest videojoc ha estat, sense cap dubte, el treball més gran i llarg que he realitzat mai, i durant la seva realització he aprés una gran quantitat de coses.

He aprés sobre l'exagerada quantitat de feina que comporta la creació i desenvolupament d'un videojoc. El projecte m'ha ensenyat i demostrat la importància d'un bona planificació tant de temps com de feina a l'hora d'emprendre una tasca tan gran.

També he aprés com les diferents parts del procés de desenvolupament com la programació, les animacions i la música estan relacionades la unes amb les altres i com cada una d'aquestes parts requereixen coneixements i eines diferents per realitzar-se. Parlant de les eines, també he aprés la importància d'escollir l'eina adequada per resoldre cada problema. Per sort, Unity conté totes les eines que he necessitat durant el desenvolupament d'aquest videojoc.

He pogut veure la importància de tenir un bon disseny de les classes, sobretot en els projectes grans. Planificar les classes necessàries i les seves funcionalitats abans de començar a escriure codi ha estat un aspecte clau per fer possible la realització del videojoc en els tres mesos durant els que s'ha desenvolupat.

El projecte ha ressaltat la importància de escriure classes i funcions generals per a que es pugin aplicar en diferents situacions sense haver d'escriure més del necessari. Això em porta a una mancança personal de la que m'he adonat i és que es podria millorar l'aspecte de generalització del codi que he escrit. No s'han utilitzat herències fora del Behavior Tree, hi han algunes funcions repetides en diferents classes i línies de codi que es podrien haver obviat amb una millor estructura. Aquestes errades es poden atribuir a la meva falta d'experiència amb projectes d'aquesta magnitud i és un punt a millorar de la meva metodologia com a programador.

El projecte m'ha mostrat les meves fortaleses i mancances i tinc la sensació que l'experiència m'ha fet créixer com a informàtic. Al final s'han assolit les metes plantejades i m'he donat compte de la satisfacció que es guanya al crear un videojoc des de zero. La conclusió final és que, tot i poder millorar, he creat un videojoc del que puc estar orgullós.

12. Treball futur

El videojoc desenvolupat va ser dissenyat inicialment per tal de ser escalable. Par tant es poden dur a terme una gran quantitat de millores.

La demo del videojoc només mostra la primera zona del calabós, la qual té l'estètica d'un calabós tradicional amb parets de pedra de colors foscos. Una vegada derrotat el "boss" de la zona, es passaria a la següent, la qual podria comportar un canvi estètic, com afegir vegetació o, fins i tot, nivells amb foc, per oferir nous obstacles a superar, un canvi d'escenari i donar pistes visuals sobre el "boss" de la zona. Per tal de dur a terme aquesta millora, també caldria crear una major quantitat d'habitacions prefabricades de cada tipus. Un benefici de la generació procedural del mapa és que no caldrà pensar en els dissenys dels nivells que s'afegeixin. Cada zona podria contenir més variació d'enemics o versions més fortes dels enemics de zones anteriors, amb IAs i mecàniques més complexes.

Un altre punt a millorar serien les recompenses que rep el personatge. Actualment, el joc només ofereix una nova millora al derrotar l'enemic del final d'una zona, però no ofereix cap tipus de recompensa al derrotar els enemics que es troba pel camí. Això es podria millorar implementant una tenda al cada certs nivells per comprar petites millores com fer que l'espasa tingui més atac, l'armadura tingui més defensa o escurçar el temps de "cooldown" del dash. Amb aquesta nova característica, es pot fer que el personatge rebi diners al derrotar els enemics per gastar-se en aquesta tenda.

A demés de la derrota d'enemics, també s'hauria de recompensar l'exploració. Tot i que el videojoc proporciona formes d'augmentar el nivell d'exploració al jugador, no proporciona els incentius per fer que el jugador vulgui explorar. Això es podria millorar amb un sistema de tresors ocults i curtcircuits per a un recorregut més ràpid i segur fins al final del nivell.

Amb el sistema monetari mencionat anteriorment, també es podria oferir l'opció de gastar diners per curar al personatge principal després de la lluita amb el "boss" de la zona. D'aquesta manera afegim el repte de gestionar els recursos per tal de completar el videojoc.

El sistema de guardar i carregar dades es podria millorar ja que, al no ser part del enfocament principal del joc, és un sistema rudimentari que utilitza PlayerPrefs per guardar informació entre escenes.

Finalment, es podria implementar un sistema de modificació de dificultat per donar encara més rejugabilitat al joc.

13. Bibliografia

[1] Stack OverFlow, Stack Exchange Network, 2016

https://stackoverflow.com/

[2] Unity Technologies. (s.d). Unity, Forums. Recuperat el 2019

https://forum.unity.com/

[3] Unity Technologies. (s.d). Unity, Asset Store. Recuperat el 2019

https://assetstore.unity.com/

[4] Unity Technologies. (s.d). Unity, Scripting API. Recuperat el 2019

https://docs.unity3d.com/ScriptReference/

[5] Leaf Corcoran, Itch.io, 2013

https://itch.io/game-assets/free

[6]Spelunky Generator Lessons, Darius Kazemi, Tiny Subversions

https://tinysubversions.com/spelunkyGen/

[7] Thirslund, Asbjørn (Àlies Brackeys). (s.d.). Brackeys. Recuperat el 2019

https://www.youtube.com/user/Brackeys/

14. Manual d'usuari i/o instal·lació

14.1 Instal·lació

El videojoc és un executable per Windows, el qual no requereix cap mena d'instal·lació de software addicional ni de la pròpia aplicació. A l'executar el programa ja ens desplaçarem directament al menú principal del joc.

14.2 Objectiu del joc

En aquest videojoc, el jugador controlarà un mercenari en busca de riqueses i entrarà a un misteriós calabós per obtenir-les. El calabós estarà dividit en zones i cada zona en nivells generats proceduralment. El jugador haurà de trobar el camí fins al final de cada nivell i derrotar al monstre "boss" al últim nivell de cada zona abans de poder passar a la següent part del calabós. En cas de que fos derrotat, el jugador tornarà al principi per tornar-ho a intentar. Cada vegada que això passi però, els nivells canviaran.

Al ser un videojoc amb aspectes Roguelike, no té un final concret i es pot seguir jugant fins que el jugador es cansi de la partida. Es pot considerar el final d'una "run" quan el personatge principal derrota a tots els enemics "boss" i arriba a l'última planta del calabós, però una vegada allà el jugador tindrà l'opció de continuar jugant amb totes les millores i un nivell de dificultat més elevat.

14.3 Configuració

A l'iniciar el videojoc, el jugador es trobarà amb el menú principal, on podrà navegar al menú d'inici de joc, navegar al menú de so o sortir de l'aplicació. Dins del menú de so, es podrà modificar el volum general del videojoc o tronar al menú principal. Aquesta no és l'única manera de controlar el volum, ja que també es pot fer des del menú de pausa durant la partida. Al menú d'inici de joc el jugador tindrà les opcions de començar una nova partida, continuar la partida guardada en cas de que ja en tingui una començada o tornar al menú principal.

14.4 Controls

- **Moviment:** El jugador podrà controlar al personatge principal a través de les tecles A i D per moviment lateral, W i S per escalar una vegada obtingui la millora i Barra Espaiadora per saltar.

- **Dash:** Tot i que la resta de millores són passives, el dash és una habilitat que cal activar amb la tecla Shift per realitzar-la

- **Atacar:** El jugador podrà atacar en la direcció que el personatge està mirant amb el clic esquerre del ratolí

- **Pausa:** Per aturar la partida i obrir el menú de pausa, el jugador ha de prémer la tecla Escape. Pot sortir del menú i reprendre la partida amb la mateixa tecla o clicant l'opció de continuar.

- **Interactuar:** El jugador podrà interactuar amb certs elements com la porta de sortida prement la tecla E.