Universitat Oberta de Catalunya

uoc.edu

The Little Wizard

Marc Simon Criballés
Grau d'Enginyeria Informàtica
TFG Videojocs

Consultor: Joel Servitja Feu
Professor: Javier Luis Cánovas Izquierdo

5 Gener de 2019

FITXA DEL TREBALL FINAL

Títol del treball:	*The Little Wizard*
Nom de l'autor:	*Marc Simon Criballés*
Nom del consultor/a:	*Joel Servitja Feu*
Nom del PRA:	*Javier Luis Cánovas Izquierdo*
Data de lliurament (mm/aaaa):	*01/2020*
Titulació o programa:	*Grau d'Enginyeria Informàtica*
Àrea del Treball Final:	*Videojocs*
Idioma del treball:	*Català*
Paraules clau	*videojoc, aventura, repte*

Resum del Treball (màxim 250 paraules): *Amb la finalitat, context d'aplicació, metodologia, resultats i conclusions del treball*

El sector dels videojocs sempre m'ha cridat l'atenció. Fins al moment, només l'havia vist des de la part del jugador, però davant la possibilitat de realitzar el TFG en aquesta àrea, vaig decidir que era el moment de conèixer com era l'altra vessant i, de passada, demostrar les habilitats adquirides durant aquests anys a la Universitat.

Així doncs, amb la realització d'aquest treball es pretén conèixer com és el procés de desenvolupament d'un videojoc i els diferents elements que hi intervenen partint des de zero. A més, també es pretén guanyar experiència amb el desenvolupament mitjançant una de les eines més utilitzades en aquest sector com és el cas d'**Unity**.

Per assolir aquests objectius s'ha decidit realitzar un joc en 2D anomenat **The Little Wizard** que combina l'aventura amb la resolució de puzles.

En aquest treball es detallarà el procés seguit per a la seva creació juntament amb les decisions preses al llarg del projecte i els detalls més importants de la seva implementació.

Com a resultat d'aquest treball, s'ha obtingut un joc força entretingut que podrà fer passar una bona estona a qui decideixi jugar-hi.

Abstract (in English, 250 words or less):

The video game industry has always caught my attention since I was a kid. However, until this project I had only seen it from the player's point of view. For this reason, when I had the opportunity to make a project on this area I decided that it was the right moment to know the other side of video games. Moreover, it was a good way to show the skills learnt during this years on the university.

So, the main objective of this project is to know all the processes involved on video game development and gain experience using tools related to this tasks, in special, **Unity** as it is one of the most used game engines at the moment.

To achieve these objectives I decided to create a 2D game called **The Little Wizard** that combines the adventure genre with puzzle solving.

This document will outline the process followed for the creation of the game along with the decisions made throughout the project and details of its implementation.

As the result of this work, I've gained a lot of experience on game development and I managed to create a short funny video game that will make players spend a good time.

Índex

1. Introducció

1.1 Context i justificació del Treball

Realitzar un treball final és una tasca que requereix esforç i dedicació. En aquest sentit, considero que és de vital importància escollir una temàtica que t'aporti motivació. En aquest cas, com la majoria de nois de la meva generació, de petit m'agradava molt jugar als videojocs i ara malauradament ja no disposo de tant temps per fer-ho. És per això, que des d'un inici la meva idea ha estat desenvolupar un videojoc.

Un cop vaig tenir clar que volia crear un joc, vaig estar pensant quin tipus de jocs m'agraden. De petit, acostumava a jugar a jocs d'aventura en els quals s'havia d'aconseguir objectes per passar al següent nivell, resoldre puzles, matar monstres, etc. Així doncs, vaig decidir realitzar un joc que contingués aquest tipus d'elements.

Tot i així, cal tenir en compte que no disposava d'experiència prèvia en el desenvolupament de videojocs i per això també ha estat un repte afegit. Per tant, em vaig plantejar realitzar un joc entretingut i que, a la vegada, la seva lògica no fos molt complexa.

Així doncs, l'objectiu d'aquest treball és introduir-me en el món del desenvolupament de videojocs creant el meu propi joc i posar en pràctica les habilitats adquirides durant aquests anys en la Universitat. També em plantejo adquirir nous coneixements relacionats amb el desenvolupament de videojocs, en especial, en l'entorn d'Unity.

Amb la realització d'aquest treball, espero adquirir nous coneixements que m'obrin portes de cara el meu futur professional i de passada obtenir un joc el qual sentir-me'n orgullós.

1.2 Objectius del Treball

Amb la realització d'aquest treball s'esperen assolir els següents objectius:

- Crear el meu primer joc
- Aprendre l'entorn de desenvolupament d'Unity i, en especial, els seus components per realitzar jocs en 2D
- Conèixer en primera persona com és el procés de crear un joc i els diferents elements que hi intervenen
- Guanyar experiència en la programació de videojocs

1.3 Enfocament i mètode seguit

Com que no disposava d'experiència prèvia en el desenvolupament de videojocs i el temps per realitzar el projecte era limitat, vaig considerar que la millor opció era realitzar un joc que no fos excessivament complex ni tampoc gaire innovador. Així doncs, vaig decidir realitzar un joc que incorporés elements presents en altres jocs. Amb aquest enfocament podia centrar-me més en aprendre com implementar els diversos elements i no tant en la pròpia mecànica del joc.

De la mateixa manera, en l'aspecte gràfic i en les animacions que han d'aparèixer en el joc vaig considerar que el més senzill era partir de recursos gratuïts realitzats per altres persones i d'adaptar-los si era necessari. Dissenyar cada element de zero suposava una càrrega de treball massa elevada. Tot i així, hi ha alguns elements gràfics en el joc que han estat elaborats per mi. Tots els elements utilitzats en el joc que no hagin estat creats per mi s'inclouran en l'apartat de la bibliografia.

1.4 Planificació del Treball

La realització d'aquest treball s'ha dividit en una sèrie de tasques que s'han agrupat en quatre fases. Aquestes fases coincideixen en les entregues de les PACs. A continuació s'enumeren les diferents tasques i la data límit per a la seva execució juntament amb un diagrama de Gantt que ho resumeix de manera visual.

PAC1 Disseny del videojoc (18/09/19 – 29/09/1)

- Idea del joc
- Conceptualització
- Planificació del projecte

PAC2 Versió parcial (30/09/19 – 03/11/19):

- Aprendre Unity
- Crear escena per les proves
- Crear objecte jugador bàsic:
    - Sistema de moviment

    - Animació
    - Col·lisions
    - Sistema de Vida
    - Atac
- Crear objecte per als monstres:
    - Moviment
    - Animacions
    - Col·lisions
    - Seguiment del jugador
    - Atac
    - Sistema de vida

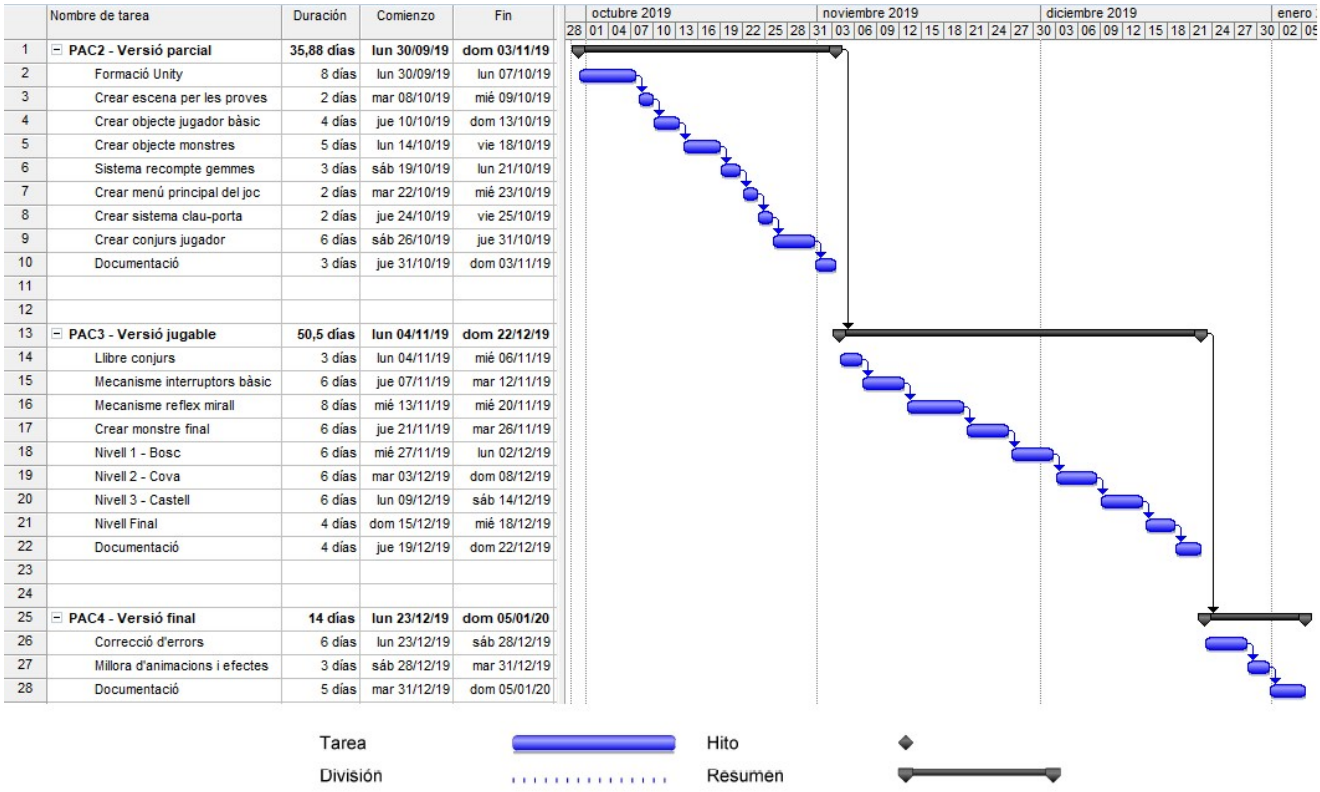
- Sistema de recompte de les gemmes:
    - Recompte de les gemmes
    - Porta per finalitzar el nivell
- Crear menú principal
- Crear sistema clau-porta
- Crear conjurs jugador:
    - Escut
    - Raig simple
    - Pluja de rajos
- Documentació

PAC 3 Versió jugable: (04/11/19 – 22/12/19):

- Crear els llibres de conjurs
- Mecanisme interruptors bàsic:
    - Interruptor
    - Pilona retràctil
- Mecanisme reflex mirall
- Crear monstre final
- Crear nivell 1- Bosc:
    - Dissenyar
    - Inserir elements al nivell
    - Proves
- Crear nivell 2- Cova:
    - Dissenyar
    - Inserir elements al nivell
    - Proves
- Crear nivell 3- Castell:
    - Dissenyar
    - Inserir elements al nivell
    - Proves
- Crear nivell Final:
    - Dissenyar
    - Inserir elements al nivell
    - Proves
- Documentació

PAC 4 Versió final (22/12/19 – 05/01/20)

- Correcció d'errors
- Millores gràfiques (animacions,*sprites*)
- Documentació



	Nombre de tarea	Duración	Comienzo	Fin
1	**PAC2 - Versió parcial**	**35,88 días**	**lun 30/09/19**	**dom 03/11/19**
2	Formació Unity	8 días	lun 30/09/19	lun 07/10/19
3	Crear escena per les proves	2 días	mar 08/10/19	mié 09/10/19
4	Crear objecte jugador bàsic	4 días	jue 10/10/19	dom 13/10/19
5	Crear objecte monstres	5 días	lun 14/10/19	vie 18/10/19
6	Sistema recompte gemmes	3 días	sáb 19/10/19	lun 21/10/19
7	Crear menú principal del joc	2 días	mar 22/10/19	mié 23/10/19
8	Crear sistema clau-porta	2 días	jue 24/10/19	vie 25/10/19
9	Crear conjurs jugador	6 días	sáb 26/10/19	jue 31/10/19
10	Documentació	3 días	jue 31/10/19	dom 03/11/19
11				
12				
13	**PAC3 - Versió jugable**	**50,5 días**	**lun 04/11/19**	**dom 22/12/19**
14	Llibre conjurs	3 días	lun 04/11/19	mié 06/11/19
15	Mecanisme interruptors bàsic	6 días	jue 07/11/19	mar 12/11/19
16	Mecanisme reflex mirall	8 días	mié 13/11/19	mié 20/11/19
17	Crear monstre final	6 días	jue 21/11/19	mar 26/11/19
18	Nivell 1 - Bosc	6 días	mié 27/11/19	lun 02/12/19
19	Nivell 2 - Cova	6 días	mar 03/12/19	dom 08/12/19
20	Nivell 3 - Castell	6 días	lun 09/12/19	sáb 14/12/19
21	Nivell Final	4 días	dom 15/12/19	mié 18/12/19
22	Documentació	4 días	jue 19/12/19	dom 22/12/19
23				
24				
25	**PAC4 - Versió final**	**14 días**	**lun 23/12/19**	**dom 05/01/20**
26	Correcció d'errors	6 días	lun 23/12/19	sáb 28/12/19
27	Millora d'animacions i efectes	3 días	sáb 28/12/19	mar 31/12/19
28	Documentació	5 días	mar 31/12/19	dom 05/01/20

Figura 1: Diagrama de Gantt

1.5 Breu sumari de productes obtinguts

- Videojoc executable per a l'entorn *dekstop (Windows)*
- Projecte d'Unity amb el codi font del videojoc
- Vídeo amb un tràiler del joc
- Vídeo on s'exposa el procés de desenvolupament
- Memòria del treball final

1.6 Breu descripció dels altres capítols de la memòria

En la resta de capítols es realitzarà una descripció detallada sobre quin ha estat el joc a desenvolupar, les consideracions de disseny seguides així com les característiques més importants dels diversos elements implementats en el joc.

2. Disseny

2.1 Descripció del joc

El videojoc desenvolupat consisteix en un joc d'aventura que combina l'acció amb la resolució de puzles. En començar la partida, es reprodueix una petita escena introductòria que posa el jugador a la pell del personatge i explica quin és l'objectiu del joc. A partir d'aquí, és el jugador qui ha de controlar les accions del personatge i avançar en la història.

La temàtica del joc pertany al gènere fantàstic i està relacionada amb la màgia. El joc ens explica la història d'un jove aprenent de bruixot el qual un dia un altre bruixot, que s'està fent vell, li encomana les tasques de recuperar unes gemmes perdudes en un bosc i una recepta robada per un ciclop. Si el jove aconsegueix superar les dues tasques quedarà demostrada la seva validesa com a mag i el bruixot vell accedirà a ser el seu mestre.

Així doncs, el jugador tindrà l'objectiu principal en el joc de recuperar totes les gemmes i la recepta robada. Inicialment estava previst realitzar la cerca de les gemmes en tres nivells diferents (bosc,cova i castell) i un de final en el qual havia d'enfrontar-se amb el ciclop i recuperar la recepta robada.

Quan s'acostava la data d'entrega en la qual el joc havia d'estar pràcticament tancat, es va decidir reduir el nombre de nivells per falta de temps, passant d'aquesta manera a un únic nivell més extens per a trobar les gemmes i un nivell final per derrotar el ciclop i obtenir la recepta.

Pel que fa l'acció i la resolució de puzles, al llarg del camí el jugador haurà d'esquivar perills i defensar-se dels enemics que anirà trobant. També haurà de trobar la manera d'activar algun mecanisme que li permeti accedir a una zona inicialment inaccessible utilitzant elements del seu entorn.

El jugador podrà realitzar atacs cos a cos amb una vareta màgica o llançant conjurs. Per donar més noció de progrés, inicialment el jugador només podrà realitzar atacs cos a cos. Per aprendre un conjur el jugador haurà de trobar un llibre. Hi haurà un llibre per cada conjur. El primer llibre que trobi d'un determinat conjur servirà per aprendre'l, els següents serviran per pujar-lo de nivell.

El jugador disposarà d'una certa quantitat de vida que disminuirà si és atacat per algun monstre o trepitja un parany. Si el jugador es queda sense vida morirà i haurà de tornar a iniciar el nivell. Per recuperar vida podrà fer ús de pocions que trobarà en el camí. De la mateixa manera, el

jugador tindrà associat una determinada quantitat de mana que li servirà per realitzar conjurs.

Cada conjur consumirà una determinada quantitat de mana. Si el jugador no té suficient mana no podrà realitzar el conjur. Un cop realitzat el conjur, haurà d'esperar un petit interval de temps per tornar-lo a utilitzar.

Finalment, també es va decidir afegir al jugador una determinada quantitat d'energia. Aquesta energia es gasta al realitzar un atac cos a cos amb la vara màgica i es torna a recuperar amb el pas del temps. Aquesta decisió es va prendre per limitar el nombre d'atacs que pot realitzar el jugador amb la vara de manera repetida.

2.2 Definició dels personatges i/o elements

Un cop descrit el joc, passem a definir més concretament els personatges i elements que apareixeran en el joc.

- **Jove bruixot:** És el personatge principal de la història i el que controla el jugador. Té una determinada quantitat de vida, de mana i d'energia. Pot realitzar atacs cos a cosa amb la vara màgica i llançar conjurs.
- **Enemics:** Són éssers animats que persegueixen i ataquen el bruixot quan es troben a un cert radi de distància. N'hi que poden atacar a distància i d'altres que ho fan cos a cos.
    - **Atac cos a cos:**
        - **Goblin:** Persegueix el jugador i l'ataca amb l'espasa.
        - **Cobra:** Persegueix el jugador i el mossega.
    - **Atac a distància:**
        - **Arquer:** Patrulla en una determinada zona del mapa i quan té el jugador a una certa distància el persegueix i/o dispara.
        - **Bolet:** Persegueix el jugador quan està a prop seu i l'ataca llençant un projectil.
        - **Tronc:** Inicialment està adormit, si el jugador passa a prop seu, el persegueix i l'ataca fent aparèixer una arrel amb punxes al costat del jugador.
        - **Planta:** Dispara projectils en vàries direccions cada cert interval de temps i es teletransporta cap a un altre punt del mapa (dins d'un cert radi) de manera aleatòria.

- **Ciclop:** Es tracta d'un *boss*, és a dir, un monstre que apareix en l'últim nivell i que el jugador haurà de matar per passar-se el joc. Tira pedres al jugador i dispara un raig làser amb el seu ull. També pot fer tremolar el terra i deixar atordit el jugador durant un curt període de temps.
- **Gemmes:** Parts del braçalet que el jove bruixot ha de trobar. En el nivell s'indicarà quantes n'hi ha. Per passar al següent nivell caldrà que el jugador les hagi trobat totes.

- **Porta següent nivell:** Un cop el jugador tingui totes les gemmes, apareixerà un portal que permetrà accedir al següent nivell.
- **Pocions**: N'hi haurà per recuperar vida i per recuperar mana. El jugador les utilitzarà quan hi passi per sobre.
- **Llibre de conjurs:** És un objecte que podrà recollir el jugador quan explori els nivells. Permetrà aprendre un nou conjur o pujar de nivell un que ja té.
- **Recepta:** És un objecte que ha de recollir el jugador després de matar el ciclop per passar-se el joc.
- **Baüls i gerros:** Poden contenir objectes al seu interior. Quant el jugador els colpeja o els dispara tiren els objectes que hi ha en el seu interior.
- **Conjurs màgics:**
    - **Raig d'energia:** Dispara un projectil en la direcció que apunti al jugador. Si aquest raig toca algun enemic se li restarà una determinada quantitat de vida. També permet activar mecanismes.
    - **Escut**: Protegeix al bruixot dels atacs durant uns segons. Si el jugador toca una trampa l'escut desapareix.
    - **Pluja de rajos:** Genera una sèrie de rajos que causen dany als monstres que es troben a prop del lloc on s'ha llençat el conjur.
- **Mecanismes:** Permeten activar o desactivar algun element del joc. Com a elements activables tindrem ponts que apareixen i desapareixen, una pilona retràctil i elements que es mouen o es deixen de moure. Com a elements activadors dels mecanismes tindrem palanques, botons al terra i comptadors d'enemics que s'han de matar.
- **Complements per als mecanismes:** Consisteixen en elements que ajudaran al jugador a activar algun mecanisme:
    - **Estàtua:** Objecte que pot empènyer el jugador. Es pot utilitzar per deixar polsat un botó.
    - **Mirall:** N'hi ha en vàries direccions i angles. Reflecteixen el raig del jugador i permeten activar palanques que no són accessibles directament.

- **Clau:** Objecte que permet obrir una porta tancada amb clau.
- **Trampes:** Consisteixen en objectes que es trobarà en el camí que causaran dany al jugador si aquest passa per sobra o el toquen.
    - **Tronc amb punxes:** Tronc que es mou en horitzontal o en vertical.
    - **Trampa de l'ós:** Objecte fixat al terra que causa dany al jugador si aquest el trepitja.
    - **Punxes al terra:** punxes al terra que surten cada cert interval de temps.
- **Canonada:** Objecte que es pot situar a les parets i que dispara projectils en horitzontal o en vertical depenent de la seva orientació.

2.3 Diagrama de classes UML del joc independent de la tecnologia

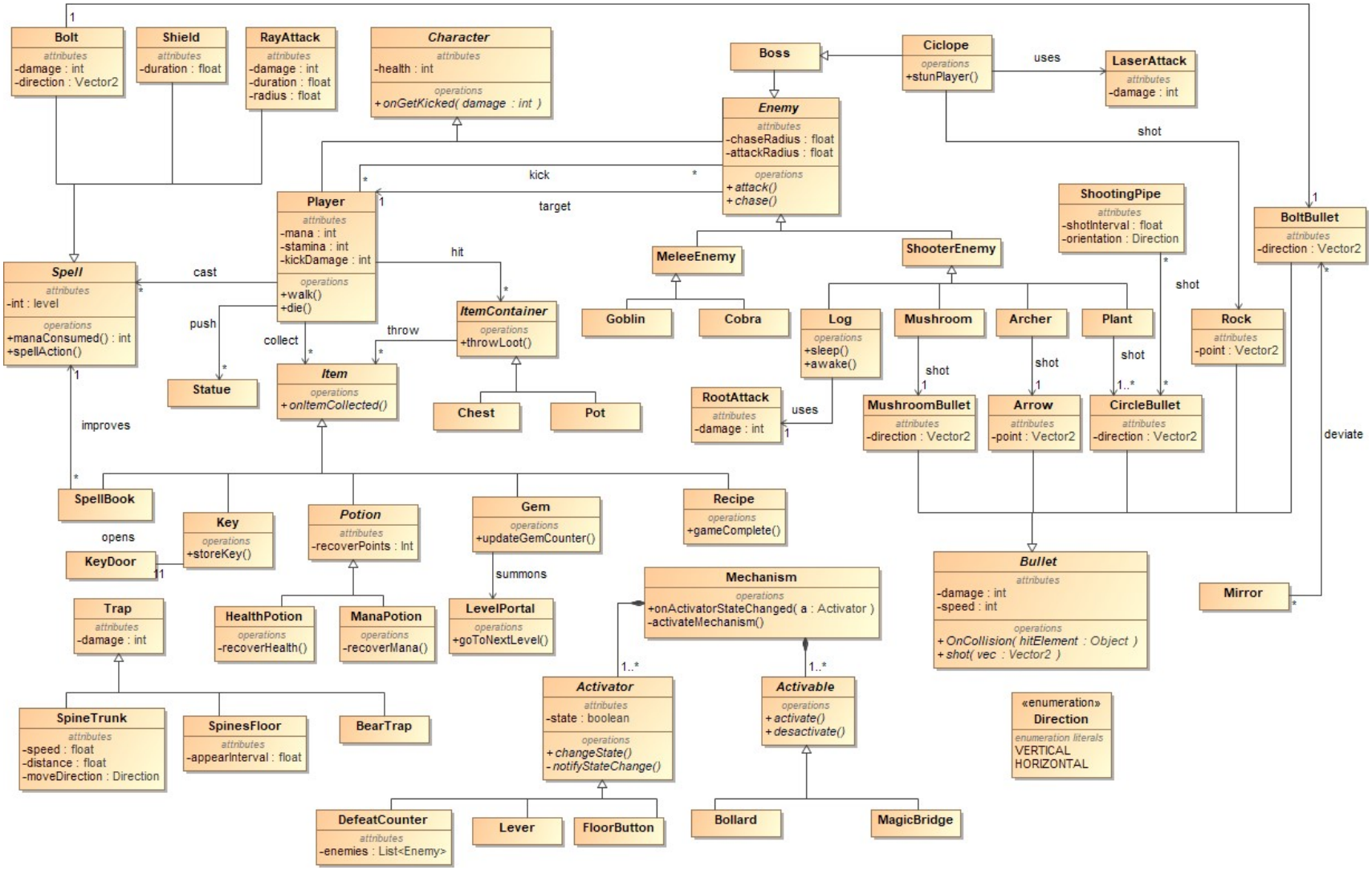


Figura 2: Diagrama de classes amb els diferents elements del joc

2.4 Gràfics

Pel que fa a l'aspecte gràfic del joc, s'ha considerat que l'opció més senzilla era realitzar-lo en 2D i amb una perspectiva *top-down*, és a dir, que es vegi al jugador al mig de la pantalla des de dalt i l'àrea del seu voltant.

Respecte a l'estil gràfic, s'ha decidit utilitzar l'estil *pixel-art* perquè hi ha una gran varietat *d'assets* disponibles de manera gratuïta d'aquest estil. A més, si es necessari també són senzills de dibuixar/animar.

Figura 3: Perspectiva i gràfics del joc desenvolupat

En quan a les animacions dels personatges l'ideal hagués estat realitzar-les en vuit direccions, però per estalviar temps s'ha decidit fer-les en quatre direccions (amunt, avall, esquerra i dreta).

Per al disseny dels gràfics del joc he utilitzat **Aseprite**, ja que es tracta d'un programari especialitzat en el disseny i animació de gràfics de l'estil *pixel-art*. Aquest és un programa que desconeixia abans de començar el projecte i que m'ha facilitat molt les tasques d'animació i generació de *sprites-sheets*.

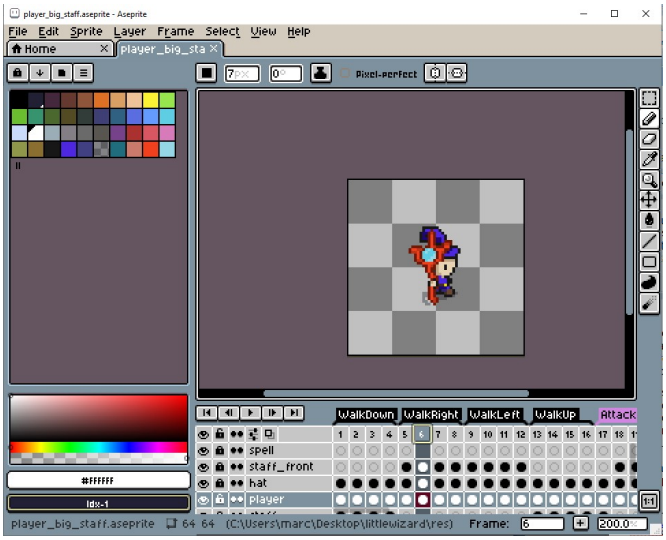


Figura 4: Disseny del personatge mitjançant Aseprite

2.5 Plataforma de destí

Generalment, un dels objectius principals de crear un joc és que pugui ser jugat per al major nombre possible de persones. Una manera d'aconseguir-ho és fer que el mateix joc estigui disponible en diferents plataformes.

Tot i així, en aquest cas no ho he considerat com un objectiu tant prioritari. He preferit centrar més els esforços en polir els detalls del joc que la plataforma concreta de destí. Per aquest motiu, el joc s'ha pensat inicialment per ser jugar únicament en la plataforma *desktop*, ja que és la més còmode per realitzar proves.

2.6 Interacció joc-jugador

El joc tindrà un menú principal en el qual es podrà iniciar el joc o modificar certes opcions com, per exemple, el volum. Dins d'un nivell, el jugador podrà controlar totes les accions que pot realitzar el personatge principal, és a dir, moure'l, recollir objectes i atacar. Dintre d'un nivell, el joc informarà els estats del jugador (vida,mana,energia) i un resum de les gemmes trobades i les que li queden per trobar.

Inicialment, es tenia previst controlar el moviment del jugador i l'elecció dels conjurs amb el teclat i llançar els conjurs o atacar amb el ratolí. Però després de provar una primera implementació utilitzant aquest sistema, es va optar per controlar tots els moviments el jugador utilitzant únicament el teclat. Aquesta decisió es va prendre perquè la primera implementació donava alguns petits errors visuals, com per exemple, que el jugador realitzés un conjur en la direcció oposada a la que caminava.

2.7 Referències a videojocs existents

Com s'ha comentat anteriorment, per desenvolupar aquest joc he agafat com a referència altres jocs existents del mateix estil. Un dels que més s'hi assembla és *Ittle Dew*[1]. Es tracta també d'un joc d'aventura en el qual el jugador ha de resoldre una sèrie de puzles, trobar objectes i defensar-se dels monstres que es va trobant.

Pel que fa a l'activació d'alguns mecanismes, utilitzaré un estil similar als puzles que apareixen en el joc *CrossCode*[2]. En aquest joc el jugador ha d'utilitzar uns objectes que fan rebotar els projectils que dispara o ha de derrotar un determinat nombre d'enemics.

[1] Ittle Dew: http://www.ittledew.com/ (25/09/19)
[2] Puzle CrossCode: http://www.radicalfishgames.com/wp-content/uploads/bounce-switch.gif (28/09/19)

2.8 Avaluació d'*engines* per al desenvolupament.

Unity és un motor de videojocs desenvolupat per *Unity Technologies* que permet desenvolupar jocs i animacions, tant en 2D com en 3D, per a la gran majoria de plataformes. Si els ingressos anuals obtinguts no superen els 100.000$, es pot utilitzar gratuïtament mitjançant una subscripció al pla personal.

Aquesta plataforma permet incorporar ràpidament als projectes components genèrics utilitzats en la majoria de jocs, com per exemple, la detecció de col·lisions o la física. Aquests components es poden adaptar posteriorment modificant els seus paràmetres a través d'Scripts en llenguatge C#.

Actualment és l'*engine* més utilitzat per al desenvolupament de videojocs i disposa d'una gran comunitat d'usuaris al darrere. Degut a aquest fet, hi ha una gran quantitat de tutorials i documentació que són de gran ajuda per als qui s'inicien al món del desenvolupament de videojocs.

D'altra banda, **Unreal Engine 4** és un motor de videojocs multiplataforma creat per l'empresa *Epic Games* que permet crear jocs 2D i 3D, tot i que està pensat més per a 3D. És el segon *engine* més utilitzat arreu del món per darrere d'Unity i es pot utilitzar gratuïtament. Si els beneficis del joc superen els 3.000$ caldrà pagar el 5% en concepte de *royalties*.

El motor gràfic 3D d'Unreal Engine és més potent que el de Unity però també és més complex. Per la qual cosa, està més pensat per a professionals i projectes grans. A diferència d'Unity utilitza el llenguatge C++ per programar el comportament dels elements del joc. Per reduir la complexitat en la programació les darreres versions incorporen un sistema de programació visual que anomenen *Blueprints*. Aquest sistema té l'avantatge que permet connectar elements i funcions sense haver d'escriure codi.

Finalment, l'últim *engine* avaluat **és Game Maker Studio**. Es tracta d'un motor de videojocs multiplataforma desenvolupat per "Yoyo Games" pensat principalment per a jocs 2D. En aquest entorn s'utilitza un sistema de programació visual en format *drag&drop* i un llenguatge de programació propi anomenat "*Game Maker Language*". Aquest motor és especialment adequat per als qui tinguin pocs coneixements de programació.

Un dels inconvenients que té és que només es permet obtenir gratuïtament una versió de prova amb funcionalitat limitada. Si es vol adquirir una llicència amb tota la funcionalitat els preus van dels 39 als 1500$ segons els mesos de subscripció i les plataformes en les que es poden publicar els jocs.

Tenint en compte la informació exposada anteriorment, considero que l'*engine* més adequat per realitzar aquest projecte és **Unity.** He decidit

utilitzar aquest i no un altre principalment per un motiu. Com que no disposo d'experiència en el desenvolupament de videojocs, la gran quantitat de tutorials i informació que hi ha disponible a la xarxa sobre aquest *engine* seran de gran ajuda en les tasques de desenvolupament.

2.8 Resum d'eines utilitzades per realitzar el projecte

- **Unity**: Per al desenvolupament del joc en general.
- **Visual Studio**: Per a la generació del codi font del joc.
- **Aseprite**: Per dissenyar, animar o adaptar els diferents *sprites* que apareixen en el joc.
- **MagicDraw**: Elaboració de diagrames UML.
- **Audacity**: Retocar efectes de so.
- **Github**: Control de versions del projecte.

3. Implementació

3.1 Resum del procés seguit.

Com estava previst des d'un inici en la planificació, el pas previ a la implementació ha estat dedicar uns dies a aprendre els conceptes bàsics sobre Unity, els seus components i l'API de C# per programar el joc.

Seguidament, s'ha procedit a implementar un petit nivell de prova on anar desenvolupant els diferents elements del joc. En gran mesura, l'ordre d'implementació ha vingut donat per la planificació inicial.

Per a la gran majoria d'elements, les tasques d'implementació han consistit en obtenir i adaptar els recursos gràfics necessaris, crear les animacions a Unity i programar-ne la lògica.

Un cop desenvolupats la majoria d'elements, s'ha passat a dissenyar i implementar un nivell que els incorporés. En finalitzar aquest punt, s'ha observat un endarreriment en el calendari que feia inviable implementar tots els nivells previstos (tres nivells més un nivell final) amb el temps que restava per a la finalització del projecte.

Per aquest motiu, s'ha decidit introduir canvis en la planificació del projecte. D'aquesta manera s'ha passat a realitzar un primer nivell més extens i a planificar una escena inicial i un nivell final senzill per tenir el joc tancat en la data de finalització del projecte. El temps sobrant s'ha dedicat a afegir so al joc.

3.2 Variables compartides i sistema d'esdeveniments.

Per desacoblar els diferents elements del joc, s'ha utilitzat una solució trobada en la secció *how-to*[3] d'Unity que consisteix en implementar variables compartides entre objectes com si es tractessin *d'assets* i un sistema per notificar esdeveniments mitjançant **ScriptableObjects**.

Concretament, s'han definit variables compartides pels estats del jugador (vida,mana, energia) i pel del *boss* (vida i mana). L'ús d'aquest sistema ha permès desacoblar el càlcul d'aquests estats amb la seva representació en la interfície gràfica. Per al joc se n'han definit dos tipus. **ObservaleInteger** per als enters i **ObservableFloat** per als nombres en coma flotant.

Pel que fa al sistema d'esdeveniments, s'han definit dos tipus d'esdeveniments:

- **Esdeveniment simples**: Notifiquen un esdeveniment (criden una funció) als objectes que hi estan subscrits sense passar cap paràmetre. S'han utilitzat per:
    - Gestionar els diàlegs.
    - Gestionar els canvis de zona.
    - Gestionar *Timelines*.
    - La recol·lecció d'objectes.
    - Notificar l'estat del joc(pausa, continuar i fi del joc).
    - Lluita amb el *boss*.

- **Esdeveniments de conjur**: Per actualitzar en la interfície gràfica la informació relacionada amb els conjurs, resultava més senzill realitzar les accions si es coneixia el conjur al qual feia referència l'esdeveniment. Per aquest motiu es va crear una versió del sistema d'esdeveniments simples per tal que passés com a paràmetre el conjur al qual feia referència l'esdeveniment. S'han definit esdeveniments per:
    - Pujar de nivell un conjur.
    - Notificar que s'ha realitzat un conjur.
    - Notificar que s'ha seleccionat un conjur.

3.2 Gestors del joc

En cada nivell hi ha una sèrie d'objectes del tipus *singleton* encarregats de gestionar aspectes concrets del joc:

- **GameManager:** Conté les funcionalitats per reiniciar el nivell, sortir del joc i passar al següent nivell. També s'encarrega de reiniciar altres gestors en finalitzar un nivell.

[3] *Scriptable Objects*: https://unity3d.com/es/how-to/architect-with-scriptable-objects (30/11/19)

- **LevelManager**: S'encarrega de portar el seguiment de les gemmes que hi ha en el nivell, de les que falten per trobar i de guardar la clau. També és l'encarregat de calcular l'alçada en la que es troba una determinada posició en el nivell del joc.
- **DialogManager:** Gestiona la creació de diàlegs i dels missatges.
- **SpellsManager:** Gestiona les operacions relacionades amb els conjurs:
    - Seleccionar els conjurs segons les tecles premudes.
    - Guardar el nivell dels conjurs.
    - Càlcul de mana gastat per un conjur segons el nivell.
    - Càlcul del temps d'espera d'un conjur segons el nivell.
    - Càlcul del poder d'atac d'un conjur segons el seu nivell.

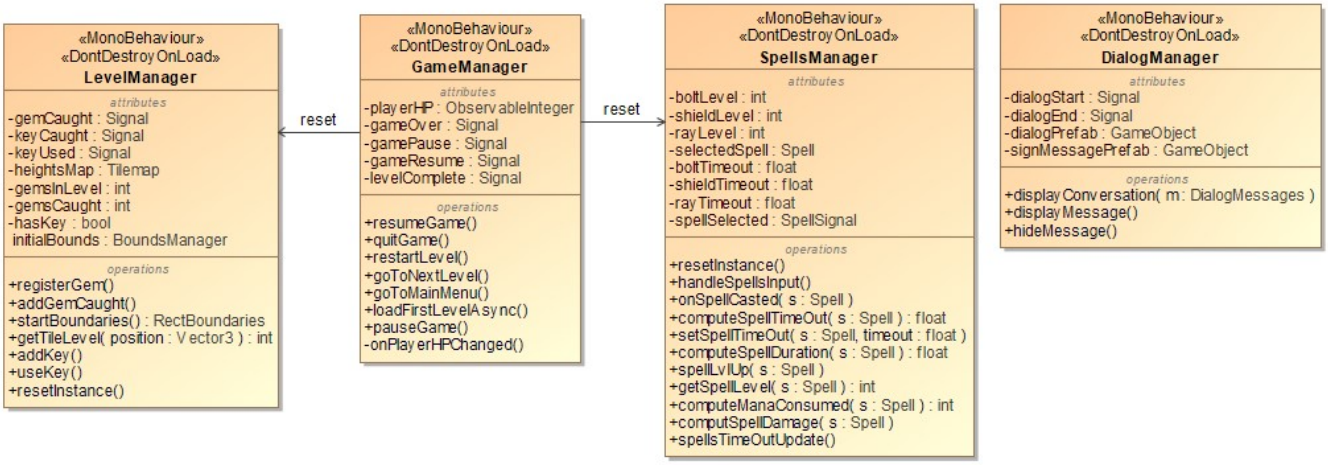


Figura 5 Diagrama de classes dels gestors del joc

3.3 Projectils

Inicialment, es tenia previst implementar projectils que es dirigissin en línia recta un cop disparats. Després d'implementar l'arquer es va observar que el moviment de la fletxa en línia recta no era gaire real. Per aquest motiu, vaig buscar la manera de poder fer que alguns projectils seguissin una trajectòria parabòlica.

La solució trobada ha consistit en calcular una trajectòria similar a la parabòlica utilitzant la funció per a les **corbes quadràtiques de Bezier**[4]. A més, la derivada d'aquesta funció també m'ha permès obtenir la direcció on apunta la fletxa en cada punt de la trajectòria. També s'ha trobat en un fòrum[5] els càlculs necessaris per aconseguir que la velocitat d'aquest projectil tingui una velocitat uniforme al llarg del seu recorregut.

[4] Corba de Bezier https://es.wikipedia.org/wiki/Curva_de_B%C3%A9zier (30/12/2019)
[5] Moviment uniforme dels projectils: https://gamedev.stackexchange.com/questions/27056/how-to-achieve-uniform-speed-of-movement-on-a-bezier-curve (30/12/2019)

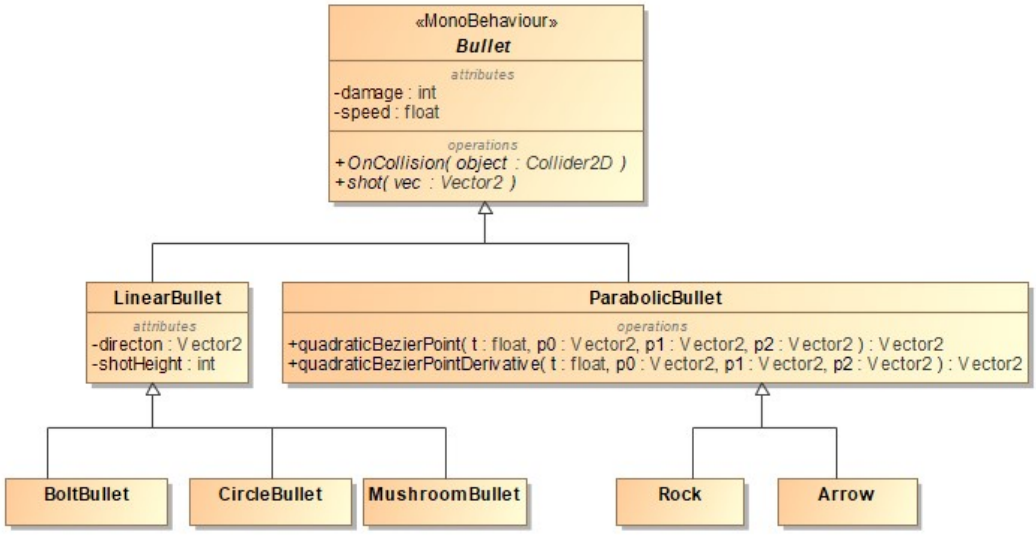


Figura 6: Diagrama de classes dels projectils

El mètode per disparar un projectil és similar en els dos casos. Primer es crea una instància del projectil a partir del seu *prefab* i seguidament es crida a l'operació **shot()** passant com a paràmetre el punt de destí o la direcció depenent del projectil.

Finalment comentar també que per què el sistema de la física detectés correctament les col·lisions dels projectils en moviment, ha calgut assignar en el component **RigidBody2D** de cada projectil la propietat "*Collision detection*" a *Continuous.*

3.4 Jugador

La implementació del jugador s'ha realitzat a través de diferents tasques. La primera de totes ha consistit en obtenir *l'sprites-sheet* amb els diferents moviments del jugador. Per realitzar aquesta tasca, s'ha utilitzat com a base el personatge i les animacions publicades per l'usuari **ArMM1998** a opengameart[6]. Bàsicament, els canvis introduïts han consistit en modificar el jugador per què portés una vareta màgica, crear les animacions d'atac amb la vara, afegir-li un barret i canviar el color de la roba.

[6] Base del jugador: https://opengameart.org/content/zelda-like-tilesets-and-sprites (30/09/2019)

Figura 7: Base utilitzada per crear el jugador i el jugador

La següent tasca ha consistit en la implementació de la lògica del jugador a Unity. Per fer-ho s'ha creat un objecte amb una sèrie de components associats per tal que tingui el comportament esperat. El més rellevat és un **Script** anomenat **Player** que s'encarrega de gestionar els moviments del jugador en funció de les tecles polsades. L'estratègia seguida per gestionar aquets moviments ha consistit en aplicar el patró **Estat**.

Per aplicar aquest patró, s'han definit els quatre estats bàsics en els que pot estar el jugador: quiet, caminant, atacant i realitzant un conjur. També, s'ha decidit des de quins estats o condicions es pot passar d'un estat a l'altre.

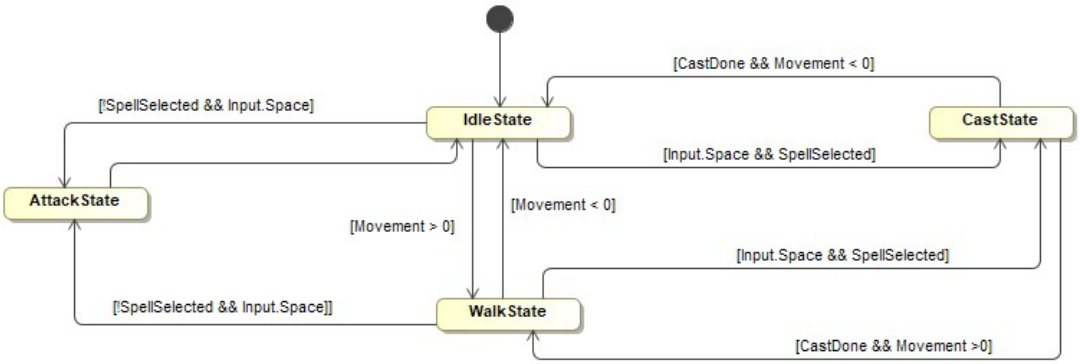


Figura 8: Diagrama d'estats del jugador

Seguidament, s'ha creat una classe per cada estat en les que s'implementen les operacions **handleInput**() i **Act**(). La primera determina el següent estat en funció de les tecles polsades i la segona realitza les accions pertinents d'aquell estat. Cada una d'aquestes classes té una referència al component **Animator** i al seu **RigidBody** per tal que cada classe pugui moure i/o animar el jugador.

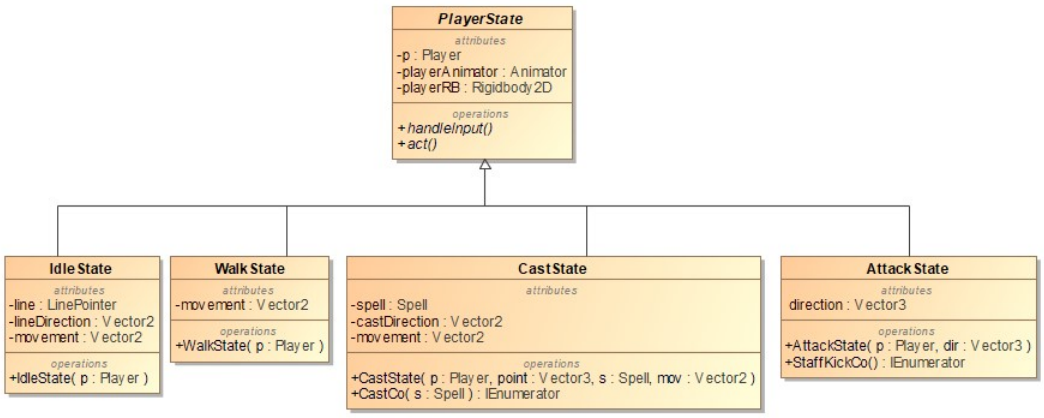


Figura 9: Classes dels estats del jugador

Així doncs, aplicant aquest esquema, la lògica que fa moure el jugador es troba a la classe **WalkState**, la que fa referència a l'atac es a **AttackState** i la del llançament de conjurs es gestiona des de **CastState**.

Per què funcioni correctament el sistema de la física d'Unity, és recomanable que les operacions que moguin objectes en les que hagi d'intervenir la física, es realitzin dins la funció **FixedUpdate**. Per aquest motiu, la crida a l'operació **Act** es realitza dins d'aquesta funció. En canvi, per obtenir les tecles premudes és aconsellable realitzar-ho en la funció **Update**, ja que si es realitza en **FixedUpdate** es pot passar per alt alguna tecla premuda pel jugador.

```
public override void Update() {

    base.Update();

    if (!freeze) {
        currentState = currentState.handleInput();
    }

}

0 references
public void FixedUpdate() {

    if (!freeze) {
        currentState.act();
    }
```

Figura 10: Fragment de codi per a la gestió dels estats del jugador

Pel que fa als components genèrics, d'Unity s'ha utilitzat la combinació de **Rigidbody2D** i **BoxCollider2D** per fer que el jugador pogués topar amb el seu entorn i el component **SpriteRenderer** per mostrar la imatge del jugador.

Com que tant el jugador com els enemics utilitzen pràcticament els mateixos components i els dos poden ser colpejats, vaig decidir crear un component abstracte de tipus script anomenat **Character** que implementés les funcionalitats compartides.

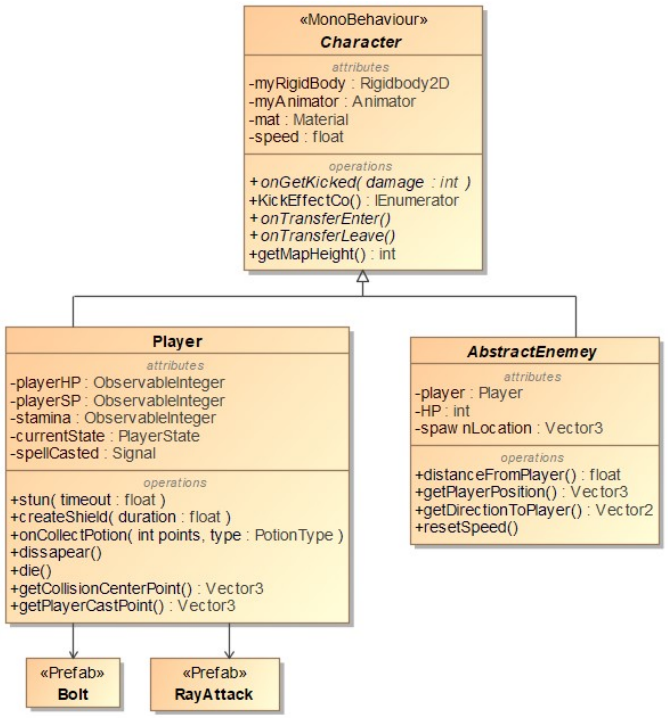


Figura 11: Classes del jugador i dels enemics

També s'ha utilitzat el component **Animator** per crear les animacions del jugador en les quatre direccions i les condicions per passar d'una animació a una altra. Comentar també que les animacions han estat realitzades mitjançant *Blend Trees*.

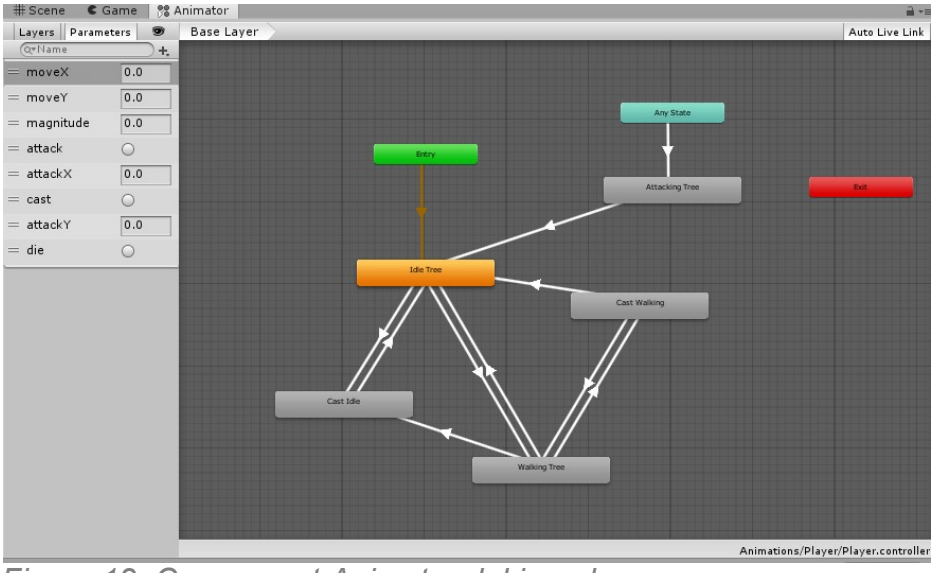


Figura 12: Component Animator del jugador

Pel que fa a la lògica dels càlculs de la vida del jugador i de la seva mort, es realitza en l'operació **onGetKicked.** Bàsicament aquesta operació consisteix en restar de la vida el valor de l'atac rebut. En el cas que aquesta arribi a zero, s'activa un objecte que mostra l'animació d'una explosió i torna el jugador invisible deshabilitant el component **SpriteRenderer**.

3.5 Atac del jugador

Per a realitzar l'atac amb la vara s'ha afegit un objecte al jugador que conté un *collider* de tipus *trigger*. Durant l'animació d'atac, aquest objecte està actiu i segueix la posició de la vara. S'hi ha afegit també un component de tipus *script* anomenat **StaffKick**, que implementa la lògica de la col·lisió dels altres objectes rellevants per la vara.

```
public void OnTriggerEnter2D(Collider2D other) {

    if (other.gameObject.tag == Enemy.TAG) {
        AbstractEnemy enemy = other.gameObject.GetComponent<AbstractEnemy>();
        enemy.OnGetKicked(KickPower);

    }else if (other.CompareTag(ItemContainer.TAG)) {

        ItemContainer container = other.GetComponent<ItemContainer>();
        container.open();
    }
}
```

Figura 13: Fragment de codi per a la gestió dels atacs amb la vara

Finalment, també es va decidir que quan el jugador realitzés un atac amb la vara també fes un moviment curt però ràpid en la mateixa direcció que l'atac. Aquest efecte permet al jugador moure's més ràpidament durant uns instants i també requereix una mica més d'habilitat per atacar.

3.6 Conjurs

Com s'ha comentat anteriorment, part de la lògica dels conjurs està implementada dins del gestor de conjurs. D'altra banda, l'encarregat d'instanciar-los és el jugador quan es troba en l'estat **CastState**, més concretament, dins l'operació **Act().**

Per identificar els conjurs s'ha creat l'enum **Spell** amb els valors NONE, BOLT,SHIELD i RANGE_ATTACK. En el constructor de l'estat **CastState** s'especifica mitjançant aquest tipus de dades el conjur que s'ha de realitzar.

Com que el procés de llançar un conjur ha de durar més d'un frame, s'ha implementat la lògica per instanciar els conjurs en la corrutina **CastCo**() Per a cada conjur cal realitzar accions diferents:

- **Raig d'energia**: Consisteix en instanciar un projectil lineal a partir del seu prefab i disparar-lo en direcció a un punt determinat. Abans de disparar-lo (funcio shot()), s'assigna el dany que causarà el projectil segons el nivell del conjur i l'alçada en què s'ha disparat.

Figura 14: Conjur raig d'energia

- **Escut**: Primer de tot, s'obté del gestor de conjurs el temps que ha de durar aquest conjur. Seguidament, es mostra l'animació d'una bombolla que representa l'escut i es manté la variable del jugador **isInvencible** a cert mentre dura el conjur. Un cop finalitzat el conjur desapareix la bombolla i es torna a deixar aquesta variable a fals. En l'operació **OnGetKicked**() del jugador es comprova el valor de **isInvencible** per determinar si ignorar el cop o restar-li vida.

Figura 15: Conjur escut

- **Pluja de rajos**: S'instancia el *prefab* dedicat a l'atac dels rajos i se'n configura la durada i la quantitat de dany que genera segons els càlculs del gestor dels conjurs.

  Respecte a la implementació d'aquest *prefab*, s'ha buscat la manera de realitzar un efecte similar al d'un raig elèctric. La solució trobada ha consistit en utilitzar el component **LineRenderer**.

  Concretament, la generació d'aquest efecte consisteix en calcular una sèrie de punts que van des d'un extrem fins a l'altre de la línia. Es parteix d'un dels extrems. Per determinar el següent punt, es calcula la direcció d'aquest amb el punt situat a l'altre extrem. Seguidament, es desvia de manera aleatòria aquesta direcció seguint un factor de desviament i es calcula el punt que es troba en aquella direcció i una distància determinada.

```
void UpdateEnemyRay(EnemyRay r, float devationFactor) {

    float deviation = devationFactor; //0.5f;
    Vector3 enemyPos = r.enemy.transform.position;
    Vector3 lastPoint = transform.position;
    int i = 1;

    r.line.SetPosition(0, transform.position);//Ray starts at this gameobject position

    //while the last point of the ray is not close enough
    while (Vector3.Distance(lastPoint, r.enemy.transform.position) > 0.5f) {

        r.line.positionCount = i + 1;

        //Get direction from the last point of the ray to the enemy
        Vector3 fwd = Vector3.Normalize(r.enemy.transform.position - lastPoint);

        // Get deviation point
        fwd = RandomDeviation(fwd, deviation);

        // Next point of the ray is lastPoint + desviation
        fwd += lastPoint;
        r.line.SetPosition(i, fwd);
        i++;
        lastPoint = fwd;
    }
    r.line.positionCount = i + 1;
    r.line.SetPosition(i, enemyPos);
}
```

Figura 16: Fragment de codi per la generació dels rajos.

Inicialment el factor de desviació és 0 i, per tant, es mostra una línia recta, però amb el temps aquest valor es va incrementant. A més, com que a cada frame es tornen a calcular aquests punts es genera un efecte similar al d'un raig.

Pel que fa a l'obtenció dels enemics que es troben dins del radi d'atac, s'ha utilitzat l'operació **Physics2D.OverlapCircleAll** amb un filtre a la capa **Enemy** perquè només retorni els *colliders* que són dels enemics. Així doncs, cada enemic en el joc s'ha assignat en aquesta capa.

Figura 17: Conjur pluja de rajos.

Un cop es llança el conjur es genera l'esdeveniment **spellCasted** que notifica tant al gestor de conjurs com a certs elements de la interfície gràfica que s'ha llençat un conjur determinat.

3.7 Apuntador

Durant el joc el jugador haurà de disparar a llocs concrets per activar certs mecanismes. Per aquest motiu, he considerat que era necessari implementar algun sistema d'apuntador per tal de facilitar aquesta tasca al jugador.

Així doncs, quan el jugador està quiet, té seleccionat el conjur raig d'energia i manté polsat l'espai, es mostrarà una línia en la direcció on es dispararà el projectil. Aquest apuntador es pot moure mitjançant les tecles de direcció i també interactua amb els miralls. La gestió d'aquest apuntador es realitza quan el jugador està en l'estat inactiu, és a dir, en la classe **IdleState**.

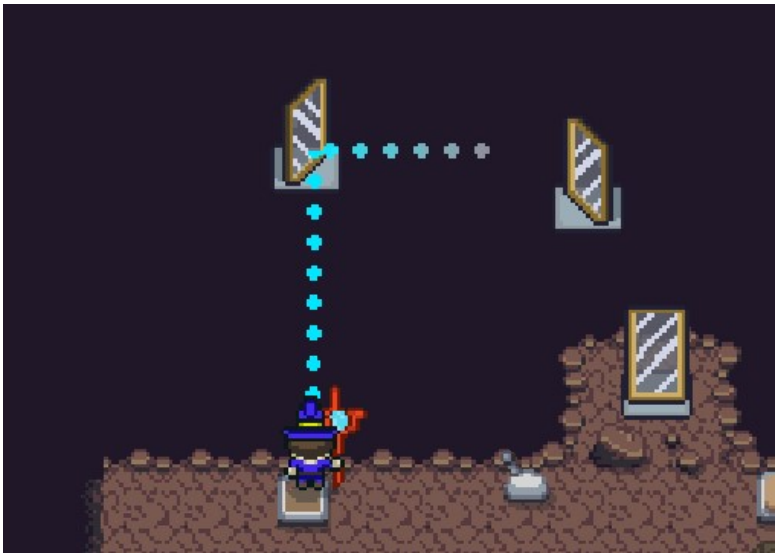


Figura 18: Apuntador

3.8 Enemics

De manera similar al jugador, per implementar els enemics, primer he hagut d'obtenir els *sprites-sheets* amb les seves animacions. N'hi ha que els he pogut utilitzar directament, d'altres que s'hi han afegit animacions per tal de tenir els moviments en les quatre direccions. i, en alguns casos, he utilitzat la mateixa plantilla que el jugador per realitzar-los o que directament els he creat de zero.

Figura 19: Enemics

D'altra banda, en el joc es poden distingir dos tipus d'enemics, els enemics normals que es troba el jugador mentre cerca les gemmes i el *boss* que es troba al nivell final i que interessa que generi uns esdeveniments determinats. Per aquest motiu, he afegit les subclasses **AbstractEnemy**, **Enemy** i **Boss** que implementen la funcionalitat compartida en cada cas.

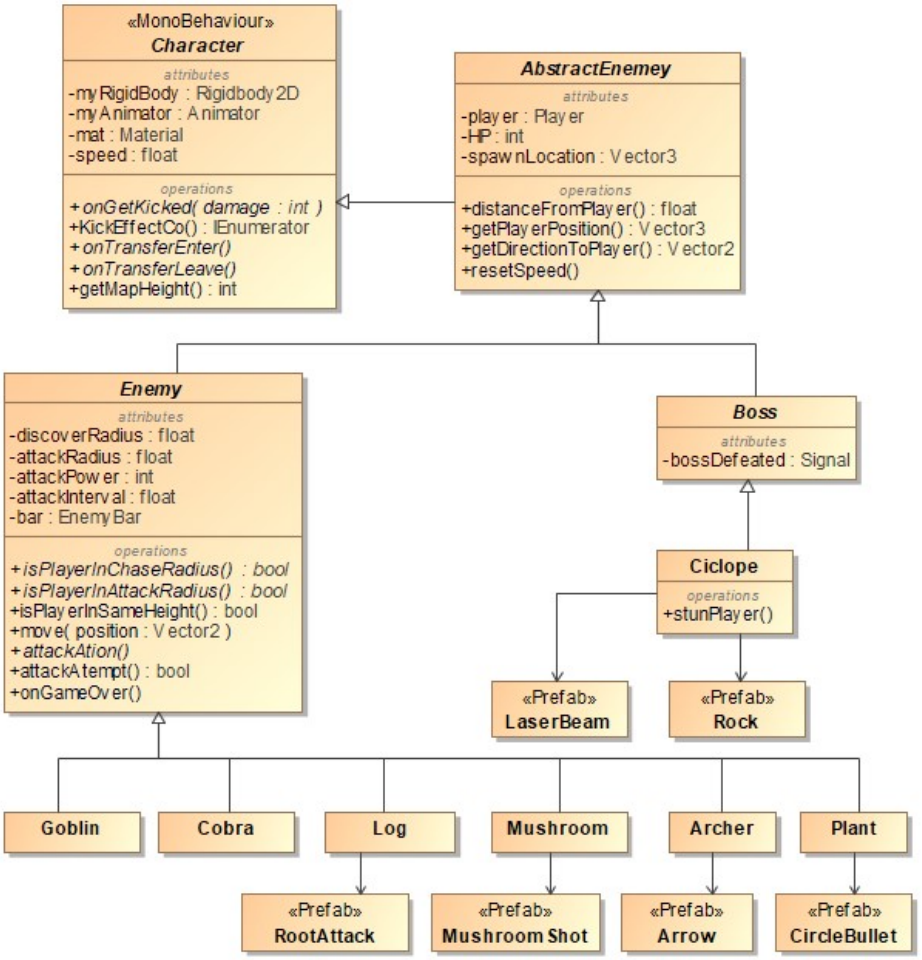


Figura 20: Diagrama de classe dels enemics

Per implementar el comportament de la majoria d'enemics vaig decidir utilitzar el sistema de màquina d'estat que proporciona el component **Animator**. Per cada estat s'ha definit un *Blend Tree* amb l'animació que ha de tenir l'enemic en aquell estat i s'hi ha afegit també un *script* subclasse de **StateMachineBehaviour** que implementa la lògica del comportament d'aquell estat. Com que el comportament dels enemics es similar, hi ha implementacions d'estats que s'han pogut reutilitzar.

Mentre realitzava proves amb els enemics em vaig adonar que seria necessari afegir a cada un d'ells una barra que indiqués la vida que li quedava, ja que sinó era molt difícil saber quan li quedava a l'enemic per morir i, a més, tampoc es veia la quantitat de dany que li causava el jugador.

A continuació es comentaran les tasques realitzades per cada enemic:

- **Goblin**: S'ha dissenyat gràficament utilitzant la mateixa base que el jugador. Pel que fa a l'atac amb l'espasa s'ha utilitzat el mateix sistema que en l'atac amb la vara del jugador, és a dir, un objecte amb un *collider* que s'activa en el moment de l'animació d'atac.

  Pel que fa al seu comportament, s'han definit dos radis, el radi d'atac i el radi de persecució. Quan el jugador es troba dins del radi de persecució es mou en direcció al jugador i quan es troba dins del radi d'atac l'ataca. Si el jugador es troba fora d'aquest dos radis està quiet. El comportament d'aquests estats s'han definit en les classes **IdleBehaviour**,**ChaseBehaviour** i **AttackGoblinS.**

- **Cobra**: Pel que fa al disseny gràfic, s'ha obtingut l'animació d'una cobra en dues direccions (dreta i esquerra) i s'ha afegit les animacions necessàries per obtenir la resta (amunt i avall). L'atac s'ha realitzat utilitzant un *collider* com en el cas del goblin.

  El comportament d'aquest enemic és el mateix que el del goblin. L'única petita diferència és que l'atac es realitza en intervals de temps i no de manera fixa com en el cas anterior. D'aquesta manera pel comportament de la cobra s'ha utilitzat les classes **IdleBehaviour**, **ChaseBehaviour** i **AttackCobra**.

- **Bolet**: S'ha obtingut l'animació d'un bolet que camina en dues direccions (dreta i esquerra) i s'han creat les animacions per a la resta de direccions. També s'han creat les animacions utilitzades en el moment de llançar el projectil i del projectil que llança.

  Per aquest enemic s'ha implementat un projectil lineal anomenat **MushroomShot** el qual se li ha afegit un sistema de partícules per a l'estela del projectil i un altre que s'executa en el moment d'explotar.

  Pel que fa al seu comportament, és similar a la resta, la diferència es troba en què en l'animació d'atac no s'ha afegit cap *StateMachineBehaviour*, sinó que en un frame de l'animació d'atac es genera l'esdeveniment que crea i dispara el projectil. Això s'ha realitzat així per sincronitzar el moment exacte en el que ha de disparar.

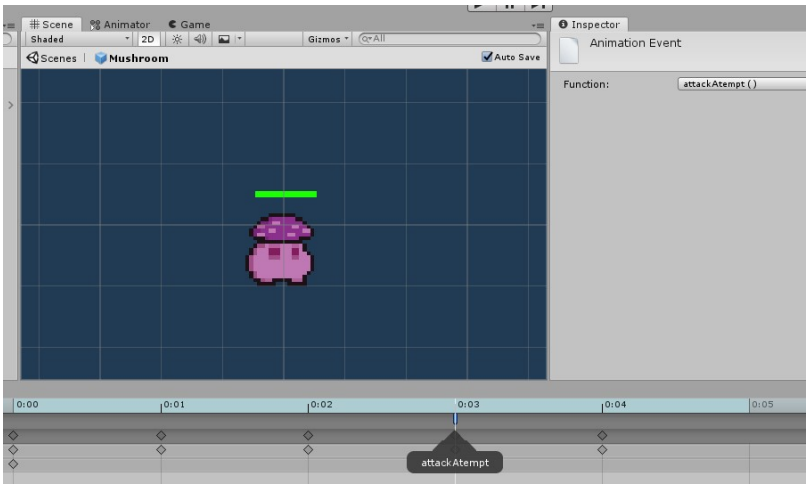


Figura 21: Esdeveniment en l'animació del bolet per disparar

- **Arbre**: S'han utilitzat directament les animacions realitzades pel mateix creador de la base del personatge per implementar l'arbre. D'altra banda, s'han hagut de crear les animacions pel seu atac que consisteix en fer aparèixer una arrel amb punxes.

  La lògica dels estats que segueix aquest enemic es similar als anteriors però diferent, ja que, a part de perseguir el jugador, ha de despertar-se o adormir-se i l'animació d'atac la realitza l'objecte que implementa l'arrel. Per aquest motiu, no s'han reutilitzat els estats creats pels enemics comentats anteriorment i se n'han creat de nous: **SleepingLog**, **IdleLog**, **ChaseLog**.

  Pel que fa a la implementació de l'atac amb l'arrel, s'ha creat un objecte *prefab* que mostra l'animació de l'atac amb l'arrel i activa un *collider* que al tocar el jugador li resta vida. En l'operació d'atacar s'especifica el punt on es troba el jugador i es fa que l'arrel aparegui en un punt aleatori situat al voltant d'aquest punt. Això s'ha realitzat així per tal que l'arrel no toqui sempre el jugador.

- **Arquer**: S'ha pres com a partida el disseny del goblin i s'ha modificat perquè utilitzés un arc. A més, per fer més realista la fletxa, se n'ha buscat una que tingués animacions per a les setze direccions.

  Pel que fa al comportament de l'arquer, es va decidir que l'acció de patrullar consistís en fer moure l'arquer de manera que visités una sèrie de punts. Per fer més imprevisible els seus moviments, es va decidir fer que el següent punt a visitar per l'arquer fos decidit de manera aleatòria. La lògica dels seus estats es troba en les classes **patrolGoblinA**, **chaseGoblinA** i **shotGoblinA**.

  Tal i com s'ha comentat anteriorment, la fletxa de l'arquer s'ha implementat com un projectil parabòlic.

- **Planta**: S'han creat les animacions de la planta i del tipus de projectil que dispara. A diferència de la resta d'enemics, no s'ha utilitzat el component **Animator** pels estats de la planta. El comportament de la planta s'ha implementat com una màquina d'estat en l'*script* **Plant**. Per implementar les seves accions s'ha fet us de corrutines.

  S'ha definit l'*enum* **PlantState** pels estats amb els que pot estar la planta (IDLE,ATTACKING,TELEPORTING i GAME_OVER) i les corrutines **TeleportCo**() i **PlantShotCo**() que realitzen els moviments de teletransportar i de disparar respectivament.

  També s'han definit les variables per als intervals de temps en que es pot realitzar una acció determinada i el radi en el que s'ha de trobar el jugador perquè es pugui realitzar l'acció. En la funció **Update()** es porta un control dels intervals de temps per a les dues accions i de l'estat en que es troba la planta per determinar quina ha de ser l'acció a realitzar.

3.9 Boss

El ciclop s'ha implementat en dues parts. En la primera s'han implementat els seus atacs i la segona s'ha dissenyat una màquina d'estat que combini el moviment del ciclop per la zona amb els aquests atacs.

L'atac de tirar una pedra s'ha implementat de manera similar a la fletxa, ja que es tracta també d'un moviment parabòlic. En aquest cas no era necessari obtenir la direcció de la corba en cada punt, fent rotar la pedra mentre seguia la trajectòria era suficient per aconseguir un bon efecte visual.

Per l'acció d'atordir el jugador, s'ha afegit la corrutina **stunCo()** a la classe **Player** que posa a cert durant uns instants la variable booleana **freeze**. Aquesta variable desactiva el control del teclat i les accions de l'estat en el que està. Per visualitzar que el jugador ha quedat atordit es mostren també unes estrelles que giren sobre el seu cap. Per reforçar més aquest efecte s'ha implementat en la càmera la corrutina **shakeCo()** que, durant uns instants, va movent la càmera de la seva posició inicial de manera aleatòria i proporciona un efecte que dóna la sensació que el terra tremola.

L'últim atac implementat en el ciclop es tracta del raig làser. Per implementar-lo, s'ha considerat que la forma més senzilla era utilitzar un **LineRenderer** per mostrar visualment on era el làser i l'operació **Physics2D.CircleCast** per determinar on es produeixen les col·lisions.

A grans trets, el raig làser consisteix en generar primer una línia entre el punt on es troba el qui la dispara i el punt on s'apunta. Seguidament, es va desplaçant la direcció d'aquesta línia amb el temps seguint la funció

cosinus i un angle màxim definit. Aquest comportament està implementat en la corrutina **moveCo()** de la classe **LaserBeam**. En el cas de detectar una col·lisió, s'ha fet aparèixer també una bola del mateix color del làser amb un sistema de partícules afegit per visualitzar millor on s'ha produït.

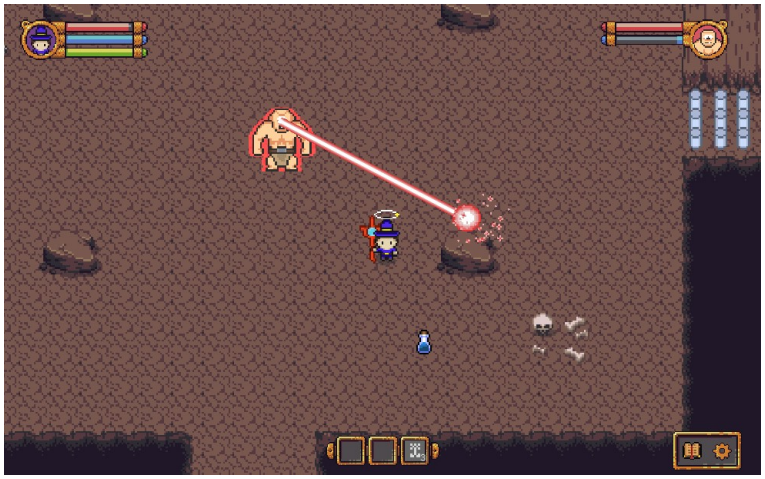


Figura 22: Atac làser del ciclop

En el moment d'implementar el ciclop, es va definir també de manera exacta quin havia de ser el seu comportament, ja que en la fase de disseny només s'havien descrit com havien de ser els seus atacs, però no per quina zona es mouria.

Finalment, es va decidir fer que el ciclop patrullés pel centre de la zona on es produeix la lluita amb el jugador seguint un cercle i tirant-li pedres quan aquest es troba a una certa distància i també atordint-lo a cada cert interval de temps. L'atac amb làser es va decidir realitzar com un atac especial quan el ciclop tingués una barra d'energia carregada al màxim.

Com en la resta d'enemics, s'ha utilitzat el component **Animator** i **StateMachineBehaviours** per implementar el comportament del ciclop.

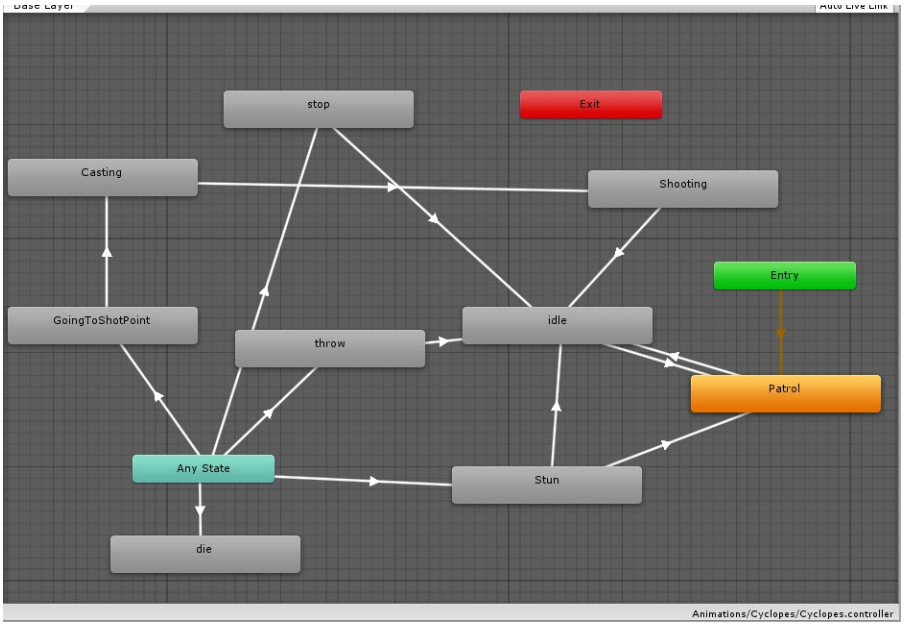


Figura 23: Component Animator del ciclop

A diferència de la resta d'enemics, per la vida del ciclop i la seva energia s'han utilitzat variables compartides del tipus **ObservaleInteger** i **ObservableFloat** que s'utilitzen per mostrar visualment en la interfície gràfica el nivell de vida i de mana del monstre. En morir es genera l'esdeveniment **onBossDefeated** que és utilitzat per fer abaixar una pilona i reproduir un so que indica al jugador que ha guanyat el combat.

En l'aspecte gràfic, s'han obtingut les animacions d'un ciclop en dues direccions (dreta i esquerra) i s'ha creat la resta (amunt i avall).

3.10 Objectes

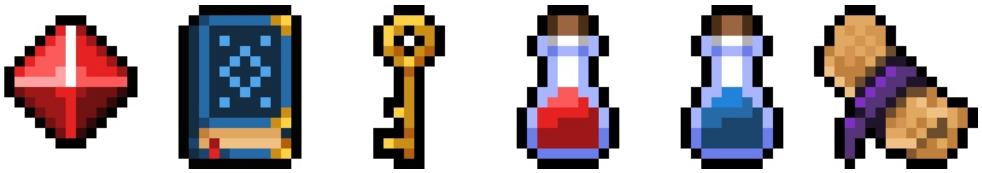


Figura 24: Objectes que pot recollir el jugador en el joc

Pels objectes que pot recollir el jugador en el joc s'ha definit una classe abstracte anomenada **Item,** que implementa el comportament genèric i delega a les subclasses la implementació de l'acció concreta a realitzar quan el jugador recull l'objecte. D'aquesta manera, la majoria d'objectes només han d'implementar l'operació **onItemCollect()** i tenir un collider de tipus *trigger* que pugui detectar les col·lisions amb el jugador. S'ha decidit que quan el jugador passa per sobre de l'objecte automàticament el recull.

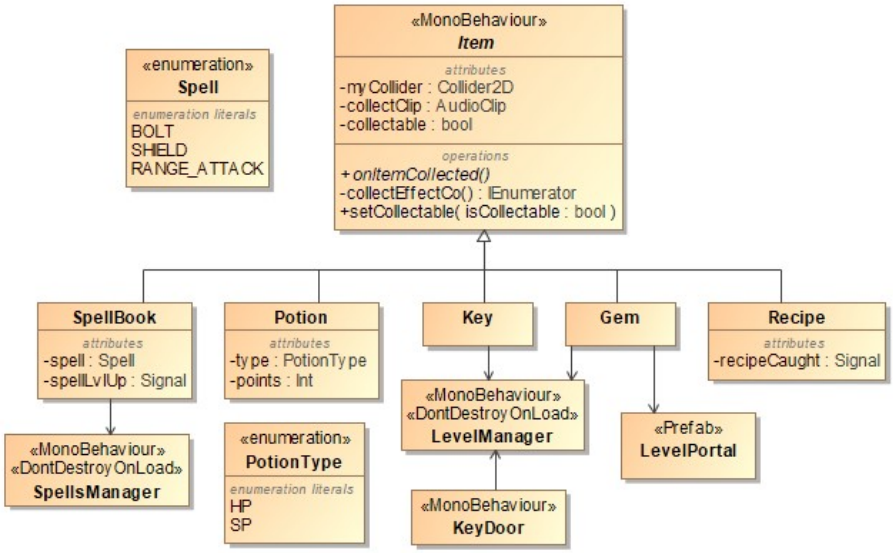


Figura 25: Classes de la implementació dels objectes

Passem a repassar la implementació de cada objecte:

- **Gemma**: En el moment de carregar-se l'escena, notifica al gestor del nivell (LevelManager) la seva presència per tal que aquest pugui actualitzar el comptador. Quan el jugador recull l'objecte notifica al gestor del nivell que s'ha recollit una gemma perquè actualitzi el comptador.

  Cada gemma defineix un punt en el mapa on ha de sortir el portal. Quan el jugador agafa l'última gemma es crea un portal en aquell punt. Al travessar el portal el jugador passa al nivell final.

Figura 26: Portal per passar al següent nivell

- **Llibre de conjurs**: Se n'ha implementat un per cada tipus de conjur que pot realitzar el jugador. Quan l'objecte és recollit es crida una funció del gestor de conjurs perquè pugi de nivell el conjur.

- **Clau**: Notifica en el gestor del nivell que s'ha agafat una clau. Aquest, a la vegada, genera un esdeveniment que és utilitzat en

la interfície gràfica per mostrar una clau en un panell. Aquest objecte forma part del mecanisme clau-porta.

Pel que fa a la porta, té un *collider* tipus *trigger* que quan el jugador hi entra en contacte es comprova en el gestor del nivell si té la clau. Si la té, es mostra una animació de la porta obrint-se i s'activa un portal que permet accedir a una altra zona.

- **Pocions**: Al ser recollides, obtenen una referència al component **Player** del jugador i criden l'operació **onCollectPotion**() passant com a paràmetre el tipus de poció i els punts que restitueix.

- **Recepta**: És l'últim objecte que recull el jugador i serveix per passar-se el joc. En el moment de recollir-se es genera un esdeveniment que fa reproduir el *timeline* amb l'escena final del joc.

3.11 Contenidors d'objectes

En aquest cas, es tractava d'implementar un element que tirés objectes quan el jugador el colpejava amb la vara o quan li disparava un projectil. Com que la funcionalitat havia de ser la mateixa pels dos tipus de contenidors previstos, s'ha creat una classe abstracta anomenada **ItemContainer** amb la lògica genèrica encarregada de detectar quan es produeix alguna de les col·lisions comentades. Aquesta delega en les subclasses la implementació de l'operació **open()** perquè iniciï el procés per tirar-los i la **getThrowPoint()** que ha de retornar la posició on s'ha de llançar l'objecte.

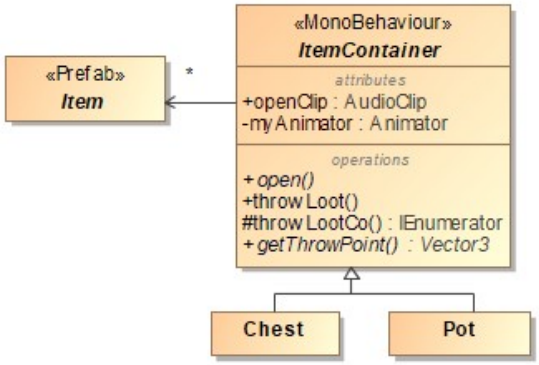


Figura 27: Classes dels contenidors d'objectes

Tant en el gerro com en el baül, en l'operació open() utilitzen el component **Animator** perquè mostri una animació amb un esdeveniment associat que indica el frame exacte en el que s'han de tirar els objectes.

3.12 Zones i transicions

Un nivell pot estar format per diverses zones o àrees tancades a les que el jugador pot accedir, per exemple, una cova, un bosc, etc. i per passar, d'una zona a una altre, es realitzen transicions.

Hi havia la possibilitat de repartir aquestes zones en diferents **Scenes** d'Unity o posar-les totes en una. Després d'avaluar les dues opcions, es va considerar que l'opció més senzilla era que totes les zones estesin en una mateixa *Scene*, ja que d'aquesta manera resultava més fàcil comptar totes les gemmes que hi ha en un mateix nivell.

Cada zona té un *tilemap* associat amb 5 capes: **Background**, **BackgroundDeco**, **WalkCollision**,**Obstacles** i **Top**. Les dues primeres i la última són simplement per decorar l'espai on es mourà el jugador. En canvi, la tercera i la quarta, a part de decorar, també indiquen les zones estàtiques on es produiran les col·lisions amb el jugador o els enemics. Per aconseguir-ho s'ha utilitzat en cadascuna d'aquestes dues capes la combinació dels components d'Unity **RigidBody2D**, **TileMapCollider2D** i **CompositeCollider2D**.

La diferència entre WalkCollision i Obstacles està en el fet que la primera permet que els projectils lineals disparats en la mateixa alçada la travessin i, en canvi, en la d'obstacles no. Per implementar aquesta funcionalitat s'ha utilitzat una detecció de col·lisions basada en capes. S'ha assignat a WalkCollision la capa de col·lisions Background i a Obstacles la capa de col·lisions Wall. Els projectils assignats a la capa Bullet no col·lisionen amb Background però si amb la capa Wall.

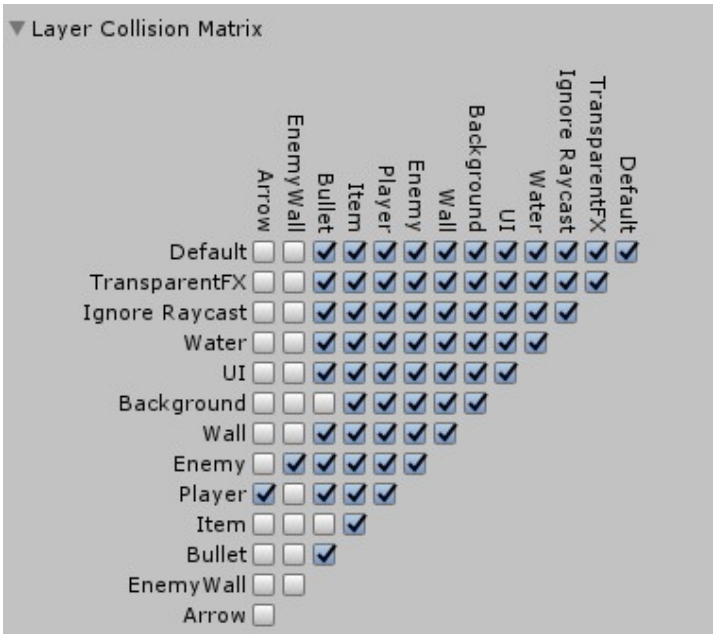


Figura 28: Matriu de les col·lisions per capes

Per a cada zona també s'han definit els límits els quals no ha de traspassar la càmera. Els límits es defineixen en un objecte present a cada zona anomenat **Bounds**. Bàsicament aquest objecte defineix el rectangle format pels punts superior esquerre i inferior dret de la zona.

Pel que fa a les transicions, s'ha creat un objecte portal amb un *collider* tipus *trigger* i s'ha creat un script anomenat **PortalTransfer** que gestiona les transicions entre zones. Cada portal té una referència al portal on ha de transportar el jugador i genera els esdeveniments **transferEnter** i **transferLeave** per indicar que comença o finalitza una transició.

S'ha definit una distribució per a la jerarquia d'objectes en una zona de manera que cada portal es troba situat a **nomZona/portals/portalX** i l'objecte encarregat que gestiona els límits de la zona a **nomZona/bounds**.

En el moment de realitzar una transició, el portal obté de l'altre la posició on ha d'aparèixer el jugador i els límits de la zona on es troba l'altre portal. Amb aquesta informació mou el jugador i la càmera a la posició indicada i també n'actualitza els límits.

Perquè tot aquests moviments no es vegin s'ha afegit un panell transparent a la interfície gràfica que captura els senyals transferEnter i transferLeave. En rebre el primer, el panell s'enfosqueix fins a tornar-se negre i en rebre el segon es torna transparent.

Per evitar que el jugador o els enemics es moguin durant la transició, també capturen aquests senyals i es realitzen les tasques pertinents perquè estiguin quiets.

3.13 Càmera

La càmera és un element bastant simple, però alhora molt important, ja que és el que ens permet visualitzar el què succeeix en el joc. En el joc n'apareixen dues. La primera es troba únicament en el menú principal i té com a únic objectiu que es pugui visualitzar el menú principal.

En canvi, a la resta de nivells hi ha una càmera que segueix el jugador per la pantalla però no es permet que surti dels límits de la zona.

Inicialment, la càmera obté del gestor del nivell els límits inicials (Bounds) de la zona on comença el nivell el jugador. Quan el jugador realitza una transició entre zones, s'actualitzen aquests límits en la càmera.

En la càmera utilitzada en els nivells també s'ha afegit una corrutina per donar un efecte similar a la tremolor del terra. Aquest efecte l'utilitza el ciclop en l'acció d'atordir el jugador.

3.14 Sistema d'alçades

Com que en *l'sprites-sheet* obtingut per dissenyar l'entorn del joc es podien crear zones amb diferents alçades, vaig considerar implementar un sistema per evitar que disparant des d'un punt visualment més baix, es toqués un punt que visualment hauria d'estar situat a un altra alçada o nivell i a l'inrevés. Aquest sistema només havia de ser per als projectils que es disparen en línia recta, ja que en els que tenen un moviment parabòlic, ja hi ha un cert efecte d'alçada en el moviment del projectil.

El sistema implementat consisteix en afegir un ***Tilemap*** al nivell dedicat a les alçades i pintar cada cel·la amb un *sprite* diferent segons l'alçada. Aleshores només cal saber amb quin *sprite* s'ha pintat la cel·la per obtenir-ne l'alçada.

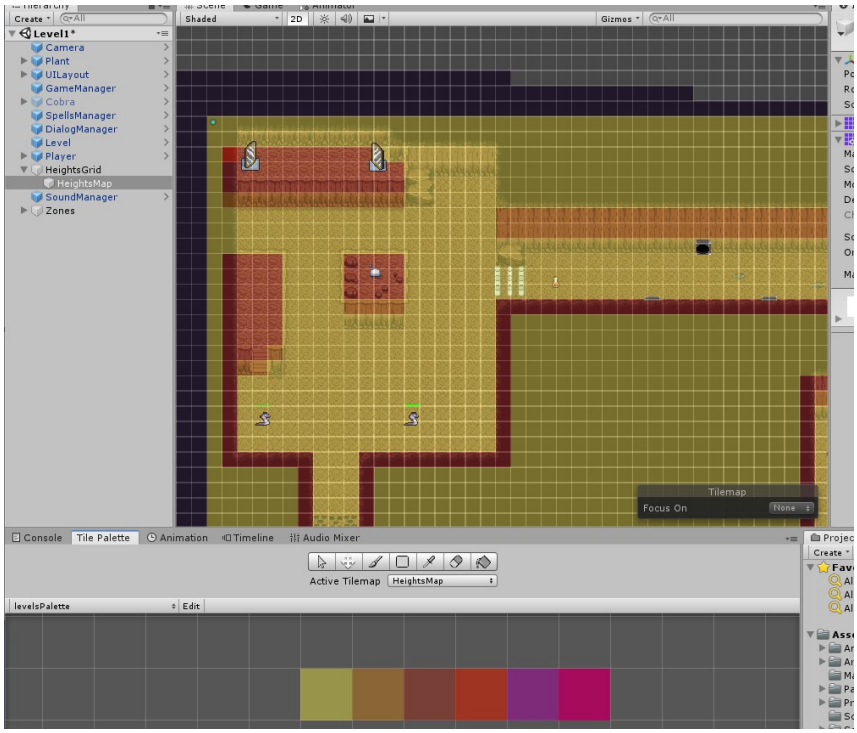


Figura 29: Tilemap per al sistema d'alçades

Els *sprites* generats des d'un *sprites-sheet* en Unity tenen el nom del *sprites-sheet* més un índex numèric que comença en 0 i s'incrementa en 1 a cada *sprite.* Per tant, si es pinta adequadament el *tilemap* seguint aquest mateix ordre, obtenir l'alçada serà equivalent a obtenir l'índex del *sprite* que hi ha en una determinada posició. Comentar també que aquest *tilemap* es troba en una capa inferior a la resta, per la qual cosa és invisible per al jugador. Només s'ha posat a una capa superior per poder editar els nivells.

```
public int getTileLevel(Vector3 worldPosition) {

    Tile tile = heightsMap.GetTile<Tile>(heightsMap.WorldToCell(worldPosition));
    if (tile == null) {
        return 999;
    }

    Regex regex = new Regex(@"\d+");
    Match m = regex.Match(tile.sprite.name);
    return int.Parse(m.Value);
}
```

Figura 30: Fragment de codi pel càlcul de les alçades

Amb aquest sistema cal obtenir l'alçada del punt on es troba el jugador o l'enemic abans de disparar el projectil. També cal comprovar en cada frame el nivell en que es troba el projectil. Si el nivell de la posició on es troba en aquell instant és superior al que hi havia quan s'ha disparat, cal forçar un impacte.

```
private void Update() {
    if (!collided) {

        if (LevelManager.Instance.getTileLevel(transform.position) > shotHeight) {
            collided = true;
            SoundManager.Instance.playEffect(bulletHitClip);
            onCollision(transform.position);
        }
    }
}
```

Figura 31: Fragment de codi pel càlcul de l'alçada del projectil

3.15 Diàlegs i senyals

Tal i com estava previst en la descripció del joc, en el moment d'iniciar la partida, s'ha de mostrar un text o escena introductòria que posi el jugador a la pell del personatge. Vaig pensar que una bona manera d'introduir el personatge al joc era realitzant una conversa entre el jove bruixot i el seu futur mestre. Així doncs, vaig decidir implementar un petit sistema per a les converses.

L'element bàsic d'una conversa és un missatge. De cada missatge en volem saber l'emissor i també una petita imatge que identifiqui més fàcilment qui és. Per aquest motiu, s'ha creat la classe **DialogMessage** que conté les dades d'un missatge.

```
[System.Serializable]
7 references
public class DialogMessage{

    public string charName;
    [TextArea(3, 10)]
    public string message;
    public Sprite charImage;
}
```

Figura 32: Classe DialogMessage

També s'ha dissenyat un element per a la interfície gràfica anomenat **DialogBox** que donat un vector de **DialogMessage** mostra per ordre els missatges de la conversa.

Figura 33: Diàleg en el joc

Per mostrar el text dels missatges de la conversa s'ha utilitzat un sistema vist en altres jocs en el qual el text es va mostrant amb el temps, per donar més la sensació que el personatge està parlant. Aquest efecte també s'ha reforçat amb so. Per confirmar que el jugador ha llegit el missatge, fins que no prem l'espai no es mostra el següent.

Per mostrar els diàlegs s'ha implementat un objecte del tipus **DontDestroyOnLoad** anomenat **DialogManager** que s'encarrega d'instanciar els diàlegs i també els missatges dels senyals.

En la generació dels diàlegs es genera un esdeveniment **onDialogEnter** i en finalitzar el diàleg es genera l'esdeveniment **onDialogLeave**. Aquests esdeveniments són utilitzats en el jugador per deixar-lo immòbil mentre dura la conversa.

Un cop implementat aquest sistema de converses, també s'ha utilitzat per realitzar una conversa prèvia a la lluita amb l'enemic final i quan el jugador es passa el joc.

3.16 Escenes

En el joc s'han implementat tres escenes: la primera al principi del joc per introduir el personatge, la segona en trobar-se l'enemic final i la tercera quan el jugador s'ha passat el joc.

Aquestes escenes s'han implementat mitjançant **Timelines** d'Unity i un objecte amb un component **PlayableDirector** assignat. Val a dir que la implementació d'aquestes escenes utilitzant aquests elements no ha estat tant senzilla com s'esperava, ja que s'ha hagut de trobar la manera de sortejar una sèrie de *bugs* inesperats que retornaven al jugador a la posició inicial en el moment de finalitzar o posar en pausa la reproducció del *timeline*. A més, quan finalitzava l'animació l'objecte del jugador quedava encallat i no es movia.

Per solucionar aquests problemes s'ha implementat el component de tipus *script* **TimelineDirector** que s'encarrega d'iniciar, pausar i continuar l'animació.

Bàsicament, es necessita posar en pausa l'animació quan s'inicia un diàleg i continuar-la quan aquest finalitza. Per aquest motiu, s'ha aplicat el sistema de senyals que genera les converses per pausar l'animació quan es genera l'esdeveniment **onDialogEnter** i de continuar-la quan es produeix l'esdeveniment **onDialogLeave.** Per pausar l'animació s'ha modificat la velocitat de reproducció de l'animació mitjançant l'operació **playableGraph.GetRootPlayable(0).SetSpeed(0)** i per continuar-la la mateixa operació però amb el valor 1 a SetSpeed()**.**

Pel que fa a la solució per desencallar el jugador, ha consistit en desactivar temporalment el controlador del component **Animator** del jugador abans d'iniciar un *timeline* i de tornar-lo a activar quan finalitza.

3.17 Mecanismes

S'ha implementat un sistema genèric per als mecanismes que aplica el patró observador. Un mecanisme consisteix en un objecte que conté objectes activadors i objectes activables. En el moment de carregar l'escena cada activador i cada activable es registra en el mecanisme. Aleshores, quan un activador canvia d'estat ho notifica al mecanisme i aquest calcula quin és l'estat general del mecanisme i activa o desactiva els objectes activables que té registrats.

- **Activables:**
  - **Pilona retràctil:** En l'estat desactivat té un *collider* que impedeix al jugador travessar-lo i quan passa a estar actiu aquest *collider* es desactiva.
  - **Pont màgic:** De manera similar a la pilona, quan està desactivat és invisible per al jugador i té un *collider* a cada extrem del pont que impedeix al jugador travessar-lo. Quan s'activa es torna

visible i s'activen dos *colliders* als laterals del pont per evitar que el jugador surti per un lloc inesperat.

- **Pausador:** És un component utilitzat per pausar el moviment dels miralls que apareix en una de les coves. En si mateix no té representació però la seva lògica és exactament la mateixa que la resta.

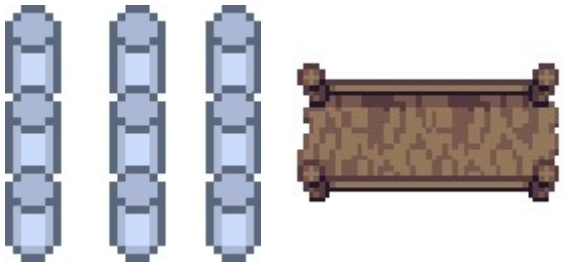


Figura 34: Elements activables en el joc

Per realitzar les animacions i l'activació/desactivació dels *colliders* s'han utilitzat **AnimationClips** i el component **Animator** d'Unity. S'ha definit una variable de tipus booleà anomenada **activate** que al canviar de valor passa l'objecte d'un estat a un altre.

Activadors:

- **Botó al terra**: Consisteix en un objecte que conté un *collider* de tipus *trigger* a la zona on es pot prémer el botó. Per saber quan cal activar o desactivar el botó, s'ha implementat un sistema que compta el nombre d'objectes que hi ha sobre el botó utilitzant les operacions **OnTriggerEnter2D** i **OnTriggerExit2D**.

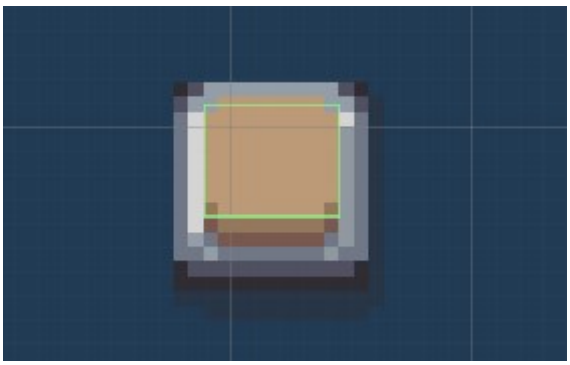


Figura 35: Botó del terra amb el seu collider

```
private void OnTriggerEnter2D(Collider2D other) {

    if (other.tag == "Staff")
        return;

    objectsAbove++;
    state = true;
    myAnimator.SetBool("activate", true);
    mechanism.notifyStatusChange(this);
}
0 references
private void OnTriggerExit2D(Collider2D other) {

    objectsAbove--;

    if(objectsAbove <= 0) { //Desactivate

        state = false;
        myAnimator.SetBool("activate", false);
        mechanism.notifyStatusChange(this);
    }
}
```

Figura 36: Fragment de codi per a l'activació d'un botó

- **Palanca**: A diferència del botó al terra, per implementar la palanca han estat necessaris dos *colliders*. El primer evita que el jugador pugui passar per sobre i també comprova les col·lisions amb els projectils. El segon és de tipus *trigger* i comprova si s'ha produït una col·lisió amb la vara del jugador. Quan es produeix algun d'aquests dos tipus de col·lisions la palanca canvia d'estat.

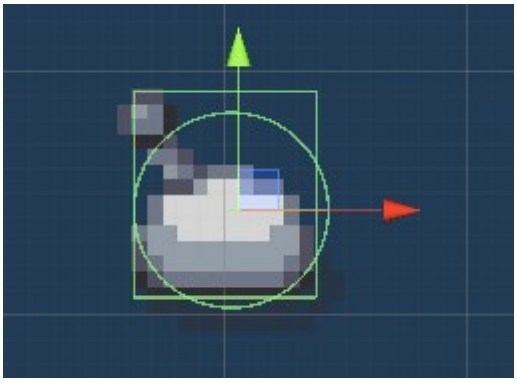


Figura 37: Palanca amb els seus colliders

- **Marcador:** Aquest element està format per diversos objectes seguint una jerarquia determinada. L'objecte arrel conté el component **EnemiesCounter** que s'encarrega de comptar els enemics que s'han de matar i d'actualitzar el marcador.

  La resta d'objectes que descendeixen de l'arrel són els monstres que s'han de matar i l'objecte anomenat **Counter** que mostra en un marcador aquesta informació.

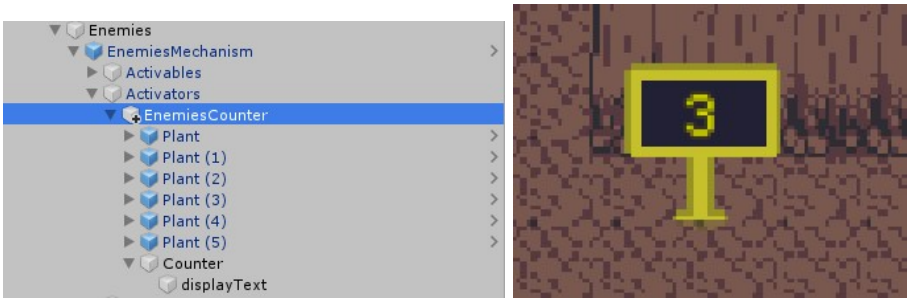


Figura 38: Marcador i la seva jerarquia d'objectes.

La comprovació dels enemics que queden per matar es realitza dins la funció **Update** del component **EnemiesCounter**.

```
protected override void  Start() {
    base.Start();
    enemiesCurrent = transform.GetComponentsInChildren<AbstractEnemy>().Length;
    counter = transform.GetComponentInChildren<Counter>();
    counter.setText(enemiesCurrent.ToString());
}

0 references
void Update() {

    int current = transform.GetComponentsInChildren<AbstractEnemy>().Length;
    if( current != enemiesCurrent) {
        enemiesCurrent = current;
        counter.setText(enemiesCurrent.ToString());
    }

    if(enemiesCurrent == 0 && !notified) {
        state = true;
        mechanism.notifyStatusChange(this);
    }
}
```

Figura 39: Fragment de codi per la gestió del marcador.

3.18 Complements per als mecanismes

S'han implementat en el joc dos elements que ha d'utilitzar el jugador per activar algun mecanisme:

- **Miralls:** S'han implementat mitjançant un objecte pare amb un *collider* que segueix el contorn del marc i un objecte fill amb un *collider* de tipus *trigger* que segueix el contorn del vidre. El de l'objecte pare s'encarrega de les col·lisions amb el jugador i el de l'objecte fill és l'encarregat de modificar la direcció del projectil.

  Perquè l'objecte fill pugui detectar les col·lisions, primer cal que l'objecte pare ignori les col·lisions amb els projectils. Això s'ha aconseguit utilitzant un sistema de col·lisions basat en capes. En aquest cas, s'ha assignat a l'objecte pare la capa **Background** (no col·lisiona amb la capa Bullet) i s'ha deixat l'objecte fill a la **Default**.

  Per aconseguir l'efecte de reflexió del mirall en els projectils s'ha assignat, en cada orientació del mirall, el vector corresponent a la seva normal. Aleshores, per desviar la direcció del projectil només ha calgut aplicar la funció **Vector2.Reflect** amb la normal del mirall i la direcció original del projectil per obtenir la direcció en la que hauria de sortir reflectit.

Figura 40: Projectil llançat pel jugador i desviat per un mirall

- **Estàtues**: La funcionalitat que el jugador pugui empènyer l'estàtua ha estat força senzilla d'implementar. Només ha calgut afegir un component **RigidBody2D** de tipus ***dynamic*** a l'objecte de l'estàtua amb una massa associada més gran que la del jugador, i un **collider** per rebre les col·lisions amb el jugador. El motor de la física d'Unity ha fet la resta. Únicament ha calgut reiniciar la velocitat a zero quan el jugador deixava d'estar en contacte amb l'estàtua, per evitar que continués lliscant.

  També ha calgut trobar algun mecanisme per evitar que l'estàtua quedés atrapada en alguna de les parets del joc, ja que si això succeïa no hi havia manera que el jugador pogués passar-se el nivell. La solució triada ha consistit en implementar un botó vermell similar al que s'ha utilitzat per activar mecanismes. En aquest cas, el botó retorna les estàtues a la seva posició inicial quan el jugador s'hi posa a sobre. Aquest botó també serveix per reiniciar els mecanismes.

Figura 41: Botó vermell per reiniciar estàtues

3.19 Trampes

Figura 42: Conjunt de trampes que apareixen en el joc

Mentre estava desenvolupant el joc vaig pensar que a part dels enemics, el jugador també podia fer front a altres perills. Per aquest motiu, vaig decidir implementar una sèrie de trampes i així donar més varietat en el joc. A continuació es comentaran les trampes implementades:

- **Trampa de l'ós**: Es tracta d'un objecte amb un *collider* tipus *trigger* que detecta quan el jugador passa per sobre seu. Quan es dóna aquest cas, es mostra l'animació que es tanca la trampa i es crida l'operació **OnGetKicked** en el jugador.

- **Tronc amb punxes**: Es tracta d'un objecte amb un *collider* que es mou en vertical o horitzontal una distància determinada i després canvia de sentit. Per una banda, està compost per l'script **SpineTrunk** que s'encarrega de gestionar les col·lisions amb el jugador i treure-li vida.

  Per l'altra, s'ha desenvolupat un component genèric anomenat **MovingElments** per poder moure elements amb aquest comportament. Per implementar-lo, s'han definit *l'enum* **MovingDirection** per indicar en cada cas si el moviment ha de ser vertical o horitzontal i tota la lògica necessària per poder canviar, segons les necessitats, el sentit inicial del moviment i la seva velocitat. Per aconseguir que el moviment sigui d'anada i tornada s'ha utilitzat l'operació **Mathf.PingPong**.

  Finalment, comentar també que el disseny gràfic d'aquesta animació ha estat creat expressament per aquest joc.

- **Punxes a terra**: Es tracta d'unes punxes que hi ha al terra i que surten cada cert interval de temps. Per implementar aquesta trampa ha estat necessari sincronitzar el moment en que apareixen les punxes amb l'activació del *collider*. Igual que la resta de trampes, quan el jugador les toca li causen un dany.

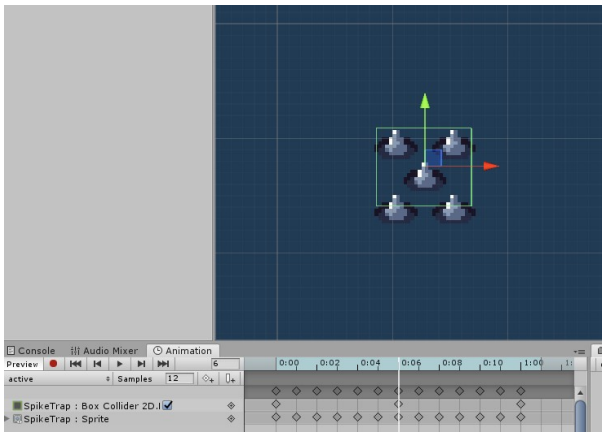


Figura 43: Activació del collider en el frame oportú

També s'ha afegit un sistema per poder fer que surtin en instants diferents. Aquest sistema consisteix en afegir un determinat temps extra abans d'activar el mecanisme per primera vegada. Per implementar aquesta funcionalitat ha estat útil utilitzar la funció **InvokeRepeating**, ja que permet realitzar crides a funcions de manera repetida cada cert interval de temps.

- **Canonada**: S'ha creat el disseny de la canonada i s'ha implementat de manera similar a la trampa de punxes a terra. La diferència, en aquest cas, és que dispara projectils lineals en una de les quatre direccions en què es configuri.

Després de realitzar proves, vaig veure que si s'utilitzava el conjur de l'escut les trampes es podien evitar massa fàcilment. Per aquest motiu, vaig decidir afegir la característica a totes les trampes, excepte la canonada, de que si el jugador les tocava amb l'escut aquest desapareixes.

3.20 Interfície gràfica

Com en la majoria de jocs, ha estat necessari afegir una sèrie de menús i elements gràfics que permetin al jugador entrar i sortir del joc, seleccionar opcions determinades o proporcionar-li ajuda.

A més, durant la partida, també ha estat necessari disposar d'un *HUD* que mostri gràficament informació rellevant per al jugador, com per exemple, la quantitat de vida que li resta. Així doncs, passem a repassar els elements de la interfície gràfica que s'han implementats en el joc:

- **Menú principal**: S'ha implementat un menú principal senzill amb un panell que permet al jugador iniciar la partida, sortir del joc o modificar el volum.

Figura 44: Menú principal del joc

- **Instruccions inicials:** S'ha implementat un panell que informa al jugador sobre els controls bàsics del joc. Bàsicament, mostra informació sobre com moure el jugador, atacar i llançar conjurs. Aquest panell es mostra al finalitzar l'escena introductòria i no es torna a mostrar més durant el joc.

Figura 45: Panell amb les instruccions bàsiques

- **Menú pausa:** El jugador pot pausar en qualsevol moment el joc polsant la tecla ESC, la tecla P o fent clic amb el ratolí al símbol de l'engranatge. Quan es realitza alguna d'aquestes accions, el joc queda en pausa i es mostra el menú que permet continuar amb la partida, modificar el volum o sortir del joc.

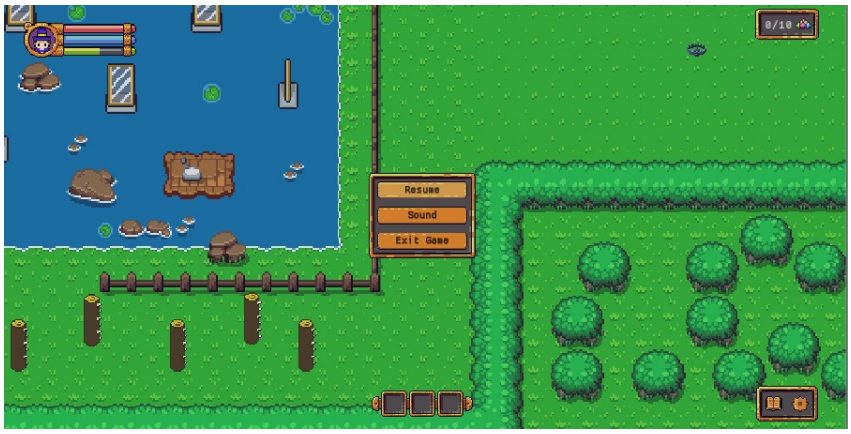


Figura 46: Menú pausa

- **Menú *game over***: Es va considerar que en cas que el jugador mori, es tingui la possibilitat de tornar a realitzar el nivell. Així doncs, quan el jugador mor es genera un esdeveniment que fa mostrar el menú de fi de partida. En aquest menú es mostren les opcions per tornar a realitzar el nivell o sortir del joc.

Figura 47: Menú game over

- **Llibre de conjurs:** Per informar millor sobre els diferents conjurs que hi ha en el joc, les tasques que realitzen, el nivell que té el jugador de cada conjur i les instruccions per llançar-lo, vaig decidir crear una espècie de llibre de conjurs en la interfície gràfica que permetés visualitzar tota aquesta informació.

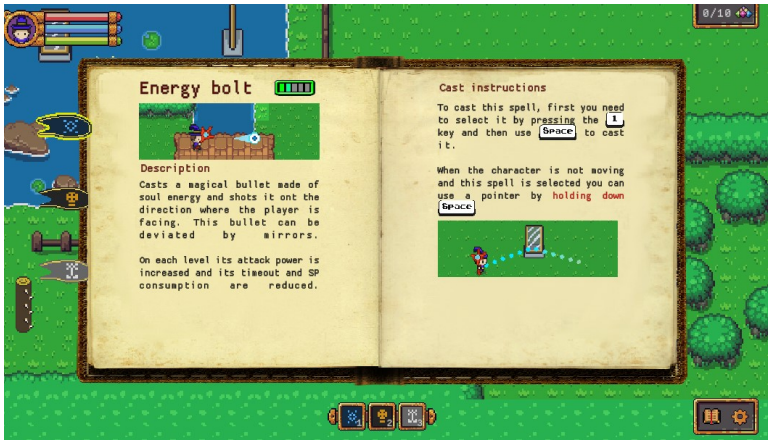


Figura 48: Llibre de conjurs

- **HUD**: S'han afegit una sèrie de components visuals que mostren una informació concreta al jugador durant la partida:

    - **Barra d'estats del jugador**: Mostra per cada estat del jugador (vida, mana i energia) una barra d'un color diferent que indica el percentatge que té d'aquell estat.

Figura 49: Barra d'estats del jugador

    - **Barra d'estats del boss:** S'utilitza una versió modificada de la barra d'estats del jugador per indicar la vida que li queda al boss i la quantitat de mana que té.

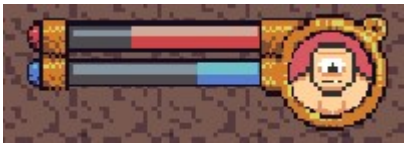


Figura 50: Barra d'estats del ciclop

    - **Panell dels conjurs**: Mostra visualment quin és l'estat de cada conjur i la tecla assignada al conjur. S'identifica cada conjur per un determinat símbol. Els estats en què pot estar un conjur són:
        - **No aprés**: No es mostra el seu símbol en el panell.
        - **Aprés:** Es mostra el seu símbol en el panell.
        - **Seleccionat**: Es mostra un marc groc sobre el símbol del conjur.
        - **Carregant-se**: Es mostra una animació sobre el símbol del conjur que dóna la noció que s'està carregant.

Figura 51: Panell dels conjurs

    - **Panell d'objectes:** Mostra les gemmes trobades i les que falten per trobar i també una clau quan el jugador la recull.

Figura 52: Panell d'objectes

- **Panell amb el menú d'opcions i el llibre de conjurs:** Conté les icones que pot clicar el jugador amb el ratolí per obrir el menú d'opcions o el llibre de conjurs.

Figura 53: Panell amb el menú d'opcions

3.21 Text

Per mostrar text tant en la interfície gràfica com en el comptador d'enemics, s'ha utilitzat el component **TextMehsPro** perquè incorpora moltes més funcionalitats que el component de text d'Unity.

Pel que fa al tipus de lletra, s'ha utilitzat **basis33**, ja que segons el meu punt de vista és un tipus de lletra que encaixa bé amb els jocs de l'estil *pixel art*.

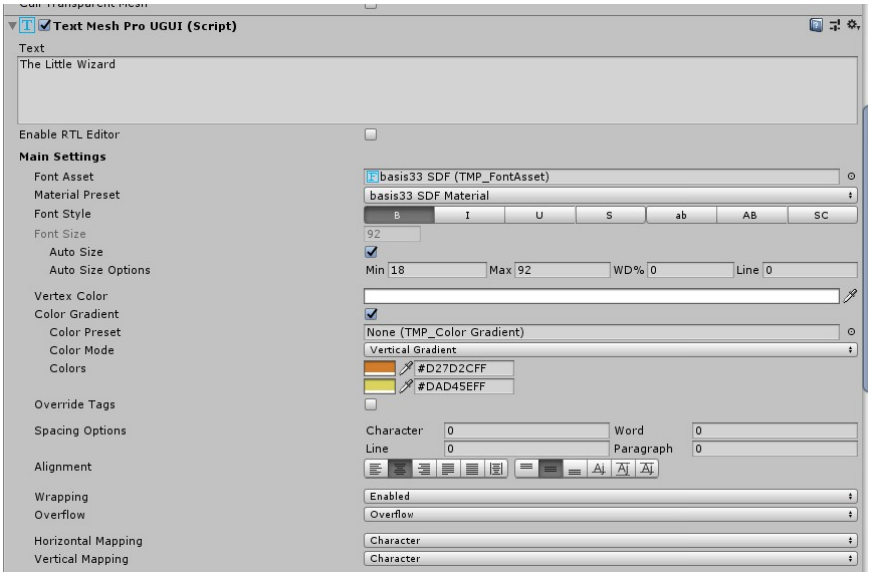


Figura 54: Disseny del títol del joc amb TextMeshPro

3.22 *Shaders*

Per fer més visible el dany causat en el jugador o en l'enemic, vaig decidir que s'havia de mostrar un efecte similar al flash. Després de buscar com ho podia realitzar sense haver de crear més animacions, vaig trobar un tutorial[7] amb el codi d'un *shader* que permetia aconseguir aquest efecte.

Durant la cerca també vaig trobar un tutorial[8] que incloïa un *shader* que permetia mostrar una línia de color al contorn de l'*sprite* i vaig decidir implementar-lo en la planta, de manera que aquest efecte indiqui que la planta està a punt de disparar. Posteriorment també es va decidir afegir aquest efecte al ciclop quan dispara el raig làser.

Combinant els dos *shaders* vaig obtenir el *shader* **OutlineFlash** que permet realitzar les dues funcions. Així doncs, per al jugador i els enemics he assignat un material al component **SpriteRenderer** que utilitza aquest *shader*.

Figura 55: Efectes realitzats pel shader OutlineFlash

3.23 So

L'última tasca d'implementació ha consistit en afegir so. Per fer-ho s'ha utilitzat un **AudioMixer** amb diversos **AudioMixerGroups** i s'ha creat la classe **SoundManager** de tipus **DontDestroyOnLoad** que gestiona la gran majoria dels sons que es reprodueixen en el joc.

Bàsicament en el joc es distingeixen dos tipus de sons. Per una banda, hi ha les cançons que sonen en cada zona o al menú principal i que es reprodueixen en bucle i, per l'altra els efectes sonors que tenen associats els objectes en el joc i que es reprodueixen un sol cop.

Tot i així, en la creació dels grups de l'AudioMixer s'han separat els efectes de la veu del jugador amb la resta d'efectes per poder ajustar millor el volum entre els dos.

[7] Flash Shader http://gamedevmalang.com/sprite-flash-in-unity/ (03/01/2020)
[8] Tutorial outline shader https://www.youtube.com/watch?v=vqDOirux0Es (03/01/2020)

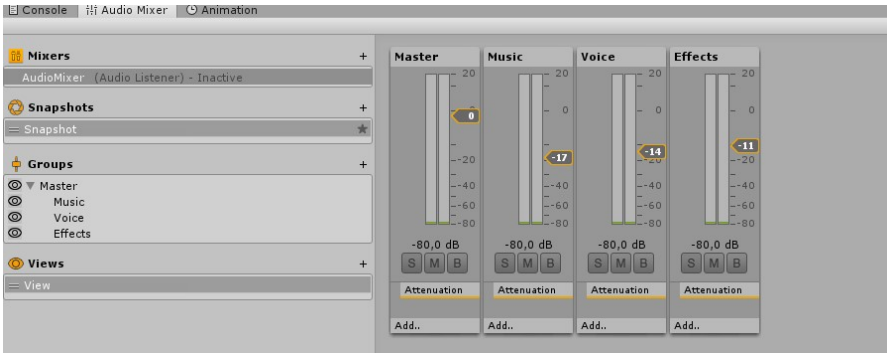


Figura 56: Configuració de l'AudioMixer

En la classe **SoundManager** es crea un **AudioSource** per cada AudioMixerGroup i es defineixen les operacions changeSong(), playVoice() i playEffect() per reproduir el tipus de sons comentats anteriorment. A part d'això, també incorpora operacions per reproduir sons quan es produeixen determinats esdeveniments, com per exemple, quan el jugador derrota el *boss* i es genera el senyal **onBossDefeated**. També es defineix l'*enum* **ZoneSong** per poder fer referència, en les transicions de zona, a la cançó que s'ha de reproduir.

D'altra banda, també s'encarrega de guardar les preferències del jugador respecte el volum de so. En aquest cas, s'ha utilitzat la classe **PlayerPrefs.** Per poder canviar el nivell de so s'ha creat un panell per a la interfície gràfica que permet ajustar el volum.

Figura 57: Panell per canviar el volum

4. Conclusions

Un cop finalitzat aquest treball, les meves sensacions són bastant positives, ja que he pogut assolir la majoria d'objectius plantejats al començar el projecte. A més, el joc obtingut és força entretingut i permet passar una bona estona a qui decideixi jugar-hi.

Tot i així, el procés de desenvolupament del videojoc, en alguns moments, ha estat complicat. Doncs com ja he esmentat anteriorment, no disposava d'experiència ni coneixements previs tant en la programació de videojocs com en les eines utilitzades. En aquest sentit, ha estat un gran encert utilitzar una eina tant popular com és Unity perquè la quantitat de tutorials i informació que hi ha m'han estat de gran ajuda.

Crear el meu primer joc era un repte personal i ara que he acabat el treball puc dir que l'he pogut assolir. A part d'haver aconseguit crear un joc, he pogut conèixer de primera mà com és el procés de creació d'un joc i els diferents elements que hi intervenen. A més, també m'ha servit per guanyar experiència en eines utilitzades en aquest sector, com és el cas d'Unity.

Planificar un treball en el què es realitzin tasques que un mateix no ha fet mai comporta un cert grau de risc. En aquest cas, la planificació establerta al començament del projecte va ser força correcta i bastant ajustada al que necessitava. Malgrat tot, degut a la meva inexperiència, algunes de les tasques a realitzar m'han costat més temps del previst. A més, també s'han afegit alguns elements inicialment no previstos. Això ha provocat un endarreriment en el calendari que ha implicat haver de renunciar en alguns dels nivells que inicialment tenia en ment. No obstant això, estic satisfet perquè es manté l'essència del joc plantejat al començar el projecte.

A part d'afegir més nivells, sóc conscient que hi ha certs elements en el joc que es podrien millorar, com per exemple, la persecució dels enemics es podria haver implementat amb l'algorisme A* per evitar les col·lisions amb els obstacles que hi ha en el nivell. També es podrien haver afegit zones on el jugador pogués caure. Malauradament, el temps de que disposem per realitzar el treball és bastant ajustat i s'haurà de deixar aquestes tasques pendents per si decideixo continuar amb el desenvolupament d'aquest joc en un futur.

Per acabar, comentar la meva satisfacció amb els resultats aconseguits i així com també l'experiència viscuda en la realització d'aquest treball.

5. Glossari

Assets: Elements que componen els objectes en el joc (imatges, animacions, so etc.)

Boss: Enemic que surt al final del nivell o del joc i que és més fort que la resta d'enemics.

Collider: Component que es pot assignar als objectes d'Unity per delimitar la zona on es produeixen col·lisions.

Corrutina: És una tipus de funció en Unity que permet aturar temporalment la seva execució i continuar per on passava en el següent frame o al cap d'un temps determinat.

Drag&drop: Acció d'arrossegar i deixar anar, generalment, realitzada mitjançant el ratolí o a través d'una pantalla tàctil.

HUD: Element gràfic que es mostra en tot moment a la pantalla durant el nivell i que mostra informació d'utilitat pel jugador, com per exemple, els punts de vida.

Mana: Energia màgica utilitzada per realitzar un conjur en el joc.

Sprite: Imatge o part d'un sprites-sheet.

Sprites-sheet: Conjunt d'imatges agrupades en una imatge més gran. Generalment, contenen les animacions d'un determinat element o una imatge dividida en parts.

Patró singelton: Patró de disseny de programari utilitzat per restringir el nombre d'instàncies d'una classe a un sol objecte.

Patró estat: Patró de disseny de programari utilitzat quan es vol obtenir diferents comportaments d'un objecte segons el seu estat intern.

Patró observador: Patró de disseny de programari utilitzat quan un objecte vol ser notificat d'unes determinades accions realitzades per un altre objecte o programa.

Prefab: Es tracta d'un GameObject d'Unity el qual s'ha guardat tota la configuració dels elements que el component i els seus valors. Serveix com a plantilla per instanciar nous objectes amb la mateixa configuració.

Tilemap: Graella formada per cel·les de la mateixa mida (Tiles) i que, cada una d'elles, pot mostrar una imatge.

Timeline: Component d'Unity que permet realitzar seqüències d'esdeveniments en el joc.

6. Bibliografia

En aquest apartat es llistaran els documents consultats no citats directament dins del contingut de la memòria i els recursos utilitzats en la implementació del joc.

6.1 Documents

- Unity: https://en.wikipedia.org/wiki/Unity_(game_engine) (25/09/19)
- Unreal Engine: https://en.wikipedia.org/wiki/Unreal_Engine (25/09/19)
- Game Maker Studio: https://en.wikipedia.org/wiki/GameMaker_Studio (25/09/19)
- Comparativa Unity – Unreal Engine : https://sundaysundae.co/unity-vs-unreal/ (25/09/19)
- Patró estat: https://gameprogrammingpatterns.com/state.html (02/10/19)
- Patró observador https://en.wikipedia.org/wiki/Observer_pattern (03/11/19)
- API C# d'Unity: https://docs.unity3d.com/2018.4/Documentation/ScriptReference/ (03/10/2019)

6.2 Gràfics

- Base per al jugador i els goblins, arbre, gerro, botó, palanca, pont , estàtua, *tilemap*: https://opengameart.org/content/zelda-like-tilesets-and-sprites (02/10/2019)
- Cobra: https://elthen.itch.io/2d-pixel-art-cobra-sprites (27/10/2019)
- Bolet i bruixot vell: https://superdark.itch.io/enchanted-forest-characters (20/10/2019)
- Ciclop: https://elthen.itch.io/2d-pixel-art-cyclops-sprites (20/12/2019)
- Objectes (gemma,pocions,recepta,llibres,clau): https://kyrise.itch.io/kyrises-free-16x16-rpg-icon-pack (20/10/2019)
- Baül: https://opengameart.org/content/pixel-art-destructible-object (20/12/2019)
- Trampa de l'ós i punxes al terra: https://opengameart.org/content/animated-traps (20/12/2019)

- Explosió projectil: https://opengameart.org/content/animated-explosions (04/01/2020)
- Explosió mort del jugador: https://opengameart.org/content/explosion-animations (04/01/2020)
- Portal: https://elthen.itch.io/2d-pixel-art-portal-sprites (20/12/2019)
- Elements GUI: https://opengameart.org/content/ui-pieces i https://opengameart.org/content/golden-ui (04/01/2020)
- Tecles: https://gerald-burke.itch.io/geralds-keys (04/01/2020)
- Basis33: https://www.1001fonts.com/basis33-font.html (04/01/2020)

6.3 Sons

- Veu jugador: https://cicifyre.itch.io/rpg-voice-starter-pack (04/01/2020)
- Música del menú i de les zones: https://opengameart.org/content/jrpg-collection (04/01/2020)
- Cop al jugador,pedra del ciclop,llibre,clau,pocions,baül,gerro, arquer, fletxa i estàtua: https://simon13666.itch.io/sound-starter-pack (04/01/2020)
- Efecte text: https://freesound.org/people/hz37/sounds/463751/ (04/01/2020)
- Trampa de l'ós: https://freesound.org/people/ThePriest909/sounds/278203/ (04/01/2020)
- Làser: https://freesound.org/people/MusicLegends/sounds/344315/ (04/01/2020)
- Explosió del projectil del bolet, explosió del projectil de la planta, gemma, portal: https://opengameart.org/content/512-sound-effects-8-bit-style (04/01/2020)
- Tronc amb punxes: https://opengameart.org/content/rpg-sound-pack (04/01/2020)