Universitat de Girona
Escola Politècnica Superior

Creació d’un videojoc de plataformes musical en 2D: Komorebi

PROJECTE/TREBALL DE FI DE GRAU
Grau en Disseny i Desenvolupament de Videojocs

Document: Memòria
Alumnes: Pol Darder(GDDV) i Quim Lloret(GEINF)
Tutor: Gustavo Patow
Departament: IMAE
Àrea: LSI
Convocatòria: 09/2022

Índex

1. Introducció **8**
1.1 Introducció 8
1.2 Motivacions 9
1.3 Propòsit 9
1.4 Objectius 9
1.5 Separació de tasques 10
2. Estudi de viabilitat **12**
2.1 Recursos humans 12
2.2 Viabilitat tecnològica 12
2.2.1 Hardware 12
2.2.2 Software 13
2.3 Viabilitat legal 13
2.4 Viabilitat econòmica 13
2.5 Estat de l'art 15
2.5.1 Metodologia de cerca 15
2.5.2 Resultats destacats 16
2.5.3 Taula de comparació 19
2.5.4 Resultats en la comparativa 20
3. Metodologia **22**
4. Planificació **24**
4.1 Tasques planificades 24
4.1.1 Planificació del joc 24
4.1.2 Estudi de diferents motors 24
4.1.3 Estudi del motor gràfic Unity i del llenguatge C# 24
4.1.4 Cerca, creació i preparació d'elements gràfics 24
4.1.5 Disseny i implementació d'un prototip amb Unity 24
4.1.6 Disseny i implementació del joc 24
4.1.7 Disseny i implementació de la interfície d'usuari 24
4.1.8 Disseny i implementació de IA 25
4.1.9 Cerca, creació i integració de música 25
4.1.10 Verificació i proves dels algoritmes desenvolupats 25
4.1.11 Creació d'una demo del videojoc 25
4.1.12 Documentació 25
4.2 Temps estimat 26
5. Marc de treball i conceptes previs **28**
5.1 Introducció als motors de videojocs 28
5.2 Motors de videojocs 31
5.2.1 Unity 31

5.2.2 Godot Engine 34
5.2.3 Unreal Engine 36
5.3 Motor escollit 38

6. Requisits del sistema 39
6.1 Requeriments funcionals 39
6.1.1 Llistat de requeriments funcionals 39
6.1.2 Identificació dels actors 39
6.2 Requeriments no funcionals 40

7. Estudis i decisions 43
7.1 Visual Studio Code 43
7.2 Audacity 43
7.3 Bosca Ceoil 44
7.4 FMOD Studio 45
7.5 Piskelapp 45
7.6 Google Drive 46
7.7 Diagrams.net 46

8. Anàlisis i disseny del sistema 48
8.1 Casos d’ús 48
8.1.1 Diagrames de casos d’ús 48
8.1.1.1 Diagrama de casos d’ús del menú principal 48
8.1.1.2 Diagrama de casos d’ús del menú de pausa 49
8.1.1.3 Diagrama de casos d’ús del menú de Game Over 49
8.1.2 Fitxes de casos d’ús 50
8.1.3 Diagrames d’activitats 55
8.1.3.1 Diagrama d’activitats del menú principal 55
8.1.3.2 Diagrama d’activitats del menú d’opcions 57
8.2 Interfícies 58
8.3 Core i Core Components 58
8.4 Màquines d’estat 60
8.4.1 Concepte de la màquina d’estats 60
8.4.2 Estructura i funcionament general de la nostra màquina d’estats 60
8.4.3 Script StateMachine 61
8.4.4 Script Estat (Superclasse) 62
8.4.5 Script Entitat 64
8.5 Shader 65
8.6 Checkpoints 66
8.7 Lluita amb el Boss 67
8.7.1 Escenari de lluita 68
8.7.2 Boss 69
8.7.3 Gestió de la lluita per fases 71
8.8 Efecte de parallax 73
8.9 Interfícies d’usuari 75
8.9.1 Menú principal 76

8.9.2 Menú de pausa 77
8.9.3 Menú de Game Over 79
8.9.4 Diàlegs 80
8.10 Sincronització de la música amb el gameplay 82
8.10.1 Sincronització de l'FMOD Studio amb Unity 82
8.10.2 Creació de la música amb marques 84
8.10.3 Creació de l'Audio Manager 84
8.10.4 Creació d'objectes que es sincronitzen amb la música 85

9. Implementació i proves 86
9.1 Recerca d'elements gràfics 86
9.1.1 Personatge principal 86
9.1.2 Enemics 86
9.1.3 Final Boss 86
9.1.4 Terreny 87
9.1.5 Background 88
9.1.6 Altres elements de l'escena 88
9.2 Interfícies 90
9.3 Core 91
9.4 Core Components 92
9.4.1 Appearance 92
9.4.2 CollisionSenses 92
9.4.3 Combat 93
9.4.4 Inventory 94
9.4.5 Movement 95
9.4.6 Stats 95
9.4.7 Muntatge del Core i els Core Components a l'entitat 96
9.5 Màquina d'estats del Player 97
9.5.1 Superclasse dels estats del Player: Script PlayerState 97
9.5.2 Gestió de l'input: Input System i InputHandler 98
9.5.3 Entitat controlada per la màquina d'estats: script Player 100
9.5.4 Estructura jeràrquica i canvis entre estats 102
9.5.5 Estats de la màquina pel Player 103
9.5.5.1 SUPERESTAT: GroundedState 104
9.5.5.2 GroundedState > IdleState 105
9.5.5.3 GroundedState > MoveState 105
9.5.5.4 GroundedState > LandState 105
9.5.5.5 SUPERESTAT: AbilityState 106
9.5.5.6 AbilityState > JumpState 106
9.5.5.7 AbilityState > DashState 107
9.5.5.8 AbilityState > WallJumpState 108
9.5.5.9 AbilityState > AttackState + Weapon 109
9.5.5.9.1 Arma: Script Weapon 109
9.5.5.9.2 Script AttackState 112
9.5.5.10 SUPERESTAT: TouchingWallState 113

9.5.5.11 TouchingWallState > WallGrabState 114
9.5.5.12 TouchingWallState > WallClimbState 114
9.5.5.13 TouchingWallState > WallSlideState 114
9.5.5.14 InAirState (Estat sense Superestat) 115
9.5.5.15 LedgeClimbState (Estat sense Superestat) 117
9.5.5.16 DeadState (Estat sense Superestat) 119
9.6 Shader 120
9.6.1 Vida diegètica 120
9.6.2 Efecte de 'hit' 122
9.6.3 Efecte de realçar contorn al tenir el dash disponible 123
9.6.4 Efectes de parpalleig 125
9.7 Checkpoints 128
9.8 Màquines d'estats pels enemics 129
9.8.1 Possibles estats dels enemics 129
9.8.1.1 Idle State 129
9.8.1.2 Move State 130
9.8.1.3 Attack State 130
9.8.1.3.1 MeleeAttack State 131
9.8.1.3.2 RangeAttackState 132
9.8.1.3.3 ElectricBombAttack State 132
9.8.1.3.4 LaserBeamAttackState 132
9.8.1.4 Charge State 133
9.8.1.5 Dodge State 133
9.8.1.6 LookForPlayer State 134
9.8.1.7 PlayerDetected State 134
9.8.1.8 Dead State 135
9.8.2 Herència d'estats 135
9.8.3 Tipus d'enemics 137
9.9 Lluita amb el Boss 140
9.9.1 Boss 140
9.9.1.1 Barra de vida 140
9.9.1.2 BossBehaviour 140
9.9.1.3 BossCombatController 141
9.9.2 Gestió de lluita per fases 141
9.9.2.1 Implementació de BossBattlePhaseManager 141
9.9.2.2 Implementació de l'inici de la lluita 141
9.9.2.3 Implementació de la fase 1 142
9.9.2.4 Implementació de la fase 2 142
9.9.2.5 Implementació de la fase 3 143
9.10 Efecte de parallax 145
9.11 Interfícies d'usuari 146
9.11.1 Menús dins el GameController 146
9.11.2 Menú principal 147
9.11.3 Menú de pausa 154

9.11.4 Menú de Game Over 156
9.11.5 Diàlegs 157
9.12 Creació de la música 159
9.13 Sincronització de la música amb el gameplay 161
9.13.1 Creació de l’Audio Manager 161
9.13.2 Creació d’objectes que es sincronitzen amb la música 165
9.13.2.1 Plataformes que apareixen i desapareixen 165
9.13.2.2 Plataformes de salt 167
9.13.2.3 Trampes làser 169
9.13.2.4 Plataformes que es desplacen 171
10. Implantació i resultats 175
10.1 Legislació i normativa vigent 175
10.2 Resultats de la màquina d’estats de Player 175
10.2.1 Idle State 175
10.2.2 MoveState 175
10.2.3 Land State 176
10.2.4 Jump State 176
10.2.5 DashState 176
10.2.6 WallJump State 177
10.2.7 AttackState 177
10.2.8 WallGrabState 178
10.2.9 WallClimbState 178
10.2.10 WallSlideState 178
10.2.11 InAirState 179
10.2.12 LedgeClimbState 179
10.3 Resultats dels efectes del shader 180
10.3.1 Vida diegètica 180
10.3.2 Efecte de ‘hit’ 180
10.3.3 Efecte de realçar contorn al tenir el dash disponible 180
10.3.4 Efectes de parpalleig 181
10.4 Parallax 182
10.5 Interfícies d’usuari 184
10.5.1 Menú principal 184
10.5.2 Menú de pausa 185
10.5.3 Menú de Game Over 186
10.5.4 Diàlegs 186
10.6 Elements que es sincronitzen amb la música 187
10.7 Lluita amb el Boss 189
10.8 Màquines d’estats pels enemics 191
10.8.1 BallAndChain 191
10.8.1.1 Idle State 191
10.8.1.2 Move State 191
10.8.1.3 LookForPlayer State 192
10.8.1.4 PlayerDetected State 192

10.8.1.5 Charge State 192
10.8.1.6 MeleeAttack State 192
10.8.1.7 Dead State 193
10.8.2 Archer 193
10.8.1.1 Idle State 193
10.8.1.2 Move State 193
10.8.1.3 LookForPlayer State 194
10.8.1.4 PlayerDetected State 194
10.8.1.5 Dodge State 194
10.8.1.6 MeleeAttack State 195
10.8.1.7 RangeAttack State 195
10.8.1.8 Dead State 195
10.9 Checkpoints 195
11. Conclusions 197
12. Treball futur 198
13. Bibliografia 199
14. Annexos 200
15. Manual d'usuari 201
15.1 Iniciar el joc 201
15.2 Controls 201
15.3 Objectiu del joc 202
15.4 Menú d'opcions 202

1. Introducció

1.1 Introducció

El món dels videojocs ha viscut una evolució molt gran des dels inicis. A mesura que la tecnologia ha avançat, aquesta ha servit als videojocs per a poder desenvolupar produccions amb millors gràfics, mecàniques més complexes, dispositius innovadors i avenços cada vegada més establerts en la indústria. Tot i que les grans empreses desenvolupadores de videojocs han fet ús de tecnologia cada vegada més avançada per tal de crear noves característiques pels jocs i augmentar l'expectativa dels jugadors, una bona part del públic segueix fent créixer cada vegada més l'interès pels jocs 'indie'. Aquests jocs estan desenvolupats per companyies independents, les quals disposen de menys desenvolupadors i, normalment, fan ús d'una tecnologia menys innovadora. Al no disposar de tants recursos, les companyies 'indie' han buscat crear jocs innovadors en quan a creativitat, estil i jugabilitat per tal de cridar l'atenció del públic consumidor. A l'hora de proposar-nos el repte de crear un videojoc, ens vam sentir inevitablement lligats a aquesta categoria de desenvolupadors.

Donats els nostres coneixements tècnics i l'objectiu del nostre projecte, vam decidir que el nostre joc es basaria en el gènere *Action Platformer,* el qual incorpora elements de plataformes i de combat al mateix temps.

Com a produccions 'indie' destacades en els últims anys que comparteixin característiques amb aquest gènere, hi podem posar *Celeste*, la qual opta per una estil de joc *Platformer* i una estética Pixel Art, o HollowKnight, que afegeix acció i combat.

Alhora però, aquest gènere és cada vegada més popular, i per tal de diferenciar-nos de la competència, vam decidir incorporar un altre element a l'estructura principal del nostre joc: la música. La música com a element principal dels videojocs no és cap novetat, i els jocs que es basen en ella s'engloben en el gènere de videojocs "musicals". Aquest gènere aprofita la música per a crear una interacció satisfactoria amb el jugador, el qual moltes vegades ha de seguir el ritme d'una peça de música per poder avançar. Videojocs destacats d'aquest gènere podrien ser el *Rock Band* o el *Geometry Dash.*

Així doncs, ens volem situar entre mig dels dos gèneres per tal de crear un títol innovador, que combini la jugabilitat i el caràcter dels Action Platformers i la diversió i interactuabilitat dels jocs musicals.

1.2 Motivacions

Les motivacions principals rere aquest projecte són essencialment les de posar en pràctica els coneixements que hem obtingut al llarg dels nostres estudis. Treballar sobre un projecte estructurat que ens permeti fer-nos una idea de la metodologia que segueixen les empreses de desenvolupament de videojoc independents, així com aprendre noves tècniques que ens puguin ser útils de cara a futurs projectes.

1.3 Propòsit

Amb el desenvolupament d'aquest projecte, ens proposem crear un videojoc que es diferencïi dels videojocs existents del mateix gènere, oferint al jugador una experiència de joc diferent i innovadora gràcies a l'incorporació de la música com a tret diferenciatiu.

1.4 Objectius

En aquest projecte es vol dissenyar i crear un videojoc en 2D de gènere Action Platformer - Musical, implementant les mecàniques essencials dels jocs de plataformes i combat, entre les quals s'inclouen el moviment i habilitats del jugador, els enemics i l'enemic final. Juntament amb això, es vol incloure la música com a element central del videojoc, permetent que controli elements propis de l'ambientació i l'escenari, creant noves possibilitats en la jugabilitat que marquin la diferència entre el nostre joc i la competència.
En quant a l'estètica, es vol utilitzar l'estil "Pixel art" per representar gràficament els diferents elements que formaran part del videojoc, creant un disseny definit i coherent, agradable a la vista pel jugador.
Es poden llistar el objectius d'una manera més concreta com:

- Dissenyar les escenes i nivells del joc.
- Aprendre i millorar el coneixement de C# en la programació de videojocs.
- Aprendre el funcionament del Unity 2D / Aprofundir en el funcionament de Unity 2D.
- Aprendre a utilitzar eines externes al Unity com FMOD o Piskelapp
- Entendre les màquines d'estat i implementar-les pel comportament del jugador i dels enemics.
- Fer ús del Shader Graph de Unity per crear feedback per jugador.
- Implementar un sistema d'events que sincronitzi la música amb els canvis en l'escena que s'hi vegin afectats
- Finalitzar la documentació del projecte.

1.5 Separació de tasques

Al haver realitzat el projecte en parelles, hem hagut de separar-nos les tasques entre nosaltres i també determinar quines són les tasques conjuntes:

- **Pol**
    - **Core:** Implementar l'element de comunicació entre la màquina d'estats i els personatges.
    - **Core Components:** Implementar un sistema de components que permeti, de manera escalable, afegir funcionalitats a qualsevol entitat controlada per la màquina d'estats. Implementar els components necessaris pel joc.
    - **Màqiuna d'estat del jugador:** Implementar la màquina d'estats que controlarà el moviment del jugador. Es pot separar en els següents sub-apartats:
        - **Sistema d'inputs:** Sistema captador dels inputs necessaris pel moviment del jugador i el tractament d'aquests.
        - **Màquina d'estats:** Funcionament base del sistema.
        - **Estats:** El comportament del personatge en cada un dels estats.
    - **Interfícies de comunicació**: Implementar interfícies a partir de les quals es puguin comunicar els diferents elements de l'escena.
    - **Recerca dels elements gràfics**: Cercar, crear i incloure al projecte els elements gràfics que en formen part.
    - **Implementació del Shader2D pel jugador amb shader graph:** Crear un Shader2D amb l'eina *Shader Graph* de Unity que permeti millorar el feedback sobre l'estat del jugador en cada moment.

- **Quim**
    - **Interfície d'usuari:** Disseny i implementació de les UIs.
        - **Menú de pausa**: Implementar el menú de pausa del videojoc i les seves opcions.
        - **Menú Game Over:** Implementar el menú de Game Over i les seves opcions.
        - **Menú principal:** Implementar el menú principal del videojoc i les seves opcions.
        - **Diàlegs:** Dissenyar i implementar un sistema de diàlegs que serviran per a informar al jugador durant la partida.
    - **Màquina d'estats pels enemics:** Implementar una màquina d'estats que controlarà el comportament dels enemics, juntament amb la creació d'enemics amb diferents comportaments.
    - **Implementació del Boss i de la lluita de fases amb aquest:** Dissenyar i implementar la lluita contra un Boss. Es pot dividir en les següents subtasques:
        - **Boss:** Disseny i implmentació del comportament del Boss.
        - **Boss UI**: Disseny i implementació d'una barra de salut per al Boss.

        - **Escenari de lluita:** Disseny i muntatge de l'escenari on es realitzarà la lluita amb el Boss, juntament amb els elements necessaris
        - **Control de la lluita per fases:** Implementar un sistema que controli l'estat de la lluita i depenent de la fase en què estigui implementar una lògica o una altra.
    - **Creació de la música principal:** Creació de la música principal del videojoc amb el programa BoscaCeoil.
    - **Sincronització de la música amb gameplay:** Implementar la sincronització de la música amb el gameplay. Es pot dividir en el següent:
        - **Integració de FMOD a Unity**
        - **Muntatge de la música a FMOD Studio**
        - **Creació d'objectes que es sincronitzaran amb la música**
    - **Efecte de paral·lax:** Implementar un efecte de paral·latge pels escenaris del videojoc.

- Junts
    - **Planificació del joc**
    - **Estudi de diferents motors**
    - **Estudi del motor gràfic Unity i del llenguatge C#**
    - **Cerca i preparació d'elements gràfics**
    - **Disseny i implementació d'un prototip amb Unity**

Pel que fa la distribució dels pesos entre els diferents apartats dins el desenvolupament del projecte, aquests es poden veure a la **Figura 1.1.**

Estètica	10%
Narrativa	0%
Mecàniques	70%
Tecnologia	20%

Figura 1.1: Taula d'autoavaluació

Hem considerat que la part principal en el desenvolupament es centra en crear una bona jugabilitat, principalment formada pel moviment i el combat entre el jugador i els enemics. Per tant, l'apartat de implementació de les mecàniques és el de més pes.
El pes de l'apartat de tecnologia ve principalment a causa de la tasca de sincronitzar la música amb el gameplay, integrant la llibreria FMOD a Unity i realitzant el muntatge de la música a FMOD Studio. Per altra banda, els elements gràfics del projecte havien de mantenir una relació en estil, color i forma, de manera que el resultat final fos coherent. Tot i això, la majoria d'aquests elements gràfics han estat agafats de fonts d'ús lliure i adaptats al joc. És per això que se li ha donat un pes inferior als apartats de mecàniques i tecnologia.
L'element narratiu no ha sigut desenvolupat en el projecte.

2.
Estudi de viabilitat

Abans d'iniciar el desenvolupament del projecte, realitzem un estudi de la viabilitat d'aquest a nivell de recursos humans, requeriments tecnològics i requeriments econòmics, seguit d'una valoració de l'estat de l'art per tal d'analitzar si el nostre projecte tindria cabuda entre els competidors del mercat.

2.1 Recursos humans

En un estudi de desenvolupament de videojocs hi solen treballar un conjunt de persones amb perfils tècnics diferents, les quals agafen un rol respectiu a la seva especialització. Cada un d'aquests rols és de vital importància per tal de acabar obtenint un resultat de bona qualitat, i la comunicació entre aquests permet mantenir la coherència general del producte. Així doncs, podem llistar els perfils tècnics més habituals com:

- Dissenyador: Encarregat de desenvolupar el concepte del joc, així com el guió, la jugabilitat, les mecàniques, els escenaris i els nivells.
- Programador: Implementen el disseny i el funcionament del joc.
- Artista: Es fa càrrec dels elements visuals del joc, així com del seu estil gràfic.
- Productor musical: s'encarrega de produir els efectes de so i la música del joc.

En el cas del nostre projecte, tota la feina corresponent a aquests diferents rols s'ha hagut de repartir entre dues persones.

2.2 Viabilitat tecnològica

2.2.1 Hardware

Pel desenvolupament del treball l'únic element de hardware que s'ha necessitat han sigut els nostres ordinadors personals, els quals presenten les següents especificacions:

Ordinador 1

- **Processador**: Intel Core i7-8750 CPU 2.2GHz
- **Targeta gràfica**: NVIDIA GeForce GTX 1050
- **Memòria RAM**: 16GB

Ordinador 2

- **Processador**: Intel(R) Core(TM) i7-10700K a 3.80GHz

- **Targeta gràfica**: Nvidia Geforce RTX 2070
- **Memòria RAM**: 32 GB

2.2.2 Software

Pel que fa al programari, el projecte s'ha desenvolupat sobre el motor de jocs Unity amb l'IDE Visual Studio Code vinculat i utilitzant el llenguatge C#.

Per tots els aspectes relacionats amb la música, des de la seva creació fins a la introducció d'aquesta en el funcionament del joc s'ha utilitzat el següent *software*:

- Audacity
- Bosca Ceoil
- FMOD

En quant al programari utilitzat per la creació i modificació de *Sprites*, llistem:

- Piskelapp

Per les tasques relacionades amb organització, redacció i creació d'esquemes hem utilitzat:

- Google Docs
- Draw

En l'apartat estudis i decisions es fa una descripció detallada de cada un dels programaris esmentats, així com una explicació de l'ús que li hem donat en el nostre projecte.

2.3 Viabilitat legal

En quant a drets legals i propietat intel·lectual, tots els recursos externs que hem inclòs en el nostre projecte s'han obtingut de fonts d'ús lliure i, sempre que així s'especifiqui, donant crèdit a l'autor corresponent. Tot el programari utilitzat s'ha descarregat de fonts oficials.

2.4 Viabilitat econòmica

Per tal de realitzar l'estudi de la viabilitat econòmica del nostre projecte, separem els costos totals en dos apartats, per una part els costos dels recursos tècnics i per l'altra els dels recursos humans. Cal especificar que el projecte no ens suposarà cap cost econòmic real, però en els següents apartats es planteja com si es tractés d'un projecte professional per tal de fer-nos una idea de quins costos suposaria tirar endavant el desenvolupament d'un videojoc com el nostre.

Costs del recursos tècnics

Com s'ha esmentat en l'apartat de viabilitat tecnològica, el projecte s'ha desenvolupat mitjançant els ordinadors personals que ja posseiem anteriorment. A més, tot el programari utilitzat és gratuït, per tant, el cost real dels recursos tècnics és de 0€. Tot i així, plantegem una situació hipotètica en la qual no haguéssim disposat de cap d'aquests recursos prèviament a l'inici del treball. Donada aquesta situació, tot el cost dels recursos tècnics recauria en el cost dels ordinadors:

- Ordinador 1: ≈ 1.100 €
- Ordinador 2: ≈ 2.300 €

Total ≈ 3.400€

Cost dels recursos humans

Per tal de calcular el cost hipotètic dels recursos humans d'un equip fictici, suposem que aquest està format per 4 integrants, un per cada rol del desenvolupament, llistats en l'apartat de recursos humans.

Prenent com a referència la informació d'aquestes pàgines:
https://www.crehana.com/blog/animacion-modelado/cuanto-gana-desarrollador-videojuegos/
https://promocionmusical.es/salidas-profesionales/compositor-musica-videojuegos/
https://www.glassdoor.es/Sueldos/2d-artist-sueldo-SRCH_KO0,9.htm

Podem establir un sou mig estimat per cada posició de l'equip:

- 1750€ per la posició de **programador** de videojocs.
- 2375€ per la posició de **dissenyador** de videojocs.
- 2142€ per la posició **d'artista** 2D per videojocs.
- 1570€ per la posició de **productor musical** per videojocs.

Amb aquests sous, i considerant que un mes té 30 dies, els sous per dia queden així:

- Programador: 58,3€/dia
- Dissenyador: 79,1€/dia
- Artista: 71,4€/dia
- Productor musical: 52,3€/dia

Ara toca agrupar les tasques de cadascun dels dos tipus i mirar quants dies sumen en total:
- Total dies de tasques de programador: 41
- Total dies de tasques de dissenyador: 49
- Total dies de tasques d'artista: 21
- Total dies de tasques de productor musical: 4

Així doncs, el cost total puja a (62 * 58,3 + 49 * 79,1) = 7490€

Cost dels recursos humans
La conclusió de la viabilitat econòmica és que tirar endavant el nostre projecte en una situació real en la que no disposem anteriorment de cap recurs tècnic, suposaria una inversió d'aproximadament uns 3.400€ + 7490€ = 10890€

2.5 Estat de l'art

Abans de començar amb el desenvolupament del projecte, és útil i necessari analitzar l'estat de l'art. Un estudi de la competència ens permet analitzar si el projecte tindria alguna cabuda dins el mercat actual. Per tal de dur-lo a terme, cal cercar els principals competidors, escollint els que ofereixen característiques semblants a les que volem que tingui el nostre projecte. Així doncs, per acotar aquesta cerca cal fixar-se en elements com la jugabilitat, l'estètica, el gènere o el públic objectiu, contrastar-los amb el nostre producte, comprovar si aquest es desmarca de la resta i si, per tant, val la pena tirar-lo endavant.

2.5.1 Metodologia de cerca

Per tal de trobar la competència més similar a la nostra proposta de projecte, hem separat les característiques més representatives d'aquest:

- Jocs musicals de plataformes
- Jocs musicals d'acció
- Jocs musicals en 2D

Utilitzant el motor de cerca Google, hem utilitzat les següent paraules claus (utilitzant l'anglès ja que ens proporciona més resultats):

- Musical platformer games (jocs musicals i de plataformes)
- Musical action games (jocs musicals i d'acció)
- 2D Action platformer games (jocs en 2D d'acció i plataformes)
- 2D Musical games (jocs musicals en 2D)

Dels resultats obtinguts, hem seleccionat els més destacats i els que ofereixen característiques més semblants a la nostra proposta.

2.5.2 Resultats destacats

- **Symphonia**

Figura 2.1: Captura in-game de *Symphonia*

Es tracta d'un joc en 2D de plataformes desenvolupat per l'estudi independent *Sunny Peak,* en el qual controlem a un personatge dins una maquinària d'orquestra per tal d'anar arreglant els instruments que en formen part i poder realitzar una concert final. Es podria dir que la música té un pes molt important en l'ambientació i l'argument d'aquest joc, afectant també a la jugabilitat en seguides ocasions. Tot i això, no es tracta d'un element que afecti a les mecàniques del joc, sinò que més aviat ho fa en moments puntuals de l'argument. Té una estètica de dibuix a mà que li dona un caràcter original i diferent.

- **Celeste**

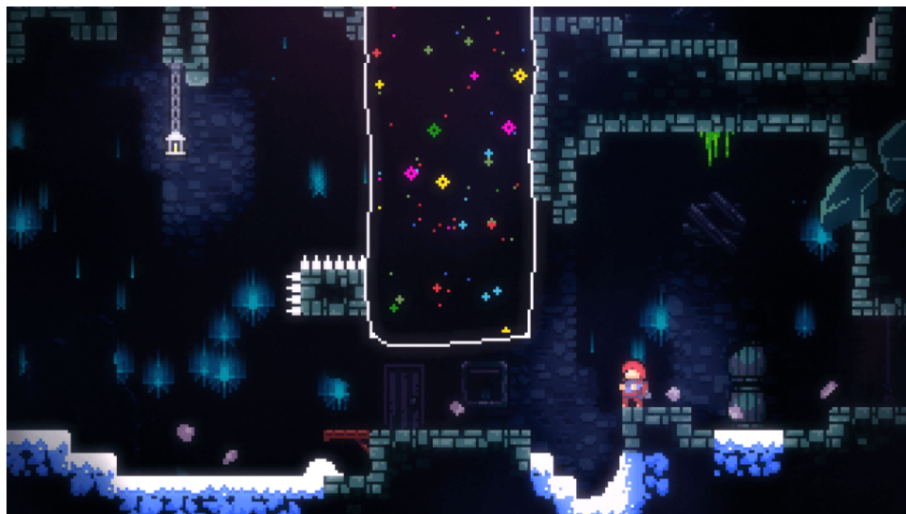


Figura 2.2: Captura in-game de *Celeste*

Celeste és una producció de l'estudi *Extremely OK Games*, el qual està format per dos desenvolupadors: un artista i un programador. Es tracta d'un joc de plataformes en 2D en el qual anem pujant una muntanya a través de nivells cada més complicats. Un dels aspectes que més s'ha valorat pels jugadors és el moviment del personatge, siguent aquest molt fluït, responsiu i agradable d'utilitzar. A *Celeste* es fa ús d'un bon disseny de nivell, combinat amb la introducció de noves mecàniques al llarg del joc, la qual cosa fa que no es faci repetitiu i mantingui al jugador interessat. A més, el joc segueix un fil narratiu corresponent al de la protagonista, el qual està rere molts dels elements que ens anem trobant als diferents escenaris i nivells. Utilitza una estètica *Pixel Art*, recordant a l'estil 8-bits.

- **Just Shapes & Beats**

Figura 2.3: Portada de *Just Shapes & Beats*

Aquest es tracta d'un videojoc musical d'acció en 2D, desenvolupat per membres de l'estudi *Berzerk Studio*. En aquest controlem a un quadrat, i necessitem esquivar i passar a través de formes per tal de superar els diferents nivells. La música hi desenvolupa un paper crucial en el fet que el comportament de les formes enemigues es regeix segons el ritme de la música. Així doncs, el jugador es beneficia de seguir el ritme per tal de poder superar els reptes, creant una jugabilitat original i divertida. Utilitza un estil de dibuix vectorial minimalista, útil i encertat pel seu propòsit.

- Hollow Knight

Figura 2.4: Captura in-game de *Hollow Knight*

Hollow Knight és un reconegut joc de gènere *Metroidvania* publicat per l'estudi *Team Cherry.* Aquest joc presenta un món obert en 2D explorable lliurement pel jugador, combinant mecàniques pròpies dels jocs de plataformes amb mecàniques de combat. Es tracta d'una producció bastant gran, dins la qual s'ha tingut cura de molts aspectes com una narració complexa, personatges secundaris, objectes, varietat d'enemics, diferents atacs i habilitats, etc. Creant així una obra completa en tots els aspectes. Utilitza un estil de dibuix a mà.

- Crypt of the Necrodancer

Figura 2.5: Captura in-game de *Crypt of the Necrodancer*

Aquest videojoc, desenvolupat per l'estudi *Brace Yourself Games*, és una barreja entre el génere d'exploració *roguelike* amb un joc musical. Les accions que pot fer el protagonista són més efectives si es realitzen al ritme de la música, amb la qual cosa

agafar el ritme és de vital importància a l'hora de jugar-hi. A més, el joc dóna la oportunitat al jugador de importar música personalitzada i de fer servir com a dispositiu d'entrada un panell de ball similar al de les recreatives. Utilitza un estil *Pixel Art*.

- **Billie Bust Up**

Figura 2.6: Portada de *Billie Bust Up*

Aquest és un videojoc l'estudi *Blueprint Games* que encara està en desenvolupament. Com indica la portada, es tracta d'un joc de plataformes musical en 3D. Tot i que no s'ha publicat la versió final del joc, sí que ha sortit la versió *Alpha*, i s'han pogut entreveure unes quantes de les característiques que tindrà aquest joc en la versió final. En aquest joc la música són cançons amb lletra, les quals afecten a la jugabilitat de varies maneres, entre elles: la lletra de la cançó serveix com a guía al jugador, els atacs dels enemics s'activen al ritme de la música i la música s'adapta segons la manera en que es jugui. A part d'això, el joc compta amb un argument de fons i una gran quantitat de personatges secundaris.
Es podria dir que utilitza una estètica *Cartoon*, que li dona un aspecte desenfadat i alegre.

2.5.3 Taula de comparació

Per tal de comparar els resultats obtinguts en la cerca, s'han establert els següents paràmetres a tenir en compte, junt amb els valor que podran agafar:

- **Dimensió:** Referint-nos a si el joc està desenvolupat en 3D o en 2D. (2D/3D)
- **Plataformes:** Si es pot incloure dintre el gènere de les plataformes. (SI/NO)
- **Combat:** Si presenta elements de combat. (SI/NO)

- **Importància de la música en la jugabilitat:** Determina el nivell amb el que la música condiciona la jugabilitat dins el joc, referint-se a si les accions que ha de dur a terme el jugador estan estretament relacionades amb la música. (1-5)

- **Estil gràfic:** Quin és l'estil gràfic del videojoc.

Joc	Dimensió	Plataformes	Combat	Música	Estil gràfic
Symphonia	2D	SI	NO	3	Dibuix a mà
Celeste	2D	SI	NO	0	Pixel Art
Just Shapes & Beats	2D	NO	NO	5	Vectorial
Hollow Knight	2D	SI	SI	0	Dibuix a mà
Crypt of the Necrodancer	2D	NO	SI	5	Pixel Art
Billie Bust Up	3D	SI	NO	4	Cartoon
Komorebi	2D	SI	SI	4	Pixel Art

Figura 2.7: Taula de comparació. Si la característica representa un valor diferent a la del nostre projecte, la casella es sobresalta en vermell, en verd altrament.

2.5.4 Resultats en la comparativa

Mitjançant la taula de comparació, hem pogut observar les similituds i diferències entre els jocs de la competència i el projecte que volem dur a terme. El joc que més similituds presenta ha resultat ser *Crypt of the Necrodancer*, el qual comparteix 4 de les 5 característiques amb el nostre. Tot i així, creiem que el nostre projecte és distintiu gràcies a les següents diferències:

- La perspectiva de la càmera en el noste projecte és d'una projecció ortogràfica paral·lela a l'escenari, mentre que en *Crypt of the Necrodancer* s'utilitza una perspectiva *Top-Down*, en la qual s'observa el personatge des d'una posició elevada.

- La falta de plataformes en *Crypt of the Necrodancer* fa que la jugabilitat sigui completament diferent a la del nostre projecte. En l'anterior, el jugador es pot moure en quatre direccions (amunt, avall, esquerra i dreta), i quan ho fa es mou una unitat

sencera cap a la direcció desitjada (semblant al moviment que faria una torre d'escacs que només es pogués moure una casella), la qual cosa funciona molt bé gràcies a la correlació d'aquest moviment amb el ritme de la música. A diferència, en el nostre joc el moviment del jugador és completament lliure dins els eixos del pla.

- Per últim, a *Crypt of the Necrodancer*, el jugador ha de seguir el ritme de la música en quasi cada acció que vulgui fer, des de l'inici fins al final. En el nostre projecte, el jugador comptarà amb un grau més gran de llibertat respecte a la música, referint-nos amb això a que, tot i haver de seguir el ritme per tal de superar molts dels obstacles que es trobarà, les accions que faci no estaràn condicionades pel ritme d'aquesta.

Respecte als altres resultats, hem observat que els jocs de plataformes, els quals més s'acosten al tipus de jugabilitat que volem obtenir, no inclouen la música com a element central. Els que ho fan, com per exemple ho fa *Symphonia* i *Billie Bust Up*, manquen d'altres característiques com el combat o la dimensió, i presenten un tipus d'estètica diferent.

La conclusió general que extraiem de l'estat de l'art és que el nostre projecte presenta caracterísitiques distintives suficients com per desmarcar-se de la resta de competidors, fins i tot dels que més s'hi assimilen. Per tant considerem el nostre un projecte amb cabuda dins el mercat actual.

3.
Metodologia

A l'hora de desenvolupar un projecte (i en el nostre cas en grup), hi ha diverses metodologies que es podrien adoptar. Entre elles hi ha la Waterfall, Scrum, Agile, etc.

Tot i això s'ha seguit una metodologia personalitzada proposada pel tutor i acceptada per nosaltres ja que aquesta s'adapta millor als objectius del projecte que no pas les altres metodologies.

Els passos a seguir que dicta aquesta metodologia són els següents:

1. Escollir el projecte que es vol desenvolupar
2. Decidir quines eines es faran servir (també el motor de joc) i aprendre-les a fer servir juntament amb el llenguatge de programació necessari (en el nostre cas el C#)
3. Estructurar el projecte en diferents parts a realitzar
4. Desenvolupar una part del projecte
5. Realitzar comprovacions per tal d'assegurar que la part realitzada funciona correctament i que s'ha realitzat al complert
    a. Si la comprovació determina que no s'ha realitzat la part correctament, es torna al punt 4 per a fer els retocs pertinents i així acabar-la.
    b. Si els resultats són els esperats, es finalitza la part actual i es passa a la seguent.
6. Fer una unió de diferents parts i comprovar que el conjunt funciona correctament
    a. Si es veu que falla alguna cosa o el resultat no agrada, es torna al punt 4 per a fer els canvis necessaris.
7. Crear models de proves per així fer comprovacions del correcte funcionament global amb més exemples i així intentar assegurar el bon funcionament.
    a. Si es veu que falla alguna cosa o el resultat no agrada, es torna al punt 4 per a fer els canvis necessaris.
8. Realitzar la redacció (revisada) del projecte.

L'objectiu al finalitzar una part és no haver de modificar-la posteriorment. D'aquesta manera s'eviten errors arrossegats que poden portar molts mals de cap i per tant temps perdut innecessari.

Aquest n'és el seu diagrama de flux:

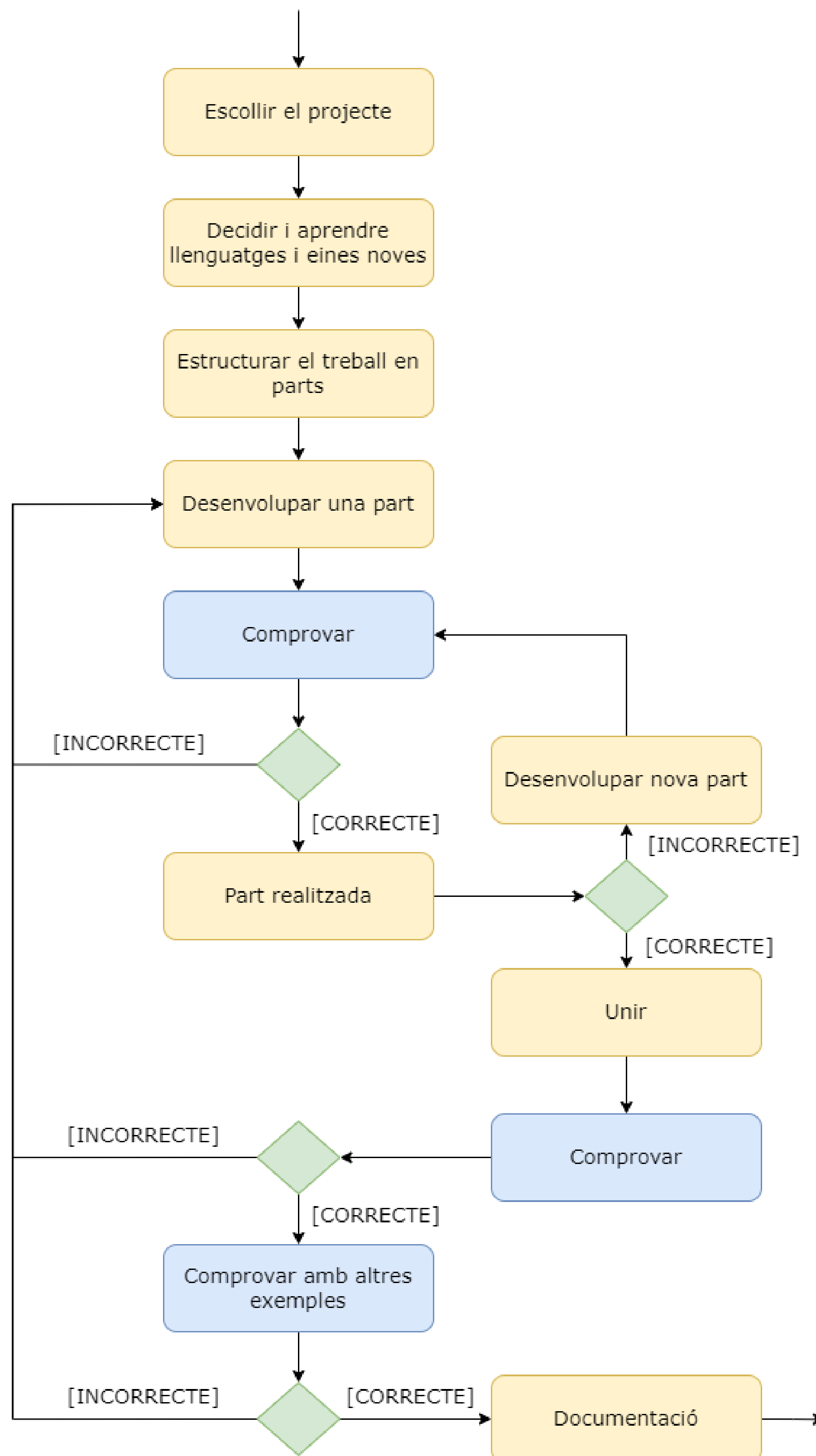


Figura 3.1: Diagrama de flux de la metodologia utilitzada

4.
Planificació

A l’hora de realitzar un projecte és important fer una valoració del temps que s’ha dedicat a la realització d’aquest. En aquest apartat es veurà la planificació que s’ha seguit durant la implementació del projecte.

4.1 Tasques planificades

4.1.1 Planificació del joc

Es defineix l’objectiu del joc, les característiques principals, i els diferents elements amb els que podrà interactuar el jugador al llarg del joc.

4.1.2 Estudi de diferents motors

Aquesta tasca consisteix en estudiar diversos motors de videojocs per aquest projecte i escollir-ne un, que va ser el motor Unity.

4.1.3 Estudi del motor gràfic Unity i del llenguatge C#

En aquesta s’estudia el funcionament del motor de videojoc Unity i el llenguatge de programació C# mitjançant tutorials i vídeos del internet.

4.1.4 Cerca, creació i preparació d’elements gràfics

Es cerquen tots els sprites necessaris per animar el joc: les caselles, les seleccions, etc. Si cal, s’animen, com és el cas de les unitats (les animacions són cadenes de sprites, o imatges en successió).

4.1.5 Disseny i implementació d’un prototip amb Unity

S’inicia el projecte creant un prototip on només hi hagi un mapa petit, i un personatge que es pugui moure pel mapa.

4.1.6 Disseny i implementació del joc

Es desenvolupa les interaccions entre els elements, el sistema i la lògica del joc.

4.1.7 Disseny i implementació de la interfície d’usuari

En aquesta tasca s’afegeixen les interfícies d’usuari per poder veure la informació dels elements que es troben dintre el joc.

4.1.8 Disseny i implementació de IA

Aquesta tasca tracta de dissenyar i implementar l'IA del joc per tal que el jugador pugui jugar contra un adversari.

4.1.9 Cerca, creació i integració de música

Es busca els elements de so a utilitzar per les diferents escenes del joc.

4.1.10 Verificació i proves dels algoritmes desenvolupats

Es comprova que els algoritmes implementats funcionin de la manera esperada.

4.1.11 Creació d'una demo del videojoc

Un cop tot testejat i que tot funcionin correcte, es crea una demo del videojoc que compti amb tots els elements realitzats fins el moment.

4.1.12 Documentació

La documentació és una tasca constant que s'ha de portar a terme durant tot el projecte. Els diferents mètodes, funcions i idees que van apareixent durant el desenvolupament s'han de plasmar a la documentació.

4.2 Temps estimat

Tasca	Dies	Dependències
(**T1**) Estudi de diferents motors	2	Cap
(**T2**) Estudi de Unity i C#	5	T1
(**T3**) Identificació d'usuaris	1	T2
(**T4**) Documentació de requisits	1	T3
(**T5**) Planificar el videojoc	7	T4
(**T6**) Disseny del nivell	18	T5
(**T7**) Efecte de parallax	3	Cap
(**T8**) Checkpoints	3	T7
(**T9**) Altres elements de l'escena	10	T8
(**T10**) Disseny i implementació de la interfície d'usuari	14	T5, T8
(**T11**) Core i Core Components	14	T5, T8
(**T12**) Màquina d'estats del personatge	21	T11
(**T13**) Màquina d'estats dels enemics	21	T10, T11
(**T14**) Implementació de la lluita del Boss	15	T12, T13
(**T15**) Creació de la música	2	T14
(**T16**) Interfícies i interacció entre personatge i enemics	19	T12, T13
(**T17**) Sincronització de la música amb el videojoc	15	T15
(**T18**) Preparació, cerca i creació d'elements gràfics	13	T16
(**T19**) Disseny i implementació d'un prototip	58	T17, T18
(**T20**) Verificació i proves dels algoritmes	4	T19
(**T21**) Documentació	120	T20

Figura 4.1: Taula de temps estimat

Hem realitzat un diagrama de Gantt per a il·lustrar els temps estimats de les tasques a realitzar:

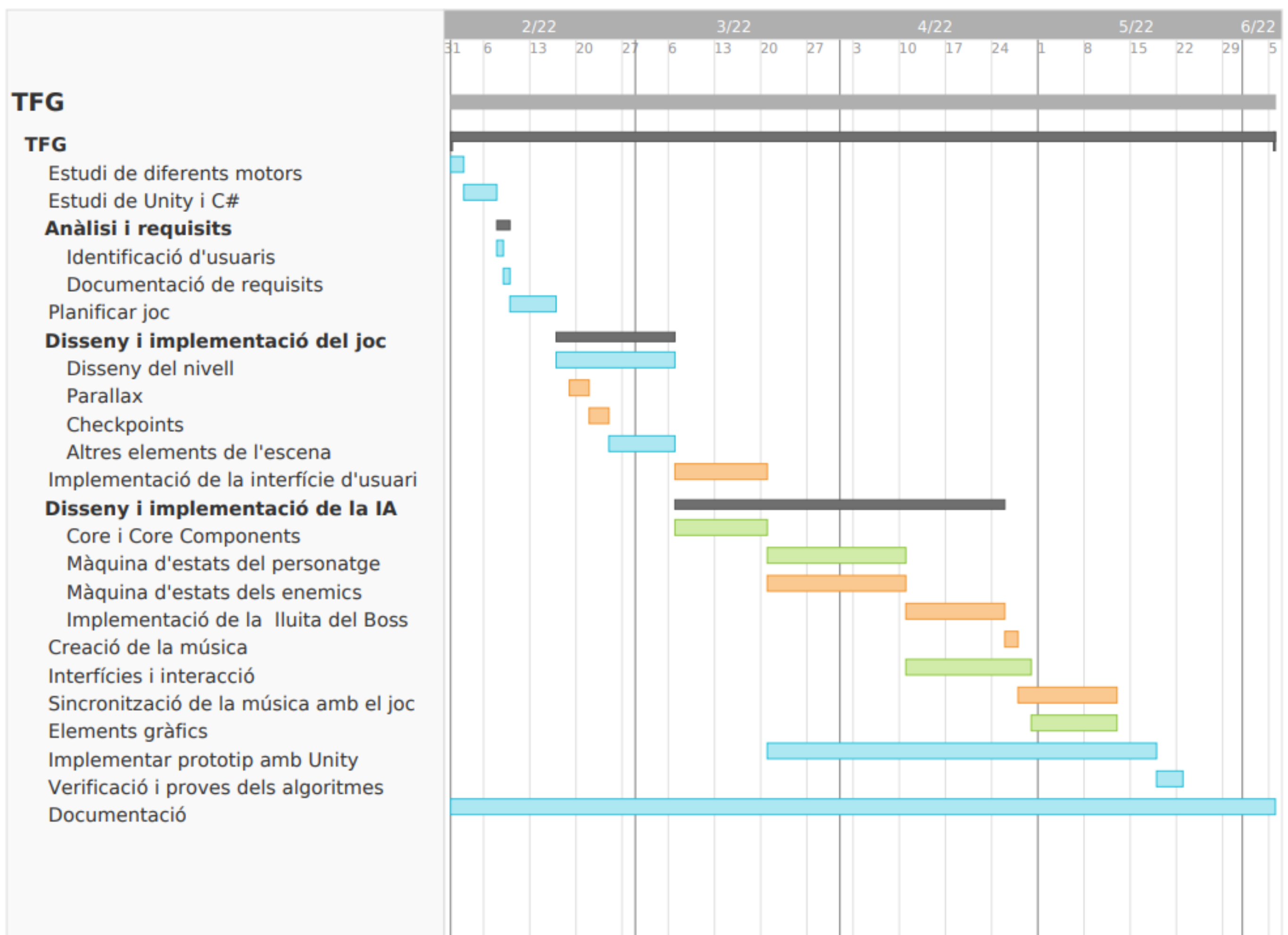


Figura 4.2: Diagrama de Gantt

Els colors representen el següent:

- **Blau**: tasques comunes
- **Verd**: tasques d'en Pol
- **Taronja**: tasques d'en Quim

5.
Marc de treball i conceptes previs

En aquest apartat veurem què és un motor de videojoc, els motors de videojocs més utilitzats avui en dia, i finalment el motor que s'ha escollit per desenvolupar el projecte i el perquè.

5.1 Introducció als motors de videojocs

Un motor de videojocs fa referència a una aplicació de programari que permeten dissenyar, crear i representar videojocs. Els principals dispositius per on es fan funcionar els videojocs resultants són les videoconsoles i els ordinadors. Així doncs, l'objectiu dels motors de videojocs és facilitar-li la feina als desenvolupadors (i d'aquesta manera no s'han de preocupar d'aspectes més tècnics com el funcionament dels nivells més baixos del sistema). Tot i això, la majoria de motors gràfics de renom també ofereixen eines per a programar a un nivell més baix.

Per tant, per un desenvolupador de videojocs és molt important escollir un motor a l'hora de desenvolupar el videojoc que té pensat crear.

Aquests són els avantatges de treballar amb un motor de videojocs:

- **Suport, comunitat i assets**

  Aquest aspecte pot ser un gran motiu per a decidir si utilitzar un motor de videojocs o no. Els desenvolupadors de videojocs estan acostumats a interactuar entre ells, preguntar-se i respondre qüestions de desenvolupament. Solen formar part de comunitats que comparteixen coneixement (i la majoria dels membres usen motors de videojocs). A més, la gran majoria dels motors de videojocs ofereixen ajuda i documentació sobre els seus components. També es solen oferir exemples d'ús, plantilles de projectes i altres recursos que ajuden als creadors de videojocs a manejar l'eina. Per últim, els motors de videojocs solen proporcionar botigues de recursos (Asset Stores) les quals ajuden a agilitzar el desenvolupament. S'hi solen vendre eines, plugins, sprites, etc. Aquests són desenvolupats per la mateixa empresa o per altres desenvolupadors.

- **Plataformes i fragmentació**

  Actualment, els desenvolupadors intenten oferir els jocs en el major nombre possible de plataformes (com iOS, Android, Web, PlayStation, etc.). Els motors solen oferir la possibilitat d'exportar els jocs fàcilment a alguna d'aquestes plataformes.

Un altre aspecte a tenir en compte és la gran varietat de resolucions i formats de pantalla que existeixen avui en dia. Poden ser des de 380x460 píxels fins a 1920x1080 o fins i tot més. I també la orientació (pot ser vertical o horitzontal). Els motors de joc solucionen la majoria d'aquests problemes.

També s'han de tenir en compte les plataformes de desenvolupament. Moltes de les eines disponibles són compatibles amb algunes de les més extenses: Windows, Mac i Linux. Hi ha una gran tendència a utilitzar la Web com a plataforma per a usar motors de videojocs (normalment basats en HTML5).

- **Facilitat d'ús, curva d'aprenentatge i productivitat**

  Un dels avantatges que ofereixen els motors de videojocs és el de la facilitat d'ús. La idea és que els motors de videojocs ofereixen una sèrie d'eines que faciliten tant la programació com la part gràfica. A més, l'objectiu de les persones que dissenyen aquestes eines és fer que siguin fàcils i àgils de fer servir, que siguin intuïtives i amb una corva d'aprenentatge el més baixa possible.La idea és que sigui més fàcil aprendre crear videojocs amb motors que no pas, per exemple, a crear videojocs amb llenguatge C++ (on la corva d'aprenentatge és força alta). La majoria dels motors més populars disposen d'infinitat de tutorials en línia. D'aquesta manera, l'aprenentatge i la resolució de dubtes és molt més eficaç.

- **Cost assequible**

  Aparentment, pagar per un software de tercers es pot tornar un cost extra que pot ser fàcilment eludible si creem un videojoc des de zero. Actualment els motors de videojocs han baixat dràsticament el preu de les llicències, i la majoria tenen un model de llicència gratuita. Per aquest motiu, aquest model està molt bé per a desenvolupadors Indie o per a estudiants que volen tenir a l'abast motors de videojoc potents a cost zero.

Per part contrària, també hi ha inconvenients:

- **Baix poder de control**

  Tot i que la majoria són avantatges, també tenen inconvenients. Un d'ells és que sempre existeix algun aspecte que no s'ofereix o que no s'ajusta de la manera en la que el creador de videojocs la vol o necessita per a treballar. Moltes d'aquestes limitacions solen ocórrer ja que a vegades es vol tenir un alt poder de control sobre el que es programa. Si un desenvolupador està acostumat a programar bit a bit es

pot veure negativament afectat per aquestes limitacions tècniques que presenten la majoria de motors de videojocs. Tot i això, els desenvolupadors dels motors de videojocs escolten aquestes crítiques i a mesura que van fent actualitzacions ja intenten incorporar noves eines que ajuden a cobrir aquestes mancances.

Hem vist que un motor de videojocs permet “amagar” les seves capes de baix nivell.

Així doncs, a la **Figura 5.1** descriurem la seva estructura interna:

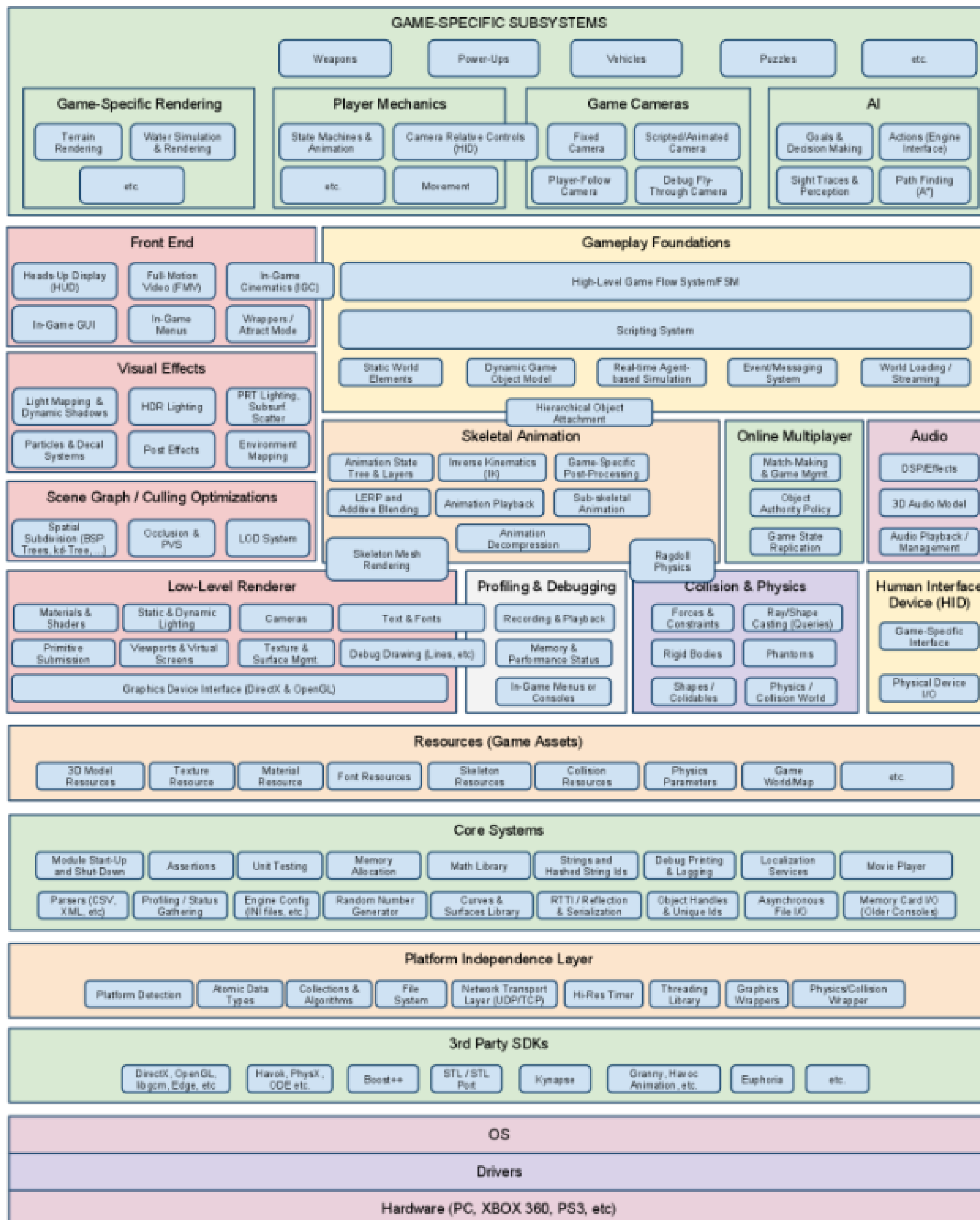


Figura 5.1: Diagrama de l’estructura interna d’un motor de videojocs

5.2 Motors de videojocs

En aquest apartat repassarem les principals característiques d'uns quants dels motors de videojocs més populars avui en dia.

5.2.1 Unity

El motor de videojocs multiplataforma Unity va ser llançat per Unity Technologies per primera vegada a la Conferència Mundial de Desenvolupadors d'Apple el 2005. Va ser construït exclusivament per a funcionar i generar projectes en equips de la plataforma Mac.

Cinc anys més tard es va llançar Unity 3 i va començar a introduir més eines que els estudis d'alta gamma solien oferir. Al 2015 es va llançar Unity 5, desenvolupat per a creadors de videojocs amb més espectatives. El motor de videojocs ha anat evolucionant fins a arribar a la versió vigent: Unity 2021. Es pot veure el logo actual de l'empresa a la **figura 5.2**.

Figura 5.2 : Logo de Unity

Com a característiques principals del Unity, podem destacar:

- Es pot fer servir juntament amb altres programes com: Blender, 3ds Max, Maya, Adobe, Cinema 4D, etc.
- El motor gràfic utilitza OpenGL, Direct3, OpenGL ES i interfícies propietàries
- Es poden escriure Shaders de tres maneres diferents: Surface shaders, Vertex ó Fragment Shaders
- Els programadors solen utilitzar UnityScript, C# ó Boo.
- Actualment és una bona opció tant per videojocs 2D com 3D.

Unity compta amb una versió gratuïta, on s'hi poden fer la gran majoria de feines que es proporcionen.

Un gran avantatge que té aquest motor de videojocs és que compta amb una Asset Store (la qual es va llançar l'any 2010). Allà s'hi poden trobar des de models 3D o

textures fins a música i efectes de so. A més, es possibilita el comerç a desenvolupadors i jugados mitjançant la plataforma DMarket.

Les **figures 5.5, 5.4, 5.5, 5.6 i 5.7** son imatges d'alguns dels videojocs més populars que s'han creat amb el motor de videojocs Unity.

- **League of Legends: Wild Rift**

Figura 5.3 : Gameplay de League of Legends: Wild Rift

- **Hollow Knight**

Figura 5.4 : Gameplay de Hollow Knight

- **Ori and the Will of Wisps**

Figura 5.5 : Gameplay de Ori and the Will of Wisps

- Phasmophobia

Figura 5.6 : Gameplay de Phasmophobia

- Pokémon GO

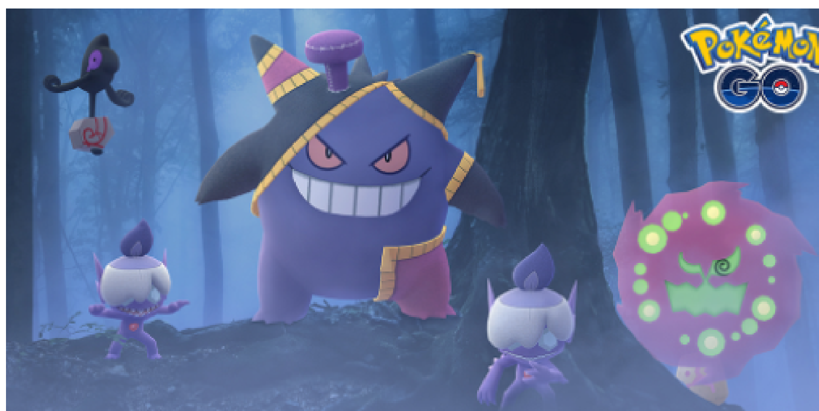


Figura 5.7 : Imatge promocional de Pokémon GO

5.2.2 Godot Engine

El motor de videojocs Godot és lliure i de codi obert. Permet treballar en 2D i en 3D. El motor funciona en sistemes Windows, OSX, Linux i BSD. Permet exportar els videojocs creats a PC (Windows, OS i Linux) i per a telèfons mòvils i tablets (Android, iOS), i per a HTML5.

Godot va ser creat l'any 2001, i va passar a ser de codi obert el febrer del 2014. Com a curiositat, el nom de "Godot" està relacionat amb la obra teatral 'Esperando a Godot' de Samuel Beckett. El logo de Godot es representa a la **figura 5.8**.

Figura 5.8 : Logo de Godot Engine

Com a característiques principals de Godot hi ha les següents:

- Renderització amb motor gràfic OpenGL ES
- Editor visual per a sombrejats
- Editor d'animació sofisticat
- Motor 2D sofisticat, independent i complert
- Suport a múltiples plataformes
- Suport per a varis llenguatges de programació

A les **figures 5.9, 5.10 i 5.11** es veuen imatges d'alguns dels videojocs més populars que s'han creat amb el motor de videojocs Godot.

- **Gun-Toting Cats**

Figura 5.9 : Gameplay de Gun-Toting Cats

- **Anthill**

Figura 5.10 : Gameplay de Anthill

- **Tanks of Freedom**

Figura 5.11 : Gameplay de Tanks of Freedom

5.2.3 Unreal Engine

Unreal Engine és un motor de videojocs creat per a l'empresa Epic Games. La primera versió es va llançar l'any 1998 amb un videojoc de tir en primera persona anomenat Unreal. Des de llavors, el motor de videojocs ha anat evolucionant, traient Unreal Engine 2 al 2002, Unreal Engine 3 al 2006, Unreal Engine 4 al 2015 i finalment Unreal Engine 5 al 2021 (el qual segueix vigent fins a dia d'avui). El logo de l'empresa queda il·lustrat a la figura 5.12

Figura 5.12 : Logo de Unreal Engine

Aquestes són algunes de les característiques més destacables de Unreal Engine (més concretament Unreal Engine 4 i 5):

- Motor gràfic molt potent
- Bona associació amb la realitat virtual
- Es pot editar en temps real
- Es poden crear jocs de gran envergadura i complexitat ja que es programa en C++
- És gratuït

Alguns dels jocs més populars que s'han fet amb el motor de videojocs Unreal Engine es poden veure a les figures **5.13, 5.14, 5.15 i 5.16**.

- **Abzû**

Figura 5.13 : Gameplay de Abzû

- **Ark: Survival Evolved**

Figura 5.14 : Gameplay de Ark: Survival Evolved

- **Days Gone**

Figura 5.15 : Gameplay de Days Gone

- **Tekken 7**

Figura 5.16 : Gameplay de Tekken 7

5.3 Motor escollit

Havent fet un estudi previ dels motors de videojocs i havent tingut en consideració els aspectes mencionats prèviament, hem decit escollir el motor de Unity pels següents motius:

- Té una comunitat molt activa, el qual fa que sigui molt ràpida la cerca de tutorials i altre tipus d'informació i ajuda.

- Proporciona una documentació excel·lent.

- Soporta videojocs en 2D, i té un apartat de creació específic per a jocs en aquest format (Unity 2D).

- Es programa en C#. El llenguatge C# és molt utilitzat al món professional i ens pot servir per a perfeccionar-lo.

- Té una corba d'aprenentatge molt baixa, ajudant al creador amb moltes eines de fàcil enteniment i utilització.

6.
Requisits del sistema

En aquest apartat veurem els requeriments de l'aplicació. Els hem dividit en dues categoriess: els requeriments funcionals (descriuen quins són els serveis que ens oferirà l'aplicació independentment de la implementació) i els no funcionals (ens informen sobre les restriccions que venen imposades pel client o pel propi problema).

6.1 Requeriments funcionals

En aquest apartat es descriuran els requeriments funcional, els serveis que oferirà aquesta aplicació, sense tenir en compte la implementació.

6.1.1 Llistat de requeriments funcionals

Aquests són els requeriments funcionals que ha de satisfer l'aplicació:

- El jugador haurà de poder ajustar el volum de la música i dels efectes de so del videojoc al iniciar-lo
- El jugador haurà de poder iniciar una partida al iniciar el videojoc
- El jugador haurà de poder sortir del videojoc al iniciar-lo
- El jugador haurà de poder ajustar el volum de la música i dels efectes de so del videojoc durant la partida
- El jugador haurà de poder controlar un personatge i les mecàniques
- El jugador haurà de poder entendre la relació entre la música i el gameplay (sincronització)

6.1.2 Identificació dels actors

Ara cal identificar quins són els actors de l'aplicació. Entenem com actor una entitat externa a l'aplicació com podria ser una persona, un sistema, etc, que té un rol concret a l'hora d'interactuar amb l'aplicació.

En el nostre cas, només hi ha present un únic actor, que serà l'usuari que interactua amb tot el sistema (el jugador). A la **figura 6.1** es pot veure un esquema d'aquesta interacció.

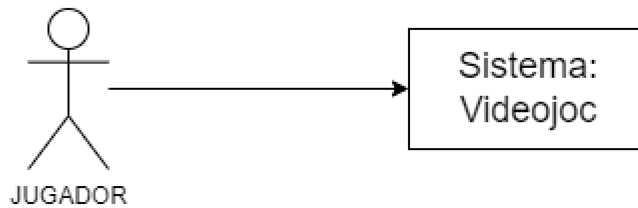


Figura 6.1: Diagrama d'interacció de l'actor amb el sistema

Tots els usuaris que utilitzin l'aplicació disposen dels mateixos privilegis.

6.2 Requeriments no funcionals

Els requeriments no funcionals no fan referència a funcionalitats que pot fer el jugador, sinó a aspectes més apartats del territori del jugador, com les característiques que s'han de tenir en compte a l'hora de dissenyar el sistema (com la seguretat del sistema, hardware necessari pel funcionament òptim, etc).

Els requeriments no funcionals són aquells que fan referència a la qualitat i característiques que s'han de tenir en compte a l'hora de dissenyar el sistema, com ara els recursos necessaris per a un bon funcionament, la seguretat necessària que ha de tenir el sistema, etc.

Pel nostre projecte no són necessàries mesures de seguretat extres a l'hora de guardar dades confidencials o de controlar l'accés a l'aplicació. A més, al haver realitzat el projecte amb un motor de videojocs, les possibles restriccions de la plataforma ja són les que el propi motor té.

Pel que fa als dispositius d'entrada, l'usuari pot interactuar amb el sistema mitjançant el ratolí i el teclat.

Pel que fa als requeriments tècnics, hem realitzat la implementació del projecte i les proves resultants d'aquest en dos ordinadors diferents. Aquests, tenen aquesta configuració:

ORDINADOR 1

- **Processador**: Windows 10 Pro
- **Memòria RAM**: 32 GB
- **Sistema Operatiu**: Intel(R) Core(TM) i7-10700K a 3.80GHz
- **Tarjeta gràfica**: Nvidia Geforce RTX 2070
- **Memòria SSD**: 1TB

ORDINADOR 2

- **Processador**: Windows 10
- **Memòria RAM**: 16GB
- **Sistema Operatiu**: Intel Core i7-8750 CPU 2.2GHz
- **Tarjeta gràfica**: NVIDIA GeForce GTX 1050
- **Memòria SSD**: 1TB

Ara bé, aquests no són els requeriments tècnics mínims per a poder executar aquest videojoc. Els requeriments mínims són els que té el motor de videojocs que hem fet servir (Unity). Aquests requeriments es veuen a la **figura 6.2**.

- **CPU**: Arquitectura x86, x64 amb suport de conjunt d'instruccions SSE2
- **Tarjeta gràfica**: Compatible amb DX10, DX11, DX12.
- **Sistema operatiu**: Windows 7 (SP1+), Windows 10 ó Windows 11

Desktop

Operating system	Windows	Universal Windows Platform	macOS	Linux
Operating system version	Windows 7 (SP1+), Windows 10 and Windows 11	Windows 10+, XBox Series X|S, HoloLens	High Sierra 10.13+	Ubuntu 20.04, Ubuntu 18.04, and CentOS 7
CPU	x86, x64 architecture with SSE2 instruction set support.	x86, x64 architecture with SSE2 instruction set support, ARM, ARM64.	Apple Silicon, x64 architecture with SSE2.	x64 architecture with SSE2 instruction set support.
Graphics API	DX10, DX11, DX12 capable.	DX10, DX11, DX12 capable GPUs.	Metal capable Intel and AMD GPUs	OpenGL 3.2+, Vulkan capable.
Additional requirements	Hardware vendor officially supported drivers. For development: IL2CPP scripting backend requires Visual Studio 2015 with C++ Tools component or later and Windows 10+ SDK.	Hardware vendor officially supported drivers. For development: Windows 10+ (64-bit), Visual Studio 2015 with C++ Tools component or later and Windows 10+ SDK.	Apple officially supported drivers. For development: IL2CPP scripting backend requires Xcode. Targeting Apple Silicon with IL2CPP scripting backend requires macOS Catalina 10.15.4 and Xcode 12.2 or newer.	Gnome desktop environment running on top of X11 windowing system Other configuration and user environment as provided stock with the supported distribution (such as Kernel or Compositor) Nvidia and AMD GPUs using Nvidia official proprietary graphics driver or AMD Mesa graphics driver.
	For all operating systems, the Unity Player is supported on workstations, laptop or tablet form factors, running without emulation, container or compatibility layer.			

Figura 6.2: Requeriments tècnics mínims per utilitzar Unity

7.
Estudis i decisions

En aquest apartat es fa esmena de cada un dels programaris que s'ha utilitzat durant el desenvolupament del projecte. Per a cada un d'ells es realitza una descripció i s'especifica l'ús que li hem donat, així com a quina part del projecte ha sigut més útil. Cal recordar que el motor de jocs Unity, s'explica a l'apartat 5.2 Motors de videojocs.

7.1 Visual Studio Code

Gratuït i de codi obert, Visual Studio Code és l'editor de codi o IDE (Integrated Development Environment) que hem escollit perquè ens ajudi a l'hora d'escriure la lògica del nostre joc. Es tracta d'un editor molt popular desenvolupat per Microsoft, el qual disposa de gran quantitat d'eines que faciliten el treball al programador. Entre aquestes hi ha la depuració, el control de versions de Git integrat, el resaltat de sintaxis, la finalització intel·ligent, el desplaçament ràpid a definicions o la refactorització de codi.
Hem trobat l'ús d'aquest editor molt encertat per treballar juntament amb Unity. Visual Studio Code disposa de la possibilitat d'afegir extensions d'una manera fàcil i ràpida gràcies al *VS Code Extension Marketplace,* la qual cosa ens ha permès accedir a extensions de Unity que faciliten encara més el treball d'escriure codi. El logo del Visual Studio Code es mostra a la **figura 7.1**.

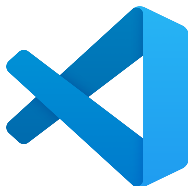


Figura 7.1 : Logo de Visual Studio Code

7.2 Audacity

L'Audacity és un programa gratuït per a l'edició d'àudio. Ofereix una interfície senzilla de manera que fins i tot la gent que no està acostumada a editar música ho pot fer amb certa senzillesa.

En el nostre cas hem fet servir l'Audacity per a comprimir la música exportada del Bosca Ceoil. D'aquesta manera, la música sona més "forta" ja que no el volum era força baix al exportar-ho del Bosca Ceoil. El logo de l'Audacity es mostra a la **figura 7.2**.

Figura 7.2 : Logo de l’Audacity

7.3 Bosca Ceoil

Bosca Ceoil és un programari gratuït i de codi obert que ha buscat oferir un entorn de producció de música senzill i fàcil d'utilitzar fins i tot per la gent que no està familitarizada amb cap mena de *software* musical. Aquesta eina ofereix una interfície senzilla que permet, a grans trets, escollir quines notes volem que toquin els instruments i escollir quins instruments toquen les notes. Les notes que escribim a la pista es poden encapsular en patrons per tal de duplicar-los o fer-los sonar rere altres pistes i, mica en mica, acabar formant una peça musical.
Pel que fa al nostre projecte, aquest ha estat el programa des del qual s’ha creat la música que sona durant el transcurs del nivell i dicta el comportament de nombrosos elements. La interfície del Bosca Ceoil es mostra a la **figura 7.3**.

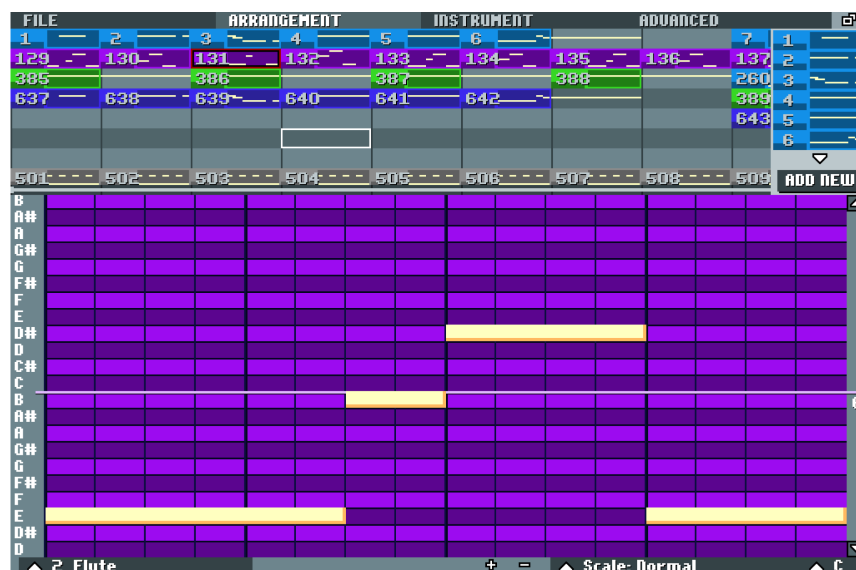


Figura 7.3 : Interfície de Bosca Ceoil

7.4 FMOD Studio

FMOD Studio és un motor d'efectes de so orientat als videojocs i aplicacions. Va ser creat per la companyia Firelight Technologies, que estan presents a nombrosos sistemes operatius.
Ofereix totes les eines necessàries per dissenyar, crear i optimitzar l'àudio adaptatiu.
També facilita la integració d'àudio a qualsevol joc. Permet una integració immediata amb Unity i Unreal Engine o una integració més lenta amb un motor de joc personalitzat mitjançant les versions C++ o C# de l'API.
Pel que fa al projecte, l'hem fet servir per a sincronitzar la música de fons amb elements del gameplay (plataformes, trampes, etc.). FMOD Studio permet posar marques a instants de temps de la música, a la vegada que àrees de bucle, etc. Des del Unity, detectarem aquestes marques i un cop detectades cridarem a mètodes d'objectes (i d'aquesta manera sincronitzarem el ritme de la música amb el joc). El logo de l'FMOD Studio es mostra a la **figura 7.4**.

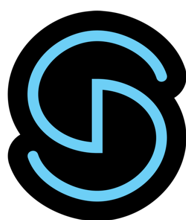


Figura 7.4 : Logo de l'FMOD Studio

7.5 Piskelapp

Piskelapp és un editor de *sprites* en línia, amb suport pels navegadors Chrome, Firefox, Edge i Internet Explorer 11. Està enfocat a treballar amb l'estil de *Pixel Art*, oferint un canvas específic per al seu ús, així com un conjunt d'eines que faciliten la creació i edició d'animacions tradicionals *frame* a *frame.* Aquesta ha sigut la principal eina de creació i modificació dels *sprites* del nostre joc. Una de les característiques d'aquest editor que més útil ens ha sigut és la de poder modificar tots els píxels d'un cert color, canviant-lo per un altre. D'aquesta manera hem pogut canviar el color dels *sprites* perquè s'adaptin a la paleta de colors que volíem aconseguir. Es pot veure un exemple d'edició d'un spritesheet amb el Piskelapp a la **figura 7.5**.

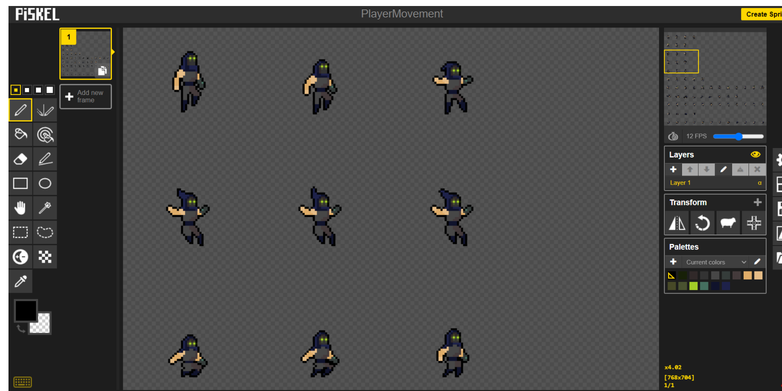


Figura 7.5 : Edició de l'aspecte del personatge amb Piskelapp

7.6 Google Drive

Google Drive ofereix el servei de Google Docs Editors, dins el qual hi és present aquest processador de text en línia. Aquesta eina permet als usuaris editar i crear documents en línia mentres col·laboren amb altres usuaris a temps real. A més, disposa d'un historial de col·laboracions i revisions que mostra els diferents canvis que s'han anat afegint al document.
Així doncs, aquest processador de text és ideal per a treballar en grup i és el que hem utilitzat per tal d'escriure la memòria del projecte.
El logo del Google Docs es mostra a la **figura 7.6**.

Figura 7.6 : Logo de Google Docs

7.7 Diagrams.net

Aquest és un programari de codi obert i gratuït de dibuix de grafs, útil per a crear diagrames de tota mena. Permet l'exportació dels diagrames a GoogleDrive, al disc dur, al OneDrive, Dropbox i Github. L'hem utilitzat durant la redacció de la memòria, per crear diagrames i esquemes.
El logo de Diagrams.net es mostra a la **figura 7.7**.

Figura 7.7: Logo de Diagrams.net

8.
Anàlisis i disseny del sistema

8.1 Casos d'ús

En aquest apartat revisarem alguns dels diagrames de casos d'ús plantejats, fitxes i diagrames d'activitats.

8.1.1 Diagrames de casos d'ús

Hem trobat adequat fer diagrames de casos d'ús ja que aquests ajuden a il·lustrar el comportament del sistema des del punt de vista dels usuaris. Són descripcions de les funcionalitats del sistema independentment de la implementació.

8.1.1.1 Diagrama de casos d'ús del menú principal

A la **figura 8.1.1** es mostra el diagrama de casos d'ús del menú principal.

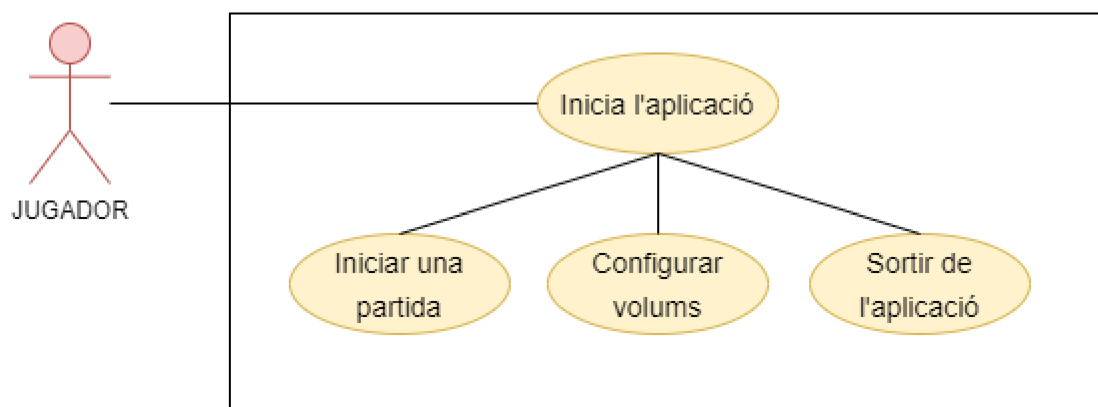


Figura 8.1.1 : Diagrama de casos d'ús del menú principal

Quan el jugador inicia l'aplicació es trobarà al menú principal. Des d'allà podrà iniciar una partida nova, configurar els volums o sortir de l'aplicació.

8.1.1.2 Diagrama de casos d'ús del menú de pausa

A la **figura 8.1.2** es mostra el diagrama de casos d'ús del menú de pausa.

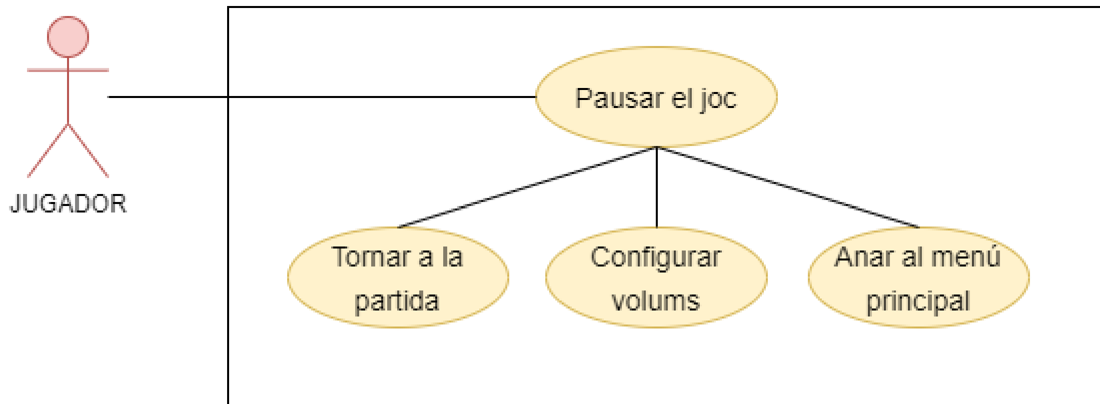


Figura 8.1.2 : Diagrama de casos d'ús del menú de pausa

Quan el jugador pausa el joc, podrà tornar a la partida, configurar els volums o anar al menú principal.

8.1.1.3 Diagrama de casos d'ús del menú de Game Over

A la **figura 8.1.3** es mostra el diagrama de casos d'ús del menú de Game Over.

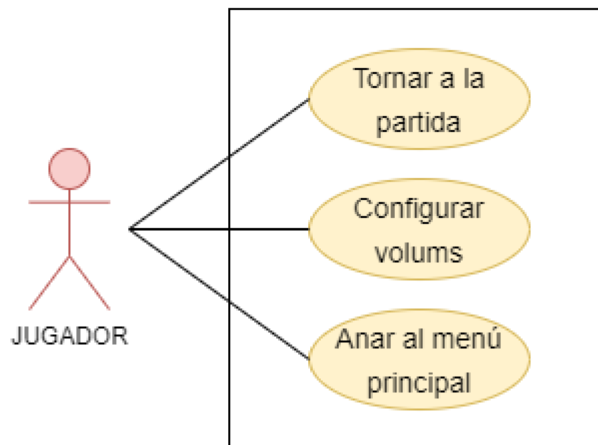


Figura 8.1.3 : Diagrama de casos d'ús del menú de Game Over

Quan el jugador perd, podrà tornar a començar la partida o tornar al menú principal.

8.1.2 Fitxes de casos d’ús

Per als casos d’ús anteriors, n’hem realitzat les seves fitxes:

Cas d’ús: iniciar l’aplicació (menú principal)	
Descripció	**Accedirem al menú principal, des d’on podem configurar les opcions de la partida**
Actor	**Jugador**
Precondició	**Cap**
Flux principal	**1. Iniciar sistema** **2. Mostrar menú principal**
Postcondició	**Aplicació iniciada correctament i menú principal mostrat**
Observacions	**Cap**

Cas d’ús: iniciar una partida	
Descripció	**Des del menú principal, el jugador podrá iniciar una nova partida**
Actor	**Jugador**
Precondició	**Haver iniciat l’aplicació**

Flux principal	**1 Iniciar sistema** **2. Mostrar menú principal** **3. Accedir a l'opció *Start***
Postcondició	**El jugador ha començat una partida nova**
Observacions	**Cap**

Cas d'ús: configurar volums	
Descripció	**Des del menú d'opcions del menú principal, el jugador pot ajustar els volums de la música i dels sons**
Actor	**Jugador**
Precondició	**Haver iniciat l'aplicació i haver seleccionat l'opció Options del menú principal**
Flux principal	**1 Iniciar sistema** **2. Mostrar menú principal** **3. Accedir a l'opció *Options*** **4. Mostrar el menú d'opcions** **5. Ajustar el volum de la música (si es desitja)** **6. Ajustar el volum dels sons (si es desitja)**
Postcondició	**El jugador ha ajustat els volums d'una posible partida**

Observacions	**Des del menú d'opcions, el jugador pot tornar al menú principal**

Cas d'ús: sortir de l'aplicació	
Descripció	**Des del menú principal, el jugador podrà sortir de l'aplicació**
Actor	**Jugador**
Precondició	**Haver iniciat l'aplicació**
Flux principal	**1 Iniciar sistema** **2. Mostrar menú principal** **3. Accedir a l'opció *Quit***
Postcondició	**El jugador ha sortit de l'aplicació**
Observacions	**Cap**

Cas d'ús: pausar el joc	
Descripció	**Des de la partida, el jugador podrà acceder al menú de pausa**
Actor	**Jugador**
Precondició	**Haver iniciat l'aplicació i començat una nova partida**

Flux principal	**1 Iniciar sistema** **2. Mostrar menú principal** **3. Accedir a l'opció *Start*** **4. Carregar la nova escena** **5. Si el jugador prem la tecla 'Esc'** **5.1 Mostrar el menú de pausa**
Postcondició	**El jugador ha accedit al menú de pausa**
Observacions	**Cap**

Cas d'ús: tornar a la partida	
Descripció	**Des del menú de pausa, el jugador pot desactivar-lo i tornar a la partida amb normalitat**
Actor	**Jugador**
Precondició	**Haver iniciat l'aplicació, començat una nova partida i accedit al menú de pausa**

Flux principal	**1 Iniciar sistema** **2. Mostrar menú principal** **3. Accedir a l'opció *Start*** **4. Carregar la nova escena** **5. Si el jugador prem la tecla 'Esc'** **5.1 Mostrar el menú de pausa** **5.2 Prémer de nou la tecla 'Esc' o seleccionar l'opció *Back***
Postcondició	**El jugador ha desactivat el menú de pausa**
Observacions	**Cap**

Cas d'ús: anar al menú principal	
Descripció	**Des del menú de pausa, el jugador pot sortir de la partida i tornar al menú principal**
Actor	**Jugador**
Precondició	**Haver iniciat l'aplicació, començat una nova partida i accedit al menú de pausa**

Flux principal	**1 Iniciar sistema** **2. Mostrar menú principal** **3. Accedir a l'opció *Start*** **4. Carregar la nova escena** **5. Si el jugador prem la tecla 'Esc'** **5.1 Mostrar el menú de pausa** **5.2 Selecciona l'opció *Main Menu***
Postcondició	**El jugador ha sortit de la partida i haaccedit al menú principal**
Observacions	**Cap**

8.1.3 Diagrames d'activitats

Finalment, hem realitzat dos diagrames d'activitats, un per il·lustrar el menú principal i l'altre pel menú d'opcions.

8.1.3.1 Diagrama d'activitats del menú principal

La **figura 8.1.4** representa el diagrama d'activitats del menú principal.

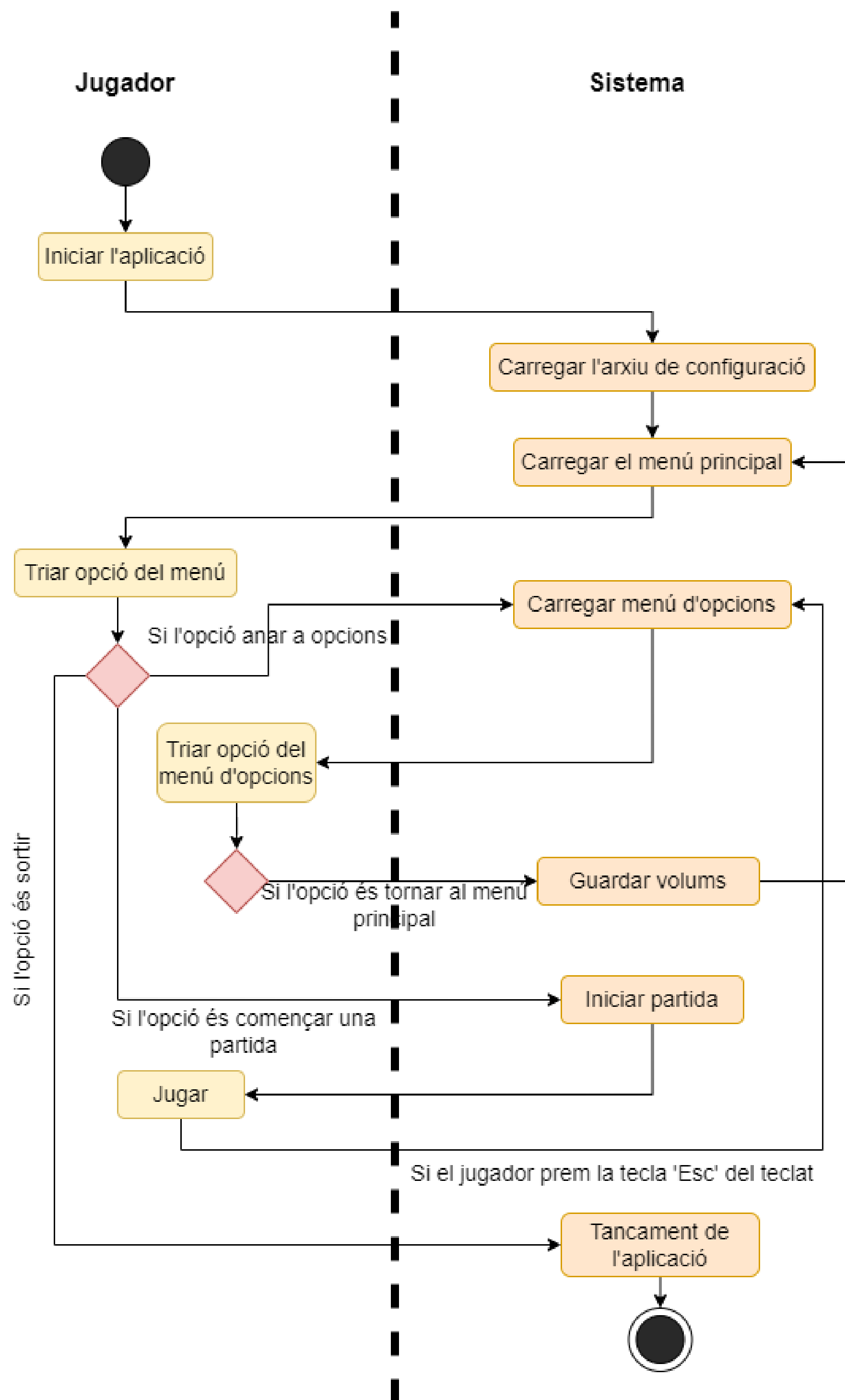


Figura 8.1.4 : Diagrama d'activitats del menú principal

8.1.3.2 Diagrama d'activitats del menú d'opcions

La **figura 8.1.5** representa el diagrama d'activitats del menú d'opcions.

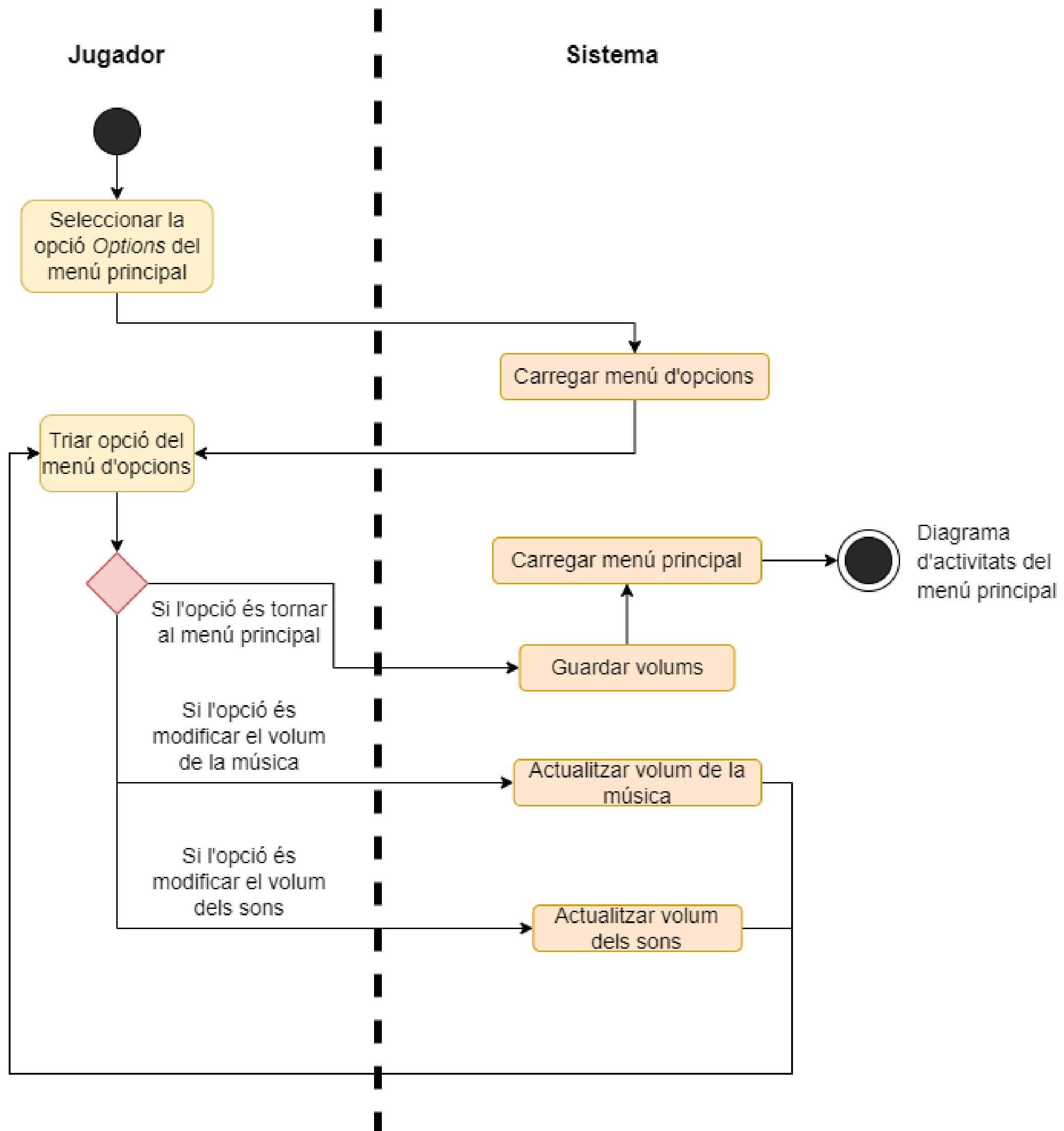


Figura 8.1.5 : Diagrama d'activitats del menú d'opcions

8.2 Interfícies

Nombrosos elements de l'escena del nostre joc comparteixen elements de comportament. Hem inclós les següents interfícies perquè les puguin implementar varis d'aquests elements i no haver de distingir entre aquests a l'hora de cridar certs mètodes. Aquestes són:

- **IDamageable**: La utilitzen tots els elements que puguin rebre mal.
- **IKnockbackable:** La utilitzen tots els elements que puguin ser empesos.
- **ICollector:** La utilitzen tots els objectes que necessitin poder obtenir, guardar i perdre col·leccionables.

Relacionada amb l'ús de les dues primeres interícies *IDamageable* i *IKnockbackable* hem creat la classe **DamagingObjects.** Aquesta és una classe de la qual poden heredar tots els objectes que volem que siguin capaços de fer mal, com el meteorit de la batalla amb el *Boss* o les fletxes dels enemics arquers.

La implementació d'aquestes interfícies i de la classe *DamagingObjects* s'explica en l'apartat Interfícies de implementació.

8.3 Core i Core Components

A l'utilitzar dues màquines d'estat per a entitats que tenen comportaments semblants, com són el Player i els enemics, moltes de les funcionalitats que s'han d'implementar perquè funcionin son iguals. Per posar un exemple, els estats del Player necessitaran canviar la velocitat del RigidBody del personatge de la mateixa manera que ho hauran de fer els estats dels enemics. Així doncs, per tal de crear un sistema més reutilitzable i evitar codi duplicat, s'incorpora el concepte de Core i Core Components a les nostres màquines d'estat.

Els **Core Components** son una col·lecció d'scripts que encapsulen funcionalitats bàsiques a les quals els estats d'entitats com el Player i els enemics necessitaran accedir. L'objectiu és crear components independents que puguem incloure fàcilment a noves entitats segons les necessitats que aquestes tinguin (mentre una entitat pot necessitar molts components, una altra potser només es beneficia de l'ús d'un sol), de manera que s'integrin al sistema que tenim implementat. Veure la **figura 8.2.1**.

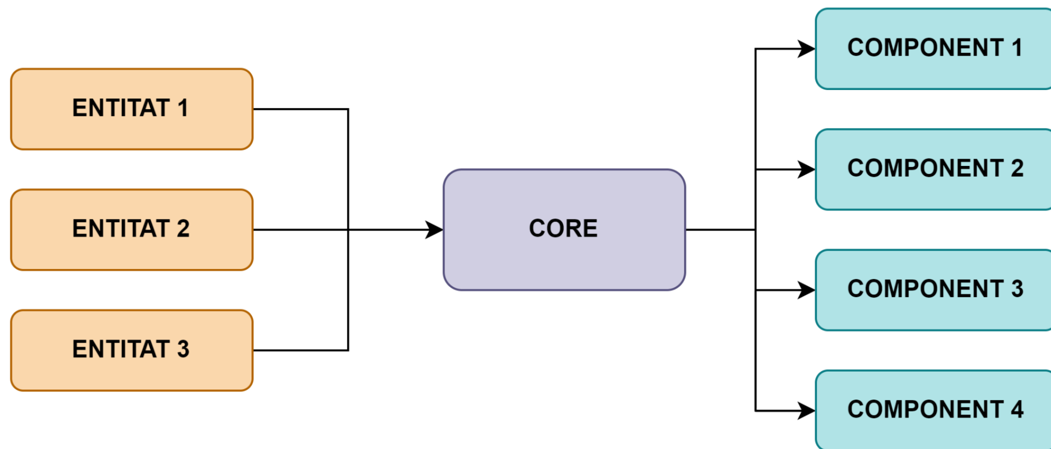


Figura 8.2.1: Diagrama de comunicació entre les entitats i els components a través del Core

Comptem amb un script **Core** al qual l'entitat tindrà referència. L'entitat donarà a cada un dels estats una referència a aquest objecte Core. Ja que el Core tindrà una llista amb els respectius **Core Components,** tots els estats poden accedir a les funcionalitats d'aquests en qualsevol moment. Veure la **figura 8.2.2**.

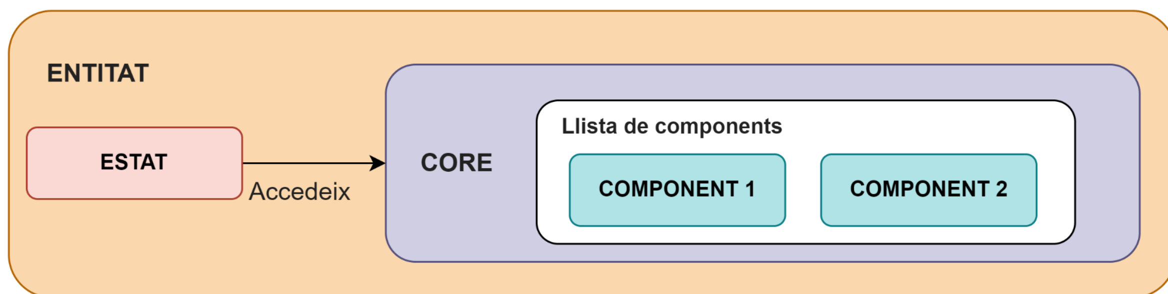


Figura 8.2.2: Estructura de comunicació entre els estats i els Core Components de l'Entitat

En les nostres màquines d'estat, hem incorporat els següents Core Components:

- **Movement:** Conté tota la funcionalitat necessària per a controlar el moviment de l'entitat (modificar velocitat, aturar, capgirar, etc.)

- **Collision Senses:** Conté els components necessaris per detectar els diferents elements de l'entorn que envolta a l'entitat en tot moment. Ens serveix per fer comprovacions sobre si estem tocant parets, el terra o cantonades.

- **Combat:** Utilitza un colisionador per tractar la lògica relacionada amb el combat. També s'encarrega de les interaccions físiques amb altres objectes de l'escena.

- **Appearance:** S'encarrega de implementar els events relacionats amb canvis en l'aparença de l'entitat.

- **Inventory:** Gestiona els objectes col·leccionables per l'entitat, així com les armes en el cas de utilitzar-ne.

- **Stats:** Implementa la lògica necessària per guardar, obtenir i modificar informació sobre les estadístiques de l'entitat.

8.4 Màquines d'estat

En el nostre projecte hem utilitzat dues màquines d'estats per implementar el moviment i el comportament, tant del personatge principal com de tots els enemics. En aquest apartat expliquem què és una màquina d'estats i mostrem l'estructura principal que segueixen les que hem implementat en el nostre videojoc.

8.4.1 Concepte de la màquina d'estats

Les entitats del nostre joc poden realitzar varies accions: saltar, caminar, atacar, etc. Si volguéssim definir el comportament d'aquestes en un sol script, necessitariem fer moltes comprovacions a cada cicle d'execució per tal de saber quina acció està realitzant i quina acció ha de realitzar a continuació. Això acabaria generant un sol programa molt llarg i inescalable.

Mitjançant l'ús de la màquina d'estats podem encapsular la condició actual de l'objecte o sistema en un **estat**, el qual decidirà el comportament. El codi de cada estat ens permet modificar el comportament de l'entitat de manera independent a la resta dels altres estats. Així doncs, la màquina d'estats s'encarrega d'assignar en quin estat està una entitat en cada moment, i aquest serà qui defininirà el comportament de l'entitat i el que farà que es canviï a un nou estat.

8.4.2 Estructura i funcionament general de la nostra màquina d'estats

La nostra màquina d'estats funciona sobre tres objectes (scripts) principals:

- **Màquina d'estats:** S'encarrega de guardar l'estat en el que es troba l'entitat en cada moment, així com de canviar entre els diferents estats.

- **Estats:** Contenen tota la lògica necessaria per definir el comportament de l'entitat durant l'espai de temps en el qual aquesta el tingui com a 'estat actual'. Tots els estats hereden d'una superclasse **Estat** que implementa el funcionament base de cada un.

- **Entitat:** Aquest és l'objecte que serà controlat per la màquina d'estats i que, conseqüentment, conté la pròpia màquina d'estats.

Per tal d'explicar de quina manera es comuniquen aquests tres objectes, cal entendre el funcionament de les **funcions esdeveniments** de Unity. Les **funcions d'esdeveniments** són mètodes que s'executen en un ordre predeterminat donat un *script*. Aquestes serveixen per a poder aplicar lògica en moments concrets de la nostra seqüència d'execució. Aquestes són les funcions d'esdeveniments principals que hem utilitzat en el nostre projecte:

- **Awake()**: La primera funció d'esdeveniments en executar-se al inicialitzar-se l'*script*. Es crida una sola vegada durant la vida d'aquest.
- **Start():** Es crida posteriorment a l'*Awake()*, però a diferència d'aquest, l'*Start()* no s'executa fins que l'objecte que contingui l'*script* estigui **actiu**.
- **FixedUpdate():** L'execució d'aquest mètode es pot donar una, cap o bé vàries vegades per cada *frame*, però sempre anirà en coordinació amb el motor de físiques de Unity. Així doncs, convé posar-hi la lògica referent al tractament amb la física per tal que tot el que estigui afectat per aquesta vagi sincronitzat.
- **Update()**: Aquest mètode es crida una vegada cada *frame* independentment de la relació *frames*/segon durant l'execució del joc.

Així doncs, la lògica que defineix el comportament d'un objecte s'ha d'escriure dins d'aquestes funcions d'esdeveniments. Aquest és el concepte principal per entendre la comunicació entre els objectes que formen la nostra màquina d'estats.

8.4.3 Script StateMachine

La màquina d'estats en sí conté una lògica molt simple:

- Guarda un Estat com a *CurrentState* (estat actual).
- Conté una funció *Initialize()* que es crida un sol cop a l'inici per establir el primer estat en el que es trobarà l'Entitat.
- Conté una funció *ChangeState(),* la qual canvia l'estat actual al nou estat que rep per paràmetre:

Veure a la **figura 8.2.3**.

```
public class StateMachine
{
    public State CurrentState { get; private set; } //Public getter, private setter

    public void Initialize(State startingState){
        CurrentState = startingState;
        CurrentState.Enter();
    }

    public void ChangeState(State newState){
        CurrentState.Exit();
        CurrentState = newState;
        CurrentState.Enter();
    }
}
```

Figura 8.2.3: Script de la màquina d'estats

8.4.4 Script Estat (Superclasse)

Aquest *script* conté la lògica comuna a tots els diferents estats. Cal recordar que cada Estat heredarà d'aquesta classe i afegirà més lògica a la que es mostra a continuació.

A partir del constructor, l'Estat rep els objectes necessaris pel funcionament:

- La pròpia **Entitat**.
- La **Màquina d'estats**.
- El **nom de l'animació corresponent a l'Estat**: Aquest paràmetre serveix perquè cada estat sigui capaç de interactuar amb l'Animator i iniciar l'execució de la seva corresponent animació, així com aturar-la.

Mètodes

Cada vegada que es canvia l'estat actual de la màquina d'estats, s'executa la funció *Exit()* de l'estat anterior i posteriorment la funció *Enter()* del nou estat. Aquestes dues funcions implementen la lògica que s'executa al sortir d'un estat i al entrar-ne:

- **Exit():** Atura l'execució de l'animació corresponent a l'estat.

- **Enter()**
  - Inicia l'execució de l'animació corresponent a l'estat.
  - Inicialitza el paràmetre **startTime**, que serveix per tenir control sobre el moment en el qual s'ha entrat en l'estat.
  - Crida l'execució de *DoChecks()*

- **DoChecks():** Buit a la superclasse, aquest mètode servirà per realitzar les deteccions de parets, el terra i elements de l'entorn a cada estat que ho necessiti.

- **LogicUpdate():** Buit a la superclasse, aquest mètode contindrà a cada estat la lògica que s'executarà a cada *frame*. Aquí dins és on també s'han de fer les comprovacions necessàries per decidir si s'ha de canviar d'estat a un de nou.

- **PhysicsUpdate():** Aquest mètode crida a l'execució de *DoChecks()*. A part, els estats hi implementaran la lògica relacionada amb la física.

Veure script de State a la **figura 8.2.4**.

```
public class State
{
    protected FiniteStateMachine stateMachine;
    protected Entity entity;
    protected Core core;

    public float startTime { get; protected set; }
    protected string animBoolName;

    public State(Entity entity, FiniteStateMachine stateMachine, string animBoolName){
        this.entity = entity;
        this.stateMachine = stateMachine;
        this.animBoolName = animBoolName;
        core = entity.Core;
    }

    public virtual void Enter(){
        startTime = Time.time;
        entity.anim.SetBool(animBoolName, true);
        DoChecks();
    }

    public virtual void Exit(){
        entity.anim.SetBool(animBoolName, false);
    }

    public virtual void LogicUpdate(){
    }

    public virtual void PhysicsUpdate(){
        DoChecks();
    }

    public virtual void DoChecks(){
    }
}
```

Figura 8.2.4: Script de State

8.4.5 Script Entitat

En aquest apartat només s'expliquen els elements d'aquest *script* que estàn relacionats directament a la màquina d'estats. La resta d'elements s'especifiquen en les explicacions de cada màquina d'estats concreta.

- **Awake()**: S'inicialitza la màquina d'estats.
- **Start():** Inicialitza el primer estat a la màquina d'estats.
- **Update():** Es crida a la funció *LogicUpdate()* de l'estat actual.
- **FixedUpdate():** Es crida la funció *PhysicsUpdate()* de l'estat actual.

Veure la **figura 8.2.5**.

```
private void Awake(){
        StateMachine = new PlayerStateMachine();
}

private void Start(){
    StateMachine.Initialize (IdleState);
}

private void Update(){
    StateMachine.CurrentState.LogicUpdate();
}

private void FixedUpdate(){
    StateMachine.CurrentState.PhysicsUpdate();
}
```

Figura 8.2.5: Fragments de l'*script* Entitat relacionats amb la màquina d'estats

Mitjançant la crida a *LogicUpdate() i PhysicsUpdate()* de l'estat actual, **a cada cicle es delega la lògica del comportament de l'entitat a l'estat en el que es trobi**. Separar aquesta lògica en dos moments diferents del cicle d'execució ens permet tenir més control sobre quan es vol que s'executi cada fragment d'aquesta. Veure la **figura 8.2.6**.

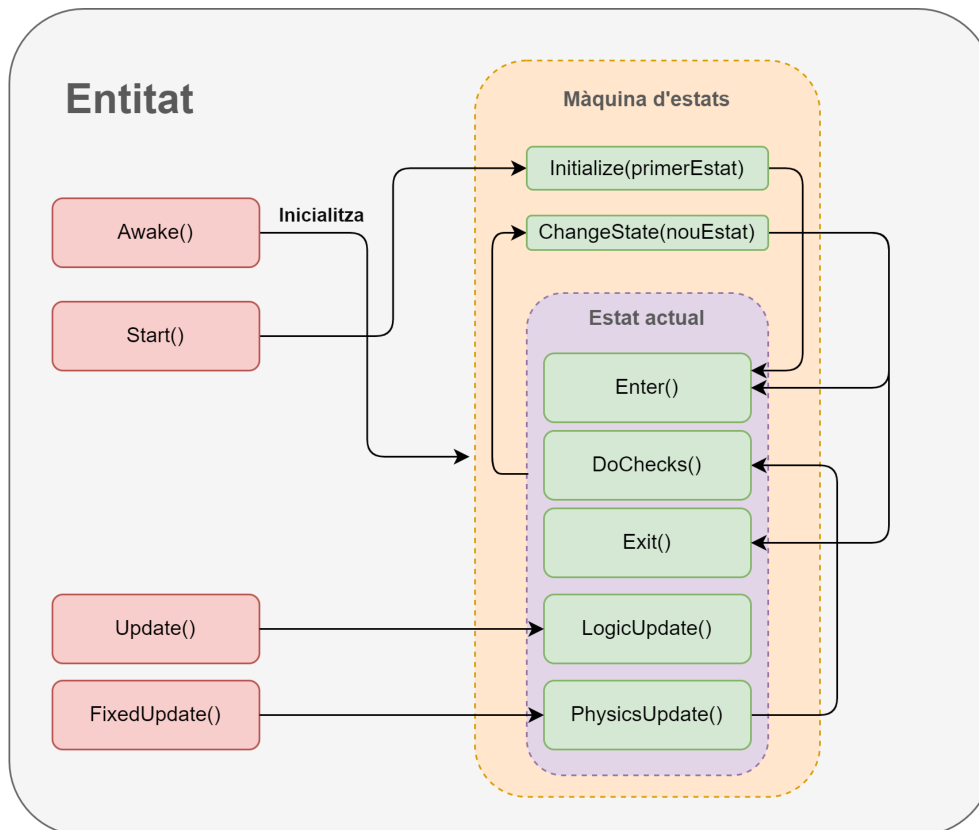


Figura 8.2.6: Estructura de la màquina d'estats i comunicació entre els elements que la formen.

8.5 Shader

Un dels requeriments que hem volgut incorporar al nostre joc és que compti amb una interfície diegètica que doni informació sobre l'estat del personatge principal a l'usuari. La opció que més encertada ens ha semblat ha estat la de modificar l'aparença del mateix personatge segons l'estat en el que es trobi. Per tal de fer això, hem optat per crear un *shader* i aplicar-lo al material que utiltza l'*sprite renderer* del *GameObject* corresponent al personatge (Player). Un material és un objecte encarregat de descriure l'aparença d'una malla o superfície a través d'una textura amb paràmetres i propietats. Aquest objecte sempre conté una referència a un objecte *Shader*, i es pot accedir via codi (script) a les propietats definides per aquest per tal de modificar l'aspecte final de la malla en temps real. Així doncs, el *shader* al qual fa referència un material es tracta d'un programa que s'executa a la GPU, i forma part de la *pipeline* de renderització, sobre la qual es fan els càlculs necessaris per determinar el color dels píxels a la pantalla. Utilitzem aquests elements per treballar sobre la *Universal Render Pipeline* de Unity, la qual està preparada per treballar amb *shaders* en 2D. Veure la **figura 8.3.1**.

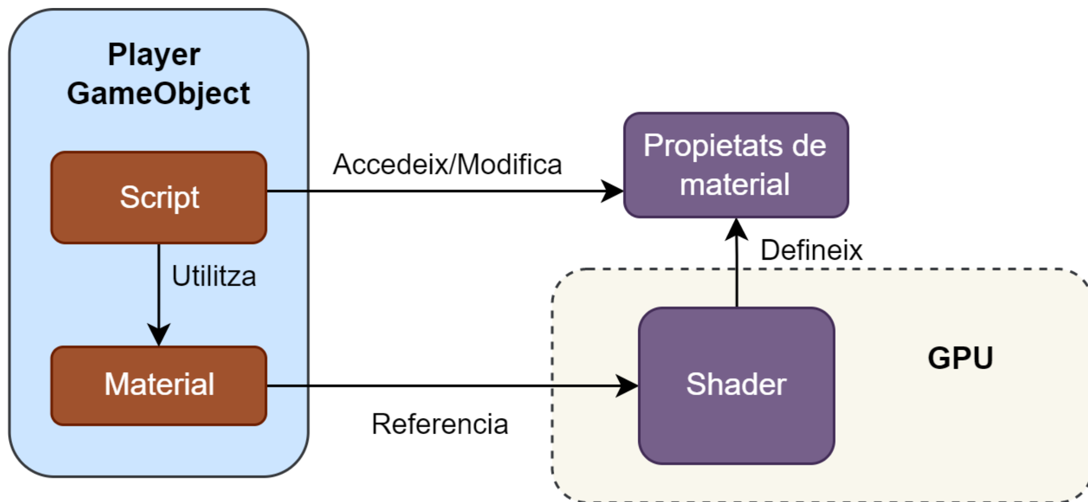


Figura 8.3.1: Diagrama de comunicació entre el material i el *shader*

La implementació dels efectes en l'aparença del personatge principal es pot veure a l'apartat *9.6 Shader*.

8.6 Checkpoints

La gran majoria de jocs en 2D de plataformes tenen un sistema de Checkpoint. Entenem com a Checkpoint un punt de reaparició pel jugador un cop s'ha mort. Són molt útils per no trencar amb la inèrcia del gameplay ja que, si haguéssim de tornar a començar la partida un cop ens haguéssim mort, podria resultar ser molt pesat per a un joc d'aquestes característiques.

Així doncs necessitarem varis objectes que seran aquests Checkpoints. Aquest objecte contindrà el següent:

- **Sprite Renderer**: contindrà l'sprite del Checkpoint
- **Animator**: necessari per animar el Checkpoint un cop hi hem arribat
- **BoxCollider2D**: necessari per a detectar al jugador
- **Script *Checkpoint***: implementa la lògica del Checkpoint

La idea és que quan el jugador s'acosta al Checkpont, aquest s'activa i es guarda la seva posició com a la de l'últim Checkpoint a on s'ha arribat. Llavors, un cop el jugador es mor, es posiciona a aquesta última posició guardada. Per a veure la implementació amb més detall, recomanem anar a la secció 9.7 Checkpoints.

En total, per al nivell de l'escenari, hem creat fins a 8 Checkpoints, col·locats en llocs estratègics.

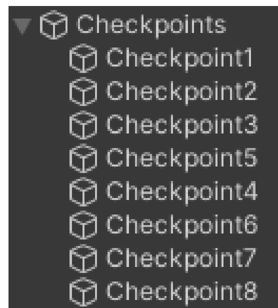


Figura 8.3.2: Objectes dels Checkpoints a la jerarquia del projecte

Per a veure resultats de la seva implementació, anar a la secció 10.9 Checkpoints.

8.7 Lluita amb el Boss

L'últim desafiament del jugador és lluitar contra el Boss. Entenem com a Boss un enemic força més poderós que la resta, amb més vida i amb comportament propi (no similar a la resta d'enemics).

El Boss es trobarà a la part final del nivell, en un escenari preparat per a la batalla amb aquest.

La batalla amb aquest enemic consta de 3 fases:

- **Fase 1**: Pluja de meteorits
- **Fase 2**: Lluita contra súbdits
- **Fase 3**: Lluita amb el Boss

A la fase 1, el jugador haurà d'esquivar meterorits que cauen del cel. Hi haurà intervals de temps on no en caurà cap i on el Boss serà vulnerable. És en aquest moment on el jugador haurà de castigar al Boss.

Quan la vida del Boss hagi baixat fins a la meitat, es passa a la fase 2.

A la fase 2, el jugador haurà de lluitar contra un nombre definit d'enemics que invoca el Boss a través de portals. Aquests enemics són els mateixos que ens hem anat trobant durant la partida. Un cop derrotats tots, passem a la fase 3.

La fase 3 és la més difícil de passar. En aquesta, el Boss invoca 2 enemics al principi. Llavors és ell el que ataca al jugador. Depenent de la distància del jugador, realitzarà uns tipus d'atacs o uns altres. Un cop la vida del Boss hagi arribat a 0, s'acabarà la tercera fase i també la lluita. Veure la **figura 8.3.3**.

Figura 8.3.3 : Diagrama de funcionament general de la lluita amb el Boss

A continuació, per a explicar més concretament aquesta lluita amb el Boss, dividirem l'explicació en diferents apartats.

8.7.1 Escenari de lluita

Per a què la lluita amb el Boss comenci, necessitem quelcom que detecti el jugador i que activi el començament de la lluita. És per això que a l'inici de l'escenari de la lluita hi ha un BoxCollider2D que detectarà la col·lisió amb el jugador i que farà començar la lluita. Aquest objecte que conté el collider l'hem anomenat BattleStartPoint. Després d'activar la lluita es desactiva per evitar tornar-la a activar de nou. Veure el collider a la **figura 8.3.4**.

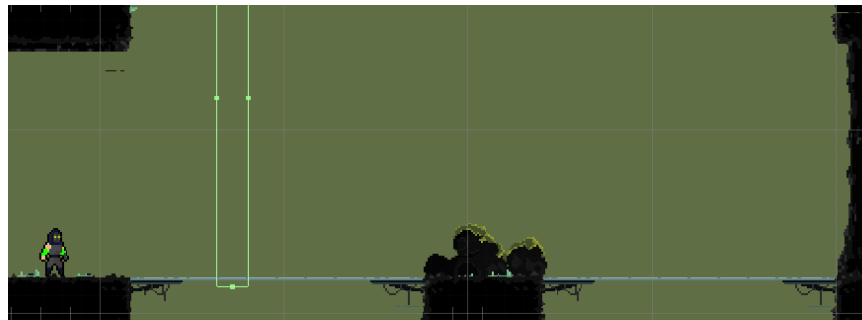


Figura 8.3.4 : BoxCollider2D *BattleStartPoint*

A l'escenari, també hi ha unes plataformes que actuen de comportes. Al iniciar la lluita, aquestes baixen, evitant que el jugador pugui tornar enrere o avançar de la lluita. Està obligat a lluitar. Veure comportes a la **figura 8.3.5**.

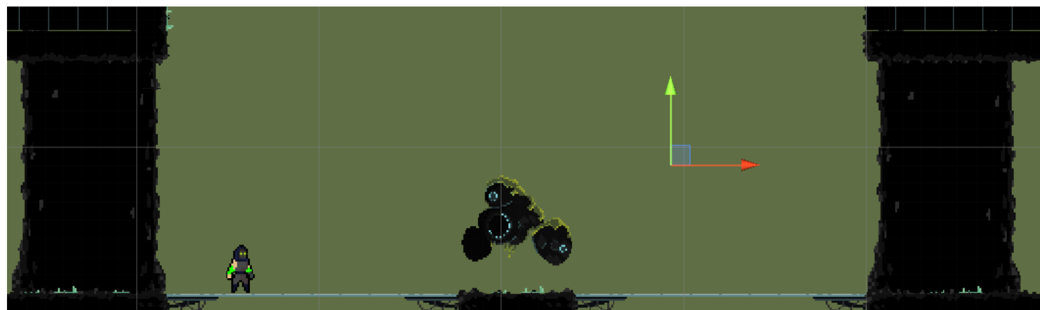


Figura 8.3.5 : Comportes que delimiten l'escenari de lluita

Un cop derrotat el Boss, aquestes pujaran i es ja es podrà avançar.

8.7.2 Boss

Ara toca parlar de l'enemic en si mateix.

- **Disseny de l'enemic**

  Primer vam crear des de zero tots els sprites del Boss (els de Idle, atac, moviment, etc). El resultat ens va quedar prou bé i ens conformavem amb l'aspecte visual. Ara bé, al canviar els sprites dels enemics vam veure que l'aspecte del Boss no lligava gaire amb el d'aquests (i tampoc amb el de l'escenari). És per això que finalment vam decidir buscar per internet sprites nous, i els vam acabar trobant. A la **figura 8.3.6** podem veure sprites del Boss.

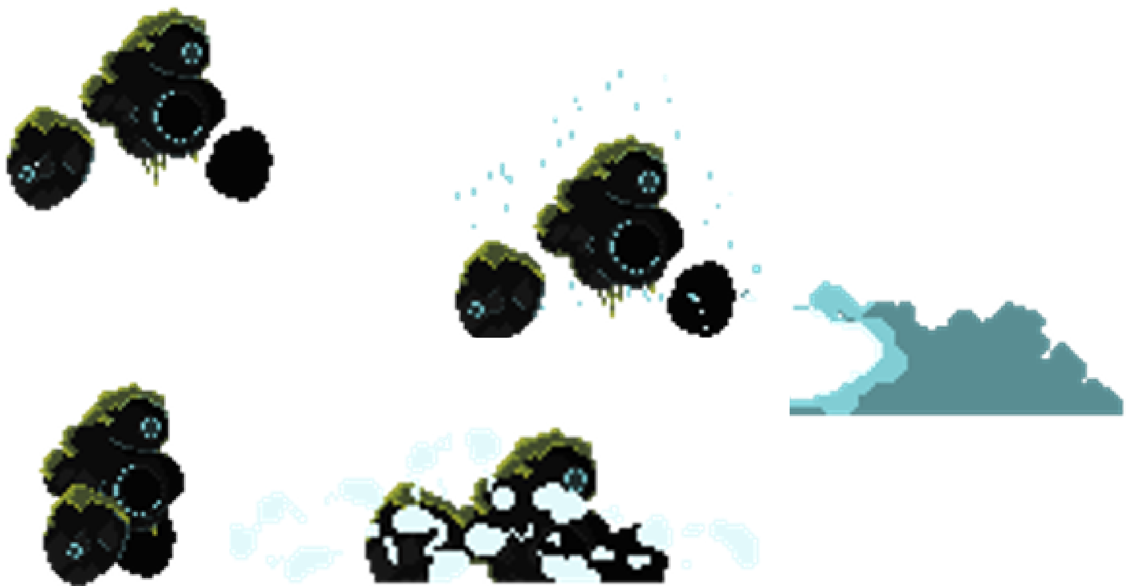


Figura 8.3.6 : Sprites del Boss

Un cop obtinguts els sprites, vam passar a fer les animacions.

El Boss disposa d'11 animacions diferents: Sleep, Wake, Idle, Move, Buff, Immune, MeleeAttack, rangeAttack, SuperAttack, BoomerangBody i Death.

A la **figura 8.3.7** podem veure la relació entre les animacions dins l'Animator.

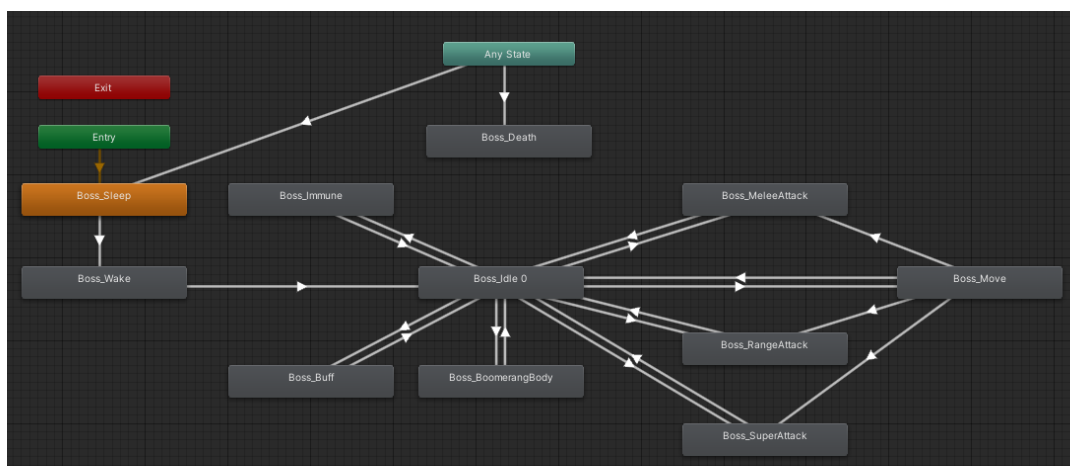


Figura 8.3.7 : Animator del Boss

Per a mostrar la vida del Boss, hem dissenyat una barra de visa personalitzada per a ell. Aquesta conté el nom de l'enemic i és de color vermell. Quan se li baixa vida al Boss, la vida a la barra no baixa de cop, sinó que hi ha un efecte de disminució de vida progressiu. Aquest efecte s'aconegueix tinguent una barra grisa a darrere i la barra vermella a sobre d'aquesta. Quan li fem mal al Boss, primer disminuïm la barra vermella, i després la barra grisa de darrere s'adapta a la mida de la barra vermella amb una velocitat determinada. Veure barra de vida a la **figura 8.3.8**.

Figura 8.3.8 : Barra de vida del Boss

Podem trobar l'script encarregat de la barra de vida del Boss aquí.

Hi ha dos scripts que ens controlen la situació de l'enemic. Per situació entenem saber-ne aspectes lògics i físics seus (quanta vida té, si està tocant el terra, si pot girar per mirar el jugador, etc.). Aquests dos scripts són els següents:

- BossBehaviour
- BossCombatController

8.7.3 Gestió de la lluita per fases

Un cop creat el Boss amb els seus components, necessitem que quan el player entri en contacte amb el collider inicial de l'escenari, hi hagi alguna cosa que comenci a gestionar les fases de la lluita. És per això que hem creat un nou objecte anomenat *BossBattlePhaseManager*.

Aquest objecte contindrà variables comunes per a totes les fases (el boss, posició del jugador, audios, límits, etc.) i també els objectes de les fases i els de les transicions entre aquestes (que s'activaran i desactivaran al canviar de fase).

A continuació podem veure diagrames de funcionament de l'inici de la lluita i de les fases:

- **Inici de la lluita (StartBattle)**

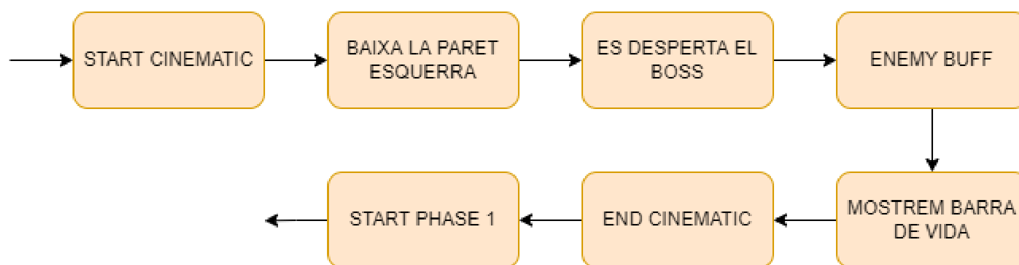


Figura 8.3.9: Diagrama de funcionament de l'inici de la lluita amb el Boss

- **Fase 1 (Phase1)**

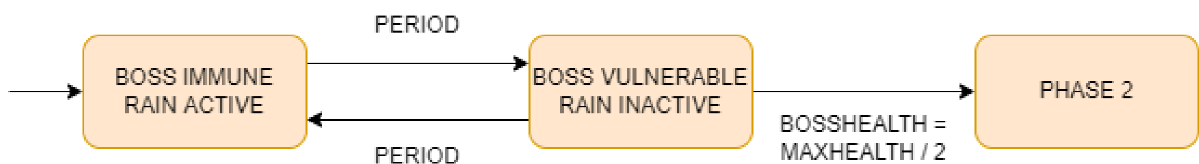


Figura 8.3.10 : Diagrama de funcionament de la fase 1

- **Fase 2 (Phase2)**

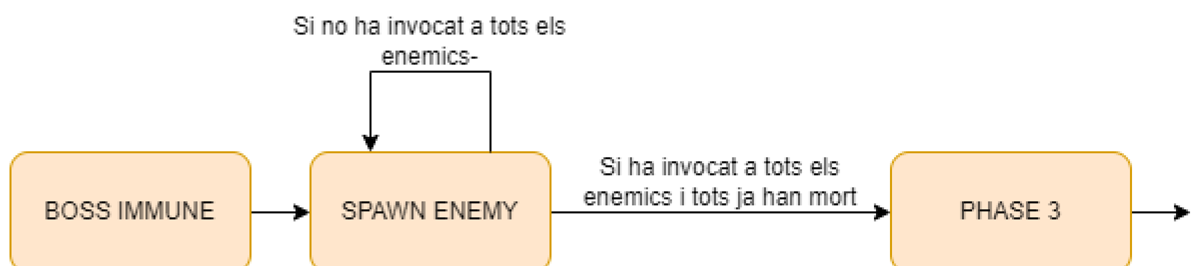


Figura 8.3.11 : Diagrama de funcionament de la fase 2

- **Fase3 (Phase3)**

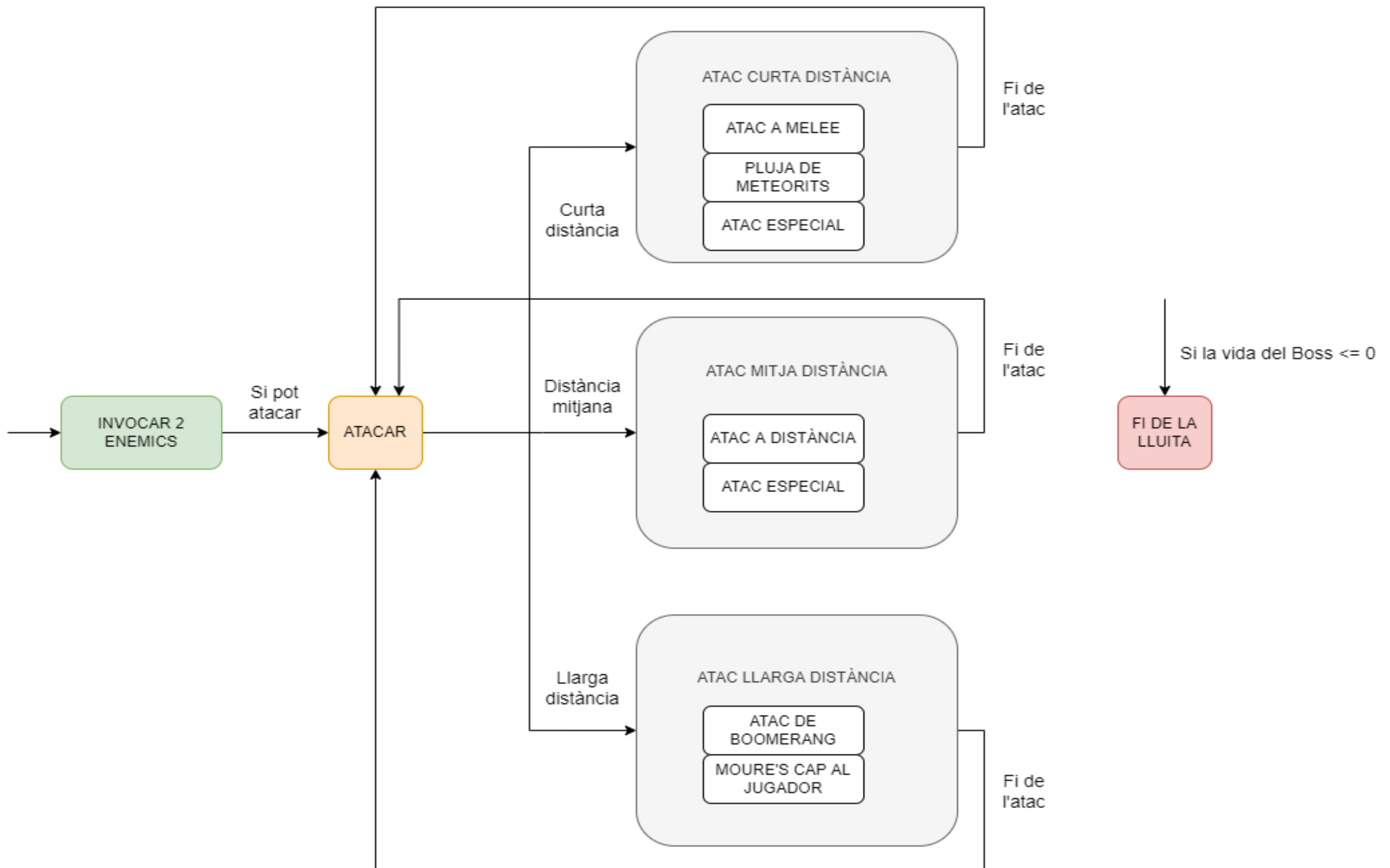


Figura 8.3.12 : Diagrama de funcionament de la fase 3

L'explicació i implementació més detallada de cadascuna de les fases la podem trobar als següents apartats:

- Implementació de BossBattlePhaseManager
- Implementació de l'inici de la lluita
- Implmentació de la fase 1
- Implementació de la fase 2
- Implementació de la fase 3

8.8 Efecte de parallax

El desplaçament de paral·laxi (comunament anomenat paral·lax) és el desplaçament d'una capa de paisatges en un videojoc en dos dimensions. Aquesta tècnica que té com a objectiu donar una impressió de profunditat en jocs 2D. Per aconseguir-ho, les capes del fons es mouen a velocitats diferents respecte la càmera.

Hi ha tres tipus de desplaçament: el vertical, horitzontal i diagonal. Per als nostres objectius i necessitats, nosaltres hem implementat l'horitzontal.

Hi ha videojocs com Shadow of the Beast que fan servir fins a 13 capes amb desplaçaments amb velocitats diferents. El més comú, però, és utilitzar-ne menys (en el nostre cas n'hem utilitzat 5).

Cal tenir en compte també que no podem disposar de capes amb infinita llargada. És per això que caldrà anar movent els sprites de les capes a mesura que anem desplaçant el jugador.

Per generar un efecte en Parallax necessitem:

Variables

- **Llargada de l'sprite de la capa**: Necessària per a saber si cal moure l'sprite de posició (i així que sembli que la capa tingui llargada "infinita").

- **Posició de d'inici de la capa:** Al igual que la llargada de l'sprite, la necessitem per el mateix.

- **Càmera:** Necessitarem la posició de la càmera.

- **Velocitat de desplaçament de la capa**: Anirà de 0 a 1, on si és 0 es mourà a la mateixa velocitat que la càmera i la velocitat serà cada vegada menor com és s'acosta a 1.

L'script encarregat de fer aquest efecte es diu ParallaxMovement, i el trobem en l'apartat d'implementació del paral·lax.

Un cop implementat l'script, ara tocava crear les capes del paral·lax les quals tindrien aquests scripts. Tan i com s'ha dit, nosaltres hem creat 5 capes. Cada capa està formada per 3 sprites. A la **figura 8.4.1** podem veure una representació dels sprites de les capes.

Capa 1

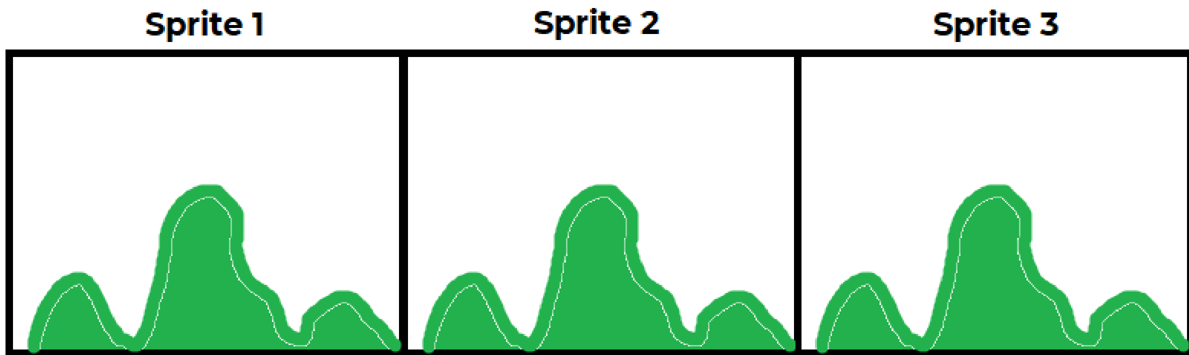


Figura 8.4.1 : Representació dels sprites de les capes

Tot seguit, hem determinat les velocitats de cadascuna de les capes:

- **Capa 1**: 1
- **Capa 2**: 0.75
- **Capa 3**: 0.5
- **Capa 4**: 0.25
- **Capa 5**: 0.15

Un cop determinades les velocitats de cada capa, només falta posar-hi els sprites. És molt complicat (almenys per a les nostres expectatives) crear des de zero 5 imatges de fons amb pixelart. És per això que hem preferit obtenir-les per internet. Tot i això, se'ns presentaven amb uns colors que no voliem ja que buscàvem un tema més fosc. Per aquest motiu, amb l'ajut de l'eina Piskelapp, vam canviar els colors dels sprites per tonalitats de verd (les capes més pròximes més fosques i les més llunyanes més clares).

Veure les capes representades a l'escena de Unity a la **figura 8.4.2**.

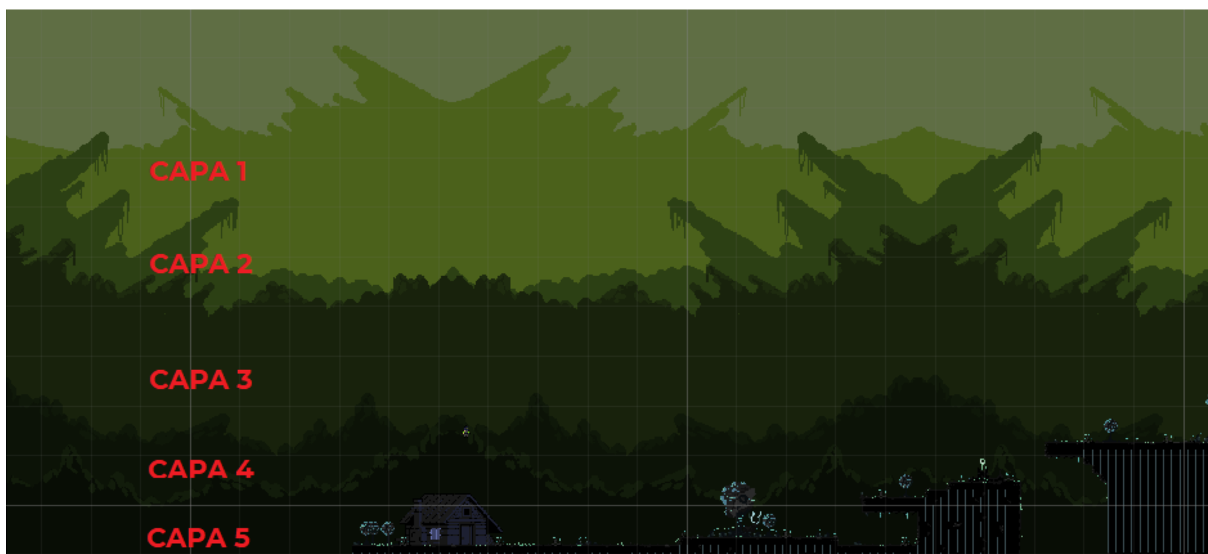


Figura 8.4.2 : Representació de les capes a l'escena de Unity

Podem veure els resultats de la implementació del paral·lax al videojoc a l'apartat de resultats del paral·lax.

8.9 Interfícies d'usuari

Com a interfícies d'usuari, n'hem implementat 4 de diferents:

- Menú Principal
- Menú de pausa
- Menú Game Over
- Diàlegs

Per a saber si estem en pausa, jugant o si hem mort, hem implementat un script anomenat GameController. Des d'aquest, es guardarà l'estat del joc (Game, Pause ó GameOver) i es podrà canviar d'estat, mostrant les interfícies respectives. Des del menú principal, al escollir començar la partida, l'estat en què ens trobem és 'Game' (que vol dir que estem jugant). Des de l'estat Game, podem anar a dos estats diferents: el GameOver (si morim contra el Boss) ó el Pause (si prenem la tecla 'Esc'). Des del menú de pausa (estem a l'estat de Pause), podem canviar a l'estat de Game prenent la tecla 'Esc' ó triant la opció 'RESUME'. Finalment, des de l'estat de GameOver, podem anar a l'estat de Game seleccionant 'RESTART'. Perquè quedi més il·lustrat, a la **figura 8.5.1** es pot veure el funcionament del GameController amb els menús.

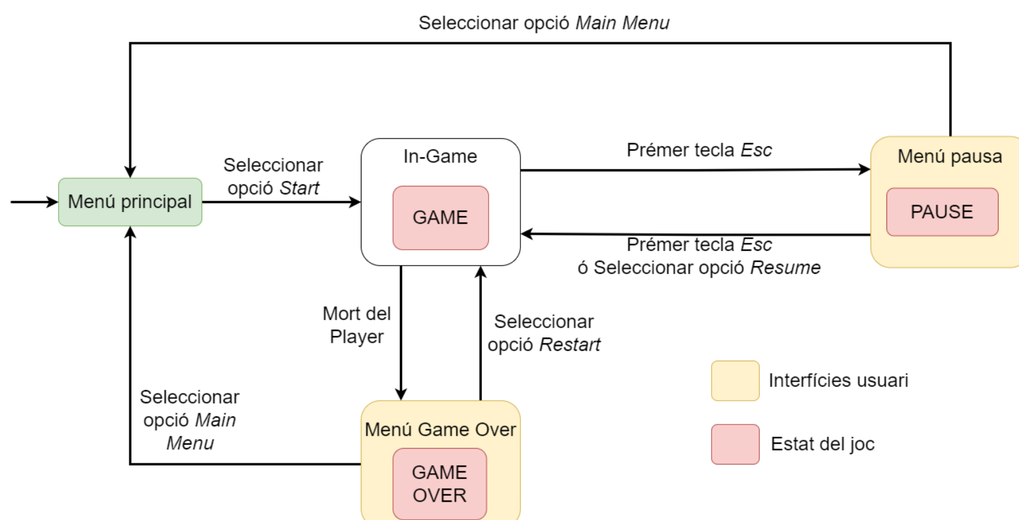


Figura 8.5.1 : Diagrama del funcionament del GameController i interfícies d'usuari

L'implementació del GameController, orientada a les interfícies d'usuari la podem trobar a l'apartat 9.11.1 Menús dins el GameController.

8.9.1 Menú principal

La nostra idea era no carregar massa el menú principal. Amb la petita magnitud del nostre projecte, no hem necessitat afegir-hi masses opcions. Per exemple, hi ha videojocs que poden carregar una partida existent, anar a un cert nivell, ajustar els gràfics, etc. Nosaltres ens hem limitat en les següents accions:

- Començar una partida
- Opcions
    - Volum de la música
    - Volum dels efectes de so
- Sortir de l'aplicació

Un cop assumides les accions que es podran dur a terme, vam fer un croquis de com seria el menú. Veure la **figura 8.5.2**.

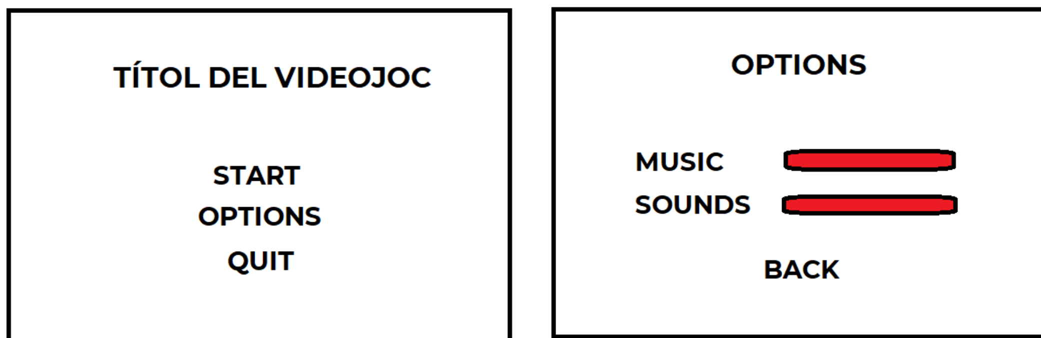


Figura 8.5.2 : Disseny de les interfícies d'usuari del menú principal

Un cop fets els dissenys sobre el paper, ara necessitavem crear els diferents elements d'aquests dissenys. Per al menú del títol, necessitem 5 GameObjects. Aquests objectes tindran un SpriteRenderer (amb l'sprite respectiu). Aquests seran el fons, el títol del videojoc, l'Start, les opcions i el Quit. També volem que al canviar d'opció i seleccionar-ne una, soni un so. És per això que necessitarem dos objctes més (Option, pel so d'escollir una opció; Selection, per seleccionar una opció). Veure la **figura 8.5.3**.

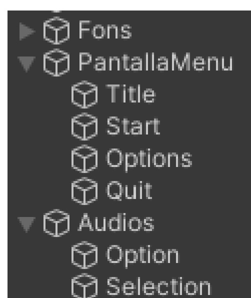


Figura 8.5.3 : Objectes del Menú Principal a la jerarquia del Unity

És força semblant per el menú d'opcions. En aquest cas necessitem un objecte pel títol, un per la música, un pel so, un per l'àudio de pujar o baixar el volum i seleccionar opcions i un per tornar enrere. A més, les barres de música i volum no són una barra intacta. Més aviat són petites barres verticals que canvien de color en funció del volum actual posat. Si no hi ha volum, es posa una creu en vermell. Veure la **figura 8.5.4**.

Figura 8.5.4 : Disseny de les barres de volum del menú d'opcions

És per aquest motiu que també necessitem un objecte per cada barra vertical un objecte per cada creu. Veure la **figura 8.5.5**.

Figura 8.5.5 : Barres de volum del menú d'opcions a la jerarquía del Unity

Un cop dissenyat el menú principal, cal aplicar-ne el seu funcionament. Aquesta aplicació s'explica més detalladament a l'apartat 9.11.2 Menú principal. Els resultats de la implementació del menú principal els podem trobar a l'apartat 10.5.1 Menú principal.

8.9.2 Menú de pausa

La gran majoria els videojocs solen tenir un menú on es pot deixar de banda el Gameplay per fer un descans o simplement per a deixar de jugar. Aquest se l'anomena el Menú de Pausa. En aquest es poden realitzar vàries accions, desde tornar al menú principal fins a ajustar configuracions visuals. El que fan la gran part dels menús de pausa és precisament pausar el gameplay, aturar el temps del joc de manera que durant la pausa no es pot moure el jugador i res de l'escena interacciona amb ell ni amb res. Ara bé, també hi ha jocs com els Dark Souls on el menú de pausa no atura el joc, sinó que el jugador s'ha d'afanyar en realitzar el que vulgui fer a aquest menú de tal manera que no es mori per cap atac enemic.

Nosaltres ens hem decantat més pel segon estil (el menú de pausa no atura el gameplay). Ho hem decidit així perquè al ser un videojoc on la música té un paper important, quedaria estrany que aquesta es pausés i per tant que els objectes que interactuen amb ella també. A més, això podria portar descompensacions entre la música i aquests objectes de manera que la sincronització acabaria fallant. També creiem que si es pot jugar mentre durant el menú de pausa, li dóna un to més orgànic a la jugabilitat i no talla amb el progrés del jugador.

Havent aclarit la decisió del tipus de menú de pausa, anem a veure com l'hem dissenyat. El disseny del menú de pausa és molt similar al del menú principal. En aquest cas però, no tenim títol i està desplaçat a baix a l'esquerra (per no molestar amb el gameplay). A la **figura 8.5.6** en podem veure un primer disseny.

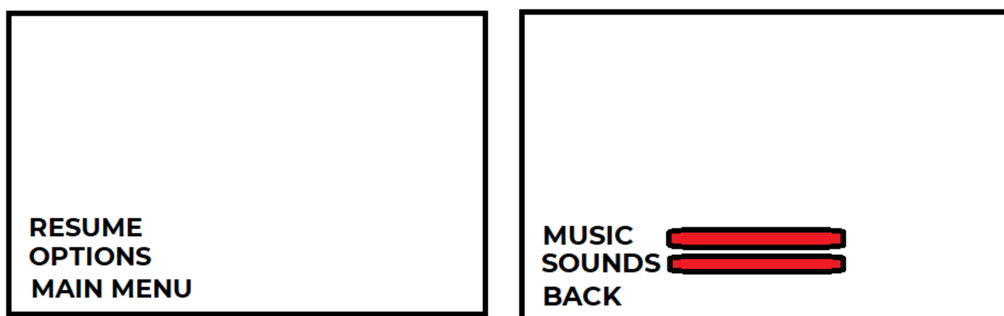


Figura 8.5.6 : Disseny de les interfícies del menú de pausa
Veiem que pel menú de pausa principal (de les dues imatges, la de l'esquerra) necessitarem 3 GameObjects(un per Resume, un per Options i un per Main Menu).

Pel menú d'opcions de pausa, en canvi, necessitarem els 3 GameObjects pels títols (Music, Sounds i Back), però també necessitarem les barres dels volums i les creus vermelles per indicar qu eel volum equival a 0 (de la mateixa manera que al menú de pausa del menú principal). Veure la **figura 8.5.7**.

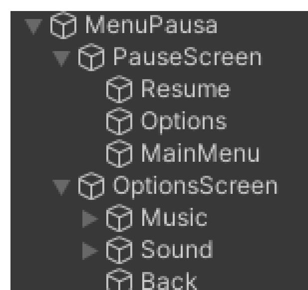


Figura 8.5.7 : Objectes del menú de pausa a la jerarquia del projecte

La lògica dels menús de pausa s'explica a l'apartat 9.11.3 Menú de pausa.

Els resultats de la implementació dels menús de pausa es poden veure a l'apartat 10.5.2 Menú de Pausa.

8.9.3 Menú de Game Over

Pel menú de Game Over (ó menú de fi de la partida) hem decidit no complicar-lo massa i implementar-hi les opcions indispensables per aquest. Aquestes opcions són les de tornar a començar la partida i la de anar al menú principal.

En aquest cas i contrari al menú de pausa, el jugador no es podrà moure ni interactuar amb l'entorn mentre no es decideixi tornar a començar la partida.

- **Disseny**

  A la figura 8.5.8 podem veure un primer disseny en brut de la interfície del menú de Game Over.

Figura 8.5.8 : Disseny de la interfície del menú de Game Over

D'aquest disseny, en destaquem 4 GameObjects necessaris:
- Títol Game Over
- Títol Restart
- Títol Main Menu
- Un panell blanc amb baixa opacitat (per a donar-li un aspecte diferent, que es noti que s'ha acabat la partida, destacar més el títol de Game Over i les opcions).

Veure la **figura 8.5.9**.

Figura 8.5.9 : Objectes del menú de Game Over a la jerarquia del projecte

La implementació de la lògica del menú de Game Over la trobem a l'apartat 9.11.4 Menú de Game Over, i els resultats a l'apartat 10.5.3 Menú de Game Over.

8.9.4 Diàlegs

Els menús no són les úniques interfícies d'usuari presents al videojoc. També disposem de diàlegs. Entenem com a diàlegs panells informatius que interaccionen amb el jugador per a informar-lo de certa situació o de guiar-lo durant la partida.

Per a la nostra situació hem volgut afegir diàlegs per a la part de tutorial del joc. A l'escena de gameplay, la primera meitat del joc té el paper d'ensenyar al jugador quines són les mecàniques del Player i com es comporta l'escenari. És per això que els diàlegs explicatius encaixen força bé amb aquesta part.

- **Disseny**

  Per al disseny dels diàlegs, hem decidit que aparegui a la part inferior de la pantalla de joc (i d'aquesta manera no ocupem l'espai de joc principal).

  A la **figura 8.6.1** es pot veure un primer disseny de la interfície dels diàlegs.

Figura 8.6.1: Disseny de la interfície dels diàlegs

Així doncs, necessitem varis objectes que conformaran el diàleg. Aquests són un per a una imatge, un pel text del diàleg, un per un panell de color clar i un per un panell una mica més fosc. Veure la **figura 8.6.2**.

Figura 8.6.2: Objectes del diàleg a la jerarquia del projecte

Finalment, necessitarem els objectes que activaran aquests diàlegs amb el text i imatges que toquin. Els hem anomenat *DialogueZones*. Aquests contenen un

BoxCollider2D per a detectar el Player i un cop detectat activar el Diàleg amb cert text i certa imatge. Veure la **figura 8.6.3**.

Figura 8.6.3: Objectes DialogueZones a la jerarquia del projecte

Per a il·lustrar millor el funcionament dels diàlegs, hem realitzat un diagrama que es mostra a la **figura 8.6.4**.

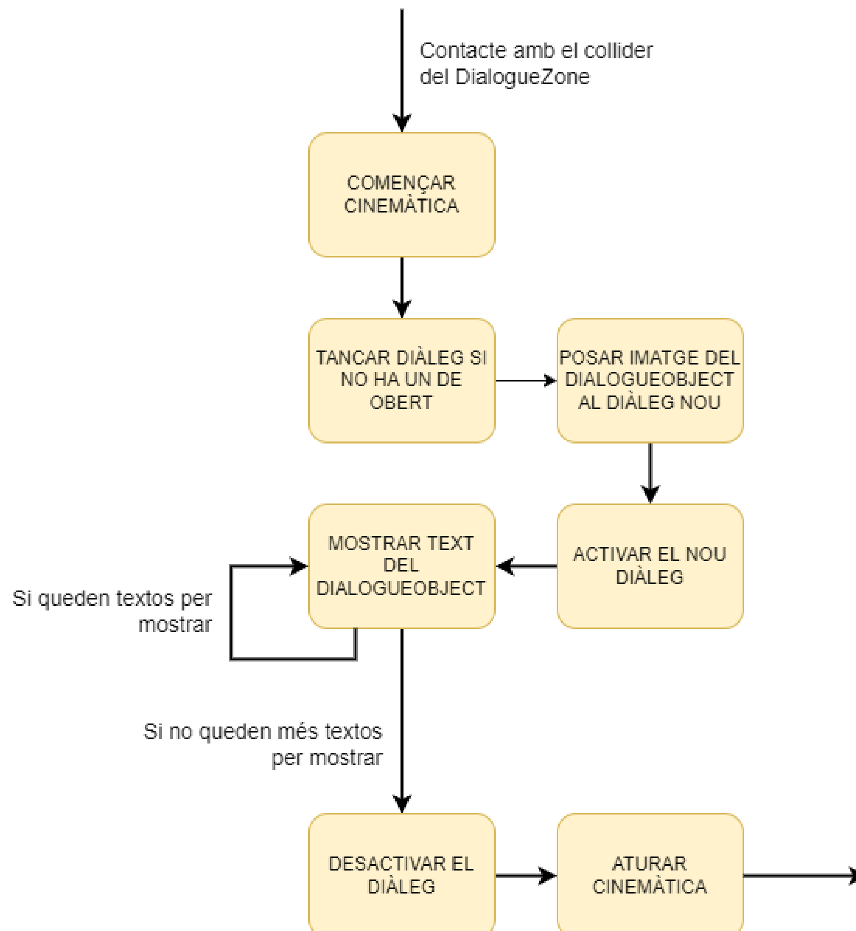


Figura 8.6.4: Diagrama del funcionament dels diàlegs()

La lògica de la implementació dels diàlegs s'explica a l'apartat 9.11.5 Diàlegs. Els resultats es mostren a l'apartat 10.5.4 Diàlegs.

8.10 Sincronització de la música amb el gameplay

Un dels requisits del nostre joc és que la música tindria un gran paper al Gameplay. No només com a música de fons sinó que certs elements del videojoc (Gameobjects) es sincronitzarien amb aquesta per a donar-li un to peculiar i diferent al gameplay. En un inici el nostre objectiu era fer un estudi de freqüències amb Fourier. Vam investigar força sobre el tema però tota la documentació i exemples que vam trobar de projectes semblants era anticuada, no s'adaptava als requeriments i versió actual del Unity (Unity 2021).

És per això que vam decidir seguir un camí alternatiu, on inevitablement no tota la sincronització es fa automàtica, sinó que hi ha una part manual (8.10.2 Creació de la música amb marques).

8.10.1 Sincronització de l'FMOD Studio amb Unity

Per a dur a terme la sincronització, el primer que vam haver de fer és descarregar-nos un programa extern anomenat FMOD. A la figura 8.7.1 es pot veure la interfície del programa.

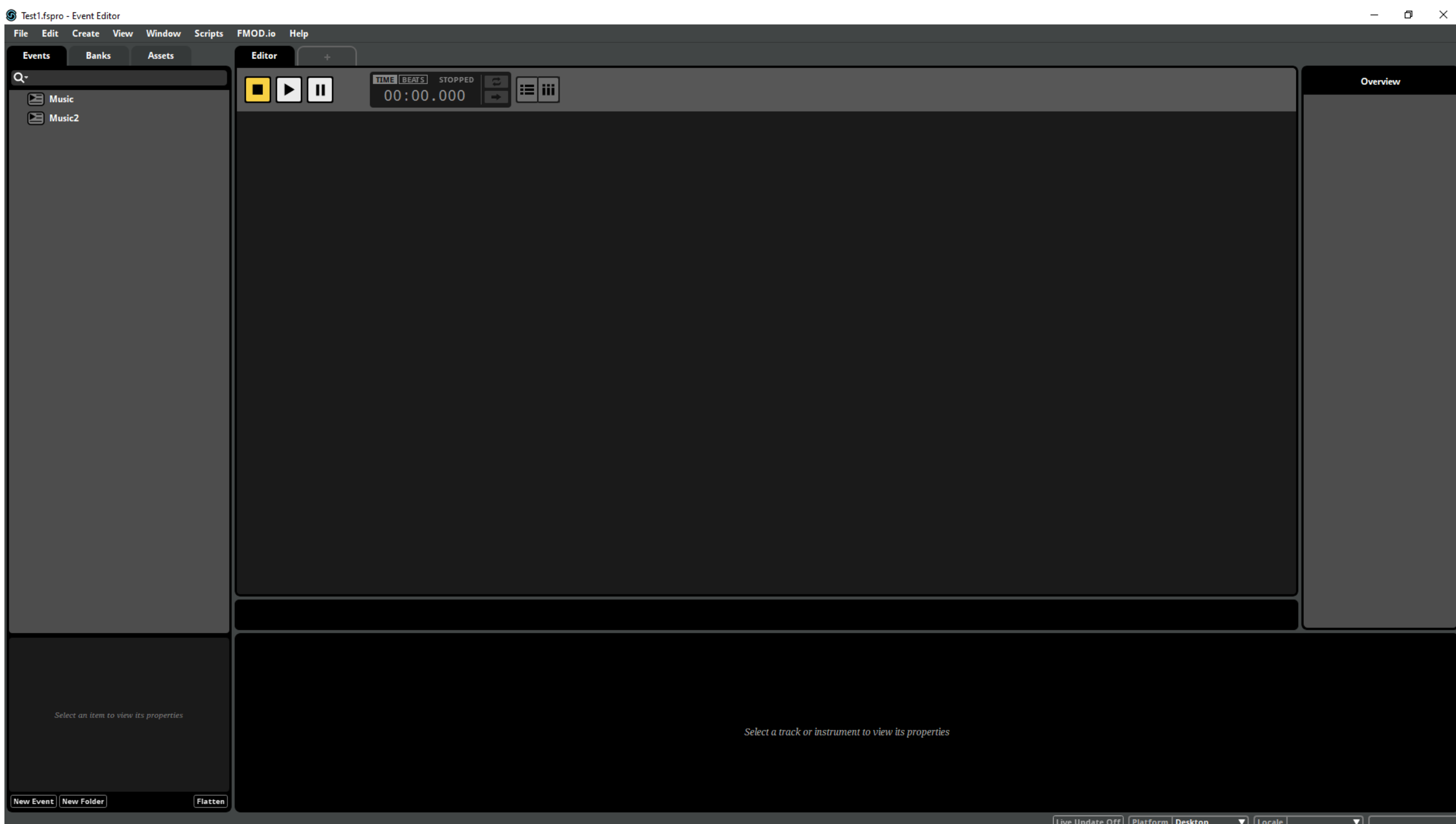


Figura 8.7.1 : Interfície de l'FMOD Studio

Un cop descarregat el programa, el següent pas és integrar-lo amb el projecte de Unity.

Per a fer-ho, el primer pas és descarregar-se un paquet des de l'Asset Store de Unity. Llavors hem de crear un directori dins de la carpeta Assets del nostre projecte. I dins d'aquesta un altre directori. En aquest i guardarem els fitxers (anomenats banks) generats al FMOD. Veure **figura 8.7.2**.

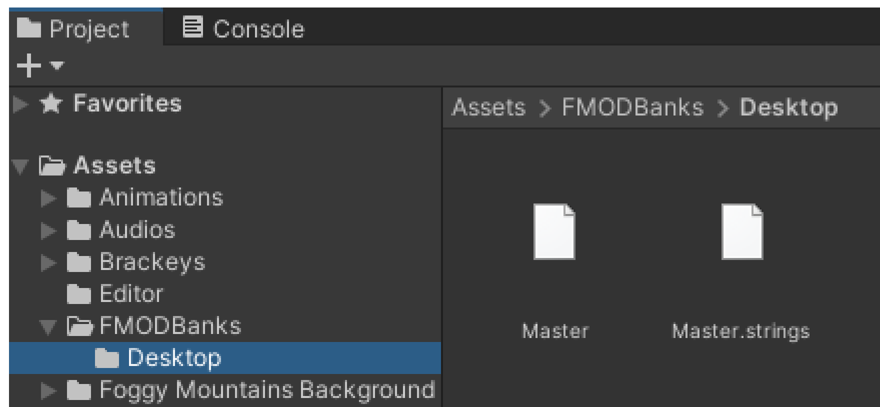


Figura 8.7.2 : Carpetes FMODBanks i Desktop creades a Unity

Un cop creats els directoris, tornem al FMOD i dins a Edit->Preferences->Build, seleccionem com a directori de sortida la nova carpeta creada. Veure **figura 8.7.3**.

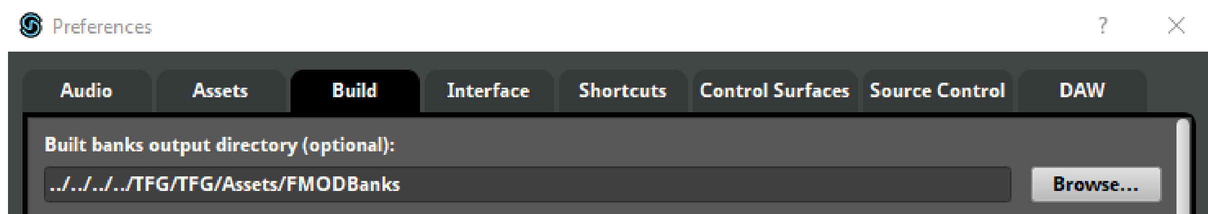


Figura 8.7.3 : Pestanya Build de l'FMOD Studio (escollim carpeta on guardar els banks)

Ara hem connectat l'FMOD amb el Unity, però no el Unity amb l'FMOD. Així doncs, dins el Unity anem a la pestanya FMOD->Edit Settings. Allà, escollim com a Build Path el mateix directori. Veure **figura 8.7.4**.

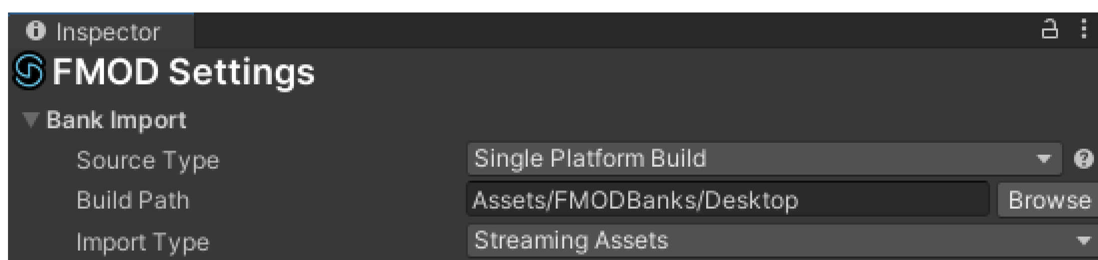


Figura 8.7.4 : Settings de l'FMOD dins del Unity

8.10.2 Creació de la música amb marques

Tal i com s'ha explicat anteriorment, FMOD permet posar marques a la línia temporal a la vegada que posar seccions de bucle. Així doncs, els passos que hem seguit a l'hora d'acabar la música i deixar-la llesta per a la sincronització:

1. Afegir la pista de música en format .wav exportada del BoscaCeoil
2. Afegir una secció en bucle (ja que no volem que s'acabi la música)
3. Afegir les marques en els punts necessaris

A la **figura 8.7.5** es pot veure el resultat després d'haver complert els 3 passos anteriors.

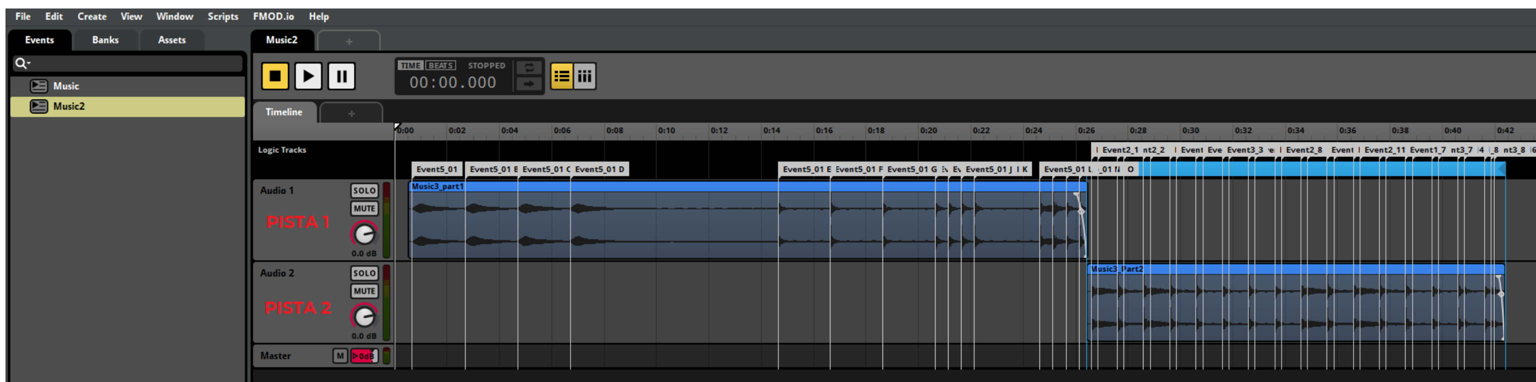


Figura 8.7.5 : Carpetes FMODBanks i Desktop creades a Unity

Com es pot veure, hem dividit la cançó en dues pistes d'audio. La primera actua com a una introducció i la segona està agrupada en una secció en bucle (és la que es repetirà indefinidament). Un cop arribats a aquest punt, compilem el projecte (Ctrl +F7). D'aquesta manera, tenim aquest bank actualitzat al projecte de Unity (ja que hem fet la integració).

8.10.3 Creació de l'Audio Manager

Hem creat la música amb marques, hem integrat el Unity amb l'FMOD, però no tenim cap manera de fer sonar aquesta música. És per això que necessitem un GameObject que se n'encarregui. Aquest objecte disposarà de 3 scripts:

- FMOD Studio Bank Loader
- VCA
- Music Manager

La implementació d'aquests scripts es troba a l'apartat 9.13.1 Creació de l'Audio Manager.

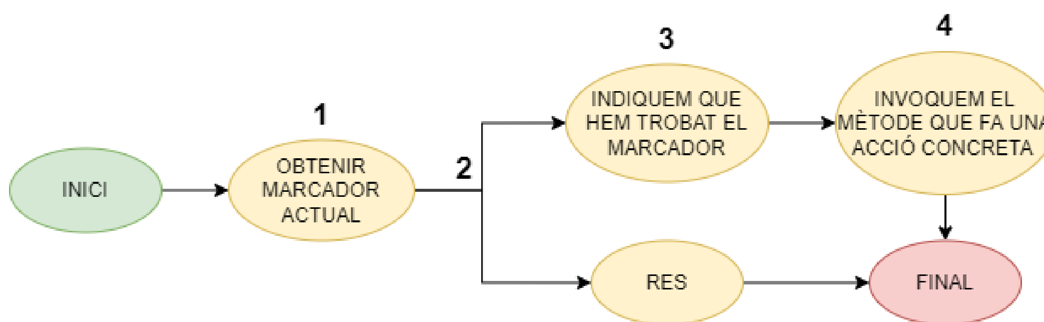
8.10.4 Creació d'objectes que es sincronitzen amb la música

Ara que ja tenim l'AudioManager creat, només queda crear els objectes que es sincronitzaran amb els marcadors de la música.

N'hem creat 5 tipus diferents:

- Plataformes que apareixen i desapareixen
- Plataformes de salt
- Plataformes que es desplacen
- Trampes làser
- Arbres

Cadascun d'aquests objectes té un script força semblant que és l'encarregat de detectar si ens hem trobat un cert marcador, i si és així realitzar una acció (invocar un mètode). Els scripts de cada tipus d'objecte els trobem a l'apartat 9.13.2 Creació d'objectes que es sincronitzen amb la música. La detecció del marcador la trobem al mètode Update(). El seu flux de treball queda mostrat a la **figura 8.7.6**.



1: string currentMarker = MusicManager.instance.timelineInfo.lastMarker;

2: if (currentMarker.StartsWith(stringToWaitFor) && !currentMarker.Equals(markerDone))

3: markerDone = currentMarker;

4: MusicAction();

Figura 8.7.6: Diagrama de flux del mètode Update()

9.
Implementació i proves

9.1 Recerca d'elements gràfics

En aquest apartat s'exposen els diferents elements gràfics que s'han utilitzat per representar tots els components de l'escena.

9.1.1 Personatge principal

Per representar el personatge principal hem optat per elegir-ne un creat per ***Bardent***, el qual disposa de totes les animacions necessàries per representar els estats de la màquina d'estats. Aquest té un aspecte de ninja, dibuixat amb un estil simple en pixel art. Veure la **figura 9.1.1**.

Figura 9.1.1: Personatge principal, creat per **Bardent**

9.1.2 Enemics

L'aspecte dels enemics inicia una tendència estètica que trobem en varis dels altres elements gràfics de l'escena. Aquests presenten un aspecte fosc, amb el negre com a color predominant i amb detalls d'un blau elèctric. Veure la **figura 9.1.2**.

Figura 9.1.2: Enemic, creat per **Penusbmic**

9.1.3 Final Boss

El *Boss* final segueix la mateixa estètica dels enemics, amb un caràcter més amenaçador gràcies a la seva mida. Veure la **figura 9.1.3**.

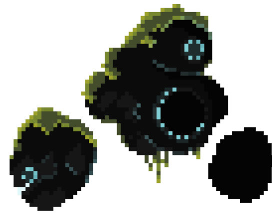


Figura 9.1.3: Boss, creat per **Penusbmic**

9.1.4 Terreny

Per tal de construir el terreny del nivell, hem utilitzat l'eina *Tilemap* de Unity, la qual utilitza un *spritesheet* per crear *tiles* individuals amb les quals es pot anar pintant l'escena quadrat a quadrat. Gràcies a aquesta eina, podem separar els tipus d'elements del terreny en diferents capes, i pintar cada capa separadament de l'altra. Veure la **figura 9.1.4**.

L'aspecte del terreny és d'un indret més aviat estrany, en el qual els elements de la natura com els arbres s'han vist deteriorats, presentant anomalies de color blau versós que aporten els detalls més vistosos. Tot i que el terreny original tenia uns colors més verdosos i clars, l'hem modificat mitjançant el Piskelapp per tal de que quedi un terreny negre, accentuant el blau en els detalls per tal que concordés amb l'aspecte dels enemics.

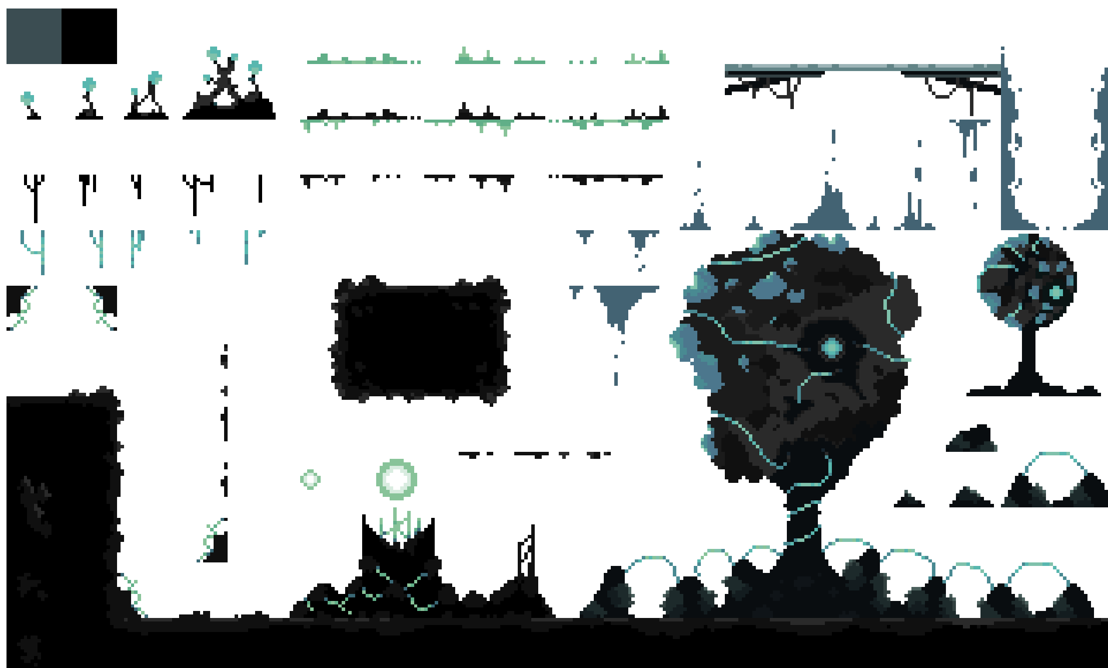


Figura 9.1.4: Tileset del terreny, creat per **Penusbmic**

9.1.5 Background

Pel *background* hem optat per aplicar un color distintiu respecta la resta de colors que tenim aplicats a l'escena. Finalment hem elegit una escala monocromàtica partint d'un verd oliva. Veure la **figura 9.1.5**.

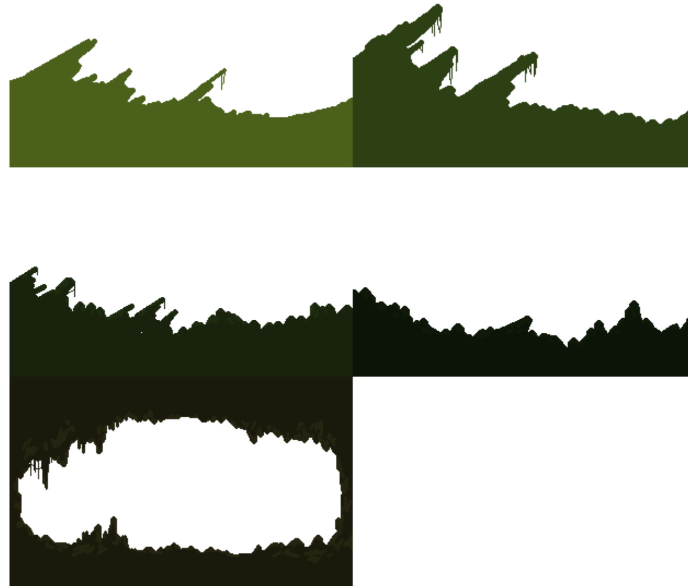


Figura 9.1.5: Capes del background, creat per **Penusbmic**

9.1.6 Altres elements de l'escena

Claus

Per representar les claus, hem escollit un sprite simple d'un color groc per tal que contrasti amb els elements foscos del fons. Al ser un element amb el que el jugador ha de interactuar, és important que es distingeixi bé de la resta d'elements. Veure la **figura 9.1.6**.

Figura 9.1.6: Clau

Flames

Les flames que el personatge ha de recollir per poder fer dash estàn representades amb sprites creats per nosaltres. Es tracten de flames de color blau, del mateix color

que l’efecte de realçat de contorn que es dibuixa al voltant del jugador quan recull una flama. Veure la **figura 9.1.7**.

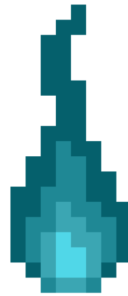


Figura 9.1.7: Flame

Animals

Els animals que es troben a l’escena són elements purament decoratius, per afegir més elements dinàmics que omplin el nivell de vida. Veure la **figura 9.1.8**

Figura 9.1.8: Animals, creat per **Penusbmic**

Plataforma de salt

La plataforma de salt que interacciona amb la música està representada amb sprites creats per nosaltres, seguint amb tons de negre i blau clar pels detalls. Veure la **figura 9.1.9**.

Figura 9.1.9: Plataforma de salt

Trampes làser
Les trampes làser que s'activen amb la música. Sprites creats per nosaltres.

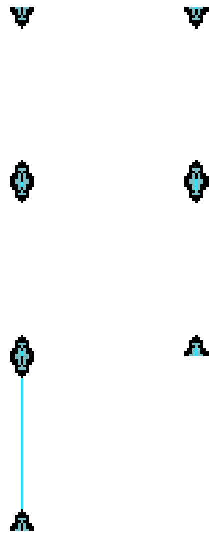


Figura 9.1.10: Trampes làser

9.2 Interfícies

- **IDamageable**: Només defineix el mètode ***Damage(float)****, el qual rep el mal que ha de patir l'entitat que implementi aquesta interfície*. La utilitzen tots els elements que puguin rebre mal.

- **IKnockbackable:** Defineix el mètode ***Knockback(Vector2, float, int)***, el qual rep un angle, una força i una direcció per tal que l'script que implementi la interfície pugui crear un efecte de 'empenta'. La utilitzen tots els elements que puguin ser empesos.

  Aquestes dues interfícies són utilitzades pel Player, els enemics, i per la classe **DamagingObject**. Aquesta és una classe de la qual poden heredar tots els objectes que volem que siguin capaços de fer mal, com el meteorit de la batalla amb el *Boss* o les fletxes dels enemics arquers. El funcionament és simple: al detectar un colisionador, es busca si l'objecte amb el que s'ha xocat implementa alguna de les dues interfícies i, de fer-ho, crida a la funció *Damage() i/o Knockback()*.

  Veure la **figura 9.2.1**.

```
protected void OnTriggerEnter2D(Collider2D collision) {
    IDamageable damageable = collision.GetComponent<IDamageable>();
    IKnockbackable knockbackable = collision.GetComponent<IKnockbackable>();

    if (damageable != null){
        damageable.Damage(damagingObjectData.damage);
    }

    if (knockbackable != null){
        knockbackable.Knockback(Vector2.one,
damagingObjectData.knockbackStrength, collision.transform.position.x >
transform.position.x ? 1 : -1);
    }

    if(destroyAfterDamaging) Destroy(gameObject);
}
```

Figura 9.2.1: Funció OnTriggerEnter2D d'un Damageable Object

Cada **DamagingObject** utilitza un ScriptableObject del qual obté els valors dels paràmetres de l'objecte.

- **ICollector:** La utilitzen tots els objectes que necessitin poder obtenir, guardar i perdre col·leccionables. Defineix els mètodes:
  - ***Collect(int)***: Recollir un col·leccionable.
  - ***Retrieve(int):*** Perdre un col·leccionable.
  - ***Has(int)***: Comprovar si té un col·leccionable.

  Els tres mètodes reben per paràmetre un enter corresponent a l'índex del col·leccionable.

9.3 Core

L'Script del Core serveix d'intermediari entre els Core Components i els estats de l'entitat. Conté uns llista on guarda tots els CoreComponents que es detectin dins la jerarquia de GameObjects. A continuació s'expliquen els mètodes:

- **AddComponent(CoreComponent)**: Afegeix un Core Component a la llista.
- **LogicUpdate()**: Executa el *LogicUpdate()* de tots els Core Components de la llista.
- **GetCoreComponent()**: La utilitzen els estats per poder tenir referència als Core Components als que volen accedir.
- **EntityDead()**: Envia un missatge *"Dead"* cap a tots els GameObjects que siguin pares d'ell mateix. Aquesta crida la rep l'script de l'entitat per saber que el personatge ha mort.

- **Respawn():** Envia un missatge *“Respawn”* cap a tots els GameObjects que siguin pares d’ell mateix. Aquesta crida la rep l’script de l’entitat per saber que el personatge ha de reaparèixer.

9.4 Core Components

En aquest apartat s’explica la implementació dels diferents Core Components.

9.4.1 Appearance

Conté una referència al material que utilitza l’*SpriteRenderer* de l’entitat. L’objectiu d’aquest component és gestionar els canvis en l’aparença d’aquesta modificant les propietats del *shader*. Els mètodes que implementa són:

- **Awake():** Obté el component Animator del GameObject de l’entitat i estableix el paràmetre *isHurt* del *shader* a ‘fals’.
- **Start():** Obté la salut inicial del personatge a través del component *Stats.*
- **Damaged(float):** Inicia la co-rutina de l’efecte de ‘hit’ del *shader.*
- **SetDamagedAppearance()**: Co-rutina de l’efecte de ‘hit’ del *shader.*

  D’aquest component hereda la classe **PlayerAppearance**, la qual tracta les característiques específiques de l’aparença i el *shader* del nostre personatge. El seu funcionament està explicat a l’apartat 9.6 Shader.

9.4.2 CollisionSenses

Encarregat de fer les comprovacions necessàries per detectar diferents elements de l’entorn. Utilitza els següents Transforms com a posicions a partir de les quals fer aquestes comprovacions:

- **GroundCheck:** Per detectar el terra.
- **WallCheck:** Per detectar les parets.
- **LedgeCheckHorizontal**: Per detectar les cantonades per tal de pujar-s’hi. (L’utilitza el Player).
- **LedgeCheckVertical**: Per detectar les cantonades al final de plataformes, per tal de no caure. (L’utilitzen els enemics)

  A part dels Transforms, té referència a la capa física a la qual es troben el terra i les parets per tal que les comprovacions puguin detectar-lo. També se li passen dos paràmetres *groundCheckRadius* i *wallCheckDistance*.

  Els mètodes que implementa corresponen a les comprovacions d’entorn. Veure la **figura 9.3.1.**

```
//Comprovació del terra
public bool Ground{
    get => Physics2D.OverlapCircle(GroundCheck.position,
groundCheckRadius, whatIsGround);
}

//Comprovació de paret a davant
public bool WallFront{
    get => Physics2D.Raycast(WallCheck.position, Vector2.right *
Movement.FacingDirection, wallCheckDistance, whatIsGround);
}

//Comprovació de cantonada pel Player
public bool LedgeHorizontal{
    get => Physics2D.Raycast(LedgeCheckHorizontal.position,
Vector2.right * Movement.FacingDirection, wallCheckDistance,
whatIsGround);
}

//Comprovació de cantonada pels enemics
public bool LedgeVertical{
    get => Physics2D.Raycast(LedgeCheckVertical.position, Vector2.down,
wallCheckDistance, whatIsGround);
}

//Comprovació de paret a darrera
public bool WallBack{
    get => Physics2D.Raycast(WallCheck.position, Vector2.right *
-Movement.FacingDirection, wallCheckDistance, whatIsGround);
}
```

Figura 9.3.1: Mètodes de CollisionSenses

9.4.3 Combat

S'encarrega de gestionar l'interacció entre l'entitat i altres elements de l'escena a partir de les deteccions efectuades per un colisionador. Implementa les interfícies *IDamageable, IKnockbackable* i *ICollector.* Els mètodes són:

- **LogicUpdate()**: Crida a *CheckKnockback()*.
- **Damage(float)**:
    - Crida a *DecreaseHealth()* del component *Stats.*
    - Crida a *Damaged()* del component *Appearance*.
    - Instancia unes partícules d'impacte en la posició del personatge.
- **Knockback(Vector2, float, int):**
    - Aplica velocitat al personatge a través del component *Movement*, amb l'angle, la força i la direcció que rep com a argument.

    - Impedeix que el jugador pugui moure el personatge durant l'instant que està en 'knockback' a través del component *Movement.*
- **CheckKnockback():** S'utilitza per comprovar si el personatge està en 'knockback' i, quan deixa d'estar-ho, tornar a permetre que el jugador pugui moure el personatge.
- **Collect(int)**: Afegeix a l'*Inventory* el col·leccionable amb l'índex que es passa per paràmetre.
- **Has(int):** Retorna 'cert' si el component *Inventory* té el col·leccionable amb l'índex que es passa per paràmetre.
- **Retrieve(int)**: Treu del component *Inventory* el col·leccionable amb l'índex que es passa per paràmetre, cridant a la funció *onRetrieve* d'*Inventory*.
- **Die()**: Desactiva el colisionador per no detectar colisions durant l'estat de mort.
- **OnSpawn()**: Reactiva el colisionador.

  Nota: El mètodes *Collect(), Has() i Retrieve()*, de la interfície *ICollector,* s'han afegit en aquest component per aprofitar el colisionador del GameObject que conté l'script del component. Tot i que contenen lògica associada a l'*Inventory*, si els volguéssim posar en l'script, hauríem d'afegir un altre colisionador, perdent optimització de recursos.

9.4.4 Inventory

Guarda les armes a les que pot accedir l'entitat, així com una llista de col·leccionables. Aquesta llista correspon a un vector de valors booleans, els quals ens indiquen si tenim o no tenim el col·leccionable. Aquests són els mètodes del component:

- **Start():** Inicializa la llista de col·leccionables amb el nombre de possibles col·leccionables que hi ha en el nivell. Aquests s'estableixen en un Enum, a través del qual podem accedir als índex de la llista mitjançant els noms dels col·leccionables en comptes d'haver de recordar l'índex de cada un d'ells. Veure la **figura 9.3.2.**

```
public enum Collectibles
{
    key, //0
    flame //1
}
```

Figura 9.3.2: Enum d'elements col·leccionables

- **OnRetrieve(int)**: Rep l'índex d'un col·leccionable i l'estableix com a 'fals' a la llista de col·leccionables. Si es tracta de la flama, també crida a la funció *OnPowerUpFinished()* de *Appearance.*
- **activateCollectibleEvent(int):** Segons el col·leccionable que es passa per argument, es fa una acció o una altra. En el nostre cas, només es crida a *OnPowerUpObtanied()* d'*Appearance* i estableix que el personatge pot fer dash si el col·leccionable és la flama.

9.4.5 Movement

Té una referència al RigidBody de l'entitat i implementa els mètodes necessaris perquè els estats puguin accedir-hi. Aquests mètodes són:

- **LogicUpdate():** Guarda la velocitat actual del RigidBody de l'entitat.
- **SetVelocityZero():** Estableix la velocitat del personatge a 0.
- **SetVelocity():** Existeixen dos mètodes amb aquest mateix nom:
    - El primer rep per paràmetre una velocitat (*float*), un angle i una direcció (1=dreta, 0=esquerra).
    - El segon només rep una velocitat (*float*) i una direcció-
      Tots dos mètodes afegeixen velocitat al RigidBody del personatge a partir dels paràmetres que reben.
- **SetVelocityX(float)**: Estableix la velocitat del personatge en l'eix horitzontal.
- **SetVelocityY(float)**: Estableix la velocitat del personatge en l'eix vertical.
- **CheckIfShouldFlip(int)**: Segons el valor que se li passa per argument, corresponent a l'input de l'eix horitzontal, decideix si s'ha de capgirar el personatge o no. Això ho fa guardant un paràmetre que correspon a la direcció en la que està mirant el personatge en cada moment. Si l'input rebut és el contrari a aquesta direcció, el personatge s'ha de capgirar.
- **CheckIfCanDash()**: Retorna el valor del paràmetre *canDash*, que controla si el personatge és capaç de fer l'habilitat de dash o no.
- **ResetCanDash():** Estableix el paràmetre *canDash* a 'cert'.
- **Flip():** Capgira al personatge.
- **MoveTo(Vector3)**: Donada una posició, transporta al personatge directament cap aquesta. S'utilitza per fer reaparèixer al personatge.

9.4.6 Stats

Implementa la lògica necessària per controlar les estadístiques de l'entitat. Guarda la quantitat de vida màxima i restant en les variables *maxHealth i currentHealth.* Els mètodes implementa són:

- **DecreaseHealth(float)**: Resta a la vida restant el nombre passat per paràmetre. En cas que la vida restant resultant sigui 0 o inferior, es crida el *Die()* de *Combat* i l'*EntityDead()* del Core.
- **IncreaseHealth(float):** Suma a la vida restant el nombre passat per paràmetre.
- **GetInitialHealt():** Retorna la vida màxima o inicial.
- **GetCurrentHealth():** Retorna la vida restant actual.
- **OnSpawn():** Reinicia el valor de la vida actual perquè sigui igual que la màxima.

9.4.7 Muntatge del Core i els Core Components a l'entitat

El Core és un GameObject fill de la entitat, que només conté l'script *Core*. Cada Core Component existeix dins un GameObject fill de Core. Veure la **figura 9.3.3.**

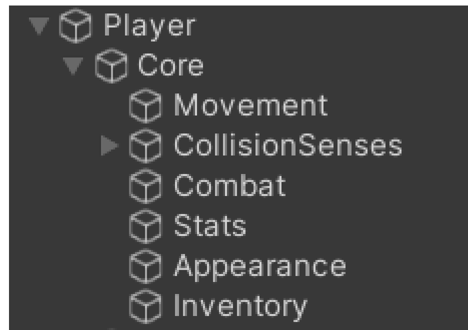


Figura 9.3.3: Jerarquía de GameObjects de Player visualitzada des de l'inspector de Unity

- **CoreComponent(Classe)**
  Per tal d'associar cada Core Component amb el Core pare, s'utilitza la classe *CoreComponent* de la qual tots els components hereden. La funcionalitat recau en afegir el propi Core Component a la llista de Core Components del Core pare:
  - S'obté el Core a través de la funció GetComponent<>();
  - Es crida a la funció AddComponent() del Core, passant-li el propi Core Component per paràmetre. Aquesta funció afegeix el Core Component a la llista del Core. Veure les **figures 9.3.4 i 9.3.5.**

```
public class CoreComponent : MonoBehaviour, ILogicUpdate{
    protected Core core;

    protected virtual void Awake()    {
        core = transform.parent.GetComponent<Core>();
        if (core == null) { Debug.LogError("There is no core on the parent"); }
        core.AddComponent(this);
    }

    public virtual void LogicUpdate() { }
}
```

Figura 9.3.4: Classe CoreComponent

```
private readonly List<CoreComponent> CoreComponents = new List<CoreComponent>();

public void AddComponent(CoreComponent component){
    if(!CoreComponents.Contains(component))
    {
        CoreComponents.Add(component);
    }
}
```

Figura 9.3.5: Fragments de codi de Core encarregats de guardar i afegir els Core Components

Alguns dels Core Components implementen lògica que s'ha d'executar a cada frame, en sincronía amb el mètode *LogicUpdate()* dels estats de l'entitat.
És per això que aquesta classe implementa una interfície *ILogicUpdate.* La funció d'aquesta és que es pugui cridar a la única funció que implementa, *LogicUpdate(),* des de l'*Update()* de l'entitat. D'aquesta manera a cada *Update()* s'executarà la lògica implementada en el *LogicUpdate()* dels estats i de tots Core Components que la utilitzin. Veure les **figures 9.3.6 i 9.3.7.**

```
private void Update(){
    Core.LogicUpdate();
    StateMachine.CurrentState.LogicUpdate();
}
```

Figura 9.3.6: Update() de l'entitat

```
public void LogicUpdate(){
    foreach (CoreComponent component in CoreComponents){
        component.LogicUpdate();
    }
}
```

Figura 9.3.7: LogicUpdate() de Core

Per últim, l'entitat obté el seu Core amb la comanda:

```
Core = GetComponentInChildren<Core>();
```

9.5 Màquina d'estats del Player

En aquest apartat expliquem la implementació les diferents parts que formen la màquina d'estats encarregada de gestionar el moviment del jugador.

9.5.1 Superclasse dels estats del Player: Script PlayerState

La lògica del *PlayerState* segueix la base del **State** explicat a l'apartat 8.4.Màquines d'estat, per tant, en aquest apartat expliquem només els elements que s'han afegit específicament pel funcionament de la màquina d'estats del *Player*.
Recordem que tots els estats del Player que s'explicaran en els pròxims apartats hereden d'aquesta classe.

- S'han afegit els següents **paràmetres:**
    - **playerData**: Serveix perquè l'estat pugui accedir a les dades necessàries relacionades amb el personatge. Li és passat al constructor.

- **isAnimationFinished:** Aquest paràmetre ens serveix per certs estats que han de saber quan s'ha acabat al seva animació corresponent. S'inicialitza a 'fals'.

- **isExitingState:**L'ús d'aquest paràmetre és controlar els moments en els quals s'ha decidit canviar d'estat, però l'estat que s'està abandonant encara té lògica executant-se. Amb aquest paràmetre podem evitar conflictes entre la lògica de l'estat que s'abandona i l'estat al que s'entra. S'inicialitza a 'fals'.

- Conté els següents **mètodes** específics
    - **AnimationTrigger()**: Aquesta acció serveix per ser cridada com a event durant una animació. Perquè cada estat no hagi de crear una funció nova que gestioni els events de l'animació, tots podran cridar a aquest mateix mètode en el cas que ho necessitin.

    - **AnimationFinishTrigger()**: De la mateixa manera que en *l'AnimationTrigger()*, aquest és un mètode que es cridarà com un event de l'animació. La idea és que es cridi a l'últim frame de l'animació de l'estat. El seu contingut posa el valor de *isAnimationFinished* a 'cert'. D'aquesta manera, podem controlar el moment en el que s'acaba l'animació d'un estat cridant a aquesta funció des de la mateixa animació
    -

9.5.2 Gestió de l'input: Input System i InputHandler

Per gestionar els inputs que es reben del jugador, hem utilitzat l'*Input System* de Unity. Amb aquest podem definir els diferents inputs i quin tipus de valor retornen. Veure la **figura 9.4.1.**

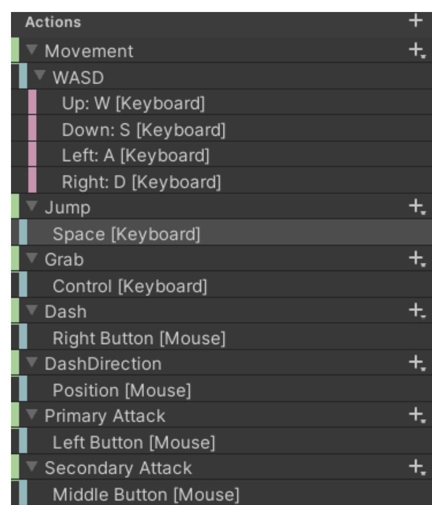


Figura 9.4.1: Diferents inputs que es reben pel control del Player

A partir d'aquests, l'*Input System* ens permet definir des de l'inspector a quines funcions volem cridar quan es detecti l'activació de cada un dels inputs. Veure la **figura 9.4.2.**

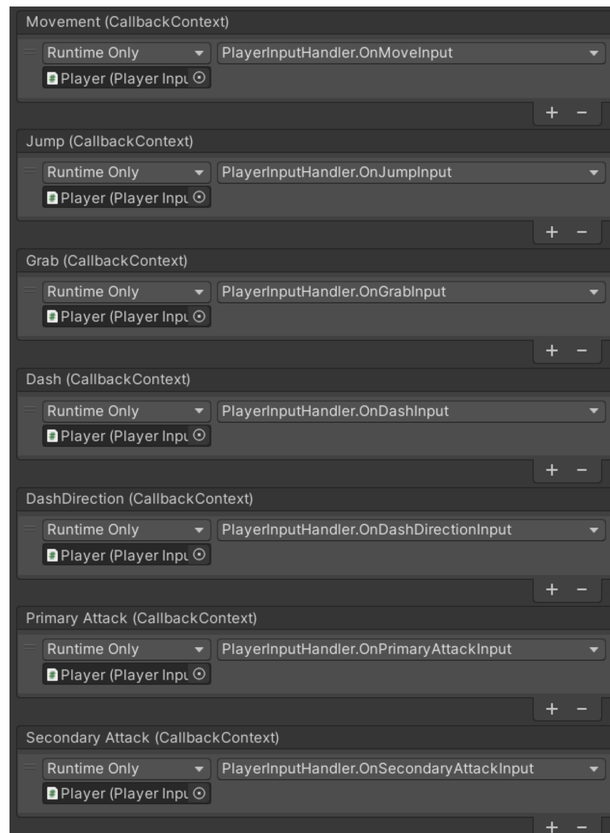


Figura 9.4.2: Assignació de funcions a cada input

L'script que conté aquests mètodes és l'*InputHandler*, a través del qual es comunica el *Player* per saber si s'ha rebut algun input i quin valor ha agafat. Quan es detecta un event de input, aquest passa a la funció que el rep un objecte de tipus **CallBackContext.** Amb aquest objecte es pot saber quan s'ha apretat i quan s'ha deixat d'apretar un botó.

Els mètodes que implementa **l'InputHandler** són:

- **Update():** Crida a *CheckJumpInputHoldTime() i CheckDashInputHoldTime()*.
- **GivePlayerControl():** Retorna el control del personatge al jugador. S'utiltiza la variable booleana ***canHandleInput*** per controlar quan els inputs del jugador poden arribar al personatge i quan no.
- **TakePlayerControl():** Estableix tots els valors de inputs a 'fals' o 0 segons convingui. Posa a 'fals' la variable *canHandleInput*.
- **OnPrimaryAttackInput(CallBackContext)**: Gestiona l'ínput d'atac.
- **OnMoveInput(CallBackContext):** Gestiona l'input de moviment en l'eix horitzontal i vertical.
- **OnJumpInput(CallBackContext):** Gestiona l'input de salt.
- **OnGrabInput(CallBackContext):** Gestiona l'input d'agafar-se a la paret.
- **OnDashInput(CallBackContext):** Gestiona l'input de l'habilitat de dash.
- **OnDashDirectionInput(CallBackContext):** Gestiona l'input corresponent al ratolí en la pantalla quan es vol escollir en quina direcció realitzar el dash.

- **CheckJumpInputHoldTime():** Controla el temps que ha passat des de que s'ha detectat l'input de salt. Quan es passa del màxim, el desactiva.
- **CheckDashInputHoldTime():** Controla el temps que ha passat des de que s'ha detectat l'input de dash. Quan es passa del màxim, el desactiva.

9.5.3 Entitat controlada per la màquina d'estats: script *Player*

Tal i com s'ha detallat en l'explicació teòrica sobre la màquina d'estat, aquest és el programa que conté la **màquina d'estats**, els **estats** en els que es pot trobar i fa ús de l'element **Core** amb tots els seus components. A part, aquest script necessita accedir i utilitzar els següents components pel correcte funcionament dels estats:

Components de l'Script Player

- **Animator:** GameObject que controla les animacions corresponents a cada estat del personatge.
- **InputHandler:** Script corresponent a la gestió de l'input del jugador. Explicat en l'apartat anterior 9.5.2 Gestió de l'input: Input System i InputHandler
- **RigidBody:** Component que permet al nostre personatge actuar sota el control de la física (rebre força, interactuar amb altres objectes de manera realista, ésser influenciat per la gravetat, etc.). A través d'aquest element podrem modificar el comportament del moviment del personatge des de cada estat mitjançant el Core Component **Movement**.
- **DashDirectionIndicator:** GameObject que ens servirà per visualitzar la direcció a la que volem fer el *dash*. Se'n fa ús a l'estat 9.5.5.7 AbilityState > DashState.
- **PlayerData:** Aquest és un *Scriptable Object* que utilitzem per tal de guardar totes les dades necessàries pels diferents estats de manera que sigui fàcil modificar-les des de l'inspector. Cada estat del *player* podrà accedir-hi per tal de obtenir els valors que facin falta. Veure la **figura 9.4.3.**

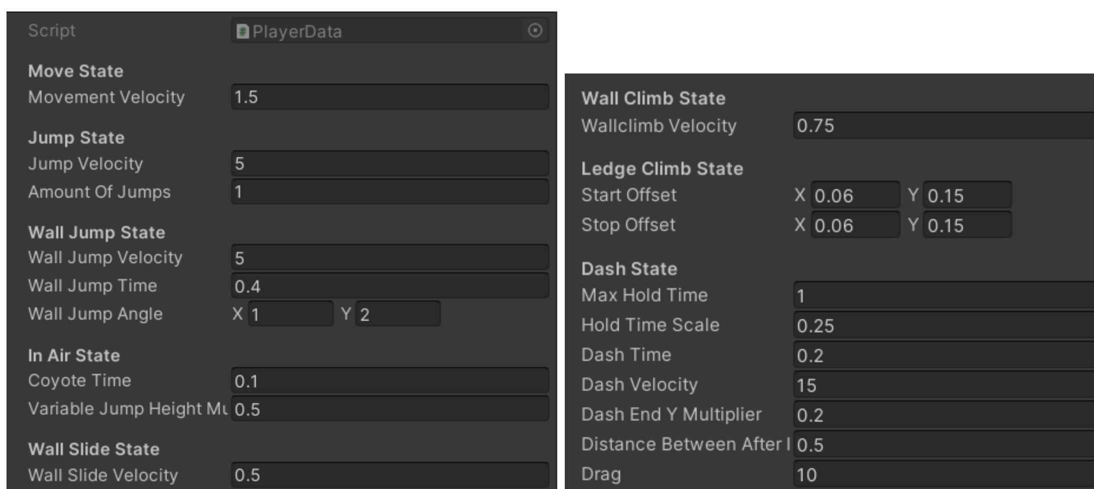


Figura 9.4.3: Scriptable Object PlayerData descde l'inspector

- **GameController**: *Script* que s'encarrega de gestionar diferents elements presents en l'escena. El seu paper dins d'aquesta màquina d'estats s'explica a l'estat 9.5.5.16 DeadState (Estat sense Superestat).

Així doncs, podem visualitzar les parts que formen part d'aquest *script* de la següent manera. Veure la **figura 9.4.4.**

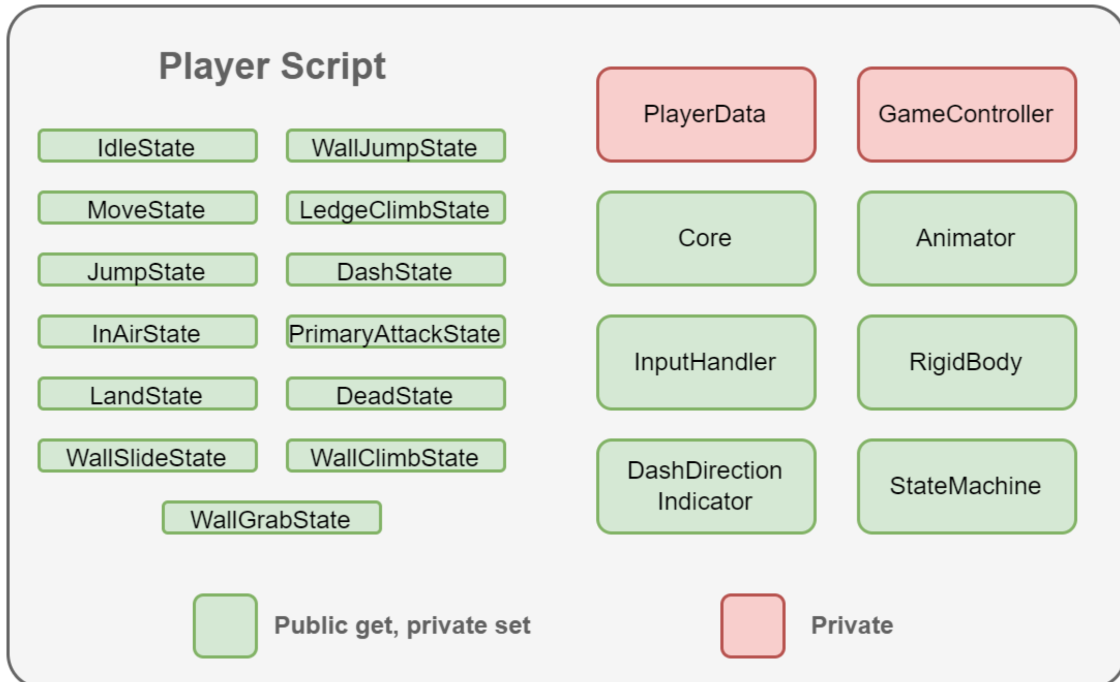


Figura 9.4.4: Diagrama amb els elements de l'script Player

<u>Mètodes de l'script Player</u>

- **Awake()**: Inicialitzem tots els estats, així com el Core i la màquina d'estats.

- **Start() :**Inicialitzem els components restants (Animator, InputHandler, RigidBody, DashDirectionIndicator). Inicialitzem l'estat d'atac *PrimaryAttackState* assignant-li una arma. L'explicació sobre aquesta part del codi es troba a l'apartat 9.4.3 Combat. Inicialitzem també la màquina d'estats amb l'IdleState (estat de 'parat' o 'quiet'). Veure la **figura 9.4.5.**

```
private void Start(){
        Anim = GetComponent<Animator>();
        InputHandler = GetComponent<PlayerInputHandler>();
        RB = GetComponent<Rigidbody2D>();
        DashDirectionIndicator = transform.Find("DashDirectionIndicator");

        PrimaryAttackState
            .SetWeapon((int) CombatInputs.primary);
```

```
        StateMachine.Initialize (IdleState);
}
```

Figura 9.4.5: Mètode Start() de Player

- **Update():** Executa el *LogicUpdate()* de cada Core Component i de l'estat actual de la màquina d'estats.

- **FixedUpdate():** Executa el *PhysicsUpdate()* de l'estat actual a la màquina d'estats.

- **AnimationTrigger():** Funció que es pot cridar com a *trigger* en el transcurs d'una animació. Crida a l'*AnimationTrigger()* de l'estat actual.

- **AnimationFinishTrigger():** Funció que es pot cridar com a *trigger* al finalitzar una animació. Crida a l'*AnimationFinishTrigger()* de l'estat actual.

- **Dead()**: Força un canvi d'estat a l'estat *DeadState*. Es crida quan el jugador mor.

- **Spawn()**: Fa reaparèixer al jugador a la posició de l'últim *CheckPoint.*

9.5.4 Estructura jeràrquica i canvis entre estats

Quan el Payer es troba en un estat, han de donar-se certes condicions perquè passi al següent. Hi han molts estats que comparteixen característiques i les condicions de sortida cap a un altre cert estat seran les mateixes. Per posar un exemple, la condició per a què el personatge salti, ja estigui aquest en l'estat de 'quiet' com en l'estat de 'córrer', serà que el jugador cliqui l'espai. De la mateixa manera, hi haurà més d'un estat que compartirà part de les condicions d'entrada.

Degut a això, en comptes d'enllaçar cada estat amb tots els altres als que pot anar, es separen els estats en **superestats** i **subestats**. El **superestat** conté els elements comuns dels seus subestats i controlarà les condicions de canvi d'estat que siguin iguals entre ells. S'encarrega dels canvis cap a altres superestats. El **subestat** hereda del seu superestat i conté els elements específics que no puguin obtenir a través de l'herència d'aquest. Només han de controlar les condicions de canvi d'estat cap a subestats del mateix superestat.

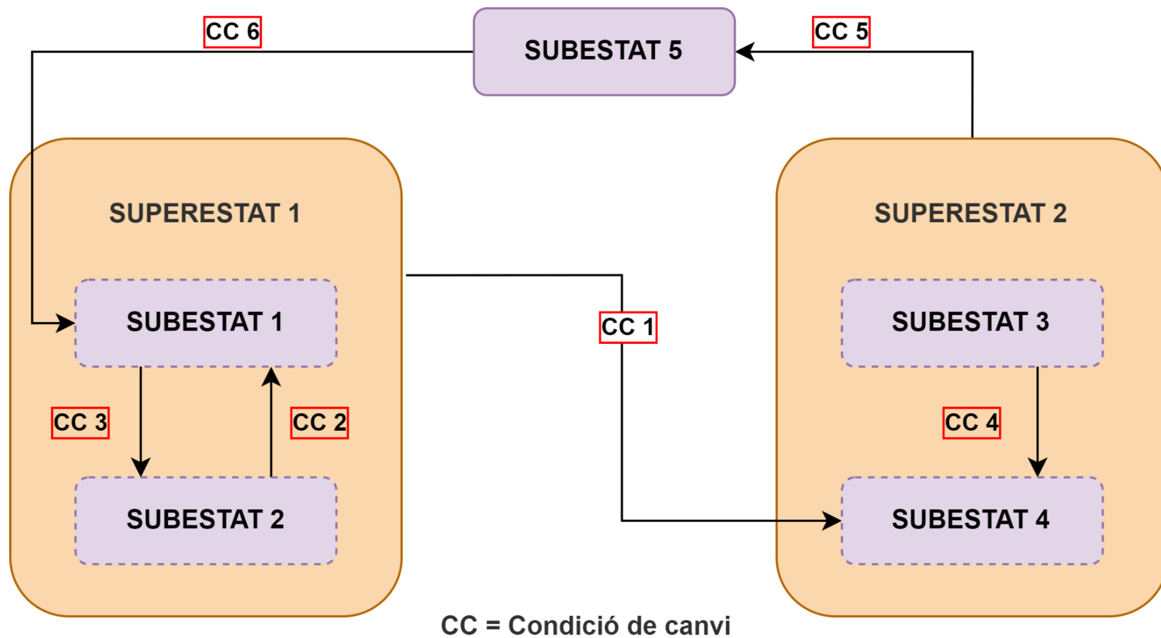


Figura 9.4.6: Diagrama d'exemple de canvis entre estats i superestats

Fixant-nos en la **figura 9.4.6** , podem veure diferents situacions que es poden donar en quant a canvis d'estat:

- La condició **CC1** es comprova al **SUPERESTAT 1**. Això significa que, gràcies a l'herència, independentment de si el personatge es troba al SUBESTAT 1 o al SUBESTAT 2, si es dona aquesta condició es canviarà d'estat al SUBESTAT 4.
- Els subestats dins un superestat només es comuniquen entre altres subestats del mateix superestat.
- Existeixen subestats que no pertanyen a superestats.

9.5.5 Estats de la màquina pel Player

Una vegada explicada l'estructura i els elements principals de la màquina d'estats del Player, en aquest apartat expliquem els superestats i tots els estats en els que es pot trobar. Per cada estat s'especifiquen les condicions de canvi a altres estats ordenades de **més a menys prioritat**, indexades per tal de trobar-les en els diferents diagrames que es mostren, i s'explica el funcionament fora de la lògica heredada de les superclasses.

9.5.5.1 SUPERESTAT: GroundedState

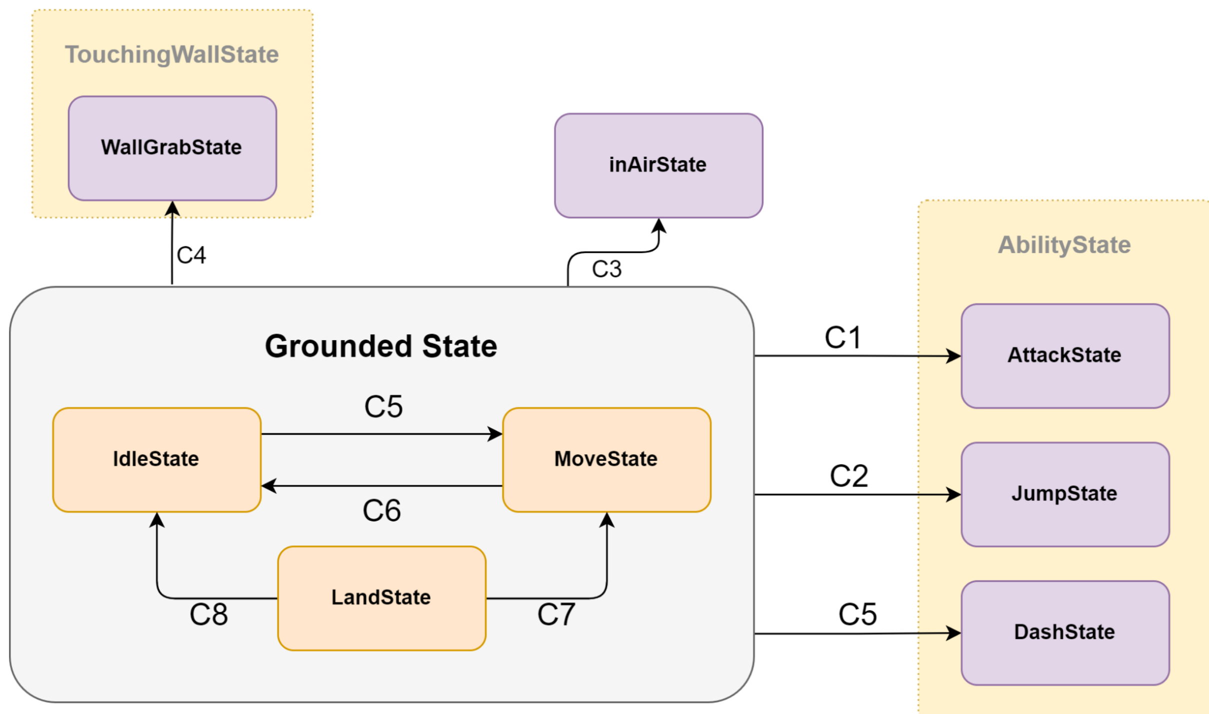


Figura 9.5.1: Diagrama de canvis d'estat des de *GroundedState* i els seus sub-estats

Engloba els estats en els que el Player pot estar mentres aquest està tocant el terra. Veure la **figura 9.5.1.**

Condicions de canvi d'estat:

- **C1:** Es detecta l'input d'atac => Canvi a AttackState
- **C2:** Es detecta l'input de salt i pot saltar => Canvi a JumpState
- **C3:** No està tocant el terra => Canvi a InAirState (Prèviament crida al mètode *StartCoyoteTime* de *InAirState*)
- **C4:** Està tocant una paret i una cantonada (es detecten les dues) i es detecta l'input d'agafar-se a paret =>Canvi a WallGrabState
- **C5:** Es detecta input de dash, i el personatge pot fer-lo => Canvi a DashState.

- **Mètodes i funcionament**

    - **Enter():** Reestableix el nombre de salts disponibles pel jugador a través del JumpState
    - **DoChecks():** Fa les comprovacions necessaries per detectar si està tocant el terra, una paret a davant o una cantonada.
    - **LogicUpdate():** Llegeix els inputs necessaris pels subestats i comprova les condicions de canvi a altres estats.

9.5.5.2 GroundedState > IdleState

El personatge està quiet al terra.

- **Condicions de canvi d'estat:**
    - **C5:** Es detecta input a l'eix X i no s'està sortint de cap altre estat => Canvi a MoveState
- **Mètodes i funcionament**
    - **Enter():** Estableix la velocitat del jugador a 0.

9.5.5.3 GroundedState > MoveState

El personatge es mou cap a la dreta o l'esquerra mentres està tocant el terra.

- **Condicions de canvi d'estat:**
    - **C6:** No es detecta input a l'eix X i no s'està sortint de cap altre estat => Canvia a IdleState
- **Mètodes i funcionament**
    - **LogicUpdate():** Comprova si el personatge s'ha de capgirar i estableix la velocitat del personatge.

9.5.5.4 GroundedState > LandState

El personatge acaba de tocar el terra després d'un salt o una caiguda.

- **Condicions de canvi d'estat:**
    - Si no està sortint d'un altre estat i:
        - **C7:** Es detecta input a l'eix X => Canvi a MoveState
        - **C8:** S'acaba l'animació de l'estat => Canvi a IdleState
- **Mètodes i funcionament**

    Aquest estat utilitza un event que es crida al final de l'animació des de l'Animator. Aquest event correspon al *AnimationFinishTrigger()*, i permet saber quan s'ha acabat l'animació.

9.5.5.5 SUPERESTAT: AbilityState

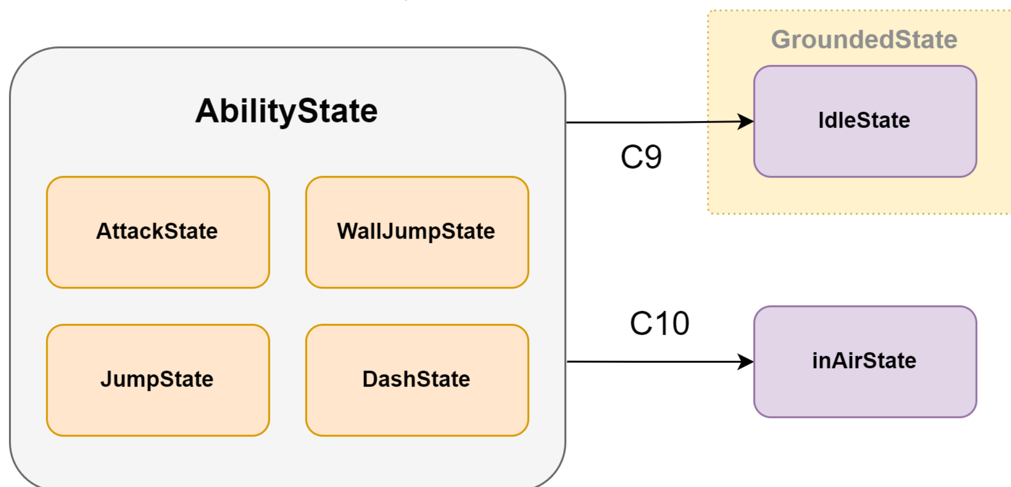


Figura 9.5.2: Diagrama de canvis d'estat d'*AbilityState*

Aquest engloba els estats corresponents a les habilitats del personatge. S'encarrega de gestionar la sortida cap al nou estat una vegada s'hagi acabat l'execució de l'habilitat. Veure la **figura 9.5.2.**

- **Condicions de canvi d'estat:**
  - Si s'ha acabat l'execució de l'habilitat i:
    - **C9:** Està tocant el terra i la velocitat vertical és 0 => Canvi a IdleState
    - **C10:** Si no està tocant el terra => Canvi a InAirState

- **Mètodes i funcionament**

  Mitjançant l'ús d'un paràmetre booleà *isAbilityDone* al que poden accedir els subestats, els quals s'encarreguen de posar-lo a 'cert', es controla el moment en el que s'acaba l'execució de l'habilitat i per tant s'ha de passar a un nou estat.

  - **Enter():** Estableix el paràmetre *isAbilityDone* a 'fals'.
  - **DoChecks():** Comprova si està tocant el terra.

9.5.5.6 AbilityState > JumpState

Habilitat de saltar del personatge.

- **Condicions de canvi d'estat:**

- Totes gestionades per el superestat pare AbilityState

- **Mètodes i funcionament**

  Aquest estat fa servir un paràmetre enter *amountOfJumpsLeft* que segueix el compte del nombre de salts que li queden disponibles per fer. Tot i que en el nostre nivell el personatge mai pot fer més d'un salt seguit sense tocar el terra, perquè així ho hem decidit. Aquest estat permet fer el que se'n diu un 'salt doble', o tants salts seguits com es vulguin.

  - **Enter():**
    - Estableix l'input de salt a 0 a través de l'*InputHandler*.
    - Afegeix velocitat a l'eix Y per tal de propulsar el personatge a l'aire.
    - Indica el final de l'habilitat una vegada ja s'ha propulsat al personatge.
    - Resta 1 a *amountOfJumpsLeft*.
    - Crida la funció *SetIsJumping()* de l'estat InAirState perquè quan hi entri aquest sàpiga que el personatge hi ha entrat a través d'un salt.
  - **CanJump():** Retorna 'cert' si *amountOfJumpsLeft* és més gran que 0. Altrament retorna 'fals'.
  - **ResetAmountOfJumpsLeft():** Reestableix *amountOfJumpsLeft* al nombre inicial.
  - **DecreaseAmountOfJumpsLeft():** Resta 1 a *amountOfJumpsLeft*.

9.5.5.7 AbilityState > DashState

Habilitat de fer *dash* del personatge.

- **Condicions de canvi d'estat:**

  - Totes gestionades pel superestat pare AbilityState

- **Mètodes i funcionament**

  Aquesta habilitat consisteix en recórrer una distànca en línia recta a gran velocitat, cap a la direcció que desitgi el jugador.
  Per implementar aquesta habilitat l'estat es separa en dues parts:
    - **Primera part:** el jugador manté el botó apretat i, amb el punter del ratolí, apunta cap a la direcció on desitja desplaçar-se. Per tal de visualitzar millor aquesta direcció, s'utilitza el *DashDirectionIndicator*, un GameObject del Player que representa una fletxa de direcció.

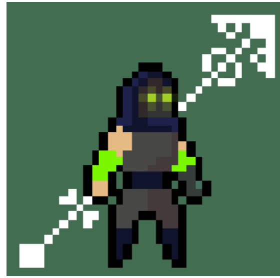


Figura 9.5.3: *DashDirectionIndicator* del Player

- **Segona part:** el jugador ha deixat anar el botó o bé s'ha acabat el temps màxm que el podia tenir apretat. S'aplica la velocitat al personatge cap a la direcció que hagi escollit el jugador.

- **Enter():**
    - Estableix a l'*InputHandler* que s'ha utilitzat l'input corresponent a l'habilitat del dash.
    - Inicialitza a 'cert' un paràmetre *isHolding* per controlar si el jugador està prement el botó o bé l'ha deixat anar.
    - Activa el *DashDirectionIndicator*
- **LogicUpdate():**
    - **Primera part:**
        - Estableix la rotació del *DashDirectionIndicator* segons la posició del ratolí obtenint l'input a través de l'*InputHandler.*
        - Si es deixa de prèmer el botó o s'acaba el temps màxim:
            - Canvia el paràmetre *isHolding* a 'fals'.
            - Treu la flama de l'*Inventory* del Player perquè no pugui tornar a fer dash.
            - Afegeix la nova velocitat al personatge.
            - Desactiva el *DashDirectionIndicator*.
    - **Segona part:**
        - Segueix establint la velocitat al personatge fins que s'acaba un temps màxim establert a *PlayerData*.
        - Quan s'acaba el temps, indica el final de l'habilitat.

9.5.5.8 AbilityState > WallJumpState

El personatge salta mentres es troba en una paret.

- **Condicions de canvi d'estat:**

    - Totes gestionades pel superestat pare AbilityState

- **Mètodes i funcionament**

  Similar a l'estat JumpState. Es diferencia en que prèviament s'ha d'establir la direcció del salt segons si està tocant a la paret o no. A més, no permet canviar a cap altre estat fins que passa un temps mínim.

  - **Enter():**
    - Afegeix velocitat el personatge mitjançant un angle i una direcció establerts a *PlayerData*.
    - Comprova si s'ha de capgirar el personatge en cas que estigués mirant de cares a la paret.
    - Resta 1 al *amountOfJumpsLeft* de l'estat *JumpState* per tal que no pugui tornar a saltar enmig de l'aire.

  - **LogicUpdate():** Continúa establint la velocitat del personatge fins que s'acaba el temps mínim configurat a *PlayerData*.
  - **DetermineWallJumpDirection(bool):** Si el personatge estava tocant a la paret quan s'ha iniciat el salt, estableix la direcció de salt cap endarrera. Altrament l'estableix cap endavant.

9.5.5.9 AbilityState > AttackState + Weapon

Estat en el qual el personatge ataca. Com que el funcionament està lligat al funcionament de les armes, en aquest apartat també s'expliquen el funcionament d'aquestes. A l'inici del desenvolupament volíem que el personatge pogués atacar amb armes diferents, les qual tindrien diferents característiques. Per tal de fer això, era necessari implementar l'objecte 'arma' separadament de l'estat d'atac del jugador. Així doncs, l'estat d'atac del jugador fa ús de l'arma que tingui assignada en el moment que el jugador ha apretat el botó d'atacar i, quan l'arma hagi acabat l'execució de la seva lògica, l'estat d'atac s'encarrega de seguir amb el fluxe de la màquina d'estats.

9.5.5.9.1 Arma: Script Weapon

Aquest és l'script base d'una arma. El GameObject que conté aquest script té dos GameObjects fills, *Weapon* i *Base* que contenen un Animator amb l'animació de l'arma i l'animació d'atac del personatge respectivament. Veure la **figura 9.5.4.**

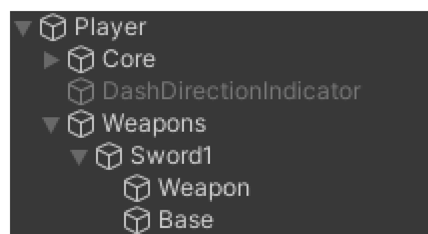


Figura 9.5.4: Jerarquía de GameObjects d'una arma a l'inspector

- **GameObject Weapon**: Conté un *Collider* encarregat de detectar els enemics, el qual segueix el recorregut de l'arma en cada frame de l'animació d'aquesta. Veure la **figura 9.5.5.**

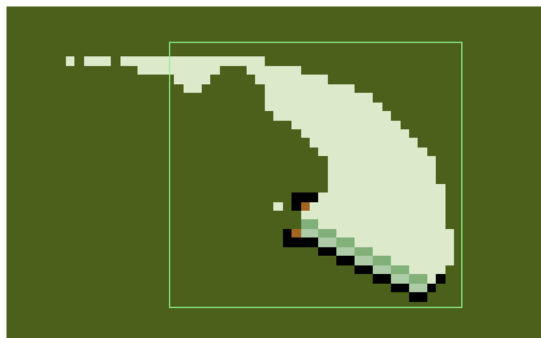


Figura 9.5.5: Frame de l'animació de l'objecte *Weapon.* El requadre verd representa el *Collider*

- **GameObject Base:** Conté l'animació d'atac del personatge, la qual crida els events de l'script *Weapon*. La funció d'aquests events és la de poder decidir en quins frames de l'animació volem que es cridin certs mètodes que definiran com es comporta el personatge durant l'atac. Aquests són mètodes que rebrà l'estat *AttackState*. Com que el GameObject que conté aquest script és el seu pare, utilitzem un script intermediari *WeaponAnimationToWeapon* per tal de comunicar els dos GameObjects. Veure la **figura 9.5.6.**

Figura 9.5.6: Frame de l'animació de l'objecte *Base.* Correspon al mateix frame de l'animació de l'objecte *Weapon* a la **figura 9.5.5.**

En el mètode *Awake(),* es desactiva el propi GameObject i conseqüentment els seus fills *Weapon i Base.* Quan es vol atacar, s'activa el GameObject i s'inicien les animacions de *Weapon* i *Base* alhora. A partir d'aquell moment, l'script *Weapon* cridarà mètodes de *AttackState* segons els events establerts en els frames de l'animació d'atac.

Per tal de poder comptar amb diferents tipus d'atac per la mateix arma, aquesta utilitza un ScriptableObject que conté una llista de paràmetres per cada atac que es vulgui tenir.

Cada atac té els següents paràmetres:

- **Nom**
- **Velocitat de moviment** : Determina la velocitat en la que avançarà el personatge quan faci l'atac.
- **Quantitat de mal:** El mal que rebrà l'enemic al que s'estigui atacant.
- **Força de 'knockback'**: La força amb la que empeny als enemics que toqui.
- **Angle de 'knockback':** La direcció en la que empeny als enemics que toca.

S'utilitza una variable *attackCounter* que manté el compte de quin ha de ser el pròxim atac a fer. D'aquesta manera es pot controlar a través de l'Animator quina animació toca reproduir segons l'atac que toqui fer i quins són els paràmetres que corresponen a aquest atac.Veure la **figures 9.5.7 i 9.5.8.**

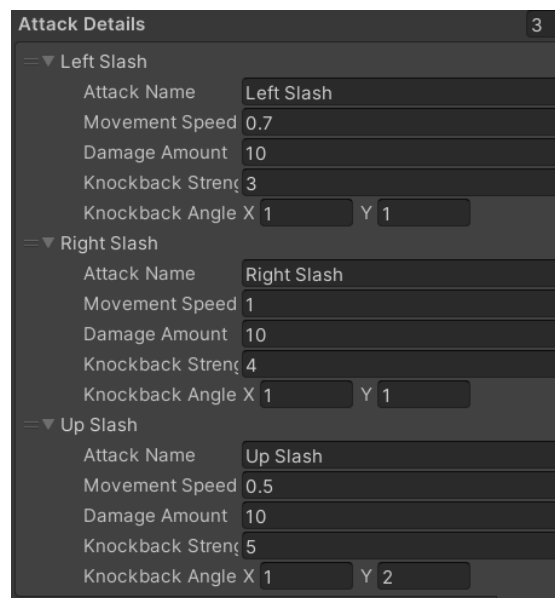


Figura 9.5.7: Scriptable Object de Weapon amb els paràmetres dels atacs

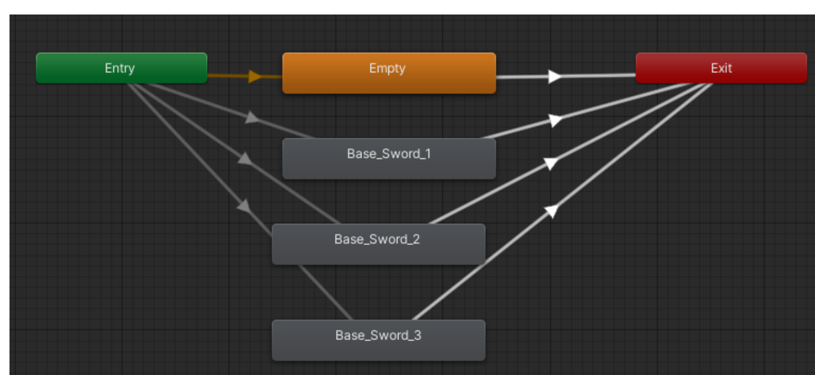


Figura 9.5.8: Animator del GameObject *Base*. Segons el valor d'*AttackCounter* es reprodueix una animació o una altra

Per tal d'aplicar mal als enemics, es busquen objectes que col·lisionin amb el *Collider* del GameObject *Weapon* i que implementin les interfícies *IDamageable* o *IKnockbackable*. Si se'n detecta algun, es crida al corresponent *Damage()* i *Knockback()* passant per paràmetre els valors de l'atac que s'estigui realitzant.

9.5.5.9.2 Script AttackState

Per tal de mostrar les animacions dels GameObjects *Weapon i Base*, l'animació corresponent a aquest estat és una de buida. Si no fos així es dibuixarien dos personatges durant els atacs.

- **Condicions de canvi d'estat:**
    - Totes gestionades pel superestat pare AbilityState
- **Mètodes i funcionament**
    - **SetWeapon(int)**: Aquest mètode es crida a l'inicialització de l'script *Player* i assigna a aquest estat l'arma que utilitzarà. Aquesta s'obté a través de l'*Inventory.*
    - **Enter()**: Activa el GameObject corresponent a l'arma, iniciant un atac.
    - **SetPlayerVelocity(float)**: Aquest mètode és cridat des d'events durant l'animació de l'atac i serveix per fer que el personatge avanci una mica durant l'atac o bé perquè no pugui avançar mentre el realitza. Rep per paràmetre la velocitat de l'atac que s'està realitzant, o bé 0 en el cas que es vulgui parar al personatge.
    - **SetFlipCheck()**: Aquest mètode és cridat des d'un event durant l'animació de l'atac. Serveix per controlar en quins moments de l'atac el jugador pot canviar la direcció cap a on està mirant i en quins no.

9.5.5.10 SUPERESTAT: TouchingWallState

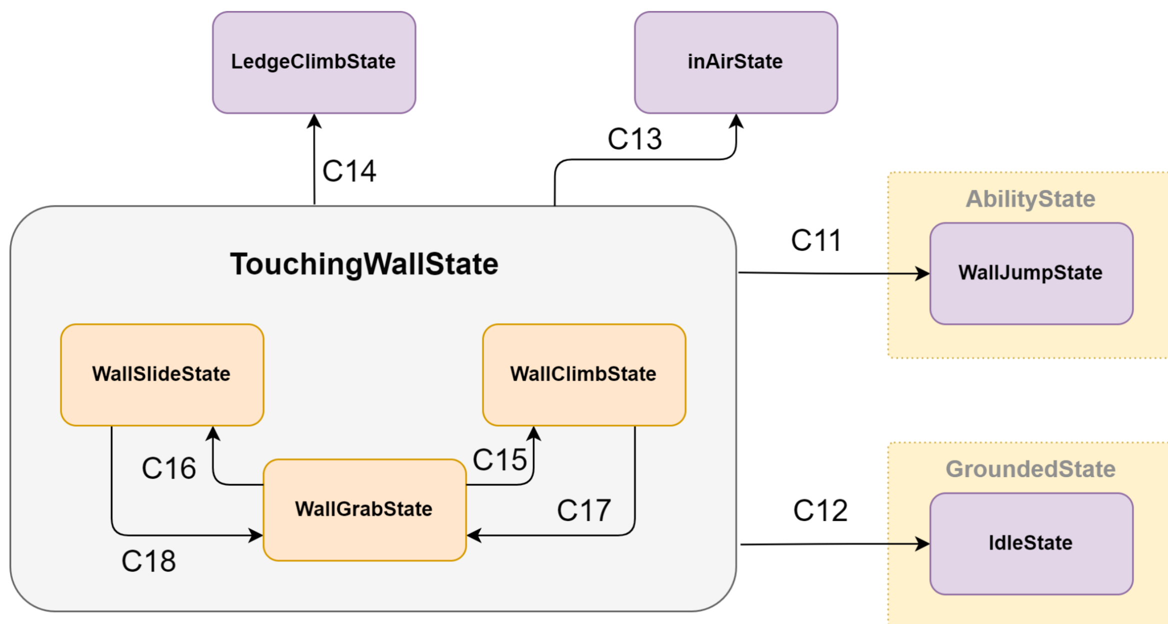


Figura 9.5.9: Diagrama de canvis d'estat des de *TouchingWallState* i els seus sub-estats

Aquest superestat engloba els estats del personatge quan aquest interactua amb una paret. S'encarrega també de fer les comprovacions sobre l'entorn necessàries, ja que tots els seus subestats en fan ús. Veure la **figura 9.5.9.**

- **Condicions de canvi d'estat:**
    - **C11:** Si es detecta l'input de salt => Canvi a WallJumpState (Crida el *DetermineWallJumpDirection* del mateix abans.
    - **C12:** Si està tocant el terra i no es reb input d'agafar-se a la paret => Canvi a IdleState
    - **C13:** Si deixa de tocar la paret o no es detecta l'input en l'eix X cap a la direcció de la paret ni l'input d'agafar-s'hi => Canvi a InAirState.
    - **C14:** Si es detecta una cantonada (està tocant la paret però el detector de cantonades no detecta la paret) => Canvi a LedgeClimbState

- **Mètodes i funcionament**
    - **DoChecks():** Comprova si està tocant el terra, si està tocant una paret a davant o si està detectant una cantonada. Al detectar una cantonada crida la funció *SetDetectedPosition*() del LedgeClimbState

9.5.5.11 TouchingWallState > WallGrabState

El personatge està agafat a una paret.

- **Condicions de canvi d'estat:**
    - Si no està sortint de cap altre estat i:
        - **C15:** Es detecta input positiu en l'eix Y => Canvi a WallClimbState
        - **C16:** Es detecta input negatiu en l'eix Y o es deixa de detectar l'input d'agafar-se a la paret => Canvi a WallSlideState

- **Mètodes i funcionament**
    - **Enter():** Inicialitza el valor del paràmetre *holdPosition*, corresponent a la posició del jugador en el moment d'entrar a l'estat.
    - **LogicUpdate():** Crida a *HoldPosition().* Això es fa per mantenir el personatge quiet en la mateixa posició, ja que sinó la força de la gravetat el faria baixar.
      HoldPosition(): Estableix la posició del jugador a la posició inicial *holdPosition*. També estableix la velocitat del personatge a 0, tant en l'eix vertical com horitzontal.

9.5.5.12 TouchingWallState > WallClimbState

El personatge escala verticalment i cap amunt per una paret.

- **Condicions de canvi d'estat:**
    - **C17:** Si no està sortint de cap altre estat i es deixa de detectar l'input en l'eix Y => Canvi a WallGrabState

- **Mètodes i funcionament**
    - **LogicUpdate():** Estableix la velocitat del personatge en l'eix Y corresponent a la velocitat d'escalada de *PlayerData*

9.5.5.13 TouchingWallState > WallSlideState

El personatge es deslliça verticalment per la paret.

- **Condicions de canvi d'estat:**

    - **C18:** Si no està sortint de cap altre estat, es detecta l'input d'agafar-se a la paret i no es detecta input de moviment en l'eix Y => Canvi a WallGrabState

- **Mètodes i funcionament**

    - **LogicUpdate():** Estableix la velocitat del personatge en l'eix Y corresponent a la velocitat de deslliçament per la paret de *PlayerData.*

9.5.5.14 InAirState (Estat sense Superestat)

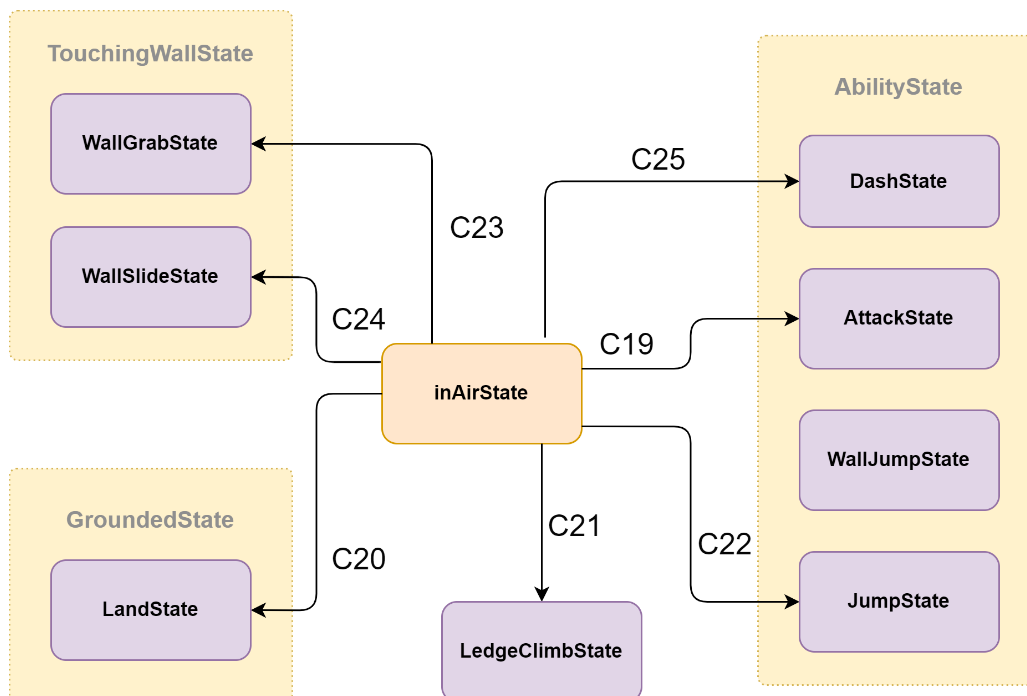


Figura 9.5.10: Diagrama de canvis d'estat des de *inArState*

Aquest estat conté la lògica corresponent a quan el jugador es troba 'a l'aire'. Hi pot arribar a través d'un salt, un dash, un salt des d'una paret o bé simplement deixant-se caure. Veure la **figura 9.5.10.**

- **Condicions de canvi d'estat:**

    - **C19:** Si es detecta l'input d'atac => Canvi a AttackState
    - **C20:** Si es detecta el terra i la velocitat en l'eix vertical és 0 => Canvi a LandState
    - **C21:** Si es detecta una cantonada i no es detecta terra => Canvi a LedgeClimbState.

- Si es detecta l'input de salt i s'està tocant una paret a davant o a darrera => Canvi a WallJumpState. (Prèviament crida la funció *DetermineWallJumpDirection* de *WallJumpState* perquè aquest pugui establir la direcció del salt.
- **C22:** Si es detecta l'input de salt i el *CanJump()* de *JumpState* retorna 'cert' => Canvi a JumpState
- **C23:** Si tant el detector de parets com el de cantonades detecten paret i es reb l'input d'agafar-se a la paret => Canvi a WallGrabState
- **C24:** Si s'està tocant una paret, la velocitat en l'eix vertical és negativa i es detecta input de moviment en l'eix horitzontal cap a la direcció on s'està mirant => Canvi a WallSlideState
- **C25:** Si es detecta input de dash i el personatge pot fer dash (ha capturat una flama anteriorment) => Canvi a DashState

- **Mètodes i funcionament**

  - **DoChecks():**
    - Es fan les comprovacions sobre si el jugador està tocant el terra, una paret a davant, una paret a darrera i una cantonada.
    - Si es detecta una cantonada, crida al *SetDetectedPosition* del *LedgeClimbState* per a què aquest pugui accedir a la posició del jugador en el moment d'entrada a l'estat.
    - Estableix els valors dels paràmetres *isTouchingWall, oldIsTouchingWall, isTouchingWallBack* i *oldIsTouchingWallBack*.

  - **CheckJumpMultiplier()**: Al *JumpState*, es posa a 'cert' un paràmetre de *InAirState* que fa que sàpiga que s'hi ha entrat a través de l'habilitat de salt. Aquest mètode serveix per a què el jugador pugui controlar l'alçada del salt del personatge segons el temps que mantingui apretat el botó de salt.
    - Mentres estigui apretant el botó, no es farà res i es deixarà que l'impuls inicial donat a l'estat de *JumpState* actui lliurement, i el personatge només es frenarà gràcies a la força de la gravetat.
    - En el moment que es detecti que es deixa anar el botó de salt, s'aplica un multiplicador (hem trobat que un valor de 0.5 funciona bé) a la velocitat vertical del personatge, fent que el salt s'acabi més ràpid.

  - **LogicUpdate():**
    - Crida a *CheckCoyoteTime()*, *CheckWallJumpCoyoteTime()* i a *CheckJumpMultiplier().*

  - **CheckCoyoteTime() i CheckWallJumpCoyoteTime()**: El concepte de 'CoyoteJump' fa referència a una funcionalitat dels jocs de plataforma que permet al jugador saltar un instant després d'haver abandonat el terra. Això ofereix un control més satisfactori per

l’usuari. Aquests dos mètodes gestionen aquesta funcionalitat de la següent manera:

- **CheckCoyoteTime():** Si el personatge entra a l’estat *InAirState* des del *GroundedState*, s’estableix un comptador de temps dins el qual, si el jugador prem el botó de salt, encara és capaç de saltar. Aquest mètode comprova si aquest espai de temps ha passat i, en cas de ser així, impedeix que el personatge pugui saltar disminuint el valor *amountOfJumpsLeft* de l’estat *JumpState.*
- **CheckWallJumpCoyoteTime():** Les variables *oldIsTouchingWall* i *oldIsTouchingWallBack* serveixen per comprovar si es dona la situació en la que, abans de fer les comprovacions de l’entorn, el jugador estigués tocant una paret i després de fer-les ja no. Això voldria dir que el jugador s’acaba de separar de la paret i, mitjançant el *wallJumpCoyoteTime,* li podem donar un petit espai de temps per poder saltar encara que ja no estigui tocant la paret.

9.5.5.15 LedgeClimbState (Estat sense Superestat)

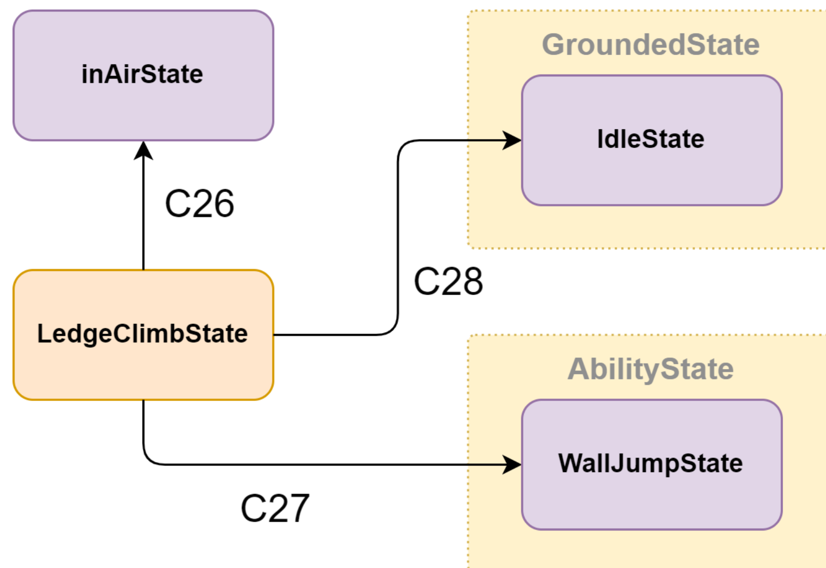


Figura 9.5.11: Diagrama de canvis d’estat des de *LedgeClimbState*

En aquest estat, el personatge s’ha trobat una cantonada i es queda penjant d’aquesta. Quan el jugador vulgui, el pot fer pujar a sobre la cantonada. Una cantonada es detecta quan el *wallCheck* està detectant una paret i el *ledgeCheck* no. Veure la **figura 9.5.11.**

- **Condicions de canvi d'estat:**

  - **C26:** Si es detecta input negatiu a l'eix vertical i el personatge encara no està pujant la cantonada => Canvi a InAirState
  - **C27:** Si es detecta l'input de saltar i el personatge encara no està pujant la cantonada => Canvi a WallJumpState
  - **C28:** Si s'ha acabat l'animació d'escalar la cantonada => Canvi a IdleState

- **Mètodes i funcionament**

  En aquest estat cal dividir el comportament del personatge en dues fases. En la primera el personatge està penjant de la cantonada sense moure's, en la segona es posa a escalar per acabar sobre aquesta cantonada. Per tal de saber en cada moment en quina de les fases es troba, s'utilitza el paràmetre *isHanging* per la primera fase i el paràmetre *isClimbing* per la segona. Així doncs, es manté la posició del personatge en una posició d'inici i, en acabar l'animació, es mou el personatge a una posició final. Aquesta ha de coincidir amb el desplaçament que ha realitzat el personatge en l'animació.

  - **Enter():**
    - S'estableix la velocitat del personatge a 0.
    - Es col·loca el personatge a la posició on l'estat anterior ha trobat la cantonada.
    - Es determina la posició exacte de la cantonada amb *DetermineCornerPosition()*.
    - S'estableixen les posicions d'inici i final d'estat pel jugador aplicant un *offset* a la posició de la cantonada calculada amb *DetermineCornerPosition()* . Aquestes posicions serveixen per poder moure al personatge a la posició final una vegada acabi l'animació d'escalar la cantonada.

  - **DetermineCornerPosition():** Aquest mètode calcula la posició de la cantonada a través del *wallCheck* i el *ledgeCheck* de *CollisionSenses.* Per entendre el funcionament d'aquest mètode, ens podem fixar en la **figura 9.5.12.**

Figura 9.5.12: Esquema de funcionament del mètode *DetermineCornerPosition*

- A partir del *wallCheck*, llença un *RayCast* en la direcció de la paret. El *RayCast* para al trobar-se la paret, a partir d'això obté la distància **xDist.**
- A partir del *ledgeCheck*, es llença un *RayCast* cap avall, amb el punt d'origen deplaçat el valor de **xDist** en direcció a la paret. El *RayCast* para al trobar-se el terra, a partir d'això s'obté la distància **yDist.**
- Amb aquestes dues distàncies s'obté la posició de la cantonada fent:
    - X = Posició X de *wallCheck* + xDist
    - Y = Posició Y de *ledgeCheck* - yDist

- Tots dos *RayCasts* detecten la paret i el terra gràcies a que aquest està dins una *Layer* determinada, la qual el *RayCast* pot distingir. Per tal d'evitar detectar la cantonada massa aviat, la distància màxima dels *RayCasts* s'ha hagut de determinar a base de prova i error.

- **LogicUpdate()**: Manté el personatge en la posició inicial fins que s'acaba l'animació.

- **Exit():** Estableix la posició del personatge a la posició final calculada a l'inici.

9.5.5.16 DeadState (Estat sense Superestat)

Estat en el que entra el jugador al morir. És l'únic estat al qual s'entra a través d'una crida de l'script *Player.* El personatge es queda quiet durant un segon en el qual s'activa l'efecte corresponent del *shader*, i després reapareix.

- **Condicions de canvi d'estat:**

    - **C29:** Quan ha passat un segon des de que s'ha entrat a l'estat => Canvi a IdleState

- **Mètodes i funcionament**

    - **Enter():**
        - Estableix la velocitat del personatge a 0.
        - Impedeix que el jugador pugui moure el personatge.
        - Crida al mètode d'*Appearance* que fa que s'iniciï l'efecte visual de mort.

    - **LogicUpdate():** Espera a que passi un segon. Quan aquest ha passat:
        - Crida a les funcions *OnSpawn()* dels components *Inventory, Stats, Appearance* i *Combat* per reiniciar les propietats d'aquests.
        - Crida la funció *Respawn()* de *Core*

9.6 Shader

Per implementar el *shader* del personatge principal hem utilitzat el *Shader Graph*, una eina de Unity per crear *shaders* de manera visual amb un entorn de treball basat en nodes, el qual ens permet també veure els resultats a temps real. En aquest apartat expliquem la creació d'aquest *shader* amb els diferents apartats i la utilització d'aquest mitjançant el Core Component Appearance.

9.6.1 Vida diegètica

El primer element de feedback en l'aparença del jugador que s'ha implementat a través del *shader* és el de mostrar la vida restant del personatge de manera diegètica. En els *sprites* del personatge, aquest té una franja de píxels d'un color carn fosc; Hem aprofitat aquest color per a implementar aquest efecte, reemplaçant-lo per un altre color segons la vida restant del personatge en cada moment creant l'aparença de que el personatge té una mena de braçalet que fa la funció que faria la típica 'barra de vida'. Hem volgut que el personatge comenci amb el braçalet verd i, a mesura que rebi mal, vagi canviant de color gradualment cap a groc i finalment vermell. Les passes a seguir per aconseguir aquest efecte han estat:

Al *shader graph:*

1. Amb el node *Sample Texture 2D LOD* mostregem la textura original, la qual és una propietat de tipus *Texture2D* que inicialitzem a que sigui l'*spritesheet* del moviment

del jugador. Aquest node retorna un vector amb 4 elements corresponents al valor dels components RGB i Alpha de la textura.

2. Per tal sobresaltar el fons utilitzem el node *One Minus,* el qual resta a 1 el valor d'entrada, en aquest cas el valor *alpha* de la textura.
3. Amb el node *Substract* eliminem el fons de la textura restant els valors RGBA amb els resultants de destacar el fons.
4. Creem una propietat de tipus Color (**BraceletColour**) que guardi el valor actual del braçalet. Utilitzem el node *Replace Color* per tal de reemplaçar el color carn de la textura original pel del braçalet.
5. Finalment, el valor resultant és enviat al *Fragment* (encarregat de calcular el valor dels colors i l'alpha).

El resultat a l'inspector és:

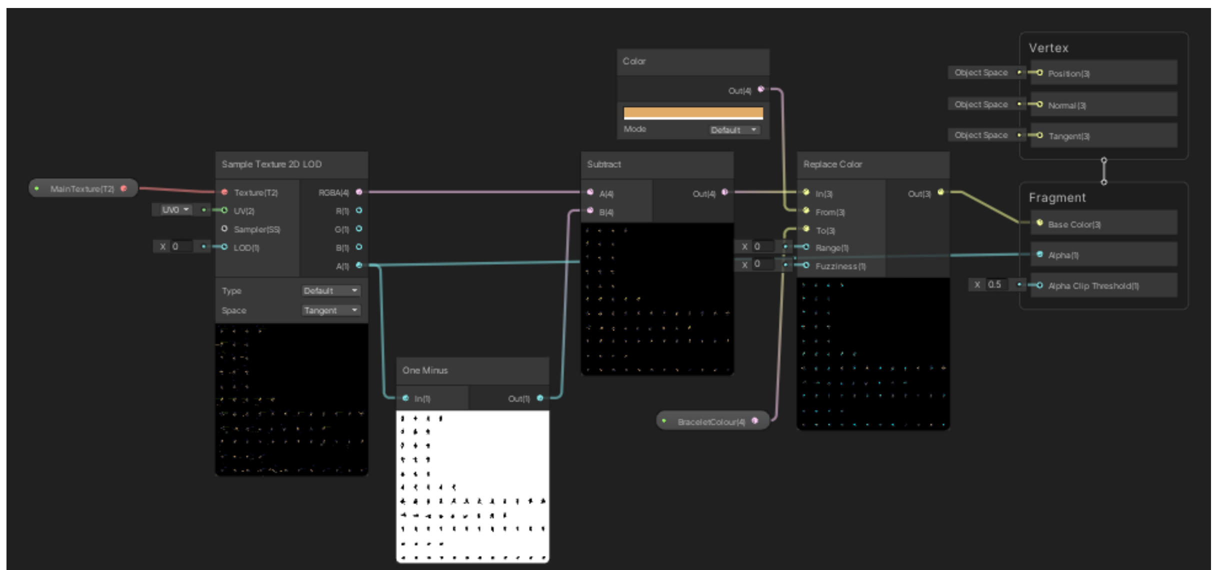


Figura 9.6.1: Nodes del Shader Graph amb l'efecte de la vida diegètica implementat

Funcionament a Player*Appearance*

- Per tal de mostrejar el color que assignem a la propietat **Bracelet Colour**, utilitzem una variable privada de tipus *Gradient*, la qual representa un gradient de colors, i la fem editable des de l'inspector. Amb la funció *Evaluate(float value)*, donat un valor entre 0 i 1 podem extreure un color d'aquest gradient. Veure la **figura 9.6.2.**

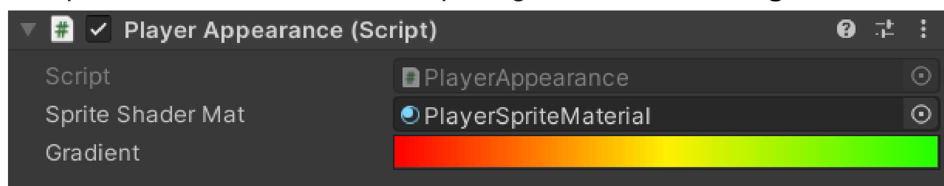


Figura 9.6.2: Gradient de mostreig a l'inspector

- Al rebre mal, el Core Component *Combat* crida a la funció *Damaged()* d'*Appearance* i li passa com a paràmetre la vida restant del jugador. En aquest mateix moment es canvia el color de **Bracelet Colour,** extraient el nou color del gradient, passant com a valor la divisió entre la vida inicial del personatge i la vida

restant actual.

El codi corresponent es mostra a la **figura 9.6.3.**

```
[SerializeField] private Gradient gradient;

protected override void Awake() {
        base.Awake
        //Inicialitzem el color a verd
        spriteShaderMat.SetColor("playerBraceletColour",
             GetColorFromGradient(1f));
}

 public override void Damaged(float currentHealth){
        spriteShaderMat.SetColor("playerBraceletColour",
      gradient.Evaluate(currentHealth/entityInitialHealth));
}
```

Figura 9.6.3: Codi que gestiona l'efecte de videa diegètica

9.6.2 Efecte de 'hit'

El que es vol aconseguir és que en el moment en que el jugador rebi mal, i durant un instant de temps curt, la seva aparença canviï de tal manera que es dibuixi tota la textura de color blanc. Els passos de la implementació són:

Al shader graph:

1. Aprofitant el resultat que ens dona el node *One Minus* en el tractament de la textura de l'efecte anterior, utilitzem un node *Invert Colors* per sobresaltar la silueta del personatge en comptes del fons. D'aquesta manera obtenim la silueta del personatge en blanc.
2. Creem una propietat tipus Bool (**isHurt**) i, mitjançant un node *Branch*, definim els valors de sortida segons si aquesta propietat pren el valor true o false.
   Si el valor és false, el node *Branch* deixarà passar els colors normals, altrament deixarà passar els colors corresponents a la silueta del personatge en blanc.

El resultat a l'inspector es mostra a la **figura 9.6.4.**

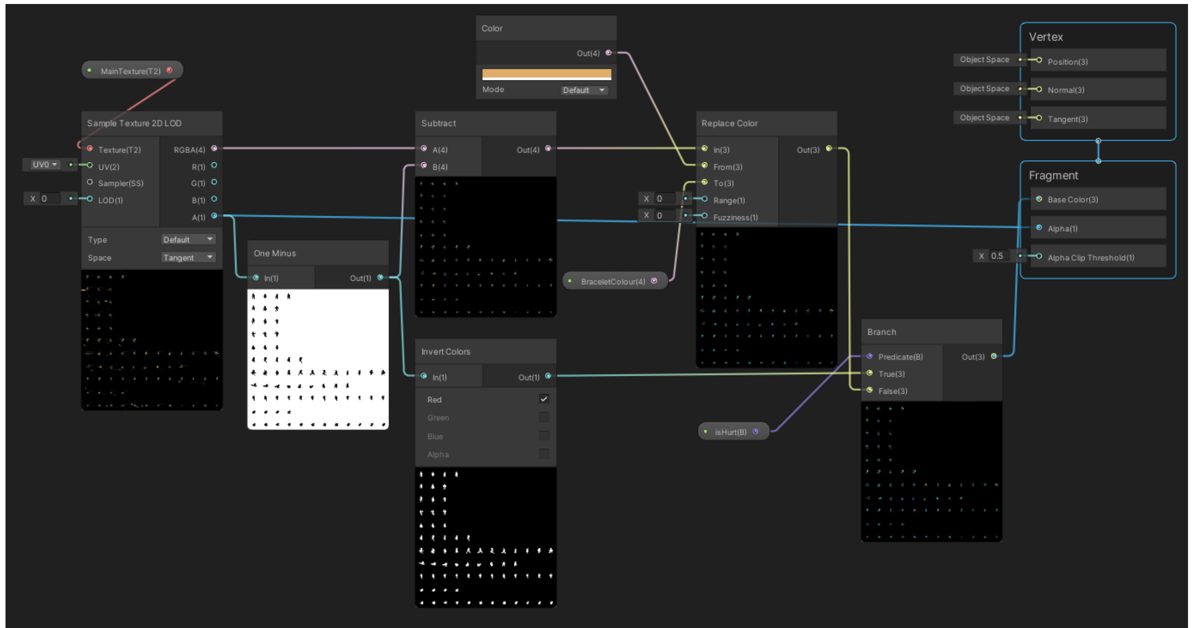


Figura 9.6.4: Nodes del Shader Graph amb l'efecte de 'hit' implementat

Funcionament a *Player Appearance*

- Implementem una co-rutina *SetDamagedAppearance()* que modifica el valor de la propietat **isHurt** a 'cert' durant un instant de temps i la torna a posar a 'fals'.
- A la mateixa funció *Damaged()* on modifiquem el color del braçalet del personatge, cridem aquesta co-rutina.

El codi corresponent es mostra a la **figura 9.6.5.**

```
public virtual void Damaged(float currentHealth){
    StartCoroutine(SetDamagedAppearance());
}

protected IEnumerator SetDamagedAppearance(){
    spriteShaderMat.SetInteger("isHurt", 1);
    yield return new WaitForSeconds(hurtAppearanceTime);
    spriteShaderMat.SetInteger("isHurt", 0);
}
```

Figura 9.6.5: Codi que gestiona l'efecte de 'hit

9.6.3 Efecte de realçar contorn al tenir el *dash* disponible

Amb aquest efecte es vol poder donar *feedback* al jugador perquè sàpiga en quins moments té la capacitat de fer dash. A l'agafar la flama, volem que el contorn del jugador canviï de color a un blau elèctric. Aquest s'ha de mantenir així fins que el jugador gasti el *dash*, moment en el qual tornarà a la normalitat. Les passes a seguir han estat les següents:

Al *shader graph*

1. Ja que el contorn del personatge està dibuixat amb un negre pur, i a l'interior aquest color no es repeteix, al reemplaçar-lo per un altre color podem modificar el contorn del jugador.
2. Similarment al mètode que hem utilitzat amb l'efecte pel canvi de color del braçalet, per aconseguir aquest altre afegim un altre node *Replace Color*. Aquest canviarà tots els píxels de color negre pur pel blau elèctric.
3. Creem una nova propietat de tipus *Bool* (**canDash**) i, mitjançant un *Branch* decidim si els colors resultants són els de la textura normal o bé amb el contorn blau. Aquesta branca es comprova abans de la que determina si el jugador s'ha de mostrar amb la silueta blanca, ja que si no ho féssim així es podria donar el cas que es mostri el jugador amb la silueta blanca i el contorn blau alhora. Veure la **figura 9.6.6.**

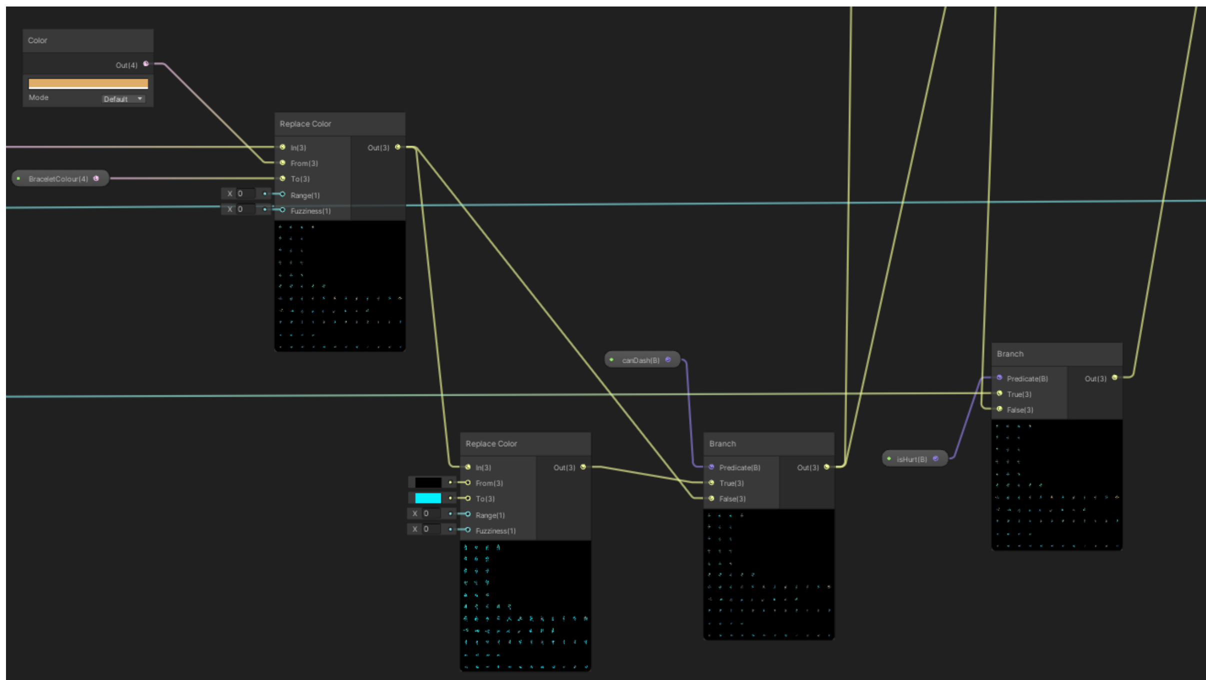


Figura 9.6.6: Nodes del Shader Graph amb l'efecte de realçar contorn implementat

Funcionament a *Player Appearance*

- A *Appearance* hem creat dues funcions públiques anomenades *OnPowerUpObtained()* i *OnPowerUpFinished()*, que posen a 'cert' i 'false' la propietat **canDash** del material.
- L'encarregat de cridar aquestes dues funcions és el Core Component *Inventory*, ja que és qui rep el missatge corresponent a 'obtenir flama' i 'perdre flama'.

El codi corresponent és:

Appearance. Veure la **figura 9.6.7.**

```
public override void OnPowerUpObtained(){
    spriteShaderMat.SetInteger("canDash", 1);
}
```

```
public override void OnPowerUpFinished(){
    spriteShaderMat.SetInteger("canDash", 0);
}
```
Figura 9.6.7: Codi que gestiona l'efecte a Appearance

Inventory. Veure la **figura 9.6.8.**

```
public void OnRetrieve(int key){
    collectibles[key] = false;
    if(key == (int)Collectibles.flame){
        Appearance.OnPowerUpFinished();
    }
}

public void activateCollectibleEvent(int key){
    switch (key){
        case (int)Collectibles.key:
            ...
        case (int)Collectibles.flame:
            Appearance.OnPowerUpObtained();
            Movement.CanDash = true;
            break;
    }
}
```
Figura 9.6.8: Codi que gestiona l'efecte a Inventory

9.6.4 Efectes de parpalleig

Expliquem en aquest apartat els dos efectes finals implementats en el *shader*, ja que comparteixen la major part de la implementació al *shader graph* pel funcionament.
El primer és l'efecte que s'activa en el moment que el jugador té poca vida restant. Voliem crear un efecte en el qual el l'sprite del personatge parpallegés lentament en vermell, donant la sensació de perill pel jugador.
El segon es tracta de l'efecte que s'activa en el moment que el jugador mor. En aquest cas també es vol obtenir un parapalleig, però de color totalment blanc i més ràpid que l'anterior.
Aquests són els passos a seguir per la implementació:

Al *shader graph:* part comuna per aconseguir el parpalleig

1. Per tal d'aconseguir un valor entre 1 i -1, utilitzem el valor del node *Time* com a input del node *Sin*. Aquest calcularà la funció de sin(x), essent x el valor de temps. Per tal d'assegurar-nos que el valor resultant està entre 1 i -1, utilitzem el node *Remap*.
2. Per tal de controlar la velocitat del parpalleig, afegim un node *Multiply* perquè multipliqui el valor de temps abans de passar-lo com a input a la funció de sinus.

3. Per tal de controlar el funcionament dels dos efectes, creem una propietat de tipus *Bool* per a cada una: **lowHealth** i **isDead**.
4. Utilitzem un node *branch* al principi del fluxe que controli si es vol iniciar el parpalleig o no. Això ho fem per evitar càlculs innecessaris en els moments que el personatge no té cap d'aquests efectes activats.
5. Amb un altre node *branch* controlem la velocitat del parpalleig (el paràmetre que multiplica el valor del node *time*). En el cas que el parpalleig estigui activat perquè el jugador té poca vida, el valor del temps es multiplica per 5, si és perquè el jugador s'ha mort, es multiplica per 40.
6. Finalment, utilitzem el node *Blend* per aprofitar el valor que fluctua entre -1 i 1. Aquest node rep dos valors RGB, un siguent la base i l'altre la barreja. Segons un tercer valor *alpha,* aquest node barreja els dos valors i retorna el resultat. Així doncs, si introduïm el valor fluctuant com a *alpha,* la intensitat d'aquesta barreja anirà variant de -1 a 1, obtenint l'efecte de parpalleig desitjat
   Segons el mode en que utilitzem aquest node, el resultat és diferent
   a. Per l'efecte de poca vida utilitzem el mode *Overlay* amb una barreja de valor (1, 0, 0) per aconseguir el color vermell. Aquest mode actua com una capa que es barreja amb els colors de la base.
   b. Per l'efecte de mort utilitzem el mode *Overwrite* amb una barreja de valor (1, 1, 1) per obtenir el color blanc. Aquest mode pinte per sobre la base els colors de la barreja, d'aquesta manera obtenim un parpelleig amb un blanc més sòlid.
7. Mitjançant *branches* controlem l'output final de manera que l'efecte de parpalleig al morir tingui preferència sobre l'efecte de 'hit'. Veure les **figures 9.6.9, 9.6.10 i 9.6.11.**

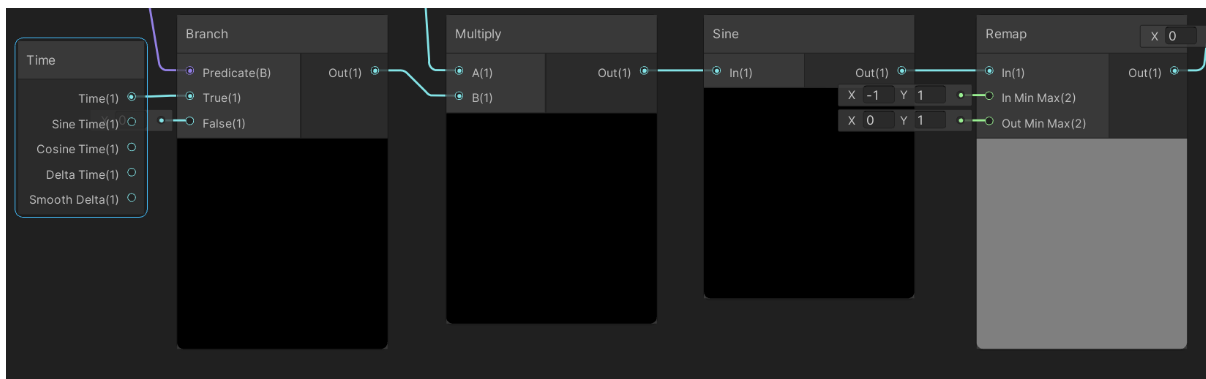


Figura 9.6.9: Nodes utilitzats per l'efecte de parpalleig

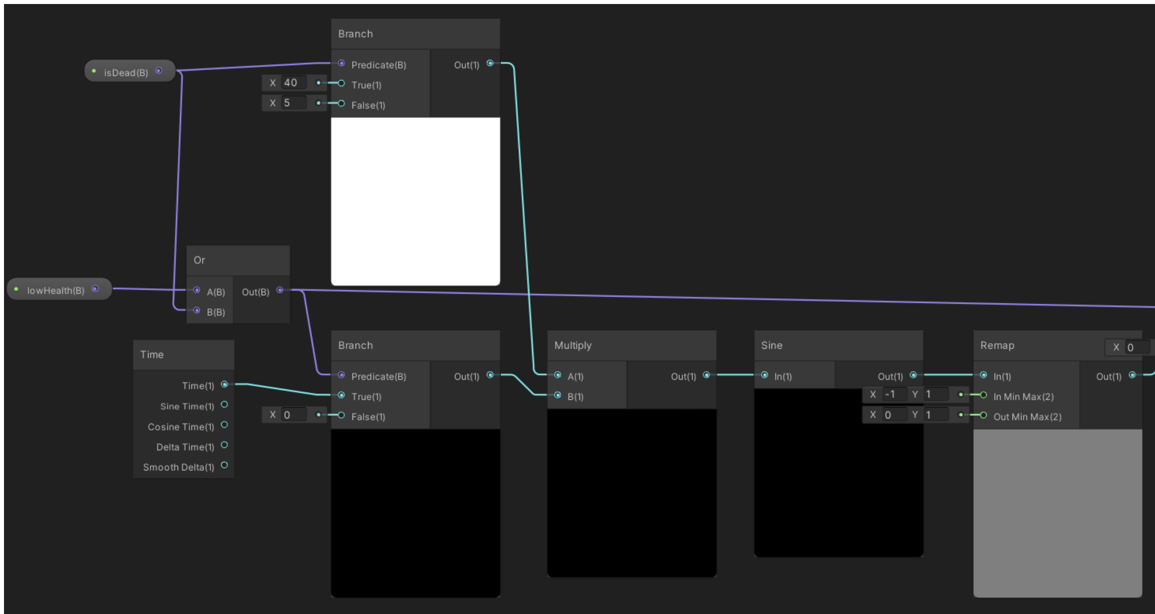


Figura 9.6.10 : Control de la velocitat del parpalleig

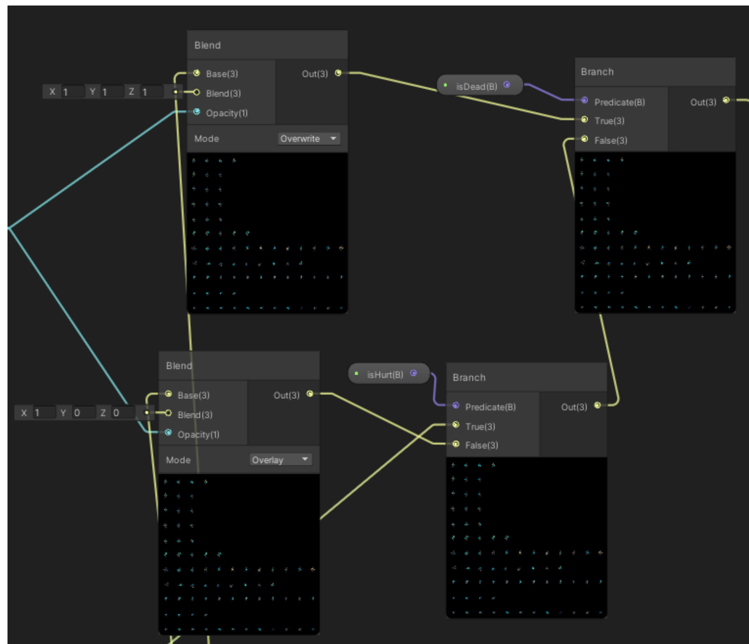


Figura 9.6.11: Nodes *blend* i *branch* pel control de l'output final

Funcionament a *Player Appearance*

- **Efecte de poca vida restant:**
    - A la funció *Damaged()* on també modifiquem el color del braçalet del personatge i executem la co-rutina per l'efecte de 'hit', comprovem si la vida restant del personatge és inferior al 20%. Si es dona el cas, es posa a 'true' el valor de la propietat del material **lowHealth**, iniciant el parpalleig en vermell. Veure la **figura 9.6.12.**

```
public override void Damaged(float currentHealth){
```

```
    ...
    if(currentHealth/entityInitialHealth < 0.2f){
        spriteShaderMat.SetInteger("lowHealth", 1);
    }
}
```

Figura 9.6.12: Codi que gestiona l'efecte de poca vida a Player Appearance

- **Efecte de mort:**
    - Afegim una funció *onDead()* que activa l'efecte establint la propietat **isDead** del material a 'cert'.
    - En el moment en que el personatge entra en l'estat *DeadState* (a la funció *Enter()*), es crida a la funció *onDead()* d'*Appearance*:

      Player Dead State. Veure la **figura 9.6.13.**

```
public override void Enter(){
    base.Enter();
    ...
    Appearance.OnDead();
}
```

Figura 9.6.13: Codi que gestiona l'efecte de mort a DeadState

PlayerAppearance. Veure la **figura 9.6.14.**

```
public override void OnDead(){
    spriteShaderMat.SetInteger("isDead", 1);
}
```

Figura 9.6.14: Codi que gestiona l'efecte de mort a Player Appearance

Tots els resultats in-game dels efectes implementats es troben a l'apartat 10.3 Resultats dels efectes del shader.

9.7 Checkpoints

En aquest apartat s'entrarà més en detall amb la implementació dels Checkpoints. La lògica de l'objecte la defineix l'script Checkpoint. Aquest conté 3 variables:

- Animator
- BoxCollider2D
- GameController

Disposa d'un sol mètode rellevant, el detector de col·lisions per trigger *OnTriggerEnter2D*. Aquí, si el jugador entra en contacte amb el BoxCollider2D del Checkpoint, es realitza el següent seguint aquest ordre:

1. Es guarda la posició del Checkpoint en un Vector3
2. Es crida el mètode UpdateCheckpoint del GameController, passant-li la posició

3. S'anima el Checkpoint
4. Desactivem el BoxCollider2D per a què ja no detecti el jugador

   Veure la **figura 9.7.1.**

```
 private void OnTriggerEnter2D(Collider2D collision)
    {
        if(collision.CompareTag("Player"))
        {
            Vector3 newPos = new Vector3(this.gameObject.transform.position.x,
this.gameObject.transform.position.y, collision.transform.position.z);
            gameController.UpdateCheckpoint(newPos);
            anim.SetBool("Activation", true);
            collider.enabled = false;
        }
    }
```

Figura 9.7.1: Mètode OnTriggerEnter2D de l'script Checkpoint

Finalment, a aquest mètode que es crida al GameController, tant sols actualitza una variable de tipus Vector3 anomenada *posCheckpointActual* que emmagatzema la posició de l'últim Checkpoint per el qual el jugador ha passat. Veure la **figura 9.7.2.**

```
public void UpdateCheckpoint(Vector3 newPosition)
    {
        posCheckpointActual = newPosition;
    }
```

Figura 9.7.2: Mètode UpdateCheckpoint de l'script GameController

Ara, quan el jugador es mori, ja podem anar a buscar aquesta posició guardada al GameController i així tornar a fer reaparéixer el jugador.

9.8 Màquines d'estats pels enemics

En aquest apartat s'explicaran els diferents estats que poden tenir els enemics. Veurem que hi ha estats que que comparteixen varis enemics i n'hi ha d'altres que només els tenen un tipus d'aquests.

9.8.1 Possibles estats dels enemics

Aquests són els tipus d'estats que poden tenir els enemics:

9.8.1.1 Idle State

Estat “per defecte de l’enemic”. En aquest esta l’enemic està quiet i no realitza cap acció. Aquest estat té un script anomenat IdleState, que hereda de State. Aquest script conté les següents variables de control:

- **bool flipAfterIdle**: controla si l’enemic es pot girar després de sortir de l’estat Idle
- **bool isIdleTimeOver**: controla si s’ha acabat el temps de idle (idleTime)
- **bool isPlayerInMinAgroRange**: controla si el jugador està dins del rang de visió de l’enemic
- **float idleTime**: temps en què l’enemic està en l’estat de Idle

Com s’arriba aquest estat:
- Des de l’estat de Move, si l’enemic ha detectat una paret o una vora

Com es surt d’aquest estat:
- Si s’ha acabat el temps de Idle
- Si l’enemic ha detectat el jugador

9.8.1.2 Move State

Estat on l’enemic es mou horitzontalment cap a una direcció determinada. L’script d’aquest estat es diu MoveState. Té les següents variables de control:

- **bool isDetectingWall**: controla si l’enemic està detectant una paret (si té una paret a davant)
- **bool isDetectingLedge**: controla si l’enemic està detectant una vora (si en té una a davant)
- **bool isPlayerInMinAgroRange**

Com s’arriba aquest estat:
- Des de l’estat Idle, si s’ha acabat el temps de Idle
- Des de l’estat LookForPlayer, si l’enemic no troba el jugador

Com es surt d’aquest estat:
- Si l’enemic ha detectat una paret o una vora
- Si el jugador esta dins del rang de visió de l’enemic

9.8.1.3 Attack State

Estat on l’enemic realitza un atac contra el jugador. L’script d’aquest estat es diu AttackState. Té les següents variables de control:
- **Tranform attackPosition**: és la posició des d’on l’enemic realitzarà l’atac

- **bool isAnimationFinished**: controla si l'animació de l'atac ha acabat

- **bool isPlayerInMinAgroRange:** controla si el Player està a prou distància per començar l'atac.

Els enemics però, poden realitzar diferents tipus d'atacs, i no tots fan el mateix. És per això que necessitem subestats d'aquest AttackState que herederan d'ell:

9.8.1.3.1 MeleeAttack State

Estat on l'enemic realitza un atac a curta distància. A l'script d'aquest estat, es refà el mètode TriggerAttack(), ja que ha de ser un tipus d'atac diferenciat de la resta d'atacs. Veure la **figura 9.8.1.**

```
public override void TriggerAttack()
    {
        base.TriggerAttack();

        Collider2D[] detectedObjects =
Physics2D.OverlapCircleAll(attackPosition.position, stateData.attackRadius,
stateData.whatIsPlayer);

        foreach (Collider2D collider in detectedObjects)
        {
            IDamageable damageable = collider.GetComponent<IDamageable>();

            if(damageable != null)
            {
                damageable.Damage(stateData.attackDamage);
            }

            IKnockbackable knockbackable =
collider.GetComponent<IKnockbackable>();

            if(knockbackable != null{
                knockbackable.Knockback(stateData.knockbackAngle,
stateData.knockbackStrength, Movement.FacingDirection);
            }
        }
    }
```

Figura 9.8.1: Mètode TriggerAttack de l'script MeleeAttackState

Aquí es detecten els elements que colisionen amb l'atac de l'enemic i els hi aplica mal i els fan retrocedir (si tenen les interfícies IDamagable i IKnockbackable).

9.8.1.3.2 RangeAttackState

Estat on l'enemic realitza un atac a distància, instanciant una fletxa. Tal i com ho fa el MeleeAttackState, es sobreescriu el mètode TriggerAttack(). Veure la **figura 9.8.2.**

```
public override void TriggerAttack()
    {
        base.TriggerAttack();

        projectile = GameObject.Instantiate(stateData.projectile,
attackPosition.position, attackPosition.rotation);
        projectileScript = projectile.GetComponent<Projectile>();
        projectileScript.FireProjectile(stateData.projectileSpeed,
stateData.projectileTravelDistance, stateData.projectileDamage);
    }
```

Figura 9.8.2: Mètode TriggerAttack de l'script RangeAttackState

Per aquest motiu, necessitem que l'script RangeAttackState que té aquest mètode tinghui una variable per a guardar la fletxa (anomenada projectile).

9.8.1.3.3 ElectricBombAttack State

Estat molt semblant al RangeAttackState. En aquest cas però, l'enemic llança una bomba que al entrar en contacte amb el jugador explota. Si cau i entra en contacte amb el terra, s'activa i finalment també explota. Veure la **figura 9.8.3.**

```
public override void TriggerAttack()
    {
        base.TriggerAttack();

        electricBomb = GameObject.Instantiate(stateData.electricBomb,
attackPosition.position, attackPosition.rotation);
        electricBombScript = electricBomb.GetComponent<ElectricBomb>();
        electricBombScript.FireElectricBomb(stateData.electricBombSpeed,
stateData.electricBombTravelDistance,
         stateData.electricBombDamage, stateData.height,
stateData.PosicioPlayer.transform);
    }
```

Figura 9.8.3: Mètode TriggerAttack de l'script ElectricBombAttackState

9.8.1.3.4 LaserBeamAttackState

Estat similar als dos últims explicats. Aquí l'enemic llança un atac làser horitzontlment cap a la direcció del jugador. El làser triga en carregar-se i un cop carregat, es llança de cop. Veure la **figura 9.8.4.**

```
public override void TriggerAttack()
    {
        base.TriggerAttack();

        laserBeam = GameObject.Instantiate(stateData.laserBeam,
attackPosition.position, attackPosition.rotation);
        laserBeam.GetComponent<LaserBeam>().attackPosition = attackPosition;
    }
```

Figura 9.8.4: Mètode TriggerAttack de l'script LaserBeamAttackState

9.8.1.4 Charge State

Estat on l'enemic es mou ràpidament cap al jugador amb l'intenció d'atacar-lo (si entrem a l'AttackState). L'script d'aquest estat s'anomena ChargeState i té les següents variables de control:

- **bool isPlayerInMinAgroRange**
- **bool isDetectingLedge**
- **bool isDetectingWall**
- **bool isChargedTimeOver**: controla si el temps de càrrega ha acabat.
- **bool performCloseRangeAction**: controla si pot realitzar una acció de curta distància (com per exemple un atac a melee).

9.8.1.5 Dodge State

Estat on l'enemic fa un salt cap enrere per a esquivar al jugador. L'script d'aquest estat s'anomena DodgeState i conté les següents variables de control:

- **bool performCloseRangeAction**
- **bool isPlayerInMaxAgroRange**
- **bool isGrounded:** comprova si l'enemic està en contacte amb el terra o amb una plataforma

- **bool isDodgeOver:** indica si l'enemic a acabat el dodge

9.8.1.6 LookForPlayer State

Estat on l'enemic busca al jugador.Durant aquest estat, l'enemic realitzarà un cert nombre de girs per a intentar veure si el jugador està davant seu. L'script d'aquest estat s'anomena LookForPlayerState i disposa d'aquestes variables de control:

- **bool turnImmediately**: indica si l'enemic s'ha de girar immediatament
- **bool isAllTurnsDone**: indica si l'enemic s'ha girat totes les vegades que ho ha de fer
- **bool isAllTurnsTimeDone**: indica si han complert tots els temps de gir
- **float lastTurnTime**: guarda l'instant de temps en què s'ha fet l'últim gir
- **int amountOfTurnsDone**: guarda el nombre de girs que l'enemic ha fet des que ha entrat a aquest estat
- **bool isPlayerInMinAgroRange**

9.8.1.7 PlayerDetected State

Estat on l'enemic ha detectat al jugador, i es prepara per a realitzar una acció. El seu script es diu PlayerDetectedState i té aquestes variables de control:

- **bool isPlayerInMaxAgroRange**: indica si el jugador està a una distància respecte l'enemic suficient com per a què aquest pugui realitzar una acció a llarga distància (p.e un Charge).
- **bool performLongRangeAction**: indica si l'enemic pot realitzar una acció de llarga distància(com per exemple un atac làser)
- **bool performCloseRangeAction**
- **bool isDetectingLedge**
- **bool isPlayerInMinAgroRange**

9.8.1.8 Dead State

Estat on l'enemic està mort (la seva vida ha arribat a 0). Des d'aquest estat es podrien plantejar vàries situacions, com invocar enemics nous, reviure, explotar, etc, depenent de cada enemic. L' script es diu DeadState, i no requereix de cap variable de control. Un cop entrem a aquest estat, és necessari que la velocitat de l'enemic sigui 0, evitant que es mogui. Veure la **figura 9.8.5.**

```
public override void Enter()
    {
        base.Enter();
        entity.atsm.deadState = this;
        Movement?.SetVelocityX(0f);
    }
```

Figura 9.8.5: Mètode Enter de l'script DeadState

9.8.2 Herència d'estats

Cada enemic té les seves peculiaritats, i és per això que no podem fer que tots els enemics tinguin aquests estats. Així doncs, cada enemic diferent tindrà uns estats propis que heretaran d'aquests estats. Per exemple, imaginem un enemic anomenat E1. Els estats possibles de E1 serien E1_IdleState, E1_MoveState, E1_RangeAttackState, etc, que heretarien de IdleState, MoveState i RangeAttackState respectivament.

També necessitarem per a cada enemic un script que heredarà de Entity. Això es fa ja que els enemics poden tenir diferent nombre i diferents estats. Aquests scripts contindran els estats respectius dels enemics i els scriptable objects que contenen les dades necessàries pels estats. Veure la **figura 9.8.6.**

```
public override void Awake()
{
  base.Awake();
  moveState = new E1_MoveState(this, stateMachine, "move", moveStateData,
this);
  idleState = new E1_IdleState(this, stateMachine, "idle", idleStateData,
this);
  playerDetectedState = new E1_PlayerDetectedState(this, stateMachine,
"playerDetected", playerDetectedStateData, this);
  chargeState = new E1_ChargeState(this, stateMachine, "charge",
chargeStateData, this);
  lookForPlayerState = new E1_LookForPlayerState(this, stateMachine,
"lookForPlayer", lookForPlayerStateData, this);
  meleeAttackState = new E1_MeleeAttackState(this, stateMachine,
"meleeAttack", meleeAttackPosition, meleeAttackStateData, this);
  deadState = new E1_DeadState(this, stateMachine, "dead", deadStateData,
this);
}
```

Figura 9.8.6: Mètode Awake de l'script Enemy1

L'últim a tenir en compte és que necessitem un lloc on guardar valors de dades per a diferents enemics i per a diferents estats. Per exemple, cal saber quina és la distància mínima entre el jugador i l'enemic per a què l'enemic el detecti, la vida màxima de l'enemic, el mal de l'atac a melee, etc. Tot aquests valor numèrics els hem de poguer fer servir d'algun lloc per a fer els càlculs necessaris dins dels estats. És per això que hem fet servir ScriptableObjects.

En primer lloc, necessitem un ScriptableObject per a guardar les dades principals per a l'enemic. Ni haurà un per a cada enemic. Per exemple, per a l'enemic 2 aquest objecte s'anomenaria E2_Base Data.

Aquests objectes contenen la següent informació (editable a l'inspector) de l'enemic. Veure la **figura 9.8.7.**

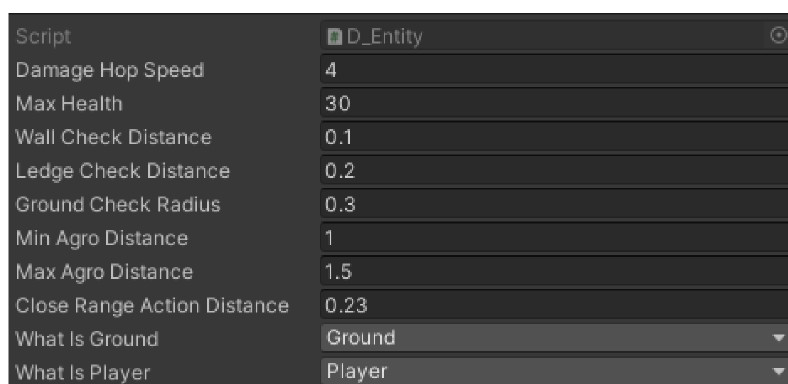


Figura 9.8.7 : ScriptableObject E2_Base Data

Tal i com hem dit, també hem de poder canviar el valor de les variables dels estats de l'enemic per a cada enemic diferent. És per això que necessitem un ScriptableObject per a cada estat. Així doncs, cada estat de cada enemic tindrà un ScriptableObject que contindrà el valor de variables que es tenen en compte a l'estat en particular. Veure la **figura 9.8.8 i 9.8.9.**

Figura 9.8.8 : ScriptableObject E3_DodgeState Data

Figura 9.8.9 : ScriptableObject E3_MeleeAttackState Data

9.8.3 Tipus d'enemics

Anem a veure quins estats tenen els diferents enemics que hem dissenyat:

Enemic BallAndChain

- E1_IdleState
- E1_MoveState
- E1_PlayerDetectedState
- E1_LookForPlayerState
- E1_ChargeState
- E1_MeleeAttackState
- E1_DeadState

Veure la **figura 9.8.10.**

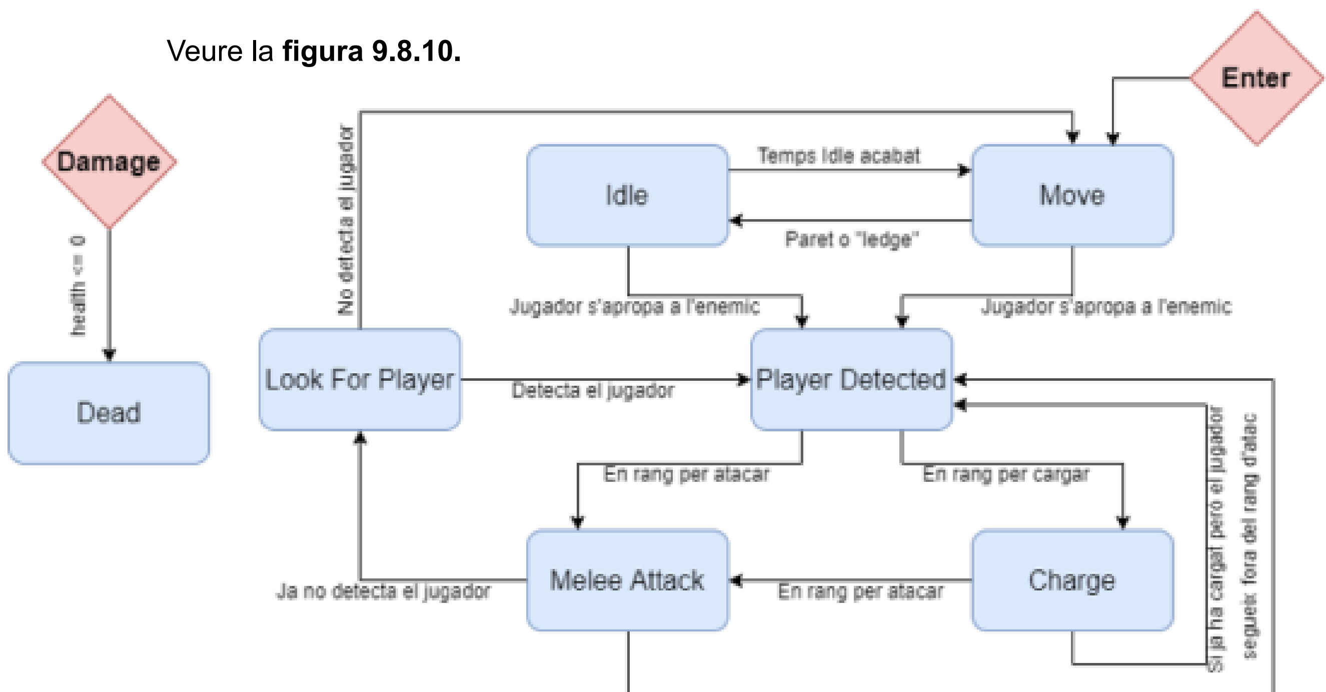


Figura 9.8.10 : Diagrama d'estats de l'enemic *BallAndChain*

Enemic Archer

- E2_IdleState
- E2_MoveState
- E2_PlayerDetectedState
- E2_LookForPlayerState
- E2_MeleeAttackState
- E2_DodgeState
- E2_RangeAttackState
- E2_DeadState

Veure la **figura 9.8.11.**

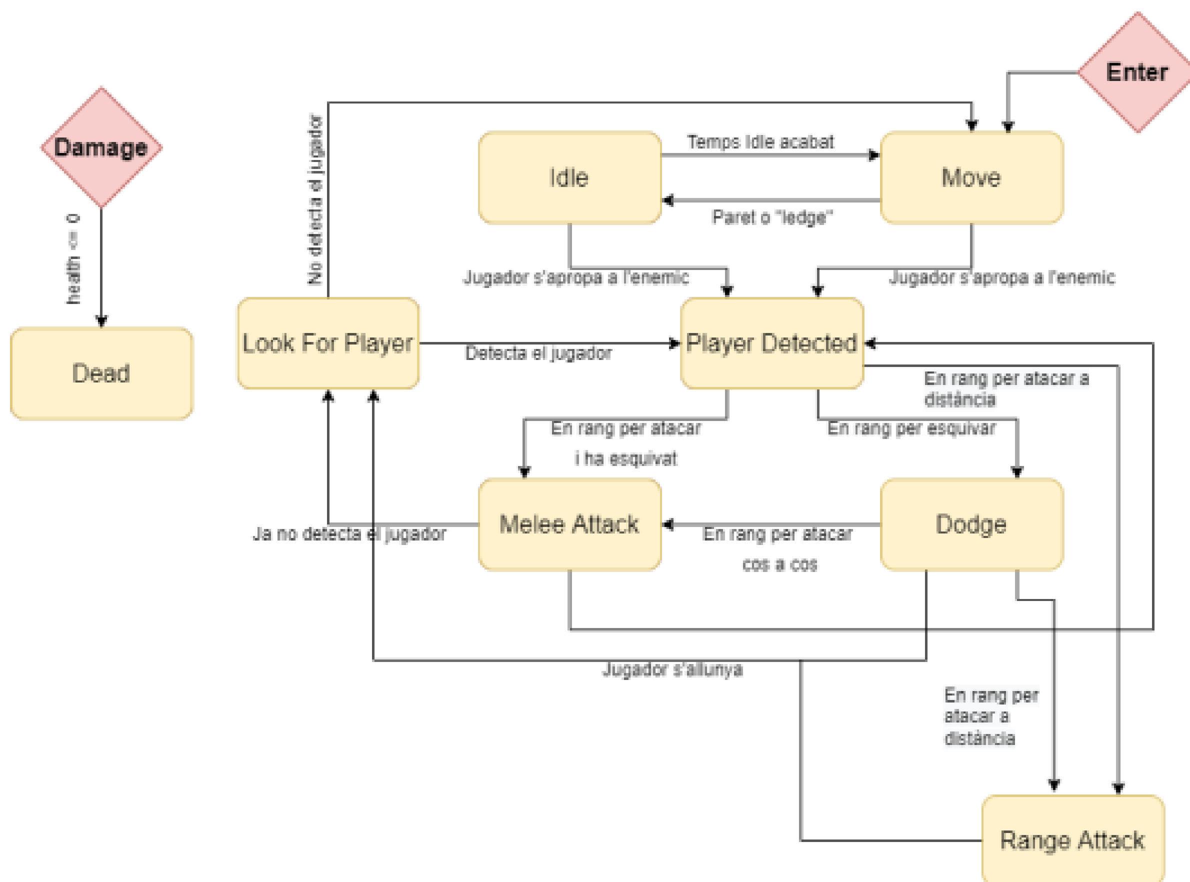


Figura 9.8.11 : Diagrama d'estats de l'enemic *Archer*

Enemics BombThrower / LaserBeamThrower

- E3_IdleState / E4_IdleState
- E3_MoveState / E4_MoveState
- E3_PlayerDetectedState / E4_PlayerDetectedState
- E3_LookForPlayerState / E4_LookForPlayerState
- E3_MeleeAttackState / E4_MeleeAttackState
- E3_DodgeState / E4_DodgeState
- E3_BombThrowAttackState / E4_LaserBeamAttack
- E3_DeadState / E4_DeadState

(el diagrames són iguals que els de l'arquer, però canviant l'estat de RangeAttack pel de BombThrowAttack ó LaserBeamAttack)

Els resultats de la implementació dels enemics i les seves màquines d'estats es poden veure a l'apartat de resultats de les màquines d'estats dels enemics.

9.9 Lluita amb el Boss

En aquest apartat explicarem tota la implementació relacionada amb la lluita contra el Boss.

9.9.1 Boss

9.9.1.1 Barra de vida

L'script que fa l'efecte de la barra de vida es pot veure a la **figura 9.9.1.**

```
public void UpdateHealthUI(){
    float fillF = frontHealthBar.fillAmount;
    float hFraction = health / maxHealth;

    if (fillB > hFraction){
        frontHealthBar.fillAmount = hFraction;
        lerpTimer += Time.deltaTime;
        float percentComplete = lerpTimer / chipSpeed;

        backHealthBar.fillAmount = Mathf.Lerp(fillB, hFraction,
percentComplete);
    }
}
```

Figura 9.9.1 : Mètode UpdateHealthUI de l'script BossHealth

9.9.1.2 BossBehaviour

Està més enfocat en gestionar l'estat en què es troba l'enemic, controlar la vida, el moviment, l'orientació respecte el jugador, etc. Cal destacar que durant la lluita hi haurà fases o moments en què el Boss serà immune als atacs del jugador. És per això que en aquest script tenim un mètode per a fer-lo immune i un altre per a que ja no ho sigui. Llavors, tenim un mètode per a fer-li mal al Boss (restar-li vida). Li farem mal, però, sempre i quan no sigui immune. També tenim un mètode anomenat LookAtPlayer() que rota el Boss 180 graus per a mirar sempre de cares al jugador. Veure la **figura 9.9.2.**

```
public void Damage(float damageAmount){
    if(!isImmune){
        currentHealth -= damageAmount;
        bossUIManager.GetComponent<BossHealth>().TakeDamage(damageAmount);

        if (currentHealth <= 0f){
            isDead = true;
        }
    }
}
```

Figura 9.9.2 : Mètode Damage de l'script BossBehaviour

9.9.1.3 BossCombatController

Aquest script està més orientat a la lluita del Boss. Aquí hi haurà els mètodes que realitzaran els seus atacs. Com a atacs del Boss tenim els següents:

- **Atac a melee**: realitza un cop en direcció vertical en baixada amb el braç cap al terra.
- **Atac a distància**: realitza un atac d'energia en àrea a una distància considerable
- **Atac especial**: carrega molta energia per a realitzar un atac poderós en àrea.
- **Llançament de boomerang**: Se li desprenen els braços i els llança cap a l'enemic. Aquests tornen cap a ell com si fossin un boomerang.
- **Invocar enemic**: Crea un portal del qual s'invocarà un enemic
- **Pluja de meteorits**: Fa que caigui meteorits del cel durant un cert període de temps.

9.9.2 Gestió de lluita per fases

9.9.2.1 Implementació de BossBattlePhaseManager

Com a mètodes en té dos per a començar la lluita i dos per cada una de les tres fases (un mètode per a començar la fase i un per acabar-la). Veure la **figura 9.9.3.**

```
public void StartPhase1(){
    startBattle.SetActive(false);
    phase1.SetActive(true);
}

public void EndPhase1(){
    phase1.SetActive(false);
    endPhase1.SetActive(true);
}
```

Figura 9.9.3 : Mètodes StartPhase1 i EndPhase1 de l'script BossBattlePhaseManager

9.9.2.2 Implementació de l'inici de la lluita

A l'iniciar la lluita, comenença una cinemàtica per aturar el jugador, llavors el Boss es desperta (animació Wake), es posa agresiu (animació Buff), s'activa la barra de vida del Boss, s'atura la cinemàtica i finalment es passa a la fase 1.

9.9.2.3 Implementació de la fase 1

Al començar la fase 1, s'activa la pluja de meterorits. Mentre cauen meterorits el Boss és immune i el jugador tant sols pot esquivar-los. Llavors, cada cert període es desactiva la pluja de meteorits, deixant el Boss vulnerable.

Per activar la pluja de meteorits es crida un mètode del GameCombatController que ja hem vist. Aquest s'anomena ActivateMeteorRain(). Un cop li hem baixat la vida del Boss a la meitat es passa a la fase 2.

9.9.2.4 Implementació de la fase 2

Un cop passada la fase 1, es fa una transició entre aquesta i la fase 2, on el Boss fa l'animació de Buff. Llavors, comença la fase 2. En aquesta, el Boss és immune i el jugador haurà de derrotar un cert nombre d'enemics que invocarà el Boss a través de portals cada cert temps (un període). Un cop derrotats tots els enemics, es passarà a la fase 3.

Per a què apareguin els enemics, s'instancia un portal que instancia un enemic aleatori (un dels diferents enemics que ens hem trobat durant la partida). Aquests portals s'instancien a una posició aleatòria entre dues posicions ja prefixades. Aquestes les hem anomenat SpawnRightLimit i SpawnLeftLimit. Conformen un rectancle imaginari. La posició del portal invocat estarà dins d'aquest rectangle. Veure les **figura 9.9.4 i 9.9.5.**

Figura 9.9.4 : Àrea d'invocació de portals

```
void Update(){
    if(enemiesSpawned >= numEnemiesToWin  && !AreEnemiesAlive()) {
        sBossBattlePhaseManager.StartPhase3();
    }

    if (Time.time > nextActionTime){
        nextActionTime = Time.time + timeBetweenSpawns;

        if(enemiesSpawned < numEnemiesToWin){
            //Invocar un enemic a una posicio
```

```
            float XSpawnPosition = Random.Range(leftLimitX, rightLimitX);
            float YSpawnPosition = Random.Range(downLimitY, upLimitY);

            Vector2 spawnPosition = new Vector2(XSpawnPosition, YSpawnPosition);
            Instantiate(portal, spawnPosition, Quaternion.identity);
            enemiesSpawned ++;

            Debug.Log("Enemies spawned:" + enemiesSpawned);
        }
    }
}
```

Figura 9.9.5 : Mètode Update de l'script Phase2

9.9.2.5 Implementació de la fase 3

Un cop passada la fase 2, entrem a la 3. El primer que passa és que el Boss invoca a 2 enemics. Llavors ja comença la lluita real, ja que ell realitzarà atacs contra el jugador. Per a determinar quin dels atacs efectuarà, es basarà en la distància entre ell i el jugador. Tenim 3 rangs de distàncies diferents, i realitzarà uns atacs o uns altres depenent del rang en què es troba el jugador. Per a cada rang, es poden realitzar varis atacs diferents (cada atac té un percentatge de probabilitat de que es faci, per exemple és més probable que a curta distància es faci un atac a melee que un atac especial).

- **Curta distància**
    - Atac a melee (50%)
    - Pluja de meteorits (30%)
    - Atac especial (20%)

- **Distància mitjana**
    - Atac a distància (50%)
    - Atac especial (50%)

- **Llarga distància**
    - Atac de boomerang (30%)
    - Moure's cap al jugador (70%)

Hi ha una variable booleana que controla si el Boss pot realitzar un atac o no. Quan realitza un atac, la variable es posa a false. Al finalitzar l'atac es posa a true. El motiu d'aquesta variable és que no ataqui per cada cicle de l'Update(), sinó que el seu comportament tingui sentit i pugui atac quan realment podria fer-ho. Per finalitzar, s'explicarà l'atac de boomerang. Veure la **figura 9.9.6.**

Atac de boomerang

```
public void BoomerangThrow(){
    bossBehaviour.SetImmuneWithoutAnimation();
```

```
    bossBehaviour.SetFlip(false);

    anim.SetBool("boomerang", true);

    Vector3 posBoss = aliveGO.transform.position;
    Instantiate(boomerangPrefab, posBoss, aliveGO.transform.rotation);
}
```

Figura 9.9.6 : Mètode BoomerangThrow de l'script BossCombatController

Durant l'atac de boomerang, el Boss és immune i no es pot girar per a mirar al jugador. Llavors, es realitza l'animació del boomerang, a la vegada que s'instancia un objecte de boomerang a la posició del Boss. Aquest objecte es desplaçarà una certa distància cap a la direcció que mira el Boss i llavors tornarà cap aquest, fent un efecte de boomerang. Si el jugador entre en contacte amb l'objecte, rep mal. Veure la **figura 9.9.7.**

```
void Update()
{
    if (facingDirection == -1){
        posFinal = new Vector2(posInicial.x - travelDistance, posInicial.y);
    }
    else{
        posFinal = new Vector2(posInicial.x + travelDistance, posInicial.y);
    }

    float distance = Vector2.Distance(transform.position, posFinal);

    if (distance <= 0.1f && !firstBoomerangStoped){
        firstBoomerangStoped = true;

        if (facingDirection == 1) facingDirection = -1;
        else facingDirection = 1;

        velocityWorkspace.Set(facingDirection * travelVelocity, rb.velocity.y);
        rb.velocity = velocityWorkspace;
    }

    if (firstBoomerangStoped){
        distance = Vector2.Distance(transform.position, posInicial);

        if (distance <= 0.01f && !secondBoomerangStoped){
            EndBoomerangAttack();
        }
    }
}
```

Figura 9.9.7 : Mètode Update de l'script BoomerangBehaviour

Al derrotar el Boss, es fa una animació de Death i es finalitza la lluita, obrint la porta de l'esquerra i trient la barra de vida del Boss.

9.10 Efecte de parallax

L’script que fa l’efecte de paral·lax es pot veure a la **figura 9.10.1.**

```
public class ParallaxMovement : MonoBehaviour
{
    private float lenght;
    private float StartPos;
    public GameObject Camera;
    public float speed;
    void Start()
    {
        StartPos = transform.position.x;
        lenght = GetComponent<SpriteRenderer>().bounds.size.x;
    }

    private void FixedUpdate()
    {
        float temp = (Camera.transform.position.x * (1 - speed));
        float distance = (Camera.transform.position.x * speed);

        transform.position = new Vector3(StartPos + distance,
transform.position.y, transform.position.z);

        if (temp > StartPos + lenght)
        {
            StartPos += lenght;
        }
        else if (temp < StartPos - lenght)
        {
            StartPos -= lenght;
        }
    }
}
```

Figura 9.10.1 : Script ParallaxMovement

Al principi (mètode Start), guardem la posició inicial a l’eix x de l’sprite de la capa. També en guardem la llargada amb “GetComponent<SpriteRenderer>().bounds.size.x”.

Llavors, en el FixedUpdate (i no pas l’Update ja que volem centrar-nos en operacions relacionades amb les físiques) calcularem la distància i una variable temporal. Llavors es mou la capa canviant la posició. Finalment, comprovem si hem de moure la capa cap a la dreta o cap a l’esquerra (per a que es generi de nou a la dreta o a l’esquerra).

9.11 Interfícies d'usuari

9.11.1 Menús dins el GameController

Dins l'script de GameController, tenim guardats els estats possibles com es mostra a la **figura 9.11.1** :

```
public enum States
    {
        Game,
        Pause,
        GameOver
    }
```

Figura 9.11.1 : Estats possibles del videojoc

Llavors, hi ha mètodes públics que canviaran l'estat actual del joc. Veure la **figura 9.11.2.**

```
public void ChangeToGameState()
    {
        ReadPreferences();
        pauseScreen.SetActive(false);
        gameState = States.Game;
        Time.timeScale = 1;
    }

    public void ChangeToGameOverState()
    {
        gameState = States.GameOver;
    }
```

Figura 9.11.2 : Mètodes ChangeToGameState() i ChangeToGameOverState() de l'script GameController

Finalment, és el mètode Update és el que s'encarrega de gestionar què passa quan es canvia d'estat. Veure la **figura 9.11.3.**

```
void Update()
    {
        if (gameState == States.GameOver)
        {
            player.GetComponent<PlayerController>().enabled = false;

            if (!gameOverShown)
            {
```

```
                gameOverScreen.SetActive(true);
                gameOverShown = true;
            }

        }

        if (gameState == States.Game)
        {
            if (Input.GetKeyDown(KeyCode.Escape) && !escapePressed)
            {
                Debug.Log("s'ha press escape");
                pauseScreen.SetActive(true);
                gameState = States.Pause;
                escapePressed = true;
            }
        }

        if (gameState == States.Pause)
        {
            if (Input.GetKeyDown(KeyCode.Escape) && !escapePressed &&
pauseScreen.active == true)
            {
                ReadPreferences();
                pauseScreen.SetActive(false);
                gameState = States.Game;
            }
        }

        if (!Input.GetKeyDown(KeyCode.Escape)) escapePressed = false;
    }
```

Figura 9.11.3 : Mètode Update de l'script GameController

9.11.2 Menú principal

Hem creat un script anomenat MenuController que s'encarraguerà de gestionar el menú. Per a moure's dins els menús, es fan servir les fletxes del teclat. Les verticals per a moure's entre opcions i les horitzontals per pujar o baixar els volums. Dins l'script hi ha una variable anomenada "screen" que ens indica en quin menú estem (pren el valor de 0 si estem al menú de títol, 1 si estem al menú d'opcions). Llavors a l'Update, depenent de a quin menú estem, funcionem d'una manera o d'una altra:

Per a saber en quina opció estem (Start, Options, Quit), disposem d'una variable entera anomenada optionMenu (que agafa els valors 0, 1 i 2 respectivament). Quan canviem d'opció, l'sprite de la nova opció canvia al color vermell. El que estava seleccionat

anteriorment canvia al color blanc. D'això se n'encarrega el mètode SelectMenu, que es crida cada vegada que es selecciona una de les opcions. Veure la **figura 9.11.4.**

```
void SelectMenu(int op)
    {
        snd_option.Play();
        option = op;

        if (op == 1) restart.sprite = restart_on;
        if (op == 2) mainMenu.sprite = mainMenu_on;

        if (optionAnt == 1) restart.sprite = restart_off;
        if (optionAnt == 2) mainMenu.sprite = mainMenu_off;

        optionAnt = op;
    }
```

Figura 9.11.4 : Mètode SelectMenu de l'script MenuController

Finalment, al tenir una de les 3 opcions seleccionades, si premem la tecla *Enter* del teclat es realitza una acció depenent de la opció:

- Start: carreguem l'escena del gameplay
- Options: Mostrem el menú d'opcions
- Quit: Sortim de l'aplicació

Veure la **figura 9.11.5.**

```
if (Input.GetButtonDown("Submit") && !pressedSubmit)
        {
            snd_selection.Play();

            if(optionMenu == 1)
SceneManager.LoadScene("PlayerControllerScene");
            if (optionMenu == 2) LoadOptionsScreen();
            if (optionMenu == 3) Application.Quit();
        }
```

Figura 9.11.5 : Codi que realitza una acció depenent de la opció triada

Si carreguem el menú d'opcions, es crida el mètode LoadOptionsScreen(), que canvia el valor de la variable screen a 1, desactiva el menú del títol i activa el menú d'opcions. Veure la **figura 9.11.6.**

```
void LoadOptionsScreen()
    {
        pressedSubmit = true;
        menuScreen.SetActive(false);
        screen = 1;
        optionOptions = optionOptionsAnt = 1;
        music.sprite = music_on;
        sound.sprite = sound_off;
        back.sprite = back_off;

        optionsScreen.SetActive(true);
    }
```

Figura 9.11.6 : Mètode LoadOptionsScreen() de l'script MenuController

De la mateixa manera que amb el menú del títol, també tenim una variable per a saber en quina opció estem dins del menú d'opcions. Aquesta s'anomena *optionOptions* (pren el valor de 0 pel volum de la música, 1 pel volum dels sons i 2 per a tornar al menú del títol). També hi ha un mètode per a canviar el color a vermell de l'opció seleccionada. Aquest s'anomena SelectOption(). Veure la **figura 9.11.7.**

```
void SelectOption(int op)
    {
        snd_option.Play();
        optionOptions = op;

        if (op == 1) music.sprite = music_on;
        if (op == 2) sound.sprite = sound_on;
        if (op == 3) back.sprite = back_on;

        if (optionOptionsAnt == 1) music.sprite = music_off;
        if (optionOptionsAnt == 2) sound.sprite = sound_off;
        if (optionOptionsAnt == 3) back.sprite = back_off;

        optionOptionsAnt = op;
    }
```

Figura 9.11.7 : Mètode SelectOption de l'script MenuController

En aquest menú però, hem de fer les coses una mica diferent. Aquí necessitem saber quins són els volums que tenim guardats, ja que quan entrem al menú d'opcions, ja s'emplenin les barres depenent dels volums guardats (per exemple si tenim guardat un volum de música 5, al entrar al menú d'opcions ja s'han de mostrar 5 barres plenes). És per aquest motiu que necessitem carregar aquestes dues dades (volum de música i sons actuals). Això ho farem tot just iniciar l'script, i se n'encarrega el mètode *ReadPreferences()*.

```
private void ReadPreferences()
    {
        musicVolume = PlayerPrefs.GetInt("MusicVolume", 5);
        soundVolume = PlayerPrefs.GetInt("SoundVolume", 4);
    }
```

Figura 9.11.8 : Mètode ReadPreferences de l'script MenuController

Amb el PlayerPrefs, es poden guardar valors a variables i que aquests valors no desapareguin al canviar d'escena o al tancar l'aplicació. És com si tinguéssim una "base de dades" des d'on podem obtenir i guardar valors a variables. En el nostre cas, obtenim els valors de les variables "MusicVolume" i "SoundVolume". El mètode GetInt de PlayerPrefs requereix de dos paràmetres. El primer és un String amb el nom de la variable a obtenir-ne el valor i el segon és un valor que se li dona a la variable en cas de que no existeixi o de que no se li hagi determinat cap valor.

Un cop hem obtingut aquests dos volums ara hem de fer que es mostrin les barres de volum respectives al volums actuals. Això se n'encarreguen els mètodes *AdjustMusic()* i *AdjustSound()*.

- **AdjustMusic()**

  Adjusta el volum de la música del menú del títol i emplena les barres que toquin del volum de la música del menú d'opcions. Si el volum de música actual és 0, mostra l'sprite de la creu vermella.

- **AdjustSound()**

  Adjusta el volum dels sons del menú del títol i d'opcions (sons de canvi de volum, canvi d'opció i selecció d'opció) i emplena les barres que toquin del volum dels sons del menú d'opcions. Si el volum de sons actual és 0, mostra l'sprite de la creu vermella. Per a canviar-li el volum als sons dels menús, busca tots els GameObjects amb un tag específic que els hi hem posat al crear-los (el tag té com a nom "Sounds") i els hi canvia el volum de l'AudioSource. Veure la **figura 9.11.9.**

```
private void AdjustMusic()
    {
        if(musicVolume == 0) music_spr[0].enabled = true;
        else music_spr[0].enabled = false;

        for (int i = 1; i <= 10; i++)
        {
            if (i <= musicVolume) music_spr[i].sprite = vol_on;
            else music_spr[i].sprite = vol_off;
        }
        menu_music.volume = (musicVolume / 10F);
```

```
    }
    private void AdjustSound()
    {
        if (soundVolume == 0) sound_spr[0].enabled = true;
        else sound_spr[0].enabled = false;

        for (int i = 1; i <= 10; i++)
        {
            if (i <= soundVolume) sound_spr[i].sprite = vol_on;
            else sound_spr[i].sprite = vol_off;
        }

        GameObject[] sounds = GameObject.FindGameObjectsWithTag("Sounds");

        foreach(GameObject sound in sounds)
        {
            sound.GetComponent<AudioSource>().volume = soundVolume / 10F;
        }
    }
```

Figura 9.11.9 : Mètodes AdjustMusic() i AdjustSound() de l'script MenuController

Amb això fet, ara ja podem fer servir aquests dos mètodes per a canviar el volum de la música i sons amb les fletxes horitzontals si estem a les opcions de Music i Sounds, respectivament. Tant sols hem de controlar que no ens passem del límit. Per exemple, si tenim 10 barres de volum de música, no podem tirar cap a la dreta 11 vegades. És per això que necessitem un indicador per a saber quantes barres tenim activades. Aquestes variables s'anomenen soundIndicator i musicIndicator. Fent que quan premem la fletxa dreta es sumi 1 a la variable i prement la fletxa esquerra es resti 1 i que mai pugui ser més petita que 0 ni més gran que 10 hem solucionat el problema. Així doncs, quan es pugui afegir o treure una barra de volum la variable soundVolume s'increementarà o decrementarà en una unitat i llavors es crida el mètode de ajustar volum respectiu (per a música o sons) que hem vist anteriorment. Veure la **figura 9.11.10.**

```
if (optionOptions == 2)
        {
            if (Input.GetKeyDown(KeyCode.LeftArrow))
            {
                Debug.Log("sound -> left");
                if (soundIndicator > 0)
                {
                    soundIndicator -= 1;
                    soundVolume --;
                    AdjustSound();
                    snd_option.Play();
                }
```

```
        }

        if (Input.GetKeyDown(KeyCode.RightArrow))
        {
            Debug.Log("sound -> right");
            if (soundIndicator < 10)
            {
                soundIndicator += 1;
                soundVolume++;
                AdjustSound();
                snd_option.Play();
            }
        }

    }
```

Figura 9.11.10 : Ajustament de volums

Finalment, tenim la opció de tornar enrere. Si estem a sobre d'aquesta i premem la tecla *Enter,* cridem a dos mètodes:

- **SavePreferences()**

  Abans hem vist un mètode per a recuperar el valor de variables guardades a PlayerPrefs. Ara és tot el contrari.. Aquest mètode s'encarrega de guardar els nous valors dels volums a les variables "MusicVolume" o "SoundVolume". Veure la **figura 9.11.11.**

```
private void SavePreferences()
    {
        PlayerPrefs.SetInt("MusicVolume", musicVolume);
        PlayerPrefs.SetInt("SoundVolume", soundVolume);
        PlayerPrefs.Save();
    }
```

Figura 9.11.11 : Mètode SavePreferences de l'script MewnuController

- **LoadMenuScreen()**

  També hem vist ja com carregar el menú de opcions amb el mètode LoadOptionsScreen(). En aquest cas però, carreguem el menú del títol, posant la variable screen un altre cop a 0, desactivant el menú d'opcions i activant el menú del títol. Veure la **figura 9.11.12.**

```
void LoadMenuScreen()
    {
        pressedSubmit = true;
        snd_selection.Play();
        screen = 0;
        optionsScreen.SetActive(false);
        menuScreen.SetActive(true);
    }
```

Figura 9.11.12 : Mètode LoadMenuScreen de l'script MewnuController

Quan passem del menú principal al joc (seleccionant l'opció Start del menú del títol), el GameController ha de saber si s'han canviat els valors dels volums i si és així modificar-los pels sons de l'escena i per la música principal. És per això que a l'inici del GameController es crida a un mètode semblant als ja vistos *ReadPreferences*().

Aquest és el seu funcionament:

1. Llegeix el valor de les variables "MusicVolume" i "SoundVolume" del PlayerPrefabs.
2. Amb el valor del volum de la música obtingut, canvia el volum de la música principal de l'escena
3. Amb el volum dels sons obtingut, canvia el volum dels sons de l'escena.

Ara bé, el més fàcil seria que la nostra música principal estigués guardada en un AudioSource. D'aquesta manera podriem canviar el volum de la música directament. Si recordem, però, la nostra música està guardada en un projecte de l'FMOD Studio. És per aquest motiu que hem creat anteriorment un VCA (mirar apartat de creació de música amb FMOD i integració amb Unity). Pe a canviar el volum de la música, l'hem de canviar en el VCA. Aquest, però no té el mateix rang de volums. El valor que obtenim del PlayerPrefs és un valor enter del 0 al 10, prerò els vca van amb decibels. Per aquest motiu hem creat un rang nou amb el qual un valor enter del 0 al 10 equival a un cert nombre de decibels. Hem guardat aquesta relació a un Vector anomenat volumesToMusicRang. Aquest Vector té 11 posicions i cadascuna té un valor en decibels (així, si "MusicVolume" equival a 3, anirem a la casella 3 del volumesToMusicRang i obtindrem el valor en decibels). Veure les **figura 9.11.13 i 9.11.14.**

```
volumesToMusicRang = new Dictionary<int, float>();
        float valor = -50f;
        for(int i = 0; i <= 10; i++){
            volumesToMusicRang.Add(i, valor);
            valor = valor + 7f;
        }
```
Figura 9.11.13 : Diccionari per a guardar rangs de volum

```
public void ChangeMusicVolume(int musicVolume)
   {
       float volumeToVCA = volumesToMusicRang[musicVolume];

       vca.vcaVolume = volumeToVCA;
       Debug.Log("Music volume: " + vca.vcaVolume);
   }

   private void ReadPreferences()
   {
       int musicVolume = PlayerPrefs.GetInt("MusicVolume", 4);
       int soundVolume = PlayerPrefs.GetInt("SoundVolume", 4);

       Debug.Log("music prefab: " + musicVolume);
       Debug.Log("sound prefab: " + soundVolume);

       //Canviem el volum de la m'sica
       ChangeMusicVolume(musicVolume);

       //Canviem el soroll dels sons (efectes especials)
       GameObject[] sounds = GameObject.FindGameObjectsWithTag("Sounds");

       foreach(GameObject sound in sounds)
       {
           sound.GetComponent<AudioSource>().volume = soundVolume / 10F;
       }
   }
```

Figura 9.11.14 : Mètodes ReadPreferences i ChangeMusicVolume

9.11.3 Menú de pausa

La lògica del menú de pausa la explicarem en dues parts:

- **Menú de pausa - principal**

  Dins del menú de pausa principal podem realitzar 3 accions: tornar a la partida, anar a les opcions o tornar al menú principal. Veure la **figura 9.11.15.**

```
if (Input.GetKeyDown(KeyCode.Return) && !pressedSubmit)
    {
        snd_selection.Play();

        if (optionPauseMenu == 1) ResumeGame();
        if (optionPauseMenu == 2) LoadOptionsScreen();
        if (optionPauseMenu == 3) GoToMainMenu();
    }
```

Figura 9.11.15 : Codi que realitza una acció depenent de la opció triada

Si triem resumir la partida, tant sols hem de cridar un mètode del GameController anomenat ChangeToGameState(), el qual passarà de l'estat de Pause a l'estat de Game. Veure la **figura 9.11.16.**

```
void ResumeGame()
    {
        gameController.ChangeToGameState();
    }
```

Figura 9.11.16: Mètode ResumeGame()

Si el que es vol és carregar el menú d'opcions, es crida al mètode LoadOptionsScreen(), que també crida a un mètode del GameControleller que s'encarrega de desactivar el menú de pausa principal i activar el menú d'opcions. Veure la **figura 9.11.17.**

```
void LoadOptionsScreen()
    {
        pressedSubmit = true;
        gameController.LoadOptionsScreen();
    }
```

Figura 9.11.17 : Mètode LoadOptionsScreen()

Per últim, si el que es vol és anar al menú principal, es crida un mètode anomenat GoToMainMenu() que carrega la escena amb nom "MainMenu". Veure la **figura 9.11.18.**

```
void GoToMainMenu()
    {
        Time.timeScale = 1;
        SceneManager.LoadScene("MainMenu");
    }
```

Figura 9.11.18 : Mètode GoToMainMenu()

- **Menú de pausa - opcions**

  El menú d'opcions de pausa funciona igual que el ja explicat menú d'opcions del menú principal. Cal destacar en aquest però, que quan es selecciona la tercera opció (la d'anar enrere), guardem els valors dels nous volums amb PlayerPrefabs i carreguem el menú de pausa principal. Veure la **figura 9.11.19.**

```
private void LoadPauseScreen()
    {
        pressedSubmit = true;
        gameController.LoadPauseScreen();
    }
```

Figura 9.11.19 : Mètode LoadPauseScreen()

A més, al no pausar-se la música mentre estem en el menú, al ajustar-ne el volum es pot escoltar com puja o baixa i, per tant, agilitzar el procés d'ajustament de volums, essent més interactiu.

9.11.4 Menú de Game Over

La lògica del menú de Game Over és el mateixa que la dels altres menús. En aquest cas però, al seleccionar la primera opció (Resume) es crida un mètode anomenat RestartGame(). Aquest obté el nom de l'escena actual i carrega l'escena amb aquest nom. Això està fet d'aquesta manera perquè, pot passar que, de treball futur, s'implementin més escenes i amb això funcionaria per a totes. Veure la **figura 9.11.20.**

Efect**Figura 9.11.20:** Mètode RestartGame()

En canvi, si es selecciona la segona opció (Main Menu), cridem al mètode GoToMainMenu() que simplement carrega l'escena del menú principal. Veure la **figura 9.11.21.**

```
void GoToMainMenu(){
    Time.timeScale = 1;
    SceneManager.LoadScene("MainMenu");
}
```

Figura 9.11.21: Mètode GoToMainMenu()

9.11.5 Diàlegs

Primer ens centrarem en el diàleg en si. Aquest té 2 scripts:

- TypeWritterEffect: aquest script s'encarrega de mostrar el text del diàleg de manera progressiva, donant l'efecte de que s'està escrivint. El mètode que s'encarrega de realitzar-ho, s'anomena Run, el qual inicia una corrutina que mostra caràcter per caràcter del text al diàleg. Veure la **figura 9.11.22.**

```
private IEnumerator TypeText(string textToType, TMP_Text textLabel){
    textLabel.text = string.Empty;

    float t = 0;
    int charIndex = 0;

    while (charIndex < textToType.Length){
        t += Time.deltaTime * typeWritterSpeed;
        charIndex = Mathf.FloorToInt(t);
        charIndex = Mathf.Clamp(charIndex, 0, textToType.Length);

        textLabel.text = textToType.Substring(0, charIndex);

        yield return null;
    }

    textLabel.text = textToType;
}
```

Figura 9.11.22: Corrutina TypeText

- **DialogueUI**: aquest script gestiona els diàlegs. Els mostra, els tanca, mostra el text usant el typeWritterEffect, etc. Aquest script utilitza un ScriptableObject anomenat DialogueObject. Aquest objecte conté un array d'strings que es diu *dialogue* i un sprite que es diu *icona*. Veure la **figura 9.11.23.**

```
[CreateAssetMenu(menuName = "Dialogue/DialogueObject")]
public class DialogueObject : ScriptableObject
{
    [SerializeField] [TextArea] private string[] dialogue;
    [SerializeField] public Sprite icona;

    public string[] Dialogue => dialogue;
}
```

Figura 9.11.23: Script DialogueObject

Llavors, podem editar aquest DialogueObject a l'Inspector del Unity. Per exemple, aques és un dels DialogueObjects de l'escena. Veure la **figura 9.11.24.**

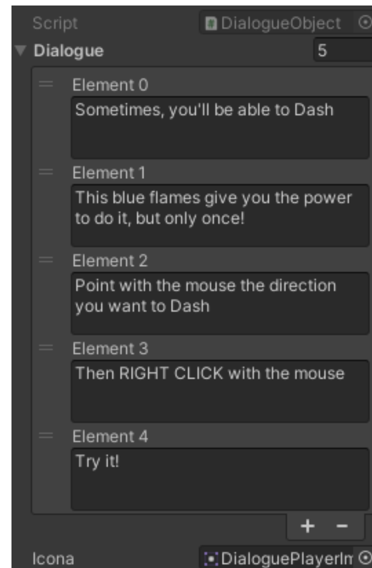


Figura 9.11.24: DialogueObject *Dash1* a l'Inspector

Conté 5 diàlegs que es mostraran un darrere de l'altre. També conté un sprite que és el que apareixarà l diàleg tal i com hem vist al disseny de la interfície.

Hi ha dos mètodes principals a l'script DialogueUI. Aquests són ShowDialogue i StepThroughDialogue.

- **ShowDialogue**: aquest mètode és l'encarregat de mostrar el diàleg. Ho fa seguint aquests passos:
    1. Comença una cinemàtica (per aturar el Player mentre el diàleg estigui present)
    2. Tanca l'últim diàleg obert
    3. Canvia l'sprite del diàleg nou per l'sprite del DialogueObject
    4. Mostra el diàleg
    5. Comença la corrutina StepThroughDialogue

    Veure la **figura 9.11.25.**

```
public void ShowDialogue(DialogueObject dialogueObject){
    EventController.Instance.CinematicStart();
    dialogueInactive = false;
    CloseDialogueBox();
    Image icona =
```

```
dialogueBox.transform.Find("Icona").gameObject.GetComponent<Image>();
    icona.sprite = dialogueObject.icona;

    dialogueBox.SetActive(true);
    StartCoroutine(StepThroughDialogue(dialogueObject));
}
```

Figura 9.11.25: Mètode ShowDialogue()

- **StepThroughDialogue**: s'encarrega de mostrar els diferents textos del DialogueObject en ordre. Es pot triar quin és el temps entre que es mostra un text i un altre. Un cop ha mostrat tots els textos, tanca el diàleg (desactivant-lo i aturant la cinemàtica perquè el Player pugui seguir). Veure la **figura 9.11.26.**

```
private IEnumerator StepThroughDialogue(DialogueObject dialogueObject)
{
    foreach (string dialogue in dialogueObject.Dialogue){
        yield return typeWritterEffect.Run(dialogue, textLabel);
        yield return new WaitForSeconds(1f);
    }

    CloseDialogueBox();
}
```

Figura 9.11.26: Mètode StepThroughDialogue()

9.12 Creació de la música

Un cop ja sabiem com sincronitzariem la música amb el gameplay, faltava crear-la. Vam preferir crear-la que obtenir-ne una ja existent, ja que d'aquesta manera podríem fer-ne una de simple i personalitzable (i és adient ja que al fer-la simple ens fa més senzilla la sincronització). Vam investigar diferents eines de creació de música com FLStudio o GarageBand però la que vam acabar escollint és BoscaCeoil, degut a la seva simplesa i fàcil utilització (baixa corba d'aprenentatge). En el BoscaCeoil, es trien instruments, i a través d'aquests es creen les pistes. Aquesta és una pista creada amb l'instrument 'Warm Pad'. Veure la **figura 9.12.1.**

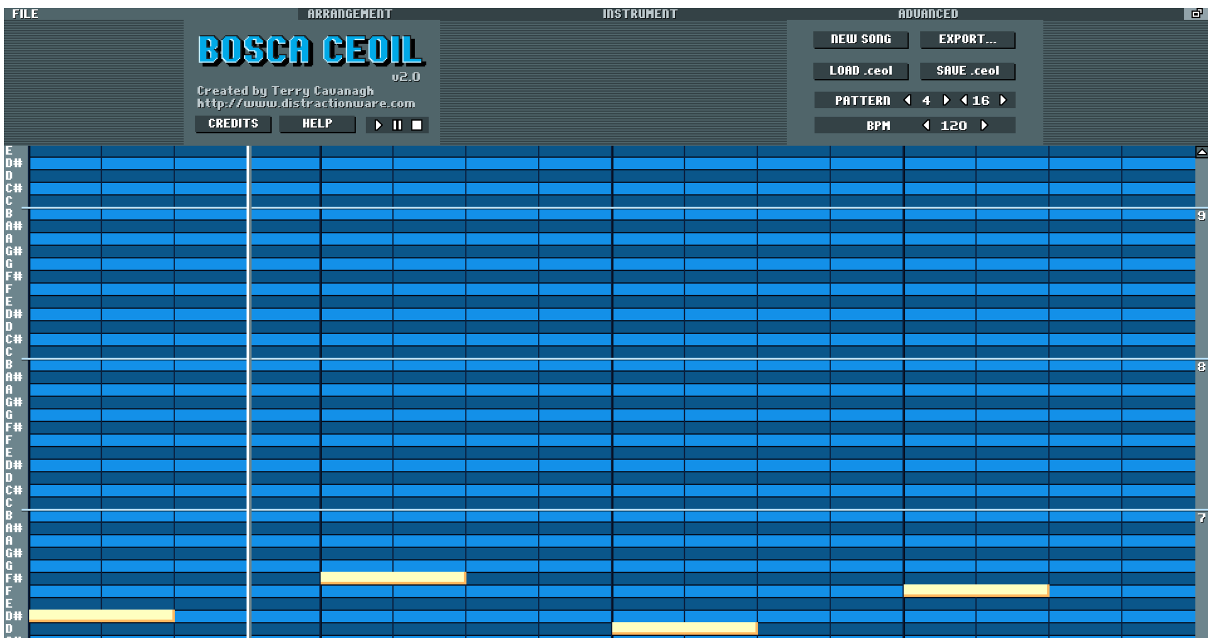


Figura 9.12.1 : Interfície del Bosca Ceoil

Un cop creades vàries pistes amb diferents instruments i melodies, es poden anar muntant per anar creant la música final. Aquest és el resultat del muntatge final de les nostres pistes. Veure la **figura 9.12.2.**

Figura 9.12.2 : Interfície del Bosca Ceoil (muntatge de pistes)

Un cop muntada la música, la vam exportar en format .wav ja que és preferible pel següent programa a utilitzar (FMOD).

9.13 Sincronització de la música amb el gameplay

9.13.1 Creació de l'Audio Manager

FMOD Studio Bank Loader

Aquest script ve proporcionat per la llibreria FMOD i és l'encarregat de determinar quin bank s'està fent servir. En el nostre cas, fem servir el bank on hi ha la música de l'FMOD (anomenat Master). Veure la **figura 9.13.1.**

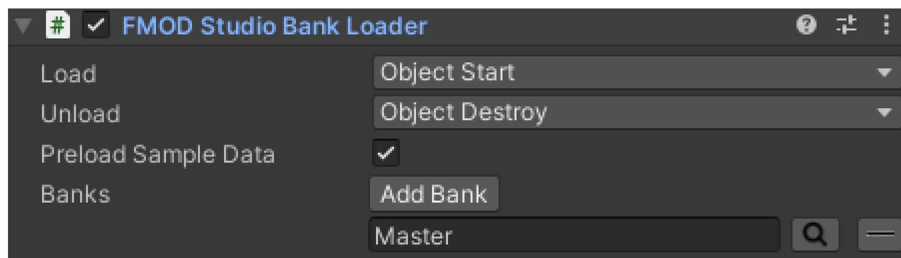


Figura 9.13.1 : Assignació de l'script FMOD Studio Bank Loader a a un GameObject dins l'Inspector

VCA

És l'encarregat d'ajustar el volum de la música. Ens serà necessari quan, des dels menús de pausa i inici, s'ajusti el volum d'aquesta.

Dins l'àmbit de FMOD, un VCA és el controlador de volum d'una pista (forma part del Mixer). Es pot veure l'script a la **figura 9.13.2.**

```
public class VCA : MonoBehaviour
{
    FMOD.Studio.VCA vca;

    [SerializeField]
    [Range(-60f, 20f)]
    public float vcaVolume;


    void Start()
    {
        vca = FMODUnity.RuntimeManager.GetVCA("vca:/MusicVCA");
    }

    void Update()
```

```
    {
        vca.setVolume(DecibelToLinear(vcaVolume));
    }

    private float DecibelToLinear(float dB)
    {
        float linear = Mathf.Pow(10.0f, dB / 20f);
        return linear;
    }

}
```

Figura 9.13.2 : Script VCA

A l'Start(), s'obté el VCA de l'FMOD i es guarda a la variable vca.

A l'Update(), actualitzem el volum de la música amb el mètode setVolume(float volume). El volum que se li passa, però, ha d'estar en format lineal i no amb decibels. És per això que necessitem un tercer mètode quae donat un nombre de decibels, el passa en format lineal.

Music Manager

Al principi d'iniciar la partida la música no sonarà. És fins que el jugador entre en contacte amb un BoxCollider2D que no comença a sonar. Ho hem decidit així per qüestions de disseny. D'això, se n'encarrega aquest mètode. Veure la **figura 9.13.3.**

```
private void OnTriggerEnter2D(Collider2D collision)
    {
        if(collision.CompareTag("Player"))
        {
            musicInstance.start();
            startMovingPlatforms = true;
            boxCollider.enabled = false;
        }
    }
```

Figura 9.13.3 : Mètode OnTriggerEnter2D de l'script MusicManager

Aquest script serà l'encarregat de gestionar la sincronització entre el Unity i l'FMOD. Més concretament, obtindrà valors de la música de l'FMOD i els guardarà en variables. La variable que contindrà quin és l'últim marcador per el qual s'ha passat es diu timelineInfo i és del tipus TimeLineInfo (tipus que ve amb la llibreria FMOD). Un TimeLineInfo és l'objecte que conté la informació del TimeLine del projecte a l'FMOD.

També és important tenir una instància de la música. Aquest l'anomenem musicInstance. Li hem de determinar quin timeLine fa servir i el callback. Veure la **figura 9.13.4.**

```
private void Start()
    {
        timelineInfo = new TimelineInfo();
        beatCallback = new FMOD.Studio.EVENT_CALLBACK(BeatEventCallback);
        timelineHandle = GCHandle.Alloc(timelineInfo, GCHandleType.Pinned);
        musicInstance.setUserData(GCHandle.ToIntPtr(timelineHandle));
        musicInstance.setCallback(beatCallback,
FMOD.Studio.EVENT_CALLBACK_TYPE.TIMELINE_BEAT |
FMOD.Studio.EVENT_CALLBACK_TYPE.TIMELINE_MARKER);
    }
```

Figura 9.13.4: Mètode Start de l'script MusicManager

El callback ensa serveix per a detectar un marcador o un beat. En el nostre cas, hem fet que pugui detectar els dos (tot i que només fem servir els marcadors). Llavors, al mètode Update() és on guardem el nom de l'últim marcador i beat pels que el timeline ha passat. Veure la **figura 9.13.5.**

```
private void Update()
    {
        if (lastMarkerString != timelineInfo.lastMarker)
        {
            lastMarkerString = timelineInfo.lastMarker;

            if (markerUpdated != null)
            {
                markerUpdated();
            }
        }

        if (lastBeat != timelineInfo.currentBeat)
        {
            lastBeat = timelineInfo.currentBeat;

            if (beatUpdated != null)
            {
                beatUpdated();
            }
        }
    }
```

Figura 9.13.5 : Mètode Update de l'script MusicManager

Finalment, el mètode següent anomenat BeatEventCallback és el mètode que acaba de sincronitzar l'FMOD Studio amb l'Unity. Més concretament, aquest mètode obté la informació del timeline del projecte a l'FMOD, les dades. Entyenem com a dades els marcadors, beats, etc. El mètode guarda el nom de l'últim marcador a la variable timelineInfo (timelineInfo.lastMarker). El mètode retorna un FMOD.RESULT, que pot ser FMOD.RESULT.OK si s'han obtingut correctament les dades del FMOD Studio o (!= FMOD.RESULT.OK) si alguna cosa ha anat malament. Veure la **figura 9.13.6.**

```
#if UNITY_EDITOR
    private void OnGUI()
    {
        GUILayout.Box($"Current Beat = {timelineInfo.currentBeat}, Last Marker =
{(string)timelineInfo.lastMarker}");
    }
#endif

    [AOT.MonoPInvokeCallback(typeof(FMOD.Studio.EVENT_CALLBACK))]
    static FMOD.RESULT BeatEventCallback(FMOD.Studio.EVENT_CALLBACK_TYPE type, IntPtr
instancePtr, IntPtr parameterPtr)
    {
        FMOD.Studio.EventInstance instance = new FMOD.Studio.EventInstance(instancePtr);

        IntPtr timelineInfoPtr;
        FMOD.RESULT result = instance.getUserData(out timelineInfoPtr);

        if (result != FMOD.RESULT.OK)
        {
            Debug.Log("Timeline Callback error: " + result);
        }
        else if (timelineInfoPtr != IntPtr.Zero)
        {
            GCHandle timelineHandle = GCHandle.FromIntPtr(timelineInfoPtr);
            TimelineInfo timelineInfo = (TimelineInfo)timelineHandle.Target;

            switch (type)
            {
                case FMOD.Studio.EVENT_CALLBACK_TYPE.TIMELINE_BEAT:
                    {
                        var parameter =
(FMOD.Studio.TIMELINE_BEAT_PROPERTIES)Marshal.PtrToStructure(parameterPtr,
typeof(FMOD.Studio.TIMELINE_BEAT_PROPERTIES));
                        timelineInfo.currentBeat = parameter.beat;
                    }
                    break;
                case FMOD.Studio.EVENT_CALLBACK_TYPE.TIMELINE_MARKER:
                    {
                        var parameter =
(FMOD.Studio.TIMELINE_MARKER_PROPERTIES)Marshal.PtrToStructure(parameterPtr,
```

```
typeof(FMOD.Studio.TIMELINE_MARKER_PROPERTIES));
                        timelineInfo.lastMarker = parameter.name;
                    }
                    break;
            }
        }
        return FMOD.RESULT.OK;
    }
}
```

Figura 9.13.6 : Mètode BeatEventCallback de l'script MusicManager

9.13.2 Creació d'objectes que es sincronitzen amb la música

9.13.2.1 Plataformes que apareixen i desapareixen

```
public class MusicPlatform : MonoBehaviour
{
    [SerializeField]
    private string stringToWaitFor = "Event1";

    private string markerDone = "noMarker";

    private BoxCollider2D collider;
    private SpriteRenderer spr;

    private void Awake()
    {
        collider = GetComponent<BoxCollider2D>();
        spr = GetComponent<SpriteRenderer>();
    }

    void Update()
    {
        string currentMarker =
MusicManager.instance.timelineInfo.lastMarker;

        if (currentMarker.StartsWith(stringToWaitFor) &&
!currentMarker.Equals(markerDone))
        {
            markerDone = currentMarker;
```

```
            MusicAction();
        }
    }

    public void MusicAction()
    {
        collider.enabled = !collider.enabled;
        spr.enabled = !spr.enabled;
    }
}
```

Figura 9.13.7 : Classe MusicPlatform

Aquí, guardem com a variables el nom del marcador que estem esperant (stringToWaitFor), que serveix per saber quan invocar un mètode que realitzi una certa acció. En el cas de les plataformes que apareixen i desapareixen té el valor de: “Event1”. També necessitem un altre String, que anomenem markerDone. Aquest serveix per determinar si ja hem trobat aquest marcador (i d’aquesta manera no invocarem el mètode que realitza una acció més d’una vegada quan trobi el marcador amb nom “stringToWaitFor”). A més, necessitem un BoxCollider2D(collider) i un SpriteRenderer(spr). Primer obtenim l’últim marcador per el que s’ha passat a la música a l’FMOD. Llavors, si no hi hem passat ja i si comença per ‘stringToWaitFor’, indiquem que ja hi hem passat i cridem el mètode MusicAction(). Veure la **figura 9.13.7.**

Per a que això funcioni, prèviament a l’FMOD, hem fet que els noms dels marcadors que volem que es sincronitzin amb aquest objecte comencin per l’String ‘stringToWaitFor’, seguit d’un número. Per exemple, a l’FMOD hem posat els marcadors “Event1_01”, “Event1_02”, “Event1_03”. etc. D’aquesta manera, aquest script (ja que té stringToWaitFor = “Event1”), cridarà MusicAction() quan passem per aquests marcadors. Veure la **figura 9.13.8.**

Figura 9.13.8 : Sprite de la plataforma que apareix i desapareix

Tant la variable stringToWaitFor com la variable markerDone també les trobem als scripts dels altres 3 objectes. També tenen tots el mètode MusicAction(), tot i que variarà en funció de l’objecte. En aquest cas, fa el que es mostra a la **figura 9.13.9**.

```
public void MusicAction()
  {
      collider.enabled = !collider.enabled;
      spr.enabled = !spr.enabled;
  }
```

Figura 9.13.9 : Mètode MusicAction() de l'script MusicPlatform

Activa el collider i l'sprite renderer de la plataforma si estan desactivats i viceversa (creant l'efecte que apareix i desapareix).

9.13.2.2 Plataformes de salt

```
public class MusicJumpPlatform : MonoBehaviour
{
    [SerializeField]
    private string stringToWaitFor = "Event1";

    private GameObject jumpCollider;
    private Animator anim;

    private string markerDone = "noMarker";

    private void Awake()
    {
        jumpCollider = transform.Find("JumpCollider").gameObject;
        anim = GetComponent<Animator>();
    }

    void Update()
    {
        string currentMarker =
MusicManager.instance.timelineInfo.lastMarker;

        if ((currentMarker.StartsWith(stringToWaitFor)) &&
!currentMarker.Equals(markerDone))
        {
            markerDone = currentMarker;

            MusicAction();
        }
    }

    public void MusicAction()
    {
```

```
        jumpCollider.SetActive(true);
        StartCoroutine(EndMusicAction(0.1f));
    }

    IEnumerator EndMusicAction(float nSec)
    {
        yield return new WaitForSeconds(nSec);

        anim.SetTrigger("Activation");
        jumpCollider.SetActive(false);
    }
}
```

Figura 9.13.10 : Script MusicJumpPlatform

Per les plataformes de salt, també tenim un Animator, ja que a més a més de l'acció (impulsar el jugador) també es farà una animació. A més, necessitem un CapsuleCollider2D, que anomenem jumpCollider. Aquest serà l'encarregat de detectar si el Player està a sobre de la plataforma de salt i si és així impulsar-lo. Veure la **figura 9.13.11.**

Figura 9.13.11 : Spritesheet de la plataforma de salt

En aquest script, el MusicAction activa el jumpCollider i llavors el desactiva al cap de 0.1 segons mitjançant una Corrutina (EndMusicAction). D'aquesta manera, el jumpCollider només està actiu un instant molt curt de temps, fent que el Player només s'impulsi en el moment exacte en què el marcador s'ha detectat. El jumpCollider conté un script anomenat PlatformJumpCollider, el qual comprova si entra amb Contacte amb el Player, i si és així, l'impulsa. Veure la **figura 9.13.12.**

```
public class PlatformJumpCollider : MonoBehaviour
{
    [SerializeField]
    private float jumpForce = 100f;

    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
collision.gameObject.GetComponent<Rigidbody2D>().AddForce(Vector3.up *
jumpForce);
        }
    }
}
```

Figura 9.13.12 : Script PlatformJumpCollider

9.13.2.3 Trampes làser

```
public class MusicLaserTrap : MonoBehaviour
{
    [SerializeField]
    private string stringToWaitFor = "Event2";

    private Animator anim;
    private CapsuleCollider2D collider;
    public float damage;

    private string markerDone = "noMarker";

    private void Awake()
    {
        anim = GetComponent<Animator>();
        collider = GetComponent<CapsuleCollider2D>();
    }

    void Start()
    {
        collider.enabled = false;
    }

    void Update()
    {
        string currentMarker =
MusicManager.instance.timelineInfo.lastMarker;

        if (currentMarker.StartsWith(stringToWaitFor) &&
```

```
!currentMarker.Equals(markerDone))
        {
            markerDone = currentMarker;

            MusicAction();
        }
    }

    public void MusicAction()
    {
        //Canvi a animació activation
        anim.SetBool("Activation", true);
    }


    public void EndActivation()
    {
        //Canvi a animació idle
        anim.SetBool("Activation", false);

        //Disable capsule collider
        collider.enabled = false;
    }

    public void EnableCollider()
    {
        //Enable capsule collider
        collider.enabled = true;
    }
}
```

Figura 9.13.13: Script MusicLaserTrap

Per a les trampes làser, també necessitem un Animator (anim) i un CapsuleCollider2D (collider). Aquest cop, però, el collider ens servirà per a fer mal al Player (i no impulsar-lo, com ho feia la plataforma de salt). Veure la **figura 9.13.14.**

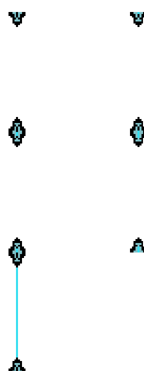


Figura 9.13.14 : Spritesheet de les trampes làser

Aquí, el mètode MusicAction() tant sols activa l'animació d'activació de la trampa làser, i serà durant aquesta que s'activarà i desactivarà el collider (a l'inici i final de l'animació respectivament) mitjançant Animation Events. Veure la **figura 9.13.15.**

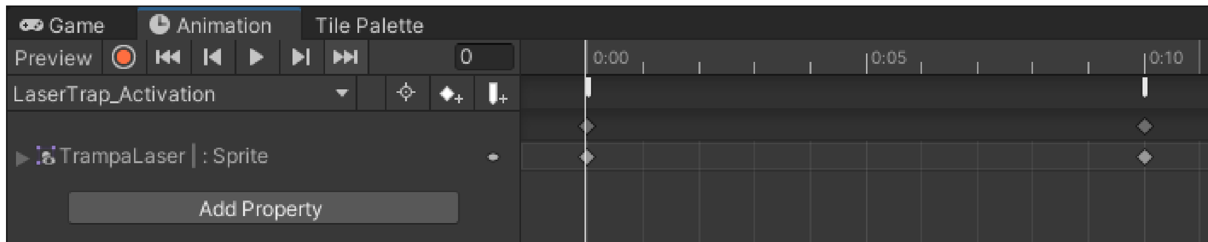


Figura 9.13.15 : Animació de l'activació de les trampes làser amb Animation Events

9.13.2.4 Plataformes que es desplacen

Les plataformes que es desplacen funcionen de manera diferent. La idea és que quan el marcador actual és l'esperat per l'objecte (stringToWaitFor), la plataforma es canvia de direcció en un eix (vertical o horitzontal). Veure la **figura 9.13.16.**

Figura 9.13.16 : Sprite de les plataformes que es desplacen

Per això necessitem més variables:

- **period**: periode de la plataforma per canviar de direcció (en el nostre cas és de 2 segons). És a dir, cada 2 segons canvia de direcció.
- **nextActionTime**: variable que ens determina quin és el temps on s'haurà de canviar de direcció.
- **canMove**: boleà que determina si la plataforma es pot moure o no.
- **speed**: velocitat en què es mou la plataforma.
- **movingLeft**: boleà que determina si la plataforma s'està movent cap a l'esquerra.
- **movingUp**: boleà que determina si la plataforma s'està movent cap amunt.
- **verticalPlatform**: boleà que determina si la plataforma es mou verticalment (true) o horitzontalment (false).

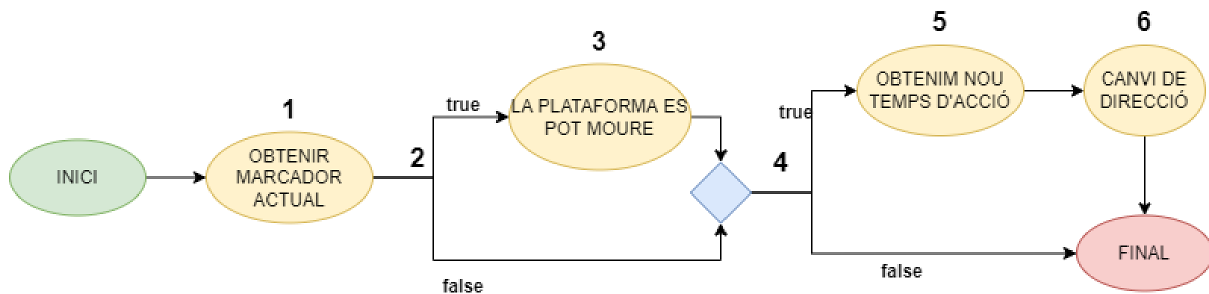
Un cop definides aquestes dues variables, anem a veure els mètodes principals. Veure la **figura 9.13.17.**

```
void Update()
    {
        currentMarker = MusicManager.instance.timelineInfo.lastMarker;

        if (currentMarker.StartsWith(stringToWaitFor) && !canMove)
        {
            canMove = true;
        }

        if(canMove)
        {
            if (Time.time > nextActionTime)
            {
                nextActionTime = Time.time + period;

                if (!verticalPlatform) movingLeft = !movingLeft;
                else movingUp = !movingUp;

            }
        }

    }

    private void FixedUpdate()
    {
        if (canMove)
        {
            if (!verticalPlatform)
            {
                if (movingLeft) transform.Translate(Vector2.left *
Time.deltaTime * speed);
                else transform.Translate(Vector2.right * Time.deltaTime *
speed);
            }
            else
            {
                if (movingUp) transform.Translate(Vector2.up *
Time.deltaTime * speed);
                else transform.Translate(Vector2.down * Time.deltaTime *
speed);
            }
        }
    }
```

Figura 9.13.17 : Mètodes Update() i FixedUpdate() de l'script MovingPlatform

Al mètode Update és on determinem quan hem de fer un canvi de direcció i cap a quina direcció. Al FixedUpdate, com que va més orientat a les físiques, és a on realitzarem el moviment de la plataforma cap a la direcció triada a l'Update. Al principi ho vam fer tot a l'Update i vam veure que el moviment de la plataforma no era gents fluït, així que ho vam solucionar afegint el FixedUpdate.

Veiem el funcionament de l'Update a la **figura 9.13.18.**



1: string currentMarker = MusicManager.instance.timelineInfo.lastMarker;

2: if (currentMarker.StartsWith(stringToWaitFor) && !canMove)

3: canMove = true

4: if(canMove)

5: nextActionTime = Time.time + period;

6: if (!verticalPlatform) movingLeft = !movingLeft;
else movingUp = !movingUp;

Figura 9.13.18 : Diagrama de funcionament del mètode Update() de l'script MovingPlatform.

Si el marcador comença per 'stringToWaitFor', la plataforma es pot moure.Si la plataforma es pot moure, en cada periode es canviarà la direcció de desplaçament.

Ara entra el FixedUpdate, que a causa de la seva simplicitat, no és necessari fer-ne un diagrama. Tant sols, si la plataforma es pot moure, es mourà cap a la direcció determinada a l'Update a la velocitat 'speed'. La direcció pot ser: amunt, avall, dreta o esquerra. A més, la plataforma de desplaçament disposa d'un CapsuleCollider2D semblant a la de les plataformes de salt. Aquesta, quan detecti que el Player està a sobre, farà que aquest es desplaci amb la plataforma. Aquest és el l'script d'aquest collider. Veure la **figura 9.13.19.**

```
public class MovingPlatformCollider : MonoBehaviour
{
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if(collision.CompareTag("Player"))
```

```
        {
            Debug.Log("contacte amb player");
            collision.transform.SetParent(transform);
        }
    }

    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            collision.transform.SetParent(null);
        }
    }
}
```

Figura 9.13.19 : Script MovingPlatformCollider

Quan el Player entra en contacte amb el collider, es fa que el “pare” del Player sigui el collider. D’aquesta manera, la posició del Player serà la mateixa que la del collider i per tant es desplaçarà amb la plataforma. De manera inversa, quan el Player surt del collider, es fa que el “pare” del Player sigui null. Així ja no es mourà amb la plataforma.

10.
Implantació i resultats

10.1 Legislació i normativa vigent

El projecte desenvolupat no presenta cap problema des del punt de vista legislatiu. No s'ha tingut en compte el Reglament General de Protecció de dades (RGPD), ja que el sistema en cap moment tracta cap tipus de dades relatives a l'usuari. Sobre la llei de serveis de la societat de la informació i comerç electrònic (LSSICE), el projecte no constitueix una activitat econòmica en cap dels sentits.

10.2 Resultats de la màquina d'estats de Player

10.2.1 Idle State

Figura 10.1.1: Player en IdleState

10.2.2 MoveState

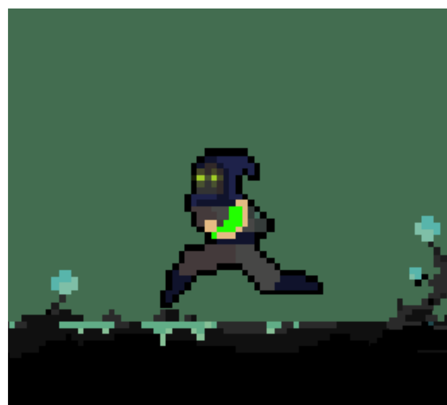


Figura 10.1.2: Player en MoveState

10.2.3 Land State

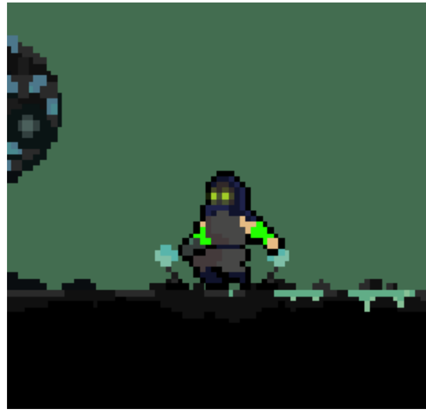


Figura 10.1.3: Player en LandState

10.2.4 Jump State

Figura 10.1.4: Player en JumpState

10.2.5 DashState

Figura 10.1.5: Player en l'inici del DashState (el jugador està mantinguent apretat el botó de dash)

Figura 10.1.6: Player en l'execució del dash

10.2.6 WallJump State

Figura 10.1.7: Player en WallJumpState (Acaba de saltar des de la paret que li queda darrera)

10.2.7 AttackState

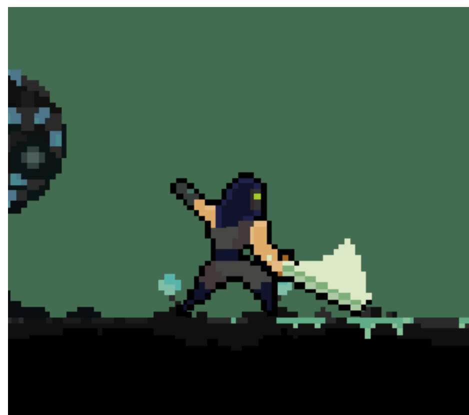


Figura 10.1.8: Player en AttackState

10.2.8 WallGrabState

Figura 10.1.9: Player en WallGrabState

10.2.9 WallClimbState

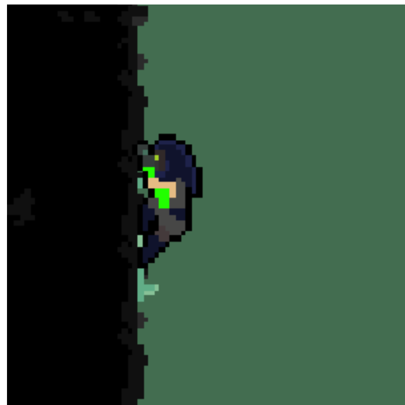


Figura 10.1.10: Player en WallClimbState

10.2.10 WallSlideState

Figura 10.1.11: Player en WallSlideState

10.2.11 InAirState

Figura 10.1.12: Player en InAirState

10.2.12 LedgeClimbState

Figura 10.1.13 : Fase *isHanging* de l'estat *LedgeClimbState*

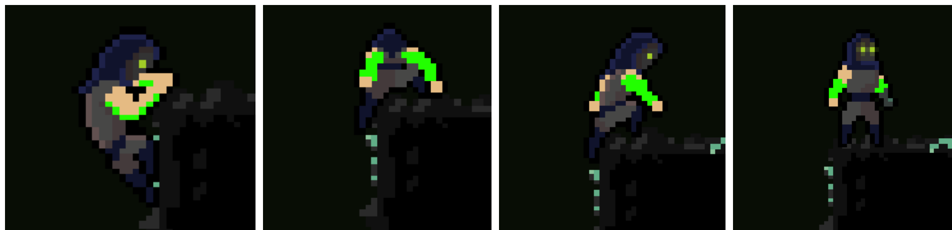


Figura 10.1.14 : Frames de l'animació del personatge durant la fase *is Climbing* de l'estat *LedgeClimbState*

10.3 Resultats dels efectes del *shader*

10.3.1 Vida diegètica

Figura 10.2.1: Visualització de l'efecte. D'esquerra a dreta, la vida del jugador en cada imatge és: 100%, 60%, 40%, 20%, 0%

10.3.2 Efecte de 'hit'

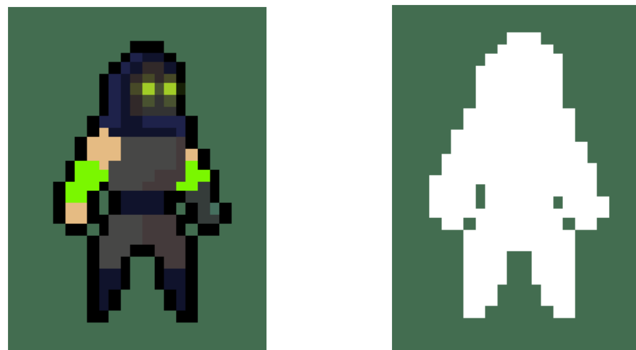


Figura 10.2.2: Aspecte original a l'esquerra, aspecte amb l'efecte de 'hit' activat

10.3.3 Efecte de realçar contorn al tenir el *dash* disponible

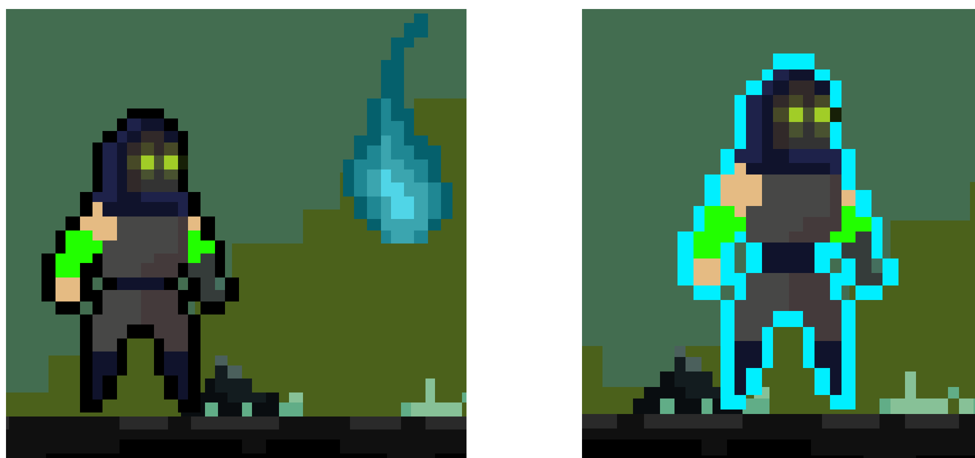


Figura 10.2.3: Personatge abans i després d'agafar la flama (indicador de dash)

10.3.4 Efectes de parpalleig

Figura 10.2.4: Efecte de parpalleig al morir

Figura 10.2.5: Efecte de parpalleig a l'estar sota el 20% de vida

10.4 Parallax

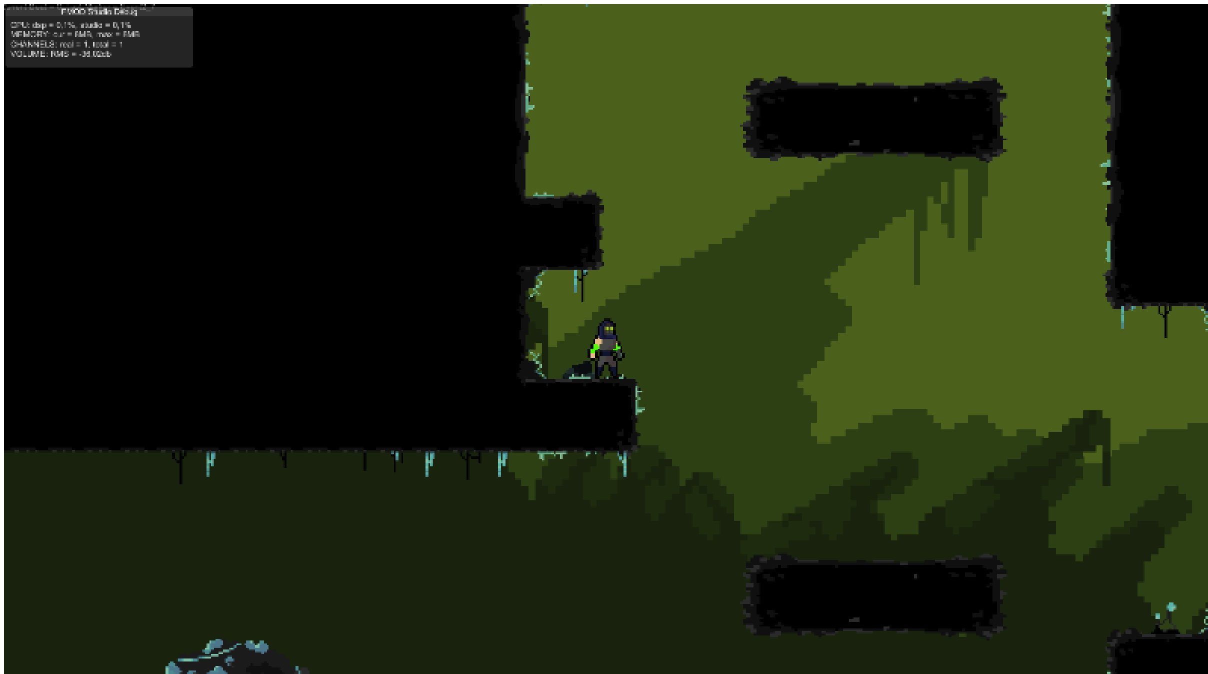


Figura 10.3.1 : Escena in-game on es veuen 3 capes del parallax

Figura 10.3.2 : Escena in-game on es veuen 4 capes del parallax

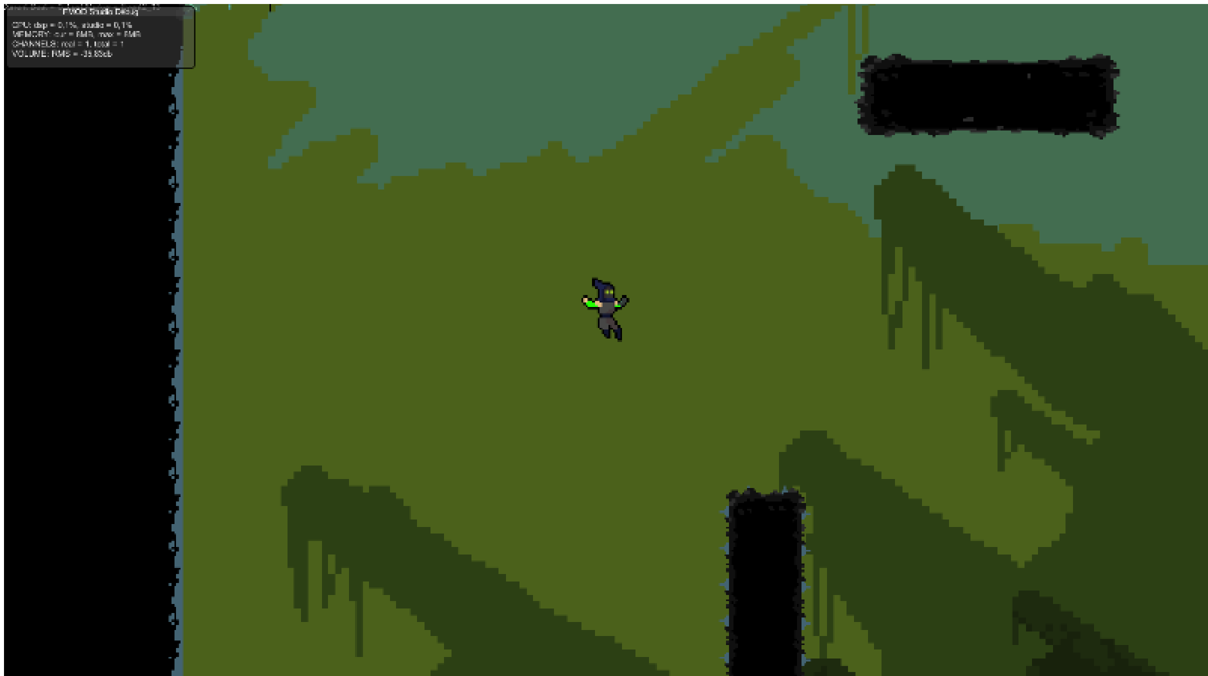


Figura 10.3.3 : Escena in-game on es veuen 4 capes del parallax

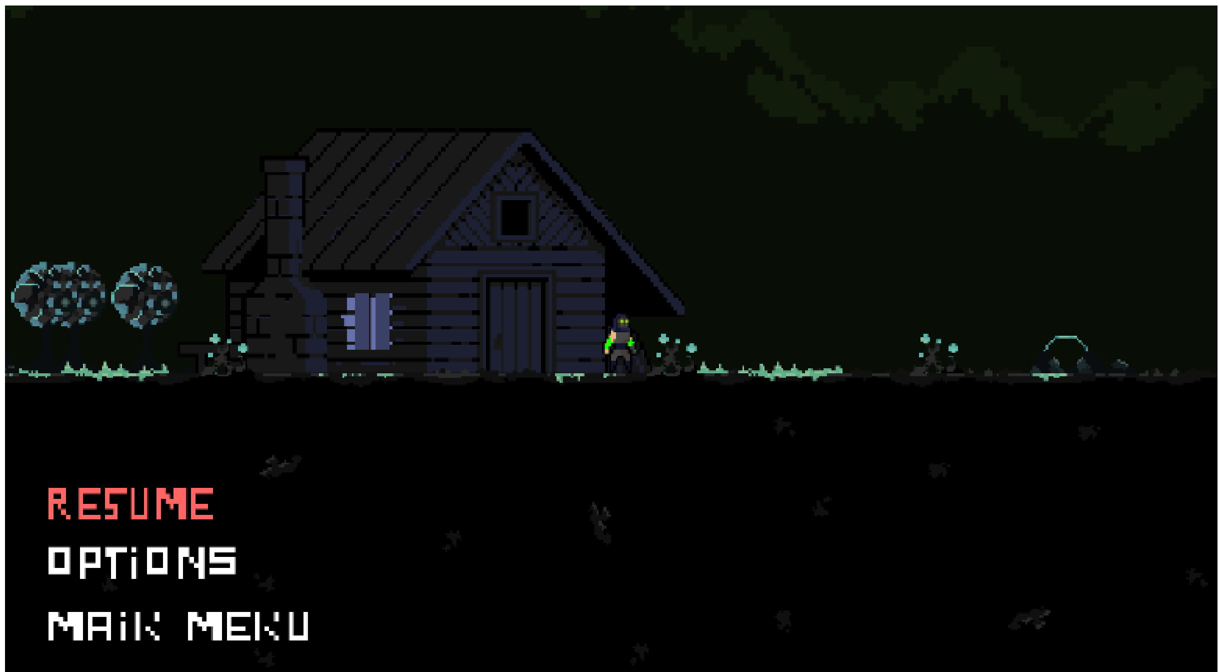
10.5 Interfícies d'usuari

10.5.1 Menú principal

Figura 10.4.1 : Menú del títol del menú principal

Figura 10.4.2 : Menú d'opcions del menú principal

10.5.2 Menú de pausa



10.4.3 : Escena del gameplay amb el menú de pausa principal activat

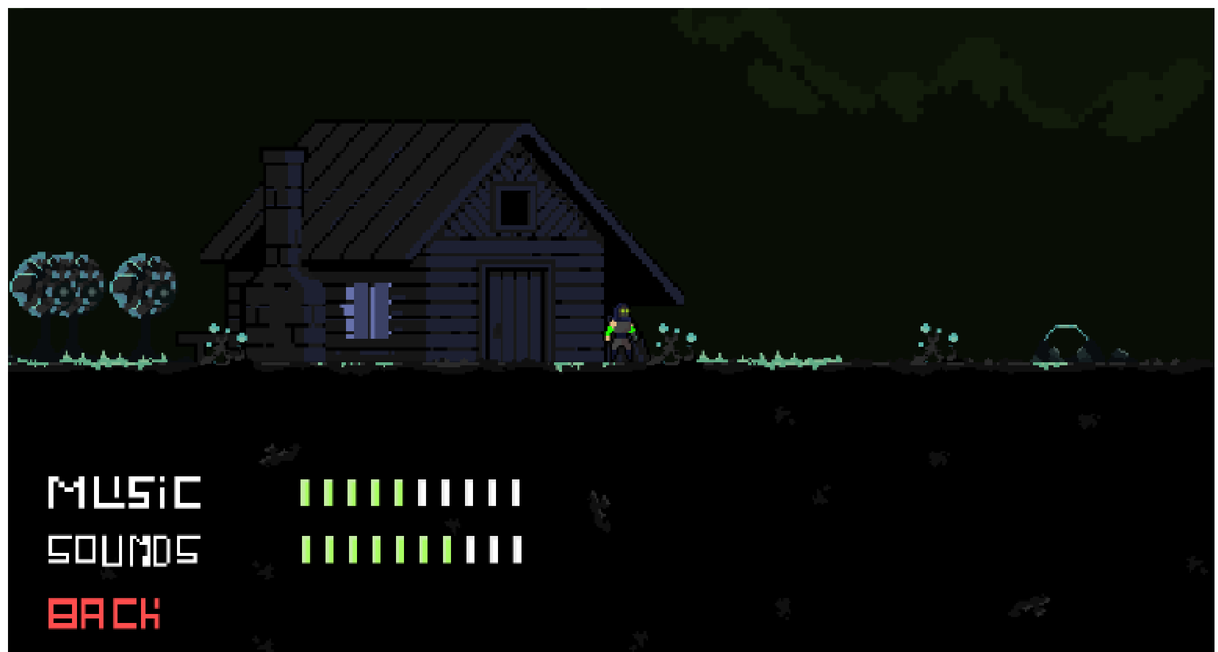


Figura 10.4.4 : Escena del gameplay amb el menú de pausa (opcions) activat

10.5.3 Menú de Game Over

Figura 10.4.5: Menú de Game Over a l'escena amb l'opció *Restart* seleccionada

10.5.4 Diàlegs

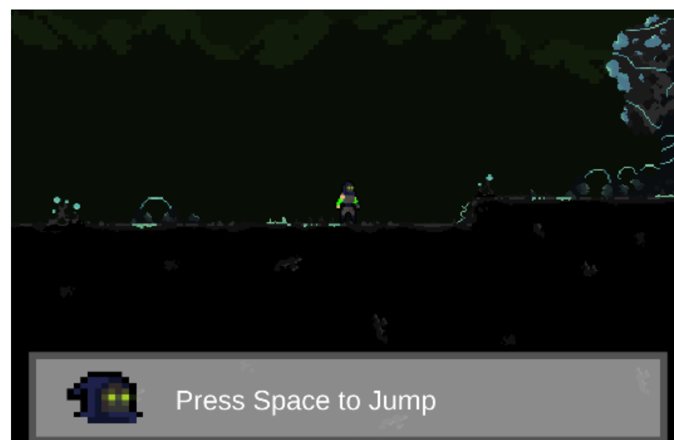


Figura 10.4.6: Escena del joc amb diàleg de *Jump* actiu

Figura 10.4.7: Escena del joc amb diàleg *Climb* actiu

Figura 10.4.8: Escena del joc amb diàleg *Dash1* actiu

10.6 Elements que es sincronitzen amb la música

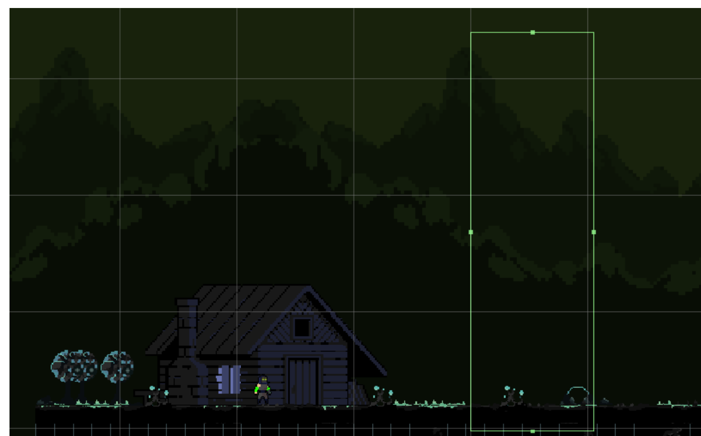


Figura 10.5.1 : Escena in-game on es veu el BoxCollider2D del MusicManager

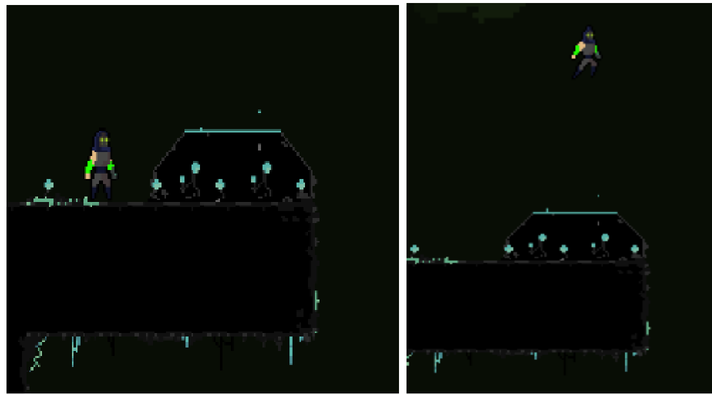


Figura 10.5.2: Escena in-game on es veu la Plataforma de salt

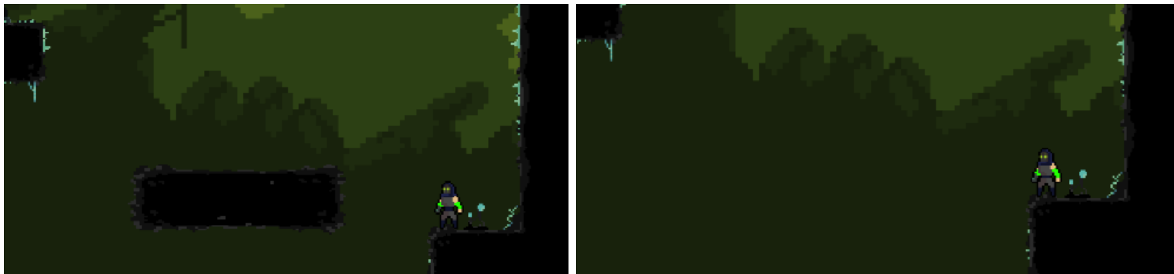


Figura 10.5.3 : Escena in-game on es veu la plataforma que apareix i desapareix (esquerra: apareix, dreta: desapareix)

Figura 10.5.4 : Escena in-game on es veu el la plataforma de desplaçament

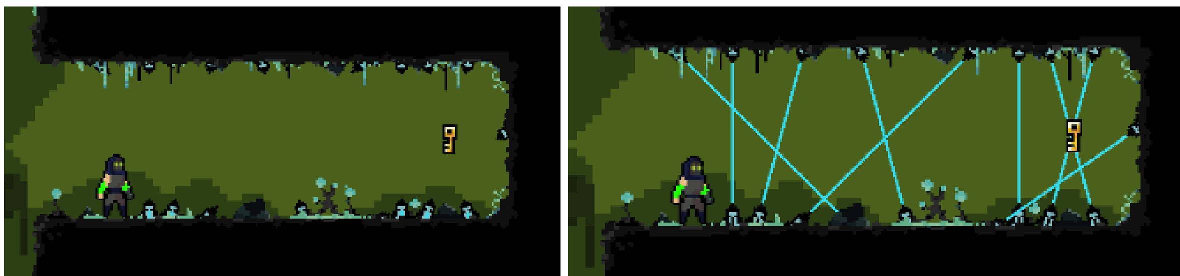


Figura 10.5.5 : Escena in-game on es veuen les trampes làser (esquerra: desactivades, dreta: activades)

Figura 10.5.6 : Escena in-game on es veuen els arbres
(esquerra: desactivats, dreta: activats)

10.7 Lluita amb el Boss

Figura 10.6.1 : Fase 1: pluja de meteorits

Figura 10.6.2 : Fase 2: Invocació d'enemics amb portals

Figura 10.6.3 : Fase 3: atac de boomerang

Figura 10.6.4 : Fase 3: player rep impacte de l’atac especial del Boss

Figura 10.6.5 : Fase 3: atac a melee del Boss

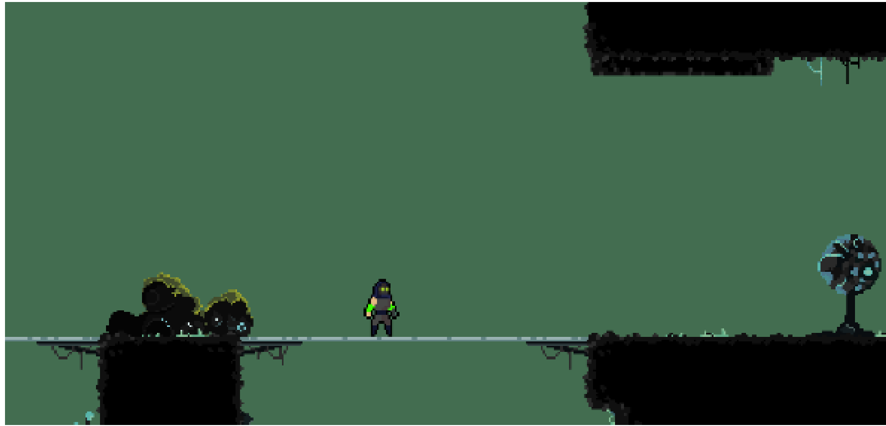


Figura 10.6.6 : Fi de la lluita: s'obre la comporta

10.8 Màquines d'estats pels enemics

Pels resultats de les màquines d'estats dels enemics ens centrarem en els dos principals enemics que ens trobarem durant la partida: el BallAndChain i l'Archer:

10.8.1 BallAndChain

10.8.1.1 Idle State

Figura 10.7.1: BallAndChain en estat de Idle

10.8.1.2 Move State

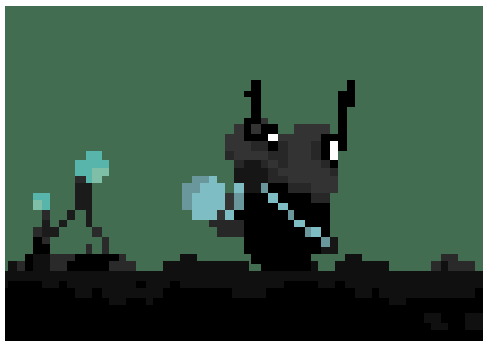


Figura10.7.2: BallAndChain en estat de Move

10.8.1.3 LookForPlayer State

Figura 10.7.3: BallAndChain en estat de LookForPlayer

10.8.1.4 PlayerDetected State

Figura 10.7.4: BallAndChain en estat de PlayerDetected

10.8.1.5 Charge State

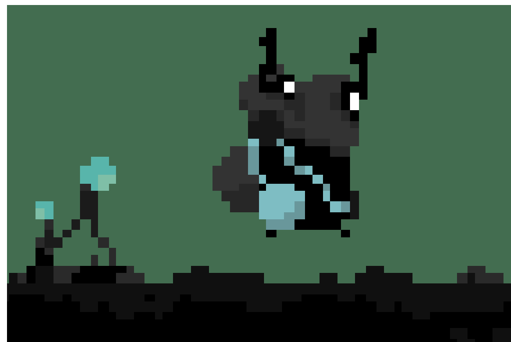


Figura 10.7.5: BallAndChain en estat de Charge

10.8.1.6 MeleeAttack State

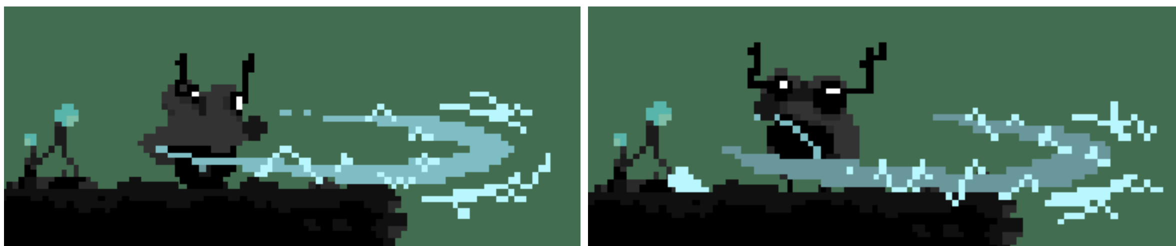


Figura 10.7.6: BallAndChain en estat de MeleeAttack (esquera: 1r atac, dreta: 2n atac)

10.8.1.7 Dead State

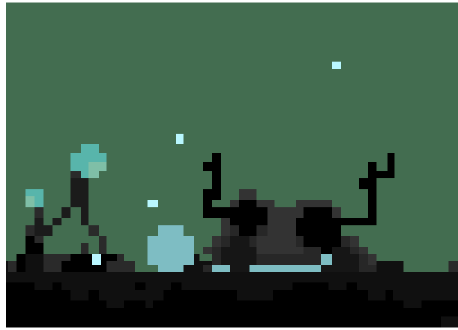


Figura 10.7.7: BallAndChain en estat de Dead

10.8.2 Archer

10.8.1.1 Idle State

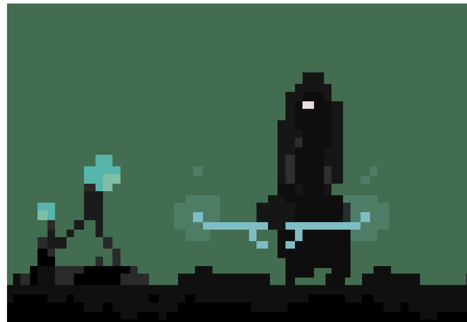


Figura 10.7.8: Archer en estat de Idle

10.8.1.2 Move State

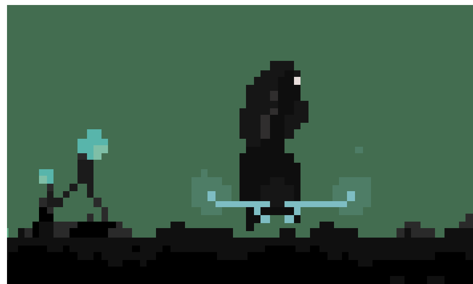


Figura 10.7.9: Archer en estat de Move

10.8.1.3 LookForPlayer State

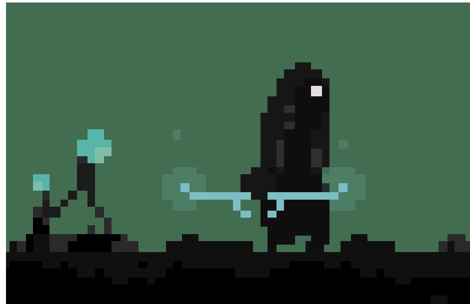


Figura 10.7.10: Archer en estat de LookForPlayer

10.8.1.4 PlayerDetected State

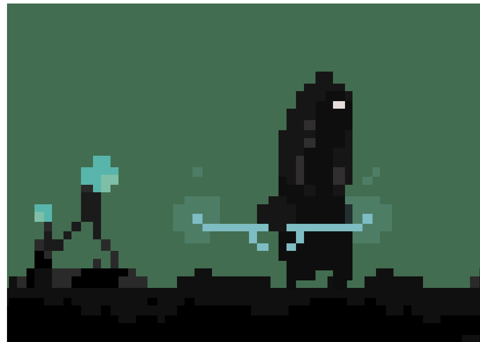


Figura 10.7.11: Archer en estat de PlayerDetected

10.8.1.5 Dodge State

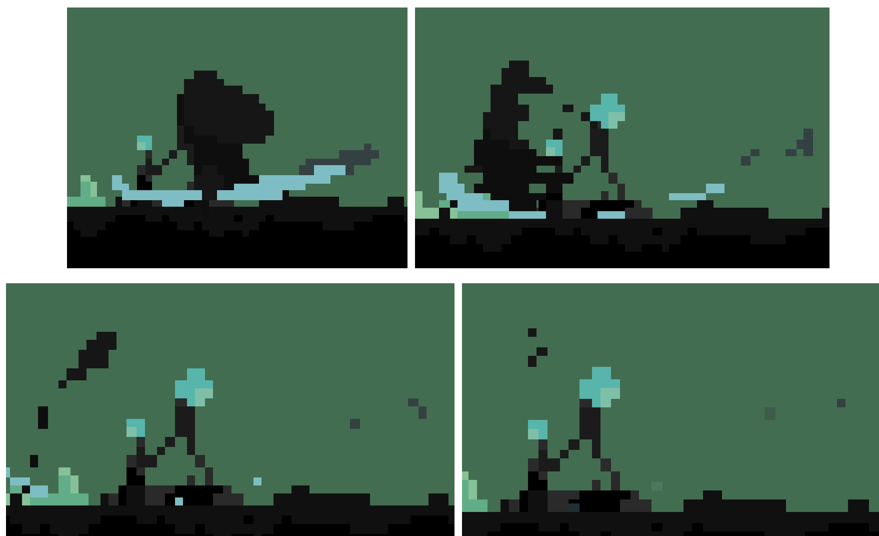


Figura 10.7.12: Archer en estat de Dodge

10.8.1.6 MeleeAttack State

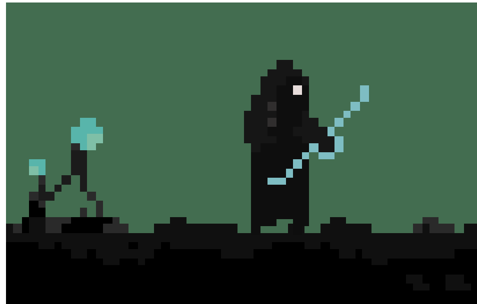


Figura 10.7.13: Archer en estat de MeleeAttack

10.8.1.7 RangeAttack State

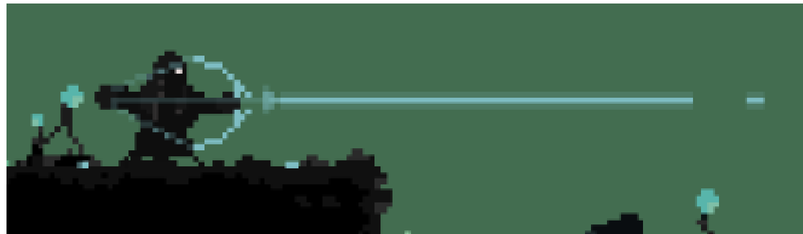


Figura 10.7.14: Archer en estat de RangeAttack

10.8.1.8 Dead State

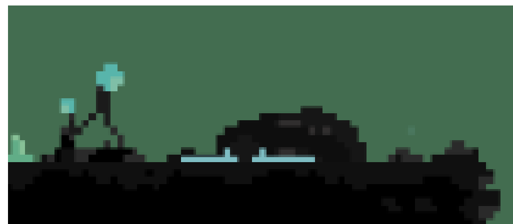


Figura 10.7.15 : Archer en estat de Dead

10.9 Checkpoints

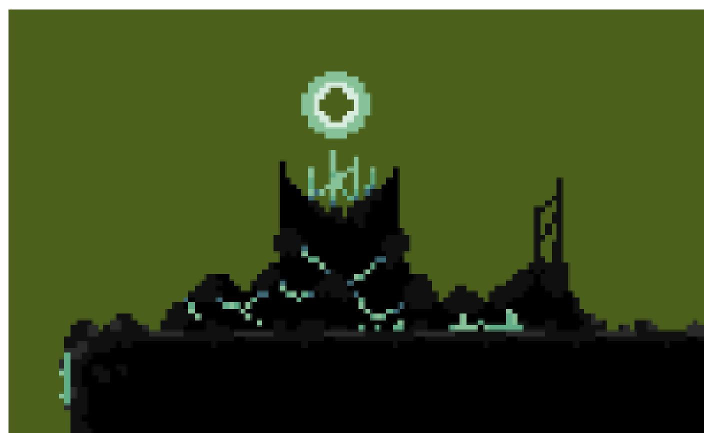


Figura 10.8.1: Checkpoint desactivat

Figura 10.8.2: Checkpoint activat

11.
Conclusions

Durant l’elaboració del projecte hem arribat a les conclusions següents:

- És molt fàcil al principi pensar en gran i tenir moltes idees per al videojoc. A l’hora de la veritat, però, de totes aquelles idees que un ha tingut en podrà implementar tant sols unes quantes ja que no es disposa de temps il·limitat i les tasques es poden complicar, i per tant, allargar. És per això que hem vist que és molt important una primera fase de disseny i organització.

- Si es volgués realitzar un projecte més gran, seria necessari més personal. Per a crear un videojoc 2D semblant al nostre serien necessaris especialistes de diferents àmbits (artístic, disseny, programació, musical, etc).

- Finalment, aquest projecte ha estat el més gran que hem realitzat mai, la vegada que entretingut. Hem pogut assolir els nostres objectius que teniem inicialment pensats i tenim ganes de seguir amb aquest projecte per a fer-lo més professional.

12.
Treball futur

Un cop hem acabat el projecte, hem pogut veure que hi ha apartats d'aquest que o bé ens hauria agradat fer-los d'una altra manera o bé que es podrien ampliar. Aquest és un llistat de les possibles futures tasques a realitzar per el videojoc:

- **Creació de més enemics diferents amb més estats**: al haver creat els enemics a partir d'una màquina d'estats, s'obre un món de possibilitats. Nous enemics, nous atacs, enemics amb escuts, accions un cop morts, etc.

- **Implementació d'un atac a distància per al jugador**: actualment el jugador només disposa d'una atac (atac amb espasa). Seria interessant que també pogués atacar a distància.

- **Creació d'una història per al videojoc**: hem deixat de banda la part més narrativa del videojoc. Es podria pensar una bona història i adaptar el que faci falta per a què lligui amb aquesta.

- **Creació de més nivells amb músiques diferents**: al fer que l'entorn s'adapti a la música, es podrien realitzar més músiques amb les seves peculiaritats i personalitzades per nivell.

- **Sincronització automàtica de la música amb el gameplay:** actualment la sincronització de la música amb el gameplay no és del tot automàtica. Al fer la detecció d'instants de temps per marcadors, hi ha hagut d'haver una feina manual de posar-los a la línia temporal del projecte a l'FMOD Studio. Un possible treball futur podria ser fer un anàlisi de l'espectre de freqüències de la música per a determinar quins són els pics importants i a partir d'allà trobar els instants de temps clau. Al principi aquesta va ser la nostra idea però tota la documentació i projectes de demostració estaven molt anticuats i per tant desactualitzats.

- **Afegir sons al joc**: hem deixat de banda una mica la part sonora del joc. Creiem que és força important i per això un bon treball futur seria buscar sons o crear-los nosaltres per a molts aspectes del videojoc (atacs, enemics, mal, boss, etc.).

13.
Bibliografia

Wikipedia. (2022). *Motores de videojuegos.*
https://es.wikipedia.org/wiki/Motor_de_videojuego

Wikipedia. (2022). *Unity*.
https://es.wikipedia.org/wiki/Unity_(motor_de_videojuego)

Wikipedia. (2022). *Unreal Engine*.
https://es.wikipedia.org/wiki/Unreal_Engine

Wikipedia. (2022). *Godot*.
https://es.wikipedia.org/wiki/Godot

Unity. (2022) *Unity*.
https://unity.com/es

Itch.io. (2022). *Sprites*.
https://itch.io/

Youtube. (2022). *Boss Fight*.
https://www.youtube.com/watch?v=PaT_43fmT_k&list=PLRpO-g5Y98AysUpFP7gNMjbBZ53LKn7gQ&index=20&t=104s&ab_channel=BravePixelG

Youtube. (2022). *Complex Enemy Behaviour using Finite State Machines.*
https://www.youtube.com/watch?v=UeNKI5HZI3o&list=PLy78FINcVmjA0zDBhLuLNL1Jo6xNMMq-W&index=14&ab_channel=Bardent

Youtube. (2021). *Color Swapping Shader Graph.*
https://www.youtube.com/watch?v=I-a2IRPyzys.

Youtube. (2021).*Tilemaps 2D*.
https://www.youtube.com/watch?v=ryISV_nH8qw

gameprogrammingpatterns.com. (2021). *Game Programming Patterns*.
https://gameprogrammingpatterns.com/introduction.html

14.
Annexos

Al ser una memòria força extensa, hem preferit que el nostre Annex sigui el propi projecte sencer del Unity. En aquest projecte s'hi poden trobar tots els Assets que hem creat, buscat i fet servir durant la realització del projecte.

Inclou fitxers de codi, sprites, animacions, prefabs, etc.

Hem organitzat la carpeta del projecte per a què s'identifiquin fàcilment els diferents tipus de recursos:

- Els scripts es localitzen a la carpeta: *Assets/Scripts*
- Els sprites es localitzen a la carpeta: *Assets/Sprites*
- Els prefabs es localitzen a la carpeta: *Assets/Prefabs*
- Les animacions es localitzen a la carpeta: *Assets/Animations*
- Les escenes es localitzen a la carpeta: *Assets/Scenes*

15.
Manual d’usuari

15.1 Iniciar el joc

Per iniciar el joc cal obrir la carpeta del projecte i executar el fitxer executable “Komorebi.exe”. Un cop iniciat el joc es mostrarà el logotip de Unity i s’iniciarà el joc, mostrant-se el menú principal.

15.2 Controls

- **Moviment horitzontal:** El jugador es mou cap a la dreta i l’esquerra amb les tecles AD.

- **Moviment en parets**:
    - Quan el jugador estigui agafat a una paret, pot utilitzar la tecla W per escalar cap amunt.
    - Per tal de passar a deslliçar-se per la paret, pot utilitzar la tecla S o bé deixar anar la tecla Control i prémer la tecla de moviment corresponent a la direcció on es trobi la paret.
    - Si es prem la tecla Space mentres s’està a una paret, el personatge saltarà cap a la direcció oposada a aquesta.

- **Salt:** Tecla Space.

- **Dash:** S’inicia el dash amb la tecla Shift. Amb el cursor del ratolí s’elegeix la direcció cap a on es vol fer el dash. El dash no es realitzarà fins que es deixi anar la tecla Shift, o quan hagi passat el temps màxim.

- **Atac:** Clic esquerra del ratolí.

- **Pausa:** Tecla Esc.

15.3 Objectiu del joc

L’objectiu principal del joc és arribar al final del nivell superant els obstacles i sobrevivint diferents enemics que van apareixent pel camí, per tal d’arribar finalment a l’última zona en la qual s’ha de derrotar al *Boss* final.

15.4 Menú d’opcions

Quan es posa el joc en pausa, es pot accedir al menú d’opcions, a través del qual es pot configurar el volum de la música i dels efectes de so.