Universitat de Girona
Escola Politècnica Superior

Projecte fi de grau

Estudi: Grau en Enginyeria Informàtica

Títol: Creació d'un joc d'estil Metroidvania 2D amb perspectiva lateral

Document: Memòria

Alumnes: Judit Quintana i Wilber Bermeo

Tutor: Gustavo Patow
Departament: IMAE
Àrea: LSI

Convocatòria (mes/any): Setembre 2022

Universitat de Girona
Escola Politècnica Superior

PROJECTE FI DE GRAU

Creació d'un joc d'estil Metroidvania 2D amb perspectiva lateral

Autors:
Judit QUINTANA MASSANA
Wilber BERMEO QUITO

Setembre 2022

Grau en Enginyeria Informàtica

Tutor:
Gustavo PATOW

Agraïments

A totes aquelles persones que ens han ajudat a l'hora de realitzar aquest projecte.

Al nostre tutor el Doctor Gustavo Ariel Patow, pel seu suport i ajut, i per acompanyar-nos durant tota la trajectòria del projecte.

Al professor Francesc Xavier Costa Brugue per ajudar-nos a resoldre els matisos artístics del nostre projecte.

A les nostres amistats i els nostres companys de la universitat pel seu interès i per alleugeriment de la feixuguesa del camí que ha suposat aquest treball.

A tots els betatesters, per destinar les seves hores i els seus esforços a donarnos crítiques constructives que ens han ajudat a millorar i engrandir projecte.

I com no, a les nostres famílies, pel seu estímul, recolzament i estima inqüestionable. Per haver hagut d'aguantar la nostra pitjor cara en els moments més durs d'aquest viatge.

A tots vosaltres, aquest projecte és el que és gràcies a vosaltres.

Índex

1 Introducció **1**
1.1 Motivacions . . . 3
1.2 Proposit . . . 3
1.3 Objectius . . . 3
1.4 Separació de tasques . . . 6

2 Viabilitat **9**
2.1 Viabilitat tecnològica . . . 9
2.1.1 Hardware . . . 9
2.1.2 Software . . . 10
2.2 Viabilitat legal . . . 10
2.3 Inversió inicial . . . 10
2.4 Recursos humans . . . 11
2.5 Estudi del mercat . . . 13
2.5.1 Metodologia utilitzada per cercar la competència . . . 13
2.5.2 Resultats obtinguts . . . 13
2.5.3 Anàlisi dels resultats obtinguts . . . 16
2.5.4 Resultats . . . 17

3 Metodologia **19**
3.1 Metodologies de gestió de projectes . . . 19
3.1.1 Waterfall . . . 19
3.1.2 Scrum . . . 19
3.1.3 Kanban . . . 20
3.2 Metodologia escollida . . . 20
3.2.1 Diagrama de flux de la metodologia . . . 21

4 Planificació **23**
4.1 Planificació prèvia . . . 23
4.2 Tasques planificades . . . 23
4.2.1 Planificació del joc . . . 24
4.2.2 Estudi del diferents motors . . . 24
4.2.3 Estudi del programari . . . 24
4.2.4 Cerca, creació i preparació d'elements gràfics . . . 24
4.2.5 Disseny i implementació d'un protoip amb Unity . . . 24
4.2.6 Disseny i implementació del joc . . . 24
4.2.7 Disseny i implementació de l'interficie d'usuari . . . 25

4.2.8 Disseny i implementació de IA 25
4.2.9 Cerca i integració de música 25
4.2.10 Verificació i proves 25
4.2.11 Creació d'una demo del videojoc 25
4.2.12 Documentació 25
4.3 Temps estimat 26
4.4 Resultats esperats 30

5 Marc de treball i conceptes previ 31
5.1 Motors de videojocs 33
5.1.1 Unreal Engine 33
5.1.2 Unity 3D 34
5.1.3 Godot 35
5.1.4 CryENGINE 36
5.2 Motor triat 37

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

7 Estudi i decisions 43
7.1 Sistema Operatiu 43
7.2 C# 44
7.3 Visual Studio Code 44
7.3.1 C# for Visual Studio Code 45
7.3.2 Unity Code Snippets 45
7.4 Action 45
7.5 Unity 45
7.5.1 Component 45
7.5.2 Monobehavior 46
7.5.3 Game Object 47
7.5.4 Prefab 48
7.5.5 Tag 48
7.5.6 Transform 48
7.5.7 Collider 48
7.5.8 Layer 48
7.5.9 Rigidbody 48
7.5.10 Layer Collision Matrix 49
7.5.11 Sorting Layer 49

7.5.12 Sorting Group 49
7.5.13 Camera 50
7.5.14 Cine Machine 50
7.5.15 Canvas 50
7.5.16 Audio Source 50
7.5.17 Audio Clip 51
7.5.18 Audio Listener 51
7.5.19 Scriptable Object 51
7.5.20 Light 2D 52
7.5.21 Scene Manager 53
7.5.22 Editor 54
7.6 DOTween 56
7.7 Behavior Designer 56
7.7.1 Task 57
7.7.2 Task Status 58
7.7.3 Task API 59
7.7.4 Parent Task API 60
7.7.5 Variables 61
7.7.6 Conditional Aborts 62
7.7.7 Behavior Trees o Finite State Machines 62
7.8 Ilustrator 64
7.9 Spine 64
7.10 Git 66
7.11 GitHub 66
7.11.1 Issues 66
7.11.2 Pull Request 67
7.11.3 GitHub Projects 67
7.11.4 GitHub Pages 69
7.11.5 GitHub Actions 69
7.12 Diagrams.net 75
7.13 LaTeX 75

8 Anàlisi i disseny del sistema **77**
8.1 Descripció general 77
8.2 Història i ambientació 78
8.3 Casos d'ús 78
8.3.1 Diagrames de casos d'ús 78
8.3.2 Diagrama d'activitats 81
8.4 Model de clases 87
8.4.1 Personatge principal 87
8.4.2 Interacció de l'escena amb el personatge principal 89

8.4.3 Disseny del enemics 90
8.4.4 Disseny de classes de gestió de sessió 92
8.5 Disseny del funcionament 93
8.5.1 Personatge principal 93
8.5.2 Enemics 98
8.5.3 NPC 111
8.5.4 Objectes d'escena interactuable 115
8.5.5 Interfícies d'usuari 121
8.6 Estil i art conceptual 127
8.6.1 Estil 127
8.6.2 Art conceptual 130
8.7 Diseny del mapa 137
8.7.1 Illa Omed 138
8.7.2 Illa dels Homes 142

9 Implementació i proves 147
9.1 Personatge principal 147
9.1.1 Córrer 147
9.1.2 Desplaçament ràpid 148
9.1.3 Salt 149
9.1.4 Gestió de la seqüència de punys 153
9.1.5 Gestió de l'habilitat de raig 153
9.1.6 Curació 154
9.1.7 Rebre mal 155
9.1.8 Protecció 156
9.1.9 Horientació del personatge 157
9.2 Interacció de l'escena amb el personatge. 158
9.2.1 Component Hazard 158
9.2.2 Component DeadlyObject 160
9.3 Lluita per fases 162
9.4 Tasques 166
9.4.1 Detecció de canvi de fase 166
9.4.2 Detecció de l'orientació del jugador en l'escena 166
9.4.3 Orientar la IA cap al jugador 167
9.4.4 Llançament de projectils 168
9.4.5 Destrucció de la intel·ligència artificial 169
9.5 Gestió de recursos dintre de l'escena 172
9.5.1 Gestió dels cadàvers dels enemics 175
9.6 Gestor de la sessió de joc 177
9.7 Punts de control 181
9.8 Gestió de canvis d'escena 183

9.9 Control de càmera . . . 187
9.10 Tilemap . . . 189
9.11 Contigut d'escena per capes . . . 192
9.12 Il·luminació d'escenes . . . 194
9.12.1 Instal·lació i preparació . . . 195
9.12.2 Tipus d'il·luminació. . . . 197
9.12.3 Gestió de l'il·luminació . . . 198
9.13 Control de música i efectes sonors . . . 200
9.13.1 Component BackgroundMusicPlayer . . . 200
9.13.2 Component SoundManager . . . 202
9.13.3 Component UIVolumeSettings . . . 203
9.14 Gestió de guardat i càrrega de dades . . . 204
9.15 Plataformes i portes dinàmiques . . . 206
9.15.1 Plataformes bàsiques . . . 206
9.15.2 Plataformes que s'activen amb palanques . . . 208
9.16 Interfícies d'usuari . . . 212
9.16.1 Menús . . . 212
9.16.2 In-Game Canvas . . . 220
9.17 Gestió de diàlegs en NPCs . . . 237
9.18 Cinematogràfiques . . . 242

10 Implantació i resultats 245
10.1 Legislació i normativa vigent . . . 245
10.2 Personatge principal . . . 246
10.2.1 Desplaçament . . . 246
10.2.2 Desplaçament ràpid . . . 247
10.2.3 Salt i Aterratge . . . 248
10.2.4 Atac . . . 249
10.2.5 Curació . . . 250
10.2.6 Dany i protecció . . . 251
10.3 Interacció de l'escena amb el personatge . . . 252
10.4 Lluita per fases . . . 253
10.4.1 Apache Pig . . . 253
10.4.2 FALSE Knight . . . 254
10.5 Tasques . . . 255
10.5.1 Orientar la IA cap el jugador . . . 255
10.5.2 Llançament de projectils . . . 256
10.5.3 Destrucció de la intel·ligència artificial . . . 257
10.6 Gestió de recursos dintre de l'escena . . . 258
10.7 Punt de guardat . . . 259
10.8 Punts de control . . . 260

10.9 Canvi d'escena 261
10.10 Càmera 262
10.11 Tilemap 263
10.12 Contigut d'escena per capes 264
10.13 Il·luminació d'escenes 265
10.13.1 Il·luminació per ressaltar elements 265
10.13.2 Il·luminació per ressaltar elements perillosos 266
10.13.3 Il·luminació per elements decoratius 267
10.14 Plataformes 268
10.14.1 Plataformes simples 268
10.14.2 Plataformes dinàmiques 269
10.15 Portes 270
10.16 Interficies d'usuari 271
10.16.1 Menus 271
10.16.2 In-Game Canvas 274
10.16.3 Diàlegs 278
10.17 Cinematografíes 280
10.18 Illa Omed 282
10.19 Illa dels Homes 286

11 Conclusions **293**

12 Treball futur **297**

Bibliografia **299**

Annexos **303**

Manual d'Usuari **305**

Índex de figures

1.1 Principals mercats de videojocs, ingresos en millions. 2021 2

2.1 Captura in-game Hollow Knight 13
2.2 Captura in-game Guacamelee! 14
2.3 Captura in-game Owlboy 14
2.4 Captura in-game Night in the woods 15
2.5 Captura in-game Celeste 15
2.6 Captura in-game The Legend of Zelda: The Wind Waker 16

3.1 Diagrama de flux de la metodologia 22

4.1 Estudi esperat de la planificació del projecte 27
4.2 Planificació resultant del projecte 29

5.1 Esquema d'un motor de videojocs. 32
5.2 Logo de Unreal Engine. 33
5.3 Logo de Unity. 34
5.4 Logo de Godot. 35
5.5 Logo de CryENGINE. 36
5.6 Logo de Unity. 37

6.1 Interacció del jugador amb el sistema. 39

7.1 Logo Windows 10. 43
7.2 Logo Visual Studio Code. 44
7.3 Cicle de vida d'un component. 47
7.4 Matriu de col·lició de capes del projecte. 49
7.5 Sorting Group per donar profunditat a les escenes. 50
7.6 Comparativa de espais de colors. 52
7.7 Editor d'Unity. 54
7.8 Exemple d'un arbre de comportament en Behavior Tree. 57
7.9 Cicle de vida d'una tasca. 60
7.10 Exemple d'animació per ossos en Spine. 64
7.11 Personatge, separat per parts, llest per ser animat amb Spine. . . . 65
7.12 Exemple GitHub Projects 68

8.1 Diagrama de casos d'ús del menú d'inici. 79
8.2 Diagrama de casos d'ús d'inici de sessió. 79
8.3 Diagrama de casos d'ús dintre del joc. 79

8.4 Diagrama de casos d'ús del menú de joc. 80
8.5 Diagrama de casos d'ús menú de configuracions. 80
8.6 Diagrama d'activitats d'ús d'habilitats. 81
8.7 Diagrama d'activitats d'interacció amb objectes que fan mal. . . . 82
8.8 Diagrama d'activitats de fases d'un Boss. 83
8.9 Diagrama d'activitats de diàleg. 84
8.10 Diagrama d'activitats moviment plataforma simple. 85
8.11 Diagrama d'activitats de música. 86
8.12 Model de clases del personatge principal. 87
8.13 Model de clase de dades de l'estat del personatge principal. 88
8.14 Model de classes per fer mal al personatge. 89
8.15 Diagrama de classes dels enemics. 90
8.16 Diagrama de classes de la lògica de control de sessió de joc. . . . 92
8.17 Ajax. 93
8.18 Màquina d'estats d'animacions de l'Ajax. 94
8.19 Sequencies atac principal. 94
8.20 Desplaçament normal. 95
8.21 Desplaçament per salt. 95
8.22 Desplaçament per habilitat adquirida. 96
8.23 Llançament d'un projectil com habilitat. 96
8.24 El personatge rep mal i entra a fase de recuperació. 97
8.25 Curació després d'ingerir fruita de l'arbust. 97
8.26 Mort i pèrdua de control sobre el personatge. 98
8.27 Versió blava d'un Crawl Slime. 98
8.28 Versió rosada d'un Crawl Slime. 99
8.29 Versió verda d'un Crawl Slime. 99
8.30 Versió blava d'un Wheeler Slime. 100
8.31 Versió rosada d'un Wheeler Slime. 100
8.32 Grup de Green Wheeler Slime. 101
8.33 Grup de Worm Slime. 101
8.34 Worm Slime explotant. 102
8.35 Sequencia d'atac de la planta. 102
8.36 Mosca bàsica 103
8.37 Grup de mosques petites. 103
8.38 Apache Pig. 104
8.39 Màquina d'estats d'animacions del Boss. 105
8.40 Arbre de comportament del Boss. 105
8.41 Apache Pig fent una envestida. 106
8.42 Apache Pig saltant sobre el jugador. 106
8.43 Apache Pig instanciant una roca que travessa a Ajax. 107
8.44 Sequencia atac principal. 107

8.45 Apache Pig saltant sobre el jugador. 108
8.46 Mort del Boss. 108
8.47 FALSE Knight. 109
8.48 Màquina d'estats d'animacions. 109
8.49 Arbre de comportament. 110
8.50 Inici de la lluita amb el FALSE Knight. 110
8.51 Mort del Boss. 111
8.52 El Capità Java. 111
8.53 Màquina d'estat del Capità Java. 112
8.54 El Capità Java en el joc. 112
8.55 Cassandra. 113
8.56 Altaveu de comunicació de la Cassandra. 113
8.57 Màquina d'estat del la Cassandra. 114
8.58 Retrobament entre la Cassandra i l'Ajax. 114
8.59 Objectes ànima 115
8.60 Líquids. 115
8.61 Palanca. 116
8.62 Porta que s'activa amb palanca. 116
8.63 Plataforma estàtica. 117
8.64 Plataforma dinàmica amb moviment vertical. 117
8.65 Plataforma dinàmica amb moviment lateral. 117
8.66 Plataforma dinàmica amb palanca. 118
8.67 Fruites de curació. 118
8.68 Tresor. 118
8.69 Estàtua de guardat. 119
8.70 Trampolí. 119
8.71 Punt de control. 120
8.72 Menú d'inici. 121
8.73 Menú d'inici de sessió. 121
8.74 Menú de configuració. 122
8.75 Menú de mapatge de tecles. 122
8.76 Interfície de càrrega. 123
8.77 Menú de joc. 123
8.78 Notificació de nova habilitat. 124
8.79 Interfície de joc. 124
8.80 S'informa que prement la teclas *S* es pot consumir una fruita. . . . 125
8.81 Diàleg de Cassandra. 125
8.82 Interfície de joc acabat. 126
8.83 Estil 1. 127
8.84 Estil 2. 128
8.85 Estil 3. 128

8.86 Estil 4. 128
8.87 Estil 5. 129
8.88 Estil 6. 129
8.89 Estil 7. 129
8.90 Dissenys inicials de l'Ajax. 131
8.91 Disseny final de l'Ajax. 131
8.92 Dissenys inicials de la Cassandra. 132
8.93 Disseny final de la Cassandra. 132
8.94 Concepte artístic de Cassandra i Mr. C. 132
8.95 Dissenys inicials del Capità Java. 133
8.96 Disseny Final del Capità Java. 133
8.97 Disseny inicial d'en Jason. 134
8.98 Disseny final d'en Jason. 134
8.99 Disseny inicial d'Apache Pig. 135
8.100 Disseny final d'Apache Pig. 135
8.101 Dissenys inicials dels diferents menús. 136
8.102 Dissenys finals dels menus. 136
8.103 Llegenda d'escena. 137
8.104 Visualització general de la illa Omed. 138
8.105 Illa Omed, escena 1. 139
8.106 Illa Omed, escena 2. 139
8.107 Illa Omed, escena 3. 140
8.108 Illa Omed, escena 4. 140
8.109 Illa Omed, escena 5. 141
8.110 Illa Omed, escena 6. 141
8.111 Visió general de la illa dels Homes. 142
8.112 Illa dels Homes, escena 1. 143
8.113 Illa dels Homes, escena 2. 143
8.114 Illa dels Homes, escena 3. 143
8.115 Illa dels Homes, escena 4. 144
8.116 Illa dels Homes, escena 5. 144
8.117 Illa dels Homes, escena 6. 144
8.118 Illa dels Homes, escena 7. 145
8.119 Illa dels Homes, escena 8. 145
8.120 Illa dels Homes, escena 9. 146

9.1 Diagrama d'activitats del salt personatge principal. 152
9.2 Collider que detecta sobreposició amb el personatge principal. . . 160
9.3 Component Hazard afegit a ApachePig per fer mal per interacció. 160
9.4 Exemple d'arbre de comportament simple. 162
9.5 Nou comportament en arbre simple. 163

9.6 Arquitectura d'arbre de comportament per IA complexa. 165
9.7 Components d'un punt de control. 181
9.8 Posicionament de l'objecte SceneTeleporter a l'escena. 184
9.9 Posicionament de l'objecte SceneEntrance a l'escena. 185
9.10 Videojocs famosos que utilitzen el disseny per tilemaps. 189
9.11 Peces de mosaic que hem utilitzat dels paquets de Kenney Assets. 189
9.12 Utilització de les peces de Kenney Assets per crear escenaris diversos. 190
9.13 Escena amb sols el base tilemap actiu. 190
9.14 Exemple de tilemap decoratiu. 191
9.15 Escena amb sols el tilemaps de dany actius. 191
9.16 Escena amb tots els tilemaps actius. 191
9.17 Les capes definides en el projecte 192
9.18 Sistema d'ordenació dels objectes visuals de l'escena segons la profunditat desitjada. 193
9.19 Ombra blanquinosa envolta al personatge principal 194
9.20 Fons desenfocat per ressaltar la resta de l'escena 195
9.21 Formes amb degradats 195
9.22 Crear una pipleline i assignar-la a les configuracions de gràfics d'Unity. 196
9.23 Creació i assignació d'un render 2D a una pipline. 196
9.24 Canvi de material automàtic per l'escena o projecte. 197
9.25 Diferents intensitats a la Global Light en una escena. 198
9.26 Free form light bàsica en escena. 198
9.27 Free form light per donar profunditat a les coves. 199
9.28 Enemic amb i sense efecte lluminós. 199
9.29 Objecte decoratiu amb i sense efecte lluminós 199
9.30 Trigger que activa el moviment de la plataforma 207
9.31 Sprites contenidors de vida. 221
9.32 Efecte de construcció. Tom & Jerry | Monster Jerry 228
9.33 Sprite utilitzat per l'ombra 230
9.34 Sprites utilitzats per el cronometre de les habilitats. 232

10.1 Personatge principal en repos. 246
10.2 Personatge principal corrent. 246
10.3 Personatge principal fent desplaçament ràpid. 247
10.4 Personatge principal danyant els enemics mentre fa el desplaçament ràpid. 247
10.5 Personatge principal saltant. 248
10.6 Personatge principal aterrant. 248
10.7 Personatge princiapl fent sequència d'atac bàsic. 249

10.8 Personatge principal instanciant raig. 249
10.9 Efecte de curació sobre el personatge principal. 250
10.10 Personatge principal rep mal. 251
10.11 Personatge principal en estat de recuperació. 251
10.12 Personatge interaciona amb un component Hazard. 252
10.13 Personatge interaciona amb un component DeadlyObject. 252
10.14 Apache Pig en les diferents fases. 253
10.15 Apache pig canviant de fase. 253
10.16 FALSE knight en les diferents fases. 254
10.17 FALSE knight en el canvi de fase. 254
10.18 Whorm seguint el jugador amb l'orientació. 255
10.19 False Knight seguint el jugador amb l'orientació. 255
10.20 Planta llançant un projectil. 256
10.21 Apache Pig llençant un projectil. 256
10.22 Destrució IA Crawler per falta de vida. Deixa un sprite. 257
10.23 Destrució IA Apache Pig per falta de fases. Roman a l'escena l'ultim frame de l'animació de mort. 257
10.24 Destrució IA FALSE Knight per falta de fases. Produeix un efecte de partícules. 257
10.25 Exemples gestió de cadavers. 258
10.26 Activació d'un punt de guardat. 259
10.27 Sequència de retorn a un punt de control. 260
10.28 Sequència de com Ajax entra a l'escena. 261
10.29 Escena de càrrega. 261
10.30 Configuració càmera dins del joc. 262
10.31 Càmera afectada pel seu rang. 262
10.32 Exemple escena amb tilemap base i tilemap decoratiu pel fons. . . 263
10.33 Exemple escena amb tilemap base i tilemap decoratiu pels objectes. 263
10.34 Exemple escena amb tilemap base, tilemap decoratiu pel fons i tilemap que danya el jugador. 263
10.35 Distrubucció de capes visuals en Omeled Island. 264
10.36 Distrubucció de capes visuals en Men Island. 264
10.37 Il·luminació player. 265
10.38 Il·luminació a Capità Java. 265
10.39 Il·luminació de la lava. 266
10.40 Il·luminació d'un enemic. 266
10.41 Il·luminació que emfatitza l'entrada al laboratori. 267
10.42 Plataformes estatiques. 268
10.43 Plataforma dinàmica amb moviment horitzontal. 268
10.44 Plataforma dinàmica amb moviment vertical. 268
10.45 Plataforma mecànica. 269

10.46 Porta mecànica. . . . 270
10.47 Menu principal i submenu per escollir partida. . . . 271
10.48 Menu de pausa. . . . 272
10.49 Menu de configuració. . . . 273
10.50 Menu del mapatge de tecles. . . . 273
10.51 Visualització de l'estat de vides i habilitats del personatge principal. 274
10.52 Efecte de perdre vida. . . . 275
10.53 Efecte reflex. . . . 275
10.54 Barra de vida amb una sola vida. . . . 275
10.55 Barra de vida sense vides. . . . 275
10.56 Panell d'habilitats sense el dash, amb el dash i amb el dash en cooldown. . . . 275
10.57 Notificació. . . . 276
10.58 Notificació gran. . . . 276
10.59 Consell d'interacció. . . . 277
10.60 Capità Java en Men Island. . . . 278
10.61 Capità Java en Omeled Island. . . . 278
10.62 Cassandra parlant per altaveu al Laboratori. . . . 279
10.63 Apache Pig empeny a Ajax. . . . 280
10.64 Retrobomanent amb Cassandra. . . . 281
10.65 Crèdits. . . . 281
10.66 Visualització general de la illa Omed. . . . 282
10.67 Illa Omed, scena 1. . . . 283
10.68 Illa Omed, scena 2. . . . 283
10.69 Illa Omed, scena 3. . . . 283
10.70 Illa Omed, scena 4. . . . 284
10.71 Illa Omed, scena 5. . . . 284
10.72 Illa Omed, scena 6. . . . 285
10.73 Visualització general de l'illa dels Homes. . . . 286
10.74 Illa dels Homes, scena 1. . . . 287
10.75 Illa dels Homes, scena 2. . . . 287
10.76 Illa dels Homes, scena 3. . . . 288
10.77 Illa dels Homes, scena 4. . . . 288
10.78 Illa dels Homes, scena 5. . . . 289
10.79 Illa dels Homes, scena 6. . . . 289
10.80 Illa dels Homes, scena 7. . . . 290
10.81 Illa dels Homes, scena 8. . . . 290
10.82 Illa dels Homes, scena 9. . . . 291
10.83 Illa dels Homes, scena 10. . . . 291

11.1 Gràfica de les valoracions 294

1 Tecles utilitzades per moure's dins del joc 305

Índex de taules

2.1 Característiques del maquinàri 9
2.2 Cost per maquinàri 9
2.3 Cost per programari 10
2.4 Inversió inicial 10
2.5 Salari per perfil tècnic 12
2.6 Aproximat d'inversió d'una empresa 12
2.7 Taula de característiques dels jocs del mercat. 17
2.8 Taula de característiques del projecte. 17

4.1 Taula de tasques 26

6.1 Compatibilitat WebGL en navegadors. 41

11.1 Mitjana de les valoracions dels 15 enquestats. 294

Capítol 1

Introducció

Arquitectura, escultura, pintura, música, literatura, dansa i cinema… Aquestes són les considerades set belles arts del món. La humanitat ha avançat de tal grau que les arts s'han expandit, evolucionat i redefinit al llarg dels segles des de les prehistòriques pintures rupestres fins al cinema contemporani.

A partir dels noranta, hi ha el polèmic debat de si els videojocs poden ser considerats el vuitè art. Llavors, quan és que alguna cosa pot ser considerat art? Segons la RAE l'art és defineix com: "manifestació de l'activitat humana mitjançant la qual s'interpreta el real o es plasma l'imaginari amb recursos plàstics, lingüístics o sonors". D'aquesta manera, podríem definir als videojocs com "Artés plàstiques, lingüístiques i sonores interactives", ja que el jugador participa dintre d'un producte que mostra seqüències (plàstiques), lingüístiques (guió) i sonores (bonda sonora).

La significativa evolució dels videojocs ha permès que aquests puguin contenir elements comuns a les set belles arts. O algú pot negar que els jocs actuals no tenen petits indicis de cinema, literatura, pintura o música?

En l'àmbit tècnic, els videojocs han patit una gran evolució des dels seus orígens. L'avenç tecnològic, l'augment del nombre de desenvolupadors i empreses en aquest sector, ha permès crear productes amb mecàniques més complexes, dispositius físics i plataformes innovadores amb les quals interactuar i consumir videojocs com a producte i art.

El mercat dels videojocs està dominat per grans companyies Triple A[1], amb produccions enormes en quant a qualitat i inversió econòmica. Però això no implica que no existeixi un gran nombre de desenvolupadors o companyies *indies*, o també coneguts com a desenvolupadors o companyies independents, les quals disposen de menys personal i menys capital, però que a la vegada poden generar productes d'altíssima qualitat amb grans innovacions quant a l'estil, jugabilitat entre altres.

[1]Classificació informal utilitzada per als videojocs produïts i distribuïts, típicament tenint màrqueting i desenvolupament d'alt pressupost.

Espanya es troba entre els deu mercats que més consumeixen i inverteixen diners en el món del videojoc, ocupant la posició desena, després d'Itàlia i Canadà i com a punters en la llista, Xina i Estats Units. Veure figura 1.1. [statista 022]

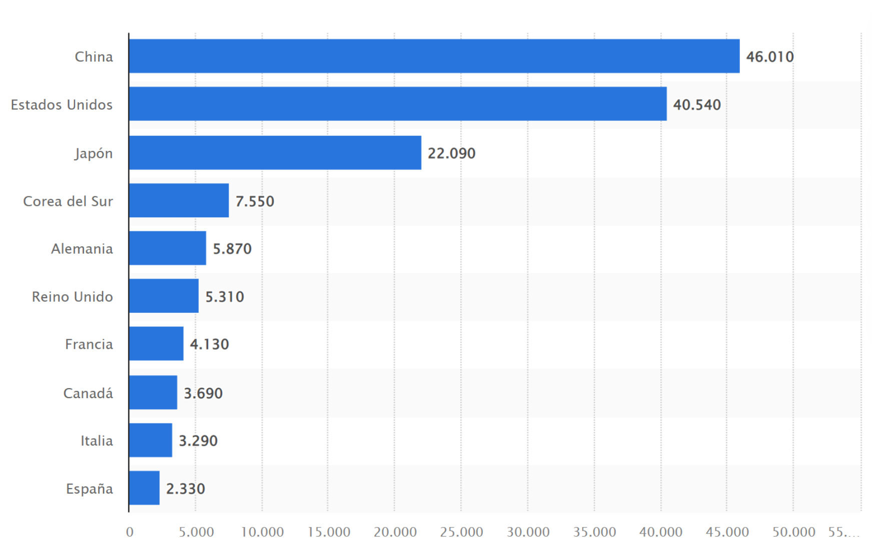

Figura 1.1: Principals mercats de videojocs, ingresos en millions. 2021

La pandèmia i l'aïllament social, van impulsar el consum dels videojocs a escala global. Tot i així, la pandèmia ha fet que molts videojocs es posposessin fins aquest any 2022, el que significa que per aquest i els pròxims anys, s'estima que el valor generat per aquesta indústria sigui encara més gran.

El nostre projecte és una producció de caràcter *indie*, un videojoc basat en el gènera Metroidvania, el qual incorpora elements de plataformes, combat i la base de qualsevol videojoc Metroidvania, l'exploració amb limitacions segons el disenyador del videojoc, que poden ser superades gracies a l'obtenció de noves habilitats, activacións de plataformes, entre altres. D'aquesta manera, moltes vegades el jugador es veu forçat a tornar a zones on ja havía estat per trobar nous camins i formes d'avençar. Generalment, aquest gènere sol donar títols amb desplaçament 2D i vista ortogràfica, encara que sempre hi ha excepcions.

1.1 Motivacions

Rere la creació d'aquest projecte, hi han les ganes de demostrar el que hem après al llarg de la carrera, tècnicament, pel que fa a gestió, i el que és més important que ens ha donat la carrera, que és aprendre a aprendre.

Sabíem que el nostre projecte implicava d'endinsar-se en una sèrie de tecnologies que no es veuen en els nostres estudis, però que, en principi, són tecnologies de fàcil aprenentatge per a un informàtic. També, el desenvolupament d'un videojoc ens podria enriquir en característiques que potser no són el nostre fort.

Els dos hem estat uns grans aficionats als videojocs des de petits: Hem sacrificat hores de son per poder estar més estona davant de la pantalla, hem rebut càstigs per jugar en moments inapropiats i hem sentit una alegria inquantificable en passar-nos aquell nivell que no sabíem superar. Acabar la carrera creant una de les coses que ens havia fet estimar el món dels computadors era un final poètic pels dos.

Per aquest fet, ens feia molta il·lusió poder participar en el desenvolupament d'una creació d'aquest estil, i veure, de primera mà, com es planteja la producció, el desenvolupament, l'entrega del producte final i a la vegada, aprendre o millorar altres qualitats com ara, animació, disseny i guionització d'una història, entre d'altres.

1.2 Proposit

El propòsit d'aquest treball de final de carrera d'Enginyeria Informàtica és la creació d'un videojoc 2D de temàtica *Metroidvania* amb el motor de videojocs Unity. Tenim la intenció de diferenciar-lo de la resta de jocs d'aquesta temàtica utilitzant un univers no connectat entre si: un món format per illes

1.3 Objectius

Per poder complir propòsit del projecte, hem de ser capaços fer efectius els següents objectius:

- Planificació del joc.
- Estudi de les alternatives a les eines finalment seleccionades.

- Utilitzar i treure profit de les característiques d'Unity.
- Dominar un dels llenguatges de programació que suporta Unity, C#.
- Disenyar i crear *sprites*.[2]
- Crear animacions amb ossos[3] i amb *sprites*.
- Cerca de material de domini públic per afegir-los al projecte.
- Crear la mecànica de moviment i d'habilitats del jugador.
- Crear un sistema d'activació d'habilitats adquirides al llarg del joc.
- Crear una diversitat d'enemics amb tècniques de *lifelike AI*.
- Crear escenes que mantinguin la consistència estètica *flat* sense oblidar les característiques pròpies d'un Metroidvania.
- Crear elements interactuables amb el jugador dintre l'escena.
- Crear un sistema de punts de guardats i recuperació d'estat a l'entrar al joc.
- Crear un sistema de menús.
- Crear un sistema de notificacions d'esdeveniments.
- Crear un sistema per indicar el *cooldown* de les habilitats i les vides del personatge.
- Crear un mecanisme d'interacció del personatge amb les escenes i enemics.
- Crear un sistema de so que permeti canviar la música i el soroll de fons, segons l'escena i circumstància de la partida.
- Crear un sistema de regulació de so.
- Encriptar dades de l'estat partida.
- Crear *NPC*[4] interactuables amb un sistema de diàleg.

[2] Conjunt d'imatges que es combinen en un mateix fitxer i que són freqüentment utilitzats en el desenvolupament de videojocs

[3] Mètode d'animació d'objectes que utilitza ossos encadenats en esquelets lineals o ramificats amb relacions principals i secundàries

[4] Un personatge no jugador o personatge no jugable. NPC per la seva sigla en anglès, de non playable character. És un personatge controlat pel director de joc

- Treballar amb eines de gestions de versions.
- Treballar amb eines de gestions de projectes.
- Publicar el videojoc resultant i el codi font.
- Crear un mecanisme de desplegament continuo.
- Documentar el projecte

1.4 Separació de tasques

Al ser un projecte desenvolupat en grup, trobem necesari especificar la repartició de les tasques.

- Judit
  - Disenyar i crear *sprites*.
  - Crear escenes que mantinguin la consistència estètica flat sense oblidar les característiques pròpies d'un Metroidvania.
  - Crear un sistema de menús.
  - Crear un sistema de punts de guardats i recuperació d'estat en entrar al joc.
  - Crear un sistema de punts de control.
  - Encriptar dades de l'estat partida.
  - Crear *NPC* interactuables amb un sistema de diàleg.
  - Crear un sistema de notificacions d'esdeveniments.
  - Crear un sistema per indicar el *cooldown* de les habilitats i les vides del personatge.
- Wilber
  - Crear la mecànica de moviment i d'habilitats del jugador.
  - Crear un mecanisme d'interacció del personatge amb les escenes i enemics.
  - Crear un sistema d'activació d'habilitats adquirides al llarg del joc.
  - Crear una diversitat d'enemics amb tècniques de *lifelike AI*.
  - Crear un sistema de so que permeti canviar la música i el soroll de fons, segons l'escena i circumstància de la partida.
  - Crear un sistema de regulació de so.
  - Crear un mécanisme de desplegament continuo.
- Junts
  - Planificació del joc.
  - Estudi de les alternatives a les eines finalment seleccionades.
  - Utilitzar i treure profit de les característiques d'Unity.
  - Dominar un dels llenguatges de programació que suporta Unity, C#.

- Animar amb ossos i amb *sprites*.
- Cerca de material de domini públic per afegir-los al projecte.
- Crear elements interactuables amb el jugador dintre l'escena.
- Treballar amb eines de gestions de versions.
- Treballar amb eines de gestions de projectes.
- Publicar el videojoc resultant i el codi font.
- Documentar el projecte.

CAPÍTOL 2
Viabilitat

2.1 Viabilitat tecnològica

Pel desenvolupament d'aquest projecte no s'ha necessitat grans despeses. El cos de la infraestructura és inexistent a excepció del desgast per l'ús. Les despeses han estat inversions en programari d'edició d'imatge, animació i el programari utilitzat per generar les intel·ligències artificials.

2.1.1 Hardware

La Taula 2.1 és un breu resum de les característiques del *Hardware* usat pel desenvolupament del projecte.

	Màquina 1	Màquina 2
Processador	AMD Ryzen 5 3600	Intel Core i7-7500U
Memòria	16 GB	8 GB
Gràfica	NVIDIA Geforce GTX 1650	Intel HD Graphics 620

Taula 2.1: Característiques del maquinàri

Tot i això, anem a fer un còmput aproximat de què ens hagués costat si comencéssim complement de zero.

	Màquina 1	Màquina 2
Cost	1100 €	700 €
Total: 1800 €		

Taula 2.2: Cost per maquinàri

La Taula 2.2 ens deixa veure que les màquines són de gammes diferents. Mentre que la màquina 1 és un PC d'escriptori, la màquina 2 és un portàtil. Trobem que maquinàries amb més rendiment faciliten el desenvolupament, sobretot pel que fa a esperes per càrrega, ja que el desenvolupament d'un videojoc implica fer ús de programari complex i pesat. Però de la mateixa manera, podem dir que no és necessari tenir màquina d'alta gamma, com s'ha demostrat al llarg del projecte amb la màquina dos.

2.1.2 Software

La majoria del programari fet servir en el projecte és programari lliure, però no tot. A continuació llistem únicament els que vam haver de comprar llicència per poder utilitzar-los.

	Cost
Spine	65 €
Ilustrator	24 €
Behavior Designer	85 €
Total: 174 €	

Taula 2.3: Cost per programari

L'explicació de què són, perquè serveixen i com s'han fet ús cadascuna de les tecnologies de la ç Taula 2.3 ho trobem al Capítol 7, juntament amb la resta de programari fet servir al llarg del projecte.

2.2 Viabilitat legal

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

2.3 Inversió inicial

En l'hipotètic cas que volguéssim recrear el desenvolupament d'aquest projecte des de zero i volguéssim mantenir les mateixes configuracions en quant a maquinari, programari i recursos externs, ens veuríem forçats a tenir una inversió inicial de 1874 €.

	Cost
Màquinaria	1700 €
Programari	174 €
Recursos externs	0 €
Total: 1874 €	

Taula 2.4: Inversió inicial

2.4 Recursos humans

La creació d'un videojoc típicament esta lligada a un equip amb perfils tècnics varis, que s'intercomuniquen per tal de mantenir una coherència general en el procés de desenvolupament i augmentar els benèfics del producte final, pel que fa a temps d'entregues, difusió, entre d'altres.

Podem llistar els perfils tècnics més essencials com:

- Dissenyador: Encarregat de desenvolupar el concepte del joc, així com el guió, la jugabilitat, mecániques de joc, el disseny d'escenaris i nivells.
- Programador: Implementa el disseny i el funcionament del joc.
- Artista: S'encarrega dels elements visuals del joc, així com definir l'estil gràfic.
- Compositor musical: Es fa càrrec de crear efectes de so i la música del joc.

A causa del caràcter de producció del projecte, recordem que es tracta d'una producció *indi*, aquests rols s'han hagut de repartir entre nosaltres, els creadors del projecte.

El perfil de Compositor musical, malauradament no s'ha pogut suplir, a causa de la nostra falta de coneixement en l'àmbit de la música. Tot i axiò, el nostre projecte té sons i música, i ha estat possible gràcies a plataformes web, on es dona accés de forma gratuïta i amb llicència Creative Commons a elements d'aquests tipus.

Una empresa dedicada al desenvolupament de videojocs molt probablement no trigaria més de tres mesos en fer el desenvolupament un projecte similar al nostre. Així doncs, podríem fer un càlcul aproximat de la inversió d'una empresa contractant als perfils esmentats prèviament, juntament amb un aproximat d'inversió en maquinària i programari com gastos fixs.

El sou mitjà anual dels perfils tècnics de la taula 2.5 l'hem tret de les següents fonts:

- Programador de videojocs [Jobted 022]
- Dissenyador de videojocs [Tokio 022]
- Artista de videojocs [glassdoor 022]
- Compositor musical de videojocs [promociónmusical 022]

	Cost
Programador	32.100 €
Dissenyador	30.000 €
Artista	30.213 €
Compositor musical	28.960 €
Anual: 121.273 €	
Tres mesos: 30.250 €	

Taula 2.5: Salari per perfil tècnic

Suposem un cost fix de maquinària i programari 1.500 € per perfil tècnic.

	Cost
Fixes	6.000 €
Recursos humans	30.250 €
Total: 36.250 €	

Taula 2.6: Aproximat d'inversió d'una empresa

2.5 Estudi del mercat

Abans de començar amb el desenvolupament d'un projecte, és útil i necessari analitzar l'estat del mercat actual. Un estudi de la competència ens permet analitzar si el projecte tindrà alguna cabuda dins del món dels videojocs. Per tal de dur-lo a terme, cal cercar els principals competidors, escollint els que ofereixen característiques semblants a les que volem en el nostre projecte. Així doncs, per acotar aquesta cerca cal fixar-se en elements com la jugabilitat, l'estètica o el gènere, 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 utilitzada per cercar la competència

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

- Metroidvania
- 2D amb prespectiva lateral
- Exploració d'illes
- Estetica flat

Hem fet una cerca utilitzant aquestes característiques per trobar els jocs del mercat que més s'assemblen al que es vol desenvolupar. Dels resultats obtinguts, hem seleccionat els més destacats i els que ofereixen característiques més semblants a la nostra proposta.

2.5.2 Resultats obtinguts

2.5.2.1 Hollow Knight

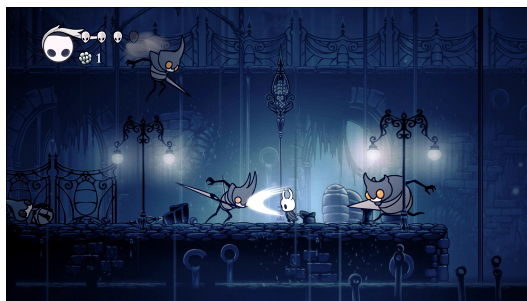

Figura 2.1: Captura in-game Hollow Knight

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.

2.5.2.2 Guacamelee!

Figura 2.2: Captura in-game Guacamelee!

Guacamelee! és un joc de gènere Metroidvania desenvolupat i publicat per DrinkBox Studios. Aquest joc presenta un món obert amb prespectiva 2D i inclou un mode cooperatiu de dos jugadors. És un joc clarament inspirat en mexic i on l'objectiu del protagonista, Juan, és rescatar la filla del president del poble de les mans de Carlos Calaca que la vol sacrificar per fer un ritual que li permetrà pendre el control del món dels morts i dels vius. Utilitza una estètica de dibuix flat.

2.5.2.3 Owlboy

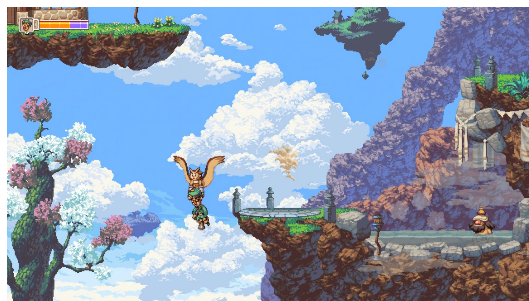

Figura 2.3: Captura in-game Owlboy

Owlboy és un joc de gènere Metroidvania publicat per l'estudi D-Pad. El joc és conegut pel seu llarg desenvolupament que va començar el 2007 i va acabar el 2016. Owlboy té lloc en una sèrie d'illes flotants localitzades al cel. El jugador controla Otus, membre d'una raça híbrida barreja d'humà i mussol. L'objectiu del joc és protegir la comunitat dels atacs dels pirates. El joc és reconegut per la seva bellesa. Fa ús de l'estil PixelArt.

2.5.2.4 Night in the woods

Figura 2.4: Captura in-game Night in the woods

Night in the Woods és un videojoc d'aventura que va ser desenvolupat per Infinite Fall, un estudi fundat pel dissenyador de videojocs Alec Holowka, i l'animador i il·lustrador Scott Benson. Molts, més que un videojoc, el consideren una historia en format gràfic. L'argument narra les històries d'una gata antropomòrfica anomenada Mae, qui recentment va abandonar la universitat i ha tornat a la seva ciutat natal per trobar canvis inesperats. Utilitza una estètica de dibuix flat.

2.5.2.5 Celeste

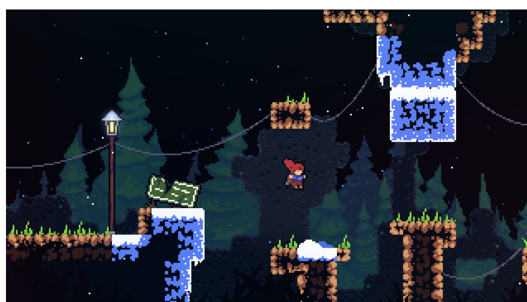

Figura 2.5: Captura in-game Celeste

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 cop més complicats. Un dels aspectes que més s'ha valorat pels jugadors és el moviment del personatge, essent aquest molt fluid, 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. Fa servir una estètica Pixel Art, recordant a l'estil 8-bits.

2.5.2.6 The Legend of Zelda: The Wind Waker

Figura 2.6: Captura in-game The Legend of Zelda: The Wind Waker

The Legend of Zelda: The Wind Waker és un videojoc d'acció-aventura del 2002 desenvolupat per la filial EAD i distribuït per Nintendo. És un dels jocs que compon la famosa saga Zelda. La història del joc es desplega, per primera vegada a la sèrie, en un arxipèlag d'un vast oceà. El jugador controla a Link, que lluita contra el malvat Ganondorf pel control d'una relíquia sagrada coneguda com la Triforça. A la major part del joc, el personatge navega pel mar, viatja entre illes i travessa masmorres i temples per obtenir el poder necessari amb què enfrontar a Ganondorf. L'aventura comença una vegada que la seva petita germana és segrestada per un enorme ocell que arriba a l'illa on l'heroi habita. Utilitza una estètica Chibi.

2.5.3 Anàlisi dels resultats obtinguts

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

- **Dimensió**
  Joc està desenvolupat en 3D o en 2D. (2D/3D)

- **Metroidvania**
  Si es pot incloure dintre el gènere Metroidvania. (SI/NO)

- **Combat**
  Si presenta elements de combat. (SI/NO)

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

- **Exploració**
  Si l'exploració es basa en un mon format per diferentes illes. (SI/NO)

Joc	Dimensió	Metroidvania	Combat	Estil gràfic	Exploració
Hollow Knight	2D	SI	SI	Dibuix a mà	NO
Guacamelee!	2D	SI	SI	Flat	NO
Night in the woods	2D	SI	NO	Flat	NO
Celeste	2D	NO	NO	Pixelart	NO
Owlboy	2D	SI	SI	Pixelart	NO
Wind waker	3D	NO	SI	Chibi	SI

Taula 2.7: Taula de característiques dels jocs del mercat.

Joc	Dimensió	Metroidvania	Combat	Estil gràfic	Exploració
Erlang Legacy	2D	SI	SI	Flat	SI

Taula 2.8: Taula de característiques del projecte.

2.5.4 Resultats

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 Guacamelee!, el qual comparteix 4 de les 5 característiques amb el nostre. Tot i això, creiem que el nostre projecte és distintiu gràcies a les següents diferències:

- Guacamelee! està totalment ambientada a Mèxic. És el seu tret distintiu i no s'asembla a l'estètica que nosaltres volem seguir: una ambientació més medieval i de colors no saturats.

- El nostre joc compta amb la característica de l'exploració no continuada i per illes que el fa únic respecte als altres metroidvanias.
- Les habilitats del jugador a guacamele estan molt basades en en la lluita cos a cos. Les nostres habilitats no venen basades de cap lluita en específic, són totes les habilitats que podia aconseguir un robot, ja que el personatge principal és un androide.

Respecte als altres resultats, hem observat que els jocs els quals més s'acosten al tipus de jugabilitat que volem obtenir, no inclouen l'estètica flat. El que ho fa, Night in the woods, manca d'altres característiques, com el combat.

La conclusió general que extraiem de l'estat del mercat és que el nostre projecte presenta característiques distintives suficients per a 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.

CAPÍTOL 3

Metodologia

La metodologia són tots els passos que es reconeixen a l'hora de la planificació i la gestió d'un projecte. Pot anar des de la gestió de recursos fins a la coordinació de l'equip de treball, o inclús la relació amb tots els interessats en els resultats del projecte.

3.1 Metodologies de gestió de projectes

Per projectes de desenvolupament de programari existeixen un munt de metodologies diferents que s'adapten a diferents necessitats. Aquestes són algunes de les més utilitzades. [Asana 022]

3.1.1 Waterfall

La metodologia Waterfall (o cascada) és una de les metodologies més populars per la seva fàcil implementació. Coneguda com a cicle de vida de desenvolupament de sistemes, és un procés lineal en què el treball es realitza de manera escalonada i en ordre seqüencial. Aquesta forma de treballar, similar a una cascada, és la que li posa nom.

Aquesta metodologia es recolza en la idea que les tasques estan vinculades per dependència, així que és molt important l'ordre de desenvolupament de les mateixes: hem de finalitzar cada tasca abans de començar amb la següent. Promou l'ordre, la claredat i la comunicació en tot el procés.

És ideal per treballar en projectes grans que tenen moltes parts involucrades, ja que els passos clars i les dependències al llarg del projecte ajuden a donar seguiment a la feina necessària per assolir els objectius.

3.1.2 Scrum

Scrum es basa en *sprints*. Els *sprints* són cicles de treball curts que es fan servir per crear una bona comunicació de com avança el projecte. Aquests cicles duren

d'una setmana a dues i s'organitzen amb equips de fins a 10 persones.

A la metodologia Scrum existeix un Scrum Master: un gerent de projectes que dirigeix les reunions, les demostracions i els "spints". Un bon Scrum Master ha d'aconseguir connectar tots els participants del projecte i garantir que les tasques es finalitzin a temps.

La valoració de les persones i la col·laboració dels equips per sobre dels processos és una de les virtuts que també la fa popular. Aquesta metodologia pot ser adequada tant per a equips petits com grans.

3.1.3 Kanban

La metodologia Kanban representa les tasques pendents del projecte usant elements visuals. Aquest enfocament s'utilitza per visualitzar millor els fluxos de treball i el progrés dels projectes.

Aquest mètode no té un procés clarament definit, molts equips el fan servir de maneres diferents. El concepte més important que cal tenir en compte és que l'objectiu principal és centrar-se en les tasques més importants mantenint una estructura simple.

Els taulers Kanban són ideals per a equips de totes les mides, especialment per als equips que treballen remot, ja que els taulers Kanban ajuden als membres de l'equip a visualitzar fàcilment la feina i a mantenir-se al dia sense importar des d'on treballin.

3.2 Metodologia escollida

Tot i que hi ha un munt de metodologies diferents, cap s'adaptava a com volíem a les necessitats del nostre projecte. Finalment, s'ha seguit una metodologia personalitzada proposada pel nostre tutor.

Els passos a seguir de la metodologia són els següents:

1. Elecció del projecte a realitzar.

2. Escollir quines eines i llenguatges s'utilitzaran i aprendre a utilitzar-los.

3. Estructurar el projecte en diferents mòduls i tasques per tal de facilitar la seva repartició i desenvolupament.

4. Escollir i desenvolupar un dels mòduls i/o tasques que han sorgit en el punt 3 en desglossar el projecte.

5. Comprovar que la part desenvolupada funciona correctament.

   (a) Si s'avalua que no s'ha realitzat correctament, es tornarà al punt 3 i es realitzaran els canvis pertinents per assolir l'objectiu.

   (b) Si s'avalua correctament, es dona per finalitzat el desenvolupament del mòdul i/o tasca.

      i. Si queden parts a desenvolupar es tornarà al punt 4.

      ii. Si tots els mòduls han sigut desenvolupats s'avançarà al punt 6.

6. Unir les diferents parts del projecte i comprovar el funcionament

   (a) Si amb la unió no s'obté el funcionament esperat, es tornarà al punt 4, analitzant les falles i fent el pertinent per solucionar-les.

   (b) Si la unió dona un resultat satisfactori, s'avançarà al punt 7.

7. Crear models de proves per així fer comprovacions del correcte funcionament.

   (a) Si el resultat no és l'esperat, es tornarà al punt 4, analitzant les falles i es farà el pertinent per solucionar-les.

   (b) Si la unió dona un resultat satisfactori, s'avançarà al punt 8.

8. Redactar la documentació del projecte.

Per organitzar-nos hem utilitzat les taules Kanban a través de Github Projects, tal com expliquem a la secció 7.11.3, per mantenir un ordre en les tasques i visualitzar fàcilment la feina restant.

3.2.1 Diagrama de flux de la metodologia

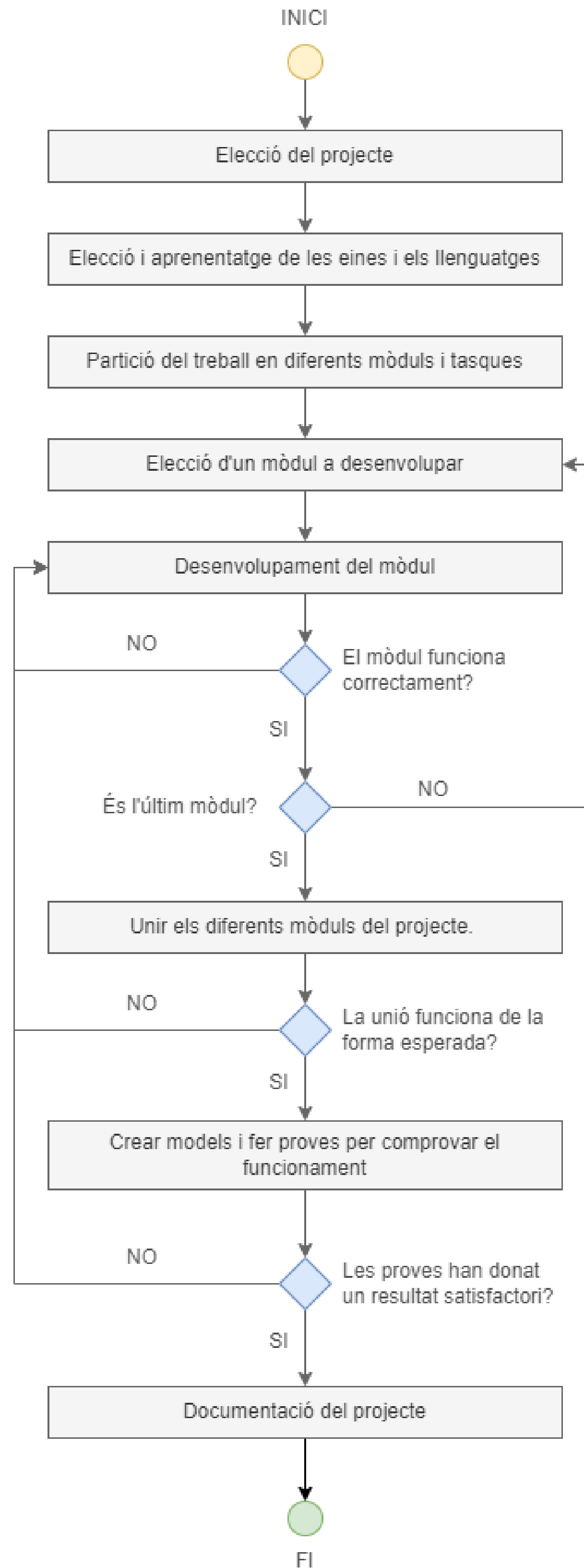

Figura 3.1: Diagrama de flux de la metodologia

CAPÍTOL 4

Planificació

Una part molt important en qualsevol projecte és la planificació. Una correcta planificació ens ajuda a establir la prioritat correcta de cadascuna de les tasques i tenir un millor control del temps per executar un projecte amb èxit. En el nostre cas era crucial, ja que els dos necessitàvem compaginar el projecte amb l'últim curs de la carrera i amb les nostres respectives feines.

En aquest capítol veurem com vam organitzar la planificació a l'inici del projecte i ho compararem amb el temps destinat real.

4.1 Planificació prèvia

Vam començar la preparació d'aquest treball l'estiu del 2021. En aquestes dates els membres del grup ens vam reunir i vam decidir que volíem desenvolupar un videojoc pel treball de final de carrera. En el transcurs d'aquell mateix estiu els dos ens vam documentar sobre el funcionament d'Unity i vam realitzar diferents proves per familiaritzar-nos amb el motor.

Així i tot, l'inici del desenvolupament considerem que no comença fins a finals de gener, després que el nostre tutor ens acceptés la proposta del projecte i on junts vam acabar d'assentar les bases del mateix.

4.2 Tasques planificades

Hem dividit el treball en 12 tasques. Aquestes es poden dividir en 5 apartats:

- Plantejament i aprenentatge
- Disseny
- Implementació
- Proves
- Documentació

A continuació explicarem cada tasca en detall.

4.2.1 Planificació del joc

En aquesta primera tasca es va definir l'objectiu del joc, les característiques principals, els diferents elements amb els quals podrà interactuar al llarg del joc i les regles principals.

4.2.2 Estudi del diferents motors

En aquesta tasca s'estudia el mercat de motors i s'escull quin motor s'utilitzarà per desenvolupar el projecte.

4.2.3 Estudi del programari

En aquesta fase es va estudiar el funcionament del motor de videojocs Unity i el llenguatge de programació que utilitza: C#. També s'estudia com utilitzar altres eines per la creació del videojoc, com el programa d'il·lustració Illustraitor i el software d'animació 2D Spine. L'aprenentatge es fa a través de cursos en línia, vídeos informatius de YouTube, pàgines web d'aprenentatge, i la documentació oficial d'aquests programaris i llenguatges.

4.2.4 Cerca, creació i preparació d'elements gràfics

Es cerquen tots els sprites necessaris per formar el joc: caselles, enemics, elements decoratius, etc. i es creen els sprites que necessàriament han de ser personalitzats per cenyir-se al fil argumental: personatge principal, NPCs, enemics finals, elements decoratius, etc. També es creen les animacions pertinents un cop acabat la cerca i creació dels elements gràfics.

4.2.5 Disseny i implementació d'un protoip amb Unity

S'inicia el projecte creant una primera versió del jugador. Aquest només ha de poder moure's per un espai reduït. En aquesta fase es busca establir les bases de la mobilitat. També es defineix i es crea una càmera que satisfaci les necessitats del nostre joc.

4.2.6 Disseny i implementació del joc

Es desenvolupa les interaccions amb l'entorn: vida del jugador, interacció entre els elements (com palanques o cofres de recompenses), s'implementen les habilitats del jugador.

4.2.7 Disseny i implementació de l'interficie d'usuari

S'implementen els menús per començar, aturar o sortir del joc. S'afegeixen les interfícies d'usuari per poder veure la informació dels elements que es troben dintre el joc com la vida del jugador o quines habilitats posseeix.

4.2.8 Disseny i implementació de IA

Es dissenya i s'implementa diferents intel·ligències artificials pels enemics. Depenent de l'enemic aquestes són més o menys complexes.

4.2.9 Cerca i integració de música

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

4.2.10 Verificació i proves

Realització de proves per assegurar el correcte funcionament dels diferents elements del joc. Revisió del projecte perquè complís amb tots els requisits presentats a l'apartat de plantejament de joc i disseny.

4.2.11 Creació d'una demo del videojoc

Creació d'una demo del videojoc que compti amb tots els elements desenvolupats fins al moment. En aquesta part inclou la creació de les cinematogràfiques que enriqueixen l'experiència de l'usuari.

4.2.12 Documentació

Documentació del projecte. Aquesta és una tasca constant que es realitza al llarg del projecte. La seva duració va des del començament del projecta fins al final.

4.3 Temps estimat

El projecte estava plantejat perquè s'iniciés a finals de gener i dures fins a finals d'agost. Dedicaríem un total de, aproximadament, cent cinquanta dies.

Taula de tasques				
Nº	Tasca	Dies dedicats	Dependències	Assignació
1	Estudi gestor de projectes	1	-	Comú
2	Elecció del motor de videojocs	1	-	Comú
3	Estudi de Unity i C#	5	2	Comú
4	Estudi de Illustraitor	2	-	Judit
5	Estudi de Spine	5	-	Comú
6	Estudi d'una IA pels enemics	5	3	Wilber
7	Planificació del videojoc	3	-	Comú
8	Cerca d'elements gràfics	2	-	Wilber
9	Creació d'elements gràfics	10	2	Judit
10	Animació d'elements gràfics	10	5, 8 i 9	Comú
11	Mecàniques bàsiques	20	3	Wilber
12	Creació d'enemics bàsics	25	6 i 11	Wilber
13	Creació d'interfícies d'usuari	10	3	Judit
14	Sistema de salvament	25	3 i 11	Judit
15	Creació de NPCs	15	3	Judit
16	Altres d'elements del joc	20	3	Comú
17	Creació d'enemics finals	20	12	Wilber
18	Implementació de música i sons	10	13, 15, 16 i 17	Wilber
19	Creació d'un prototip	30	18	Comú
20	Testeig i proves	10	20	Comú
21	Documentació	150	-	Comú

Taula 4.1: Taula de tasques

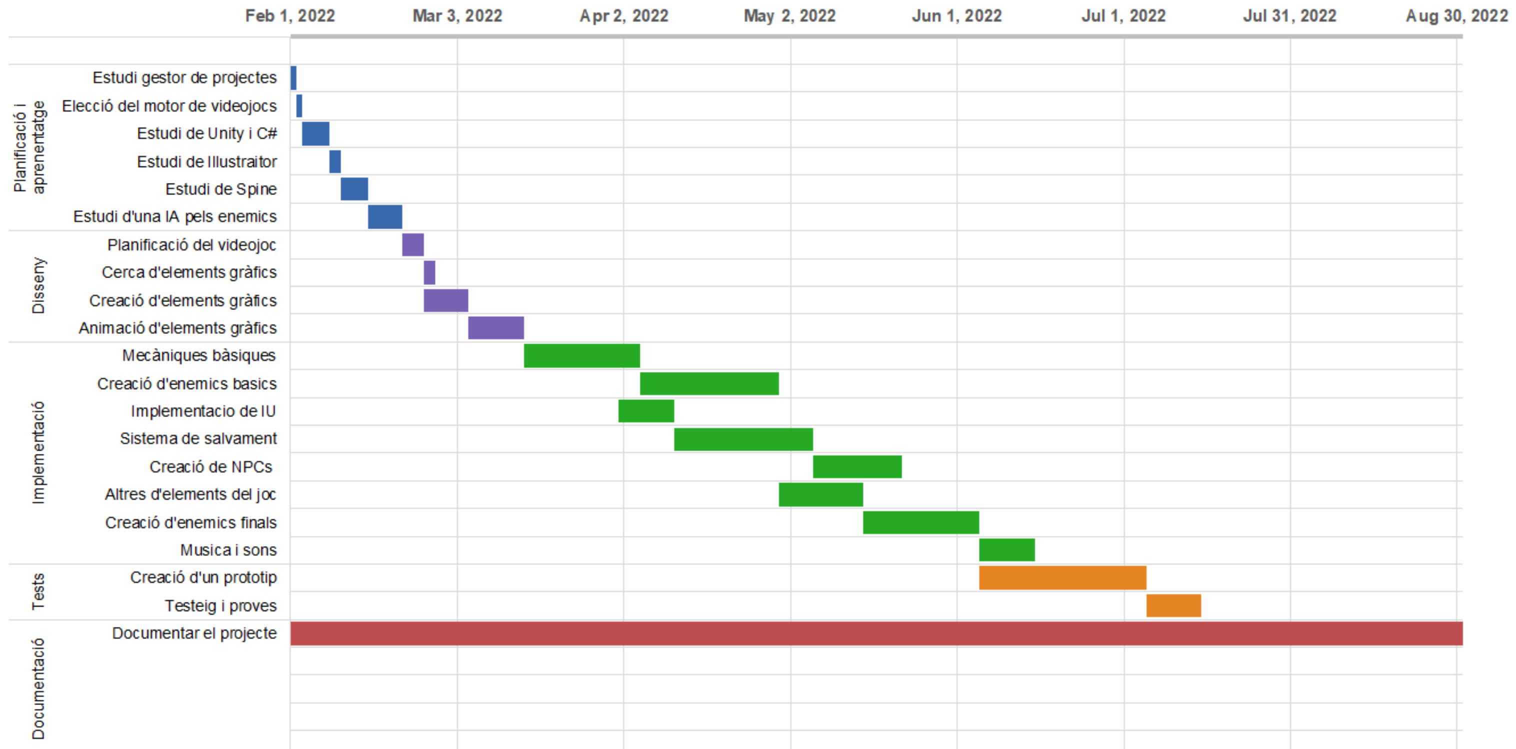

Figura 4.1: Estudi esperat de la planificació del projecte

Tot i tenir el projecte ben planificat, el més probable és que surtin problemes i imprevistos. Durant el desenvolupament del projecte també en vam tenir i això va canviar significativament la planificació.

Problemes que han canviat la planificació:

- **Indecisió al escollir una IA**
  No vam ser capaços d'escollir una IA pel control dels enemics a l'inici del projecte. Al haver tantes possiblitats dubtavem sobre quina seria la millor per les nostres necessitats. La solució va ser deixar aquest apartat a una etapa mes llunyana del projecte.

- **Problemes de contrast**
  Un altre problema que va sorgir va ser respecte l'art del videojoc. A mitjans del desenvolupament vam veure com el personatge principal del videojoc es perdia en el mapa: els colors no estaven ben contrastats. Al no haver estudiat mai art no era un problema que ens haguessim plantejat. Ens trobavem en una etapa molt avançada del desenvolupament i canviar l'art ens representava un gran problema. Despres de discutir el problema amb el tutor i un altre professor especialitzat en l'art dels videojocs, ens vam adonar que podriem solucionar el problema utilitzant llums. La llibreria de Unity Lights 2D 7.5.20 ens oferia tot el necessari per treballar amb llums en un projecte 2D. Vam haver d'estudiar el funcionament de la llibreria i posteriorment introduir-la al projecte.

- **Imprevistos respecte la durada de les tasques**
  Algunes tasques es van allargar i d'altres es van retrassar.

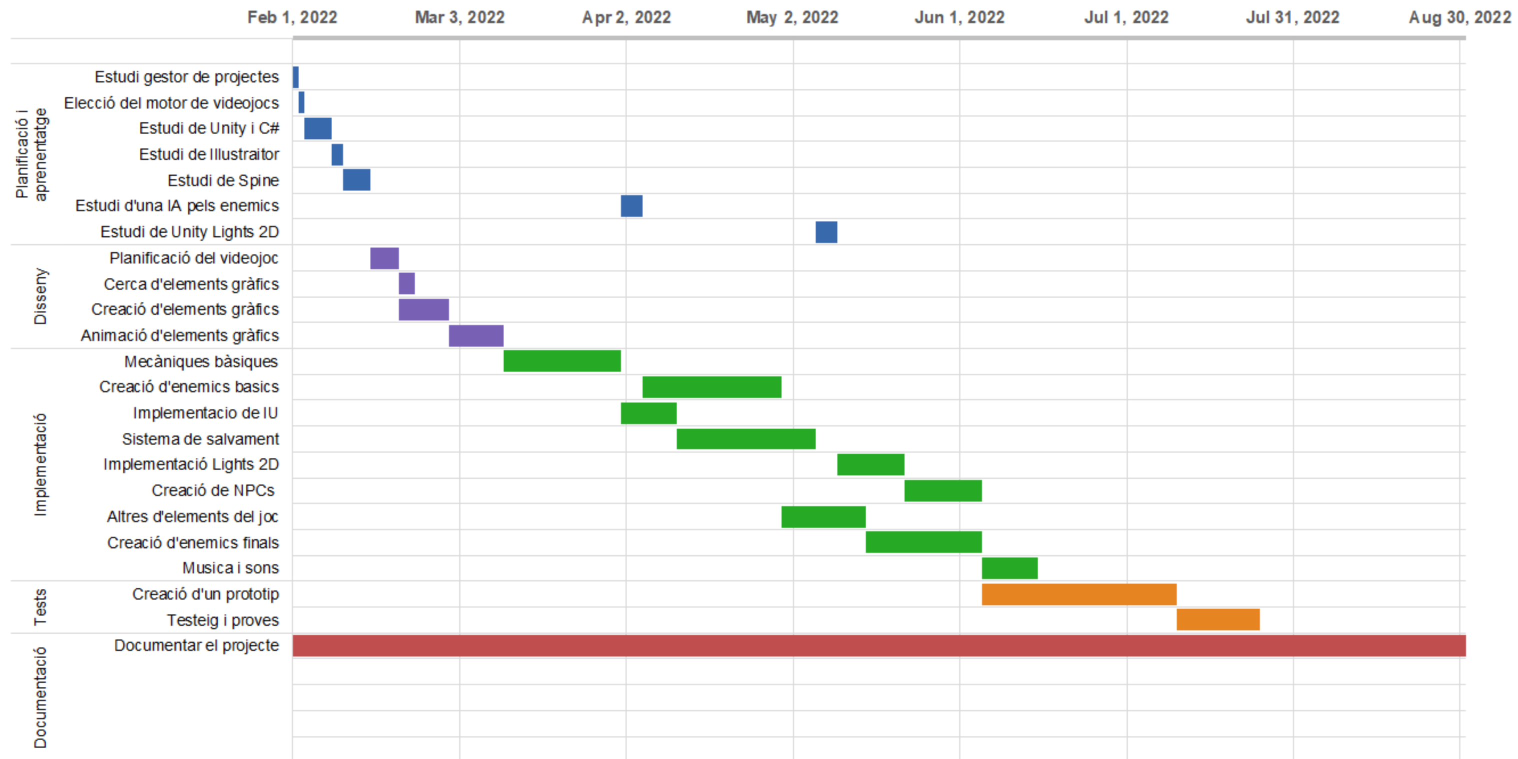

Figura 4.2: Planificació resultant del projecte

4.4 Resultats esperats

L'objectiu d'aquesta planificació és poder crear un videojoc completament funcional, entretingut, fàcilment expansible i que compleixi amb tots els requisits presentats a l'apartat de plantejament del joc i disseny.

CAPÍTOL 5

Marc de treball i conceptes previ

Fer un videojoc pot ser relativament simple a molt complicat, tot depèn de si ens hem d'encarregar de crear la infraestructura i l'entorn de desenvolupament juntament amb el videojoc, o bé, si hem de fer exclusivament el videojoc sobre un entorn dissenyat per fer aquesta tasca.

El mercat del desenvolupament de videojocs ofereix una ampla varietat d'alternatives d'entorns de desenvolupament. Aquests entorns són comunament coneguts com a motors de videojocs.

En la majoria dels casos, a l'hora de desenvolupar un videojoc, s'utilitza com a eina principal de desenvolupament un motor de videojocs. Un motor de videojoc és un programari que permet la creació, disseny, representació i que comprèn un gran nombre de funcionalitats com renderització, física, col·lisió, so, lògica per codi, animació, intel·ligència artificial, xarxa, streaming i un gran, etcètera que depèn del complex i ample del motor.

Un motor de videojoc esta format per altres motors més petits, entre ells, el motor gràfic, el motor físic. El motor gràfics tracta l'aspecte visual del videojoc, que generen imatges sintètiques integrant o canviant informació visual i espacial. Els motor físics s'ocupen d'integrar les lleis de la física, i són responsables de simular accions reals, a través de variables com la gravetat, la massa, la fricció, la força i la flexibilitat. Veure Figura 5.1.

La funció principal d'un motor de videojocs és la de facilitar la feina als desenvolupadors de videojocs fent que aquests s'abstreguin de la implementació a baix nivell, l'abstracció de maquinari és essencial per al desenvolupament de videojocs multiplataforma.

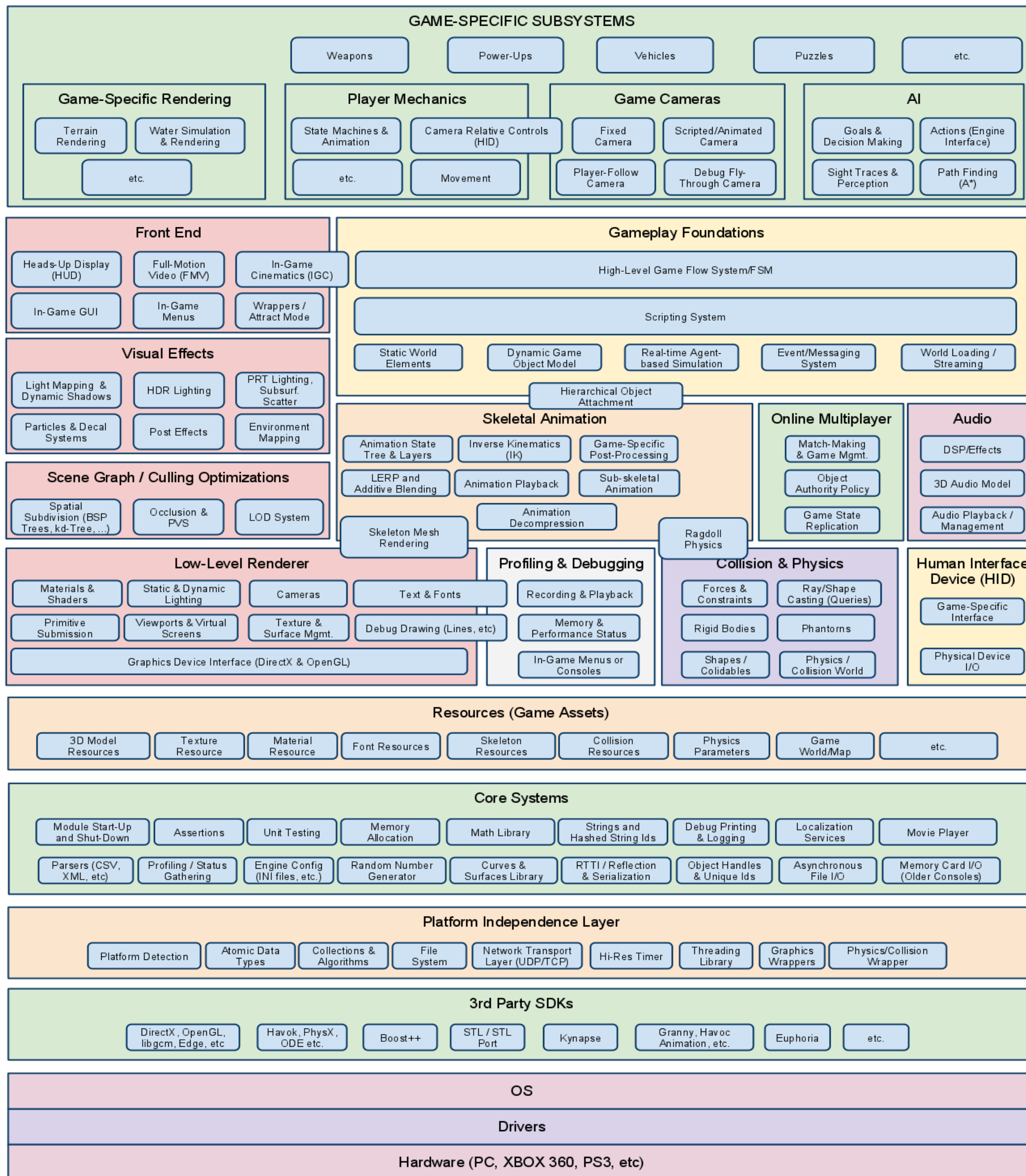

Figura 5.1: Esquema d'un motor de videojocs.

5.1 Motors de videojocs

A continuació veurem alguns dels motors de videojocs més populars en el mercat avui dia.

5.1.1 Unreal Engine

Sens dubte, un dels motors més utilitzats. La versió 5 està causant estralls per la quantitat de possibilitats que ofereix. Un motor gràfic molt potent desenvolupat per Epic Games que ofereix les eines necessàries per crear un videojoc de principi a fi. Videojocs com Fortnite, Gears of War, la saga Batman Arkham, la saga Borderlands, Unreal Tournament o Devil May Cry han estat fets amb Unreal Engine.

Un motor especialitzat en videojocs 3D que permet assolir un nivell de realisme tal que ha fet que indústries com la cinematogràfica o l'arquitectònica comencin també a fer servir per a les seves produccions. Per fer videojocs 2D hi ha programes externs com Paper 2D, que disposa d'eines de grafisme i animació per crear jocs bidimensionals.

Unreal Engine permet afegir lògica amb el llenguatge de programació C++ o bé amb Blueprint Visual Scripting. Aquest últim és un sistema de scripts de joc complet basat en el concepte d'utilitzar una interfície basada en nodes per crear elements de joc des de l'editor Unreal.

Figura 5.2: Logo de Unreal Engine.

5.1.2 Unity 3D

Aquest motor de creació de videojocs és un dels més coneguts juntament amb Unreal Engine. Un motor superrobust, potent i fàcil de fer servir. A més, és compatible amb moltíssimes plataformes i té grans comunitats d'usuaris al voltant.

Unity compta amb una versió gratuïta, però és necessari canviar a una llicència per desenvolupadors quan s'arribi a certes quotes de benefici. 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 hi poden trobar models 3D, textures, música, efectes visuals i sonors.

Històricament, Unity havia suportat els llenguatges Boo, Unity Script (el qual és una versió de JavaScript, però no idèntic) i C#, però l'11 d'agost del 2017 van fer un comunicat explicant les raons per no continuar suportant Unity Script [Unity 022i] i ja prèviament havien discontinuat Boo a causa del poc ús.

Figura 5.3: Logo de Unity.

5.1.3 Godot

Godot és un motor de jocs 2D i 3D d'ús general dissenyat per admetre tota mena de projectes. Es pot utilitzar per crear jocs o aplicacions que després es pot llençar a ordinadors d'escriptori o dispositius mòbils, així com a la web.

Godot és programari lliure i de codi obert, publicat sota la Llicència MIT. Va ser desenvolupat inicialment per un estudi de jocs argentí. El desenvolupament va començar el 2001, i el motor es va reescriure i va millorar enormement des del llançament de codi obert, el 2014.

Els llenguatges de programació suportats nativament per Godot són: GDScript, VisualScript i C#. Godot, amb la tecnologia GDNative de la qual són propietaris, permet que el motor interactuï amb biblioteques compartides natives en temps d'execució. Això implica que es pot usar Godot amb codi natiu sense compilar-lo amb el motor. I, ja que Godot és de codi obert, ha permès que la comunitat doni suport a altres llenguatges de programació que no són els propis de Godot, com ara: C, C++, Haskell, Python, entre altres [Godot 022].

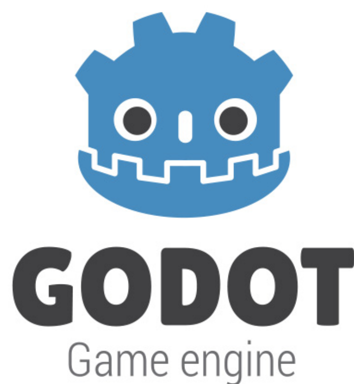

Figura 5.4: Logo de Godot.

5.1.4 CryENGINE

CryENGINE es tracta d'un motor de videojocs desenvolupat per l'empresa alemanya Crytek. Inicialment, es va construir com un motor de demostració per Nvidia, però en veure el seu potencial, es va decidir implementar-lo per al videojoc Far Cry.

La particularitat més gran que ofereix aquest motor és la seva increible característica gràfica, la qual és reconeguda com una de les millors de tot el sector, com també el sistema de físiques.

CryENGINE falla pel que fa a les seves interfícies d'usuari i la seva facilitat d'ús, el qual provoca que només aquells amb experiència puguin accedir a ell de forma còmoda. CryENGINE utilitza LUA com a llenguatge de programació

Figura 5.5: Logo de CryENGINE.

5.2 Motor triat

Figura 5.6: Logo de Unity.

Unity ha sigut el motor que hem triat pel desenvolupament del projecte, hem decidit escollir aquest pels següents motius:

- Té una comunitat molt ampla i activa, el que facilita la cerca d'informació i ajuda.
- Proporciona una documentació excel·lent.
- Conté molt probablement el mercat més gran d'*assets* i eines d'ús gratuït i de pagament.
- L'editor Unity, a parer nostre, ofereix les millors i més recents eines per desenvolupar i llançar videojocs 2D.
- La mateixa Unity proporciona cursos gratis per aprendre com usar l'editor i d'eines que s'acostumen a fer servir per al desenvolupament de videojocs.

CAPÍTOL 6

Requisits del sistema

En aquest capítol estipularem els requisits de la nostra aplicació els quals agrupen en gran manera els objectius i les funcionalitats amb les quals ha de complir. Dividirem aquests requisits en dos grans apartats: els requisits funcionals, que fan referència a les funcionalitats que el sistema ha de dur a terme; i els no funcionals, que fan referència a les restriccions de disponibilitat dels recursos, seguretat, interfícies externes, la forma en la qual es desenvolupa el projecte, etc.

6.1 Requeriments funcionals

6.1.1 Identificació dels actors

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

El projecte només té un únic actor, que és el jugador que interactua amb tot el sistema. Tots els jugadors tenen els mateixos privilegis.

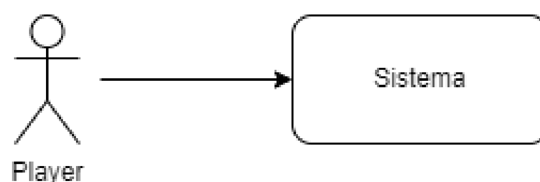

Figura 6.1: Interacció del jugador amb el sistema.

6.1.2 Llistat de requeriments funcionals

Els Requeriments funcionals de la nostra aplicació són els següents:

- El jugador ha de ser capaç de crear una nova instància de joc o recuperar la instància de joc anterior.

- El jugador ha de poder configurar el volum de la música del joc així com els efectes especials de so, dintre del joc i en la pantalla d'inici.
- El jugador ha de poder veure la configuració de mapatge de tecles dintre del joc i en la pantalla d'inici.
- El jugador ha de poder guardar la partida en certs punts dintre del joc.
- El jugador ha de poder controlar el moviment del personatge principal així com les mecàniques.
- El jugador ha de poder gaudir d'una varietat ampla de mecàniques de joc; plataformes, portes, trampolins, etc.
- El jugador ha de poder tenir una sensació de progrés al llarg de joc, adquirint noves habilitats i augmentant el nombre màxim de vides.
- El jugador ha de poder interactuar amb elements de les escenes, siguin NPC, enemics o altres objectes.
- El jugador ha de poder gaudir d'una ampla varietat d'interfícies d'usuari com ara, diàlegs, notificaciones, sistema d'habilitats disponibles, sistema de vides, entre d'altres.
- El jugador ha de poder sortir de l'aplicació en tot moment, des del menú d'inici i des del joc.

6.2 Requeriments no funcionals

- El desenvolupament del joc segueix una metodologia de revisió constant proposada pel tutor, i suportada amb la metodologia Kanban gràcies a Github Projects.
- El desenvolupament es fa en el sistema operatiu Windows 10 per evitar problemes de compativilitat o no existència del programari en altres sistemes.
- Ha d'haber-hi un mecanisme de desplegament continuo per tal de treure automàticament noves versions del joc.
- Tot i que no hi ha varietat de rols i que les dades que es generen en el joc no són dades crítiques, les dades que es guarden per recuperar sessió la de joc han d'estar xifrades.

- El resultat del videojoc ha de poder córrer en qualsevol plataforma d'escriptori amb navegador que suporti WebGL. Podem veure el suport de plataforma i navegador en la documentació oficial d'Unity [Unity 022j].

Navegador d'escriptori	Plataformes d'escriptori
Google Chrome	Windows, macOS, Linux
Mozilla Firefox	Windows, macOS, Linux
Apple Safari	macOS
Microsoft Edge	Windows, macOS, Linux

Taula 6.1: Compatibilitat WebGL en navegadors.

CAPÍTOL 7

Estudi i decisions

En aquest apartat enumerarem cadascun dels programaris utilitzats pel desenvolupament del projecte, donarem una descripció específica de l'ús que l'hem donat a cadascuna de les eines i en alguns casos, explicarem l'API [1] del programari per donar context a la seva utilitat.

7.1 Sistema Operatiu

Nosaltres hem fet servir el Sistema Operatiu Windows 10 Home amb arquitectura X64. Tot i que podríem haver fet servir altres alternatives, la realitat és que o bé no havíem treballat mai amb els altres sistemes, o bé temíem que algun dels programaris que pensaven utilitzar no tinguin suport.

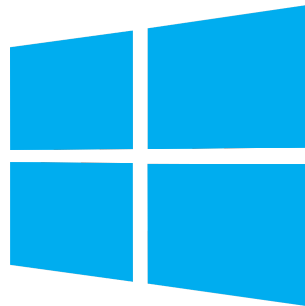

Figura 7.1: Logo Windows 10.

[1] *Application Programming Interface, és un conjunt de subrutines, funcions i procediments que ofereix certa biblioteca per ser utilitzada per un altre programari com una capa d'abstracció*

7.2 C#

C# és un llenguatge de programació multiparadigma, desenvolupat inicialment per Microsoft com a part de .NET Core[2]. C# deriva dels llenguatges C i C++, aquest també té similituds amb el llenguatge de programació Java.

La raó d'escollir aquest llenguatge per a desenvolupar el projecte és que dels 3 llenguatges de programació suportats per Unity: Boo, UnityScript i C#; C# és l'únic que a dia d'avui és rellevant dintre del ecosistema.

7.3 Visual Studio Code

Visual Studio Code és un editor de text lleuger però potent, que s'executa a l'escriptori i està disponible per a Windows, macOS i Linux. Aquest programari està desenvolupat per Microsoft i és de codi obert. Visual Studio Code permet afegir extensions al nostre gust de manera fàcil i ràpida gràcies al VS Code Extension Marketplace.

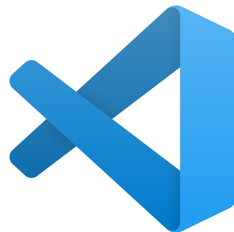

Figura 7.2: Logo Visual Studio Code.

Al mercat hi ha moltes alternatives a VSCode[3], per exemple; Visual Studio, també desenvolupat per Microsoft, però aquest és un IDE[4]. La raó principal de per què hem escollit VScode envers Visual Studio, tot i ser aquest últim l'entorn per defecte, és perquè és un IDE, i generalment els IDE requereixen més recursos que un editor de text. Les altres raons de per què aquest editor respecte altres és perquè és gratuït i ja teníem certa experiència treballant-hi amb ell.

[2]És la plataforma de desenvolupament de Microsoft més moderna, de codi font obert, multiplataforma i d'alt rendiment per a la creació de tot tipus d'aplicacions.

[3]Visual Studio Code

[4]Un entorn de desenvolupament integrat (IDE) és una suite de programari que consolida les eines bàsiques necessàries per escriure i provar programari

7.3.1 C# for Visual Studio Code

Aquesta extensió proporciona les següents funcions dins de VS Code [Microsoft 022].

- Eines de desenvolupament lleugeres per a .NET Core.
- Gran suport d'edició de C#, inclòs el ressaltat de sintaxi, IntelliSense, entre altres.
- Suport de depuració per a .NET Core.

7.3.2 Unity Code Snippets

Aquesta extensió pretén ser la col·lecció completa de fragments de codi d'Unity per a Visual Studio Code. Aprofita les últimes funcions de generació de codi de Visual Studio per a crear codi més ràpidament. [Silva 022].

7.4 Action

És un tipus de dades pròpia de C# de referència que es pot utilitzar per encapsular un mètode o una funció anònima. Molt útil per demorar l'execució de lògica i ajuda al desacoblament de codi.

7.5 Unity

En els apartats 5.1.2 i 5.2 hem explicat el que aquesta eina és i que ens permet fer, però ara explicarem com està distribuït l'editor d'Unity, sistema de capes, les físiques, l'ordenament per capes, l'ordenament per grups de capes, que és un objecte dintre d'Unity, que és un Canvas, el sistema de Tiles, el sistema de càmera intel·ligent, el sistema de llums, el Scriptables Object i més components que ens interessa ressaltar.

7.5.1 Component

Els components agrupen tots els atributs que un Game Object podria arribar a tenir, des d'elements 3D, llum, so, animacions, components propis creats per nosaltres en forma de scripts[5], entre d'altres.

[5]Un script és un document que conté instruccions, escrites amb un o varis llenguatges de programació.

7.5.2 Monobehavior

MonoBehaviour és la classe base de la qual deriva cada script d'Unity. Quan s'utilitza C# per crear nous components, s'ha de derivar explícitament de MonoBehaviour.

Els components en Unity tenen un cicle de vida intern. Monobehavior exposa aquest cicle per metodes que podem sobreescriure. A continuació remarquem alguns dels punts de cicle de vida més rellevants. Veure Figura 7.3.

- Awake. Aquesta funció sempre es crida abans de qualsevol funció Start i també just després que un prefab és instanciat. (Si un GameObject està inactiu durant el començament, Awake no és cridat fins que es torni actiu).

- Start. És executat abans de la primera actualització de fotograma només si la instància de l'script està activada.

- Update. És crida una vegada per fotograma. És la funció principal per a les actualitzacions de fotograma.

- OnDestroy: Aquesta funció és executada després que totes les actualitzacions de fotograma per a l'últim fotograma de l'existència de l'objecte.

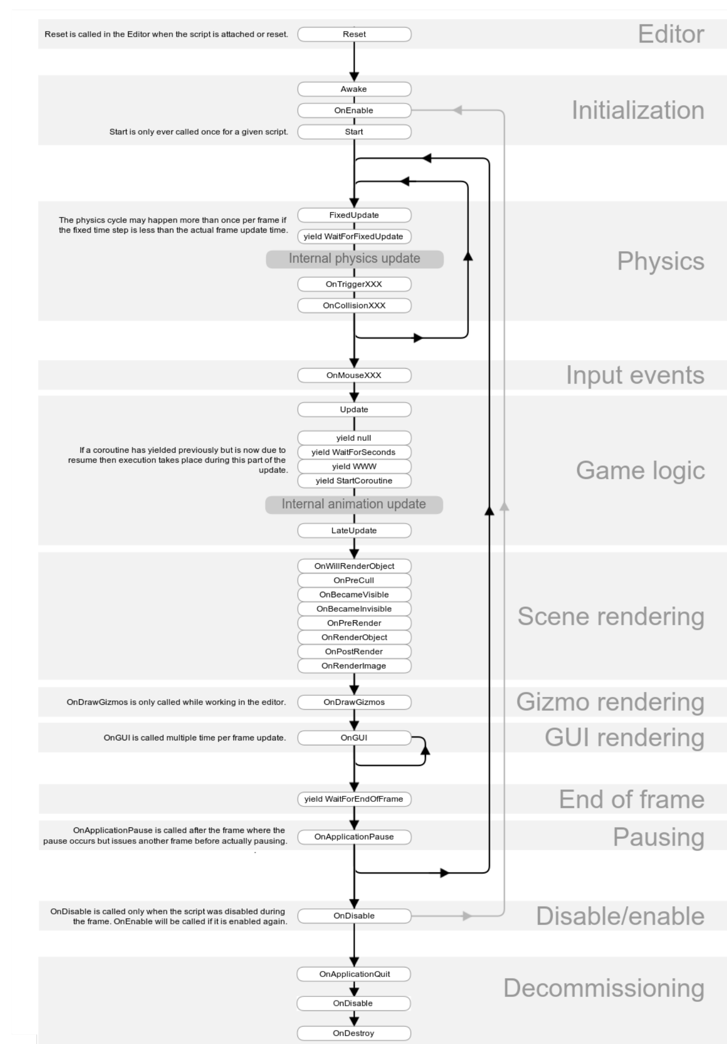

Figura 7.3: Cicle de vida d'un component.

7.5.3 Game Object

Els Game Object són objectes fonamentals en Unity i representen objectes en l'escena. Aquests no aconsegueixen res per si mateixos, però funcionen com a contenidors per a Components que implementen la veritable funcionalitat.

Els Game Object es tracten de tots els elements que nosaltres dipositem en una escena i que podem desplaçar, escalar i eliminar, ja que sempre tenen associat un Component de transformació. Un Game Object es defineix pels diferents Components i les seves configuracions.

7.5.4 Prefab

Els Game Object poden ser configurats i guardats a disc, un cop guardats es converteixen en Prefabs. Els Prefabs es poden utilitzar repetidament en totes les escenes. Els Prefabs afegits a les escenes no són estàtics, els valors per defecte es poden canviar i això dona molta flexibilitat.

7.5.5 Tag

Propietat d'un Game Object que permet trobar elements dintre de l'escena. Està pensat per fer-se servir excepcionalment en elements importants i de identificació inmediata, com ara el jugador.

7.5.6 Transform

Componenet que permet modificar la posició, rotació i escala d'un Game Object.

7.5.7 Collider

Un Collider és un component d'Unity que permet saber quan dos objectes estan en contacte, o se superposen en l'espai. Existeixen dues classes de Colliders: els que estan pensats per fer efecte de col·lisió i els que estan pensats per emetre esdeveniments de superposició. Aquests últims no interactuen amb els primers provocant col·lisió.

7.5.8 Layer

Propietat d'un Game Object que assigna a quina capa física pertany un objecte en l'escena.

7.5.9 Rigidbody

El Rigidbody és un element d'Unity que s'utilitza per fer que un element sigui afectat per la configuració de les físiques del motor. Aquest component en permet conferir una massa, fricció i més caràcteristiques físiques als elements. També ens permet restringir per quin dels eixos del món l'element es podrà desplaçar i sobre quins eixos podrà girar. Normalment, els Rigidbody van associats amb almenys un Collider.

7.5.10 Layer Collision Matrix

La matriu de col·lisions de capes ens permet, entendre la interacció entre capes físiques. Perquè hi hagui una interacció necessitem tenir almenys un Collider i un Rigidbody com a components en un Game Object, i assignat les capes físiques correctes sobre els elements que volem que tinguin interacció. Veure Figura 7.4.

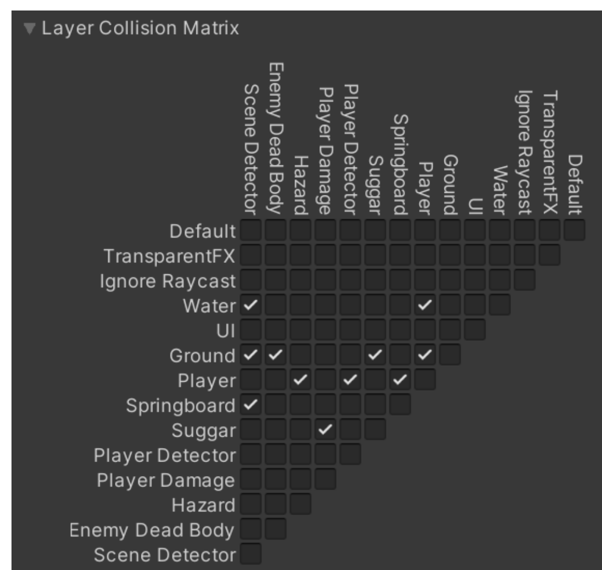

Figura 7.4: Matriu de col·lició de capes del projecte.

7.5.11 Sorting Layer

Propietat d'un Game Object que assigna a quina capa visual pertany en l'escena.

7.5.12 Sorting Group

Component que s'afegeix a un Game Object per tal que els fills i ell mateix pertanyin a un mateix grup de capa visual.

En la Figura 7.5 podem veure part de l'estructura típica de jerarquia dintre de les nostres escenes. Dintre de l'objecte *Scene* tenim diversos objectes fills de primer nivell que representen les capes visuals. Cada un d'aquests objectes tenen un component Sorting Group associat, que farà que tots els fills s'ordenin visualment segons el Sorting Layer assignat al Sorting Group de l'objecte pare. D'aquesta manera és que fem una sensació de profunditat a les escenes.

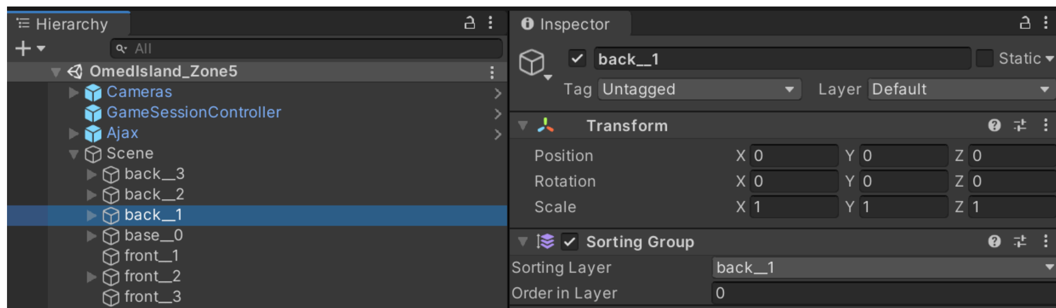

Figura 7.5: Sorting Group per donar profunditat a les escenes.

7.5.13 Camera

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

7.5.14 Cine Machine

És un conjut d'eines per a càmeres dinàmiques, intel·ligents, que permet fer les millors tomés apareguin segons una composició i interacció d'escena, cosa que permet afinar, modificar, experimentar i crear comportaments de càmera en temps real.

Aquesta eina ens ha alliberat d'implementar la lògica de càmera per tal que en cada escena la càmera segueixi al jugador i sempre mostri el contingut de les escenes sense sortir-se dels límits.

Aquesta eina no ve per defecte en l'editor d'Unity i s'ha d'instal·lar fent ús del Package Manager de Unity.

7.5.15 Canvas

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

7.5.16 Audio Source

Component que permet reproduir àudio a través d'un Game Object.

7.5.17 Audio Clip

Component que encapsula un àudio en format Ogg, MP3, entre altres. Aquests component amb l'Audio Source generen so en les escenes.

7.5.18 Audio Listener

Component que permet sentir els sons generats pels diferents AudioSource. Aquest element pot estar en qualsevol lloc de l'escena, però generalment està en la càmera.

7.5.19 Scriptable Object

Un Scriptable Object és un contenidor de dades que podeu utilitzar per desar grans quantitats de dades, independentment de les instàncies de classe. La diferència entre un Scriptable Object i Prefab és que, per fer servir el Prefab hem de fer una instància nova, és a dir, fer una còpia. A l'hora d'usar un Scriptable Object no calen fer instàncies, s'accedeix de forma immediata per referència.

Ara bé, s'ha de tenir en compte el següent, quan s'utilitza l'Editor d'Unity, es poden desar dades a ScriptableObjects durant l'edició i en temps d'execució perquè els ScriptableObjects utilitzen l'espai de noms de l'Editor i els scripts de l'Editor. En una compilació desplegada, però, pot provocar problemes perquè els ScriptableObjects afegits als scripts tindran l'espai de nom dels scripts i, per tant, pot deixar de ser un espai compartit i comú, per solucionar aquest problema el que es pot fer és que tots els scripts que depenguin del ScriptableObject facin servir un tercer script com a servei, d'aquesta manera es manté un únic espai de nom per ScriptableObject.

L'avantatge principal dels Scriptable Object és que ens donen un gran desacoblament, ja que permeten que objectes de les escenes interaccionin amb objectes d'altres escenes mitjançant aquest medi. Per exemple, nosaltres per disseny, en cada escena tenim un objecte jugador diferent, és a dir, còpies d'aquest, però la interfície d'usuari és sempre la mateixa. Totes les còpies del jugador coneixen de l'estat del jugador, vides, habilitats, entre altres, i a la vegada la interfície d'usuari ens informa del nombre de vides i del refredament de les habilitats, ja que accedeixen al mateix recurs compartit.

7.5.20 Light 2D

La il·luminació moderna dels videojocs és complexa i es fa una sèrie de tècniques i models matemàtics que intenten simular el comportament complex de la llum a mesura que rebota i interactua amb el món. Simular la il·luminació global amb precisió és un repte i pot ser computacionalment costós.

Per il·luminar el nostre projecte primer hem de triar un espai de color. L'espai de color determina les matemàtiques que utilitza Unity quan barreja colors en càlculs d'il·luminació o llegeix valors de Textures. En molts casos, la decisió sobre quin espai de color es fa servir està determinada per les limitacions de maquinari de la plataforma de destinació. Unity suporta dos espais de colors; Linear Color Space i Gamma Color Space. Veura Figura 7.6.

Nosaltres vam escollir Linear Color Space com a espai de colors, ja que té un avantatge important i és que els colors subministrats als Shaders dins de les escenes s'il·luminen de manera lineal a mesura que augmenten les intensitats de la llum. Amb l'alternativa, Gamma Color Space, la brillantor comença a convertir-se ràpidament en blanc a mesura que augmenten els valors, cosa que és perjudicial per a la qualitat de la imatge. S'ha de dir, però que tampoc vam tenir alternativa, ja que quan construíem el joc per plataforma WebGL, no hi havia manera que funciones amb Gamma Color Space. En el Capítol 9 s'explica àmpliament el nostre problema amb aquesta gamma de color, i el suport per WebGL.

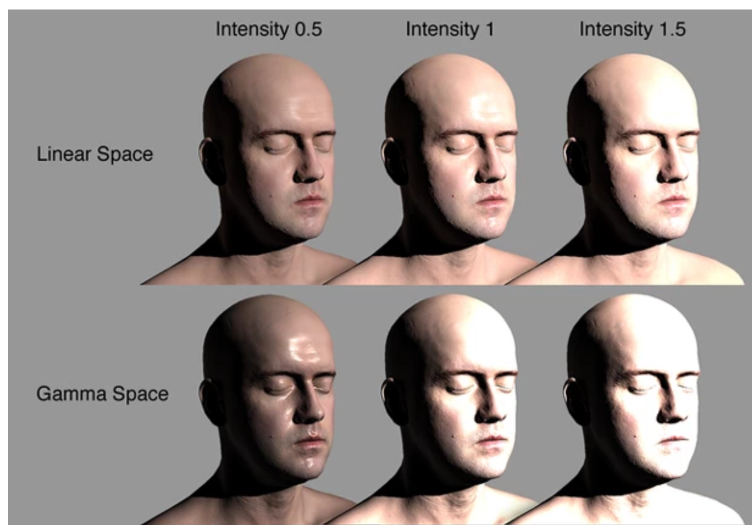

Figura 7.6: Comparativa de espais de colors.

Hi ha diferents tipus de llums, a continuació fem un llistat de les més impor-

tants:

- Llum global, il·luminen tots els objectes de les capes d'ordenació orientades. Només es pot utilitzar una llum global per estil de barreja i per capa d'ordenació.
- Punt de llum, es pot pensar com un punt de l'espai 3D des del qual s'emet llum en totes direccions. Són útils per crear efectes com bombetes, resplendor d'armes o explosions, on s'espera que la llum irradiï des d'un objecte.
- Amb forma lliure, la llum es crea a partir d'un polígon editable. Ens va ser útil a l'hora d'emetre llum de les parts de l'escena on hi havia lava per exemple.

La informació respecte a la llum i Unity és estret de les mateixes publicacions d'Unity, com ara [Unity 022b] o [Unity 022c] i clarament està, de la nostra experiència treballant amb aquesta eina.

7.5.21 Scene Manager

El gestor d'escenes fa un seguiment de les escenes del joc, permetent canviar entre elles. De manera bàsica, proporciona un lloc centralitzat per carregar i descarregar les escenes, fer un seguiment de quina es carrega i gestionar la descàrrega d'aquesta escena quan se'n carrega una de nova. Aquest mecanisme ve per defecte dintre de l'API d'Unity.

7.5.22 Editor

L'editor d'Unity està compost per múltiples finestres, nosaltres destaquem les que es veuen en la Figura 7.7, encara que hi ha moltes més finestres de les que em podríem parlar.

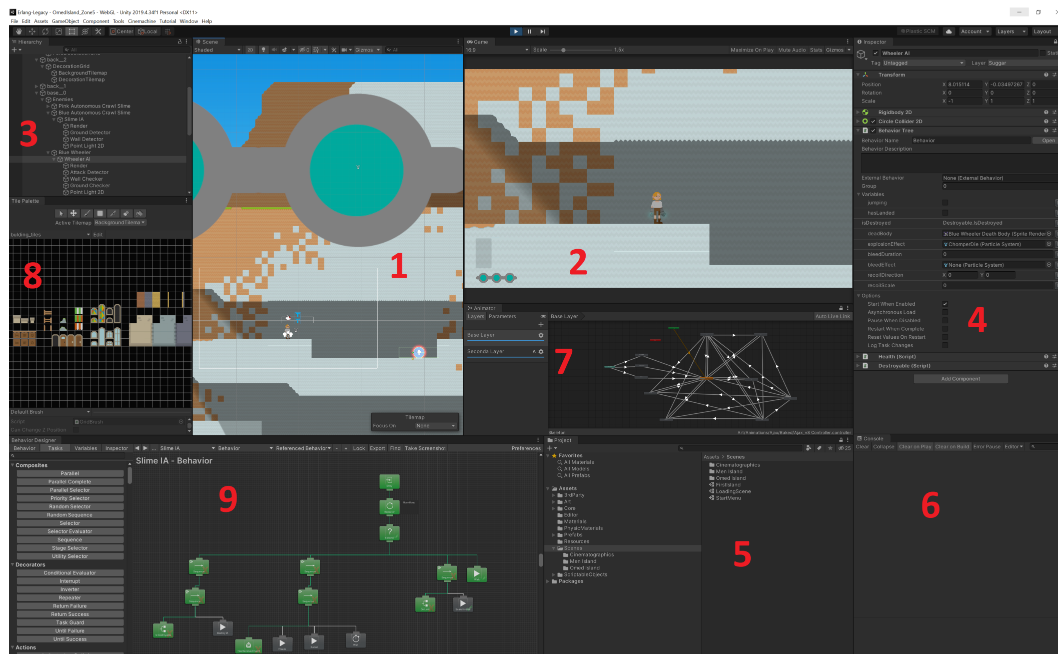

Figura 7.7: Editor d'Unity.

7.5.22.1 Scene

Finestra des de la qual podem interactuar amb l'escenari amb vista 2D o 3D i modificar, afegir, o treure els elements que conté l'escena actual. Aquesta finestra la vam utilitzar molt perquè és la que ens permet dissenyar i construir les escenes i menús.

7.5.22.2 Game

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

7.5.22.3 Hierarchy

Aquesta finestra mostra tots els Game Object de l'escena actual com també la relació jeràrquica que puguin tenir aquests.

7.5.22.4 Inspector

L'inspector es tracta d'un apartat que permet visualitzar tots els Components que un Game Object té adjuntant. Aquests components passen per components propis d'Unity com és el component Transform o per components propis creats per Scripts en C#.

7.5.22.5 Project

Aquesta finestra permet accedir a tots els elements utilitzats en el projecte navegant entre carpetes.

7.5.22.6 Console

Mitjançant aquesta finestra, podem depurar en temps d'execució fent servir instruccions de sortida de pantalla, i també ens informa d'errors i alertes en temps de compilació.

7.5.22.7 Animator

Eina que permet la creació de màquines d'estats d'animacions. És a dir, permet connectar les nostres animacions fent ús de transicions que necessiten o no regles per passar d'estat a estat.

7.5.22.8 Tilemap

Aquesta eina no ve amb l'editor d'Unity per defecte, s'ha d'instal·lar fent ús del Package Manager d'Unity. L'eina permet tenir diferents paletes de Tiles. Els Tiles són especials perquè es poden posar sobre components Tilemap, que fa de llenç, d'aquesta manera podem pintar sobre l'escena quadrat a quadrat i dissenyar el contingut de l'escena senzillament.

7.5.22.9 Behavior Designer

Aquesta finestra conte l'eina que permet dissenyar intel·ligències artificials gràficament.

7.6 DOTween

DoTween és un motor d'animació orientat a objectes, ràpid, eficient i totalment segur per a Unity, optimitzat per a usuaris de C#, gratuït i de codi obert, amb un munt de funcions avançades [DOTween 022].

Podem enumerar alguna de les característiques més importants de DOTween com:

- Velocitat i eficiència, tot s'emmagatzema a la memòria cau i es reutilitza per evitar assignacions al col·lector de brossa inútilment.
- IntelliSense[6] and type-safety[7].
- API lògica i fàcil d'utilitzar. Una API creada per augmentar l'eficiència, l'intuïtivitat i la facilitat d'ús.
- Mode segur. El mode segur és opcional. DOTween s'ocupa d'ocurrències inesperades, com ara que objectes de l'escena es destrueixin mentre es juga.

Aquesta llibreria dona un nivell d'abstracció més alt que si treballéssim amb les característiques pròpies que ens ofereix C# i Unity.

7.7 Behavior Designer

Behavior Designer és un programari privatiu creat per Opsive [Opsive 022]. Aquesta eina ens permet crear arbres de comportament. Un arbre de comportament és un tipus d'intel·ligència artificial que s'utilitza generalment en el món dels videojocs, però no exclusivament, per exemple també s'utilitzen en sistemes de control i robòtica.

Els arbres de comportament permet crear agents que canvien entre un conjunt finit de tasques modularment. El poder d'aquest enfocament és que permet crear llargues col·leccions de tasques i que els agents canvien de comportament de manera fluida.

[6] Funció de autocompletat.

[7] En el context de la semàntica denotacional, la seguretat de tipus significa que el valor d'una expressió de tipus τ, és un membre de bona fe del conjunt corresponent a τ.

Behavior Designer és una eina basada en gràfics per Unity que permet dissenyar arbres de comportament de forma visual. Els arbres de comportament es poden construir en un enfocament purament basat en codi, però la representació visual permet entendre el comportament dels agents d'una manera més fàcil.

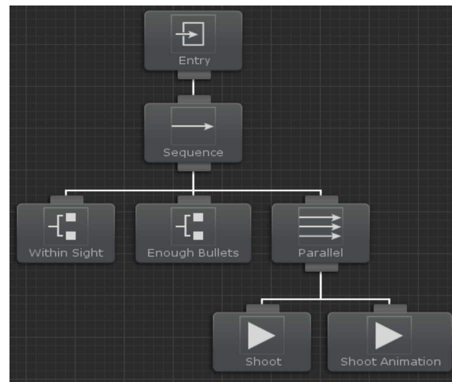

Figura 7.8: Exemple d'un arbre de comportament en Behavior Tree.

Alguna de les característiques més rellevants d'aquesta eina són:

- És un editor visual.
- Té eines per depurar.
- Té variables locals i globals que permeten comunicar-nos amb tasques d'un mateix arbre o d'arbres externs.
- Té moltes tasques ja programades i que venen amb el mateix entorn nativament.
- Té execució dinàmica, el que significa que es poden interrompre subarbres amb condicions.
- Té una API molt potent i és possible l'extensió de lògica d'aquesta, com veurem més endevant.

Aquesta eina ens ha permès crear tots els enemics del videojoc. A continuació expliquem alguns dels conceptes més essencials d'aquesta eina.

7.7.1 Task

En el nivell més simple, els arbres de comportament són una col·lecció de tasques. Hi ha quatre tipus diferents de tasques: acció, condicional, compost i decorador.

Les tasques d'acció són probablement les més fàcils d'entendre, simplement alteren l'estat del joc d'alguna manera. Les tasques condicionals avaluen propietats del joc. Per exemple, l'arbre de la Figura 7.8, té dues tasques condicionals i dues tasques d'acció. Les dues primeres tasques condicionals comproven si hi ha un enemic a la vista de l'agent i després s'assegura que l'agent tingui prou bales per disparar la seva arma. Si ambdues condicions són certes, s'executaran les dues tasques d'acció. Una de les tasques d'acció dispara l'arma i l'altra tasca reprodueix una animació de tir. El poder real dels arbres de comportament entren en joc si s'ajunten múltiples subarbres. Les dues accions de tir podria formar un subarbre. Si una de les tasques condicionals anteriors falla, es podria fer executar un altre subarbre realitzen un conjunt de tasques diferents d'acció, com ara fugir de l'enemic.

Les tasques compostes són tasques principals que contenen una llista de tasques secundàries. En la Figura 7.8 hi ha dues tasques compostes, aquestes són les tasques, seqüència i paral·lel. La tasca de seqüència executa cada tasca una vegada fins que s'hagin executat totes les tasques. Primer executa la tasca condicional que comprova si hi ha un enemic a la vista. Si un enemic està a la vista, executarà la tasca condicional que verifica si a l'agent li queden bales. Si l'agent té prou bales, la tasca paral·lela executa la tasca de disparar l'arma i reprodueix l'animació de tir. On una tasca de seqüència executa una tasca secundària alhora, una tasca paral·lela executa tots els seus fills alhora. Hi han moltes més tasques de tipus compostes que venen per defecte en Behavior Designer.

L'últim tipus de tasca és la tasca de tipus decorador. La tasca de tipus decorador és una tasca principal que només pot tenir un fill. La seva funció és modificar d'alguna manera el comportament de la tasca filla. En la Figura 7.8 no hi ha cap tasca de tipus decorador.

7.7.2 Task Status

Un dels principals temes de l'arbre de comportament que hem deixat de banda fins ara és l'estat de retorn d'una tasca. És possible que tinguem una tasca que triga més d'un fotograma a completar-se. Per exemple, la majoria de les animacions no comencen i acaben en un sol fotograma. A més, les tasques necessiten una manera de dir-li a la tasca principal si la condició era certa o no. Així, la tasca principal pot decidir si ha de continuar executant els seus fills o no. Aquest problema és resolt mitjançant l'estat de la tasca. Una tasca es troba en un dels tres estats diferents: corrent, èxit o fracàs.

7.7.3 Task API

Les tasques tenen una API similar a Unity MonoBehaviour 7.5.2, això ens va simplificar molt escriure les nostres pròpies tasques.

- OnAwake();

  OnAwake es crida una vegada quan l'arbre de comportament està habilitat. Semblant a un constructor.

- OnStart();

  OnStart es crida immediatament abans de l'execució. S'utilitza per configurar qualsevol variable que s'hagi de restablir de l'execució anterior.

- TaskStatus OnUpdate();

  OnUpdate execució de la tasca per cada fotograma.

- OnEnd();

  OnEnd es crida després de l'execució en cas d'èxit o fracàs.

- OnPause(bool paused);

  OnPause es crida quan el comportament es posa en pausa o es reprèn.

- float GetPriority();

  Retorna la prioritat de la tasca, utilitzada pel selector de prioritats.

- OnBehaviorComplete();

  OnBehaviorComplete es crida després que l'arbre de comportament s'acabi d'executar.

- OnReset();

  L'inspector crida OnReset per restablir les propietats públiques

- Behavior Owner;

  Mante una referència al comportament propi d'aquesta tasca.

El diagrama de la Figura 7.9 representa el cicle de vida d'un tasca de Behavior Designer.

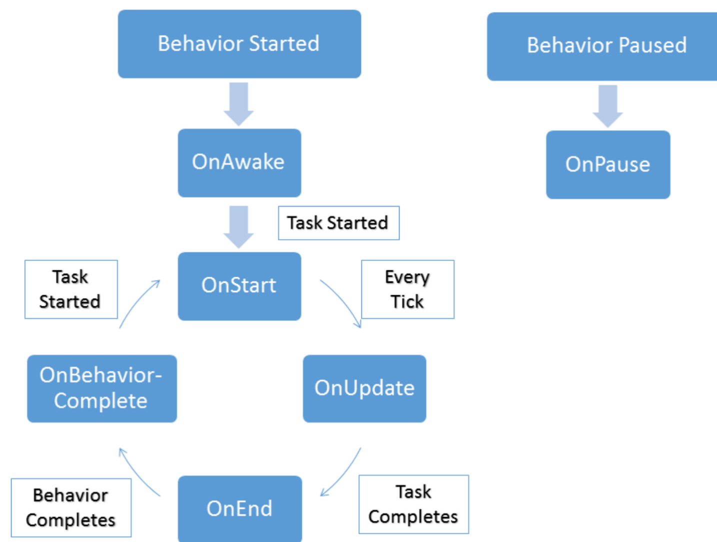

Figura 7.9: Cicle de vida d'una tasca.

7.7.4 Parent Task API

Les tasques de pares, també conegudes com a tasques compostes o tasques decoradores vistes en l'apartat 7.7.1, tenen API més ampla i complex ja que han de poder gestionar l'execució de les seves tasques filles.

- int MaxChildren();

  El nombre màxim de fills que pot tenir una tasca pare. Normalment, serà 1 o int.MaxValue

- bool CanRunParallelChildren();

  Valor booleà per determinar si la tasca actual és una tasca paral·lela.

- int CurrentChildIndex();

  L'índex de l'actual tasca filla en execució.

- bool CanExecute();

  Valor booleà per determinar si la tasca actual es pot executar.

- TaskStatus Decorate(TaskStatus status);

  Aplicar un decorador a l'estat de l'execució filla.

- OnChildExecuted(TaskStatus childStatus);

  Notifica a la tasca pare que el fill s'ha executat.

- OnChildStarted();

  Notifica a la tasca pare que el fill ha començat a executar-se.

- OnChildStarted(int childIndex);

  Notifica a la tasca pare que el fill en un índex ha començat a executar-se.

7.7.5 Variables

Un dels avantatges dels arbres de comportament és que són molt flexibles, ja que les tasques estan vagament acoplades, és a dir, una tasca no depèn d'una altra per operar. L'inconvenient d'això és que de vegades es necessiten que certes tasques comparteixin informació. Per exemple, és possible que tinguem una tasca que determini si un objectiu és a la vista. Si l'objectiu és a la vista, és possible que tinguem una altra tasca per moure's cap a l'objectiu. En aquest cas les dues tasques s'han de comunicar entre elles, de manera que la tasca de moure's faci que la intel·ligència artificial es mogui en direcció a l'objectiu detectat en la tasca de detecció. En un arbre de comportament tradicional això es resol codificant. Amb Behavior Designer és molt més fàcil perquè s'utilitzen variables compartides.

Per exemple, podríem definir la següent variable i compartir-la amb les dues tasques esmentades prèviament.

```
public SharedTransform target;
```

Behavior Designer te un munt de variables compartides creades per defecte. Hi ha casos, però, que necessiten definir els nostres propis tipus de variables compartides. Per exemple, per defecte no existeix una variable compartida de tipus Collider2D. A continuació, demostrem com es pot crear una variable compartida per Behavior Designer de tipus Collider2D.

```
using BehaviorDesigner.Runtime;
using UnityEngine;

namespace Core.IA.Bahavior.SharedVariable
{
    [System.Serializable]
    public class SharedCollider2D :
        SharedVariable<Collider2D>
    {
        public static implicit operator
            SharedCollider2D(Collider2D value)
        {
            return new SharedCollider2D { Value = value };
        }
```

```
    }
}
```

Podem declarar llavores, una variable compartida de tipus Collider2D com:

```
public SharedCollider2D collider;
```

Es poden fer filigranes i crear variables compartides de tota mena, tot i que les variables compartides de col·leccions genèrica comporten problemes, ja que funcionen en desenvolupament, però a l'hora de fer la construcció del projecte generen problemes i no funcionen.

7.7.6 Conditional Aborts

Per saber perquè són útils els avortaments per condicions, primer hem de saber que l'execució típica d'un arbre de comportament és d'adalt a baix i d'esquerra a dreta. Això pot provocar que, si ens quedem en un subarbre pou, és a dir un subarbre que s'executa per sempre, no podrem mai tornar a executar els subarbres de més a l'esquerra.

Els avortaments per condició permeten que l'arbre de comportament respongui dinàmicament gràcies als diferents tipus d'avortaments que s'apliquen únicament sobre les tasques de tipus composició.

Hi ha 3 tipus d'avortaments per condició són:

- Lower priority, s'avalua quan està activa qualsevol tasca a la dreta del subarbre actual.
- Self, s'avalua quan qualsevol tasca dintre del mateix subarbre està activa
- Both, s'avalua quan qualsevol tasca de la dreta o del mateix subarbre està activa

7.7.7 Behavior Trees o Finite State Machines

Els arbres de comportament tenen alguns avantatges respecte als FSM[8]; proporcionen molta flexibilitat, són molt potents i són molt fàcils de fer-hi canvis.

El primer avantatge de l'arbre de comportament és la flexibilitat. Amb un FSM, com s'executen dos estats alhora? L'única manera és crear dos FSM separats. Amb un arbre de comportament l'únic que s'ha de fer és afegir la tasca de

[8] Finite State Machine

paral·lelització i ja està: totes les tasques secundàries s'executaran en paral·lel.

Un altre avantatge dels arbres de comportament és que són potents. Això no vol dir que els FSM no ho siguin, és només que són potents de diferents maneres. Al nostre punt de vista, un arbre de comportament permet que la intel·ligència artificial reaccioni a l'estat actual del joc més fàcil que les màquines d'estats finits. És més fàcil de crear un arbre de comportament que reaccioni a tota mena de situacions, mentre que caldrien molts d'estats i transicions amb una màquina d'estats finits per tal de tenir una intel·ligència artificial similar.

Un últim avantatge de l'arbre de comportament és que són molt fàcils de fer-hi canvis. Un dels motius pels quals els arbres de comportament es van fer tan populars és perquè són fàcils de crear amb editors visuals. Si es vol canviar l'ordre d'execució estatal amb un FSM, s'han de canviar les transicions entre estats. Amb un arbre de comportament, tot el que s'ha de fer és arrossegar la tasca a una posició diferent. No ens hem de preocupar per les transicions.

Dit això, els arbres de comportament i els FSM no han de ser mútuament exclusius. Un arbre de comportament pot descriure el flux de la intel·ligència artificial mentre que el FSM pot descriure la funció.

7.8 Ilustrator

Illustrator és un editor gràfic vectorial desenvolupat per Adobe Systems. Va ser la nostra eina principal de desenvolupament d'Assets, el personatge principal, els NPC que apareixen, algunes decoracions, tots els elements que es veuen en els menús i moltes coses més estan fetes amb gràfics vectorials gràcies a aquesta eina.

Ilustrator, juntament amb Spine, van permetre donar vida al joc. Ilustrator és un sistema que treballa per capes, mentre que Spine treballa amb imatges separades. Per poder fer aquest procés automàtic, vam descarregar un Script que l'empresa desenvolupadora de Spine facilita perquè les exportacions de les creacions d'Ilustrator a Spine siguin senzilles [EsotericSoftware 022]. Veure Figura 7.10.

7.9 Spine

Spine és una eina d'animació que se centra específicament en l'animació 2D per a videojocs. Spine pretén tenir un flux de treball eficient, tant per crear animacions amb l'editor com per modificar-les en runtime amb l'API de Spine.

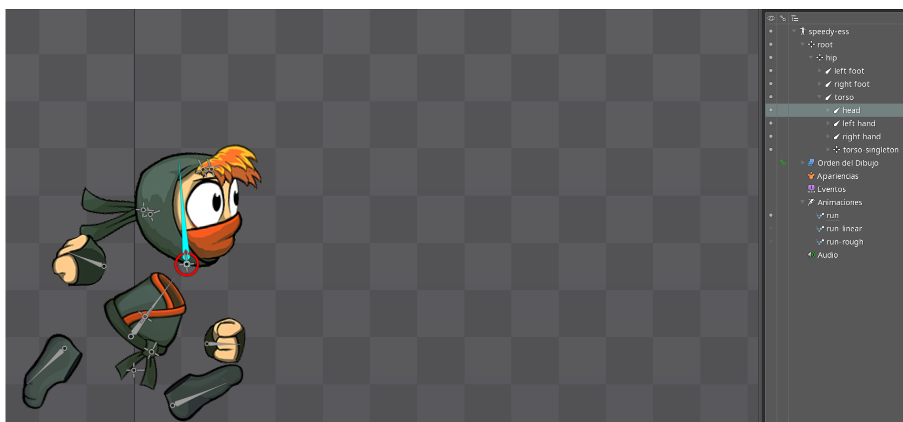

Figura 7.10: Exemple d'animació per ossos en Spine.

Beneficis d'utilitzar aquesta eina:

- L'animació a Spine es realitza adjuntant imatges als ossos i després animant-los. Això s'anomena animació per ossos i té molts avantatges respecte a l'animació tradicional fotograma a fotograma.

- L'animació tradicional requereix una imatge per a cada fotograma d'animació. Les animacions de Spine només emmagatzemen les dades de les posicions i temps relatiu dels ossos en un fitxer, la qual cosa fa que l'empaquetament de les animacions resultat sigui més lleugera.

- Les animacions de Spine requereixen molts menys recursos d'art, alliberant temps i diners.

- Spine utilitza interpolació[9], de manera que l'animació és sempre tan suau com la velocitat de fotogrames. Les animacions es poden reproduir a càmera lenta sense pèrdua de qualitat.

- Les imatges adherides als ossos es poden intercanviar per equipar un personatge amb diferents elements i efectes. Les animacions es poden reutilitzar per a personatges amb un aspecte diferent, estalviant innombrables hores.

- Els ossos es poden manipular a través de codi, permetent efectes com disparar cap a la posició del ratolí, mirar cap als enemics propers entre altres.

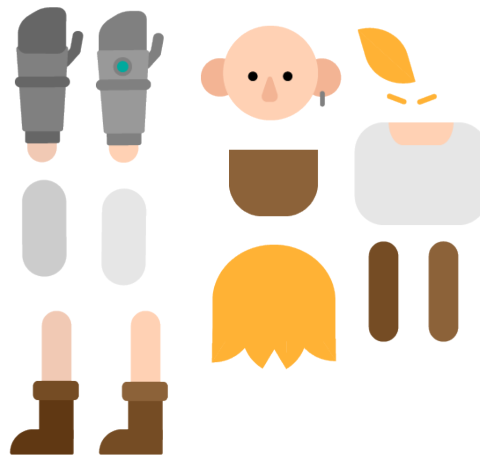

Figura 7.11: Personatge, separat per parts, llest per ser animat amb Spine.

Spine és una eina de pagament, té dues versions, la professional que permet treballar amb malles i la bàsica que no té la característica de poder treballar amb malles. Nosaltres vam utilitzar la versió bàsica, ja que trobàvem que era suficient per al caràcter de la producció.

[9] És l'obtenció de noves dades a partir d'un nombre discret de dades originals.

La raó de treballar amb Spine respecte a altres alternatives gratuïtes és que Spine és un projecte que està en continu desenvolupament i les alternatives gratuïtes a aquest, o bé no tenen un desenvolupament actiu, o bé hi ha poca informació i pobre documentacions. Un altre punt a favor de Spine és que té un seguit de llibreries de suport per a Unity. Aquestes llibreries ens permeten crear les màquines d'animació dintre de la mateixa Unity molt fàcilment, o permet modificar característiques pròpies de les animacions, així com afegir esdeveniments a cadascuna de les animacions.

7.10 Git

Git és un sistema de control de versions distribuït gratuït i de codi obert dissenyat per gestionar tot tipos de projectes, des de petits fins a molt grans amb rapidesa i eficiència.

Git el vam trobar indispensable pel desenvolupament, ja que permet treballar paral·lelament i de forma asíncrona. A més, és un mecanisme segur, perquè es pot accedir a versions anteriors del mateix projecte en cas de problemes.

7.11 GitHub

GitHub és un servei basat en el núvol que corre sobre el sistema de control de versions Git, veure apartat 7.10. Aquest permet als desenvolupadors col·laborar i realitzar canvis en projectes compartits, a la vegada que manté un seguiment detallat del seu progrés.

GitHub no el vam utilitzar únicament com a lloc per pujar i compartir el nostre codi, també el vam utilitzar per gestionar el treball i per fer la publicació del nostre joc mitjançant les conegudes GitHub Actions i GitHub Pages.

A continuació detallarem alguna de les característiques que fan a GitHub especial.

7.11.1 Issues

Un Issue és una nota en un repositori que tracta de cridar l'atenció sobre un problema. Pot ser un error a corregir, una petició per afegir una nova opció o característica, una pregunta per aclarir d'algun tema que no està aclarit correctament o moltes altres coses diferents. GitHub ens permet etiquetar, cercar o

assignar Issues, fent que la gestió d'un projecte actiu sigui més senzilla.

El sistema de seguiment d'Issues de GitHub és especial perquè està enfocat en la col·laboració, les referències entre documents i que cada Issue pot ser un petit fòrum per intercanviar opinions.

7.11.2 Pull Request

El PR[10] és el nucli del sistema col·laboratiu de GitHub. Quan fem una PR, el que estem fent en realitat és proposar canvis o afegir funcionalitats perquè algú ho integri dins del projecte. GitHub ens permet comparar el contingut de les PR a l'objectiu de manera que amb un senzill codi de colors podem veure les diferències.

Les PR estan pensades per resoldre Issues, a l'hora de fer una PR podem dir quines Issues es tanquen en cas que la Pull Request sigui acceptada i automàticament aquestes Issues passen a ser resoltes.

7.11.3 GitHub Projects

GitHub Projects, és una eina que s'utilitza per fer seguiments de les activitats del projecte fent ús de les Issues de GitHub i PR. En definitiva es fa seguiment de propostes, sol·licituds de canvis, incorporació de nou codi, i notes que es classifiquen com a targetes dintre de diferents columnes. Les columnes representen l'estat d'activitat de l'Issue. Per exemple, una Issue podria estar en procés de treball, o podria estar acabada, podem crear tantes columnes com cregui'm necessari i classificar les targetes segons l'estat. Podem arrossegar i deixar les targetes dins d'una columna, moure targetes de columna a columna i canviar l'ordre de les columnes.

Una de les característiques més important de GitHub Projects és el procés d'automatització que incorpora, es poden programar certs esdeveniments d'Issues, això implica que per exemple, a l'hora d'obrir una nova Issue, aparegui una nova carta amb la seva metadada en la columna corresponent.

[10] Pull Request

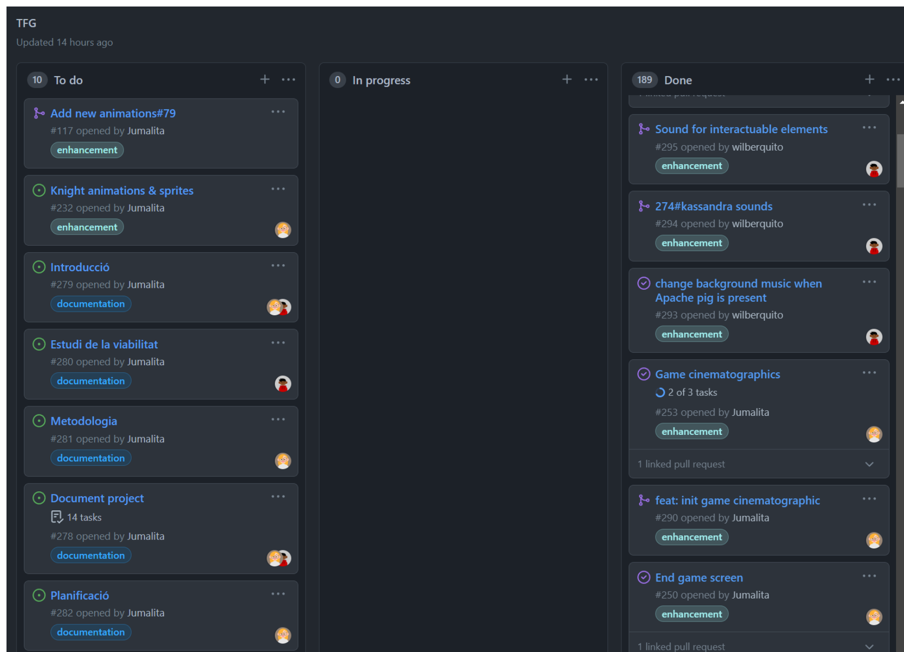

Figura 7.12: Exemple GitHub Projects

Github Projects es basa en la metodologia Kanban. Kanban és una forma d'ajudar als equips a trobar un equilibri entre el treball que necessiten fer i la disponibilitat de cada membre de l'equip. La metodología Kanban es basa en una filosofia centrada en la millora contínua, on les tasques surten d'una llista d'accions pendents en un flux de treball constant.

La metodologia Kanban s'implementa pel mitjà de taula Kanban. Es tracta d'un mètode visual de gestió de projectes que permet als equips visualitzar els seus fluxos de treball i la càrrega de treball. En una taula Kanban, el treball es mostra en un projecte en forma de taula organitzada per columnes. Tradicionalment, cada columna representa una etapa del treball. La taula Kanban més bàsica pot presentar columnes com a Treball pendent, en progrés i Terminat. Les tasques individuals — representades per targetes visuals a la taula — avançaran a través de les diferents columnes fins que estan finalitzades.

Al mercat hi ha diverses alternatives a Github Projects, altres gestors de projectes coneguts i molt usats són: Jira, Slack, Trello, entre altres. La raó de treballar amb Github Projects és que s'incorpora perfectament amb el flux de treball dels repositoris de Github, i d'aquesta manera, ho tenim tot centralitzat.

7.11.4 GitHub Pages

GitHub Pages és un servei de GitHub que ens permet oferir els nostres projectes i mostrar-los en viu en una pàgina web estàtica sense necessitat de pagar per allotjament o tenir coneixements de manteniments de servidors web o DevOps.

Ja que sabíem que Unity ens permet exportar el resultant a plataforma web, es a dir transformar el nostre joc a contingut web estàtic com és HTML, JavaScript i CSS, vam explotar aquesta eina de GitHub per publicar el joc i d'aquesta manera poder compartir-lo. S'ha de dir, però, que GitHub només permet tenir una pàgina publicada per compte, si es tenir més d'una pàgina, s'ha de pagar.

Al mercat hi ha moltes alternatives al mercat a GitHub Pages, per mencionar una; Heroku. Heroku és un servei en el núvol que permet publicar contingut web gratuïtament. De la mateixa manera que passa amb GitHub Projects, GitHub permet treballar de manera íntegra amb GitHub Pages, i per això hem optat per aquesta solució.

7.11.5 GitHub Actions

GitHub Actions és una eina que permet reduir la cadena d'accions necessàries per a l'execució del codi, mitjançant la creació d'un flux de treball. Es pot fer que GitHub reaccioni a certs esdeveniments de forma automàtica segons les nostres preferències.

Per tant, GitHub Actions permeten crear fluxos de treball que es puguin utilitzar per compilar, provar i desplegar codi. A més, dona la possibilitat de crear fluxos d'integració i desplegament continu dins del nostre repositori. Nosaltres vam fer servir Github Actions únicament com a flux de desplegament continu. Un altre punt important de les GitHub Actions és que es poden utilitzar accions creades per tercers dintre dels workflow.

Aquesta funcionalitat és gratuïta per a tots els dipòsits de codi obert i inclouen 2000 minuts al mes de compilació sense cost per als dipòsits privats. Si, al contrari, això no és suficient per a les nostres necessitats, hi ha plans de pagament.

A continuació presentem alguns conceptes per ajudar a entendre el funcionament i utilitat de GitHub Actions:

7.11.5.1 Step

Es compon d'un conjunt de tasques per poder executar un Job, veure Apartat 7.11.5.2. Aquests poden executar ordres o accions.

7.11.5.2 Job

És un conjunt de passos que s'executa en un Runner. Els Jobs poden executar-se de forma independent o seqüencial depenent de si l'èxit del nostre treball depèn de l'anterior.

7.11.5.3 Workflow

És un procediment automatitzat compost per un o diversos Jobs que s'afegeix a un repositori i es pot activar per un esdeveniment. Es defineix mitjançant els arxius YAML. Dintre d'aquests fitxers es defineix la compilació, prova, importació de paquets, llançament o desplegament d'un projecte.

7.11.5.4 Event

Són activitats específiques que desencadenen l'execució d'un Workflow. Aquest esdeveniment poden ser la creació, acceptació o modificació d'una PR, afegir un Tag[11] per tal de versionar el nostre codi, entre moltes altres.

7.11.5.5 Action

És el bloc de construcció més petit d'un flux de treball i es poden combinar com a passos per crear un Job.

7.11.5.6 Runner

És una màquina física en el núvol, generalment és un Linux amb l'aplicació de GitHub Actions ja instal·lada, la seva funció es executar les accions d'un workflow, a més, informa del progrés de les execucions i els resultats.

[11] En Git, una etiqueta o etiqueta serveix bàsicament com una rama firmada que no permuta, és a dir, sempre es manté inalterable.

Per poder fer el desplegament del joc primer l'hem de construir per a plataformes WebGL. Però hem de tenir una cosa clara, la construcció del joc no es fa sobre cap de les nostres màquines físiques sinó que es fa sobre alguna màquina del nuvol de GitHub, es a dir, sobre un Runner, veure Apartat 7.11.5.6. Obviament aquestes màquines no tenen perquè tenir les dependencies necesaries per poder fer la construcció de un projecte d'Unity. La solució a aquest problema és instal·lar un Action capaç de fer la construcció del projecte dintre del Runner.

Vam utilitzar Action de GameCI [GameCI 022] per dur a terme les tasques d'activació de llicencia i de construcció de projecte.

A continuació veurem el workflow d'activació de llicència d'Unity.

```
1  name: Get Unity license activation file
2
3  on: workflow_dispatch
4
5  jobs:
6    requestManualActivationFile:
7      name: Request manual activation fil
8      runs-on: ubuntu-latest
9      steps:
10       - uses: actions/checkout@v2
11
12       - uses: game-ci/unity-request-activation-file@v2
13         id: getManualLicenseFile
14
15       - uses: actions/upload-artifact@v2
16         with:
17           name: Manual Activation File
18           path: ${{ steps.getManualLicenseFile
19                     .outputs.filePath }}
```

El workflow d'activació de llicència s'executa manualment, ho podem veure en la línia 3. Aquest workflow només té un Job, aquest s'executa sobre un Ubuntu, mirar línia 8. Aquest Job quan acaba, utilitza un action de tercers, mirar línia 15 per poder pujar un fitxer al nostre repositori que posteriorment el descarreguem per poder fer l'activació de llicència en la pàgina oficial d'Unity [Unity 022d].

A continuació veuem el workflow de comprovació de llicència, construcció per a plataformes WebGL i publicació del joc fent ús de GitHub Pages.

```
name: Erlang-Legacy CI

on:
  push:
    tags:
      - '**'

env:
  UNITY_LICENSE: ${{ secrets.UNITY_LICENSE }}
  UNITY_EMAIL: ${{ secrets.UNITY_EMAIL }}
  UNITY_PASSWORD: ${{ secrets.UNITY_PASSWORD }}
  PROJECT_PATH: Erlang-Legacy

jobs:

  checkLicense:
    name: Check Unity license
    runs-on: ubuntu-latest
    steps:
      - name: Fail - No license
        if: ${{ !startsWith(env.UNITY_LICENSE, '<') }}
        run: exit 1

  buildWebGL:
    needs: checkLicense
    name: Build for WebGL
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2
        with:
          lfs: true

      # Cache
      - name: Cache dependencies
        uses: actions/cache@v2
        with:
          path: ${{ env.PROJECT_PATH }}/Library
          key: Library-${{ hashFiles('${{ env.PROJECT_PATH }}/Assets/**', '${{ env.PROJECT_PATH }}/Packages/**', '${{ env.PROJECT_PATH }}/ProjectSettings/**') }}
          restore-keys: |
            Library-

      # Build
```

```
      - name: Build project
        uses: game-ci/unity-builder@v2
        env:
          UNITY_LICENSE: ${{ env.UNITY_LICENSE }}
          UNITY_EMAIL: ${{ env.UNITY_EMAIL }}
          UNITY_PASSWORD: ${{ env.UNITY_PASSWORD }}
        with:
          projectPath: ${{ env.PROJECT_PATH }}
          targetPlatform: WebGL

      # Output
      - name: Upload artificat
        uses: actions/upload-artifact@v2
        with:
          name: build-WebGL
          path: build/WebGL

  deployPages:
    needs: buildWebGL
    name: Deploy to Github Pages
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Download artificat
        uses: actions/download-artifact@v2
        with:
          name: build-WebGL
          path: build

      - name: Deploy
        uses: JamesIves/github-pages-deploy-action@4.1.4
        with:
          branch: gh-pages
          folder: build/WebGL
```

Aquest últim workflow s'executa quan hi ha una nova versió del joc. El joc està versionat pels tags de Git. Veure de línia 3 a línia 6.

Per poder configurar més còmodament aquest workflow vam fer ús de variables d'entorn. Per exemple la variable d'entorn de la línia 9 representa el codi de llicència, de la 10 a la 11 l'usuari i contrasenya del propietari de la llicència. Sense aquesta informació no podem utilitzar les Action de GameCI, ja que requereixen autenticació de llicència.

Com podem veure en les línies 16, 24 i 61, aquest workflow està format de 3

accions diferents. La primera acció, la de la línia 16, s'encarrega de verificar que la llicència d'Unity està present en el nostre repositori. La segona llicència que comença en la línia 24, s'encarrega de fer la construcció amb l'accio de GameCI, UnityBuilder, mirar línia 45. I, finalment, l'última acció que comença en la línia 61, s'encarrega d'agafar el resultat de la construcció i publicar-lo, en la branca del projecte que està configurada per fer allotjament de contingut estàtic.

7.12 Diagrams.net

Diagrams.net és un programari de dibuix de gràfics multiplataforma gratuït i de codi obert desenvolupat en HTML5 i JavaScript. La seva interfície es pot utilitzar per crear diagrames com ara diagrames de flux, diagrames UML, organigrames i diagrames de xarxa. Tots els diagrames UML, diagrames de flux que apareixen en la documentació han estat creats sobre aquesta plataforma.

7.13 LaTeX

LaTeX, que es pronuncia és un sistema de preparació de documents per a una composició tipogràfica d'alta qualitat. S'utilitza més sovint per a documents tècnics o científics de mitjana a gran mida, però es pot utilitzar per a gairebé qualsevol forma de publicació.

La raó d'escollir aquest Programari sobre altres editors de textos per a la documentació, és que LaTex és d'ús gratuït, i ens agrada la manera que ens permet treballar, ja que no ens hem de preocupar gaire sobre l'aparença del document resultant i ens permet concentrar-nos a obtenir el contingut adequat. Un punt a favor de LaTex en comparativa a la típica alternativa de documentació, Microsoft Word, és que el codi font està en format text i no és un binari. Això permet que l'ús de Git amb la documentació sigui més simple, i es puguin veure les diferències entre versions de documentació clarament.

CAPÍTOL 8

Anàlisi i disseny del sistema

En aquest capítol proporcionem els problemes resolts i les solucions per donar vida al projecte. En aquesta secció veurem les solucions als requisits funcionals del Capítol 6. Aquesta secció ens permet explicar conceptualment les bases del videojoc.

8.1 Descripció general

Els Metroidvania es decanten per seguir el patró següent: El jugador comença la partida amb un personatge i ha de travessar diferents escenes dissenyades prèviament pel director del joc, en aquest cas nosaltres, els desenvolupadors, de manera que no pot accedir a certes àrees sense haver aconseguit certes habilitats o objectes. Al llarg del camí, el jugador interactua amb objectes de l'escena, s'enfronta a enemics, es cura, guarda la partida en certs punts, cau sobre zones perilloses, interactua amb NPCs i aconsegueix habilitats o elements que permeten desbloquejar certes àrees. Un Metroidvania acaba quan un jugador acaba la història principal.

En el projecte, el personatge principal Ajax, completa la història quan es retroba amb la seva germana Cassandra. Per poder fer això necessita derrotar a un gran nombre d'enemics simples i Bosses[1] generats per intel·ligència artificial amb arbres de comportament creats amb la llibreria Behavior Designer 7.7, de la mateixa manera que ha de poder trobar les habilitats que li permetràn descobrir noves zones, interactuar amb el gran nombre de plataformes creades, esquivar obstacles i en definitiva navegar per les diferents escenes.

Un jugador del projecte ha de ser capaç de començar una nova sessió de joc, reprendre una sessió de joc existent, s'ha de poder interactuar amb la música del joc de tal manera que es pugui regular la música d'ambient i els efectes sonors, també ha de poder accedir al recordatori de tecles per tal de saber que fan les tecles en tot moment. Dintre del joc, hem de poder guardar l'estat actual de la

[1]En els videojocs, un Boss és un figura, sigui una criatura o un nivell, que es troba al final d'una secció i actua com el clímax d'aquest.

partida amb la interacció de punts de guardats. També hem de poder accedir a un menú per poder interactuar amb les configuracions prèviament esmentades o per poder sortir de l'actual sessió de joc.

8.2 Història i ambientació

En el món d'Erlang Legacy la guerra reina des de temps immemorials. Ajax ha estat víctima de la violència de la guerra. El seu pare adoptiu, el científic Mr. C, el va salvar quan sols era un nen atorgant-li les extremitats robòtiques que un atac enemic li havia pres. La gent del poble creu l'Erlang: un salvador d'un altre temps que tornarà a la vida i acabarà amb la guerra d'una vegada per totes. Ajax creix al costat de Cassandra, la seva germanastra, escoltant aquestes històries. Arran de la mort de Mr. C en un atac, Ajax decideix seguir els passos del seu heroi i parar l'enfrontament entre regnes.

La història del joc es remunta en els primers passos de l'Ajax per convertir-se en un heroi: Cassandra intenta tirar endavant el laboratori del seu difunt pare, però un problema succeeix i es queda atrapada dins. Ajax, que estava entrenant a una de les illes annexes al poble, farà el possible per rescatar-la.

8.3 Casos d'ús

En aquest apartat veurem els diagrames de casos d'ús plantejats i diagrames d'activitats d'algunes de les funcionalitats que es mostren en l'Apartat 8.5.

8.3.1 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.

8.3.1.1 Menu d'inici

Veure apartat 8.5.5.1.

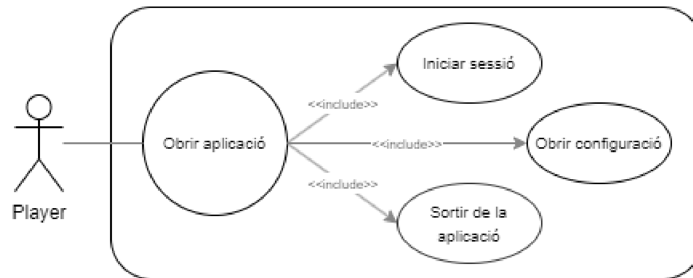

Figura 8.1: Diagrama de casos d'ús del menú d'inici.

8.3.1.2 Inici de sessió

Veure apartat 8.5.5.2.

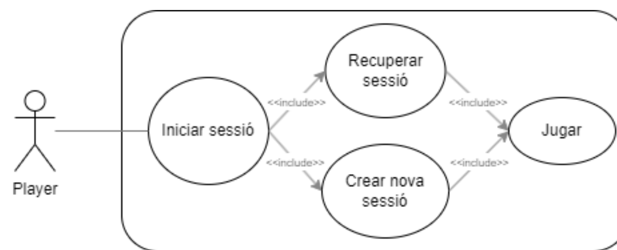

Figura 8.2: Diagrama de casos d'ús d'inici de sessió.

8.3.1.3 Dintre del joc

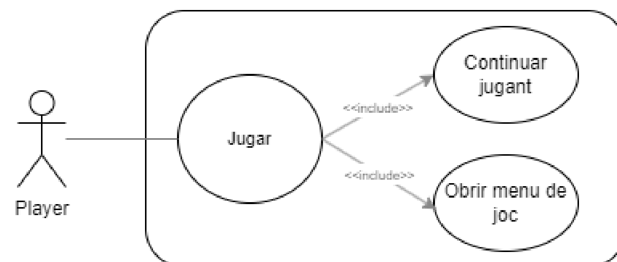

Figura 8.3: Diagrama de casos d'ús dintre del joc.

8.3.1.4 Menu de joc

Veure apartat 8.5.5.6.

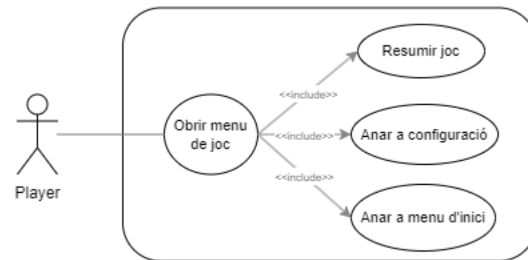

Figura 8.4: Diagrama de casos d'ús del menú de joc.

8.3.1.5 Menú de configuracions

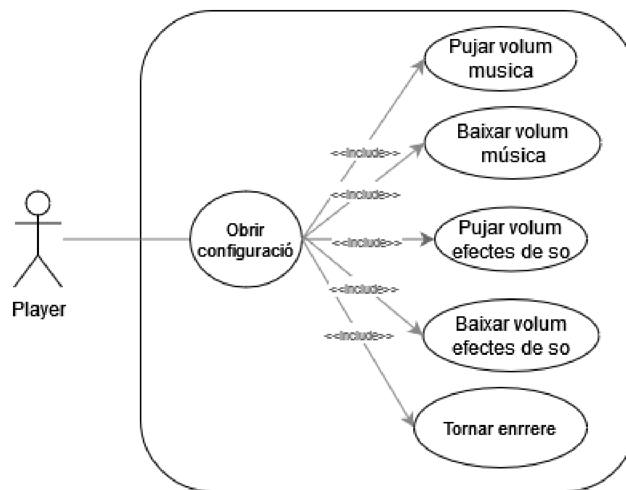

Figura 8.5: Diagrama de casos d'ús menú de configuracions.

8.3.2 Diagrama d'activitats

Els diagrames d'activitats són utils per representar la seqüencia de les activitats d'un procés. En aquest apartant veurem en alt nivell alguns dels procesos indispensables per la jugabilitat.

8.3.2.1 Ús d'habilitats

El diagrama d'activitats de la Figura 8.6 intenta expresar visualment el procés de l'ús de qualsevol habilitat del personatge principal, ja siguin les habilitats bàsiques que venen per defecte o les que s'adquereixen al llarg del videojoc. Veure apartat 8.5.1.

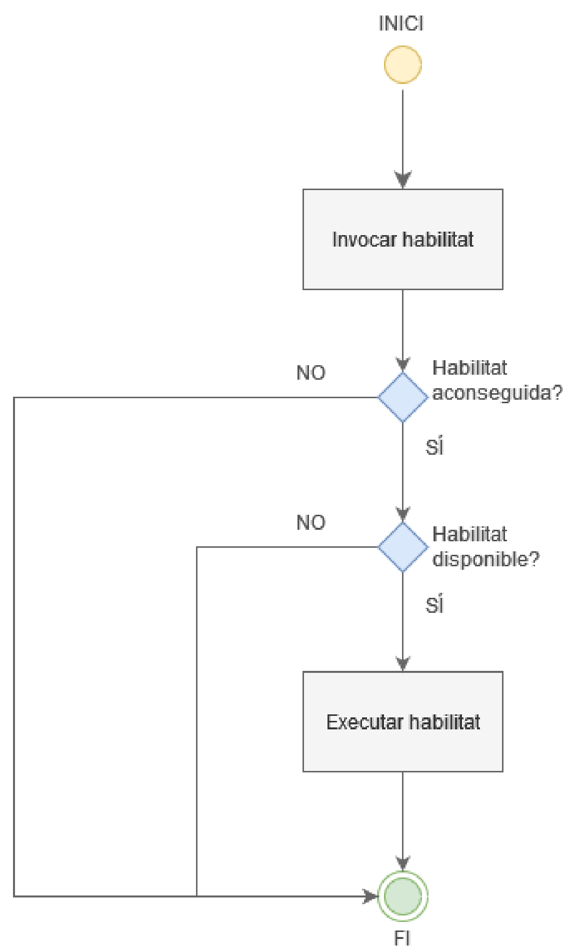

Figura 8.6: Diagrama d'activitats d'ús d'habilitats.

8.3.2.2 Interacció amb objectes que fan mal

El diagrama de la Figura 8.7 intenta expressar a alt nivell la gestió que es fa quan el personatge interactua amb algun objecte que li pot fer mal.

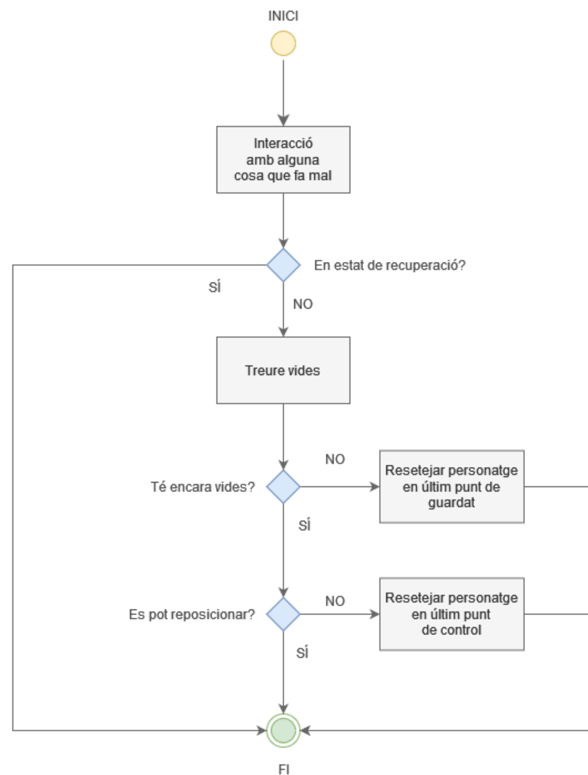

Figura 8.7: Diagrama d'activitats d'interacció amb objectes que fan mal.

8.3.2.3 Lluita per fases

En aquest apartat veurem a alt nivell com funcionen les diferents fases dels dos Bosses a través del diagrama 8.8. El dos Bosses comparteixen una mateixa estructura lògica de fases, però no d'implementació, ja que en les fases de lluita, el subconjunt d'habilitats disponibles per fase de lluita varia quant a cuantitat i tipus d'habilitat. Veure apartats 8.5.2.6 i 8.5.2.7.

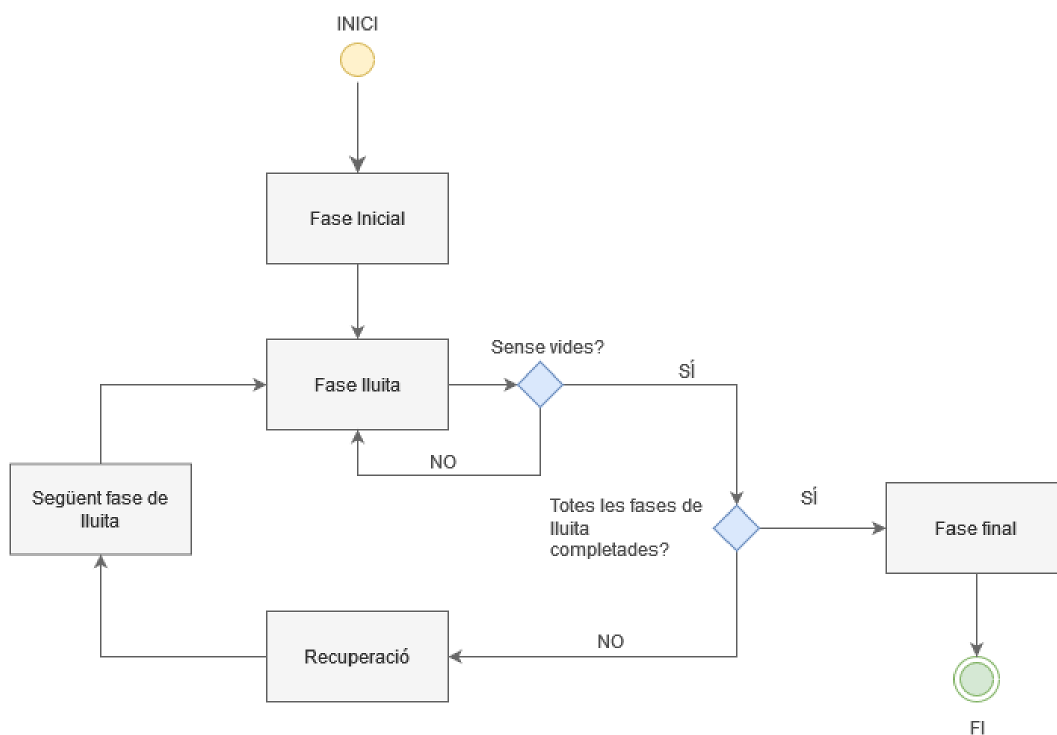

Figura 8.8: Diagrama d'activitats de fases d'un Boss.

8.3.2.4 Interacció amb diàlegs

En aquest apartat veurem el fluxe d'activitats 8.9 del procés d'interacció amb diàlegs en l'escena. Veure apartat 8.5.5.10.

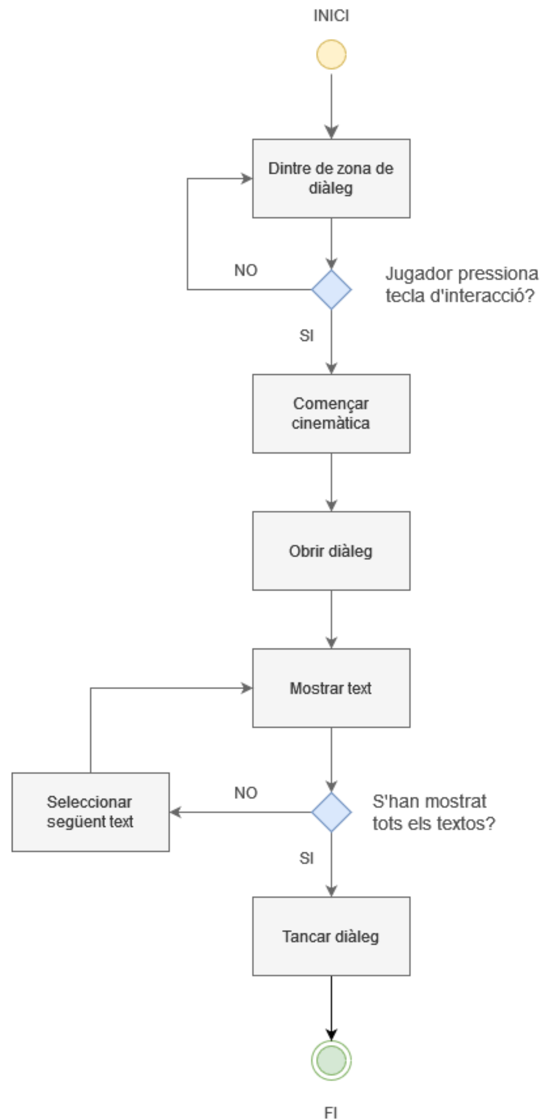

Figura 8.9: Diagrama d'activitats de diàleg.

8.3.2.5 Moviment Plataformes

En aquest apartat veurem el fluxe d'activitats 8.10 moviment d'una plataforma simple.

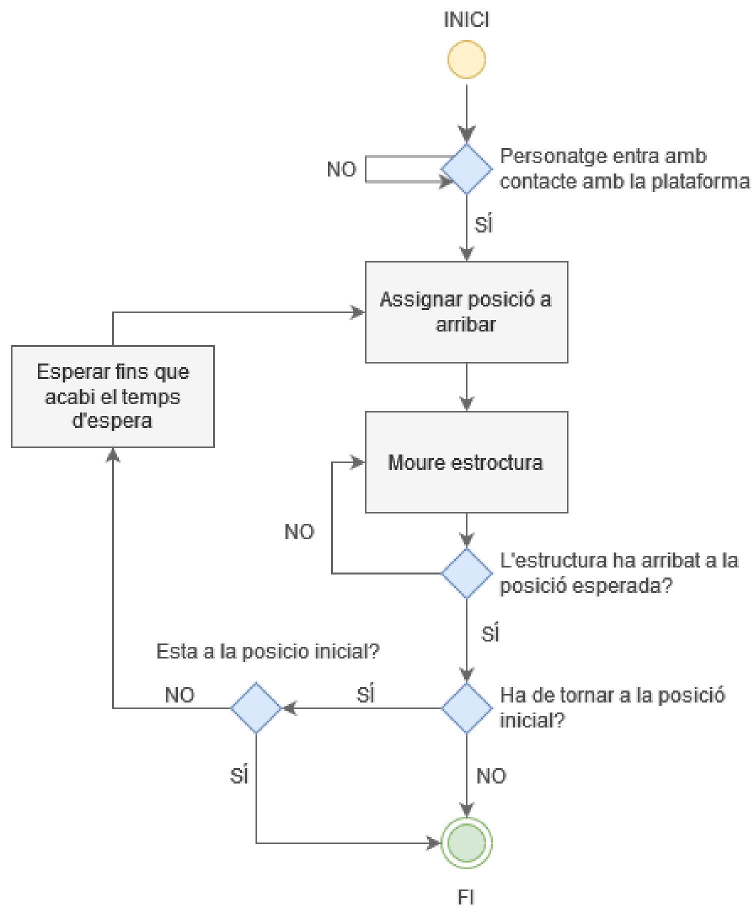

Figura 8.10: Diagrama d'activitats moviment plataforma simple.

8.3.2.6 Música del joc

Finalment, trobem interessant de representar com es gestiona la música del joc al fer canvis d'escenes. Veure figura 8.11.

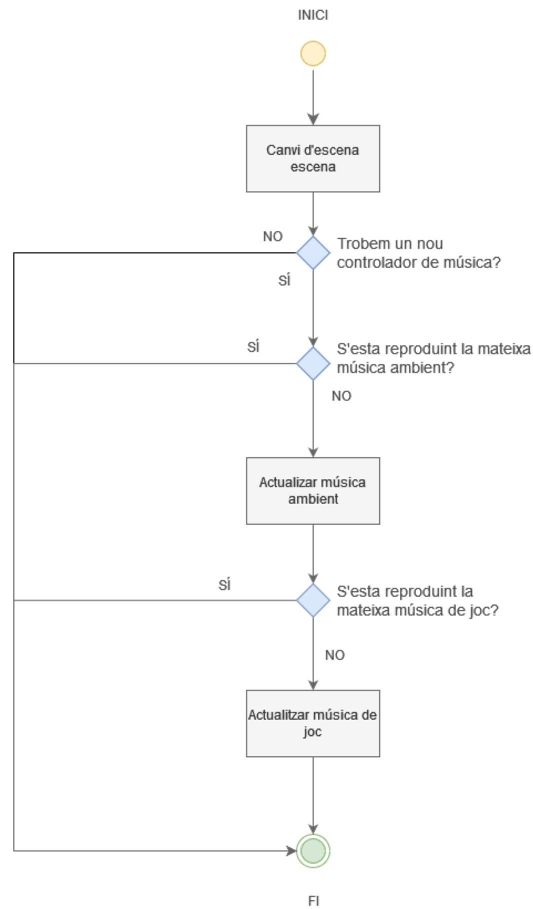

Figura 8.11: Diagrama d'activitats de música.

8.4 Model de clases

En aquest apartat exposarem les parts més interesants del diseny del model de clases del sistema.

8.4.1 Personatge principal

La lògica del personatge principal és molt ample. Vam trobar que la lògica del personatge podia ser separada en 5 components i una classe simple de dades. D'aquests components, 4 estan pensats per encapsular lògica i l'altre com orquestrador i interfície de comunicació per altres entitats que es vulguin comunicar amb el personatge principal. Finalment, la classe de dades serveix per guardar informació referent a l'estat del personatge, així com configuracions per defecte. Mirar els diagrames de classes de les Figures 8.12 i 8.13.

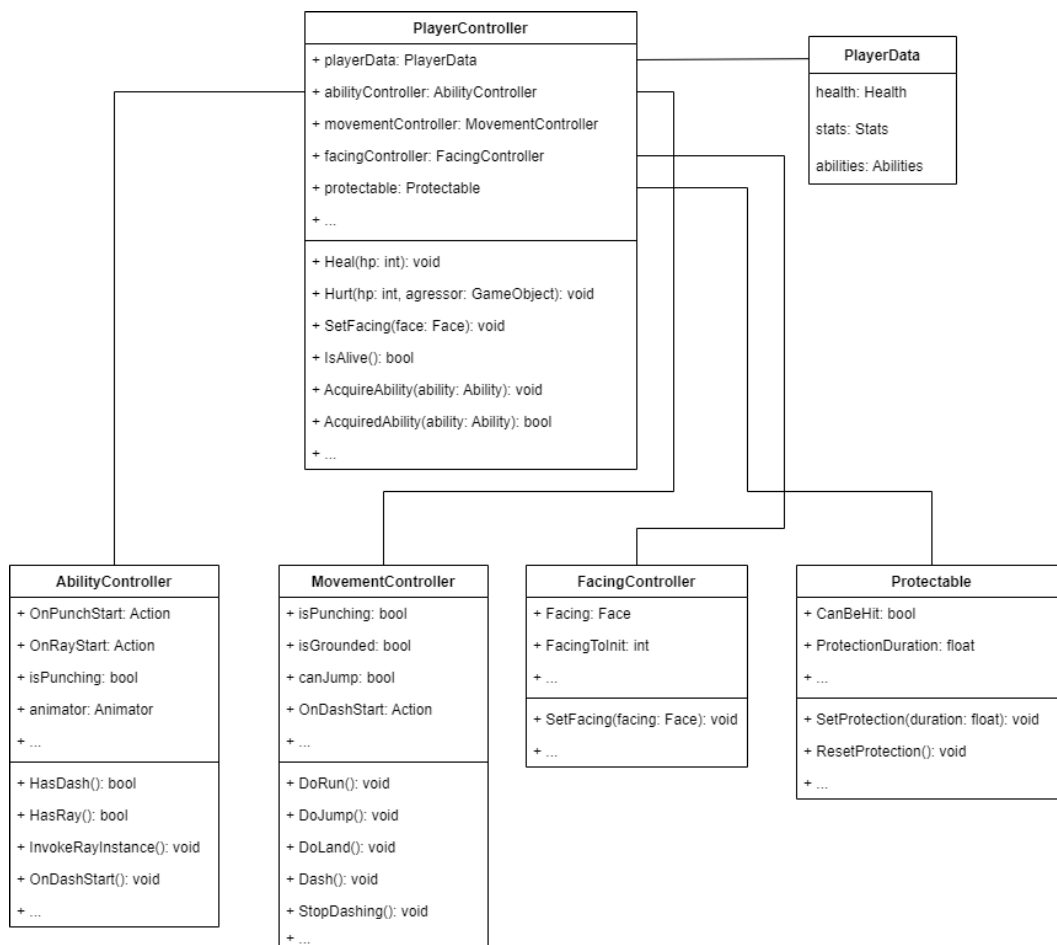

Figura 8.12: Model de clases del personatge principal.

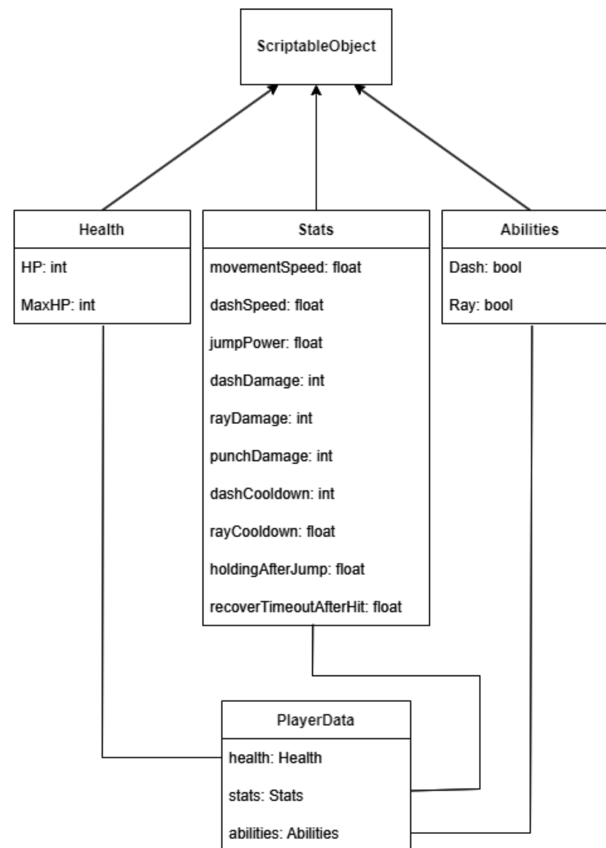

Figura 8.13: Model de clase de dades de l'estat del personatge principal.

- MovementController.

  Pensat per gestionar casos d'ús propis a la mobilitat del personatge, com ara córrer, saltar, fer desplaçament ràpid, fer que al personatge no l'afectin les físiques del joc, entre altres.

- AbilityController

  Script que gestiona la seqüència dels cops de punys i les habilitats aconseguides, entre altres.

- FacingController.

  Script que permet saber en tot moment cap a quin costat de l'eix horitzontal el personatge ha d'estar mirant.

- Protectable.

  La funcionalitat d'aquest component és gestionar el temps de protecció que ha de tenir el nostre personatge en cas d'haver rebut mal.

- PlayerController.

  Aquest script permet saber en tot moment des de qualsevol altra entitat l'estat del personatge principal, és a dir, podem saber si està saltant, fent un desplaçament ràpid, si està protegit i de més. A més té una API que permet que altres components interaccionin amb el personatge, com ara un mètode per fer-li mal, per curar-se, forçar cap a quin costat ha de mirar el personatge comunicant-se amb el script FacingController, fer adquirir una habilitat, entre d'altres.

- PlayerData.

  Guarda l'estat així com algunes configuracions pròpies del personatge. PlayerData és una classe que té referència a 3 classes de tipus ScriptableObject recordem que les instàncies d'aquest tipus de dades serveixen per encapsular dades i ajuden al desacoblament de lògica, ja que fan de recurs extern i poden ser utilitzats per diferents components del joc sense tenir dependència entre ells. Mirar Apartat 7.5.19 en cas de dubte.

8.4.2 Interacció de l'escena amb el personatge principal

A continuació veurem el diagrama de classes que hi ha darrera de l'interacció dels elements de l'escena amb el personatge. En aquest apartat veurem únicaments els casos ón el personatge es totalment pasiu, i són els elements de l'escena que modifiquen l'estat del personatge.

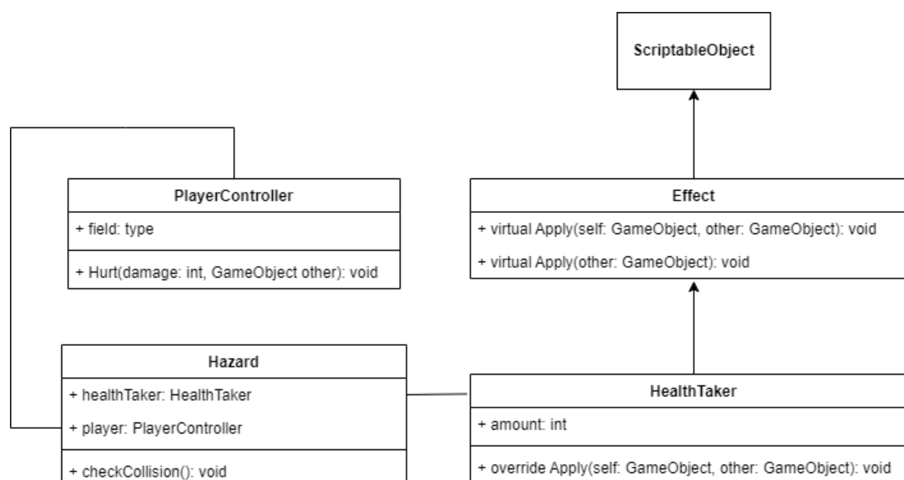

Figura 8.14: Model de classes per fer mal al personatge.

8.4.3 Disseny del enemics

Tots els enemics segueixen la mateixa arquitectura de model de clases. El que fa més o menys complex un enemics es l'implementació de l'inteligencia artificial amb els arbres de comportament. Mirar Figura 8.15.

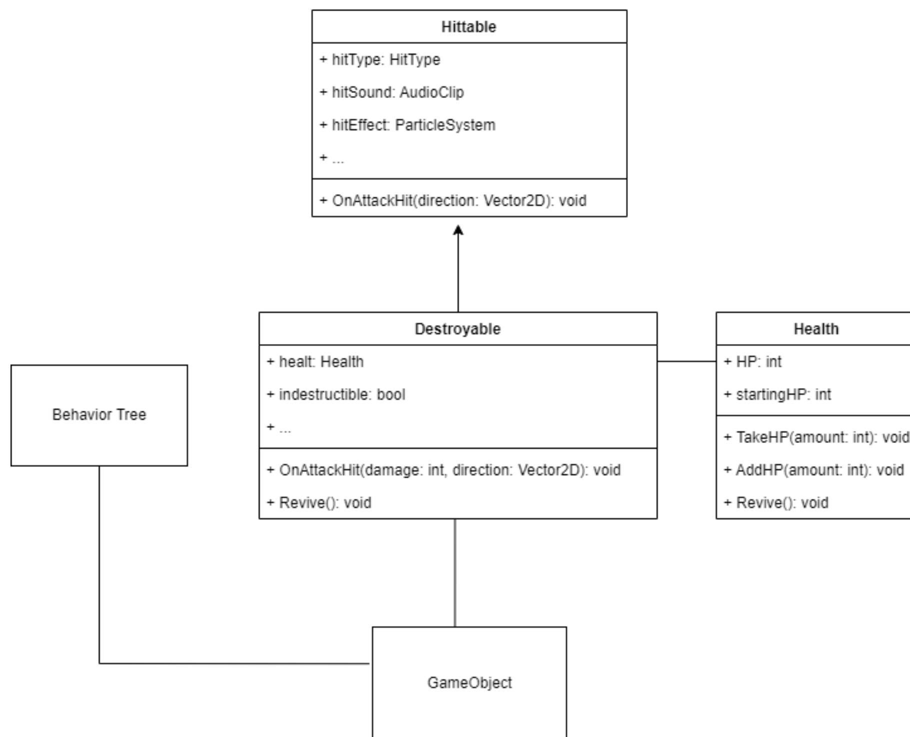

Figura 8.15: Diagrama de classes dels enemics.

A continuació explicarem el funcionament de cadascun d'aquests components.

- Hittable.

  Aquest component està pensat per afegir-lo a objectes d'escenes els quals el jugador pot picar. Els objectes que implementen aquest component gaudeixen d'una ampla varietat de reaccions als cops del personatge i és altament configurable.

- Destroyable.

  Aquest script està pensat per afegir-lo a objectes d'escena als quals, a més de poder picar-los i reaccionar als cops del personatge principal, cal un seguiment de l'estat de les seves vides. El seguiment de les seves vides es fa gràcies a una classe externa que el gestiona.

- Health.

  Aquest component gestiona les vides que tenen els enemics així com implementa altres funcionalitats com ara recuperar totes les vides que tenia d'un principi.

- Behavior Tree.

  Component que gestiona tota la lògica dels arbres de decisions, com ara l'ordre d'execució de tasques, avortaments d'execució de forma dinàmica, esdeveniments, entre altres.

L'ampla varietat d'enemics que té el projecte és pel nombre de tasques diferents creades i personalitzades i l'ordenació d'aquestes tasques en el cicle de vida d'execució de l'arbre.

8.4.4 Disseny de classes de gestió de sessió

En cada escena del joc hi ha la idea d'un gestor de la sessió. Algunes de les funcionalitats de l'objecte gestor de sessió són carregar les escenes i gestionar la intercomunicació entre els punts de sortides i entrades d'escenes, col·locar al personatge existent en l'escena en la posició i orientació adequada, també és l'encarregat de comunicar-se amb el sistema de guardat, per serialitzar i persistir l'estat del jugador i desserialitzar i recuperar l'estat del jugador i per últim guarda la posició de l'últim punt de control pel qual el personatge principal ha passat i per últim i moltes vegades gestiona les interfícies d'usuaris que apareixen en l'escena. Mirar Figura 8.16.

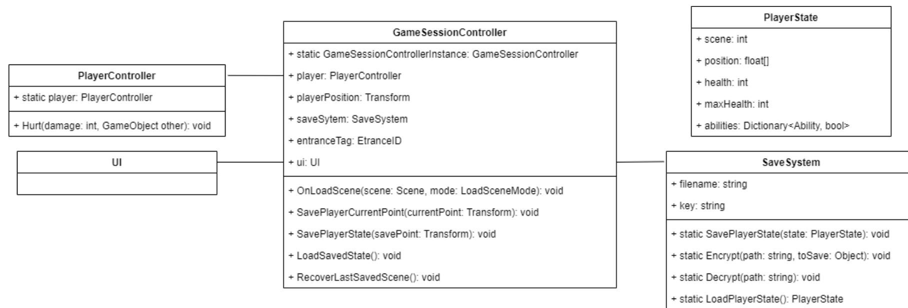

Figura 8.16: Diagrama de classes de la lògica de control de sessió de joc.

8.5 Disseny del funcionament

En aquest apartat veurem tots els mècanismes funcionals que conformen el joc, pasant des del personatge principal fins a les interficies d'usuari.

8.5.1 Personatge principal

Ajax és el personatge principal i és la forma amb què el jugador interactua amb el videojoc. Aquest personatge és una creació pròpia; ha estat dissenyat dintre del programari d'Illustrator i animat amb Spine.

Figura 8.17: Ajax.

Les accions que podem realitzar amb el personatge són les següents: atacar, córrer, fer desplaçaments llargs amb l'habilitat de Dash que, a la vegada, fa mal als enemics, saltar progressivament segons la durada d'interacció amb la tecla de salt, tirar un projectil que fa mal als enemics, rebre mal, entrar en mode recuperació on no rebrem mal mentre duri, curar-nos i finalment morir. Les habilitats Dash i de projectil són habilitats que s'aconsegueixen a mesura que s'arriben a certs punts d'algunes escenes. Per tant, s'ha de mantenir una lògica d'habilitats adquirides al llarg d'una sessió de joc.

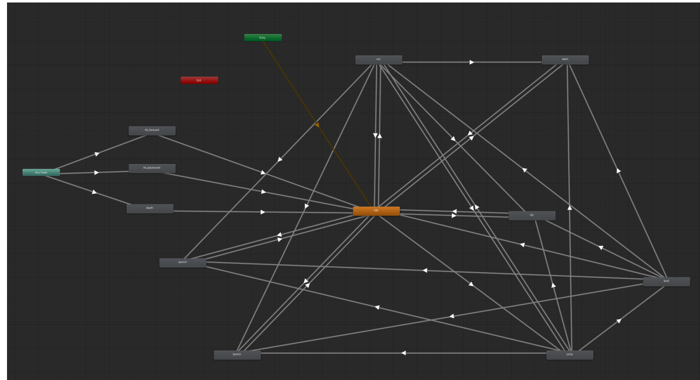

Figura 8.18: Màquina d'estats d'animacions de l'Ajax.

8.5.1.1 Atac principal

L'atac principal del personatge és fer cops amb els seus punys, tant amb la mà dreta com amb la mà esquerra. Està programat de tal manera que els punys segueixen la seqüència, esquerra després dreta o a la inversa perquè l'inici de ràfega és aleatori. Quan passa cert temps sense fer ús del puny, la seqüència es reinicia, i torna a donar-se l'aleatorietat del patró d'atac.

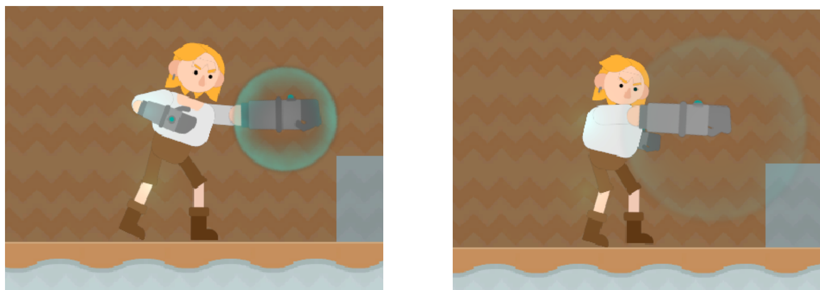

Figura 8.19: Sequencies atac principal.

8.5.1.2 Córrer

L'acció de córrer és el mitjà bàsic pel qual ens desplacem. El desplaçament és especial perquè no es fa a velocitat constant en tot moment del desplaçament, sinó que se li aplica un efecte de suavitat a l'hora de començar i deixar de moure's. Aquest efecte el que provoca és que hi hagi una sensació d'acceleració, donant sensació de dinamisme.

Figura 8.20: Desplaçament normal.

8.5.1.3 Salt

El salt del personatge principal és responsiu i depèn de quant temps es premi el botó de saltar. Clarament, no és infinit aquest salt, per tant, la durada del salt es calcula en temps de pressió i limitat per configuració.

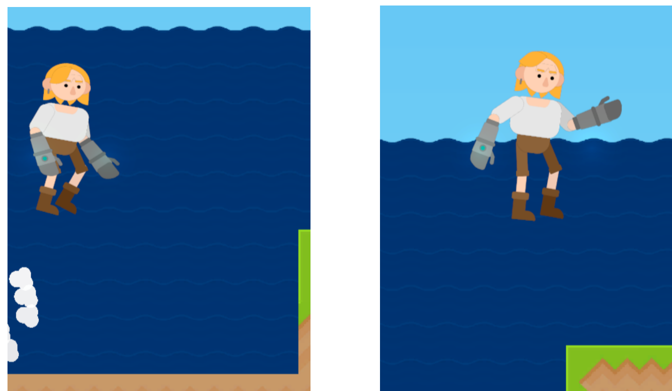

Figura 8.21: Desplaçament per salt.

8.5.1.4 Dash

El Dash és una habilitat que s'aconsegueix a mesura que anem jugant, sense aquesta habilitat és impossible arribar a certes àrees a causa de la forma en què s'ha dissenyat el joc.

Figura 8.22: Desplaçament per habilitat adquirida.

8.5.1.5 Vengeful Ray

Vengeful Ray és el nom de l'habilitat que ens permet llençar un projectil en l'eix X. El projectil està caracteritzat perquè travessa tots els enemics al seu pas i aplica mal. El projectil té un temps de vida, és a dir, que es desvaneix sol al cap d'un temps.

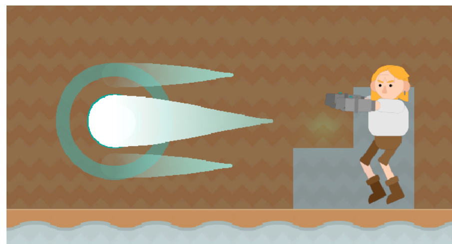

Figura 8.23: Llançament d'un projectil com habilitat.

8.5.1.6 Rebre mal i estat de recuperació

El personatge pot rebre mal per enemics o per objectes en l'escena que són perillosos perquè, a l'hora d'entrar contacte amb ells, el personatge perd vides. El personatge té la capacitat de recuperar-se d'haver rebut mal, durant un temps, de l'ordre de pocs segons. Durant aquests segons el personatge no pot rebre mal. Això ens permet reposicionar al nostre jugador en una posició més adequada per continuar jugant en cas de no haver mort pel fet de rebre mal.

Podem saber que el personatge ha entrat en fase de recuperació perquè salta una animació de parpelleig que fa canviar el color del personatge principal entre el seu color natural i un color vermellós.

Figura 8.24: El personatge rep mal i entra a fase de recuperació.

8.5.1.7 Curació

De la mateixa manera que el personatge pot rebre mal, es pot curar. És cura interactuant amb les fruites que es troben en els arbustos de les escenes o exposicions de fruites.

Figura 8.25: Curació després d'ingerir fruita de l'arbust.

8.5.1.8 Mort

Quan el personatge es queda sense vides, ja sigui perquè un enemic ha acabat amb ell, o perquè ha caigut a la lava, o a l'aigua, etcètera, es llença una animació de mort al personatge i com a jugador es perd el control. Seguidament, apareixerem en el punt de guardat més pròxim.

Figura 8.26: Mort i pèrdua de control sobre el personatge.

8.5.2 Enemics

En aquest apartat presentarem la gran varietat d'enemics construïts i també parlarem sobre les variants. Tots els enemics han estat controlats per arbres de comportaments.

8.5.2.1 Crawl Slime

Els Crawl Slime són enemics que es mouen de punt a punt d'una plataforma. Aquests enemics estan pensats per dificultar-nos l'exploració entre plataformes. Hem creat tres variants d'aquest enemic.

- Blue, aquest és el Crawl Slime bàsic, no destaca ni pel mal, vida o velocitat.

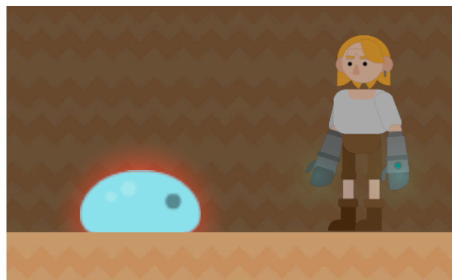

Figura 8.27: Versió blava d'un Crawl Slime.

- Pink, aquest és el Crawl Slime mare, destaca pel nombre de vides i per la grandària, però és més lent que un bàsic del mateix tipus.

- Green, aquest és el Crawl Slime més ràpid de tots, té poca vida però una velocitat endiablada.

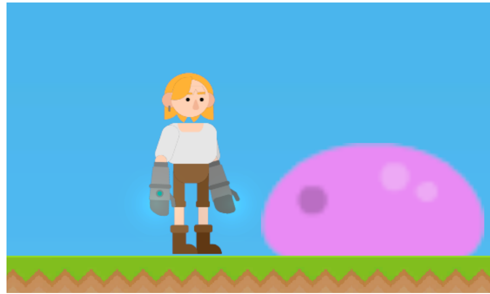

Figura 8.28: Versió rosada d'un Crawl Slime.

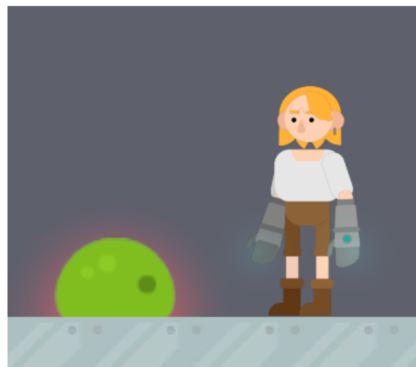

Figura 8.29: Versió verda d'un Crawl Slime.

8.5.2.2 Wheeler

Els Wheeler són uns enemics que, igual que els Crawl, es mouen de punt a punt, però a diferència d'aquests últims, aquests és llençant sobre el personatge fent-li mal. Wheeler s'inspira de Wheel[2], això perquè abans de llençar-se sobre el personatge roda sobre si mateix per donar l'advertència del salt.

- Blue, és la versió bàsica.

[2]Paraula anglesa que significa roda

Figura 8.30: Versió blava d'un Wheeler Slime.

- Pink, aquesta versió conté més vides que la bàsica i un rang d'atac menor.

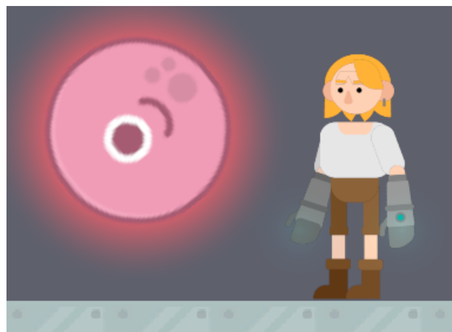

Figura 8.31: Versió rosada d'un Wheeler Slime.

- Green, aquesta versió conté poques vides, un rang d'atac més ample i més velocitat de moviment.

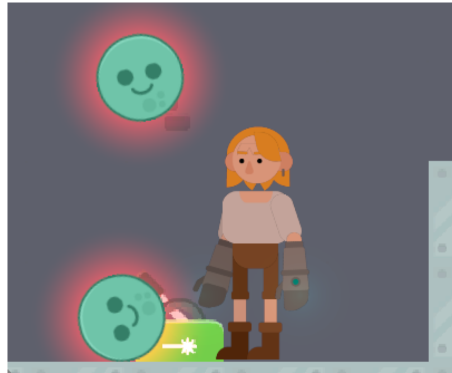

Figura 8.32: Grup de Green Wheeler Slime.

8.5.2.3 Worm Slime

Aquest tipus d'enemic s'amaga sota el terra fins que nota la presència del jugador i no el deixa passar de cap de les maneres, és com si fos un mur que s'ha de tirar a terra.

- Worm Slime bàsic, acostumen a estar en grups i no tenen cap més funcionalitat que mostrar-se i amagar-se.

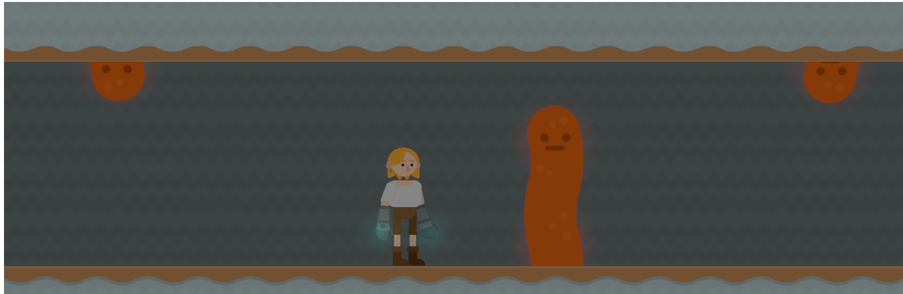

Figura 8.33: Grup de Worm Slime.

- Worm Slime explosiu, l'única diferència entre la versió explosiva i la bàsica d'aquest tipus d'enemics és que aquest cíclicament va fent explosions i si estem dintre de la zona d'explosió, el personatge perd vides.

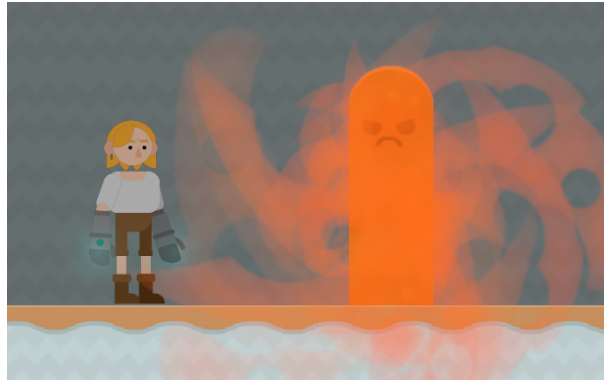

Figura 8.34: Worm Slime explotant.

8.5.2.4 Planta

La Planta és un enemic que, en notar la nostra presència, deixa anar un projectil que ens segueix. El projectil explota o bé perquè ha tocat al jugador, a l'escena, o perquè el temps de vida s'ha esgotat.

Una curiositat important de la planta és que realment són dues intel·ligències artificials combinades: la planta en si, que s'encarrega de detectar i fer una instància del projectil cada cert temps; i l'intel·ligènca del projectil, que s'encarrega de seguir al jugador per l'escena, gestionar el temps de vida i les col·lisions amb l'escena i el jugador.

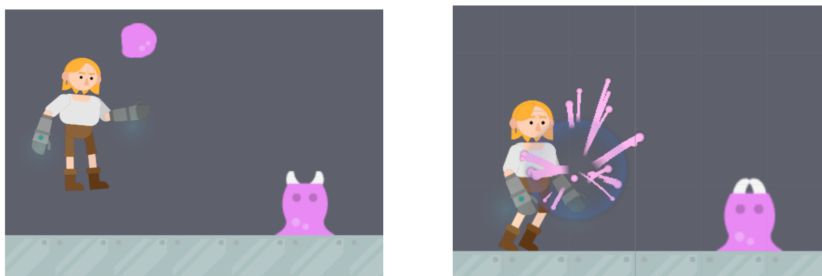

Figura 8.35: Sequencia d'atac de la planta.

8.5.2.5 Mosca

La Mosca és un enemic que vola per dos o més punts cíclicament. Aquest enemic, quan nota la presència del personatge, el segueix fins que, o bé s'ha allunyat molt del punt de vigília o bé ja no detecta al personatge perquè s'ha mogut massa ràpid i ja no l'aconsegueix a veure.

- Mosca bàsica, aquest tipus de mosca té molta vida i és difícil de matar, els moviments són lents comparada amb la mosca petita.

Figura 8.36: Mosca bàsica

- Mosca petita, és un tipus de mosca que es mou a alta velocitat i que sol estar en grups.

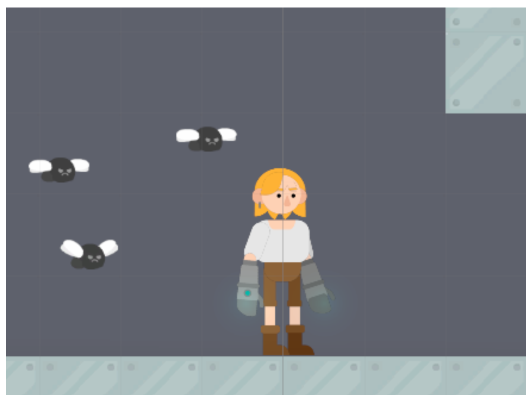

Figura 8.37: Grup de mosques petites.

8.5.2.6 Apache Pig

Apache Pig és el primer Boss que trobem en el videojoc. Les imatges que conformen aquest Boss han estat creades amb l'eina d'Adobe i totes les animacions han estat creades en Spine.

Ser un dels Boss del nostre projecte comporta tenir una intel·ligència artificial més complexa. Això ho hem aconseguit gràcies a l'extensió de lògica del codi de Behavior Designer i creant una tasca pare composta que té una llista de subarbres, on cada subarbre és un comportament del Boss i, fent ús d'una variable compartida, podem saber en quina fase està el Boss, i amb la fase podem saber el conjunt de comportaments ha d'executar.

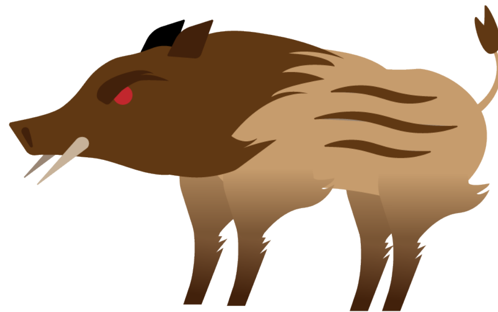

Figura 8.38: Apache Pig.

A continuació explicarem les diferents habilitats que té aquest Boss així com les fases de lluita que té. Però abans de continuar hem de dir que tots el Bosses, és a dir Apache Pig i FALSE Knight **??** en passar d'una fase a l'altre, augmenta el nombre de vides, i això provoca que en cada fase la dificultat sigui major, ja que s'incorporen noves habilitats i s'augmenta la durabilitat del combat.

La màquina d'estats que es veu en la Figura 8.39 descriu el comportament de les animacions d'aquest enemic.

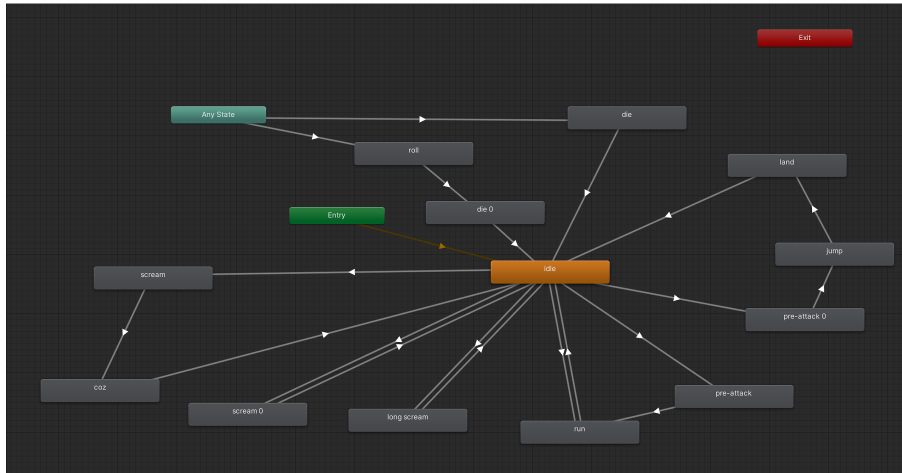

Figura 8.39: Màquina d'estats d'animacions del Boss.

Els comportaments per fase de les que el Boss disposa són definides per l'arbre de comportament de la Figura 8.40.

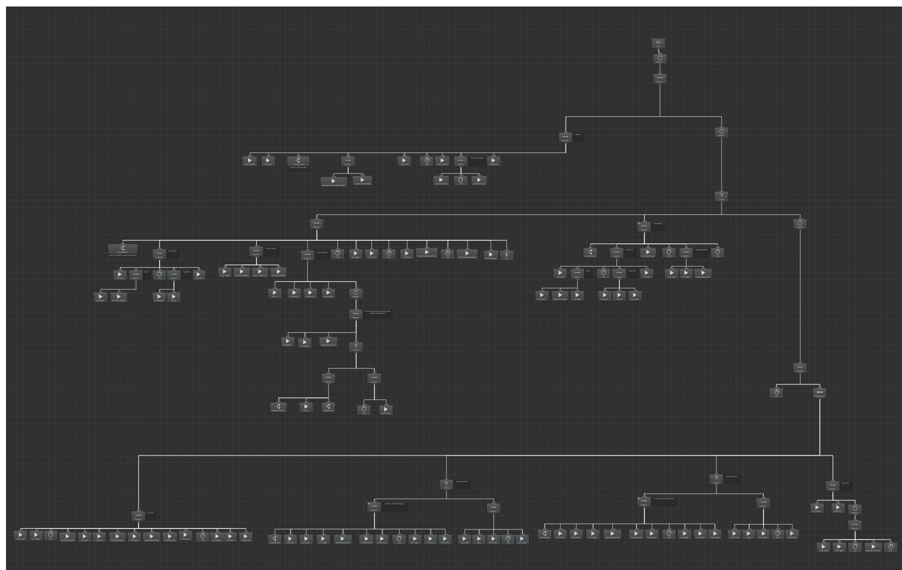

Figura 8.40: Arbre de comportament del Boss.

A continuació parlarem sobre les habilitats que aquest Boss disposa.

- Envestida, aquesta habilitat permet envestir al jugador. Queda contusionat quan xoca contra les parets que delimiten la lluita. Quan el Boss xoca amb alguna de les parets, cau un cert nombre de roques aleatories del sostre que s'han d'esquivar i es modifica la càmera per tal de donar un efecte de tremolor.

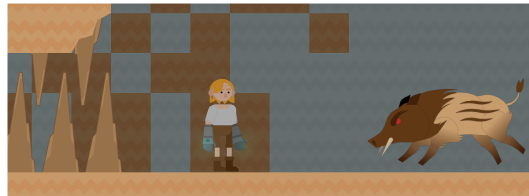

Figura 8.41: Apache Pig fent una envestida.

- Salt, és l'habilitat que permet al Boss tirar-se sobre del jugador provocant-li mal, al moment d'aterrar cauen un cert nombre roques del sostre aleatòriament. A l'aterrar, el Boss provoca un efecte de tremolor però no gaire intens. Si en algun moment Apache Pig fent el salt xoca contra les parets, queda contusionat i és un bon moment per atacar-lo.

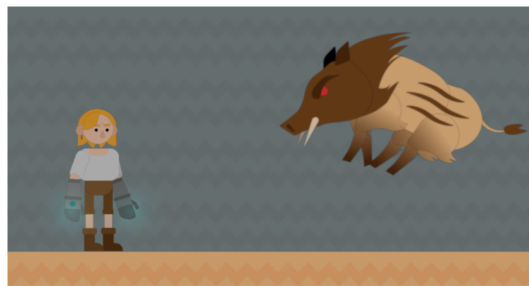

Figura 8.42: Apache Pig saltant sobre el jugador.

- Instanciar roques, Apache Pig pot fer aparèixer roques de sota del terra on està posicionat el personatge, provocant-li mal. Quan el Boss activa aquesta habilitat pot fer aparèixer roques un cert nombre de vegades, d'1 a 3, el nombre d'instanciàs que fa aparèixer és aleatori.

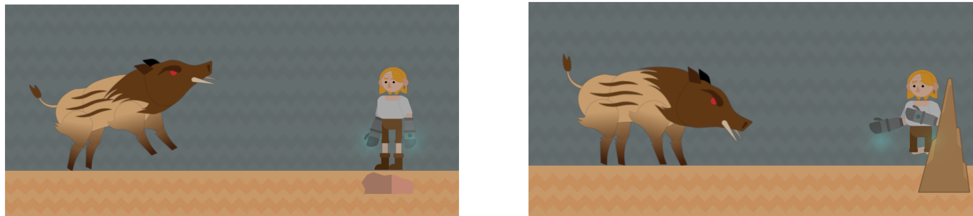

Figura 8.43: Apache Pig instanciant una roca que travessa a Ajax.

- Crit de la bèstia i coç, aquesta habilitat és l'últim intent d'acabar amb nosaltres, el Boss es posa al centre de l'escena i comença a cridar i donar coces a l'aire, girant de costat a costat, els crits provoquen que caiguin una ràfega de roques del sostre que hem d'esquivar i hem de tenir en compte que al centre hi ha el Boss donant cozes.

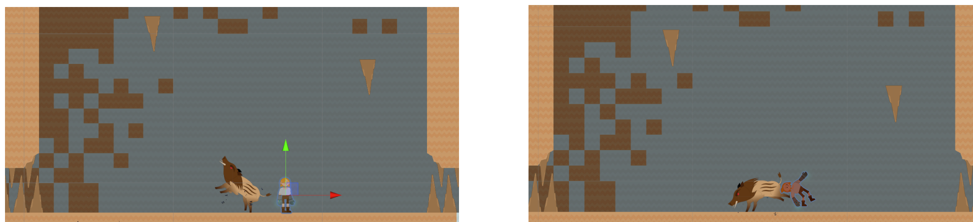

Figura 8.44: Sequencia atac principal.

A continuació explicarem les diferents fases que té aquest personatge.

- Fase inicial, en aquesta fase, Apache Pig s'adona de la presència del jugador, cau del sostre i delimita la zona de lluita fent aparèixer roques en les portes d'entrada i sortida.

Figura 8.45: Apache Pig saltant sobre el jugador.

- Fase lluita I, Apache Pig envesteix al personatge, en cada envestida aquest s'acaba xocant contra les pareds i cau a terra on és fàcil d'atacar.
- Fase lluita II, en aquesta fase Apache Pig té l'envestida i el salt, per últim s'augmenta la durabilitat.
- Fase lluita III, en aquesta fase Apache Pig augmenta encara més la durabilitat, i incorpora la l'habilitat d'instanciar roques.
- Fase lluita IV, en aquesta fase únicament té l'habilitat del crit de la bèstia i coç.
- Fase final, en aquesta el personatge cau al terra derrotat, s'obren els camins perquè les roques desapareixen i s'anuncia que s'ha derrotat al Boss per mitjans del sistema de comunicacions.

Figura 8.46: Mort del Boss.

8.5.2.7 FALSE Knight

Ara explicarem en què consisteixen cadascuna de les habilitats de l'últim Boss, aquest és el més complicat dels dos, segurament no per les seves mecàniques sinó pel temps d'espera entre comportaments.

Aquest Boss no és una creació pròpia quant a que no hi ha està dissenyat per nosaltres, les imatges que conformen les animacions del Boss l'hem tret de la següent pàgina [PCcomputer 022] que subministra contingut d'ús gratuït. Això si, nosaltres ens vam encarregar d'animar-lo ajuntant les imatges i fer la coreografia de fases fent ús dels arbres de comportament.

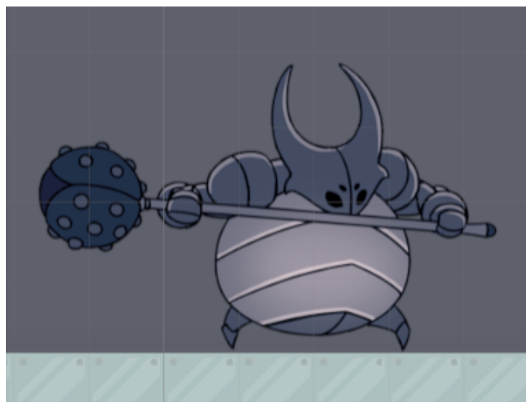

Figura 8.47: FALSE Knight.

La màquina d'estats d'animació d'aquest Boss és prou complex, ho podem apreciar en la Figura 8.48. També podem apreciar l'arbre de comportament d'aquest Boss en la figura 8.49.

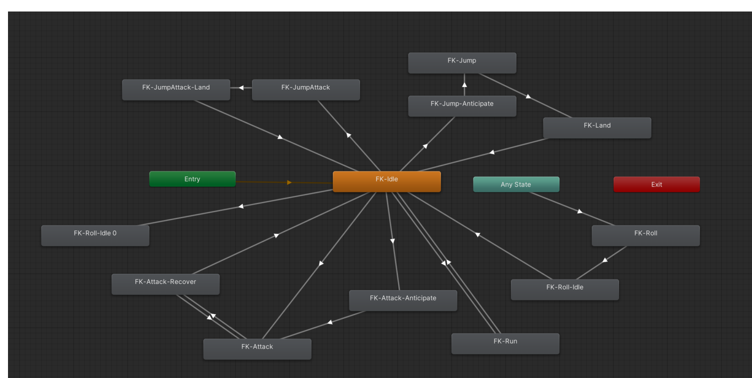

Figura 8.48: Màquina d'estats d'animacions.

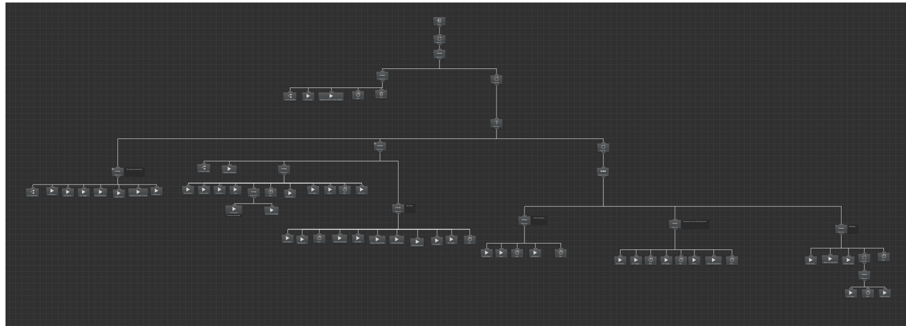

Figura 8.49: Arbre de comportament.

A continuació explicarem en què consisteixen les fases d'aquest Boss.

- Fase inicial, en aquesta fase el Boss s'adona de la nostra presència, es canvia la música per una música d'acció i tanca les portes que delimiten l'àrea de lluita.

Figura 8.50: Inici de la lluita amb el FALSE Knight.

- Fase lluita I, FALSE Knight es mou en l'escena fent salts sobre el personatge per poder aixafar-lo.
- Fase lluita II, en aquesta fase augment la durabilitat i afegeix l'habilitat d'onda de xoc.
- Fase lluita III, en aquesta fase augmenta la durabilitat i afegeix l'habilitat de ràbia.
- Fase lluita IV, aquesta fase és igual a la fase III, només augmenta la durabilitat.
- Fase final, en aquesta el personatge cau al terra derrotat, s'obren els camins perquè les roques desapareixen i s'anuncia que s'ha derrotat al Boss per mitjans del sistema de notificacions.

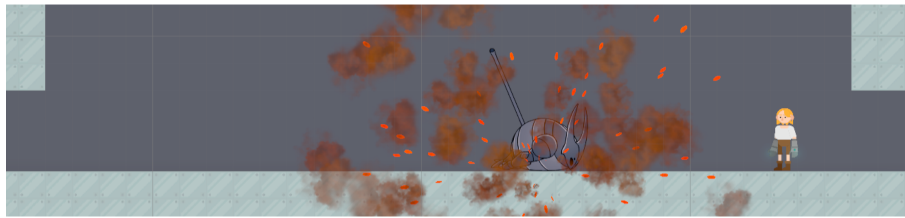

Figura 8.51: Mort del Boss.

8.5.3 NPC

Els NPC serveixen per situar-nos en la història i donar-nos consells i advertències. Tots els NPC del joc han estat creats i animats amb Illustrator i Spine.

- **Capita Java**
  Aquest NPC està pensat per donar-nos context de la història i ens fa de guia per saber quines és la meta del joc, que és bàsicament que el personatge principal es retrobi amb la seva germana.

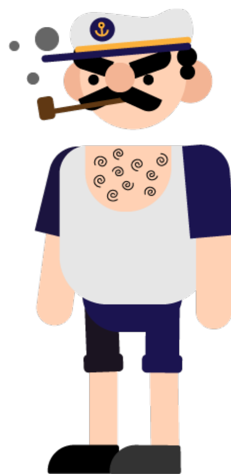

Figura 8.52: El Capità Java.

La màquina d'estats que defineix les animacions d'aquest personatge és prou simple, ho podem veure en la Figura 8.54.

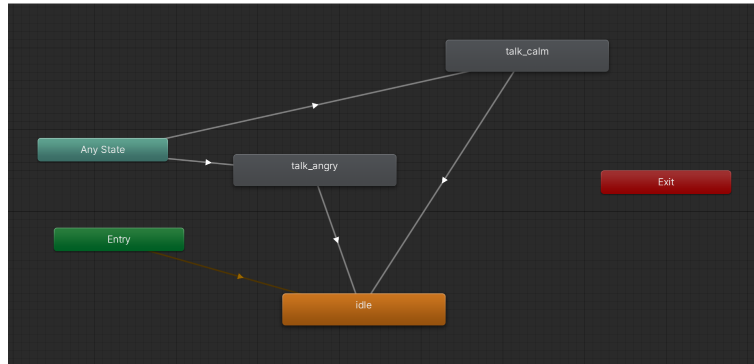

Figura 8.53: Màquina d'estat del Capità Java.

Podem trobar al Capità en Java en diferents escenes, ell s'encarrega de desplaçar-nos entre les illes que passa la història.

Figura 8.54: El Capità Java en el joc.

- **Cassandra**
  Aquest NPC és la germana del protagonista, la història del joc consisteix en retrobar-nos amb ella. La Cassandra es troba estancada en el laboratori del seu pare (personatge que no apareix en cap moment), Ajax en un intent d'ajudar-la es queda estancat en el laboratori. En el laboratori, la Cassandra es comunica amb l'Ajax a través d'uns altaveus, ens dona consells i advertències en tot moment fins que la retrobem.

Figura 8.55: Cassandra.

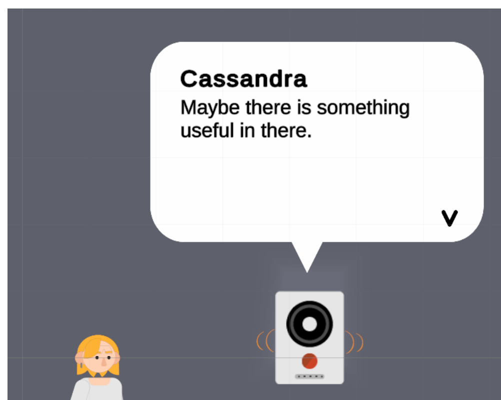

Figura 8.56: Altaveu de comunicació de la Cassandra.

La màquina d'estats que defineix les animacions d'aquest personatge és prou simple, ho podem veure en la Figura 8.57.

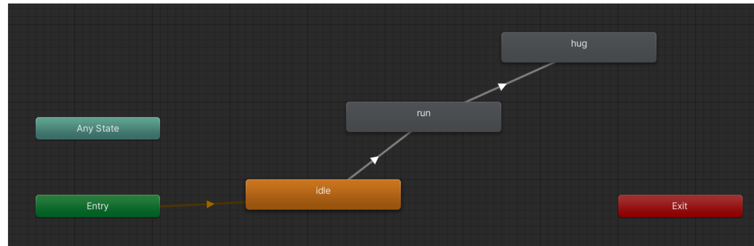

Figura 8.57: Màquina d'estat del la Cassandra.

Quan trobem a la Cassandra, ella abraça al personatge i acaba el joc. En la Figura 8.61 podem veure com es veu dintre del joc.

Figura 8.58: Retrobament entre la Cassandra i l'Ajax.

8.5.4 Objectes d'escena interactuable

En aquest apartat veurem alguns dels elements que no són ni enemics ni NPC, però que el jugador té la capacitat d'interactuar.

8.5.4.1 Ànima

Aquest tipus d'objecte s'amaga en certes àrees d'algunes escenes. En entrar en contacte amb aquests, el personatge aconsegueix habilitats noves, ja que el personatge al llarg del joc ha d'aconseguir dues noves habilitats.

Hi ha dos objectes ànimes, el primer objecte ànima que ens trobem en la història és el que dona l'habilitat de fer desplaçaments llargs, i per últim trobem a l'objecte ànima que ens permet llençar projectils.

Figura 8.59: Objectes ànima

8.5.4.2 Líquids

Els líquids en el projecte fan mal al personatge principal. El nostre personatge no sap nadar i tampoc és resistent a les altes temperatures de la lava. Per tant, hem de poder ser capaços de desplaçar-nos sense tenir contacte amb aquestes àrees.

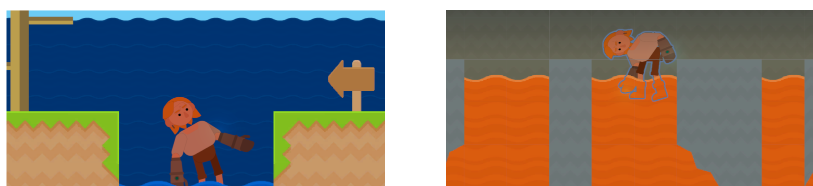

Figura 8.60: Líquids.

8.5.4.3 Palanca

La palanca és un mecanisme que ens permet diversificar el joc. És un mecanisme pensat per activar o desactivar altres mecanismes més complexos, com ara activar el moviment d'una plataforma, obrir i tancar portes, etcètera.

Figura 8.61: Palanca.

8.5.4.4 Porta

És un mecanisme utilitzat en el videojoc que delimitan àrees de la mateixa escena, o bé no ens deixen passar entre escenes. En el projecte hi ha portes simples, que es tanquen i s'obren per certs esdeveniments, per exemple, les limitacions de la zona de lluita amb els Bosses són portes simples. Finalment, tenim les portes que s'obren o es tanquen amb la interacció del personatge amb les palanques.

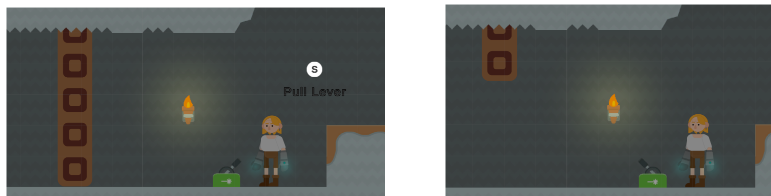

Figura 8.62: Porta que s'activa amb palanca.

8.5.4.5 Plataformes

Les plataformes són elements bàsics dels Metrodvania, per tant, en vam construir una gran varietat. El nostre joc conté plataformes estàtiques i plataformes dinàmiques. Les plataformes dinàmiques poden ser d'un únic recorregut o desplaçar-se i tornar al punt d'origen, les plataformes dinàmiques també poden ser activades per palanques.

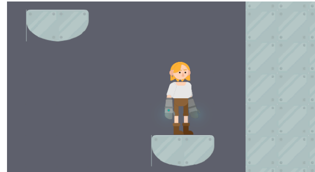

Figura 8.63: Plataforma estàtica.

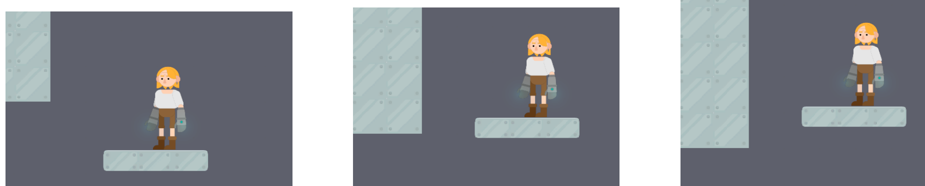

Figura 8.64: Plataforma dinàmica amb moviment vertical.

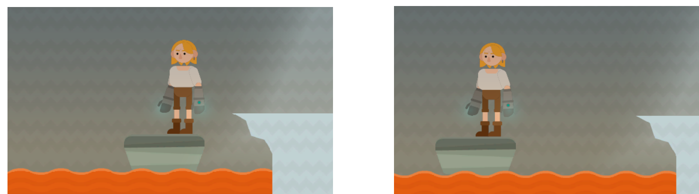

Figura 8.65: Plataforma dinàmica amb moviment lateral.

Figura 8.66: Plataforma dinàmica amb palanca.

8.5.4.6 Fruita

L'única manera que té el personatge de curar-se en el joc és menjant fruites que es troben en petits arbustos o bé en llocs d'exposició de fruites. Cada fruita consumida cura un punt de vida al personatge.

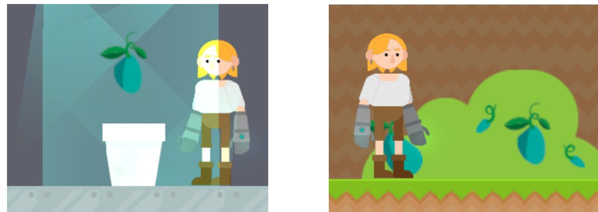

Figura 8.67: Fruites de curació.

8.5.4.7 Tresor

Un tresor és un objecte que podem trobar molt poc al llarg del lloc, ja que quan trobem un i l'obrim, aquests ens dona una vida màxima més, i ens cura totes les vides que ens faltaven.

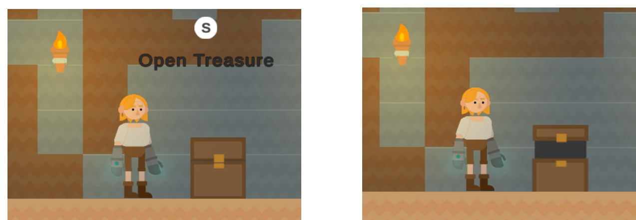

Figura 8.68: Tresor.

8.5.4.8 Estàtua de guardat

L'estàtua de guardat és un element que apareix en certes escenes, fa el que diu el seu nom, guarda l'estat de la partida. Realment no guarda l'estat de tot el joc, sinó de certs paràmetres que vam trobar interessant, com ara la vida del jugador, el nombre màxim de vides, les habilitats adquirides, entre d'altres. Quan sortim de la sessió de joc, i tornem a entrar podem reprendre la partida anterior perquè aquests paràmetres han estat guardats permanentment en un fitxer gestionat per Unity en un dels directoris de la màquina. L'implementació del funcionament esta explicada en el càpitol 9.

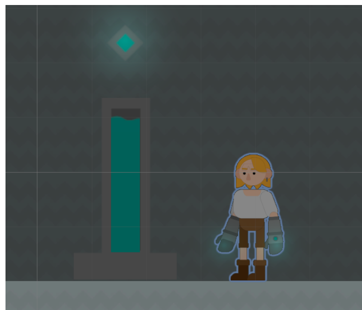

Figura 8.69: Estàtua de guardat.

8.5.4.9 Trampolí

El trampolí, tal com indica el nom, és un element que quan entre en contacte amb el jugador l'impulsa cap a l'aire. És una forma diferent de les plataformes per arribar a noves zones.

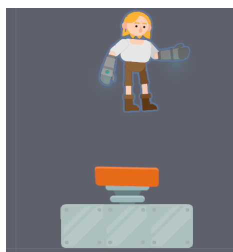

Figura 8.70: Trampolí.

8.5.4.10 Punt de control

Al llarg de totes les escenes hi ha diferents punts de control, que no són visibles. Quan el jugador els artrabessa, es guarda la posició del punt de guardat per tal

de reposicionar al jugador en un lloc segur, en cas que caigui dintre de zones no recuperables, com ara els líquids. En la Figura 8.71 podem veure un punt de guardat com una caixa verda, aquesta caixa verda és un Collider que no interactua físicament.

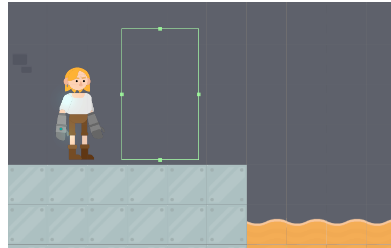

Figura 8.71: Punt de control.

8.5.5 Interfícies d'usuari

El projecte compte una gran varietat d'interfícies d'usuari que van des de menús, interfícies dintre del joc i interfície de finalització de joc.

8.5.5.1 Menú d'inici

Menú inicial en el qual ens trobem en obrir el joc i que ens permet crear una nova sessió de joc, també podem accedir a les configuracions i finalment tenim l'opció de sortir del joc.

Figura 8.72: Menú d'inici.

8.5.5.2 Menú d'inici de sessió

Gràcies a aquest menú podem crear una nova sessió de joc o bé recuperar la sessió anterior de joc.

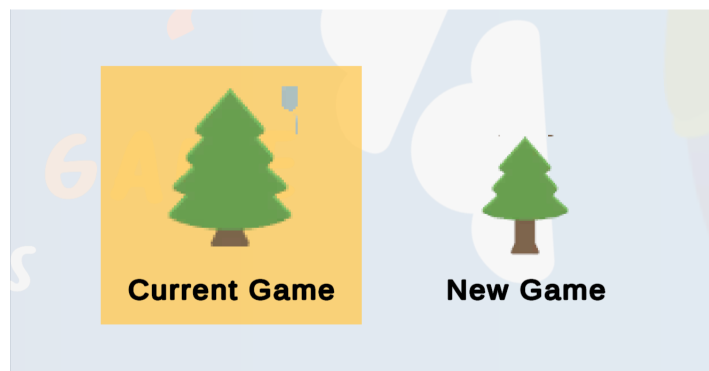

Figura 8.73: Menú d'inici de sessió.

8.5.5.3 Menú de configuracions

En aquest menú podem configurar la intensitat de la música ambient i dels efectes especials sonors així com tenim accés directe al mapatge de les tecles amb les respectives accions.

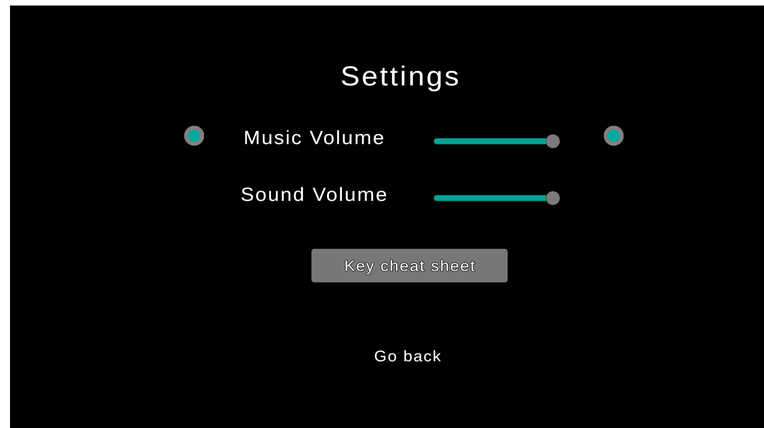

Figura 8.74: Menú de configuració.

8.5.5.4 Menú de mapatge de tecles

Des de aquest menú podem veure les tecles útils en el joc i les seves funcionalitats.

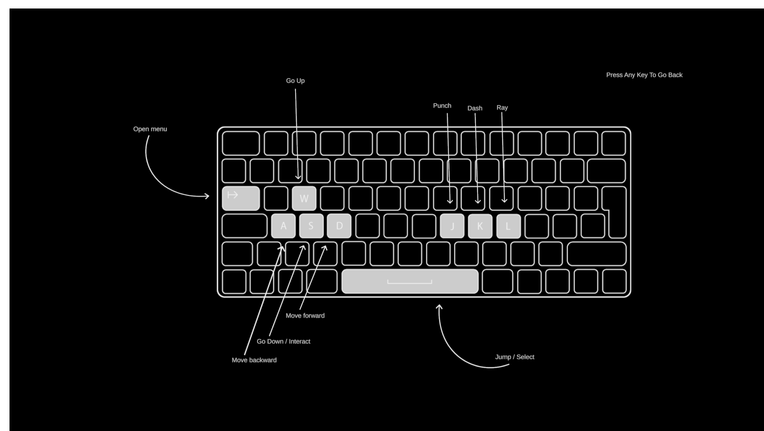

Figura 8.75: Menú de mapatge de tecles.

8.5.5.5 Interfície de càrrega

Quan el jugador passa d'escena a escena s'han de destruir els objectes de l'escena anterior i s'han de construir els de la següent escena, com aquest procés triga

un temps, i el temps depèn de la potència del maquinari sobre el qual s'estigui executant el joc, vam decidir crear una interficie lleugera que s'executi mentre es carrega la nova escena perquè l'usuari sàpiga que s'està produint un procés de càrrega.

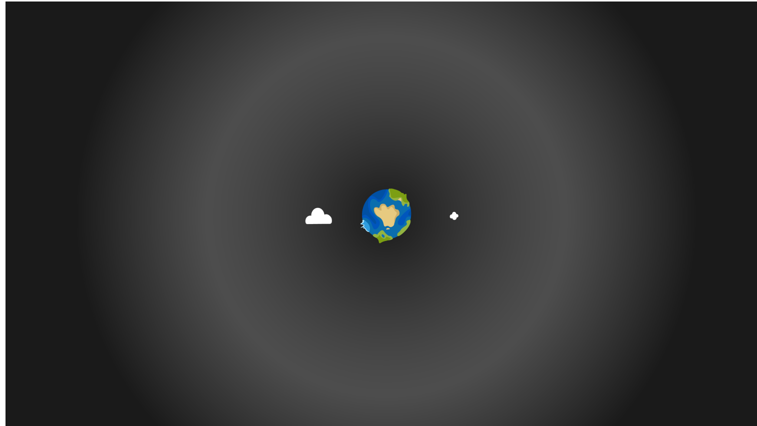

Figura 8.76: Interfície de càrrega.

8.5.5.6 Menu de joc

Aquest menú està pensat per accedir ràpidament a les configuracions i per poder tancar l'actual sessió de joc.

Figura 8.77: Menú de joc.

8.5.5.7 Interfície d'habilitat

Amb aquesta interfície presentem les noves habilitats adquirides, també expliquem el funcionament. Mentre està present, el personatge queda congelat fins que no interactuem amb el menú prenent qualsevol tecla.

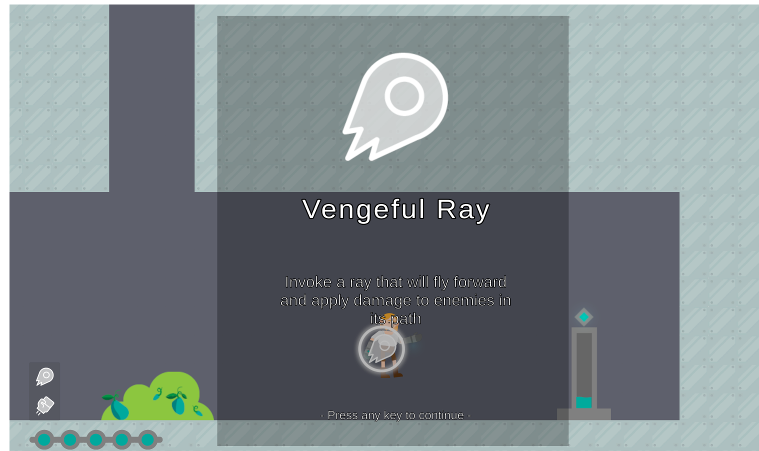

Figura 8.78: Notificació de nova habilitat.

8.5.5.8 Interfície dintre del joc

L'interfície dintre del joc és aquella que sempre està ancorat a allò que veu la càmera.

Figura 8.79: Interfície de joc.

Aquesta interfície té els següents elements:

1. Estat de les habilitats, en aquesta àrea es poden veure les habilitats adquirides i també es marca el temps de refredament cada cop que s'utilitzen.

2. Estat de la salut del personatge, aquesta àrea ens informi el nombre màxim de vides que té el personatge i el nombre de vides actuals.

3. Notificació, aquesta àrea no està sempre visible. Les notificacions apareixen quan un esdeveniment important ha passat, en cas de la Figura 8.79 podem veure que hem derrotat al FALSE Knight.

8.5.5.9 Interficie d'ajuda

Aquesta interfície està pensada perquè sortí una petita animació informant el jugador que pot interactuar.

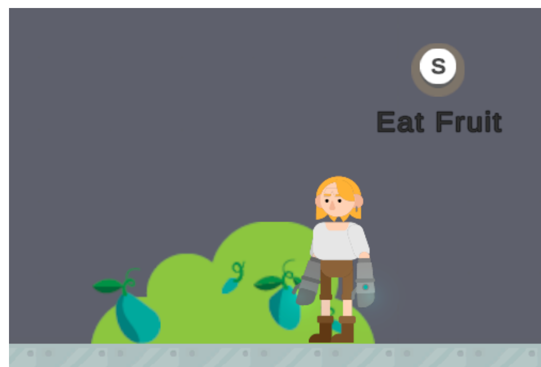

Figura 8.80: S'informa que prement la teclas *S* es pot consumir una fruita.

8.5.5.10 Interfície de diàlegs

Interactuant amb aquesta interfície es com situem al jugador dintre del context del videojoc. També ens permet informar-lo de certa situació o de guiar-lo durant la partida.

Figura 8.81: Diàleg de Cassandra.

8.5.5.11 Interficie de final de joc

Un cop es completa la història i l'Ajax troba a la seva germana, es difumina el fons i apareix l'última escena. L'última escena conté la informació dels creadors del projecte i una marieta animada que es mou d'esquerra a dreta. Amb aquesta escena fem entendre al jugador que el joc ha terminat.

Figura 8.82: Interfície de joc acabat.

8.6 Estil i art conceptual

En els apartats anteriors, per poder il·lustrar les funcionalitats que el projecte implementa, s'ha mostrat part de l'art final. Però, per poder arribar a aquest punt, vam treballar en diversos conceptes artístics i amb diferents estils fins que ens vam quedar amb el que s'ha vist fins ara.

8.6.1 Estil

Quan vam començar el projecte, vam començar experimentant amb l'estil Píxel art [WIKIPEDIA 022], però ràpidament ens vam adonar que crear contingut propi amb aquesta tècnica seria complicat perquè requereix molt de temps i fer les animacions encara es faria més complicat perquè s'han de crear a partir de seqüències d'imatges. Finalment, ens vam quedar amb l'estil vectorial.

A continuació, veurem els diferents estils alternatius al que finalment vam escollir. Veure Figures 8.83 a 8.89.

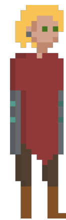

Figura 8.83: Estil 1.

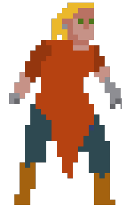

Figura 8.84: Estil 2.

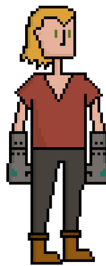

Figura 8.85: Estil 3.

Figura 8.86: Estil 4.

Figura 8.87: Estil 5.

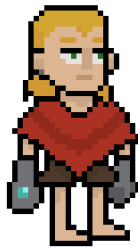

Figura 8.88: Estil 6.

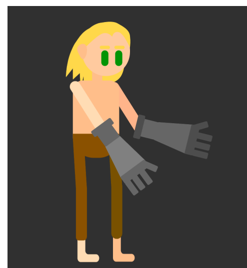

Figura 8.89: Estil 7.

8.6.2 Art conceptual

Abans de començar a fer res, i ja sàpiguen que el projecte seria un Metroidvania, ens van plantejar si el joc tindria història o no. El cas és que el projecte té història, i com tota història, necessita personatges.

El projecte incorpora una sèrie d'elements dissenyats i més tard digitalitzats amb les eines Illustrator i Spine amb la finalitat de representar a certes entitats de la història del joc.

A continuació veurem com conceptualment nosaltres imaginàvem els personatges i elements que es veuen en les diferents escenes del videojoc. Cal dir que no tots els personatges que surten a continuació apareixen en el joc. Els personatges que no apareixen formen part d'una història encara més gran a part de la presentada en el joc.

8.6.2.1 Disseny Ajax

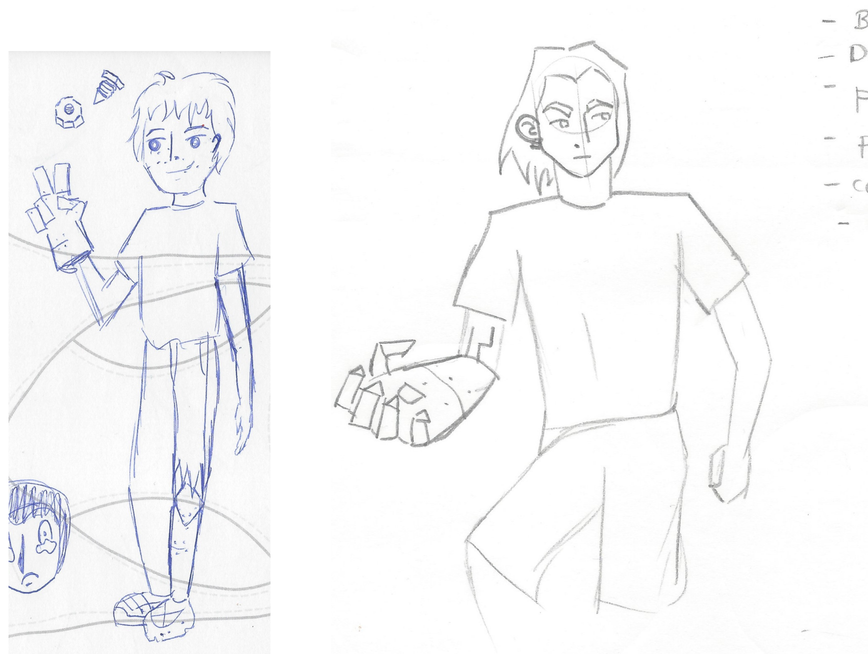

Figura 8.90: Dissenys inicials de l’Ajax.

Figura 8.91: Disseny final de l’Ajax.

8.6.2.2 Disseny Cassandra

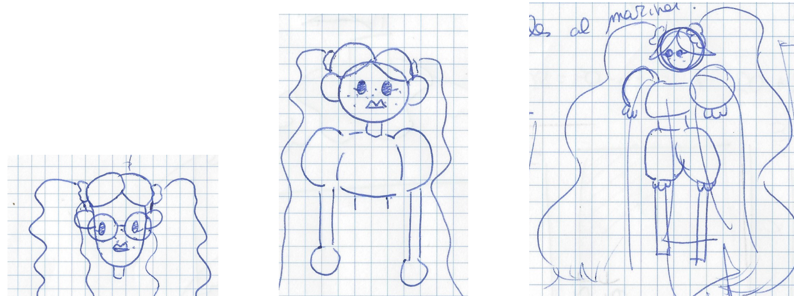

Figura 8.92: Dissenys inicials de la Cassandra.

Figura 8.93: Disseny final de la Cassandra.

Figura 8.94: Concepte artístic de Cassandra i Mr. C.

8.6.2.3 Disseny Capità Java

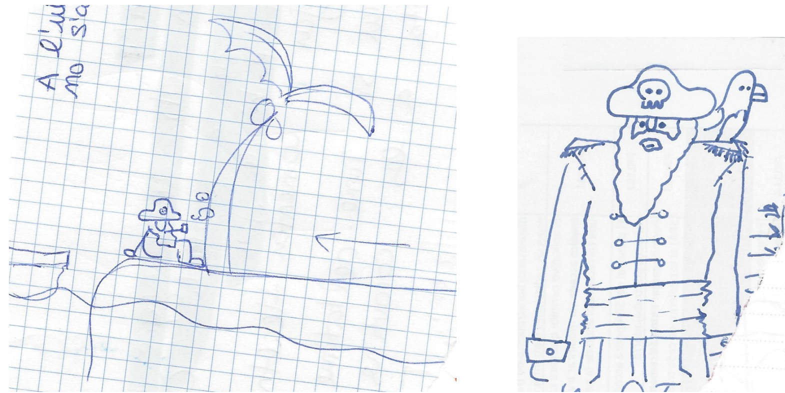

Figura 8.95: Dissenys inicials del Capità Java.

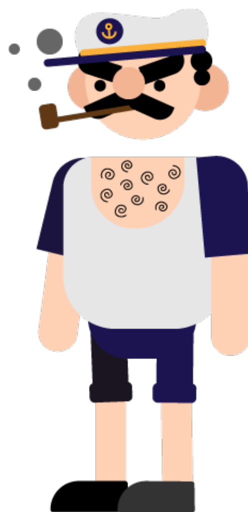

Figura 8.96: Disseny Final del Capità Java.

8.6.2.4 Disseny Jason

Figura 8.97: Disseny inicial d'en Jason.

Figura 8.98: Disseny final d'en Jason.

8.6.2.5 Disseny d'Enemics

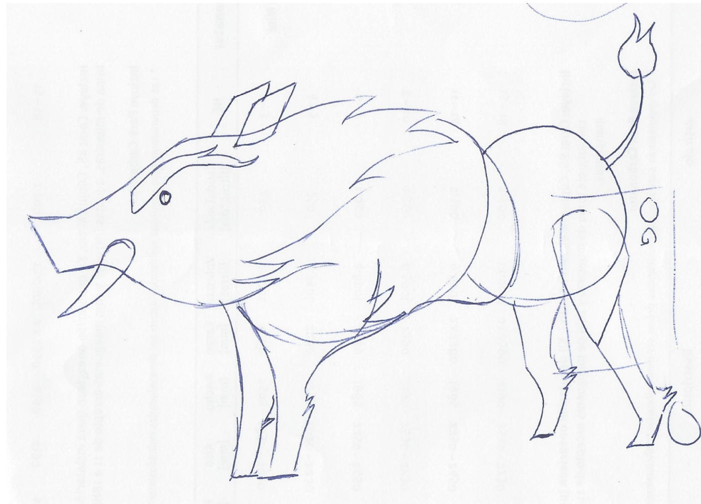

Figura 8.99: Disseny inicial d'Apache Pig.

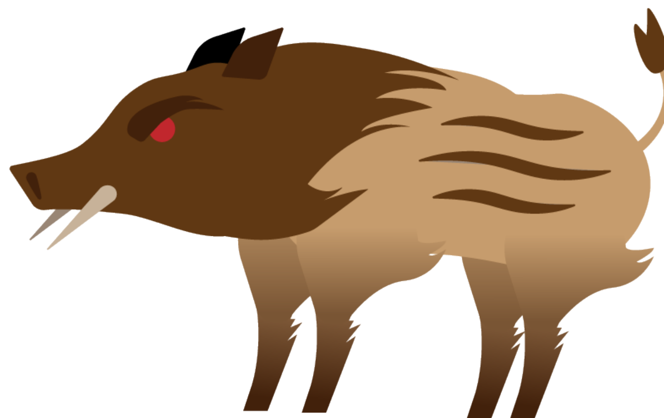

Figura 8.100: Disseny final d'Apache Pig.

8.6.2.6 Disseny d'Interficies d'usuari

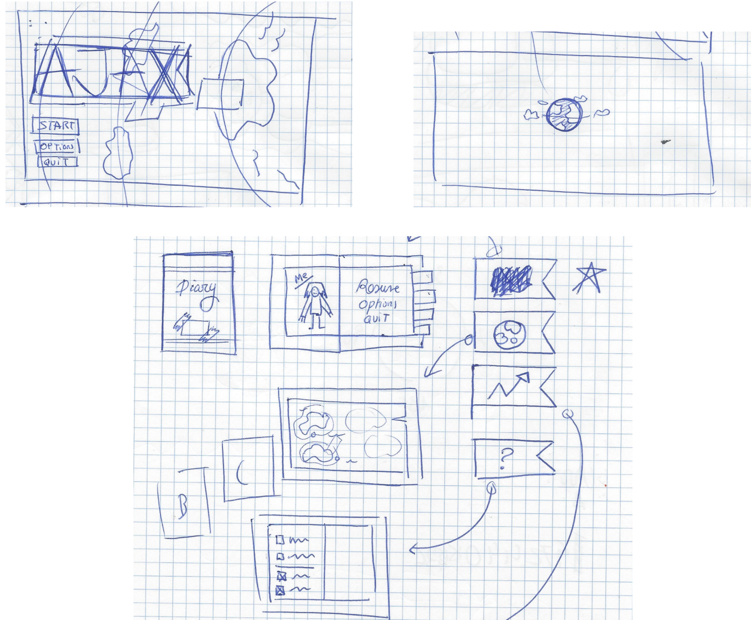

Figura 8.101: Dissenys inicials dels diferents menús.

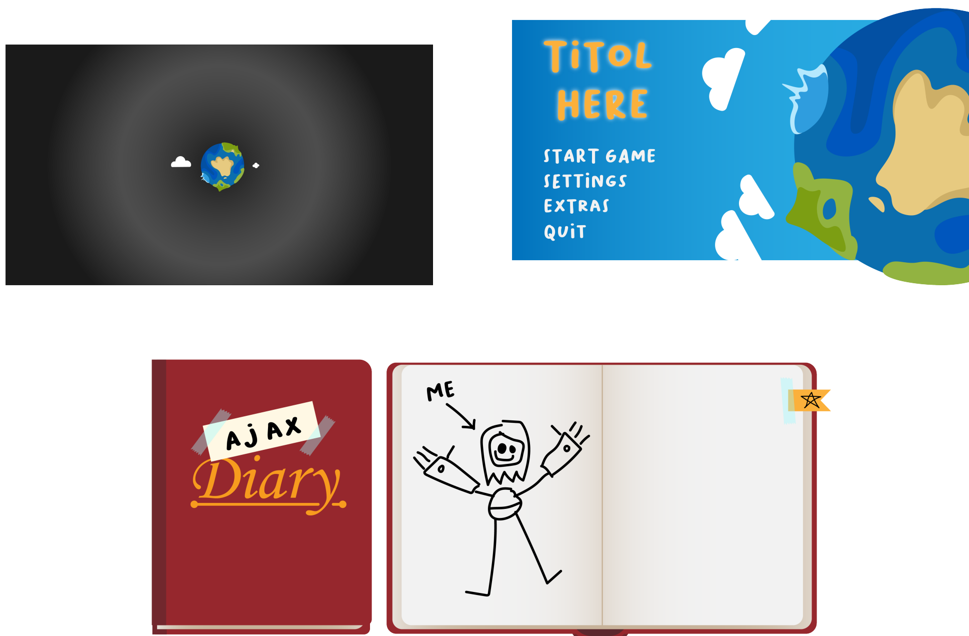

Figura 8.102: Dissenys finals dels menus.

8.7 Diseny del mapa

En aquest apartat veurem el disseny de les illes jugables per les quals el personatge passa al llarg de la història del joc, i veurem clarament les diferents escenes de les quals estan formades. Per poder distingir els diferents elements que apareixen en les imatges següents, facilitem la llegenda de la Figura 8.103.

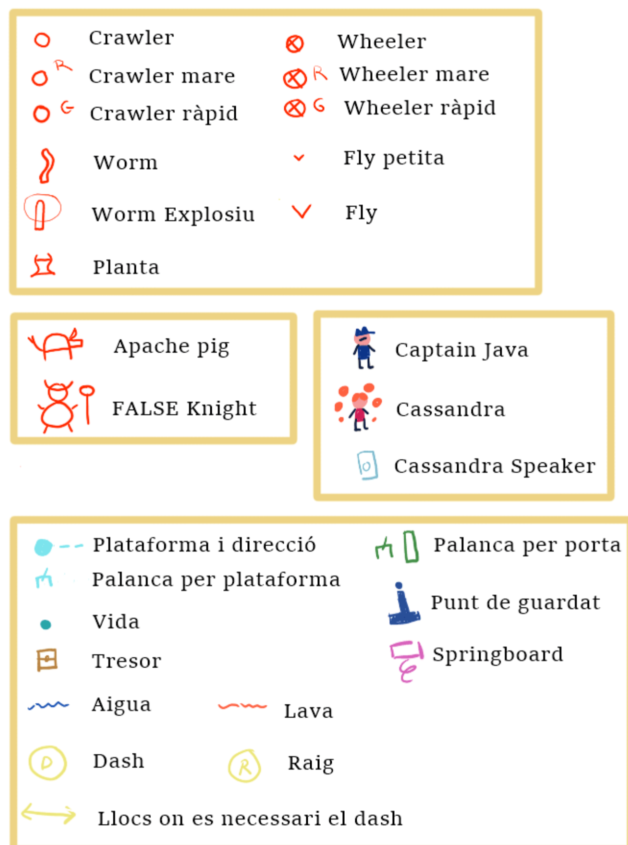

Figura 8.103: Llegenda d'escena.

8.7.1 Illa Omed

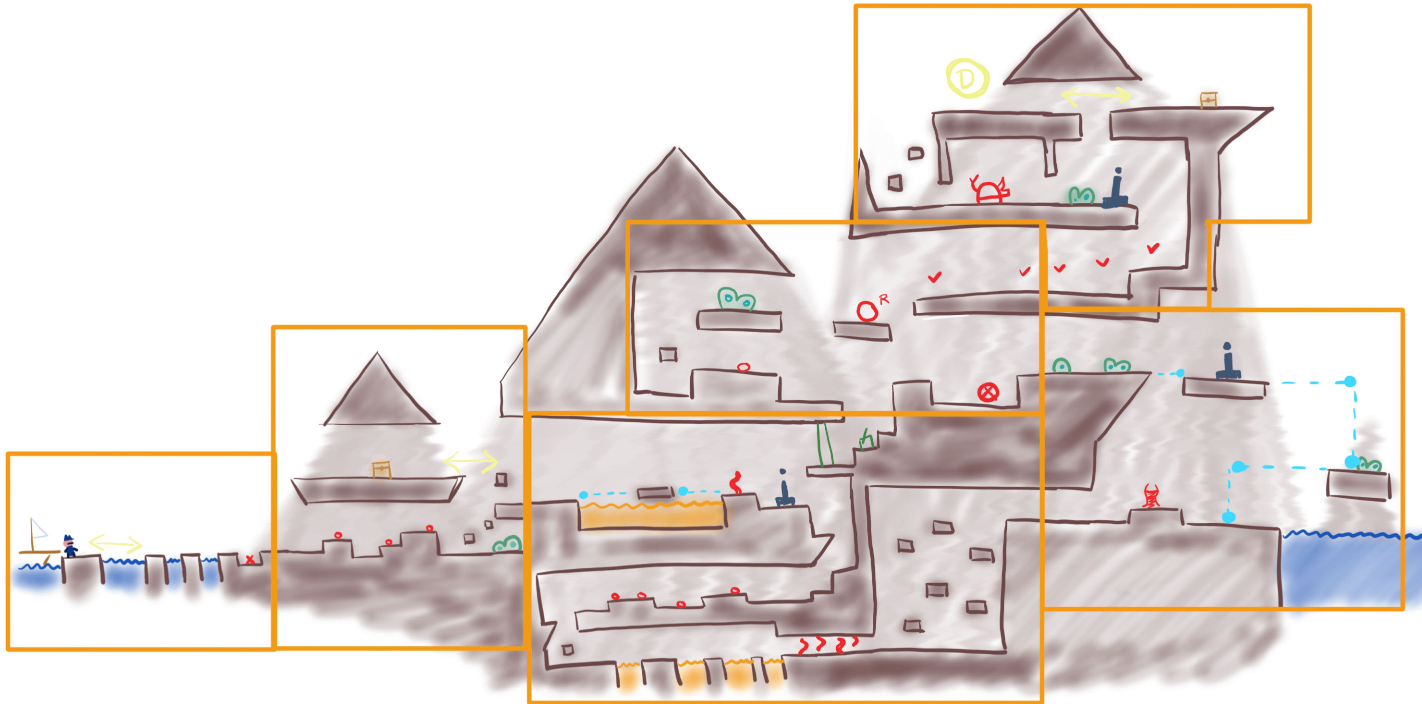

Figura 8.104: Visualització general de la illa Omed.

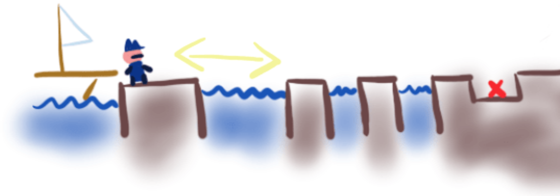

Figura 8.105: Illa Omed, escena 1.

Figura 8.106: Illa Omed, escena 2.

Figura 8.107: Illa Omed, escena 3.

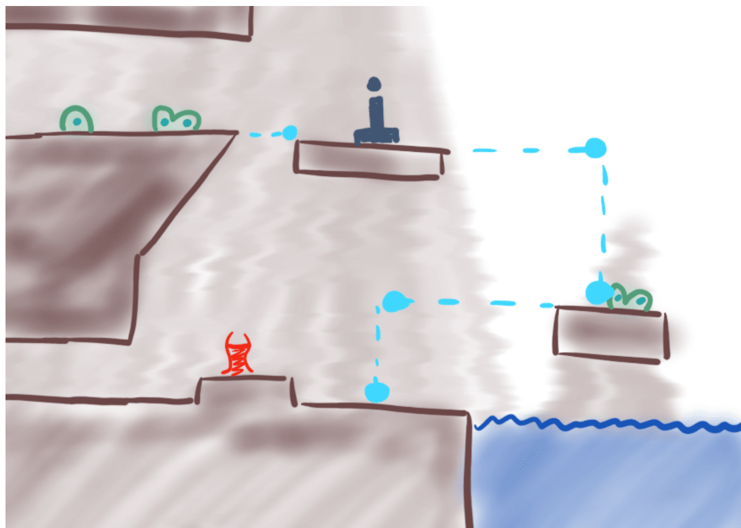

Figura 8.108: Illa Omed, escena 4.

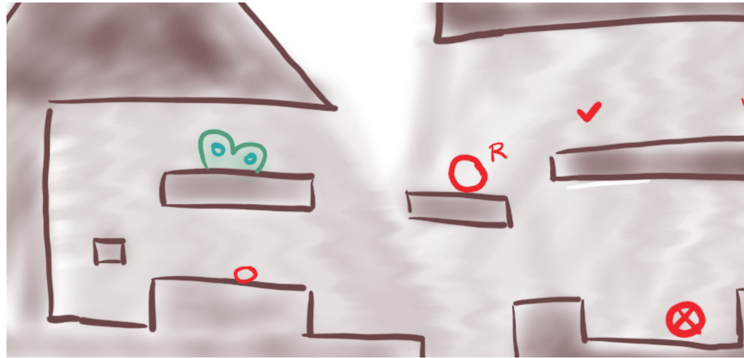

Figura 8.109: Illa Omed, escena 5.

Figura 8.110: Illa Omed, escena 6.

8.7.2 Illa dels Homes

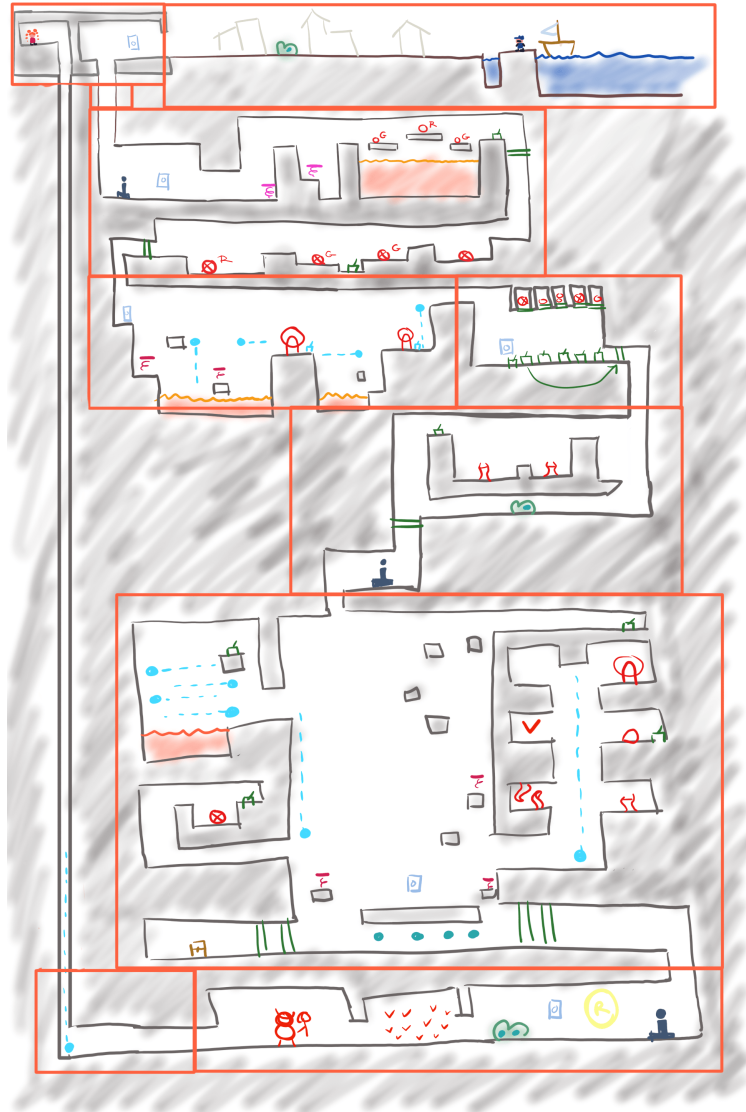

Figura 8.111: Visió general de la illa dels Homes.

Figura 8.112: Illa dels Homes, escena 1.

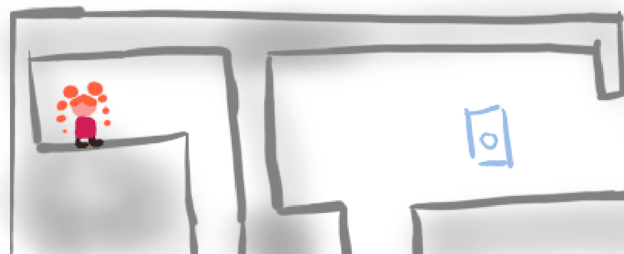

Figura 8.113: Illa dels Homes, escena 2.

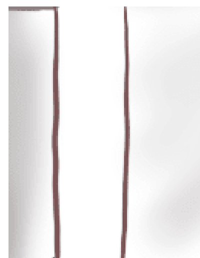

Figura 8.114: Illa dels Homes, escena 3.

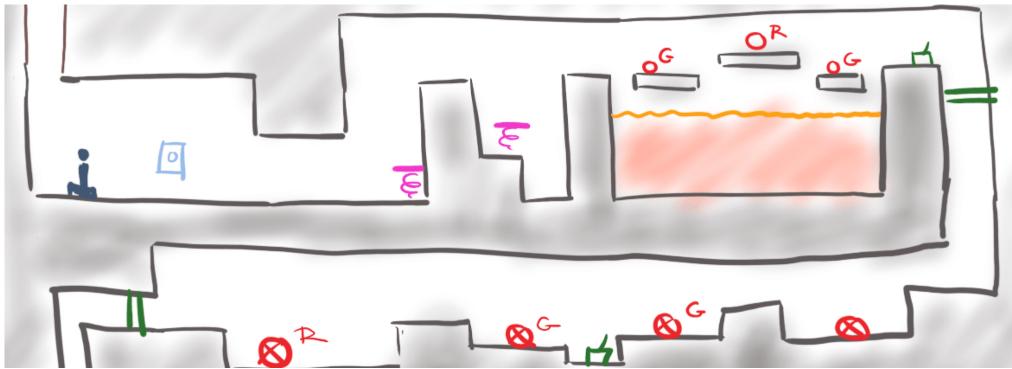

Figura 8.115: Illa dels Homes, escena 4.

Figura 8.116: Illa dels Homes, escena 5.

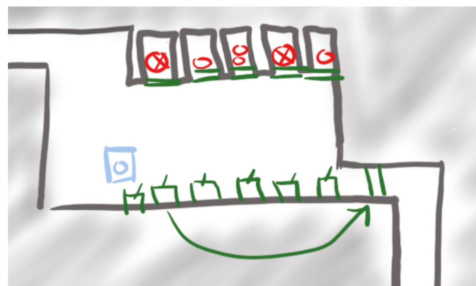

Figura 8.117: Illa dels Homes, escena 6.

Figura 8.118: Illa dels Homes, escena 7.

Figura 8.119: Illa dels Homes, escena 8.

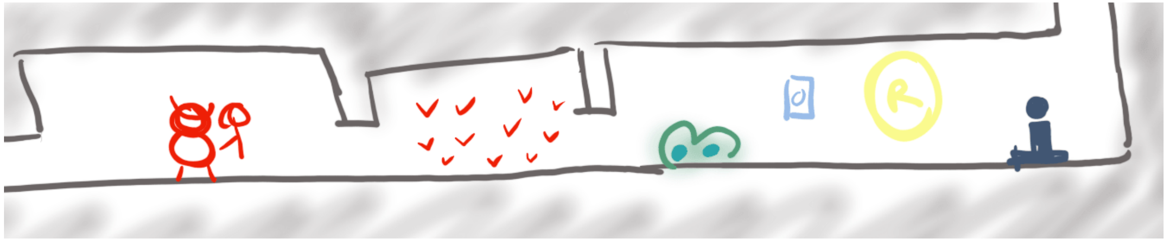

Figura 8.120: Illa dels Homes, escena 9.

CAPÍTOL 9

Implementació i proves

9.1 Personatge principal

A continuació detallarem l'implementació d'algunes de les funcionalitats del personatge principal.

9.1.1 Córrer

L'acció de córrer del personatge principal bé definit per la següent implementació:

```
private void DoRun()
{
    if (!CanRun)
        return;

        float horizontal =  Input
                            .GetAxis(CharacterActions.Run)
   ;
        float velocityX =   horizontal
                            *
                            MovementSpeed
                            *
                            Acceleration;

        if (!IsGrounded)
            // apply air drag if it's in the air
            velocityX *= AirDrag;

        var newVelocity = new Vector2
                            (velocityX,
                            Body.velocity.y);
        Body.velocity = Vector2
                        .SmoothDamp(Body.velocity,
                                newVelocity,
                                ref currentVelocity,
                                0.000001f);
        Animator.SetBool(CharacterAnimations.Running,
                Mathf.Abs(Body.velocity.x) > 0.05f);
}
```

La variable declarada en la línia 6 representa la intensitat que el jugador està prement les tecles de direcció en l'eix horitzontal. Aquest valor va entre 0 i 1. En la línia 8 es fa el càlcul de la velocitat en l'eix horitzontal que s'ha d'aplicar sobre el personatge principal. Aquest càlcul es fa amb la intensitat de les tecles de direcció, un valor constant que representa la velocitat per defecte del personatge i finalment una variable d'acceleració que s'utilitza en casos excepcionals per canviar la velocitat per defecte del personatge. En la línia 20, podem veure que s'actualitza la velocitat aplicant una funció de suavitat, per saber perquè s'aplica un efecte de suavitat en el moviment, mirar l'Apartat 8.5.1.2. Finalment, en la línia 26 s'anima al personatge amb l'animació de córrer si la velocitat supera uns mínims.

9.1.2 Desplaçament ràpid

El desplaçament ràpid, també conegut com a dash, és una de les habilitats que s'aconsegueixen al llarg del joc. El mètode que defineix el començament de l'execució d'aquesta habilitat en el controlador de moviment és la següent:

```
1  private void StartDashing()
2  {
3      if (isJumping)
4          dashMidJump = true;
5
6          isDashing = true;
7          DashTrail.widthMultiplier = 3;
8          Animator.SetTrigger(CharacterAnimations.Dash);
9          Body.velocity = Vector2.zero;
10         Body.gravityScale = 0;
11         Body.AddForce(  Vector2.right
12                         *
13                         FacingValue
14                         *
15                         DashSpeed, ForceMode2D.Impulse);
16         OnDashStart?.Invoke();
17 }
```

Si ens fixem en la línia 9 i 10, podem veure que, per codi, es treu qualsevol moviment cinemàtic que porta el personatge principal. La línia 10 treu totes les interaccions amb les físiques del joc. D'aquesta manera juntament amb la sentència de la línia 11, fem un moviment lateral evitant el moviment parabòlic descendent propi de la simulació de la gravetat del motor de físiques. Finalment, a la línia 16 trobem quelcom molt interessant, es crida l'execució de certa lògica dintre de la variable Action. A simple vista no sabem que fa aquesta execució, però això no té per què ser dolent, el que estem fent és delegar l'execució de certa lògica en un altre punt d'execució. Per exemple, quan nosaltres fem el dash, no

únicament ens desplacem a molta més velocitat sinó que tots els enemics pels quals pesem rebem ma. El control de moviment no té per què saber el com ni qui incorpora aquesta lògica de fer mal. A continuació veurem la declaració d'aquesta variable i l'assignació. Veure Apartat 7.4 en cas de dubte.

```
public Action OnDashStart { get; set; }
```

```
movementController.OnDashStart += OnDashStart;

private void OnDashStart()
{
    controllable = false;
    protectable.SetProtection(ProtectionType.INFINITE);
    abilityController.OnDashStart();
    PlayRandomSound(
        soundEffects.dash.clips,
        soundEffects.dash.volume
    );
}
```

9.1.2.1 Aplicar mal mentre es fa desplaçament ràpid

Com hem explicat prèviament, l'habilitat de desplaçament ràpid aplica mal sobre els enemics de l'escena que es troben pel camí del pas del personatge. Però la capacitat de fer mal als enemics no està implementada en la lògica de moviment i per temes de desacoblament de lògica vam fer ús de les Actions de C#, com hem detallat en l'Apartat 9.1.2. La implementació de fer mal als enemics mentre es fa el dash està en l'AbilityController. Mirar el següent codi.

```
public void OnDashStart()
{
    ActiveDashDamage();
}
```

```
private void ActiveDashDamage()
{
    if (dashParticle)
        dashParticle.Play();
    dashTrigger.Interact = true;
}
```

L'assignació d'aquesta lògica del script d'habilitats al script de moviment es fa dintre orquestrador de lògica, és a dir el PlayerController.

9.1.3 Salt

El salt del personatge principal és dinàmic, en el sentit que el lluny que arribi amb el salt depèn del temps de pressió de la tecla de salt. El codi de l'acció de

saltar és el següent:

```
public void DoJump()
{
    Action endJump = () =>
    {
        isJumping = false;
        justJumped = true;
    };

    if (Input.GetButtonDown(CharacterActions.Jump)
        &&
        CanJump) // button down, first key of jump
    {
        isJumping = true;
        holdingAfterJumpTimer = HoldingAfterJump;
        Body.velocity = new Vector2(
            Body.velocity.x,
            JumpPower);
        JumpParticles.Play();
        Animator.SetTrigger(CharacterAnimations.StartJump)
    ;
        Animator.SetBool(CharacterAnimations.Jumping,
                        isJumping);
        OnJumpStart?.Invoke();
    }
    if (Input.GetButton(CharacterActions.Jump)
        &&
        CanHoldJump) // while jumping
    {
        if (dashMidJump)
        {
            endJump();
            dashMidJump = false;
            return;
        }

        if (holdingAfterJumpTimer > 0)
        {
            Body.velocity = new Vector2(
                Body.velocity.x,
                JumpPower);
            holdingAfterJumpTimer -= Time.deltaTime;
        }
        else endJump();
    }
    if (Input.GetButtonUp(CharacterActions.Jump))
        endJump();
}
```

Podem apreciar que el comportament de principi de salt, és a dir, quan el jugador pren la tecla de salt, es defineix de la línia 9 a la línia 22. De la línia 24 a la línia 41 es defineix el que ha de fer el personatge en cas que el jugador continues prement la tecla de salt. La durada del salt està limitada per un temps de configuració, en la línia 35 podem veure que aquest temps es comprova i que en la línia 40 aquest es va disminuint, en cas que es continuï prement el botó i el temps de prémer el botó s'hagi esgotat, es finalitza el salt, mirar línia 42. En qualsevol cas si el jugador deixa de prémer el botó de salt, el salt finalitza, mirar línia 44. Per ajudar a entendre la lògica del codi, hem ajuntat el diagrama d'activitats de la Figura 9.1.

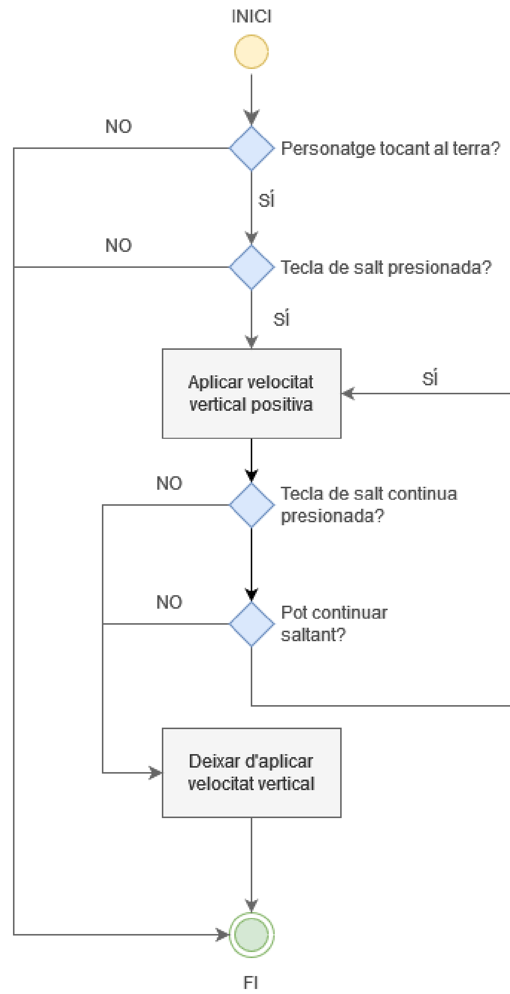

Figura 9.1: Diagrama d'activitats del salt personatge principal.

9.1.4 Gestió de la seqüència de punys

L'atac principal del personatge principal és picar amb els punys seguint una seqüència d'esquerra a dreta o de dreta esquerra. L'ordre de la seqüència és aleatori. A continuació veurem com es gestiona aquesta lògica dintre del mètode que executa el llançament d'un puny.

```

private void PunchStart()
{
    if (punching)
        return;
    punching = true;
    fist = ForgotNextFist() ? RandomFist() : NextFist();
    punchTrigger.Interact = true;
    punching = true;
    movementController.Acceleration = punchDrag;
    AnimatePunch(fist);
    OnPunchStart?.Invoke();
    punchParticle?.Play();
}
```

En la línia 7 podem veure el càlcul del següent puny a animar. Primer mirem que encara tenim memòria de la seqüència de l'execució de punys, si no és així, es llença un dels punys de forma aleatòria i en cas contrari se segueix amb la seqüència.

Per saber si encara es pot continuar amb la seqüència fem ús del mètode següent:

```
private bool ForgotNextFist()
{
    return punchMemoryTimer <= 0;
}
```

La variable de la línia 3 és una variable de classe que es va reduint a cada fotograma, com podem veure en el següent codi:

```
public void Update()
{
    if (punchMemoryTimer > 0)
        punchMemoryTimer -= Time.deltaTime;
}
```

9.1.5 Gestió de l'habilitat de raig

Aquest fitxer compte la lògica que instància l'habilitat de raig així com la gestió del refredament. Unity suporta la funcionalitat d'afegir esdeveniments en les

animacions. L'animació d'instanciar un raig del personatge és una mica llarga i per ajustar el màxim possible la cohesió entre l'animació i la creació del raig en l'escena, vam fer que sigui l'animació que executa la funció de creació del raig en el fotograma corresponent. A continuació veurem el mètode que invoca l'animació de raig.

```
private void InvokeRayAbility()
{
    animator.SetTrigger(CharacterAnimations.Ray);
    ResetRayCooldown();
    RayAbilityStart();
    OnRayStart?.Invoke();
}
```

Ara veurem el mètode que realment fa l'instancia del raig i que es executat per l'esdeveniment afegit a l'animació de raig.

```
 // called by ray player animation as event
public void InvokeRayBallInstance()
{
    Vector2 force =
        Vector2.right
        *
        FacingValue
        *
        projectileSpeed;
    VengefulProjectile instance =
        Instantiate(projectilePrefab,
                    projectileOrigin.position,
                    Quaternion.identity);
    instance.SetForce(force);
    instance.gameObject.Disposable(projectileTimeout);
    DOVirtual.DelayedCall(0.1f,
                          () => player.BaseGravity());
}
```

9.1.6 Curació

La interacció amb les fruites que trobem en el joc curen al personatge. L'efecte de curació s'aconsegueix gràcies a la següent lògica.

```
using Core.Player.Controller;
using UnityEngine;


namespace Core.ScriptableEffect
{
    [CreateAssetMenu(menuName = "Effect/HealthAdder")]
    public class HealthAdder : Effect
```

```
    {
        [SerializeField] int amount = 1;
        public override void Apply(GameObject other)
        {
            var player
                = other
                .GetComponent<PlayerController>();
            player?.Heal(amount);
        }
    }
}
```

La curació per nosaltres és un efecte. Aquest efecte en específic sobreescriu el mètode virtual de la seva classe pare, troba la instància del personatge principal en l'escena i finalment el cura un cert nombre de vides.

9.1.7 Rebre mal

A continuació veurem la lògica que s'executa quan algun element dintre de l'escena vol fer mal personatge principal:

```
public void Hurt(int damage, GameObject other)
{
        if (!CanBeHit || IsDead())
            return;

        ShakeCamera();
        Freeze();
        ResetAbilities();
        GameManager.Instance?.FreezeTime(0.01f);
        TakeDamage(other.transform, damage);
}
```

Creiem que el codi és prou explicatiu per si mateix, però explicarem algunes de les línies més interessants del mètode Hurt. En la línia número 3 es comprova si l'estat del personatge compleix els requisits per continuar l'execució. En aquest cas es necessita que es pugui picar al personatge, és a dir, que no estigui protegit i que encara estigui viu. Finalment, en la línia 10 es crida a la funció que realment modifica l'estat de les vides del personatge. La implementació d'aquesta funció la podem veure en el codi de sota.

```
private void TakeDamage(Transform agressor, int damage)
{
    SetHealth(currentHealth - damage);
    if (HasNoLifes())
        Die();
    else
        RecoverFromTakingDamage(agressor);
}
```

9.1.8 Protecció

A partir del moment en que el personatge pren mal se, li aplica una lògica de protecció que dura un periode curt de temps. Aquesta protecció la gestiona el component Protectable. Veurem un resum d'aquest component i explicarem l'implementació del mateix.

```
namespace Core.Utility
{
    public enum ProtectionType
    {
        INFINITE,
        NONE
    }

    public class Protectable : MonoBehaviour
    {
        public bool CanBeHit => hitProtectionTimer <= 0;
        private float hitProtectionTimer;
        public float ProtectionDuration {
            get => protectionDuration;
            private set => protectionDuration = value;
        }

        public void Update()
        {
            if (hitProtectionTimer > 0)
                hitProtectionTimer -= Time.deltaTime;
        }

        public void SetProtection(float duration)
        {
            hitProtectionTimer = duration;
        }

        public void SetProtection
        (
            ProtectionType protectionType
        )
        {
            if (protectionType == ProtectionType.INFINITE)
                hitProtectionTimer =
                    float.PositiveInfinity;
            else
                hitProtectionTimer = 0f;
        }
    }
}
```

En la línia 3, definim un tipus de dada que fa referència al tipus de protecció

que pot rebre el personatge. Tot i que l'efecte de protecció pot ser configurat explicitament per segons (mirar línia 24) fem ús del tipus ProtectionType en casos on realment no sabem quant de temps ha de durar l'efecte de protecció, mirar la implementació del mètode de la línia 29. En qualsevol cas, ja sigui aplicant protecció explícita per segons o pel tipus ProtectionType, el valor de la variable CanBeHit de la línia 11 canvia, ja que la variable de la qual depen l'estat es veu modificada en cada fotograma, mirar implementació del mètode Update.

9.1.9 Horientació del personatge

La gestió de per quin costat ha d'estar mirant el personatge en tot moment el gestiona el component FacingController. A continuació veurem un breu resum de l'implementació.

```
using Core.Shared.Enum;
using UnityEngine;

namespace Core.Player.Controller
{
    public class FacingController : MonoBehaviour
    {
        public Face Facing { get; private set; }

        public void Update()
        {
            bool canBeControlled = PlayerController
                                    .Instance
                                    .Controllable;
            if (!canBeControlled) return;
            float input = Input.GetAxisRaw("Horizontal");
            if (Mathf.Abs(input) <= 0) return;
            if (input < 0)
                Facing = Face.Left;
            else Facing = Face.Right;
        }
    }
}
```

Com podem veure en el codi anterior, en el mètode Update que s'executa en cada fotograma escolta les tecles que mouen al personatge en l'eix vertical, mirar línia 16. Els possibles valors que retorna la instrucció GetAxisRaw va entre -1 a 1. Realment els casos on aquest valor és 0 no ens importen, ja que actualitzaríem l'estat de la variable de Facing sense la interacció amb el jugador. Si és diferent de zero, modifiquem la variable de Facing, i en ser una variable pública, la resta de components tenen constància cap a quin costat el personatge ha de mirar.

9.2 Interacció de l'escena amb el personatge.

A continuació veurem la lògica que hi ha darrera de l'interacció dels elements de l'escena amb el personatge. En aquest apartat veurem únicaments els casos ón el personatge es totalment pasiu, i són els elements de l'escena que modifiquen l'estat del personatge. Per exemple, si caiem sobre Liquids 8.5.4.2, el personatge rep mal i es reposiciona en l'ultim punt de control pel qual ha pasat. Un altre exemple tipic es que els enimics de les escenes quan toquen al personatge principal li fan mal, encara que aquest últim estigui quiet.

9.2.1 Component Hazard

Aquest component és realment molt utilitzat al llarg del joc, tots els enemics l'incorporen. S'encarrega de detectar quan el jugador entra amb contacte amb l'objecte i aplica dany al personatge principal. Aquest script comprova a cada fotograma. Si està interactuen amb el personatge, si és així s'executa el mètode Hurt de l'API del component PlayerController.

A continuació explicarem el funcionament del component a través de la implementació per codi.

```
1  namespace Core.Combat
2  {
3      public class Hazard : MonoBehaviour
4      {
5          [SerializeField] HealthTaker healthTaker;
6  
7          public void Update()
8          {
9              CheckCollision();
10         }
11 
12         private void CheckCollision()
13         {
14             var player = PlayerController.Instance;
15 
16             if (!player.CanBeHit)
17                 return;
18 
19             if (!IsTouchingPlayer())
20                 return;
21 
22             healthTaker.Apply(
23                 gameObject,
24                 player.gameObject);
```

```
        }

        private bool IsTouchingPlayer()
        {
            var myCollider = GetComponent<Collider2D>();
            var playerCollider = PlayerController
                                    .Instance
                                    .BodyCollider;
            return myCollider.IsTouching(playerCollider);
        }
    }
}
```

En cada fotograma es comprova si es pot aplicar mal amb les condicions de les línies 16 i 19. Si és així, s'utilitza un objecte de tipus HealthTaker per aplicar una lògica, que en aquest nivell no és visible, però està clar que aplica mal. Per què s'encapsula la lògica llavors dintre d'un tipus nou? La resposta és simple, hi ha altres objectes en l'escena que poden aplicar mal al personatge principal i no tenen perquè tots tenir Hazard com a component. D'aquesta manera deleguem a un tercer la lògica d'aplicar mal i aconseguim evitar repeticions de codi al llarg dels scripts.

A continuació veurem la implementació del component HealthTaker.

```
using Core.Player.Controller;
using UnityEngine;


namespace Core.ScriptableEffect
{
    [CreateAssetMenu(menuName = "Effect/HealthTaker")]
    public class HealthTaker : Effect
    {
        [SerializeField] int amount = 1;
        public override void Apply
        (
            GameObject self,
            GameObject other
        )
        {
            var player = other
                        .GetComponent<PlayerController>();
            player?.Hurt(amount, self);
        }
    }
}
```

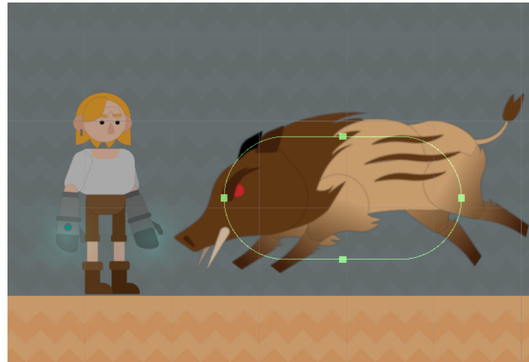

Figura 9.2: Collider que detecta sobreposició amb el personatge principal.

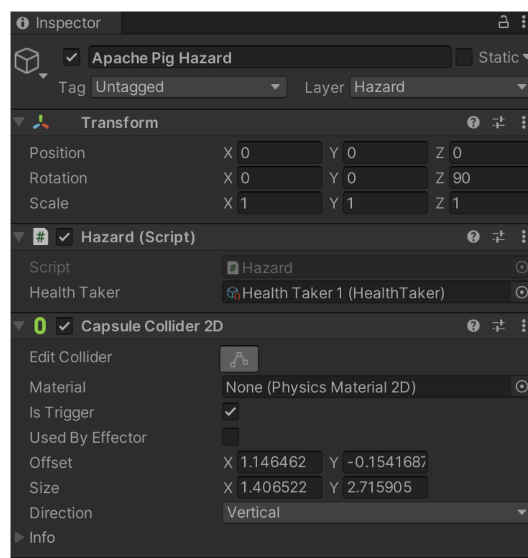

Figura 9.3: Component Hazard afegit a ApachePig per fer mal per interacció.

9.2.2 Component DeadlyObject

El component DeadlyObject està pensat per afegir-lo a objectes d'escena que destrueixen complementament els objectes amb els quals interactuen. Com que, per disseny, no volem que aquest comportament s'apliqui al personatge principal, el que fem és que, en el moment d'interactuar amb un d'aquests objectes, es torna a carregar l'escena i es reposiciona al personatge en l'últim punt de control i se li pren un nombre fix de vides. Exemples d'objectes que tenen aquest comportament: trampes, lava o aigua. A continuació veurem la implementació d'aquest script.

```
1 using System.Collections;
2 using Core.GameSession;
3 using Core.Player.Controller;
4 using UnityEngine;
```

```
5
6  namespace Core.Combat
7  {
8      public class DeadlyObject : MonoBehaviour
9      {
10         [SerializeField] int damage = 1;
11         const float cWaitTime = 0.5f;
12         bool playerIn = false;
13
14         private void OnTriggerEnter2D(Collider2D other)
15         {
16             if (!playerIn
17                 &&
18                 other.gameObject.tag == "Player")
19             {
20                 playerIn = true;
21                 StartCoroutine(ResetSavePoint());
22             }
23             else if (other.gameObject.tag != "Player")
24             {
25                 Destroy(other.gameObject);
26             }
27         }
28
29         private IEnumerator ResetSavePoint()
30         {
31             PlayerController player = PlayerController
32                                           .Instance;
33             player.Hurt(damage, gameObject);
34
35             yield return new WaitForSeconds(cWaitTime);
36
37             if (player.IsAlive())
38                 GameSessionController
39                     .Instance
40                     .PlacePlayer();
41             playerIn = false;
42         }
43     }
44 }
```

9.3 Lluita per fases

Tots els enemics a excepció del Bosses segueixen l'ordre típic de creació de comportament, és a dir, crear tasques i ajuntar-les sota la responsabilitat de tasques pares, generant així arbres que a la vegada podem ser subarbres d'altres arbres de més nivell. La Figura 9.4 representa l'execució dels comportaments anotats com 1, 2 i 3 de forma consecutiva.

Figura 9.4: Exemple d'arbre de comportament simple.

Aquesta solució és correcta i serveix per crear intel·ligències artificials simples i complexes. Però, cada cop que la intel·ligència artificial es fa més complexa és més difícil de mantenir. Per exemple, imaginem que volem crear un nou comportament, l'anomenem comportament 4, que sigui l'execució de la seqüència dels comportaments 1 i 2. Afegir aquest comportament significaria crear un nou arbre amb una tasca pare de seqüència i que a la vegada aquesta tasca de seqüència tingui les còpies dels comportaments 1 i 2. Mirar Figura 9.5.

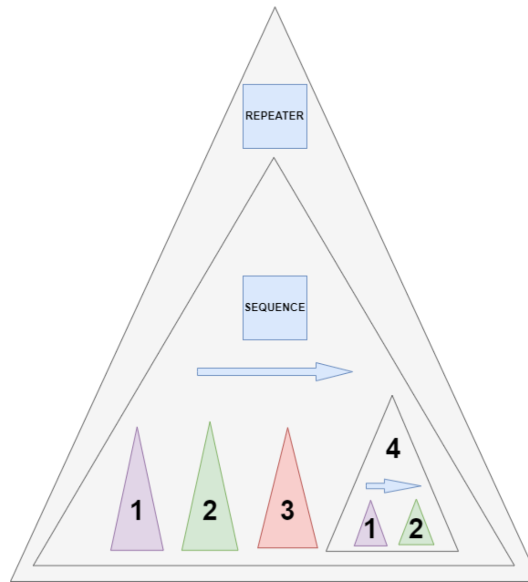

Figura 9.5: Nou comportament en arbre simple.

Com podem veure en la imatge anterior, fer combinacions de comportaments i afegir-los com un nou comportament implica la creació d'un nou arbre. Ens podem imaginar que fer totes les combinacions de subconjunts de comportaments del nostre conjunt de comportaments inicial implica que l'arbre de comportament creixi exponencialment. Si a més volguéssim implementar una intel·ligència artificial capaç d'emular un progrés, és a dir, primer agafar un subconjunt de comportaments petits i després anar agafant subconjunts amb més nombre de comportaments, o bé, seleccionar aquells subconjunts de comportaments que són més simples quant a jugabilitat i posteriorment aquells més complicats. Això, encara seria més complicat de gestionar i de mantenir a causa de les múltiples copies i gestió dels comportaments seleccionats per subconjunt de nou comportament.

La solució que vam trobar a aquest problema és la creació un nou tipus de tasca pare, capaç de seleccionar subconjunts de tasques amb una variable compartida la qual representa el progrés. D'aquesta manera, els arbres es queden amb l'arquitectura de l'arbre de comportament de la Figura 9.4 i només cal canviar la tasca pare per un altre capaç d'emular progres.

A continuació us presentem algues de les parts més interessants de la tasca pare de gestió de selector de comportaments per fases. Aquesta tasca pare hereta del component Composite que és un component propi de Behavior Designer.

```
1 using System.Collections.Generic;
2 using System.Linq;
```

```
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;
using UnityEngine;

namespace Core.AI
{
    public class StageSelector : Composite
    {
        public SharedInt CurrentStage;
        public List<string> IncludedTasksPerStage;
        private List<int> childIndexList =
                                    new List<int>();
        private Stack<int> childrenExecutionOrder =
                                    new Stack<int>();
        private TaskStatus executionStatus =
                                    TaskStatus.Inactive;

        public override void OnStart()
        {
            childIndexList.Clear();
            childIndexList =
                IncludedTasksPerStage[
                    Clamp(CurrentStage.Value)
                ]
                .Split(',')
                .Select(int.Parse)
                .ToList();

            // Randomize the indecies
            ShuffleChilden();
        }

        private int Clamp(int currentStage)
        {
            return Mathf
                .Clamp( currentStage,
                        0,
                        IncludedTasksPerStage.Count - 1);
        }

        private void ShuffleChilden()
        {
            for (int i = childIndexList.Count; i > 0; --i)
            {
                int j = Random.Range(0, i);
                int index = childIndexList[j];
                childrenExecutionOrder.Push(index);
                childIndexList[j] = childIndexList[i - 1];
                childIndexList[i - 1] = index;
```

```
52                }
53            }
54        }
55    }
```

Els únics elements que s'han de definir a la tasca són: la variable compartida que representa en quina fase estem i una llista de cadenes de caràcter, on cada cadena de caràcter són números intercalats amb comes. Aquests números representen els índexs del subarbres fills. Per exemple en la Figura 9.4, el subarbre 1 té índex 0, el 2 índex 1 i així anar fent. Mirar línies 11 i 12 del codi anterior per veure la declaració d'aquestes dues variables.

Cada cop que s'executa el mètode OnStart de la línia 20, s'agafa la llista IncludedTasksPerStage i s'accedeix per indexació a la cadena de caràcters de la fase actual. Abans de fer la indexació, es fa una normalització entre 0 i el nombre màxim de subtasques per evitar errors d'accés. Es transforma la cadena de caràcters en una llista de números, on aquesta llista de números representen els índexs dels subarbres a executar. Finalment, per donar més dinamisme, l'execució de les subtasques filles s'ordenen aleatòriament amb el mètode ShuffleChildren de la línia 43.

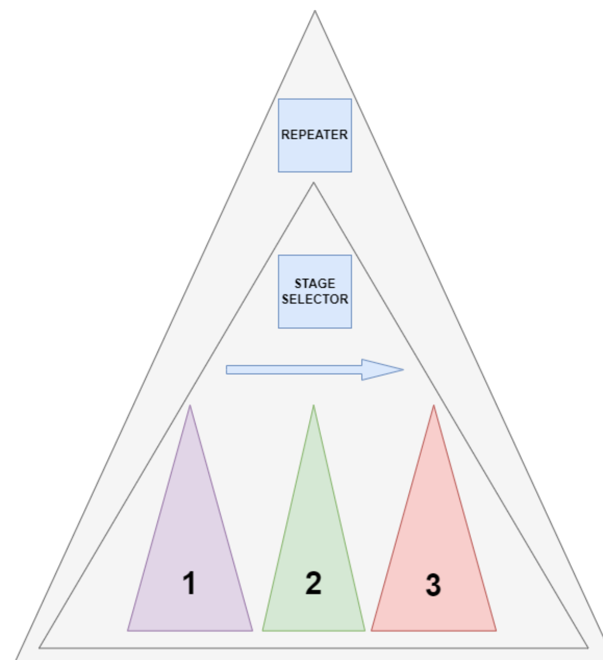

Figura 9.6: Arquitectura d'arbre de comportament per IA complexa.

9.4 Tasques

Per implementar les diferents intel·ligències artificials amb arbres de comportament, és necessari la creació de múltiples tasques. El projecte compte un nombre molt gran de tasques condicionals i tasques d'accions. A continuació explicarem la implementació d'alguna d'aquestes tasques.

9.4.1 Detecció de canvi de fase

Com hem detallat prèviament, algunes de les intel·ligències artificials tenen un comportament d'evolució. Aquest comportament d'evolució es fa a través del mecanisme de lluita per fases detallat en l'Apartat 9.3.

El canvi de fase es fa quan l'enemic té menys vides que el nombre de vides mínimes que pot tenir en una fase. Per tant, és necessari la creació d'una tasca condicional per a avaluar el nombre de vides que té l'enemic.

```
1 using BehaviorDesigner.Runtime;
2 using BehaviorDesigner.Runtime.Tasks;
3 using Core.Combat.IA;
4
5 namespace Core.IA.Task.Conditional
6 {
7     public class IsHealthUnder : EnemyConditional
8     {
9         public SharedInt healthThreshold;
10
11        public override TaskStatus OnUpdate()
12        {
13            return destroyable.CurrentHealth
14                    <
15                    healthThreshold.Value
16                    ?
17                    TaskStatus.Success
18                    :
19                    TaskStatus.Failure;
20        }
21    }
22 }
```

9.4.2 Detecció de l'orientació del jugador en l'escena

Algunes intel·ligències artificials, sobretot les que són mòbils necessiten saber l'orientació personatge principal respecte d'elles, perquè, o bé, les seves habilitats depenen de la posició del jugador, o simplement perquè se les ha de poder

escalar de tal manera que mirin al jugador per tal que no quedin rares les seves accions i que no estiguin mirant el jugador mentre les fan. La solució a aquest problema és la implementació d'una tasca que executi la lògica per saber si la intel·ligència artificial està mirant efectivament en el sentit que el personatge està o no. El següent codi implementa aquesta lògica.

```
using BehaviorDesigner.Runtime.Tasks;
using UnityEngine;

namespace Core.Combat.IA.Conditional
{
    public class FacingPlayer : EnemyConditional
    {
        public override void OnAwake()
        {
            base.OnAwake();
        }

        public override TaskStatus OnUpdate()
        {
            var toPlayer = (
                player.transform.position
                -
                transform.position).normalized;
            if (Mathf.Abs(toPlayer.x) > 0)
            {
                if (toPlayer.x * transform.localScale.x
                    >
                    0)
                    return TaskStatus.Success;
                else return TaskStatus.Failure;
            }
            return TaskStatus.Success;
        }
    }
}
```

9.4.3 Orientar la IA cap al jugador

Com bé hem explicat en l'Apartat 9.4.2 moltes vegades la intel·ligència artificial necessita orientar-se en el sentit del jugador. La tasca que executa aquesta acció és simple. Aquesta tasca només ha de ser capaç de canviar l'escalat horitzontal dintre de l'escena.

```
using BehaviorDesigner.Runtime.Tasks;

namespace Core.Combat.IA.Action
{
```

```
    public class FacePlayer : EnemyAction
    {
        float baseScaleX;

        public override void OnAwake()
        {
            base.OnAwake();
            baseScaleX = transform.localScale.x;
        }

        public override TaskStatus OnUpdate()
        {
            var scale = transform.localScale;
            scale.x = transform.position.x
                >
                player.transform.position.x
                ?
                -baseScaleX
                :
                baseScaleX;
            transform.localScale = scale;
            return TaskStatus.Success;
        }
    }
}
```

9.4.4 Llançament de projectils

Hi ha alguns enemics que tenen l'habilitat d'instanciar projectils, com ara els dos Bosses i la Planta. Per fer la instància d'aquests projectils s'utilitza la implementació, que no només instància els projectils, sinó que també permet executar la lògica que fa l'efecte de tremolor.

```
using BehaviorDesigner.Runtime;
using BehaviorDesigner.Runtime.Tasks;
using Core.Combat;
using Core.Combat.IA;
using Core.Manager;
using UnityEngine;

public class Shoot : EnemyAction
{
    public Weapon[] weapons;
    public SharedBool shakeCamera;
    public SharedFloat shakeIntensity = 1;

    public override TaskStatus OnUpdate()
    {
```

```
        foreach (var weapon in weapons)
        {
            var projectile = Object
                            .Instantiate(
                            weapon.projectilePrefab,
                            weapon.weaponTransform
                                        .position,
                            weapon.weaponTransform
                                        .rotation);
            projectile.Shooter = gameObject;
            var force =
                new Vector2(
                    weapon.horizontalForce
                    * transform.localScale.x,
                    weapon.verticalForce);
            projectile.SetForce(force);
            if (shakeCamera.Value)
                CameraManager
                    .Instance?
                    .ShakeCamera(shakeIntensity.Value);
        }
        return TaskStatus.Success;
    }
}
```

9.4.5 Destrucció de la intel·ligència artificial

Totes les intel·ligències artificials tenen el comportament de ser destruïdes pel jugador. Les més simples, perquè ja no tenen vides, i les que són més complexes perquè el nombre de fases ha estat completat. Hi ha intel·ligències artificials que, a l'hora de ser destruïdes, han de desaparèixer i deixar un cadàver i altres que l'últim estat de les animacions ja representa la mort. També hi ha intel·ligències artificials que la seva destrucció succeeix després de l'execució d'efectes de partícules, per tant, s'ha de poder gestionar la demora de l'execució d'autodestrucció.

A continuació veurem la tasca que té la feina d'emular la mort de les intel·ligències artificials separades en trossos de codi perquè la implementació de la mateixa es força extensa.

Declaració dels atributs de classe.

```
public SharedParticleSystem explosionEffect;
public SharedParticleSystem bleedEffect;
public SharedFloat bleedDuration;
public SharedSpriteRenderer deadBody;
[SerializeField] bool destroyGameObject = true;
```

Els efectes de partícules serveixen per donar dramatisme al moment de la mort de la intel·ligència artificial. Es pot definir un efecte de partícules pel sagnat i per l'explosió després del sagnat. També es pot configurar el temps de sagnat amb l'atribut bleedDuration. Amb la variable deadBody podem ajuntar un objecte que sigui el cadàver de la intel·ligència artificial després de fer l'explosió. Finalment, podem definir si la intel·ligència artificial es destrueix realment i ha de desaparèixer de l'escena o pel contrari s'ha de quedar, és molt útil en els casos que la intel·ligència tingui una animació de mort en la màquina d'estat d'animacions i no es necessita un objecte de suplantació per fer de cadàver.

Al primer fotograma que s'activa la tasca de destrucció executa el mètode OnStart, el qual és l'encarregat d'instanciar els efectes de partícules durant el temps configurat en els atributs de classe.

```
public override void OnStart()
{
    if (bleedEffect.Value)
    {
        EffectManager
            .Instance?
            .PlayOneShot(
                bleedEffect.Value,
                transform);
    }
    DOVirtual
        .DelayedCall(
        bleedDuration.Value,
        KillEnemy);
}
```

Finalment, quan ha passat el temps de sagnat, s'executa el mètode KillEnemy, aquest s'encarrega de completar la tasca, instància el cadàver del personatge si així ha sigut configurat i s'encarrega de destruir o no segons la configuració.

```
 private void KillEnemy()
{
    if (explosionEffect.Value)
    {
        EffectManager
        .Instance?
        .PlayOneShot(
            explosionEffect.Value,
            transform);
    }

    if (deadBody.Value)
    {
```

```
14          SpawnDeadBody();
15      }
16      completed = true;
17      if (destroyGameObject)
18          Object.Destroy(gameObject);
19  }
```

9.5 Gestió de recursos dintre de l'escena

Una de les problemàtiques que ens vam trobar mentre treballaven en la implementació del joc és que en temps d'execució es creen molts objectes dintre de les escenes que deixen de tenir sentit al cap de pocs segons. Per exemple, cada cop que el personatge principal salta o llença punys, es generen efectes de partícules. Aquestes partícules, encara que només es veuen un cop, estan present en l'escena. Una bona pràctica és destruir aquests objectes que es creen a mesura que deixen de ser útils per estalviar recursos.

Unity ja té pensat una forma de destruir objectes.

```
public static void Destroy(Object obj, float t = 0.0F);
```

El mètode del codi anterior rep un paràmetre objecte i un cert temps. El temps serveix per fer compte enrere en segons per destruir l'objecte.

Si per exemple, volem instanciar nous efectes de partícules podríem agafar la nova instància i executar el mètode creat per Unity amb la nova instància, gràcies al fet que els objectes es poden destruir dinàmicament li pot costar entendre.

Una altra manera d'afrontar el problema de la destrucció d'objectes en l'escena és que cada cop que es crea una nova instància, l'afegim un component on l'única funcionalitat és destruir l'objecte que el conté. Això, encara que no ho sembli, és molt més significatiu, ja que, fent ús de l'inspector d'Unity, podem veure que totes les noves instàncies tenen amb elles un component d'autodestrucció. A continuació veurem un exemple d'implementació per aconseguir aquest comportament.

```
using UnityEngine;

namespace Core.Utility
{
    public class Disposable : MonoBehaviour
    {
        [SerializeField] float lifetime = 1f;

        public float Timeout
        {
            get => lifetime;
            set => lifetime = value;
        }

        public void Start()
        {
```

```
17              Destroy(gameObject, Timeout);
18          }
19      }
20  }
```

El component Disposable, representa una cosa d'un sol ús. Té la instrucció de destruir l'objecte que el conté en el mètode Start. Recordem que aquest mètode s'executa en el primer fotograma quan el component està actiu. Per tant, l'objecte que el conté es destruirà al cap dels pocs segons configurats per la variable timeout. Mirar línia 17.

Podem afegir aquest component a qualsevol GameObject dinàmicament de la següent manera:

```
1 var disposable = gameObject
2                     .AddComponent(Disposable);
```

Aquesta implementació sembla prou bona, però encara la podem millorar. C# és un llenguatge molt potent i permet afegir mètodes als tipus existents sense crear un nou tipus derivat, recompilar o modificar d'una manera o l'altra el tipus original. Els mètodes d'extensió són mètodes estàtics, però se'ls crida com si fossin mètodes d'instància al tipus estès. Així, doncs, podem expandir el tipus GameObject amb nous mètodes, mètodes que podem cridar des de qualsevol script oblidant-nos de la implementació i únicament tenint en compte la seva funcionalitat. Per tant, el que farem serà crear un mètode expandint la classe, i que aquest mètode s'encarregui d'afegir el component Disposable a la instància, per tal de destruir-lo més endavant. Aquest mètode clar està que necessita com a paràmetre els temps en segons.

El següent codi expandeix la definició de tipus GameObject afegint-li el mètode d'autodestrucció.

```
1  using UnityEngine;
2
3  namespace Core.Utility
4  {
5      public static class GameObjectExtensions
6      {
7          public static void Disposable
8          (
9              this GameObject gameObject,
10             float timeout
11         )
12         {
13             var disposable =
14                     gameObject
15                     .AddComponent<Disposable>();
```

```
                disposable.Timeout = timeout;
            }
        }
}
```

L'ús d'aquest mecanisme ha estat molt útil a l'hora d'instanciar efectes de partícules o efectes de sons. A continuació veurem un exemple: Tenim una classe pensada per fer instàncies d'efectes de partícules. Podem veure que en la línia 12 es fa la instància d'una nova partícula, a la línia 20 es fa un càlcul per saber quant triga la partícula a fer l'efecte complet i finalment a la línia 23, fem ús del mètode Disposable com a mètode d'instància per destruir la partícula.

```

public void PlayOneShot
(
    ParticleSystem particleSystem,
    Transform position = null,
    Transform parent = null
)
{
    if (particleSystem == null || transform == null)
        return;

    var effect = Instantiate(
            particleSystem,
            position.position,
            Quaternion.identity);
    if (parent)
        effect.transform.SetParent(parent);
    effect.Play();

    var duration =  effect.main.duration
                    +
                    effect.main.startLifetime.constantMax;
    effect.gameObject.Disposable(duration);
}
```

9.5.1 Gestió dels cadàvers dels enemics

La majoria dels enemics, quan es moren, l'objecte que contenia a l'enemic es destrueix i es crea un objecte que és el cadàver, i el qual el personatge principal no pot interactuar. Aquests cadàvers són objectes nous que apareixen i s'ha de poder gestionar en quina capa visual de l'escena apareixen.

Els enemics, quan estan vius, poden estar en qualsevol de les capes visuals de l'escena, però per evitar que els cadàvers treguin importància a l'escena, vam decidir que els cadàvers sempre apareguin un una de les capes visuals mes d'endarrere. El mecanisme que uneix les capes amb els cadàvers dels enemics és es diu CorpsesManager i la seva implementació és la següent:

```
1 using Core.Shared.Enum;
2 using UnityEngine;
3
4 namespace Core.Manager
5 {
6     public class CorpsesManager : MonoBehaviour
7     {
8         private enum SpawnType
9         {
10            Global,
11            Local
12        }
13
14        [SerializeField] private SpawnType spawnType;
15        [SerializeField] private Transform
16                        localSpawnPoint;
17
18        public static CorpsesManager Instance;
19
20        public void Awake()
21        {
22            Instance = this;
23        }
24
25        public void
26                Spawn(
27                    SpriteRenderer sprite,
28                    Vector2 position)
29        {
30                var instance = Instantiate(
31                                    sprite,
32                                    position,
33                                    Quaternion.identity);
34            Attach(instance.gameObject);
35        }
```

```

        public void
            Spawn(
                SpriteRenderer sprite,
                Vector2 position,
                Face facing)
            {
                var instance =
                    Instantiate(sprite, position,
                        facing == Face.Left
                        ? Quaternion.Euler(0, -180, 0)
                        : Quaternion.identity);
                Attach(instance.gameObject);
        }
    }
}
```

La implementació d'aquesta lògica és senzilla, es crida a qualsevol de les dues variants del mètode Spawn i amb aquest mètode i la definició de la variable localSpawnPoint, es fa la instància dels cadàvers en la capa visual corresponent.

9.6 Gestor de la sessió de joc

El gestor de sessió de joc és un objecte dintre de l'escena que té associat el component GameSesionController. Aquest objecte està present des de la primera escena que es veu en entrar al joc, és a dir l'escena del menú principal. Només hi ha una instància d'aquest objecte al llarg del lloc i no es destrueix en passar entre escena i escena.

A continuació veurem els trossos de codi més interesants de la implementació d'aquest component.

Cada cop que una escena és carregada, s'executa el mètode OnSceneLoaded, el qual té la responsabilitat de filtrar les escenes jugables i de posicionar el jugador a l'entrada correcte.

```
1  private void OnSceneLoaded(
2        Scene scene,
3        LoadSceneMode mode)
4  {
5      if (nonPlayableScene)
6          return;
7
8      inDieProcess = false;
9
10     if (loadData) //player has died
11     {
12         LoadSavedData();
13         PlacePlayer();
14     }
15     else if (entranceTag != EntranceID.None)
16     {
17         SearchEntrance();
18     }
19 }
```

El mètode SavePlayerState s'encarrega de comunicar-se amb el sistema de guardat, guardant les habilitats, vides i posició del jugador en el moment de guardat.

```
1   public void SavePlayerState(Transform savePoint)
2  {
3      if (inDieProcess)
4          return;
5
6      PlayerState playerState = new
7          PlayerState(((int)SceneManagementFunctions
```

```
        .GetCurrentSceneEnum()),
        PlayerController.Instance.PlayerData
            .Health.HP,
            PlayerController.Instance.PlayerData
            .Health.MaxHP,
            savePoint.position,
            PlayerAbilitiesAdquiredSnapshot());

        SaveSystem.SavePlayerState(playerState);
        currentSavePos = savePoint.position;
}
```

El mètode SavePlayerCurrentPoint és un mètode que s'utilitza des de la lògica de punt de control per guardar l'últim punt pel qual ha passat el personatge per poder reposicionar-lo en cas que sigui necessari.

```
 public void SavePlayerCurrentPoint(
            Transform currentPointTransform)
{
    if (inDieProcess)
        return;
    currentSavePos
        = currentPointTransform
            .position;
}
```

El mètode RecoverLastSaveScene s'encarrega de recuperar les dades de l'últim estat guardat del personatge.

```
public void RecoverLastSaveScene()
{
    FindObjectOfType<InGameCanvas>()?.ActiveDeathImage();
    loadData = true;
    PlayerState playerState = SaveSystem
            .LoadPlayerState();
            StartCoroutine( Loader
                            .LoadWithDelay((SceneID)
   playerState.scene,
                            5f));
}
```

El mètode SearchEntrance ès la lògica que s'executa per trobar el punt d'entrada de l'escena a la qual ens desplacem. A més, executa l'animació d'entrada d'escena del personatge.

```

private void SearchEntrance()
```

```
3 {
4     SceneEntrance[] lstSceneEntrance =
5     FindObjectsOfType<SceneEntrance>();
6     int i = 0;
7 
8     while (
9         i < lstSceneEntrance.Length
10        &&
11        entranceTag != EntranceID.None
12    )
13    {
14        SceneEntrance se = lstSceneEntrance[i];
15        if (
16            se.gameObject
17            .CompareTag(entranceTag.ToString()))
18        {
19            se.MakeEntrance();
20            currentSavePos = se.GetEntrancePoint();
21            entranceTag = EntranceID.None;
22        }
23        i++;
24    }
25 
26    if (entranceTag != EntranceID.None)
27    {
28        Debug
29        .LogError("GameSessionController
30        .SearchEntrance: " +
31        "Entrance not found. Entrance tag: "
32        + entranceTag.ToString());
33        entranceTag = EntranceID.None;
34    }
35 }
```

Finalment, veurem l'implementació del mètode que s'encarrega de recuperar les dades guardades.

```
1 
2         private int LoadSavedData()
3         {
4             PlayerState playerState = SaveSystem.
   LoadPlayerState();
5 
6             var playerHealth = PlayerController
7                                 .Instance.PlayerData.Health;
8             playerHealth.HP = playerState.max_health;
9             playerHealth.MaxHP = playerState.max_health;
10 
11            LoadAbilitiesAdcquired(playerState);
12 
```

```
13             currentSavePos = playerState.GetPosition();
14             loadData = false;
15 
16             return playerState.scene;
17         }
```

9.7 Punts de control

Els punts de control serveixen per recordar posicions pròximes a elements que tenen un grau de dificultat per ser completades amb èxit. Per exemple, plataformes on sota hi ha lava o trampes. Fem ús d'aquests punts de control perquè d'alguna manera havíem de ser capaços de reposicionar al personatge quan cau en llocs on, en principi, no hi ha forma de recuperar-se (mirar Apartat 9.2.2), però encara té vides per poder continuar jugant.

Els objectes de control estan formats únicament dels elements necessaris perquè hi hagi detecció de físiques entre elements, que recordem són un RigidBody i almenys un Collider i finalment el component SavePoint.

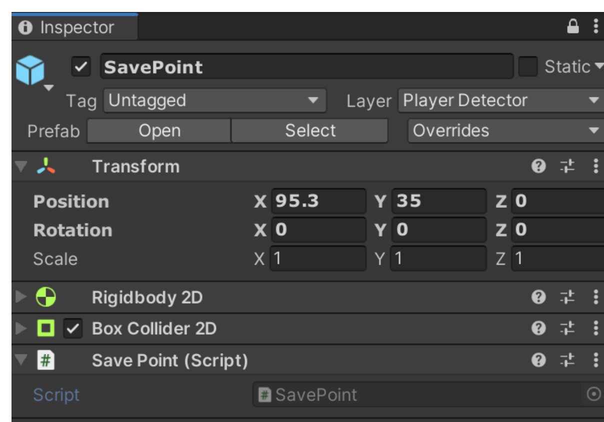

Figura 9.7: Components d'un punt de control.

```
using UnityEngine;

namespace Core.GameSession
{
    public class SavePoint : MonoBehaviour
    {
        private void OnTriggerEnter2D(Collider2D other)
        {
            if (other.gameObject.tag == "Player")
            {
                GameSessionController
                    .Instance
                    .SavePlayerCurrentPoint(transform);
            }
        }

    }
}
```

La lògica del component SavePoint és simple, només s'encarrega d'actualitzar la posició del jugador amb la posició pròpia de l'objecte de punt de control. Aquesta lògica s'executa dintre del mètode OnTriggerEnter2D que és un mètode gestionat per Unity que s'executa quan el motor de física d'Unity detecta sobreposició entre dos objectes en l'escena.

9.8 Gestió de canvis d'escena

Una de les característiques dels metroidvaines és que no s'avança de forma seqüencial pels escenaris. Això fa que una escena pugui tenir moltes connexions amb altres escenes. Davant d'aquesta necessitat, vam implementar un sistema de comunicacions entre escenes compost pel GameSessionController i dos objectes: un encarregat de comunicar quan el player canvia d'escena; el SceneTeleporter, i l'altre que fa el procés d'entrada; el SceneEntrance.

Com hem explicat a la Secció 9.6, el component GameSessionController utilitza el mètode SearchEntrance() quan es canvia d'escena, per buscar l'objecte SceneEntrance encarregat de fer l'entrada del jugador.

```
private void SearchEntrance()
{
    SceneEntrance[] lstSceneEntrance =
    FindObjectsOfType<SceneEntrance>();
    int i = 0;

    while (i < lstSceneEntrance.Length
    && entranceTag != EntranceID.None)
    {
        SceneEntrance se = lstSceneEntrance[i];
        if (se.gameObject
        .CompareTag(entranceTag.ToString()))
        {
            se.MakeEntrance();
            currentSavePos = se.GetEntrancePoint();
            entranceTag = EntranceID.None;
        }
        i++;
    }

    if (entranceTag != EntranceID.None)
    {
        Debug.LogError(
        "GameSessionController.SearchEntrance: "
        + "Entrance not found. Entrance tag: "
        + entranceTag.ToString());
        entranceTag = EntranceID.None;
    }
}
```

Com podem veure, aquest mètode busca, entre tots els objectes de tipus SceneEntrance, el que té el Tag corresponent a la variable entranceTag. Aquests objectes estan posicionats, estratègicament, a cada possible entrada de l'escena per, quan sigui necessari, fer l'animació d'entrada del jugador. Per la identifica-

ció, han d'estar etiquetats, així que els SceneEntrance d'una mateixa escena han de dur una etiqueta diferent.

```
public enum EntranceID
{
    None, E1, E2, E3, E4, E5
}
```

Això s'ha fet així perquè passar informació d'una a una altra escena és un procés que requereix de ScriptableObjects o variables en components que romanguin entre escenes; no hi ha manera de fer pas de paràmetres.

Com podem veure el SceneEntrance fa l'entrada del jugador a través del MakeEntrance().

```
public void MakeEntrance()
{
    var player = PlayerController.Instance;
    //positioning
    player.transform.position = spawnPoint.position;
    //where to face player
    Face facing =
    spawnPoint.position.x - entrancePoint.position.x > 0
    ? Face.Left : Face.Right;
    //player not controllable
    player.Controllable = false;

    MovePlayer.Trigger(entrancePoint, entranceWaitTime,
    facing, entranceTime,
    () =>
    {
        player.Controllable = true;
    });
}
```

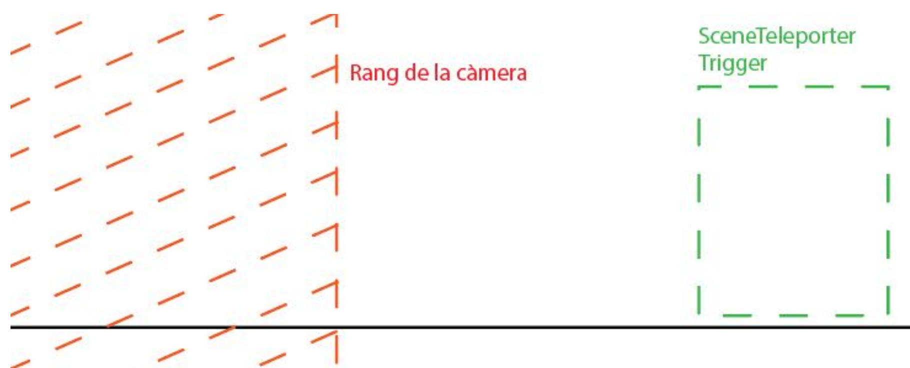

Figura 9.8: Posicionament de l'objecte SceneTeleporter a l'escena.

Aquest mètode posiciona el jugador en un punt fora del rang de la càmera i es crida el mètode MovePlayer.Trigger() que s'encarregarà de traslladar el jugador fins a la posició final de l'entrada al mateix temps que està amb l'animació de córrer. En acabar la translació, es crida l'acció final que torna el control a l'usuari.

```
public static void Trigger(Transform targetPoint, float
    waitTime, Face facing,
float moveTime = 0f, Action OnMoveEnded = null)
{
    OnStartMovement(facing);
    var player = PlayerController.Instance;
    float distance =
    Mathf.Abs(player.transform.position.x
    - targetPoint.position.x);

    player.transform.DOMove(targetPoint.position,
    moveTime <= 0f
    ? OptimalMovementTime(distance) : moveTime)
        .SetDelay(waitTime)
        .OnComplete(
            () =>
            {
                OnEndMovement(OnMoveEnded);
            }
        );
}
```

Com hem vist, la variable entranceTag del GameSessionController, és l'encarregada de, quan es fa el canvi d'escena, indicar quina és l'etiqueta a buscar. S'assigna a través del component SceneTeleporter.

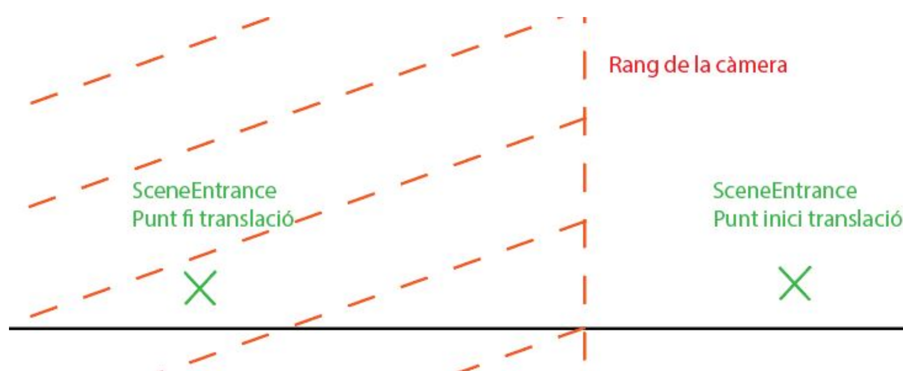

Figura 9.9: Posicionament de l'objecte SceneEntrance a l'escena.

L'objecte SceneTeleporter es col·loca fora dels límits del rang de la càmera, en els punts de sortida de l'escena. Això fa que, quan el player col·lisioni amb

el seu trigger s'assigni l'etiqueta a l'entranceTag per trobar l'entrada a l'escena que es carrega posteriorment amb el mètode LoadWithDelay().

```
public class SceneTeleporter : MonoBehaviour
{
    [SerializeField] SceneID scene;
    [SerializeField] EntranceID entranceTag;

    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.gameObject.tag == "Player")
        {
            GameSessionController.Instance.
NextSceneEntrance(entranceTag);
            StartCoroutine(Loader.LoadWithDelay(scene, 0))
;
        }
    }
}
```

El mètode Loader.LoadWithDelay() carrega l'escena de càrrega. Aquesta escena té el component LoadingMenu que, a través de LoadNextScene(), s'esperarà fins que la següent escena jugable estigui totalment carregada.

```
private LoadNextScene()
{
    Application.backgroundLoadingPriority = ThreadPriority
.Low;
    SceneManager.LoadSceneAsync(sceneName, LoadSceneMode.
Single);
}
```

Tots aquests components, a part de ser funcionals, ajuden a fer el canvi d'escena més dinàmic.

9.9 Control de càmera

A l'hora de desenvolupar el joc vam trobar que és més fàcil gestionar únicament els elements propis d'escena en comptes dels propis d'escena més els que venen d'altres escenes. D'objectes que s'han de gestionar a través de les escenes hi ha un par, però no és el comú. Les càmeres, igual que el jugador, són objectes propis d'escena. Per tant, cada que es carrega una nova escena, la càmera de l'escena ha de poder localitzar la instància de jugador principal en l'escena, enfocar-lo i seguir-lo. La implementació de trobar la instància del jugador en l'escena està en el mètode Start del script CameraManager.

Aquest component, a més a més, fa públic una sèrie de mètodes que permeten fer un efecte de tremolor. Aquests mètodes són utilitzats, per exemple, quan el jugador pren mal o quan un Boss fa un gran salt i aterra. A continuació veurem alguns dels mètodes més interessants d'aquesta classe.

La següent implementació permet assignar l'objecte que ha de seguir la camera, en aquest cas és el personatge principal de l'escena.

```
private void FollowPlayer()
{
    virtualCamera.Follow = player.transform;
}
```

Per poder fer l'efecte tremolor, es pot cridar a qualsevol dels següents mètodes.

```

public void ShakeCamera()
{
    ShakeCamera(
        intensity,
        frequency,
        duration,
        false);
}

public void ShakeCamera(
            float intensity = 1f,
            float frequency = 1f,
            float duration = 1.0f,
            bool reset = false)
{
    if (reset) Reset();

    if (Shaking)
        return;
```

```
21
22      Shaking = true;
23      virtualCameraNoise
24          .m_AmplitudeGain = intensity;
25      virtualCameraNoise
26          .m_FrequencyGain = frequency;
27
28      shakeTween = DOVirtual
29          .DelayedCall(duration, Reset);
30  }
```

9.10 Tilemap

A Unity, el component Tilemap és un sistema que emmagatzema i gestiona peces de mosaic per crear nivells 2D. Gràcies a altres components relacionats, com el Tilemap Renderer i el Tilemap Collider 2D, la seva funcionalitat pot ser molt diversa. Com ja s'explica a la Secció 7.5.22.8, aquest paquet no s'inclou a la instal·lació predeterminada d'Unity. [Unity 022g]

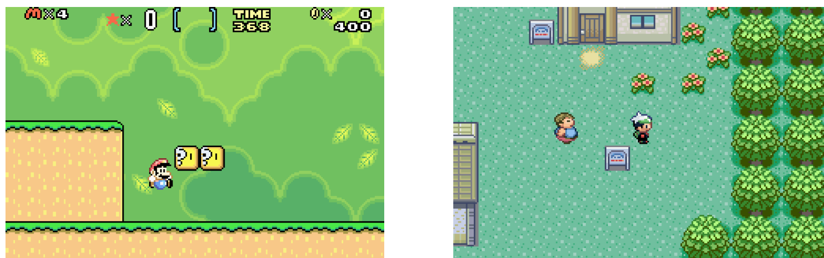

Figura 9.10: Videojocs famosos que utilitzen el disseny per tilemaps.

Hi ha molts de videojocs que estan construïts a través de tilemaps. Com podem veure a la Figura 9.10, jocs dins de les franquícies de Pokémon i Mario es troben dintre d'aquest conjunt.

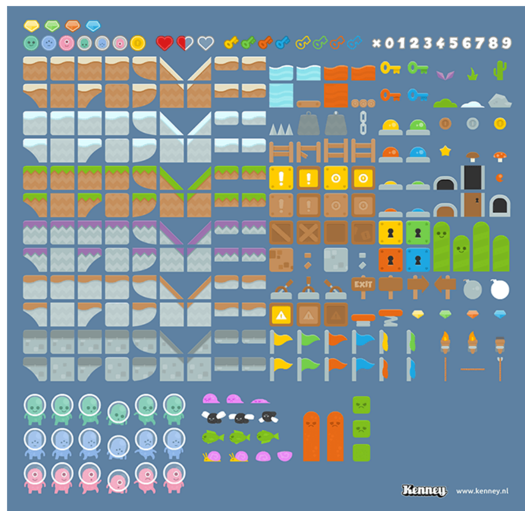

Figura 9.11: Peces de mosaic que hem utilitzat dels paquets de Kenney Assets.

Les peces de mosaic utilitzades en el projecte les hem tret de Kenney Assets: un estudi que ofereix, de forma gratuïta, recursos pel desenvolupament de vi-

deojocs [Kenney 022].

Figura 9.12: Utilització de les peces de Kenney Assets per crear escenaris diversos.

Principalment, nosaltres hem utilitzat els tilemaps de tres formes diferents:

1. **Tilemaps que construeixen l'escena**
   Una forma fàcil de construir el món a través d'aquestes peces de mosaic és afegint el component Tilemap Collider 2D als tilemaps. Aquest component fa que el tilemap reaccioni a la fisica del joc i, per conseqüent, que el jugador pugui col·lisionar amb el mateix. Això, el que fa, és convertir-lo en una eina pel desenvolupament d'escenaris. En el nostre projecte tenim un tilemap principal que utilitza aquesta tècnica per construir el món al mateix temps que decora l'escena.

Figura 9.13: Escena amb sols el base tilemap actiu.

2. **Tilemaps decoratius**
   Aquests tilemaps només tenen una funció: l'estètica, il·lustrant les diferents parts del joc. Entre ells hem utilitzat tilmaps que dibuixen les onades del fons de l'escena, tilmaps que componen el fons de l'escenari i tilemaps que contenen objectes decoratius.

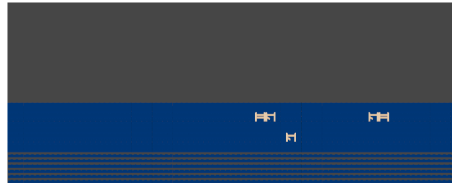

Figura 9.14: Exemple de tilemap decoratiu.

3. **Tilemaps que infligeixen mal al jugador**
   Com els tilemaps que construeixen l'escena, aquests també tenen el component Tilemap Collider 2D. Tanmateix, aquests el tenen activat com a trigger únicament per ser notificats quan el jugador entri en contacte amb ells. Al afegir-li el component Deadly Object, que el podem veure a la Secció 9.2.2, un cop el jugador entri en contacte amb el tilemap, aquest li infligira dany i, si el jugador no s'ha mort, el transportarà a l'últim punt de control.

Figura 9.15: Escena amb sols el tilemaps de dany actius.

Com ja hem anat veient a les figures 9.13, 9.14 i 9.15, tenim molts de tilemaps de l'escena que, junts, componen l'escenari 9.16.

Figura 9.16: Escena amb tots els tilemaps actius.

9.11 Contigut d'escena per capes

Unity ordena els renders segons un ordre de prioritat que depèn dels tipus i usos. Es pot especificar l'ordre de renderització a través de la cua de renderització. Els renderitzadors 2D inclouen el Sprite Renderer, Tilemap Renderer i tipus de Sprite Shape Renders. Les Sorting Layers es poden modificar i així s'adapten a les necessitats de cada projecte [Unity 022a].

Unity té els paràmetres Sorting Layer i Order in layer en els components Sprite Renderer, Tilemap Renderer i Sprite Shape Renders que permeten definir a quina capa i en quin ordre es renderitzen. Aquesta propietat ens és útil per ordenar els elements visuals dins de l'escena.

En treballar amb un joc amb estètica flat ens trobàvem que, a mesura que el projecte avançava, cada cop teníem més objectes visuals a l'escena que necessitaven estar en un ordre de renderització determinat. Amb l'augment d'aquests també augmentava la dificultat en ordenar-los, així que vam desenvolupar un sistema d'ordenació per capes.

Per aconseguir una bona organització vam utilitzar:

1. **Una bona estructura de capes.**
   Unity ens deixa definir totes les capes de renderització al nostre gust. Tanmateix, això no sempre és bo. Al final, tenir massa capes pot portar a mal de caps, així que nosaltres vam establir-ne vuit:

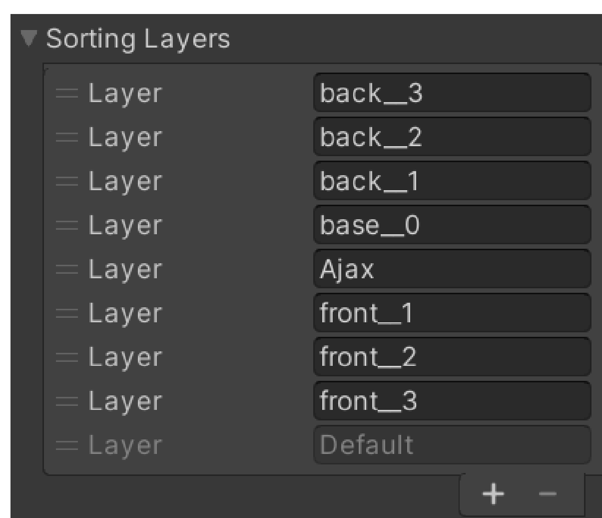

Figura 9.17: Les capes definides en el projecte

Com podem veure a la Figura 9.17, l'estructura s'inspira en la perspectiva clàssica:

- **Back 3:** S'encarrega del fons i d'elements llunyans. El tilemap que il·lustra l'aigua llunyana es troba en aquesta capa.
- **Back 2:** Tots els elements decoratius per sobre del fons es reuneixen en aquesta capa. Els tilemaps que il·lustren l'escenografia de l'illa es troben en aquesta capa.
- **Back 1:** Els elements interactuables com palanques, cofres, vides, entre d'altres, es reuneixen en aquesta capa.
- **Base 0:** Aquesta capa conté els enemics i els bosses.
- **Ajax:** Aquesta capa s'ocupa exclusivament del player.
- **Front 1:** Capa que reuneix els tilemaps de l'aigua i la lava.
- **Front 2:** Aquesta capa que reuneix les estructures bàsiques del joc. El tilemap base es troba en aquesta capa.
- **Front 3:** Reuneix els elements decoratius que queden per sobre de la resta per donar una sensació de profunditat.

2. **El component Sorting Group**
   Aquest component fa que, tant l'objecte pare, com tots els objectes fills, estiguin a la capa de renderització que s'ha assignat al sorting group. Així doncs, per acabar l'ordenació només vam crear l'objecte que es veu a la Figura 9.18. Cada un d'aquests objectes té el component Sorting Group, així que els seus fills compartiran la mateixa capa. Així doncs, vam posar a cada lloc els elements que pertanyien a cada sorting layer.

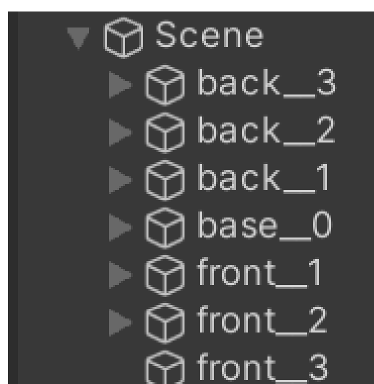

Figura 9.18: Sistema d'ordenació dels objectes visuals de l'escena segons la profunditat desitjada.

9.12 Il·luminació d'escenes

Durant la implementació i la creació de les escenes, ens vam adonar que el joc tenia un problema: el contrast. La paleta de colors del joc era molt semblant entre tots els objectes. Això causava diversos problemes de jugabilitat. Entre d'altres:

- El personatge era difícil de seguir durant el gameplay.
- Els objectes amb valor pel jugador, com vides, o els objectes interactuables, com palanques i tresors, es perdien al mig de la decoració.
- Els enemics eren difícils de veure i es confonien amb la vegetació, tot i que es trobaven en moviment.

A l'estar en una fase força avançada del desenvolupament, no ens podíem dedicar a canviar tot l'art que componia el joc així que vam demanar ajuda a Francesc Xavier Costa Brugue, professor de Disseny 2D i 3D en el grau de videojocs de la Universitat de Girona. Fransesc ens va confirmar que el problema del joc era causat pels contrastos i ens va donar diversos consells per millorar aquest aspecte de la jugabilitat:

- **Afegir ombra als sprites importants**
  Una ombra molt difuminada al contorn de personatges i enemics ajuda visualment a distingir els diferents objectes encara que els colors siguin semblants. Veure Figura 9.19.

Figura 9.19: Ombra blanquinosa envolta al personatge principal

- **Desenfoc progressiu**
  Desenfocar de forma progressiva, i en funció de la profunditat, les diferents capes de l'escenari, produeix un efecte de perspectiva que ajuda a diferenciar els personatges i objectes del fons. Veure Figura 9.20.

Figura 9.20: Fons desenfocat per ressaltar la resta de l'escena

- **Degradats**
  Els degradats aplicats a les formes, de manera molt subtil, també ajuden que es produeixi una distinció entre els objectes de l'escena. Veure Figura 9.21.

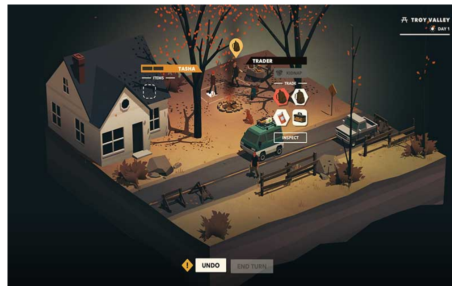

Figura 9.21: Formes amb degradats

Davant de les diferents opcions la que s'adaptava més a les necessitats del joc era la primera, veure Figura 9.19. Tot i això, igualment sorgia una qüestió: com implementar aquests canvis de forma fàcil i ràpida?

La solució se'ns va aparèixer en forma de paquet: Lightweight Render Pipeline és un pipeline de renderització creada per Unity. Aquest paquet incorpora la funcionalitat d'afegir efectes lluminosos al projecte amb una renderització ràpida i una alta qualitat. LWRP utilitza il·luminació i materials simplificats i basats físicament [Unity 022e].

9.12.1 Instal·lació i preparació

Aquest paquet es pot instal·lar des de l'administrador de paquets d'Unity. Per utilitzar aquest paquet és necessari modificar les configuracions del projecte d'Unity, crear piplines de renderització i canviar els materials utilitzats en el projecte. Els passos a seguir van ser els següents:

1. **Crear una pipeline de renderització**
   Per què els renders treballin amb llums, hem de definir una Pipleine de renderització especialitzada a les configuracions del projecte d'Unity. Per crear-ne una s'han de seguir els passos de la Figura 9.22.

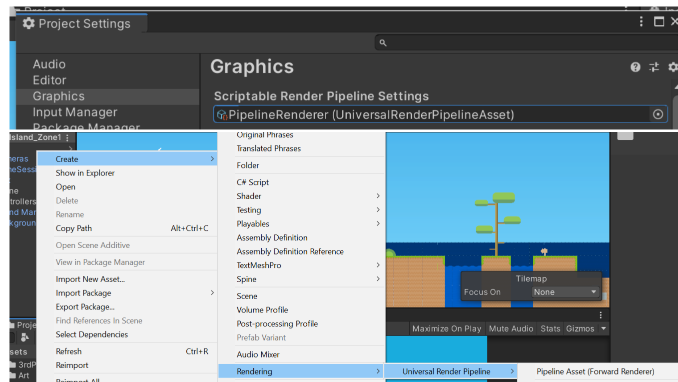

Figura 9.22: Crear una pipleline i assignar-la a les configuracions de gràfics d'Unity.

2. **Fer que la pipeline treballi amb 2D**
   Per fer que la pipline treballi en el pla 2D, hem de crear un render 2D i assignar-lo a la pipeline anteriorment creada. Veure Figura 9.23.

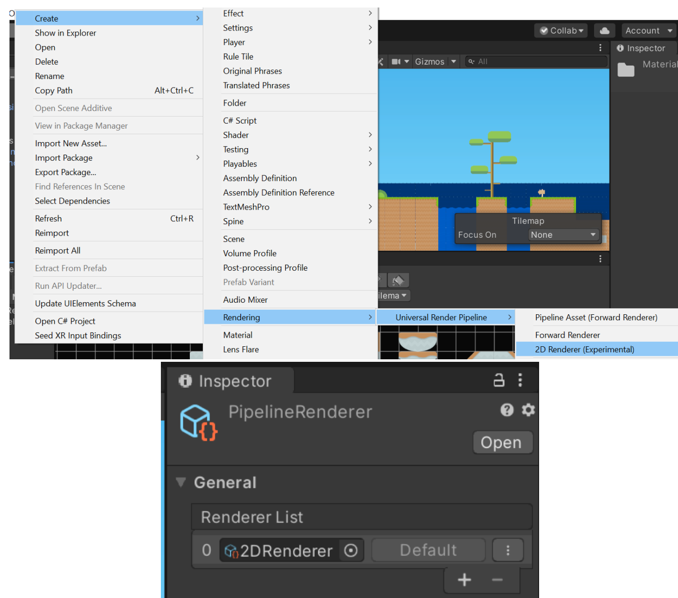

Figura 9.23: Creació i assignació d'un render 2D a una pipline.

3. **Canviar el material dels sprites**
   Les llums només reaccionen a un tipus de materials predeterminats, ai-

xí que per veure els canvis s'han de canviar els materials dels sprites. El paquet Lights 2D ve amb una funcionalitat específica per canviar tots els materials dels sprites automàticament. Veure Figura 9.24.

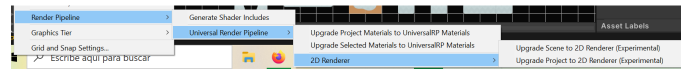

Figura 9.24: Canvi de material automàtic per l'escena o projecte.

4. **Llibreries de spine per Lights 2D**
   Spine fa servir els seus propis materials i són diferents dels utilitzats a Unity i als del paquet Lightweight RP. Spine té una extensió en el seu programari que dona suport al paquet que s'ha afegit al projecte per poder treballar amb animacions afectades per la il·luminació a l'escena

9.12.2 Tipus d'il·luminació.

Com expliquem a l'Apartat 7.5.20, hi ha 4 tipus de llums que ens ofereix aquest paquet:

- **Global Light**
  Il·lumina tota l'escena per igual.

- **Point Light**
  Un punt de llum.

- **Free form light**
  Llum que agafa la forma que es desitgi.

- **Sprite light**
  La forma és causada pel tipus de sprite que se li assigni. No s'ha utilitzat en el projecte.

Les llums es poden personalitzar en diversos aspectes. Els que nosaltres hem utilitzat més:

- **Color:** Tonalitat.

- **Intensitat:** Quina quantitat d'il·luminació desprèn.

- **Capes afectades:** A quines capes afecta la llum.

9.12.3 Gestió de l'il·luminació

Com ja s'ha explicat a la Secció 9.11, el projecte utilitza una metòdica ordenació de les capes. Això ens ha ajudat moltíssim a fer ús les llums, ja que de formes molt fàcils hem pogut fer que cada llum afectés només a les capes que nosaltres desitjàvem, mantenint una estructura molt clara al projecte.

Llums utilitzades a les escenes:

- **Global Light**
  Produeix la mínima il·luminació que nosaltres esperem de totes les capes. Veure figura 9.25.

Figura 9.25: Diferents intensitats a la Global Light en una escena.

- **Free form light bàsica**
  Aquesta llum ocupa tota l'escena i il·lumina totes les capes que s'han de veure bé: Back 3, Back 1, Base 0, Front 1 i Front 2. Aquesta llum il·luminarà bé el fons llunyà, els enemics i els objectes interactuables, al mateix temps que alguns dels objectes que quedin per sobre de l'Ajax com el terreny base, l'aigua i la lava. Veure figura 9.26.

Figura 9.26: Free form light bàsica en escena.

- **Free form light per coves**
  S 'ocupa de fer la distinció de coves respecte als espais oberts. Actuen

sobre les capes Back 2 i Ajax: Les capes del fons i del player. Veure figura 9.27.

Figura 9.27: Free form light per donar profunditat a les coves.

- **Point Lights i Freeform lights pel player, enemics i NPCs**
  Aquestes llums s'utilitzen per donar importància i visibilitat a aquests objectes. Veure figura 9.28.

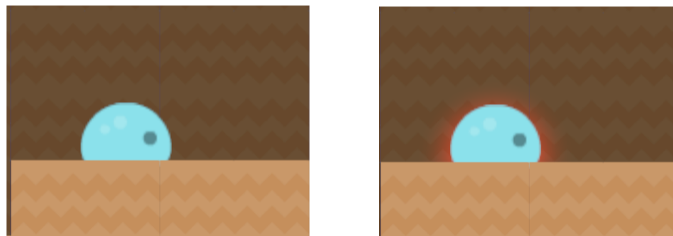

Figura 9.28: Enemic amb i sense efecte lluminós.

- **Free form lights i point lights per la lava i les torxes**
  Llums utilitzades per enriquir els elements decoratius de l'escena. Veure figura 9.29.

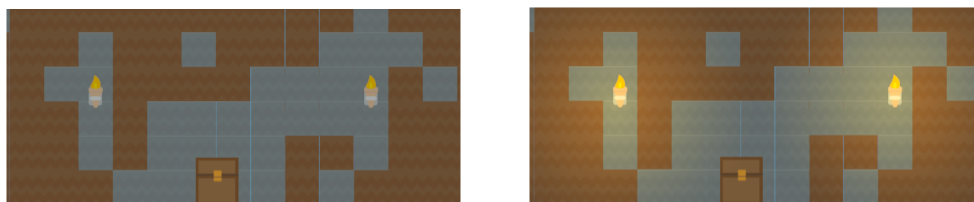

Figura 9.29: Objecte decoratiu amb i sense efecte lluminós

9.13 Control de música i efectes sonors

El control de la música i efectes sonors en un videojoc es escencial. Per això vam crear dos components, un capaç de gestionar la música i soroll de fons en les escenes, i un altre que gestiona la reproducció d'efectes sonors d'escena.

9.13.1 Component BackgroundMusicPlayer

Aquest component està pensat per gestionar la música que el jugador escolta i també el soroll de fons, és a dir, sorolls de cova, mar, etcètera.

La presència d'aquest component en les escenes està estratègicament pensada per adaptar la música amb l'ambient visual de l'escena. En entrar a escenes de menys llum, s'utilitza música més lenta i trista, en entrar en zones amb més acció es reprodueix música més dinàmica, etcètera. Recomanem veure el diagrama d'activitats de la Figura 8.11 per entendre com es gestiona l'actualització de música en canviar d'escena.

Pel que fa al codi, gran part de la lògica que s'utilitza d'aquest component està en els mètodes Update i Awake. A continuació veurem la implementació d'aquest mètode. Aquest component gestiona dos AudioSources diferents, un encarregat per la música principal i l'altre per la gestió de soroll de fons. Aquest dos AudioSource són les variables de classe m_MusicAudioSource i m_AmbientAudioSource i, com podem veure dintre del mètode Awake, es fa la seva inicialització.

```
1  void Awake()
2  {
3      s_Instance = this;
4
5      DontDestroyOnLoad(gameObject);
6
7      m_MusicAudioSource = gameObject
8          .AddComponent<AudioSource>();
9      m_MusicAudioSource.clip = musicAudioClip;
10     m_MusicAudioSource.outputAudioMixerGroup
11         = musicOutput;
12     m_MusicAudioSource.loop = true;
13     m_MusicAudioSource.volume
14                 = musicVolume;
15
16     if (musicPlayOnAwake)
17     {
18         m_MusicAudioSource.time = 0f;
19         m_MusicAudioSource.Play();
```

```
20      }
21
22      m_AmbientAudioSource = gameObject
23          .AddComponent<AudioSource>();
24      m_AmbientAudioSource.clip = ambientAudioClip;
25      m_AmbientAudioSource.outputAudioMixerGroup
26          = ambientOutput;
27      m_AmbientAudioSource.loop = true;
28      m_AmbientAudioSource.volume
29          = ambientVolume;
30
31      if (ambientPlayOnAwake)
32      {
33          m_AmbientAudioSource.time = 0f;
34          m_AmbientAudioSource.Play();
35      }
36  }
```

Com sabem, la música del joc és altament configurable. La manera en què la intensitat de la música del joc s'actualitza amb els canvis fets en la interfície d'usuari de configuració és pel mètode Update d'aquest component. En cada fotograma s'ajusta el volum del joc amb els mètodes ComputeAmbientVolume i ComputeMusicVolume amb l'ajuda del recurs compartit playerVolumeSettings.

```
1
2   private void Update()
3   {
4       m_MusicAudioSource.volume =
5           ComputeMusicVolume();
6       m_AmbientAudioSource.volume =
7           ComputeAmbientVolume();
8   }
9
10  private float ComputeMusicVolume()
11  {
12      return playerVolumeSettings
13          ?
14          playerVolumeSettings.MusicVolume
15          * musicVolume
16          :
17          musicVolume;
18  }
19
20  private float ComputeAmbientVolume()
21  {
22      return playerVolumeSettings
23      ?
24      playerVolumeSettings.SoundVolume
```

```
    * ambientVolume
    :
    ambientVolume;
}
```

Cal dir que aquest script no és original de nosaltres, sinó que pertany al paquet d'aprenentatge d'Unity 2DGameKit [Unity 022h], però li hem fet certes modificacions perquè s'adaptés a les necessitats del projecte, com ara, que tingui suport i modifiqui la intensitat de la música que es reprodueix amb les configuracions fetes pel jugador en el menú de volum.

9.13.2 Component SoundManager

Aquest component encapsula la lògica d'efectes sonors que es reprodueixen en moments puntuals al llarg de l'escena, com ara l'efecte que es reprodueix quan Ajax pica o salta, o quan algun NPC parla.

```
using Core.Utility;
using UnityEngine;
using Random = UnityEngine.Random;

namespace Core.Manager
{
    public class SoundManager : MonoBehaviour
    {
        public void PlaySound(
            AudioClip clip,
            float volume = 1.0f,
            AudioSource src = null,
            bool randomizePitch = false)
        {
            float pitch =
                randomizePitch
                ?
                Random.Range(
                    lowPitchRange,
                    highPitchRange)
                :
                1.0f;
            PlaySoundWithPitch(clip, pitch, volume, src);
        }

        public void PlayRandomSound(
            AudioClip[] clips,
            float volume = 1.0f,
            AudioSource src = null)
        {
            if (clips.Length == 0)
```

```
32                  return;
33 
34              AudioSource source = src ?? audioSource;
35              source.Stop();
36 
37              int randomIndex = Random
38                                   .Range(0, clips.Length);
39 
40              source.pitch = Random
41                              .Range(
42                                   lowPitchRange,
43                                   highPitchRange);
44              source.PlayOneShot(clips[randomIndex], volume)
    ;
45          }
46  }
```

Hem exposat alguns dels mètodes més interessants que el component suporta, els mètodes PlaySound i PlayRandomSound. L'única diferència entre aquests dos mètodes és que l'últim rep una llista de possibles sons a reproduir i agafa un d'aquests aleatòriament, mentre que en l'altre mètode s'ha de fer explicit el so a reproduir.

9.13.3 Component UIVolumeSettings

El component UIVolumeSettings gestiona el volum de la música. Escolta els diferents sliders que serveixen per apujar o abaixar els valors del volum i fa les actualitzacions pertinents en el ScriptableObject VolumeSettings que emmagatzema aquests valors.

```
1      private void Update()
2      {
3          UpdateVolumeSettings();
4      }
5  
6      private void UpdateVolumeSettings()
7      {
8          volumeSettings.MusicVolume = musicSlider.value;
9          volumeSettings.SoundVolume = soundSlider.value;
10     }
```

9.14 Gestió de guardat i càrrega de dades

Com que un dels requisits funcionals del projecte és la gestió del manteniment d'una sessió de joc, vam haver d'implementar la lògica necessària per poder persistir dades i recuperar-les. Tot i que les dades generades dintre del joc no són dades delicades, vam decidir que protegiríem les dades aplicant l'algoritme d'encriptació simètric Aes [TechTarget 022] .

Per persistir les dades, primer capturem l'estat del personatge, després el desserialitzem, és a dir, transformem els valors en tipus de dades primitives, xifrem les primitives i guardem el resultat en un fitxer amb extensió bin dintre d'un dels directoris que gestiona Unity. Per recuperar les dades, es fa la mateixa seqüència a l'inrevés, recuperem el fitxer, dexifrem els valors que hi ha dintre del fitxer, ho convertim a primitives i finalment recuperem l'estat.

L'implementació de la lògica per persistir dades encriptades es la seguent:

```
1  public static void
2            SavePlayerState(PlayerState playerData)
3 {
4     string path
5         = dataPath + playerStateFileName;
6     Encrypt(path, playerData);
7 }
```

L'implementació que ens permet recuperar les dades es la seguent:

```
1 public static PlayerState LoadPlayerState()
2 {
3     string path =
4     Application.persistentDataPath
5     +
6     "/player_stats.bin";
7     if (File.Exists(path))
8     {
9         try
10        {
11            PlayerState playerData
12            = (PlayerState)Decrypt(path);
13        }
14        catch (Exception e)
15        {
16            Debug
17            .LogError(
18                "Error loading player state: "
19                    + e.Message);
20            return PlayerStateDefaultValues();
21        }
```

```
22          return (PlayerState)Decrypt(path);
23      }
24      else
25      {
26          return PlayerStateDefaultValues();
27      }
28  }
```

9.15 Plataformes i portes dinàmiques

Hem desenvolupat diverses formes de moure objectes a l'escena i ho hem utilitzat per a les plataformes i les portes del joc.

9.15.1 Plataformes bàsiques

Hem dissenyat una plataforma que a l'entrar en contacte amb el jugador es mogui fins a una posició establerta.

Per això hem necessitat dos components: El que s'ocupa de moure la plataforma; el "MoveStructure", i el que s'ocupa de fer la crida quan el jugador es trobi sobre la plataforma; "MoveStructureCaller".

La funció Move() assigna la posició on ha d'arribar l'estructura. A partir d'aquí, cada cop que s'executa l'update l'estructura es mou cap a la nova posició. Un cop a la nova posició, si l'estructura té per defecte tornar enrere, es tornarà a assignar, al cap d'uns segons, la posició inicial com nou punt a arribar.

```
void FixedUpdate()
{
    if (target)
    {
        var currentPos = structure.transform.position;
        structure.transform.position =
        Vector2.MoveTowards(currentPos,
        target.position,
        speed * Time.deltaTime);

        if (structure.transform.position ==
        pointB.position && goBackwards)
        {
            StartCoroutine(GoBack(this.waitTime));
            target = null;
            //seting target to null so this
            //if it don't executes every frame
        }
    }

}

public IEnumerator GoBack(float time)
{
    yield return new WaitForSeconds(time);
    target = pointA;
}
```

```
28
29 public void Move()
30 {
31     if (structure.transform.position ==
32     pointA.position)
33     {
34         target = pointB;
35     }
36     else if (structure.transform.position ==
37     pointB.position)
38     {
39         target = pointA;
40     }
41
42 }
```

Com podem veure en el codi, la funció Move() és cridada quan el player col·lisiona amb el trigger de la plataforma. Veure trigger a la Figura 9.30.

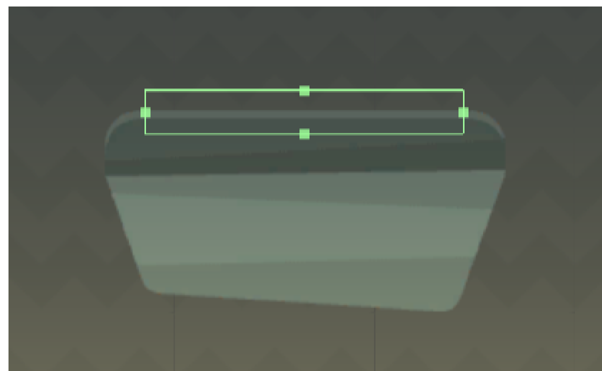

Figura 9.30: Trigger que activa el moviment de la plataforma

A causa de la implementació, mai s'assignarà una nova posició a arribar mentre la plataforma s'estigui movent. Això està fet per ser simple i clar en el moment de jugar.

```
1 public class MoveStructureCaller : MonoBehaviour
2 {
3     [SerializeField] MoveStructure moveStructure;
4
5     //pre: --
6     //post: Is collider is player, it calls the method
    Move of moveStructure.
7     private void OnTriggerEnter2D(Collider2D other)
8     {
9         if (other.gameObject.tag == "Player")
10        {
11            moveStructure.Move(); //Activation of Mecanism
```

```
12         }
13     }
14 
15 }
```

9.15.2 Plataformes que s'activen amb palanques

Utilitzant el funcionament mencionat a la Secció 9.15.1, també s'hi ha d'afegir que les estructures es puguin moure per mitjà d'un interruptor o palanca.

El component MoveStructureToPoint s'ocupa de desactivar el component MoveStructure i de moure la plataforma fins al punt desitjat. El component MoveStructureToPointCaller s'ocupa de fer la crida quan el jugador ha interactuat amb la palanca o l'interruptor.

```
1  public class MoveStructureToPoint : MonoBehaviour
2  {
3      [SerializeField] GameObject structure;
4      [SerializeField] float speed;
5      MoveStructure moveStructureEngine;
6      Transform target;
7  
8      //pre: --
9      //post: if target position is not null, every frame
   structure moves to it.
10     //      if has arribed, we set the target to none and
   if the object has a MoveStructureEngine
11     //      we enable it
12     private void FixedUpdate()
13     {
14         if (target)
15         {
16             var currentPos = structure.transform.position;
17 
18             structure.transform.position = Vector2.
   MoveTowards(currentPos, target.position, speed * Time.
   deltaTime);
19 
20             if (structure.transform.position == target.
   position)
21             {
22                 target = null;
23                 if (moveStructureEngine)
24                 {
25                     moveStructureEngine.EnableMove(true);
26                     moveStructureEngine = null;
27                 }
```

```
28
29                  }
30              }
31          }
32
33          //pre: --
34          //post: target is uptaed with new point
35          //      and if game object has a MoveStructureEngine,
        we disable it.
36          public void Activate(Transform targetPoint)
37          {
38              this.target = targetPoint;
39
40              moveStructureEngine = GetComponentInParent<
        MoveStructure>();
41              if (moveStructureEngine)
42              {
43                  moveStructureEngine.EnableMove(false);
44              }
45
46          }
47  }
```

Hi ha dos tipus d'accionadors: les palanques i els interruptors. Les palanques serveixen per a aquelles estructures que només fan una translació, com les portes. Aquestes només s'han d'activar un cop. Els interruptors serveixen per estructures com les plataformes, que poden ser activades diverses vegades.

```
1 public enum LeverType
2 {
3     Button, Handler
4 }
```

Aquests tenen petits comportaments diferents per donar més realisme a les funcions. Si és de tipus palanca, un cop activat ja no es torna la posició inicial, en canvi, aquí podem veure com l'interruptor sí que hi torna:

```
1         else if (activated && IsStructureOnPoint())
2         {
3             if (type == LeverType.Handler)
4             {
5                 ChangeSprite();
6             }
7             activated = false;
8         }
```

El mètode ChageSprite() canvia el sprite de la palanca a activat si està desactivat o viceversa. //

En el següent tall de codi podem veure com si l'accionador és de tipus interruptor s'activa i es desactiva quan l'estructura ja es troba en el lloc.

```
        if (!IsStructureOnPoint())
        {
            ChangeSprite();
            moveStructureToPointEngine.Activate(myPoint);
            activated = true;
        }
        else if (type == LeverType.Handler)
        {
            StartCoroutine(IActivateAndDesactivateHandler
());
        }
```

La corutina IActivateAndDesactivateHandler(), canvia el sprite, espera un segons i el torna a canviar per fer un efecte d'activació momentània.

```
    IEnumerator IActivateAndDesactivateHandler()
    {
        ChangeSprite();
        yield return new WaitForSeconds(waitTime);
        ChangeSprite();
    }
```

El component MoveStructureToPointCaller té la posició a on ha d'anar l'estructura, així que, en cridar el mètode Activate() de MoveStructureToPoint li passarà per paràmetre.

```
    void Update()
    {
        if (playerIn && Input.GetButtonDown(
CharacterActions.Interact))
        {
            MoveStructure();
        }
        else if (activated && IsStructureOnPoint())
        {
            if (type == LeverType.Handler)
            {
                ChangeSprite();
            }
            activated = false;
        }
    }

    private void MoveStructure()
    {
        if (!activated)
        {
            SoundManager.Instance?.PlaySound(activateSound
, 0.5f, GetComponentInChildren<AudioSource>());
            if (!IsStructureOnPoint())
```

```
23             {
24                 ChangeSprite();
25                 moveStructureToPointEngine.Activate(
   myPoint);
26                 activated = true;
27             }
28             else if (type == LeverType.Handler)
29             {
30                 StartCoroutine(
   IActivateAndDesactivateHandler());
31             }
32         }
33     }
34 
35     private bool IsStructureOnPoint()
36     {
37         return structure.transform.position == myPoint.
   position;
38     }
```

9.16 Interfícies d'usuari

9.16.1 Menús

La mobilitat pels menús, com a tot el joc, també es fa per teclat. Les tecles W i S corresponen als moviments verticals, per moure's entre les opcions. Les tecles A i D corresponen als moviments horitzontals, per ajustar els volums i moure's entre les opcions. Finalment, utilitzem la tecla Space per a la selecció.

S'ha gestionat de la mateixa forma per tots els nostres menús:

- Menú d'inici. Component StartMenu.
- Menú de pausa. Component InGameMenu.
- Menú de configuracions. Component SettingsMenu.

Pel control de l'opció dins menú, en els scripts hi ha una variable anomenada option. A l'Update, aquesta variable s'actualitza quan l'usuari interacciona amb les tecles.

```
1     private void Update()
2     {
3         if (inSettingsPage)
4             return;
5 
6         ManageSelectGameMenu();
7         ManageOptions();
8     }
```

Com podem veure en aquest tall de codi del component StartMenu, es criden les funcions ManageSelectMenu() i ManageOptions() que controlen la gestió del menú.

```
1     private void ManageOptions()
2     {
3         if (option == 0)
4         { //Start
5             if (Input.GetKeyDown(KeyCode.S))
6             {
7                 PlayNavigationSound();
8                 OnStartGameHoverOut();
9                 OnSettingsHoverIn();
10                HideSelectGameMenu();
11                option = 1;
12            }
13            else if (Input.GetKeyDown(KeyCode.Space))
14            {
```

```
15                  PlaySelectSound();
16                  LoadGame();
17              }
18          }
19          else if (option == 1)
20          { // Settings
21              if (Input.GetKeyDown(KeyCode.S))
22              {
23                  PlayNavigationSound();
24                  OnSettingsHoverOut();
25                  OnQuitHoverIn();
26                  option = 2;
27              }
28              else if (Input.GetKeyDown(KeyCode.W))
29              {
30                  PlayNavigationSound();
31                  OnSettingsHoverOut();
32                  OnStartGameHoverIn();
33                  option = 0;
34              }
35              else if (Input.GetKeyDown(KeyCode.Space))
36              {
37                  //Scene Manager
38                  PlaySelectSound();
39                  OpenSettingsPage();
40              }
41          }
42          else
43          { //Quit
44              if (Input.GetKeyDown(KeyCode.W))
45              {
46                  PlayNavigationSound();
47                  OnQuitHoverOut();
48                  OnSettingsHoverIn();
49                  option = 1;
50              }
51              else if (Input.GetKeyDown(KeyCode.Space))
52              {
53                  PlaySelectSound();
54                  Application.Quit();
55              }
56          }
57      }
```

En aquest tall de codi podem veure com s'actualitza la variable option depenent de les interaccions de l'usuari.

Per indicar a l'usuari quina opció es troba dins del menú, hem utilitzat, per una banda, els sons i, per l'altra banda, dues tècniques visuals depenent del

menú:

- Menú d'inici i menú de pausa: Aquests dos menús estan totalment animats amb Spine. Depenent de quina és l'opció seleccionada l'animació agafa una o altre forma [Leak 022].

```
        PlayNavigationSound();
        OnQuitHoverOut();
        OnSettingsHoverIn();

```

  Exemple de com s'inicia una animació amb spine:

```
    public void OnQuitHoverOut()
    {
        skeletonGraphic.AnimationState
            .AddAnimation(1, "hoverOutQuit",
            false, 0);
    }

```

- Menú de configuracions: Aquest menú ha estat programat perquè en canviar d'opció hi hagi un element que ressalti el canvi: una imatge al costat de les opcions. Aquesta canvia de posició depenent de quina opció ens trobem.

```
        MoveSelector(soundGO.transform.position.y);

```

```
    private void MoveSelector(float yPos)
    {
        PlayNavigationSound();
        selector.transform.DOMoveY(yPos, 0.2f)
        .SetUpdate(true);
    }

```

9.16.1.1 Gestió de menu de pausa

El component InGameMenuController és l'encarregat de gestionar el menú de pausa durant el gameplay. El mètode Update és el que s'encarrega de gestionar què passa quan es prem la tecla d'activar o desactivar el menú.

```
    private void Update()
    {
        if (Input.GetKeyDown(KeyCode.Tab)
        && !inSettingsPage)
        {
            if (gamePaused)
```

```
            {
                ResumeGame();
            }
            else
            {
                PauseGame();
            }
        }
    }
```

Podem veure, en els mètodes ResumeGame i PauseGame, com s'activa i es desactiva el menú, el transcurs del temps en el joc i el control del jugador.

```
    private void ResumeGame()
    {
        CloseMenu();
        Time.timeScale = 1;
        gamePaused = false;
        StartCoroutine(EnablePlayer());
        pauseMenu.SetActive(false);
    }

    private void PauseGame()
    {
        pauseMenu.SetActive(true);
        OpenMenu();

        var player = PlayerController.Instance;
        player.BlockingUI = true;

        Time.timeScale = 0;
        gamePaused = true;
    }
```

9.16.1.2 Opcions menú d'inici

Les opcions que ens ofereix el menú d'inici són les següents:

1. **Iniciar la partida**
   A l'escollir l'opció d'iniciar partida es crida el mètode LoadGame().

```
    private void LoadGame()
    {
        if (!SaveSystem.SaveGameExists())
        {
            LoadNewGame();
        }
        else
        {
```

```
            OpenSelectGameMenu();
        }
    }

```

Aquest mètode mira si hi ha una partida guardada. Si hi ha, crida el mètode OpenSelectGameMenu() que obre el submenú per escollir si es vol jugar a la partida actual o es vol començar una nova partida. El boolea inSelectGameMenu es posa a cert per activar el mètode ManageSelectGameMenu(). Aquest gestiona aquest el submenú. Com podem veure, aquest menú controla la variable optionSelectGameMenu que indica en quina opció ens trobem del submenú.

```
    private void ManageSelectGameMenu()
    {
        if (!inSelectGameMenu)
            return;

        if (optionSelectGameMenu == 0)
        { //current game
            if (Input.GetKeyDown(KeyCode.D))
            {
                optionSelectGameMenu = 1;
                PlayNavigationSound();
                MoveSelectGameSelector();
            }
            else if (Input.GetKeyDown(KeyCode.Space))
            {
                //PlaySelectSound();
                LoadCurrentGame();
            }
        }
        else
        { //op == 1 //new game
            if (Input.GetKeyDown(KeyCode.A))
            {
                optionSelectGameMenu = 0;
                PlayNavigationSound();
                MoveSelectGameSelector();
            }
            else if (Input.GetKeyDown(KeyCode.Space))
            {
                //PlaySelectSound();
                LoadNewGame();
            }
        }

    }

```

Si s'escull jugar a la partida actual, es crida el mètode LoadCurrentGame() i inicia una partida amb les dades guardades.

```
1     private void LoadCurrentGame()
2     {
3         PlayerState playerState =
4         SaveSystem.LoadPlayerState();
5         GameSessionController.Instance.LoadData =
6         true;
7         StartCoroutine(Loader.LoadWithDelay(
8         (SceneID)playerState.scene, 0));
9     }
10
```

Si s'escull començar una partida nova o si no hi havia cap partida guardada, es crida el mètode LoadNewGame() i inicia una partida nova.

```
1     private void LoadNewGame()
2     {
3         //Inicial save files creation
4         SaveSystem.InitializeGame();
5         PlayerState playerState =
6         SaveSystem.LoadPlayerState();
7         GameSessionController.Instance.LoadData =
8         true;
9         StartCoroutine(Loader.LoadWithDelay(
10        (SceneID)playerState.scene, 0));
11    }
12
```

2. **Anar al menu de configuracions**
   A l'escollir aquesta opció, es crida el mètode OpenSettingsPage() que treu el control al menú d'inici i obre el menú de configuracions indicant per paràmetre que ens trobem en una escena no jugable.

```
1     private void OpenSettingsPage()
2     {
3         inSettingsPage = true;
4         settingsMenu.SetActive(true);
5         settingsMenu.OnOpenMenu(false);
6     }
7
```

3. **Sortir del joc**
   A l'escollir aquesta opció es crida el mètode Application.Quit() que tanca el joc.

```
1     Application.Quit();
2
```

9.16.1.3 Opcions menu de pausa

Les opcions que ens ofereix el menú de pausa són les següents:

1. **Retornar al joc**
   Aquesta opció crida al mètode ResumeGame que ja hem vist al Subapartat 9.16.1.1.

2. **Anar al menú de configuracions**
   En escollir aquesta opció es crida el mètode OpenSettingsPage que treu el control al menú de pausa i obra el menú de configuracions indicant per paràmetre que trobem en una escena jugable.

```
1     private void OpenSettingsPage()
2     {
3         inSettingsPage = true;
4         settingsMenu.SetActive(true);
5         settingsMenu.OnOpenMenu(true);
6     }
7
```

3. **Sortir**
   Aquesta opció retorna al menú d'inici.

```
1         StartCoroutine(Loader.LoadWithDelay(
2         SceneID.StartMenu, 0));
3
```

9.16.1.4 Opcions menú de configuracions

Les opcions que ofereix el menú de configuracions són les següents:

1. **Ajustar so de la música i so dels efectes**
   Quan ens trobem en aquestes opcions, en tocar les tecles A i D, movem el slider. Amb la tecla A, el decrementem, i amb la tecla D, l'incrementem. Això ho gestiona el mètode ManageSlider(), que com podem veure, funciona pels dos sliders del menú.

```
1      private void ManageSlider(Slider slider, bool
    increase)
2     {
3         PlaySettingSound();
4         if (increase && slider.value < 1)
5         {
6             slider.value += slidersChangeValue;
7         }
8         else if (!increase && slider.value > 0)
9         {
```

```
10              slider.value -= slidersChangeValue;
11          }
12      }
13
```

A qui podem veure un exemple de crida pel slider de la música:

```
1           ManageSlider(
2           musicGO.GetComponentInChildren<Slider>(),
3           false);
4
```

El volum de la música s'actualitza mitjançant el component UIVolumeSettings, com s'explica a l'apartat 9.13.

2. **Veure tecles de jugabilitat**
   Aquesta opció obre una imatge amb les tecles que s'utilitzen per jugar al joc. Si ens trobem en partida, només mostrarà les habilitats adquirides, si ens trobem en el menú inicial, ho mostrarà tot.

```
1       private void ShowKeySheet()
2       {
3           PlaySelectSound();
4           if (inGame)
5           {
6               managePowersVisibility
7               .ManageAdquiredPowersVisibility();
8           }
9           else
10          {
11              managePowersVisibility
12              .ShowAllPowers();
13          }
14          optionsCanvasGroup.DOFade(0, 0.25f)
15              .SetUpdate(true);
16          keyCheatSheetCanvasGroup.DOFade(1,0.25f)
17              .SetUpdate(true)
18              .OnComplete(
19                  () =>
20                  {
21                      showingkeyCheatSheet = true;
22                  }
23              );
24      }
25
```

3. **Sortir del menú**
   Aquesta opció crida el mètode OnCloseMenu per tancar el menú de configuracions. El menú de configuracions el podem obrir des del menú d'inici

o des del menú de pausa. Això fa que hàgem de cridar a dos mètodes diferents perquè reprenguin el control. Ho gestionem de la següent manera:

```
    private void OnCloseMenu()
    {
        PlaySelectSound();
        optionsCanvasGroup.DOFade(0, 0.25f).SetUpdate(
   true).OnComplete(() =>
        {
            if (inGame)
            {
                GetComponentInParent
                <InGameMenu>()
                .CloseSettingsPage();
            }
            else
            {
                GetComponentInParent
                <StartMenu>()
                .CloseSettingsPage();
            }
        });
    }

```

9.16.2 In-Game Canvas

El Canves de dins del joc ha estat dissenyat per ser una forma fàcil d'informar a l'usuari de l'estat del jugador i dels events importants. Per això, la informació que es mostra és la següent:

- La vida del jugador
- L'estat de les habilitats del jugador
- Notificacions
- Indicacions per com actuar en situacions noves

9.16.2.1 Barra de vida

S'ha plantejat la barra de vida com una sèrie de contenidors amb líquid, com si fossin els diferents dipòsits d'un robot. Un contenidor ple significa una vida, un contenidor buit significa que li falta una vida i un contenidor trencat significa la mort.

La barra de vida no sols ha d'indicar quanta vida té el jugador, també ha d'emfatitzar quan es guanya o es perd una vida, quan el jugador està a punt de

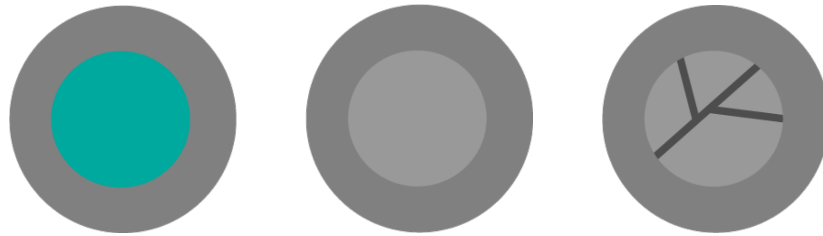

Figura 9.31: Sprites contenidors de vida.

morir, quan s'aconsegueix una vida addicional, etc. Per assolir-ho hem necessitat adjuntar una sèrie d'animacions i efectes de partícules.

La barra de vida està formada per dos components:

- **LifeBarController**
  Gestiona la barra de vida. S'ocupa de prendre i posar vides, afegir-ne i gestionar els diferents efectes.

- **LifeBarContainer**
  Gestiona el comportament d'una vida. S'ocupa d'animar la vida segons els seus canvis d'estat.

LIFE BAR CONTROLLER
Profunditzem en el funcionament del component LifeBarController. Com que les vides del jugador s'emmagatzemen en un ScriptableObject, els canvis no es notifiquen directament a la barra de vida, sinó que aquesta ha de romandre escoltant el ScriptableObject per saber quan s'ha produït un canvi. Quan es produeix un canvi, s'emmagatzema en format de tupla a una cua anomenada pendentChanges. La tupla conté dos elements, l'acció i, opcionalment, un número que pot indicar les vides a sumar o a restar. Les accions que pot sofrir la cua són els següents:

- **Setup**
  Borra la barra de vida actual i la configura des de zero.

- **GainLife**
  La barra de vida recupera un nombre de vides.

- **LoseLife**
  La barra de vida perd un nombre de vides.

- **AddNewLife**
  Afegeix una nova vida a la barra, recuperant totes les vides anteriors.

- **FillAll**
  Recupera totes les vides.

S'ha creat una classe enum per tenir-les registrades.

```
public enum LifeBarAction
{
    Setup,
    GainLife,
    LoseLife,
    AddNewLife,
    FillAll
}
```

Llavors, durant el mètode FixedUpdate(), s'avalua el ScriptableObject que conté les vides i s'emplena la cua de canvis en conseqüència.

```
void FixedUpdate()
{
    if (!modifying)
    {
        /*Update Queue*/
        if (pendentChanges.Count != 0){
            StartCoroutine(ManageQueue());
        } else if (currentLifes > 0) {
        /*Control changes*/
            if (totalLifes != playerHealth.MaxHP) {
                SetUpNewLife();
            }
            else if (currentLifes > playerHealth.HP) {
                LoseLifes(currentLifes - playerHealth.HP);
            }
            else if (currentLifes < playerHealth.HP
            && playerHealth.HP != playerHealth.MaxHP ) {
                GainLifes(playerHealth.HP - currentLifes);
            }
            if (currentLifes < playerHealth.HP
            && playerHealth.HP == playerHealth.MaxHP) {
                HealAllLifes();
            }
        }
    }
}
```

Tots els mètodes que modifiquen la cua comparteixen el mateix format. Mirem el següent exemple del mètode GainLifes():

```
private void GainLifes(int lifesUp)
{
    pendentChanges.Enqueue((LifeBarAction.GainLife,
    lifesUp));
}
```

Quan la cua de canvis no està buida s'executa el mètode ManageQueue() que aplica els canvis dictats per la cua.

```
private IEnumerator ManageQueue()
{
    if (modifying || pendentChanges.Count == 0)
        yield break;
    else modifying = true;
    (LifeBarAction, int) actionType =
    pendentChanges.Dequeue();

    switch (actionType.Item1)
    {
        case LifeBarAction.Setup:
            yield return StartCoroutine(
            SetUpLifesProcess(actionType.Item2));
            break;
        case LifeBarAction.GainLife:
            yield return StartCoroutine(
            GainLifesProcess(actionType.Item2));
            break;
        case LifeBarAction.LoseLife:
            yield return StartCoroutine(
            LoseLifesProcess(actionType.Item2));
            break;
        case LifeBarAction.FillAll:
            yield return StartCoroutine(
            FillAllLifes());
            break;
        case LifeBarAction.AddNewLife:
            yield return StartCoroutine(
            AddNewLife());
            break;
        default:
            yield return null;
            break;
    }

    modifying = currentLifes == 0;
}
```

Com podem apreciar, tots els mètodes que editen la barra de vida són corutines. Això es a causa del fet necessitem que les accions no se superposin les unes amb les altres. Tenim la variable booleana modifiying, que si està a cert no deixa fer modificacions. Gràcies a les corutines podem fer que aquesta variable no es posi a falç fins que la barra de vida hagi sofert tot el procés de canvi.

Configuració de la barra de vida

El mètode SetUpLifesProcess() borra la barra de vida existent i la torna a carregar amb totes les vides.

```
IEnumerator SetUpLifesProcess(int initialLifes)
{
    lifeContainers.Clear();
    for (int i = transform.childCount - 1; i >= 0; i--)
    {
        Destroy(transform.GetChild(i).gameObject);
    }

    totalLifes = initialLifes;
    currentLifes = initialLifes;

    for (int i = 0; i < totalLifes; i++)
    {
        var current = Instantiate(lifePrefab, transform);
        lifeContainers.Insert(0,
        current.GetComponentInChildren<LifeBarContainer>()
   );
    }

    yield return null;
}
```

Guanyar vides

El mètode GainLifesProcess() fa que la barra de vida recuperi el nombre de vides indicat, o el nombre de vides màxim en el cas que el nombre de vides indicat superi les vides totals.

```
IEnumerator GainLifesProcess(int lifesUp)
{
    if (currentLifes < totalLifes)
    {
        bool desactivateDangerEffect = currentLifes == 1;

        int lifesUpdate =
        Mathf.Min(totalLifes, currentLifes + lifesUp);

        List<IEnumerator> coroutines =
        new List<IEnumerator>();
        for (int i = currentLifes; i < lifesUpdate; i++)
        {
            coroutines.Add(lifeContainers[i].Gain());
        }

        if (desactivateDangerEffect)
        {
            ActivateDangerEffect(false);
        }
```

```
21
22          yield return
23          Function.CoroutineChaining(coroutines.ToArray());
24          currentLifes = lifesUpdate;
25
26      }
27      else
28      {
29          yield return null;
30      }
31  }
```

En aquest mètode podem veure el CoroutineChaining, que també utilitzem a molts altres mètodes del component LifeBarController. Aquest mètode s'ocupa de no retornar res fins que totes les coroutines d'un array hagin acabat l' execució de forma seqüencial. Això ens ajuda a fer que la barra de vida pugi les vides d'una en una, creant un efecte de recuperació.

```
1  public static IEnumerator CoroutineChaining(params
       IEnumerator[] routines)
2  {
3      foreach (var item in routines)
4      {
5          while (item.MoveNext()) yield return item.Current;
6      }
7  }
```

Guanyar totes les vides

El mètode FillAllLifes() fa que la barra de vida recuperi el nombre màxim de vides. En el seu pas també reflecteix les vides que el jugador ja tenia.

```
1  IEnumerator FillAllLifes()
2  {
3      bool desactivateDangerEffect = currentLifes == 1;
4
5      List<IEnumerator> coroutines= new List<IEnumerator>();
6      for (int i = 0; i < totalLifes; i++)
7      {
8          if (lifeContainers[i].HasLife())
9          {
10             coroutines.Add(
11             lifeContainers[i].Reflection());
12         }
13         else
14         {
15             coroutines.Add(lifeContainers[i].Gain());
16         }
17     }
18
19     if (desactivateDangerEffect)
```

```
20      {
21          ActivateDangerEffect(false);
22      }
23
24      yield return
25      Function.CoroutineChaining(coroutines.ToArray());
26      currentLifes = totalLifes;
27  }
```

Perdre vides

El mètode LoseLifesProcess() fa que la barra de vida perdi el nombre de vides indicat.

```
1   IEnumerator LoseLifesProcess(int lifesOut)
2   {
3       if (currentLifes > 0)
4       {
5           int lifesUpdate = Mathf.Max(0,
6           currentLifes - lifesOut);
7
8           Coroutine lastOne = null;
9           for (int i = currentLifes - 1;
10          i >= lifesUpdate; i--)
11          {
12              lastOne =
13              StartCoroutine(lifeContainers[i].Lose());
14          }
15          currentLifes = lifesUpdate;
16
17          if (currentLifes == 1)
18          {
19              ActivateDangerEffect(true);
20          }
21
22          if (currentLifes == 0)
23          {
24              ActivateDangerEffect(false);
25              StartCoroutine(DieEffect());
26          }
27
28          yield return lastOne;
29
30      }
31      else
32      {
33          yield return null;
34      }
35  }
```

Com podem veure, aquest mètode s’ocupa d’activar un efecte d’alerta quan

el jugador només te una vida amb el mètode ActivateDangerEffect(). Aquest mètode mostra una aura negre metre la vida tremola.

```
private void ActivateDangerEffect(bool activate)
{
    StartCoroutine(lifeContainers[0].lastLife(activate));
}
```

LoseLifesProcess() també s'ocupa d'activar el procés de mort, en el cas que el jugador no li quedin vides, amb el mètode DieEffect(). El mètode canvia tots els sprites per vides trencades al mateix temps que tota la barra de vida tremola.

```
private IEnumerator DieEffect()
{
    foreach (LifeBarContainer life in lifeContainers)
    {
        life.SetShake(true);
    }
    yield return new WaitForSeconds(cCoroutineShakeWait);
    foreach (LifeBarContainer life in lifeContainers)
    {
        life.BrokeLife();
        life.SetShake(false);
    }
}
```

Guanyar una nova vida

El mètode AddNewLife() fa que la barra de vida guanyi totes les vides i que s'afegeixi una vida extra.

```
IEnumerator AddNewLife()
{
    List<IEnumerator> lst = new List<IEnumerator>();
    lst.Add(FillAllLifes());
    lst.Add(NewLife());
    yield return
    Function.CoroutineChaining(lst.ToArray());
}

private IEnumerator NewLife()
{
    totalLifes++;
    currentLifes = totalLifes;

    GameObject current =
    Instantiate(emptyLifePrefab, transform);
    current.transform.SetAsFirstSibling();

    ParticleSystem particleEffect =
    Instantiate(newLifeApearingParticleEffect,
    current.transform);
```

```
22      particleEffect.Play();
23      yield return
24      new WaitForSeconds(particleEffect.main.duration * 2);
25
26      Destroy(current);
27      GameObject currentLife =
28      Instantiate(lifePrefab, transform);
29      currentLife.transform.SetAsFirstSibling();
30      lifeContainers.Add(
31      currentLife
32      .GetComponentInChildren<LifeBarContainer>());
33 }
```

En el mètode NewLife() podem veure com instanciem, primer, una vida buida per efectuar un efecte de partícules i, posteriorment, afegim la vida. Aquest efecte de partícules dona la sanació de com si s'estigués creant quelcom tal com es feia als dibuixos antics. Veure figura 9.33.

Figura 9.32: Efecte de construcció. Tom & Jerry | Monster Jerry

LIFE BAR CONTAINER

Parlem ara del LifeBarContainer. Aquest component s'ocupa bàsicament d'animar les vides segons el que els hi dicti el component LifeBarController.

Les diferents tècniques que han creat totes les animacions són:

1. **Diferents sprites**
   La primera tècnica ha sigut el canvi d'imatges. A la figura 9.31 veiem les tres imatges possibles que pot tenir una vida.

```
1    filImage.sprite = lstSprites[UnityEngine.Random.
   Range(0, lstSprites.Count)];
2
```

Aquest tall de codi ens mostra com es canvia la imatge per una de les possibles imatges de vida trencada quan el personatge perd totes les vides.

2. **La propietat fillAmount**
   Les imatges a Unity tenen una propietat anomenada fillAmount, quan el tipus de la imatge és Filled, que permet mostrar només una quantitat de la imatge. Aquesta propietat, juntament amb la llibreria DOTween ens permet emplenar del 0% al 100%, o a la inversa, en un període curt de temps per fer una sensació d'emplenar o buidar.

```
    filImage.fillAmount = 1;
    filImage.DOFillAmount(0, 1f);

```

   Utilitzem aquesta tècnica en guanyar o perdre una vida, per donar la sensació d'emplenar o buidar el dipòsit.

3. **El mètode Shake**
   El mètode Shake() activa, durant un temps determinat, el mètode ShakeProcess() a l'Update que produeix l'efecte de temblor a la vida.

```
    IEnumerator Shake(float seconds)
    {
        SetShake(true);
        yield return new WaitForSeconds(seconds);
        SetShake(false);
    }

    private void ShakeProcess()
    {
        var speed = UnityEngine.Random
        .Range(cShakeMinSpeed, cShakeMaxSpeed);
        var amount =
        UnityEngine.Random
        .Range(cShakeMinDisplacement,
        cShakeMaxDisplacement);
        transform.localPosition = new Vector2(
            startPos.x +
            Mathf.Sin(Time.time * speed) * amount,
            startPos.y +
            Mathf.Sin(Time.time * speed) * amount
        );
    }

```

   Aquest mètode l'utilitzem per fer més tens el moment on el player només té una vida o s'ha mort.

4. **Ombra**

   Hem utilitzat una imatge d'ombra per emfatitzar quan només queda una vida. Aquesta apareix poca a poc gràcies al mètode de la llibreria DOTween, DoFade(), que mostra o amaga una imatge fent servir la transparència.

```
1     img.DOFade(img.color.a == cMaxTransparency ?
2     cMinTransparency : cMaxTransparency, Random.Range
    (1.5f, 2f))
3
```

   També rota perquè no tingui una aparença tan estàtica.

```
1     Function.RotateGameObject(shadowImage.transform,
    cRotationAmount);
2
```

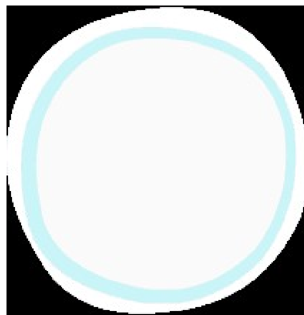

Figura 9.33: Sprite utilitzat per l'ombra

5. **Efectes de partícules**

   S'ha utilitzat un efecte de goteig quan es perd una vida, com si el dipòsit es buidés.

6. **La propietat maskable**

   Les imatges en Unity també tenen la propietat maskable. Aquesta permet que les imatges filles no es vegin fora del contorn de la imatge pare. Això fa que podem produir efectes visuals com el d'una reflexió.

```
1     public IEnumerator Reflection()
2     {
3         reflect.transform.
4         DOLocalMoveX(cReflectPosX, 0.25f);
5         yield return new WaitForSeconds(0.25f);
6         reflect.transform.localPosition =
7         new Vector2(-cReflectPosX, 0);
8     }
9
```

   Aquest mètode trasllada dues imatges blanques ràpidament perquè en el gameplay es percebi com una reflexió.

9.16.2.2 Panell d'habilitats

El panell d'habilitats s'ha creat per informar al jugador quan les habilitats estant disponibles i per mostrar si una habilitat s'està recarregant.

Si el jugador ha obtingut l'habilitat, aquesta es mostrarà al panell d'habilitats. Això ho gestiona el component PowersPanelManager, que llegeix si l'habilitat està adquirida a través del ScriptableObject.

```
private void Start()
{
    ManagePowersVisibility();
}

public void ManagePowersVisibility()
{
    rayPower.SetActive(PlayerController.Instance
    .AdquiredAbility(Ability.Ray));
    dashPower.SetActive(PlayerController.Instance
    .AdquiredAbility(Ability.Dash));
}
```

Com podem veure, el mètode ManagePowersVisibility(), és públic. Si en algun moment s'aconsegueix algun poder, es crida aquest mètode perquè actualitzar el panell.

Cada habilitat té el component UIPowerTimer. Aquest és l'encarregat de mostrar si l'habilitat es pot fer servir o està en estat de cooldown.

```
public void PowerUsed(float cooldownTime)
{
    coolingDown = true;
    waitTime = cooldownTime;
    imgContainer.fillAmount = 0;
    imgContainer.sprite = loadingSprite;
    tinkeling = false;
}

void FixedUpdate()
{
    if (coolingDown)
    {
        imgContainer.fillAmount += 1.0f / waitTime * Time.
    fixedDeltaTime;

        if (imgContainer.fillAmount == 1)
        {
            coolingDown = false;
            imgContainer.sprite = powerSprite;
```

```
20                tinkeling = true;
21                StartCoroutine(Tinkle());
22            }
23        }
24    }
```

Quan l'habilitat s'usa, es crida el mètode PowerUsed(). Aquest s'encarrega de canviar la imatge de l'habilitat per la imatge de càrrega. Llavors, dins del FixedUpdate(), s'emplenarà el fillAmount de la imatge a la mateixa velocitat que el temps de cooldown. Un cop la imatge estigui carregada, es llença un efecte de parpelleig per emfatitzar que ja es pot utilitzar.

Figura 9.34: Sprites utilitzats per el cronometre de les habilitats.

9.16.2.3 Notificacions

Les notificacions són una forma fàcil d'anunciar esdeveniments o canvis en el joc. S'han creat dues classes de notificacions.

1. **Les notificacions ràpides**
   Aquestes notificacions es fan servir per emfatitzar moments en el joc; com guanyar una vida nova. També estan pensades, en un futur, per notificar els triomfs. Veure apartat 12. El gameplay no ha de parar quan es mostren aquesta classe de notificacions.

2. **Les notificacions grans**
   Aquestes notificacions s'usen per indicar grans canvis en el joc; com l'obtenció d'habilitats noves.
   En aquesta classe de notificacions, és important obligar a l'usuari que les llegeixi amb deteniment, així que aturen durant uns segons el gameplay.

NOTIFICACIONS RÀPIDES
Les notificacions ràpides estan controlades a través del component NotificationManager. Aquest rep la informació de les notificacions a publicar pel mètode

PostNotification(). Seguidament, es crida el mètode NewNotification() que configura la notificació i la instància en el panell de notificacions.

```
public void PostNotification(string title, string
    description,
Sprite sprite)
{
    NewNotification(title, description, sprite);
}

private void NewNotification(string title, string
    description, Sprite sprite)
{
    var currentNotification = Instantiate(notification,
    this.transform);
    NotificationBehaviour notificationBehaviour =
    currentNotification.GetComponent<NotificationBehaviour
    >();
    if (notificationBehaviour != null)
    {
        StartCoroutine(notificationBehaviour.
    DisplayNotification(
            title,
            description,
            sprite,
            notificationLifeTime
        ));
    }
    else
    {
        Destroy(currentNotification);
    }
}
```

Internament, cada notificació té el component NotificationBehaviour. La coroutina DisplayNotification() s'ocupa d'emplenar els camps de la notificació; el títol, la imatge i la descripció; i s'ocupa de destruir-la al cap dels segons especificats. La destrucció de la notificació és important, aquestes notificacions indiquen accions concretes i no fa falta que romanguin permanentment a l'escena.

```
public IEnumerator DisplayNotification(string title,
    string description, Sprite sprite, float seconds)
{
    CanvasGroup cg = GetComponent<CanvasGroup>();
    this.image.sprite = sprite;
    this.title.text = title;
    this.description.text = description;
    cg.DOFade(1, 0.2f);
    yield return new WaitForSeconds(seconds);
    cg.DOFade(0, 0.5f).OnComplete(() =>
```

```
    {
        Destroy(this.gameObject);
    });
}
```

NOTIFICACIONS GRANS

Les notificacions grans usen el component BigNotificationManager. Aquest rep la informació de les notificacions a publicar a través del mètode ShowNotification(). Aquest mètode atura el gameplay, configura la notificació i la mostra per pantalla. Al cap de dos segons, el temps que s'ha decidit deixar abans que la notificació es pugui amagar, el boolà canHideNotification es posa a cert.

```
public void ShowNotification(Sprite image, string title,
   string description)
{
    Time.timeScale = 0;
    PlayerController.Instance.Controllable = false;
    this.image.sprite = image;
    this.title.text = title;
    this.description.text = description;
    notificationCanvasGroup.DOFade(1, 0.3f).SetUpdate(true
   ).OnComplete(
        () =>
        {
            info.DOFade(1, 0.2f).SetUpdate(true).SetDelay
   (2f).OnComplete(() =>
            {
                canHideNotification = true;
            });
        }
    );
}
```

Un cop el booleà canHideNotification es posi a true, el mètode Update() escoltarà a la interacció de qualsevol tecla per amagar la notificació. Un cop es cridi el mètode HideNotification(), la notificació s'amagarà i el gameplay continuarà.

```
private void Update()
{
    if (canHideNotification)
    {
        if (Input.anyKey)
        {
            HideNotification();
        }
    }
}

```

```
12 private void HideNotification()
13 {
14     canHideNotification = false;
15     notificationCanvasGroup.DOFade(0, 0.3f).SetUpdate(true
   ).OnComplete(() =>
16     {
17         PlayerController.Instance.Controllable = true;
18         info.color = Function.ColorVisible(false, info.
   color);
19         Time.timeScale = 1;
20     });
21 }
```

9.16.2.4 Indicador de tecles

L'indicador de tecles funciona com a tutorial pel jugador. Tot i que ens vam assegurar que el joc no utilitzes moltes tecles, sentíem que, en situacions noves, mostrar quines tecles s'han d'utilitzar a l'escena ajudaria amb l'experiència d'usuari.

L'indicador de tecles el compon una imatge que indica la tecla a prémer i una descripció per saber que fa la tecla. El seu comportament els dicta el component KeyIndicatorDisposer.

```
1  public void ShowTutorial(GameKey gameKey,
2  string functionallity)
3 {
4     if (gameKey == GameKey.Tab
5     || gameKey == GameKey.Space)
6     {
7         largeKey.GetComponent<CanvasGroup>().alpha = 1f;
8         defaultKey.GetComponent<CanvasGroup>().alpha = 0f;
9         currentKey = largeKey
10        .GetComponentInChildren<TextMeshProUGUI>();
11    }
12    else
13    {
14        largeKey.GetComponent<CanvasGroup>().alpha = 0f;
15        defaultKey.GetComponent<CanvasGroup>().alpha = 1f;
16        currentKey = defaultKey
17        .GetComponentInChildren<TextMeshProUGUI>();
18    }
19    function.text = functionallity;
20    currentKey.text = gameKey.ToString()
21    .Substring(0, 1).ToUpper()
22    + gameKey.ToString().Substring(1).ToLower();
23    showing = true;
```

```
    canvasGroup.DOFade(1, 0.25f);
}

public void HideTutorial()
{
    if (!showing)
        return;
    showing = false;
    canvasGroup.DOFade(0, 0.25f);
}
```

El comportament és molt simple: Quan es crida el mètode ShowTutorial() es configura i mostra el tutorial. Quan ja no es requereix, es crida el mètode HideTutorial() i s'amaga el tutorial.

L'indicador de tecles segueix al jugador per tota l'escena gràcies al component FollowPlayer.

```
public class FollowPlayer : MonoBehaviour
{
    [SerializeField] Vector3 offset = new Vector3(1,1,0);

    void Update()
    {
        if (PlayerController.Instance != null)
            transform.position =
            PlayerController.Instance.transform.position
            + offset;
    }
}
```

Per cridar el tutorial, s'utilitza al component KeyIndicatorTrigger. Es posa, o bé, als elements amb trigger que interactuen amb el jugador, o bé, a objectes buits que on la única funcionalitat és indicar un suggeriment de tecla al jugador.

El KeyIndicatorTrigger usa el mètode ShowTutorial(), quan el jugador entra amb contacte amb ell, i el HideTutorial(), quan hi deixa d'estar. Si el jugador clica la tecla que s'esta ensenyant, el tutorial també es deixa de mostrar com veiem en el mètode Update():

```
private void Update()
{
    if (playerIn
    && Input.GetKeyDown(gameKey.ToString().ToLower()))
    {
        itsNeeded = false; playerIn = false;
        KeyIndicatorDisposer.Instance?.HideTutorial();
    }
}
```

9.17 Gestió de diàlegs en NPCs

Els NPCs són personatges no jugador o personatges no jugables. En el nostre projecte, aquests interactuen amb el jugador per donar-li informació del món i la història.

Per aconseguir-ho hem hagut de crear un sistema de diàlegs enllaçat a cada NPC. Els dos components que gestionen aquesta interacció són el DialogueManager i el Dialogue.

```
public void TriggerDialogue()
{
    inConversation = true;

    var player = PlayerController.Instance;
    player.Controllable = false;
    Face facing = talkPoint.position.x - player.gameObject
   .transform.position.x > 0
    ? Face.Right : Face.Left;

    MovePlayer.Trigger(talkPoint, 0f, facing, 0, () =>
    {

        facing = talkPoint.position.x - transform.position
   .x > 0
        ? Face.Left : Face.Right;
        player.SetFacing(facing);

        GetComponentInChildren<Dialogue>().
   StartConversation(npcData, npcAnimator, () =>
        {
            player.Controllable = true;
            inConversation = false;
            actionAtEndOfConversation?.Invoke();
        });
    });
}
```

El DialogueManager s'encarrega de posicionar el jugador al costat del NPC i deixar-lo no controlable. Això ho aconsegueix a través del component MovePlayer explicat a la secció 9.8.

Un cop el player està posicionat s'accedeix al component Dialeg del NPC i s'inicia la conversació amb el mètode StartConversation(). En aquest li passes les dades del diàleg, l'animador del NPC i una Acció. L'acció està pensada per retornar el control al personatge quan el diàleg s'hagi acabat.

Les dades que es passen al diàleg les hem agrupat en una classe d'emmagatzemament anomenada NPCData. Com podem veure al següent codi aquesta conté el nom del NPC, les frases del diàleg, i els gestos que acompanyen aquestes frases.

```
[System.Serializable]
public class NPCData
{
    public string npcName;

    [TextArea(3, 135)]
    public string[] phrases;

    public NPCActions[] npcActions;

}
```

Els npcActions son tipus d'animacions que comparteixen tots els NPC. Gràcies a aquest mètode podem cridar les diferents animacions de forma genèrica.

```
public enum NPCActions
{
    idle, normal_talk, angry_talk
}
```

El component Dialogue es troba en el prefab DialogueCanvas que té cada NPC. Aquest s'encarrega d'animar com s'obre i s'amaga el diàleg i de mostrar les diferents frases de la conversa i les animacions del NPC que les acompanyen.

El diàleg s'obre amb el mètode StartConversation() aquest actualitza les variables amb els paràmetres. S'obre el diàleg amb el mètode OpenDialogue() que inicia l'animació d'obrir la bafarada. Quan la bafarada s'ha obert es crida en mètode OnDialogueOpened() comença a mostrar les frases.

```
public void StartConversation(NPCData npcData, Animator
   npcAnimator, Action onEndConversation = null)
{
    if (displayingSentences || settingUp)
        return;

    settingUp = true;

    this.npcAnimator = npcAnimator;
    this.onEndConversation = onEndConversation;

    phrases.Clear();
    for (int i = 0; i < npcData.phrases.Length; i++)
    {
        phrases.Enqueue((npcData.phrases[i],
```

```
15          NPCAnimations.ReturnHash(npcData.npcActions[i])));
16      }
17
18      OpenDialogue(npcData.npcName);
19  }
20
21  private void OpenDialogue(string name)
22  {
23      nameField.color = Function.ColorVisible(false,
        nameField.color);
24      nameField.text = name;
25      animator.SetTrigger("open_dialogue");
26  }
27
28  public void OnDialogueOpened()
29  {
30      nameField.color = Function.ColorVisible(true,
        nameField.color);
31      displayingSentences = true;
32      settingUp = false;
33      phraseEnded = false;
34      DisplayNextSentence();
35  }
```

En acabar la conversa es crida el mètode CloseDialogue() que inicia l'animació de tancar la bafarada. Quan la bafarada s'ha tancat es crida en mètode OnDialogueClosed() que restaura els valors pels inicials.

```
1   private void CloseDialogue()
2   {
3       settingUp = true;
4
5       animator.SetTrigger("close_dialogue");
6       npcAnimator.SetTrigger(NPCAnimations.Idle);
7
8       nextSentenceIndicator.DOKill();
9       nextSentenceIndicator.color = Function.ColorVisible(
        false, nextSentenceIndicator.color);
10
11      nameField.text = string.Empty;
12      conversationField.text = string.Empty;
13
14      StopSoundAfterCloseDialogue();
15  }
16
17  public void OnDialogueClosed()
18  {
19      settingUp = false;
20      displayingSentences = false;
21      onEndConversation?.Invoke();
```

```
22 }
```

Per mostrar les frases s'utilitza el mètode DisplayNextSentence() que prepara la següent frase a mostrar i indica a l'Animator quina animació toca. Seguidament, crida la corutina DisplayPhrase() que mostra la frase a poc a poc. Si en algun moment l'usuari interactua amb el personatge mentre aquest està en el procés de mostrar la frase, el booleà endPhrase es posa a cert i es mostra la frase completament.

```
1 private void DisplayNextSentence()
2 {
3     if (phrases.Count > 0)
4     {
5         PlayConversationClip();
6         phraseEnded = false;
7
8         nextSentenceIndicator.DOKill(); //needed if it's
   in de middle of DoFade
9         nextSentenceIndicator.color = Function.
   ColorVisible(false, nextSentenceIndicator.color);
10
11        (string, int) currentSentence = phrases.Dequeue();
12
13        npcAnimator.SetTrigger(currentSentence.Item2);
14
15        conversationField.text = string.Empty;
16        StartCoroutine(DisplayPhrase(currentSentence.Item1
   ));
17    }
18    else
19    {
20        CloseDialogue();
21    }
22 }
23
24 private IEnumerator DisplayPhrase(string currentSentence)
25 {
26    if (endPhrase)
27    {
28        phraseEnded = true;
29        conversationField.text = currentSentence;
30    }
31    else
32    {
33        yield return new WaitForSeconds(0.01f);
34        conversationField.text = currentSentence.Substring
   (0, conversationField.text.Length + 1);
35    }
36
```

```
37      if (currentSentence.Length == conversationField.text.
     Length)
38      {
39          phraseEnded = true;
40          nextSentenceIndicator.DOFade(1, endPhrase ? 0f : 1
     f);
41      }
42      else
43      {
44          StartCoroutine(DisplayPhrase(currentSentence));
45      }
46
47      endPhrase = false;
48  }
```

9.18 Cinematogràfiques

Les cinematogràfiques que componen el projecte estan fetes amb scripts i els mateixos elements de l'escena. Fer-ho així ens ha estalviat molt de temps i recursos gràfics.

Per entendre com funcionen, seguirem el funcionament de la primera cinematogràfica del joc que funciona amb el component C_IntroP1_controller.

Un cop carregada l'escena, mostrem les tecles del joc.

```
void Start()
{
    PlayerController.Instance.Controllable = false;
    initIndications.DOFade(1, 2f).SetDelay(2f).OnComplete
   (() =>
    {
        FadeIndications();
    });
}
```

Al cap de 5 segons, les amaguem juntament amb el fons negre. Deixem veure el paisatge de l'escena i movem els núvols per deixar veure el títol del joc durant 3 segons.

```
void FadeIndications()
{
    initIndicationsGroup.DOFade(0, 3f).SetDelay(5f).
   OnComplete(() =>
    {
        ShowTitle();
    });
}

void ShowTitle()
{
    cloud2.rectTransform.DOLocalMoveX(-1000, 5f);
    cloud1.rectTransform.DOLocalMoveX(1000, 7f).SetDelay(1
   f);
    gameTitle.rectTransform.DOScale(new Vector3(1.25f,
   1.25f, 1), 3f).OnComplete(() =>
    {
        HideTitle();
    });
}

void HideTitle()
{
    gameTitle.DOFade(0, 3f).SetDelay(5f).OnComplete(() =>
```

```
22     {
23         AjaxIsKicked();
24     });
25 }
```

Un cop amagat el títol, fem entrar l'ApachePig a l'escena per tirar una coça a l'Ajax.

```
1  void AjaxIsKicked()
2 {
3      pigAnimator.transform.DOLocalMoveX(81, 3f).SetDelay(3f
   ).OnStart(() =>
4      {
5          ChangeMusicAtmosphere();
6          PlayerController.Instance.SetFacing(Face.Right);
7          pigAnimator.SetBool("Run", true);
8      }).OnComplete(() =>
9      {
10         pigAnimator.SetBool("Run", false);
11         pigAnimator.transform.localScale = new Vector3(1,
   1, 1);
12         pigAnimator.SetTrigger("Coz");
13         pigAnimator.transform.DOLocalMoveX(81, 1f).
   OnComplete(AjaxFalls);
14     });
15 }
```

Un cop s'ha donat la coça, es crida el mètode AjaxFalls() que tira anima com l'Ajax cau pel precipici i carrega la següent escena.

```
1 void AjaxFalls()
2 {
3     PlayerController.Instance.Animator.SetTrigger("jump");
4     PlayerController.Instance.Animator.SetBool("jumping",
   true);
5     PlayerController.Instance.transform.DOLocalJump(new
   Vector3(60, 3, 1), 20f, 1, 5f).OnComplete(() =>
6     {
7         StartCoroutine(Loader.LoadWithDelay(SceneID.
   Cinematographic_Intro_P2, 0.1f));
8     });
9 }
```

Com hem pogut apreciar les cinematogràfiques es componen majoritàriament per dos elements:

1. La llibreria DOTween per girar i traslladar objectes.

2. La utilització de les animacions dels personatges.

Les quatre cinematogràfiques que podem trobar en la DEMO són les següents:

- C_IntroP1_controller: Es mostren les tecles i el títol de joc. Es presenta el primer enemic, ApachePig, que dona una coça a l'Ajax i el fa caure per un barranc.
- C_IntroP2_controller: El Capità Java li explica a l'Ajax que la seva germana necessita ajuda.
- C_IslandTransition_controller: Transició d'anar de l'illa Omed a l'illa dels Homes.
- C_EndDemo_controller: Ajax es retroba amb Cassandra, s'abracen i surten els crèdits.

CAPÍTOL 10

Implantació i resultats

10.1 Legislació i normativa vigent

El projecte desenvolupat no presenta cap problema des del punt de vista legislatiu. Tot i que el projecte té un sistema per encriptar dades i guardar-les, no s'ha tingut en compte el Reglament General de Protecció de dades (RGPD), ja que el sistema en cap moment tracta cap mena de dades critiques relatives als usuaris.

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 Personatge principal

En aquesta secció trobem les funcionalitats resultants de la implementació del personatge principal. Podem veure les especificacions de la implementació a l'Apartat 9.1

Figura 10.1: Personatge principal en repos.

10.2.1 Desplaçament

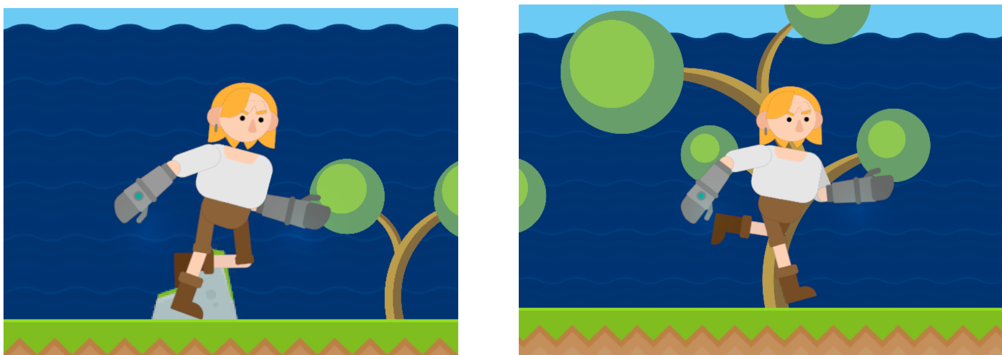

Figura 10.2: Personatge principal corrent.

10.2.2 Desplaçament ràpid

Figura 10.3: Personatge principal fent desplaçament ràpid.

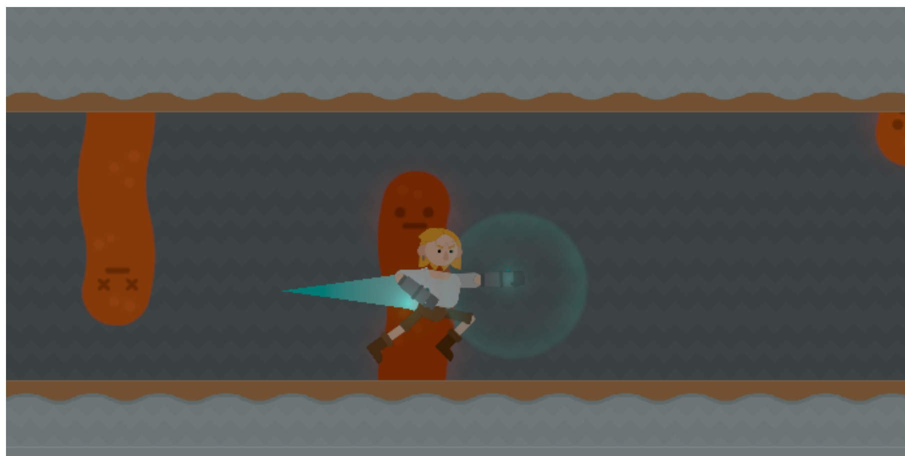

Figura 10.4: Personatge principal danyant els enemics mentre fa el desplaçament ràpid.

10.2.3 Salt i Aterratge

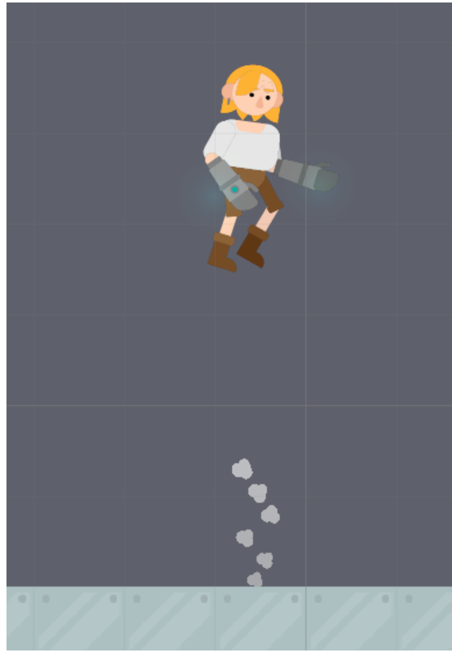

Figura 10.5: Personatge principal saltant.

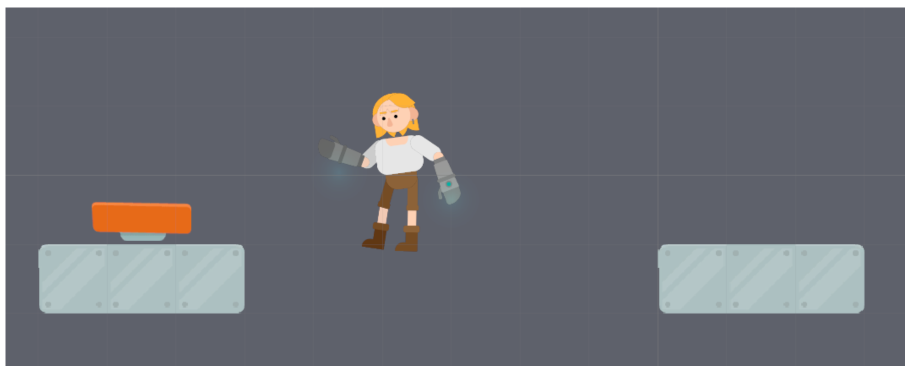

Figura 10.6: Personatge principal aterrant.

10.2.4 Atac

Figura 10.7: Personatge principapl fent sequència d'atac bàsic.

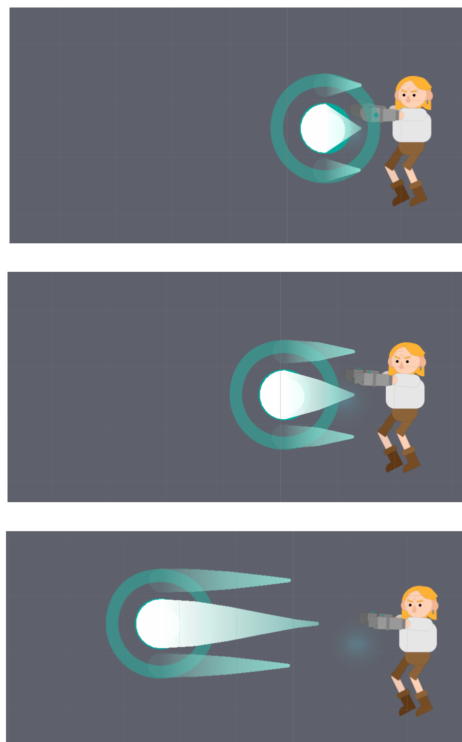

Figura 10.8: Personatge principal instanciant raig.

10.2.5 Curació

Figura 10.9: Efecte de curació sobre el personatge principal.

10.2.6 Dany i protecció

Figura 10.10: Personatge principal rep mal.

Figura 10.11: Personatge principal en estat de recuperació.

10.3 Interacció de l'escena amb el personatge

En aquest apartat podem veure les diferents maneres de com el personatge rep mal. L'explicació d'aquestes dues funcionalitats es troba a l'Apartat 9.2.

Figura 10.12: Personatge interaciona amb un component Hazard.

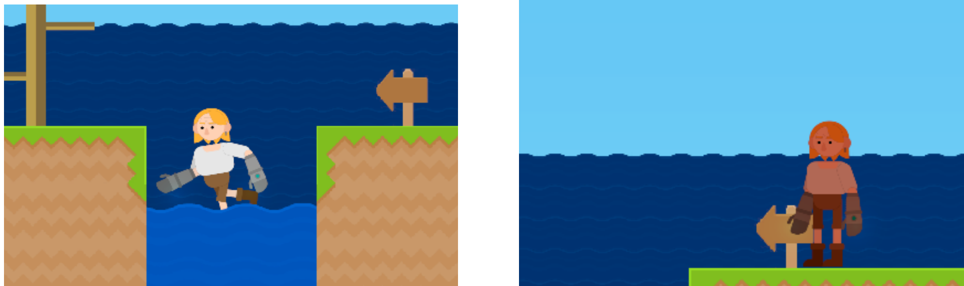

Figura 10.13: Personatge interaciona amb un component DeadlyObject.

10.4 Lluita per fases

En aquesta secció es veuen els resultats de la implementació de la lluita per fases. Aquesta funcionalitat s'ha utilitzat pels dos bosses; Apache Pig i FALSE Knight. Per veure la implementació anar a l'Apartat 9.3.

10.4.1 Apache Pig

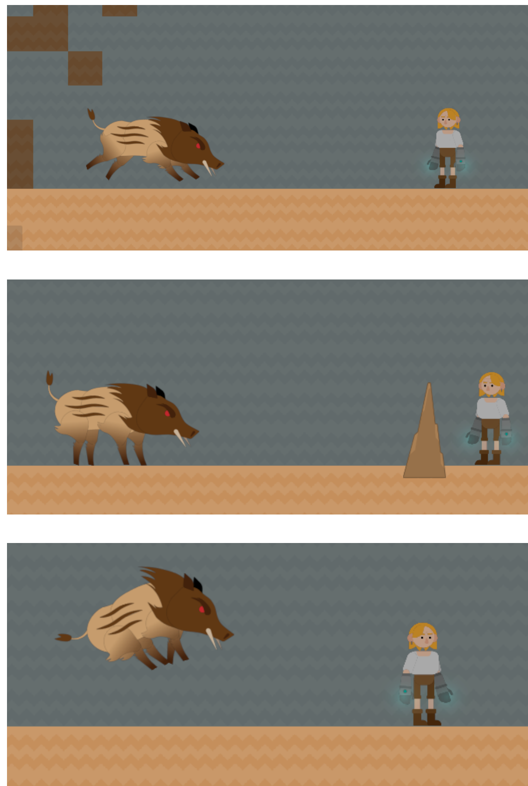

Figura 10.14: Apache Pig en les diferents fases.

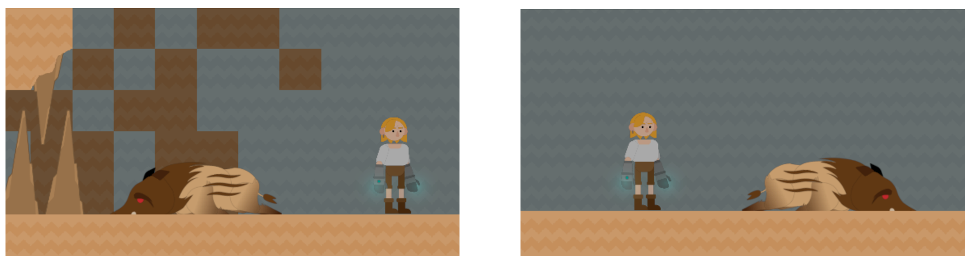

Figura 10.15: Apache pig canviant de fase.

10.4.2 FALSE Knight

Figura 10.16: FALSE knight en les diferents fases.

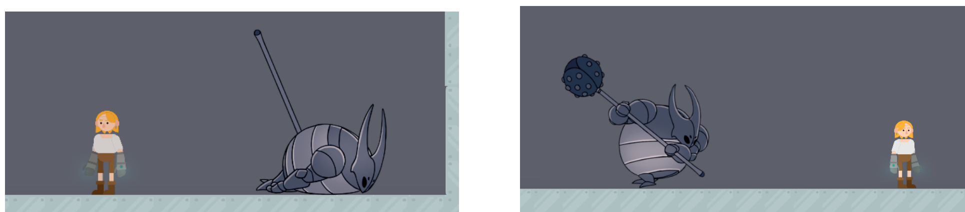

Figura 10.17: FALSE knight en el canvi de fase.

10.5 Tasques

Com s'ha explicat a l'Apartat 9.4, hi ha algunes funcionalitats de les IA que són utilitzades pels diferents enemics. En aquest apartat veurem com enemics diferents comparteixen funcionalitats.

10.5.1 Orientar la IA cap el jugador

Figura 10.18: Whorm seguint el jugador amb l'orientació.

Figura 10.19: False Knight seguint el jugador amb l'orientació.

10.5.2 Llançament de projectils

Figura 10.20: Planta llançant un projectil.

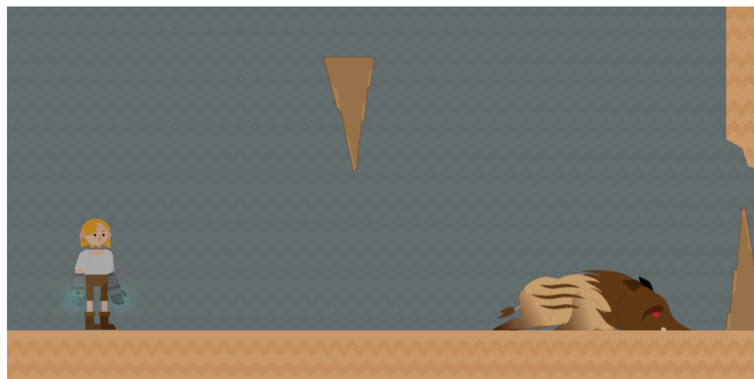

Figura 10.21: Apache Pig llençant un projectil.

10.5.3 Destrucció de la intel·ligència artificial

Figura 10.22: Destrució IA Crawler per falta de vida. Deixa un sprite.

Figura 10.23: Destrució IA Apache Pig per falta de fases. Roman a l'escena l'ultim frame de l'animació de mort.

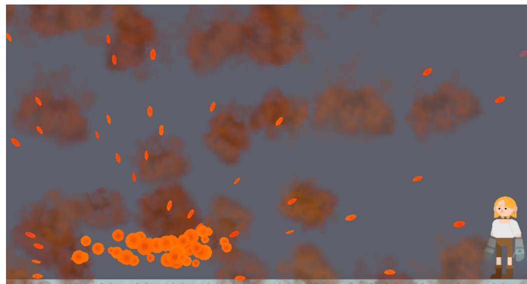

Figura 10.24: Destrució IA FALSE Knight per falta de fases. Produeix un efecte de partícules.

10.6 Gestió de recursos dintre de l'escena

A l'Apartat 9.5.1 s'explica com es gestionen els sprites resultants de les morts dels enemics. En aquest apartat podem veure diversos exemples.

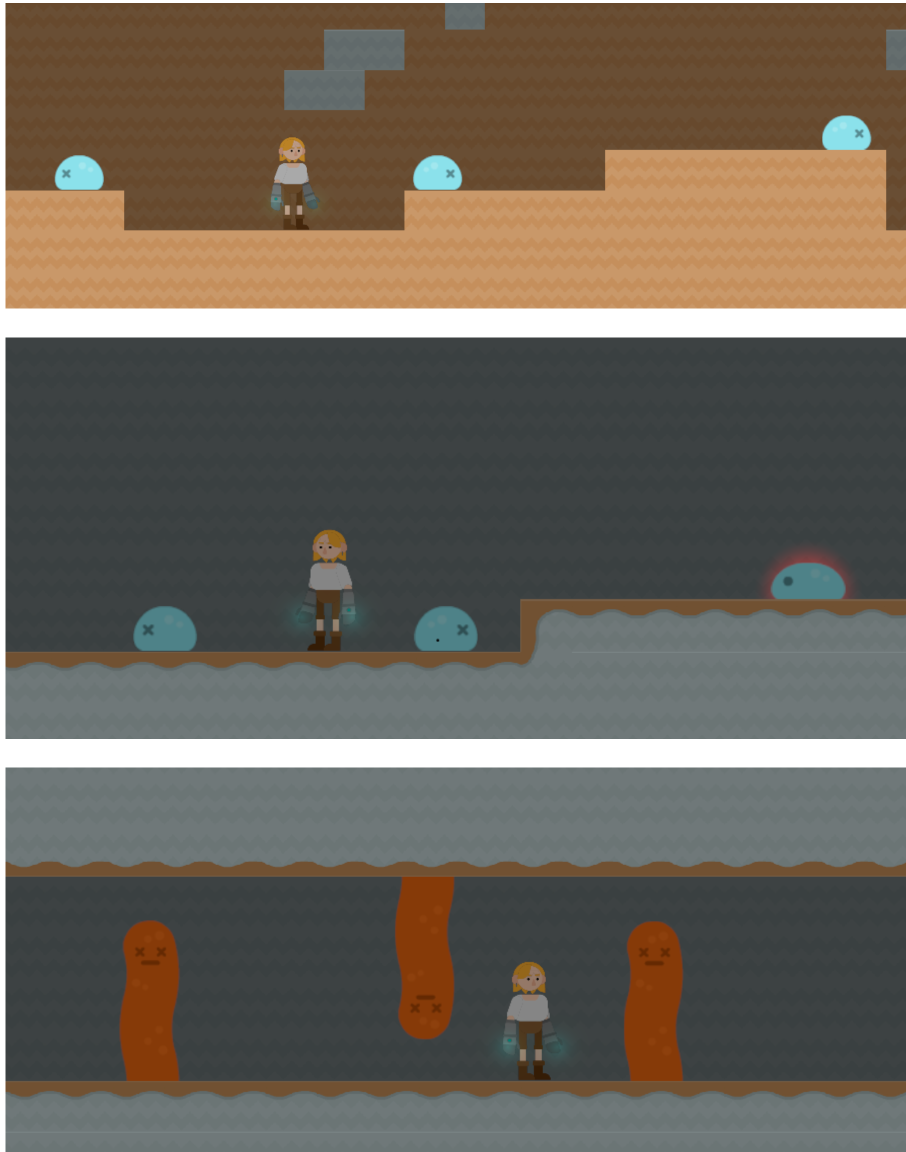

Figura 10.25: Exemples gestió de cadavers.

10.7 Punt de guardat

Ara mostrarem l'objecte utilitzat que s'ocupa de guardar l'estat del joc quan el jugador li demana. Podem veure la seva implementació a l'Apartat 9.6.

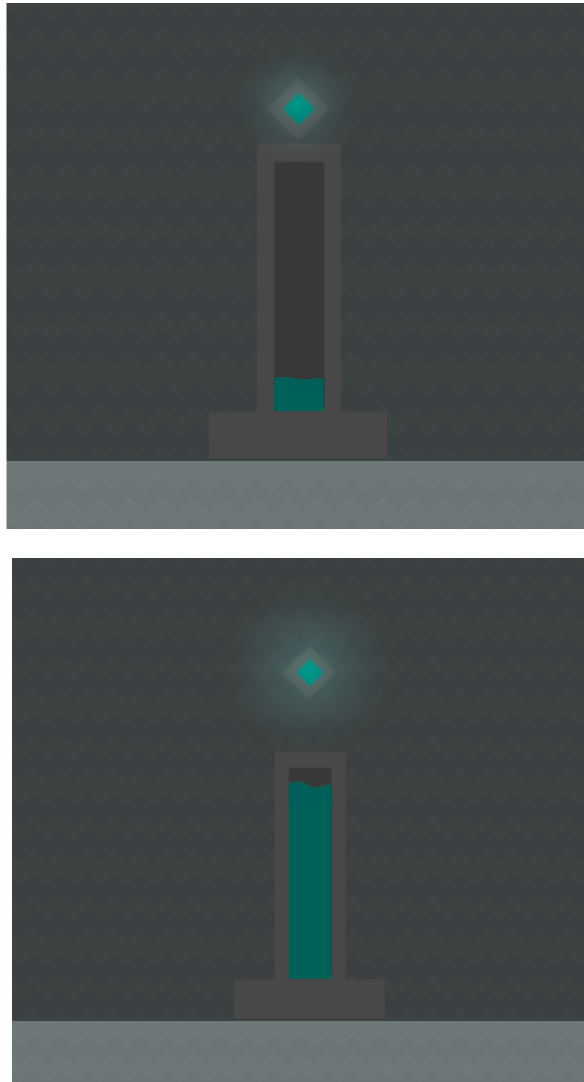

Figura 10.26: Activació d'un punt de guardat.

10.8 Punts de control

La implementació de retorn a un punt de control, explicat a l'Apartat 9.7, es pot veure il·lustrat a les següents imatges.

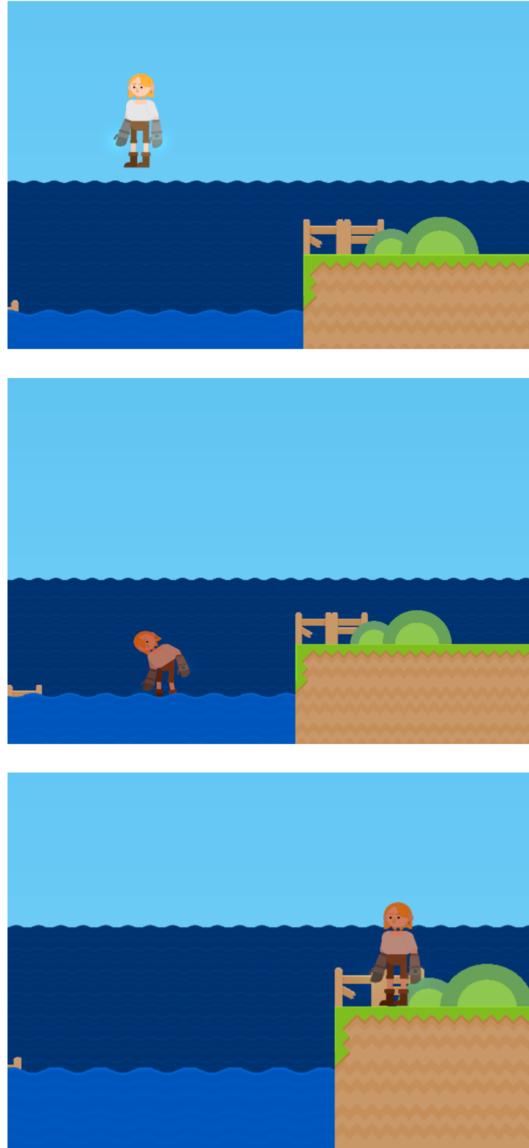

Figura 10.27: Seqüència de retorn a un punt de control.

10.9 Canvi d’escena

A continuació podem veure els resultats de com el personatge fa un canvi d’escena. L’explicació de la implementació es troba a l’Apartat 9.8.

Figura 10.28: Sequència de com Ajax entra a l’escena.

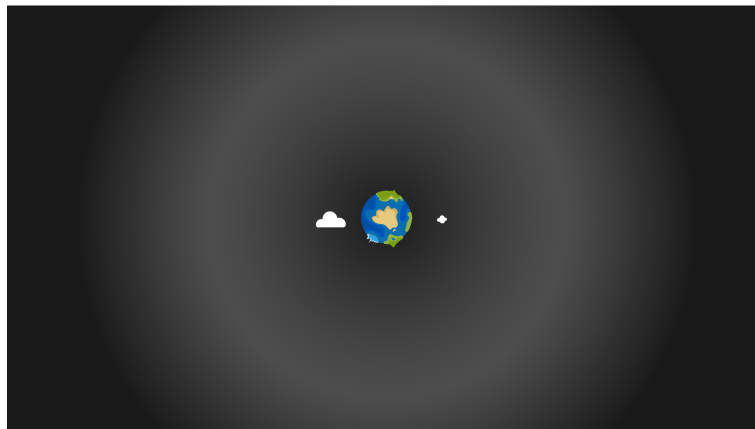

Figura 10.29: Escena de càrrega.

10.10 Càmera

La càmera ha estat configurada amb el paquet Cinemachine. En les següents imatges en podem veure els resultats.

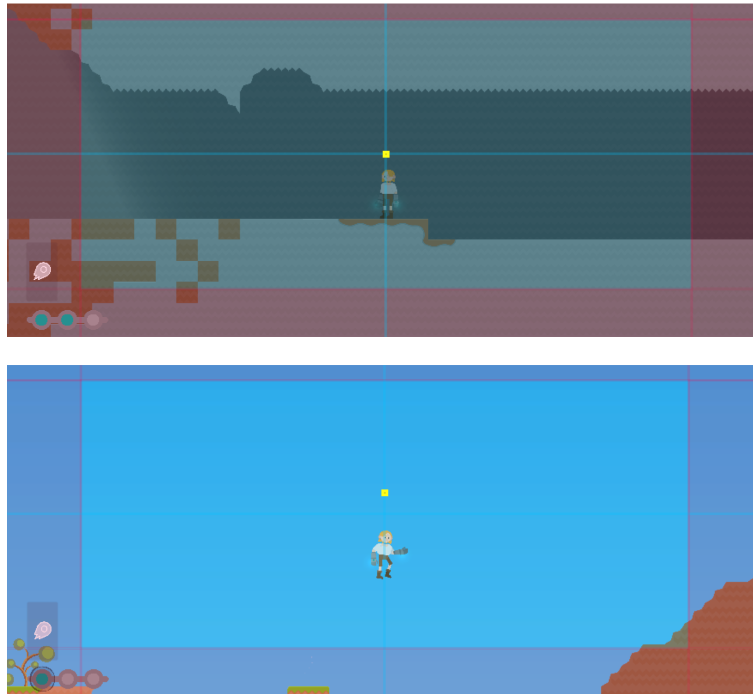

Figura 10.30: Configuració càmera dins del joc.

Com podem veure a la Figura 10.30, la creu blava, estarà sempre al centre de la pantalla del joc. Aquesta segueix en tot moment el punt groc. Aquest punt està configurat perquè estigui sempre sobre del jugador.

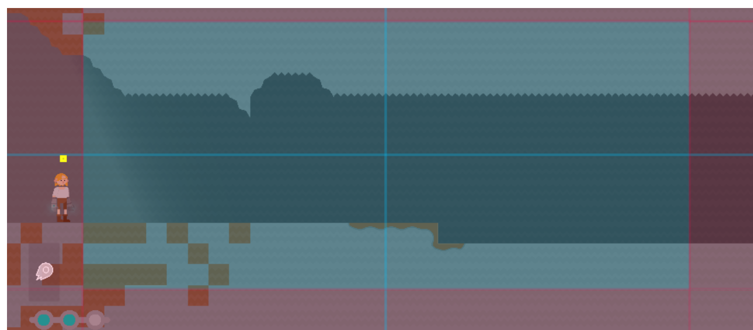

Figura 10.31: Càmera afectada pel seu rang.

Quan el jugador surti del rang de la càmera, el punt groc el seguirà i la càmera quedarà descentrada. Veure Figura 10.31.

10.11 Tilemap

En aquesta secció veurem els diferents exemples d'utilització de tilemaps explicats a l'Apartat 9.10.

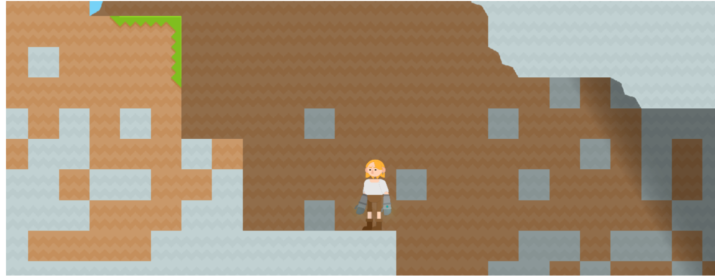

Figura 10.32: Exemple escena amb tilemap base i tilemap decoratiu pel fons.

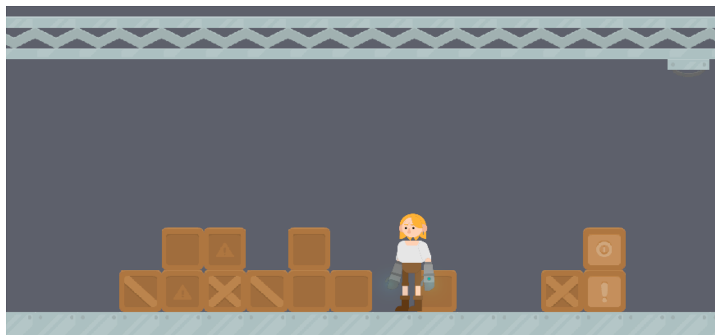

Figura 10.33: Exemple escena amb tilemap base i tilemap decoratiu pels objectes.

Figura 10.34: Exemple escena amb tilemap base, tilemap decoratiu pel fons i tilemap que danya el jugador.

10.12 Contigut d'escena per capes

Mostrem els resultats obtinguts de la utilització de les capes per crear profunditat. Veure Secció 9.11 per veure la implementació.

Figura 10.35: Distrubucció de capes visuals en Omeled Island.

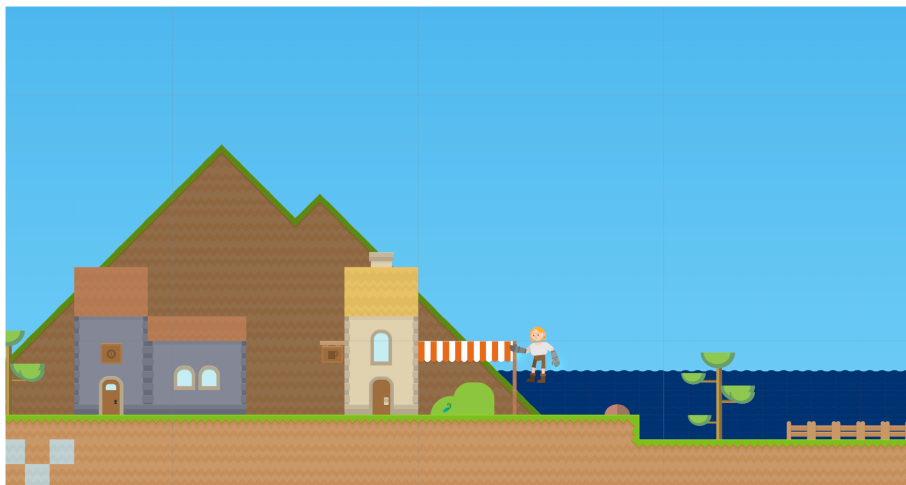

Figura 10.36: Distrubucció de capes visuals en Men Island.

10.13 Il·luminació d'escenes

A continuació veurem diferents exemples, explicats a la Secció 9.12, de com s'ha utilitzat el paquet de Unity, Lights 2D, per il·luminar l'escena.

10.13.1 Il·luminació per ressaltar elements

Figura 10.37: Il·luminació player.

Figura 10.38: Il·luminació a Capità Java.

10.13.2 Il·luminació per ressaltar elements perillosos

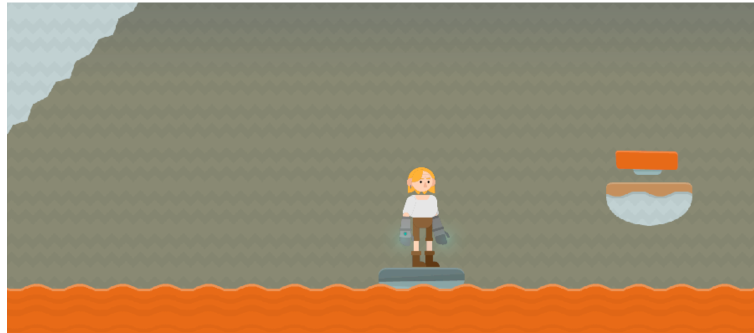

Figura 10.39: Il·luminació de la lava.

Figura 10.40: Il·luminació d'un enemic.

10.13.3 Il·luminació per elements decoratius

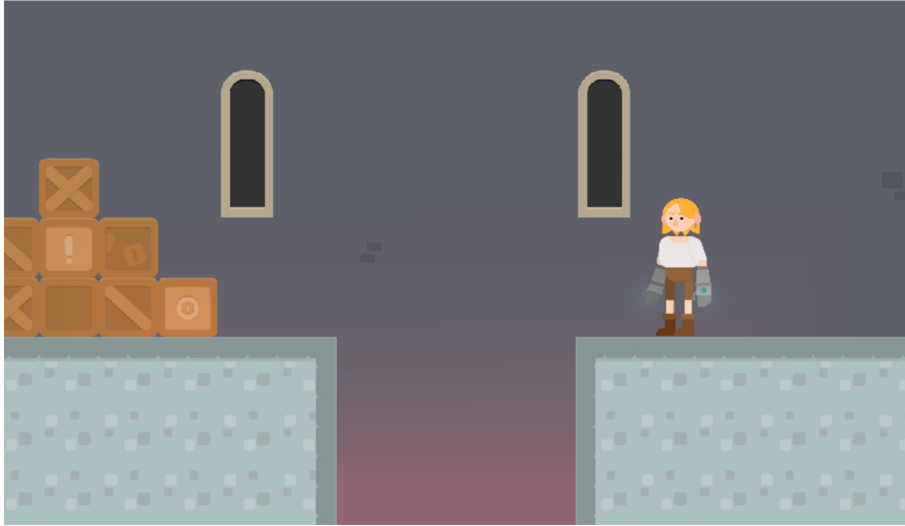

Figura 10.41: Il·luminació que emfatitza l'entrada al laboratori.

10.14 Plataformes

Ara mostrarem diferents exemples de plataformes en el joc. Entre aquests, la implementació de les plataformes dinàmiques explicades a l'Apartat 9.15.

Figura 10.42: Plataformes estatiques.

10.14.1 Plataformes simples

Figura 10.43: Plataforma dinàmica amb moviment horitzontal.

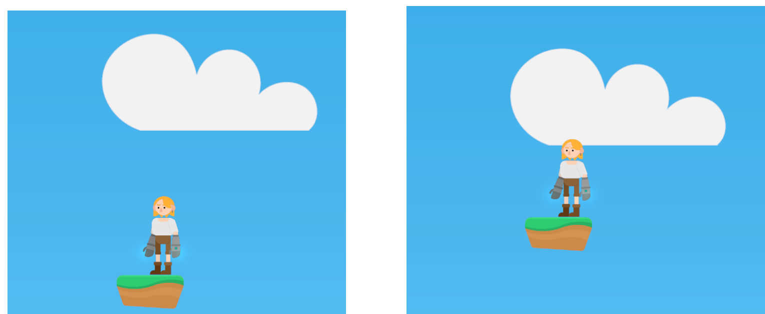

Figura 10.44: Plataforma dinàmica amb moviment vertical.

10.14.2 Plataformes dinàmiques

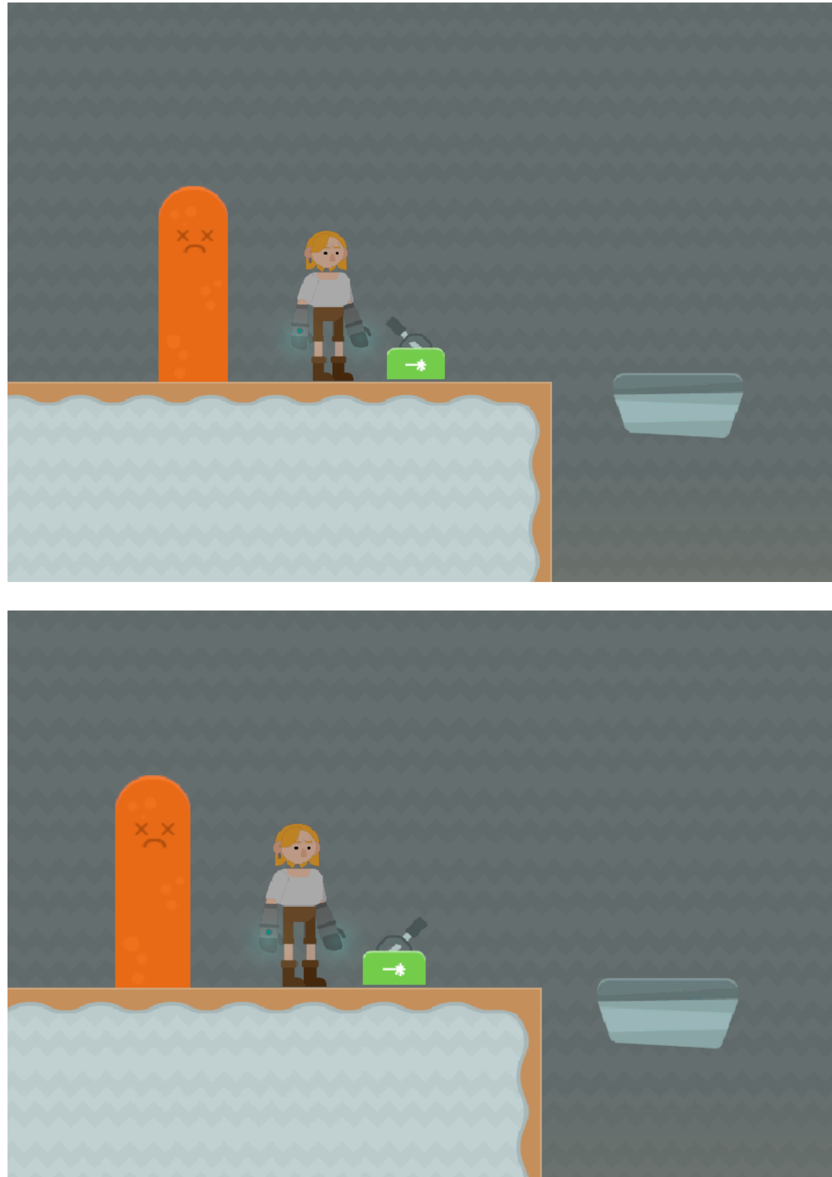

Figura 10.45: Plataforma mecànica.

10.15 Portes

En aquest apartat podem veure els resultats obtinguts de la implementació de les portes explicades a l'Apartat 9.15.

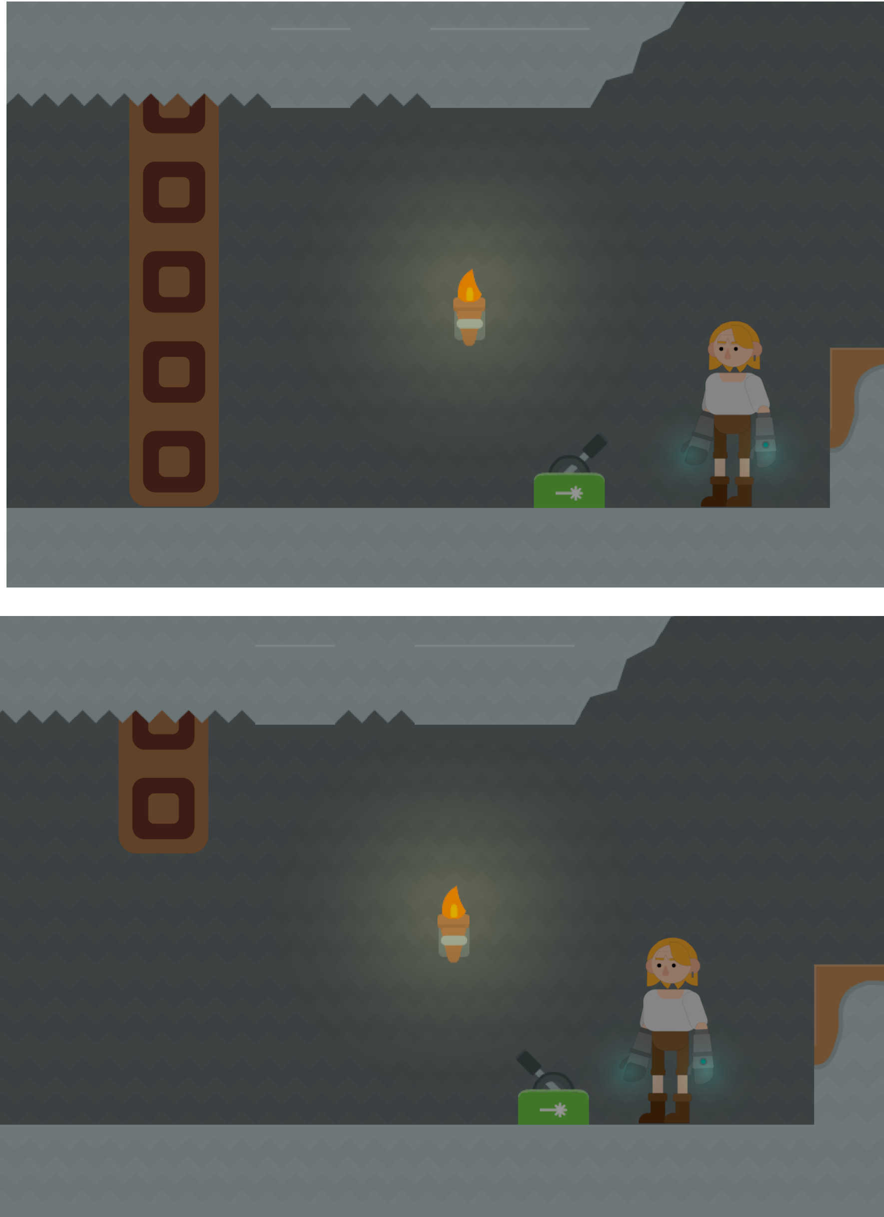

Figura 10.46: Porta mecànica.

10.16 Interficies d'usuari

En els següents subapartats mostrarem els resultats obtinguts de la implementació de les interfícies d'usuari. Veure Secció 9.16 per l'explicació del funcionament dels menús, el canvas i els diàlegs.

10.16.1 Menus

10.16.1.1 Menú d'inici

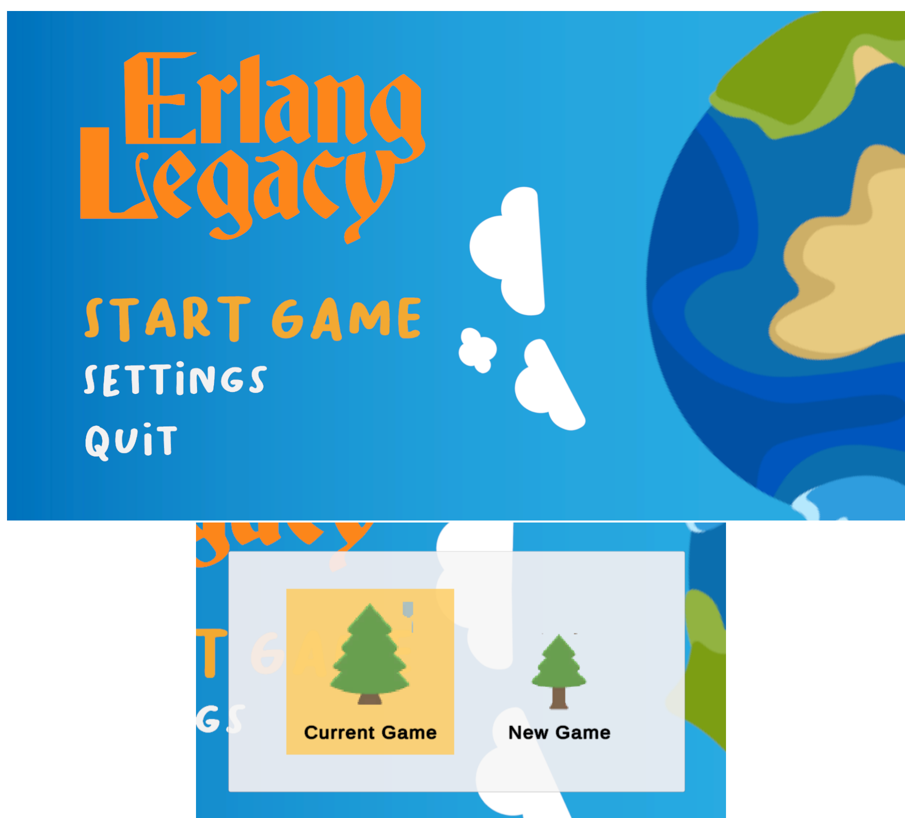

Figura 10.47: Menu principal i submenu per escollir partida.

10.16.1.2 Menú de pausa

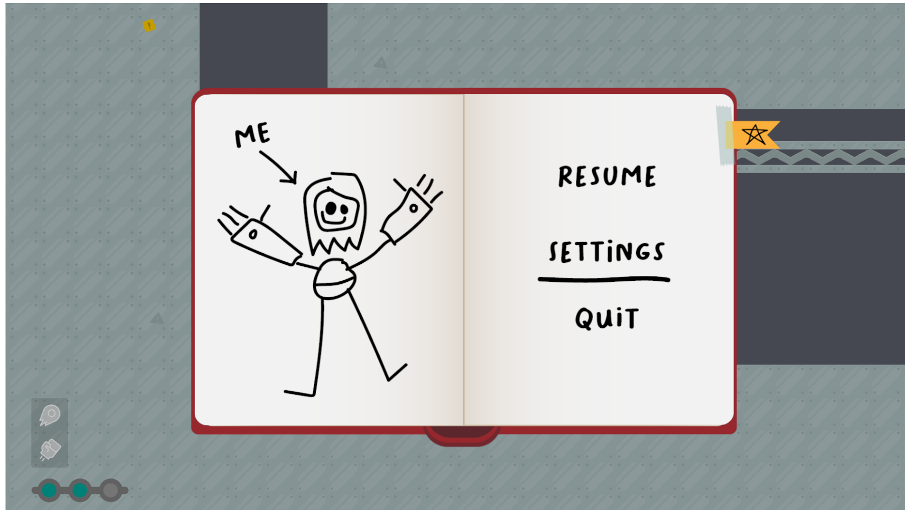

Figura 10.48: Menu de pausa.

10.16.1.3 Menú de configuracions

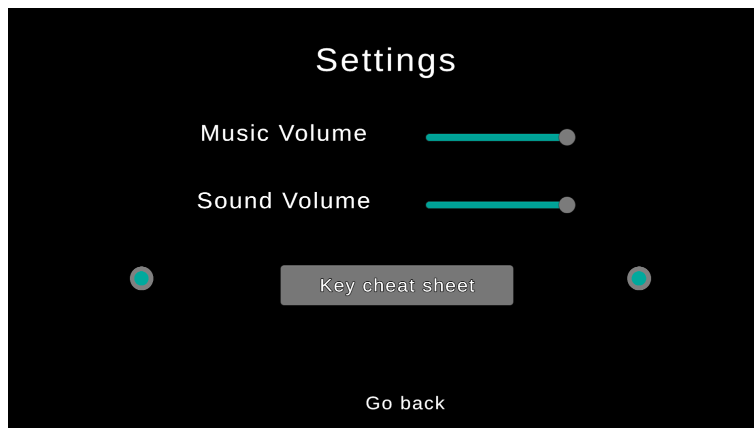

Figura 10.49: Menu de configuració.

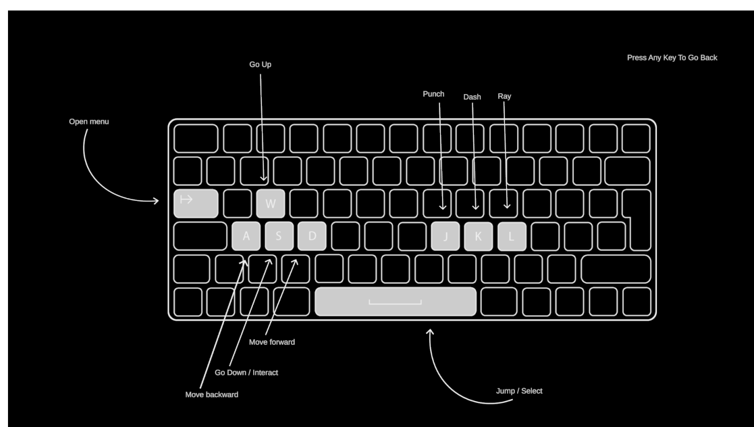

Figura 10.50: Menu del mapatge de tecles.

10.16.2 In-Game Canvas

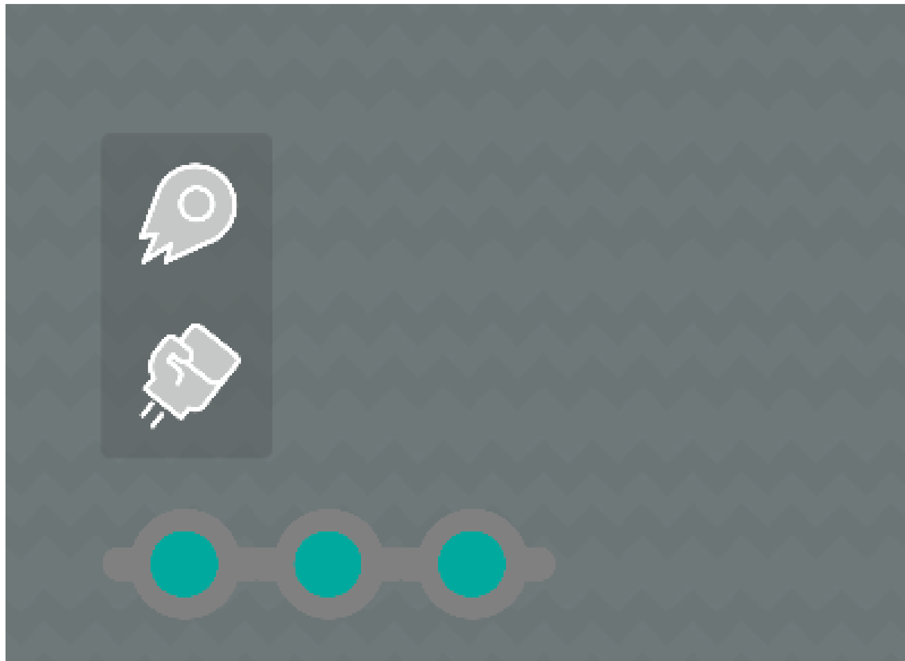

Figura 10.51: Visualització de l'estat de vides i habilitats del personatge principal.

10.16.2.1 Barra de vida

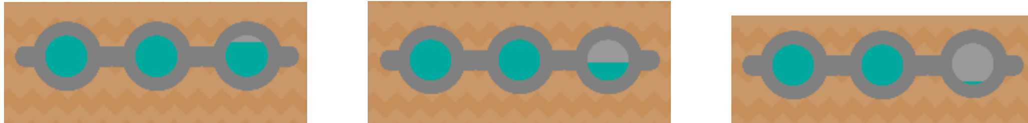

Figura 10.52: Efecte de perdre vida.

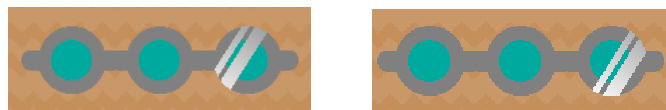

Figura 10.53: Efecte reflex.

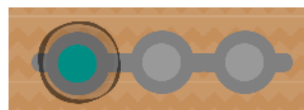

Figura 10.54: Barra de vida amb una sola vida.

Figura 10.55: Barra de vida sense vides.

10.16.2.2 Panell d'habilitats

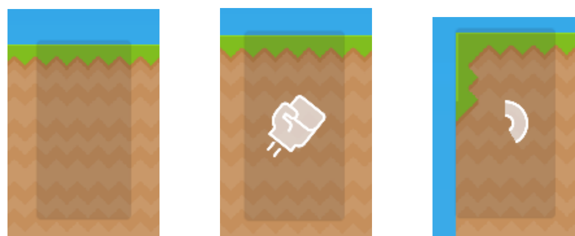

Figura 10.56: Panell d'habilitats sense el dash, amb el dash i amb el dash en cooldown.

10.16.2.3 Notificacions

Figura 10.57: Notificació.

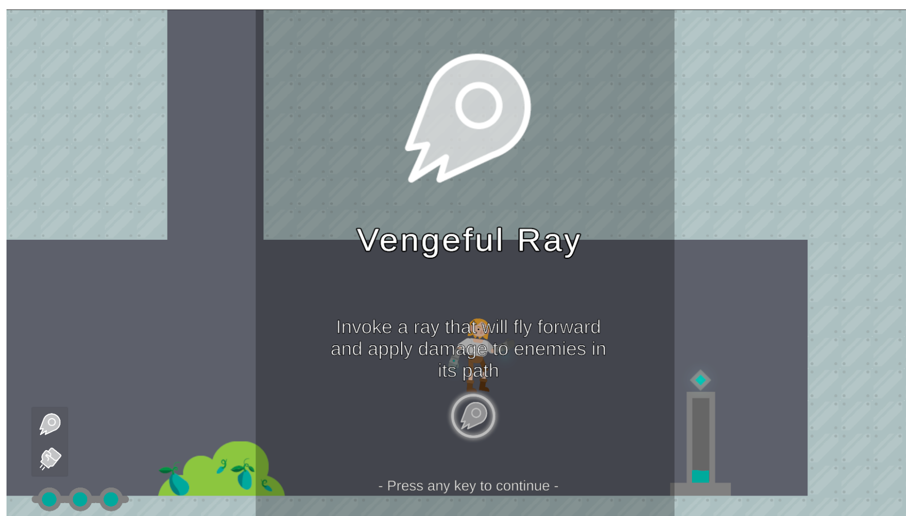

Figura 10.58: Notificació gran.

10.16.2.4 Indicador de Tecles

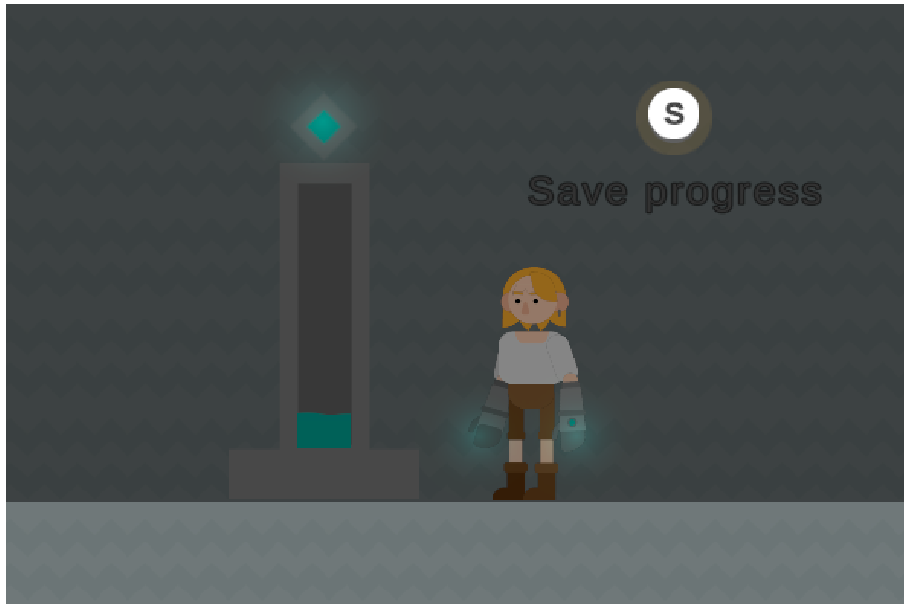

Figura 10.59: Consell d'interacció.

10.16.3 Diàlegs

10.16.3.1 Capità Java

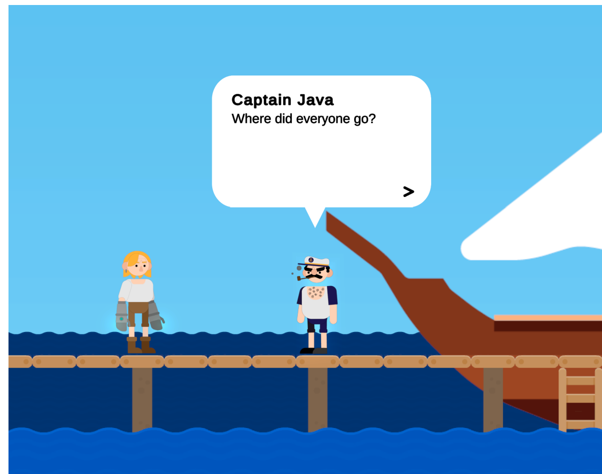

Figura 10.60: Capità Java en Men Island.

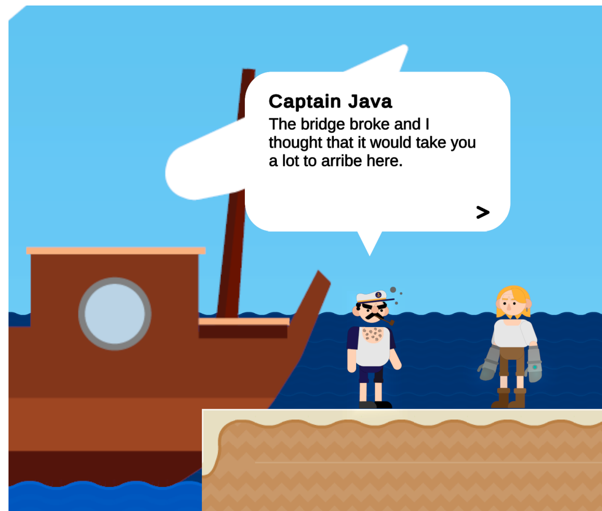

Figura 10.61: Capità Java en Omeled Island.

10.16.3.2 Cassandra

Figura 10.62: Cassandra parlant per altaveu al Laboratori.

10.17 Cinematografíes

En aquest apartat podem veure exemples dels resultats de les cinematografies.

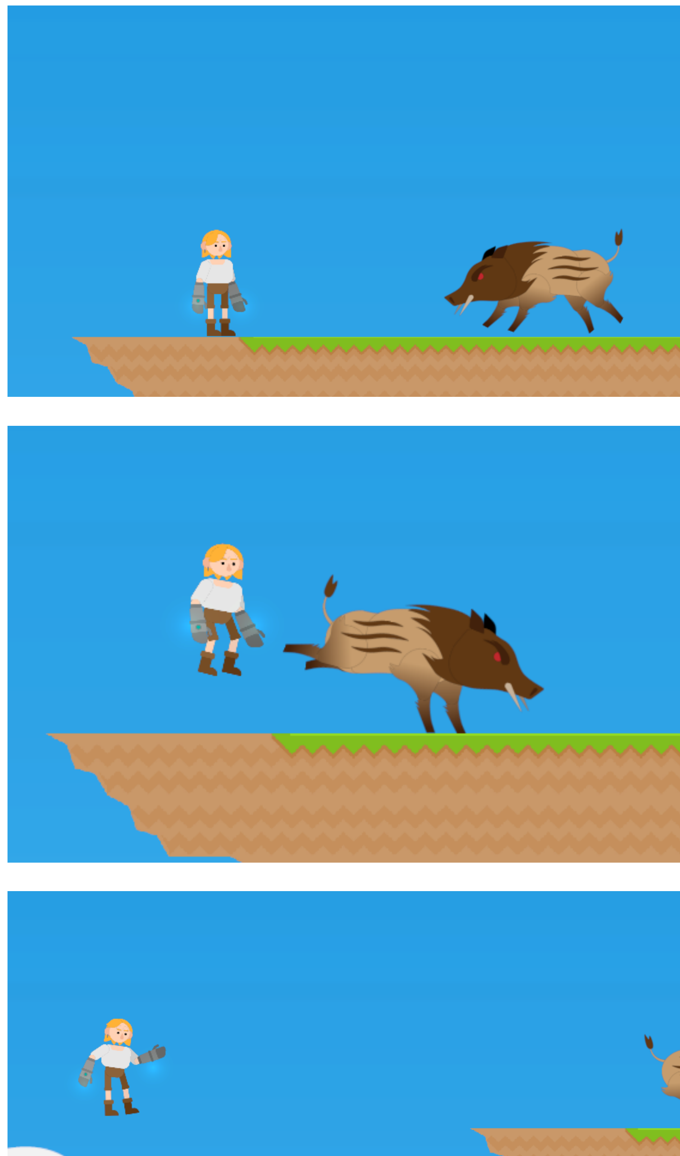

Figura 10.63: Apache Pig empeny a Ajax.

Figura 10.64: Retrobomanent amb Cassandra.

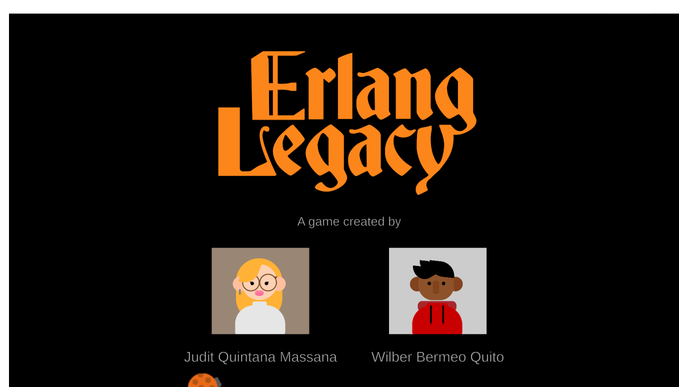

Figura 10.65: Crèdits.

10.18 Illa Omed

En aquest apartat podem veure els resultats del disseny del mapa de la primera illa, l'Illa Omed, mostrat a l'Apartat 8.7.

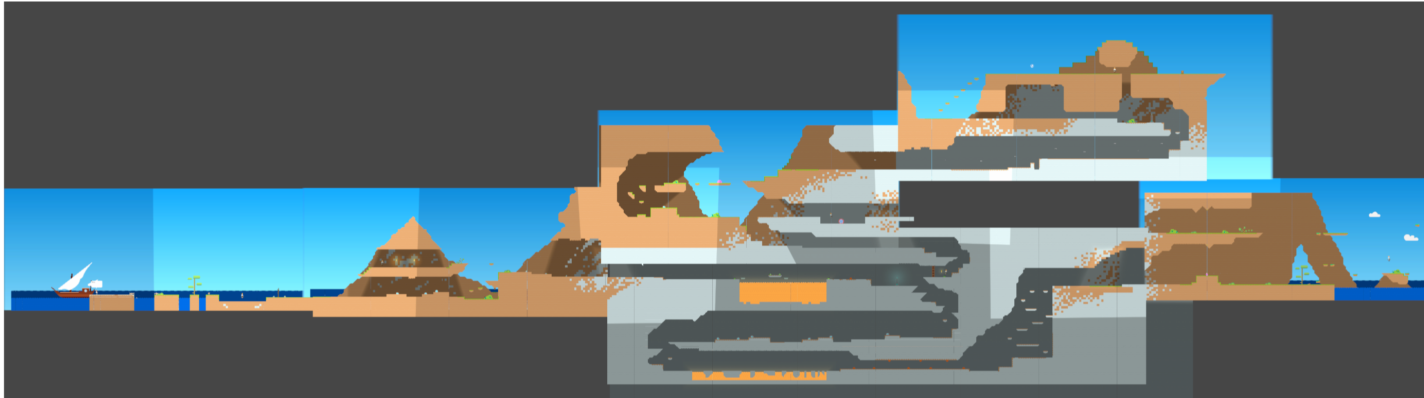

Figura 10.66: Visualització general de la illa Omed.

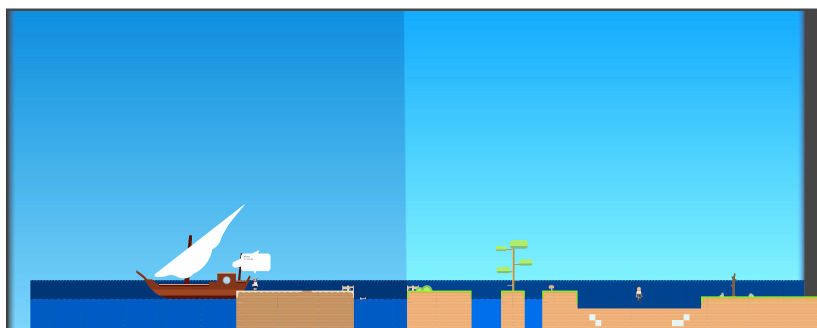

Figura 10.67: Illa Omed, scena 1.

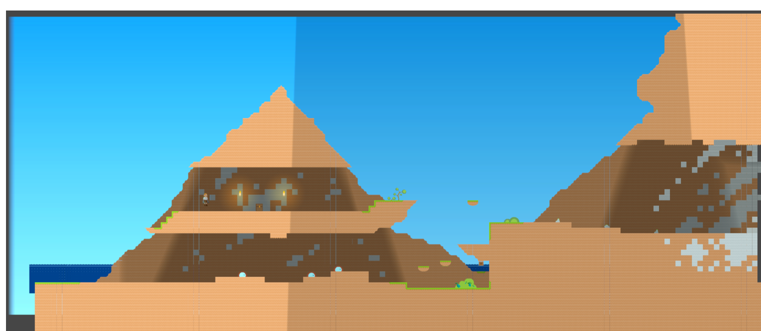

Figura 10.68: Illa Omed, scena 2.

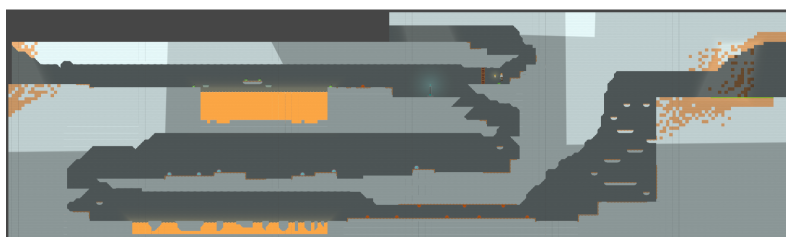

Figura 10.69: Illa Omed, scena 3.

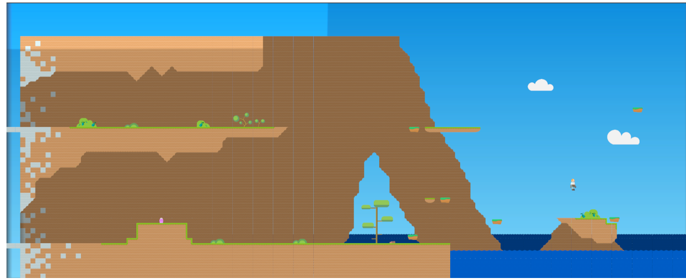

Figura 10.70: Illa Omed, scena 4.

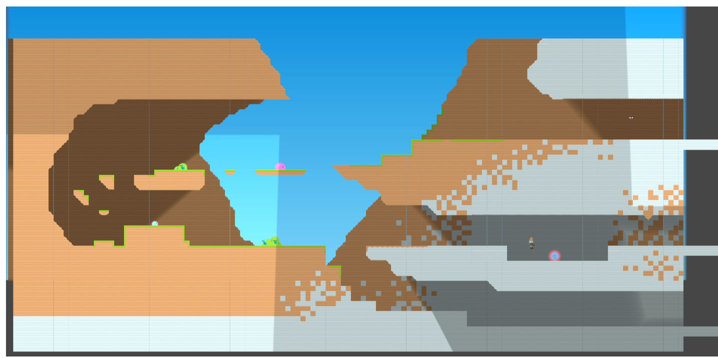

Figura 10.71: Illa Omed, scena 5.

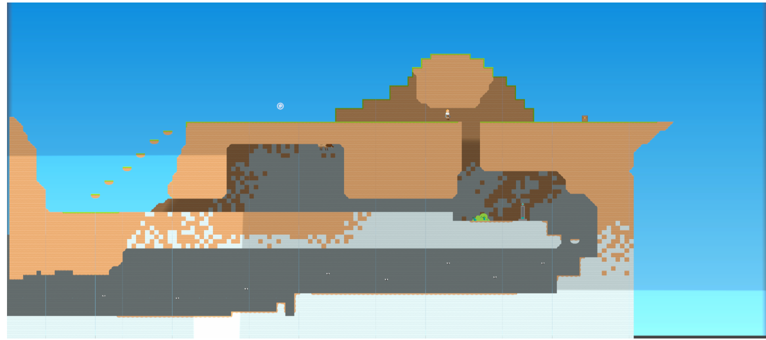

Figura 10.72: Illa Omed, scena 6.

10.19 Illa dels Homes

En aquest apartat podem veure els resultats del disseny del mapa de la segona illa, l'Illa dels Homes. Veure Apartat 8.7.

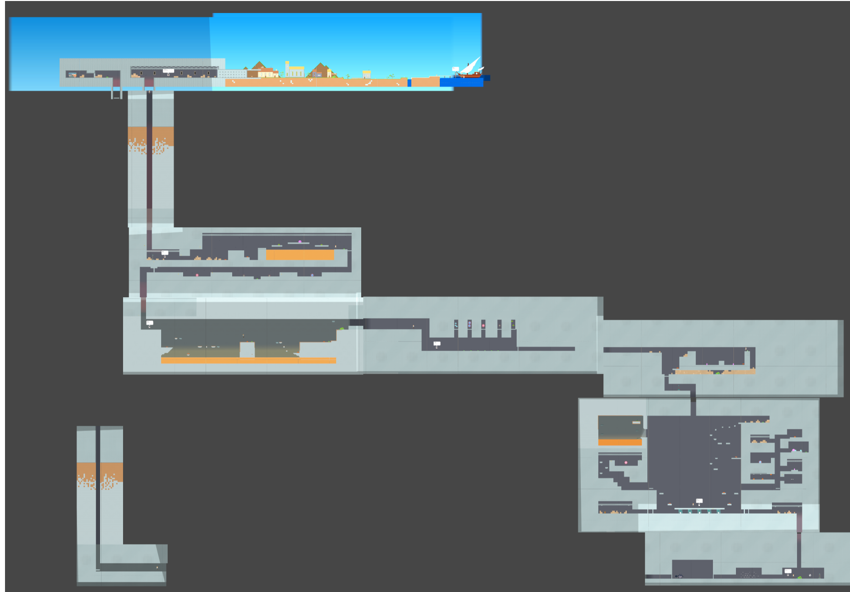

Figura 10.73: Visualització general de l'illa dels Homes.

Figura 10.74: Illa dels Homes, scena 1.

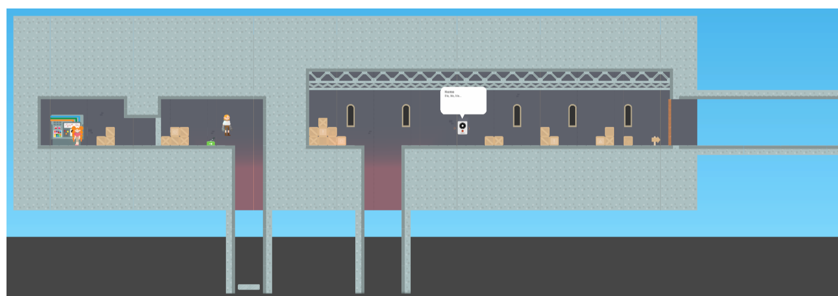

Figura 10.75: Illa dels Homes, scena 2.

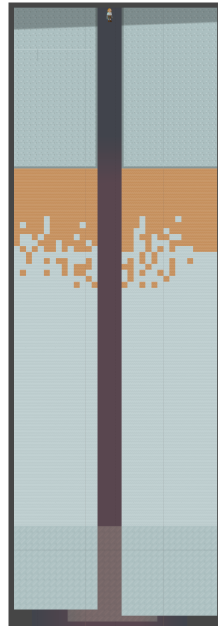

Figura 10.76: Illa dels Homes, scena 3.

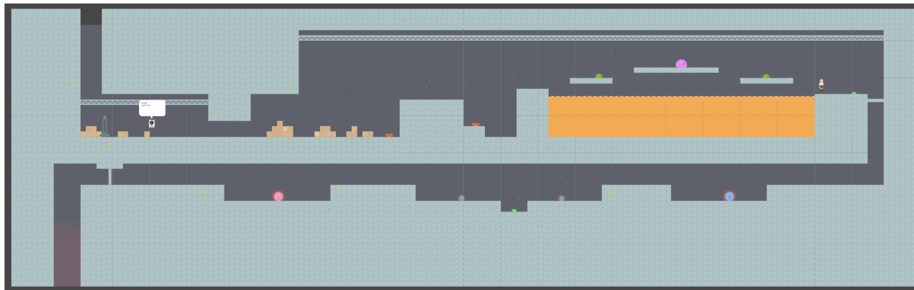

Figura 10.77: Illa dels Homes, scena 4.

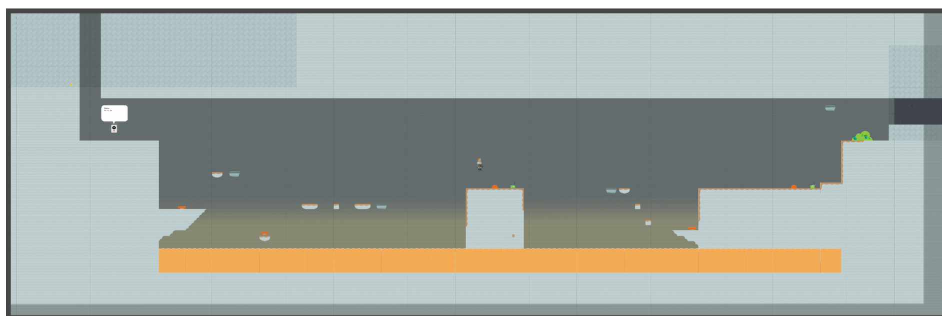

Figura 10.78: Illa dels Homes, scena 5.

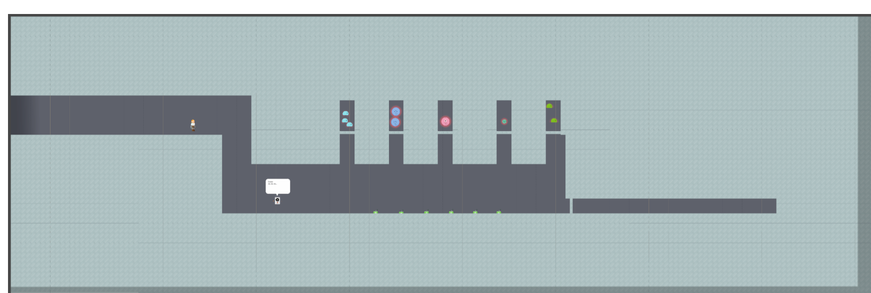

Figura 10.79: Illa dels Homes, scena 6.

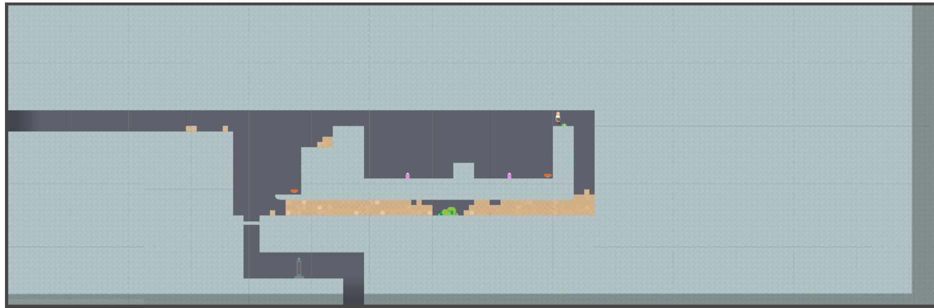

Figura 10.80: Illa dels Homes, scena 7.

Figura 10.81: Illa dels Homes, scena 8.

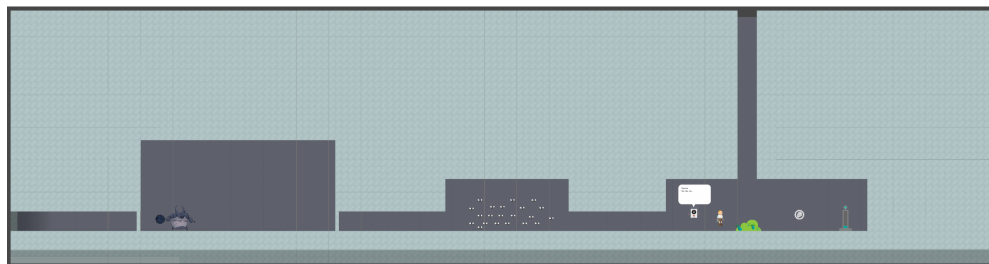

Figura 10.82: Illa dels Homes, scena 9.

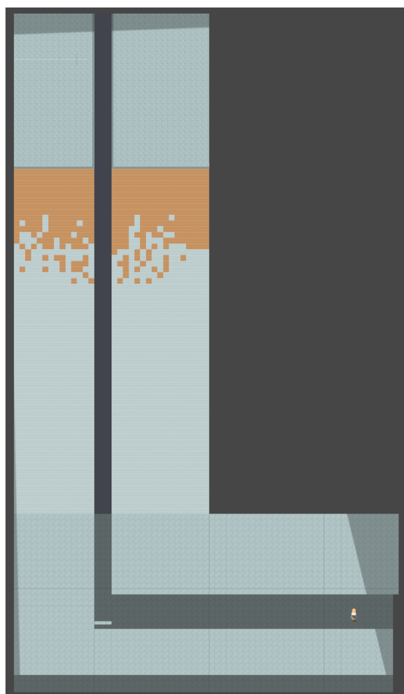

Figura 10.83: Illa dels Homes, scena 10.

CAPÍTOL 11
Conclusions

Un cop el projecte està tancat, podem concloure que:

- El temps és limitat. Tot i tenir una bona planificació de temps, la implementació de noves funcionalitats a vegades incorporen problemes, i aquests problemes modifiquin el temps estimat per funcionalitat.

- La gran majoria dels videojocs tenen un desenvolupament constant i indefinit. Aquest projecte, tot i complir amb els requisits funcionals i no funcionals establerts des d'un bon principi, es pot expandir immensurablement. Veure Apartat 12.

- La creació d'un videojoc requereix la presència de diferents perfils. Es necessita perfils tècnics i artístics per tal d'entregar un producte que, almenys funcionalment, visualment i sonorment, sigui un producte a gaudir.

- L'elecció de les eines que s'utilitzen en el desenvolupament han de ser les adequades. Un bon conjunt d'eines de desenvolupament ajuden en temps d'entregues i a la facilitat d'adaptar-se a aquestes.

- Cal destacar que el resultat del projecte proporciona una gran interactivitat amb elements d'escena, NPC, enèmics i interfícies d'usuari. Per completar el joc i gaudir de totes les seves funcionalitats, estimem uns 30 minuts de joc.

Com que el projecte ha estat pensat des de bon principi perquè sigui fàcilment distribuïble i jugable, vam seleccionar quinze persones que tenen experiència en el món dels videojocs; creadors, aficionats i jugadors, i els hi vam demanar que provessin el joc i ens donessin la seva opinió per saber en quin rumb s'ha d'encarar el projecte.

Per recopilar les diferents opinions els hi vam demanar que omplissin un formulari on s'avaluassin les diferents característiques del joc:

- **Història**
  Trama, desenvolupament de la narrativa, desenvolupament, construcció i presentació de personatges, coherència ludo narrativa.

- **Jugabilitat**
  Varietat i qualitat de les mecàniques implementades, sensacions a alhora de jugar.

- **Interaccions**
  Varietat i qualitat de les interaccions implementades, enemics, així com objectes de vida al mapa, trampolins, aigua, etc.

- **Apartat artístic**
  Disseny del mapejat, art del joc, coherència del mateix amb el món proposat.

- **Apartat sonor**
  Efectes especials i banda sonora.

Aquestes característiques s'avaluaven amb una puntuació entre 1 i 5, sent l'1 la nota mínima i 5 la màxima. El formulari també contenia un camp on els beta-testers podien deixar les seves opinions de forma més detallada sobre el joc.

	Historia	Jugabilitat	Interaccions	Apartat artístic	Apartat sonor
Mitjana	3.71	4.28	4.00	4.57	4.14

Taula 11.1: Mitjana de les valoracions dels 15 enquestats.

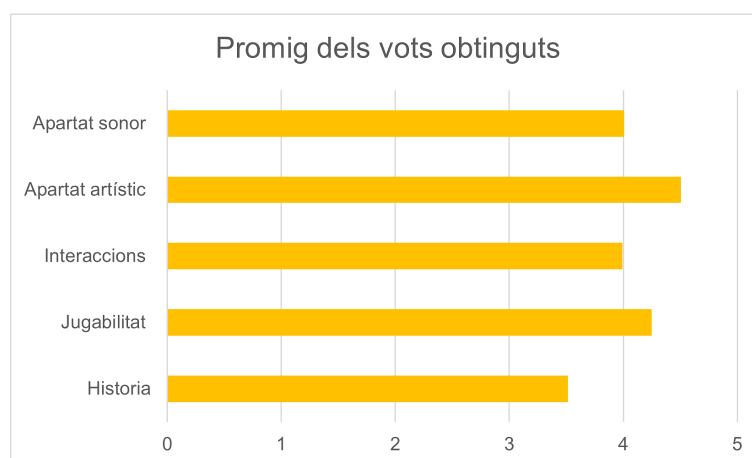

Figura 11.1: Gràfica de les valoracions

Com podem veure a la Figura 11.1 el joc ha obtingut resultats molt positius.

La nota més baixa ha estat a la història, un fet que ja ens esperàvem, ja que, en ser una DEMO, aquesta ha quedat molt reduïda. En les futures actualitzacions del projecte, la història tindrà un paper molt més important.

La nota més alta ha estat en l'apartat artístic. Cap dels integrants del grup ha cursat cap mena d'assignatura artística, així que ens esperàvem tenir una pitjor avaluació en aquest apartat. Així i tot, l'estètica única del joc ha causat un bon impacte.

Les interaccions s'han de millorar. Tot i no tenir mala nota, tant les interaccions com la jugabilitat són els apartats més crucials en un videojoc. Al cap i a la fi, no importa com sigui de bo el desenvolupament de la història, com sigui d'identificatiu l'art i com sigui d'emotiva la música, si un joc manca de jugabilitat, els usuaris es cansaran de les mecàniques i no l'acabaran.

A tall de tancament, aquest projecte ha estat un dels més grans que hem realitzat mai, i trobem que el desenvolupament ha estat entretingut. Hem pogut assolir els nostres objectius que teníem inicialment pensats i estem molt orgullosos amb el resultat final. Cal dir que tenim moltes ganes de seguir amb aquest projecte per tal d'expandir-lo i millorar-lo, i portar-lo al mercat.

Capítol 12

Treball futur

Trobem que aquest projecte és fàcilment ampliable ja que, mentre implementaven les funcionalitats dels requisits del projecte, sempre se'ns venien noves idees a implementar i que no estaven definides en els requisits. A més, clar està, que molt probablement es pot millorar l'actual implementació de lògica, sigui en rendiment o en simplicitat. A continuació farem un llistat de les possibles futures tasques i millores a tenir en compte.

- Ampliar la diversitat d'enemics explorant amb nous tipus de comportaments.
- Gestionar la presència de certs objectes en el mapa un cop s'han assolit els objectius. Per exemple, ara per ara, si derrotem a un Boss, i després sortim i tornem a entrar a l'escena, el Boss torna a sortir. Aquest és un comportament que és fàcilment gestionable però que a hores d'ara no ho està.
- Disenyar noves habilitats especials pel personatge, creant noves dinàmiques de joc.
- Crear un sistema d'elecció d'habilitats. D'aquesta manera el jugador podria escollir les habilitats que millor encaixen a la seva manera de jugar.
- Ampliar la història principal del joc. L'història actual del joc és en un rescat, però aquest realment es l'inici d'una història més ample que havíem escrit.
- Ampliar i millorar els recursos de so del joc.
- Gestionar més d'una sessió de joc a l'hora. El nostre projecte pot guardar com a màxim l'historic d'una sessió de joc.
- Afegir un mècanisme per tal de regular la dificultat del joc.
- Explorar nous mècanismes d'inteligencia artificial per implementar NPC i enemics. El projecte incorpora inteligencia artificial per arbres de comportament, però ens hagués fet molta gràcia, per exemple, implementar enemics amb tèniques de deep reinforcement learning fent ús del projecte de codi lliure de ml-agent [Unity 022f].

- Afinar algunes de les actuals animacions.
- Afegir un sistema de triomfs i com veure'ls (lligat a les notificacions, que ara es mostren, però no es guarden enlloc).
- Implementar un mapa per localitzar el personatge al llarg de les escenes.
- Afegir un sistema d'intercanvi de béns i serveis a través de monedes.

Bibliografia

[Asana 022] Team Asana. *Les 12 metodologíes més populars per la gestió de projectes*, (Accessed: Ago 2022). Available at https://asana.com/es/resources/project-management-methodologies. (Cited on page 19.)

[DOTween 022] DOTween. *A Unity Tween Engine*, (Accessed: Ago 2022). Available at http://dotween.demigiant.com/. (Cited on page 56.)

[EsotericSoftware 022] EsotericSoftware. *exports illustrator layers as individual PNGs*, (Accessed: Ago 2022). Available at https://raw.githubusercontent.com/EsotericSoftware/spine-scripts/master/illustrator/IllustratorToSpine.jsx. (Cited on page 64.)

[GameCI 022] GameCI. *docs*, (Accessed: Ago 2022). Available at https://game.ci/docs/github/builder. (Cited on page 71.)

[glassdoor 022] glassdoor. *Sueldos para el puesto de 2D Artist en España*, (Accessed: May 2022). Available at https://www.glassdoor.es/Sueldos/2d-artist-sueldo-SRCH_KO0,9.htm. (Cited on page 12.)

[Godot 022] Godot. *Scripting*, (Accessed: Ago 2022). Available at https://docs.godotengine.org/es/stable/tutorials/scripting/index.html. (Cited on page 35.)

[Jobted 022] Jobted. *Sueldo del Programador de Videojuegos en España*, (Accessed: May 2022). Available at https://www.jobted.es/salario/programador-videojuegos. (Cited on page 12.)

[Kenney 022] Kenney. *Free game assets*, (Accessed: Ago 2022). Available at https://www.kenney.nl/. (Cited on page 190.)

[Leak 022] Memory Leak. *Awesome UI animations with Unity and Spine 2D*, (Accessed: Ago 2022). Available at https://www.youtube.com/watch?v=YYxknBq06tQ&t=605s. (Cited on page 214.)

[Microsoft 022] Microsoft. *C# for Visual Studio Code (powered by OmniSharp)*, (Accessed: Ago 2022). Available at https://marketplace.visualstudio.com/items?itemName=ms-dotnettools.csharp. (Cited on page 45.)

[Opsive 022] Opsive. *Behavior designer*, (Accessed: Ago 2022). Available at https://opsive.com/assets/behavior-designer/. (Cited on page 56.)

[PCcomputer 022] PCcomputer. *False Knight sprites*, (Accessed: Ago 2022). Available at https://www.spriters-resource.com/pc_computer/hollowknight/sheet/132959/. (Cited on page 109.)

[promociónmusical 022] promociónmusical. *Compositor de Música para Videojuegos*, (Accessed: Ago 2022). Available at https://promocionmusical.es/salidas-profesionales/compositor-musica-videojuegos/. (Cited on page 12.)

[Silva 022] Kleber Silva. *Unity Code Snippets*, (Accessed: Ago 2022). Available at https://marketplace.visualstudio.com/items?itemName=kleber-swf.unity-code-snippets. (Cited on page 45.)

[statista 022] statista. *statista - Número de empresas de los principales países de la industria mundial del videojuego en 2021*, (Accessed: May 2022). Available at https://es.statista.com/estadisticas/714837/empresas-de-las-principales-industrias-del-videojuego-del-mundo/. (Cited on page 2.)

[TechTarget 022] TechTarget. *Advanced Encryption Standard (AES)*, (Accessed: Ago 2022). Available at https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard. (Cited on page 204.)

[Tokio 022] Tokio. *Sueldo de un diseñador de videojuegos: ¿cuánto cobra?*, (Accessed: May 2022). Available at https://www.tokioschool.com/noticias/sueldo-disenador-videojuegos-cuanto-cobra/. (Cited on page 12.)

[Unity 022a] Unity. *2D Sorting*, (Accessed: Ago 2022). Available at https://docs.unity3d.com/Manual/2DSorting.html. (Cited on page 192.)

[Unity 022b] Unity. *Introduction to Lighting and Rendering*, (Accessed: Ago 2022). Available at https://learn.unity.com/tutorial/introduction-to-lighting-and-rendering-2019-3. (Cited on page 53.)

[Unity 022c] Unity. *Introduction to Lights 2D*, (Accessed: Ago 2022). Available at https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@7.1/manual/Lights-2D-intro.html. (Cited on page 53.)

[Unity 022d] Unity. *License activation*, (Accessed: Ago 2022). Available at https://license.unity3d.com/manual. (Cited on page 71.)

[Unity 022e] Unity. *Lightweight RP*, (Accessed: Ago 2022). Available at https://docs.unity3d.com/2019.3/Documentation/Manual/com.unity.render-pipelines.lightweight.html. (Cited on page 195.)

[Unity 022f] Unity. *mlagent*, (Accessed: Ago 2022). Available at https://github.com/Unity-Technologies/ml-agents. (Cited on page 297.)

[Unity 022g] Unity. *Tilemap*, (Accessed: Ago 2022). Available at https://docs.unity3d.com/Manual/class-Tilemap.html. (Cited on page 189.)

[Unity 022h] Unity. *Unity2DGameKit*, (Accessed: Ago 2022). Available at https://assetstore.unity.com/packages/templates/tutorials/2d-game-kit-107098. (Cited on page 202.)

[Unity 022i] Unity. *UnityScript's long ride off into the sunset*, (Accessed: Ago 2022). Available at https://blog.unity.com/community/unityscripts-long-ride-off-into-the-sunset/. (Cited on page 34.)

[Unity 022j] Unity. *WebGL browser compatibility*, (Accessed: Ago 2022). Available at https://docs.unity3d.com/Manual/webgl-browsercompatibility.html. (Cited on page 41.)

[WIKIPEDIA 022] WIKIPEDIA. *Pixel art*, (Accessed: Ago 2022). Available at https://es.wikipedia.org/wiki/Pixel_art. (Cited on page 127.)

Annexos

En ser una memòria força extensa, hem preferit que el nostre Annex sigui el mateix projecte sencer d'Unity que s'ajunta a la documentació o bé a través del repository de GitHub del projecte el qual és públic. En el 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 els directoris del projecte perquè s'identifiquin fàcilment els diferents tipus de recursos:

- Els scripts es localitzen a la carpeta: Assets/Core
- Música, imatges, animacions, entre altres, es troben en la carpeta: Assets/Art
- Els prefabs es localitzen a la carpeta: Assets/Prefabs
- Els ScriptableObject es localitzen en la carpeta: Assets/ScriptableObjects
- Totes les escenes del joc es localitzen a la carpeta: Assets/Scenes
- Els recursos externs propis d'instal·lacions del Package Manager d'Unity o bé de fonts externes d'obtenció de recursos artístics gratuïts es troben en la carpeta: Assets/3rdParty

Manual d'Usuari

Iniciar el joc

El joc es pot jugar a través de dues plataformes.

1. **Web**
   Obrir Chrome, Firefox o Microsoft Edge i entrar al link:
   https://2wolfgames.github.io/Erlang-Legacy/

2. **Escriptori**
   Per iniciar el joc cal obrir la carpeta del projecte anomenada"Erlang-Legacy" i executar el fitxer "Erlang-Legacy.exe".

Seguidament, el joc s'iniciarà mostrant el logotip de Unity i posteriorment el menú principal.

Controls

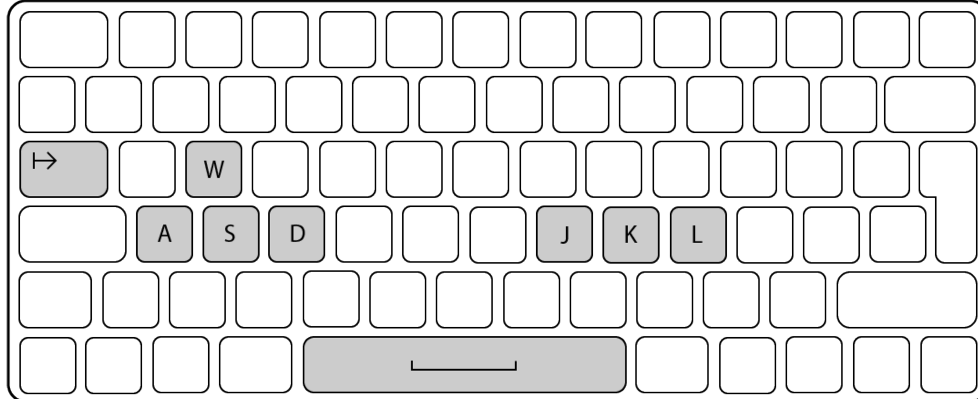

Figura 1: Tecles utilitzades per moure's dins del joc

Com podem veure a la figura 1 els controls es componen de poques tecles oportunament distribuïdes per no haver de moure la posició de les mans.

Controls en el menú

- Moviment vertical: Tecles W i S
- Moviment horitzontal: Tecles A i D
- Selecció: Tecla Space

Controls en partida

- Moviment horitzontal: El jugador es mou cap a la dreta i l'esquerra amb les tecles A i D
- Interacció: Tecla S
- Salt: Tecla Space
- Atac: Tecla J
- Dash: Tecla K
- Raig: Tecla L
- Obrir i tancar el menú: Tecla tab.

Objectiu

L'objectiu del joc és retrobar-se amb Cassandra, la germana de l'Ajax, el protagonista. Per aconseguir-ho farà falta superar els diferents enemics i puzles que ens presenta el món. En ser un metroidvania, també hi haurà zones opcionals: s'hi pot arribar si el jugador explora en més profunditat.

Menú de configuracions

Es pot accedir al menú de configuracions a través del menú inicial i del menú de pausa. Aquest deixa configurar el volum del so i la música.