Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Disseny i Desenvolupament de Videojocs

Títol: Disseny i desenvolupament d'un videojoc roguelike amb components de deck building

Document: Memòria

Alumnes: Miquel Sangrador Barcons, Blai Vilanova Macias

Tutor: Gustavo Patow

Departament: Informàtica, Matemàtica Aplicada i Estadística

Àrea: Llenguatges i Sistemes Informàtics

Convocatòria (mes/any) Setembre / 2022

Índex

1. Introducció

1.1. Introducció

Al mercat actual trobem una gran varietat de jocs tipus *Roguelike,* sobretot en referència als jocs indie, és a dir, desenvolupats per empreses amb pocs treballadors.

Aquest gènere és caracteritzat per un disseny de nivells basant en masmorres generades proceduralment i mort permanent dins cada partida. Als inicis, aquest gènere també es caracteritzava en combat per torns i moviment en graella, però la seva evolució ha donat pas a molts de videojocs d'acció en 2 dimensions, tant amb vista *Top-Down* (des de dalt) o bé amb vista lateral. I, a mesura que diferents categories de videojocs es combinen, apareixen noves propostes *Roguelike* amb mecàniques que solen corresponde a altres gèneres. Per exemple, moviment en sincronització amb la música, *Bullet Hell* (molts projectils en pantalla), mecàniques pròpies de jocs de cartes…

Si ens enfoquem en les mecàniques de combat d'aquests videojocs, es solen explotar en superfície: habilitats, diferents tipus d'armes, varietat d'enemics, etc. En canvi, en concret, hi ha videojocs que exploten al màxim aquest apartat introduint combinacions dins les pròpies armes, per aconseguir una gran varietat dins l'equipament del jugador i aconseguir el sentiment de descobriment de mecàniques de forma senzilla i freqüentment.

L'aposta més comuna sol ser fer combinatoria de objectes que afecten als projectils que el jugador dispara per a eliminar enemics, però no hem vist al mercat cap proposta enfocada en la combinació d'elements que modifiquin una arma cos a cos.

1.2. Motivacions

Així doncs la motivació principal del projecte és ocupar aquest espai dins l'indústria i veure les dificultats i beneficis que dona aquesta proposta.

També, pel que fa a nosaltres com a gaudidors d'aquest tipus de videojocs, hi ha moltes motivacions per a emprendre aquest projecte:

- El treball en conjunt en un projecte d'escala superior.

- Experimentar el procés de creació d'un videojoc.

- Posar en pràctica les habilitats adquirides durant el grau i veure com ens desenvolupem en un projecte major.

- Obtenir experiènica en el procés de decisió dins la creació d'un videojoc, debatre sobre aquests, i trobar les millors propostes.

1.3. Propòsit i objectius del projecte

El propòsit del projecte es crear un prototip d'un videojoc, incloent les diferents característiques del gènere *Roguelike* i l'exploració de la mecànica concreta exposada.

Els següents són els objectius que ens hem proposat durant el transcurs del projecte:

- Aprendre i testejar les capacitats del motor *Godot* i les seves característiques.
- Indagar en el disseny conceptual d'un videojoc i iterar sobre les idees inicials per aprofitar els avantatges de cada situació i treure'n benefici conjuntament.
- Guanyar experiència en la codificació de mecàniques complexes de forma entenedora i escalable.
- Guanyar experiència en la creació de nivells proceduralmente generats a nivell d'espai global, amb sentit i varietat.
- Aprendre els principis de l'animació clàssica en 2 dimensions.
- Experimentar el procés de creació d'estètica i animacions en 2 dimensions.
- Aprendre a dissenyar un moviment satisfactori pel tipus de joc realitzat.
- Gestionar la interacció entre els diferents elements que conté el projecte i comprendre com afecten al jugador.

1.4. Distribució de tasques

La distribució de tasques es va donar a l'inici al moment de veure els objectius del projecte i a mesura que s'anava avançant es va adaptar segons les possibilitats i preferències de cada integrant de l'equip. Ambdós membres de l'equip ens coneixem l'un a l'altre perfectament i hem tingut experiència en altres treballs de GDDV, per la distribució ha resultat una tasca senzilla.

1.4.1. Tasques comunes

Conjuntament hem realitzat les següents tasques relacionades amb la base del projecte i sobretot el disseny del mateix:

- Disseny i implementació de controls bàsics
- Disseny del moviment del personatge
- Disseny del combat del videojoc
- Testeig i depuració d'errors

1.4.2. Tasques d'en Miquel Sangrador Barcons

Estètica	30%
Narrativa	5%
Mecàniques	35%
Tecnologia	30%

El Miquel ha decidit centrar-se en mecàniques i a explorar també la part estètica i d'animació dins el projecte, ja que no havia tingut la oportunitat de explorar aquest camp en un projecte a gran escala i veure la magnitud de treball que comporta aquest apartat de cada videojoc.

El precentatge referent a l'estètica representa el treball dedicat al desenvolupament del personatge principal, efectes de post processat, objectes i escenaris, incloent les animacions.

El precentatge referent a la narrativa representa la creació del món associat al videojoc.

El percentatge referent a les mecàniques representa el disseny de les mecàniques bàsiques del videojoc i sobretot al de l'algoritme de generació procedural de masmorres.

El percentatge referent a la tecnologia representa la implementació en conjunt amb en Blai de les mecàniques bàsiques del videojoc i sobretot a la implementació de l'algoritme de generació procedural de masmorres.

1.4.3. Tasques d’en Blai Vilanova Macias

Estètica	15%
Narrativa	10%
Mecàniques	50%
Tecnologia	25%

El Blai ha decidit centrar-se en el disseny la interacció entre tots els elements que s’han creat dins del joc, ja siguin el jugador, amb el moviment i les habilitats que pot obtenir, com els enemics als que ha d'afrontar utilitzant aquestes habilitats.

El percentatge referent a l’estètica és el disseny i desenvolupament de diferents *Sprites* d’elements que apareixen a pantalla amb l’objectiu de diferenciar-los entre ells.

El percentatge referent a la narrativa és aquell que dóna sentit al joc i a com actuen els elements dins d’ell.

El percentatge referent a les mecàniques representa el disseny i la implementació dels diferents enemics que poden aparèixer en escena, així com els diferents efectes que pot adquirir el jugador i les interaccions entre ells. També es pot tenir en compte tota la part d’aleatorietat afegida referent a les possibilitats que toqui un enemic, o una habilitat mentre es juga.

El percentatge referent a la tecnologia representa la implementació en conjunt amb en Miquel de les mecàniques bàsiques del videojoc i, sobretot, a la implementació de l’algoritme de generació procedural d’enemics dins les masmorres, a part dels diferents menús amb els que el jugador interacciona amb el joc.

2. Estudi de viabilitat

2.1. Viabilitat tecnològica

El treball previ al començament del procés de creació és el de comprovar si projecte és viable dins del marc en el que treballem els dos integrants del grup, a més de si el videojoc té lloc dins del mercat en el que ens volem enfocar (jugadors de *Roguelike*).

L'objectiu final del projecte és que el joc pugui jugar-se amb PC, ja que creiem que és la indústria on podem trobar més públic objectiu. Segons aquesta notícia [LuisAvilés2021] podem observar que actualment l'ordinador és l'eina més utilitzada per jugar a videojocs, amb un 48% del total dels jugadors.

2.1.1. Hardware

Els dispositius utilitzats pel disseny i implementació del projecte són:

Blai

- Ordinador:
  - CPU: Intel Core i3-8100
  - RAM: 8 Gb
  - GPU: NVIDIA GeForce GTX 1060
- Comandament Xbox Series X
- Teclat Krom Kernel Mecànic

Miquel

- Ordinador:
  - CPU: AMD Ryzen 5 2600X
  - RAM: 16 Gb
  - GPU: NVIDIA GeForce GTX 1070
- Comandament Xbox Series X
- Teclat Krom Kernel Mecànic

2.1.2. Software

El software utilitzat a diferents parts del disseny i desenvolupament són els següents:

- Godot v3.3.4
- Github Desktop
- Photoshop
- Google Docs
- Lucidchart

2.2. Viabilitat econòmica

El principal motor amb el que es realitzarà aquest videojoc és *Godot*, el qual és gratuït, pel que no necessitarem cap despesa per utilitzar-lo. També s'utilitzarà *Github Desktop*, també gratuït, per gestionar les versions de cada un dels integrants del grup i ajuntar els progressos en un projecte comú.

Per la documentació s'ha utilitzat Google Docs per escriure la memòria i Lucidchart per fer els diagrames. Els dos programes són gratuïts, tot hi que el Lucidchart, en aquesta versió, només permet 100 elements als diagrames.

Per part artística el programa usat és *Photoshop* en la seva totalitat, per fer els *Sprites* estàtics i també animacions es pot adaptar *Photoshop*, amb un senzill plugin.

Plugin generador de *Sprite Sheets* [John Wordsworth2020]

2.3. Recursos humans

Aquest projecte és realitzat per dues persones que s'haurà d'encarregar de tot el procés de dur a terme aquest projecte. El projecte necessita aquests encarregats:

- Dissenyador de nivells - Miquel
- Programador de gameplay - Blai
- Programador d'UI - Blai
- Disseny de personatges - Miquel
- Animador - Miquel
- Tester - Blai i Miquel
- Depuració d'errors - Blai i Miquel

Ja que, dins el nostre grup, un dels integrants té un caire molt més enfocat a programar i un altre el té més aviat a dissenyar, en Blai s'enfocarà sobretot en les tasques de programació i en Miquel en la realització de la part d'estètica.

2.4. Públic objectiu i perfil del jugador

El perfil de jugador que busquem és el d'una persona jove o adulta (12 - 25 anys) familiaritzada amb aquest tipus de jocs, ja que els jugadors més *casuals* poden tenir problemes amb la corba d'aprenentatge dins aquest gènere.

Un dels avantatges dels *Roguelikes* és que cada partida que es juga dins del joc és diferent, pel que sempre passaran coses completament aleatòries i cap partida serà igual. A més, a diferència d'altres gèneres com els *MOBA (Multiplayer Online Battle Arena)*, aquest no obliga a quedar-se jugant fins que acabi la partida, ja que és una aventura d'un jugador. Això fa que l'usuari pugui pausar sempre que pugui, o si té 10 minuts lliures pugui intentar una *run* (partida ràpida).

2.5. Estudi de mercat

Abans d'enfocant-se en el procés de creació del videojoc en si, hem de veure com està el món dels *Roguelikes* en l'escena *gamer* actual. Per fer-ho hem de trobar els jocs amb les característiques més similars al nostre i comprovar si tenim aspectes distintius envers a ells.

2.5.1. Estat de l'art

Per comprovar l'estat de l'art, ens centrarem en els videojoc d'estil *Roguelike* més famosos dels últims anys.

Extret de l'article de [ManuDelgado2022].

- Hades

Hades és un videojoc estil *Roguelike* on es controla a Zagreo, el fill del propi déu Hades. El joc es basa en escapar de l'inframon utilitzant l'ajuda dels altres déus de l'Olimp, que ajudaran amb diferents benediccions que tindran diferents efectes segons el déu que les doni.

Figura 2.1. Gameplay del joc *Hades*.

Aquest és un dels jocs que més s'assemblen al que volem fer, ja que el jugador té armes de combat cos a cos i les benediccions aleatòries que utilitzen son molt semblants als efectes que volem utilitzar al projecte.

- The Binding of Isaac

The Binding of Isaac és un dels primers *Roguelikes* de l'era moderna. Va revolucionar la manera de veure els *Roguelikes* i va sentar un precedent per tots els jocs d'aquest gènere darrera seu.

Figura 2.2. Gameplay de *The Binding of Isaac*.

En aquest joc es controla a l'Isaac, un nen que ha d'escapar-se del soterrani de casa seva on l'ha tancat la seva malvada mare. Com podem veure a la **Figura 2.2**, el joc té una temàtica bastant macabra i així ho mostren el disseny dels personatges i ambientacions.

Aquest joc és un dels principals referents del nostre, ja que comparteix la manera de generació dels mapes, la vista, i alguns dels enemics que es poden trobar dins del joc.

- Rogue Legacy 2

La continuació de *Rogue Legacy 1*, un dels jocs més famosos del gènere, i amb una idea de mort bastant interessant i innovadora. En aquest videojoc hem de controlar un personatge que s'ha d'enfrontar a les diferents habitacions i enemics d'un castell embruixat. Quan aquest personatge mor, el jugador haurà de jugar amb un dels seus descendents, que poden tenir algunes habilitats o defectes diferents i aleatoris cada vegada que es mor com es mostra a la **Figura 2.3**.

Figura 2.3. Gameplay de *Rogue Legacy 2*.

El joc comparteix la idea de basar-se en les armes cos a cos, a més que té molts elements aleatoris que fan que l'experiència del jugador sigui canviant i divertida.

- Darkest Dungeon

Un dels jocs més complicats de l'actualitat podria ser *Darkest Dungeon*, un *Roguelike* basat en les clàssiques batalles per torns en un entorn *2D*. En aquest joc es controla un grup de personatges que entren a una mazmorra plagada d'enemics. Per sobreviure, s'ha d'utilitzar els pocs recursos que es té per evitar que els personatges es morin abans de superar el nivell.

Figura 2.4. Gameplay de *Darkest Dungeon*.

Com podem veure a la **Figura 2.4**, aquest joc, tot hi ser un *Roguelike*, és una idea molt diferenciada del que nosaltres volem fer, ja que, com a molt, compartim la idea del propi gènere i la de fer partides aleatòries cada vegada que es juga.

- Don’t Starve

Un altre joc aclamat per les crítiques és *Don’t Starve*, joc fet per una empresa Indie que s’aventura a barrejar dos estils tan interessants com el *Roguelike* i la supervivència. En aquest joc es controla un personatge que ha de sobreviure en un mon hostil i fosc.

Figura 2.5. Gameplay de *Don’t Starve*.

Al ser un joc de gènere supervivència, aquest no es basa en escapar ni superar una masmorra, sinó que és un món obert, on la part de *Roguelike* està en la generació del món. A més, com podem veure en la **Figura 2.5**, el joc té una estética estil Tim Burton molt interessant.

2.5.2. Comparació en el mercat

Els jocs mostrats anteriorment són realment interessants i innovadors dins el gènere del *Roguelike*, però creiem que cap introdueix l'enfocament que li posem nosaltres. Sí que és cert que hi ha algun dels mostrats que es centren en el combat cos a cosa, com l'*Hades* i el *Rogue Legacy 2*, però cap dels dos utilitzen la vista que utilitzem nosaltres, *Hades* utilitza una vista isomètrica mentre que amb *Rogue Legacy 2* és una vista *2D*.

Com a jugadors i dissenyadors de videojocs, podem assegurar que un fet com aquest fa que un joc com el que volem crear sigui completament diferent al que fa referència a l'experiència de joc.

Si ens mirem el joc que comparteix la vista amb el nostre, *The Binding of Isaac*, podem veure que aquest sí que es centra molt més en el combat a distància, cosa que fa que sigui completament diferent al nostre, ja que l'estil de lluita entre els dos variarà molt.

2.5.3. Model de negoci

Hi ha moltes maneres d'aproximar-se al mercat dels videojocs, les més famoses de l'actualitat són:

- Clàssica: es paga el joc una vegada i pots disfrutar de tots els continguts del mateix. Ex: *Sekiro*
- *Free-to-Play*: joc completament gratis que utilitza els anuncis per monetitzar-se. A vegades també s'utilitza un sistema de *skins* per vendre al jugador (diferents vestits pel teu personatge que no afecten al joc). Ex: *League of Legends*
- Sistema de *DLC*: un cop venut el joc inicial (normalment utilitzant la venta clàssica) s'afegeix contingut adicional per augmentar les hores de joc. Ex: *Far Cry 6*
- Subscripcions: El jugador ha de pagar cada mes si vol seguir jugant al joc. Ex: *World of Warcraft*
- Freemium: Jocs suposadament gratuïts en els que s'ofereix la opció de comprar elements que donen avantatges per sobre del jugador que no compra res. Ex: *Dragon City*

Un cop vist totes les maneres que es poden vendre els jocs, creiem que, al ser un *Roguelike* al que sempre se li poden anar afegint actualitzacions amb nou contingut, ja siguin enemics o habilitats, creiem que hauríem de poder vendre'l de manera Clàssica, però anar afegint actualitzacions gratuïtes pel jugador que l'ha comprat. Si en algun moment volguéssim crear una actualització massiva on portar molts canvis al joc, podríem arribar a fer un *DLC*.

3. Planificació

3.1. Tasques planificades

- Redacció i preparació de la memòria
- Planificació del projecte
- Recerca de l'estat de l'art
- Estudi de viabilitat
- Selecció de l'entorn de treball
- Disseny de mecàniques de combat
- Disseny de nivells
- Disseny d'enemics
- Implementació del personatge
- Implementació dels enemics
- Implementació de la generació de niells
- Implementació de les animacions
- Disseny de personatge
- Animació de personatge
- Construcció de l'ambientació
- Depuració d'errors
- Testing

3.2. Blocs de treball

Documentació:

- Taques:
  - Redacció i preparació de la memòria
- Temporització:
  - Al llarg del projecte
- Resultat:
  - Memòria

Recerca prèvia:

- Tasques:
  - Planificació del projecte
  - Recerca de l'estat de l'art
  - Estudi de viabilitat
  - Selecció de l'entorn de treball
- Temporització:
  - 1ª setmana Setembre 2021 - 3ª setmana Setembre 2021
- Resultats:
  - Documentació

Mecànica:

- Tasques:
  - Disseny de mecàniques de combat
  - Disseny de nivells
  - Disseny d'enemics
- Temporització:
  - 4ª setmana Setembre 2021 - 4ª setmana Octubre 2021
- Resultats:
  - Documentació

Estètica:

- Tasques:
  - Disseny del personatge
  - Animació del personatge
- Temporització:
  - 2ª setmana Abril 2022 - 4ª setmana Juliol 2022
- Resultats:
  - *Sprite Sheets* d'animacions *2D*

Narrativa:

- Tasques:
  - Construcció de l'ambientació
- Temporització:
  - 1ª setmana Desembre 2021 - 3ª setmana Desembre 2021
- Resultats:
  - Documentació

Tecnologia:

- Tasques:
  - Implementació del personatge
  - Implementació dels enemics
  - Implementació de la generació de niells
  - Implementació de les animacions
- Temporització:
  - 1ª setmana Novembre 2021 - 4ª setmana Juliol 2022
- Resultats:
  - Projecte de *Godot*
  - Escenes de *Godot*

Depuració:

- Tasques:
  - Depuració d'errors
  - Testing
- Temporització:
  - 1ª setmana Febrer 2022 - 4ª setmana Juliol 2022
- Resultats:
  - Captures i gravacions del projecte de *Godot*

3.3. Metodologia de treball

La realització del treball ha estat assolida usant *sprints*:

- Cada reunió amb el tutor del treball es defineix quines són les tasques les quals realitzarà cada integrant del grup fins la propera reunió. Així s'aconsegueix un progrés constant i no una data d'entrega generalitzada, una *deadline*.

- A la reunió s'exposa el treball realitzat durant l'interval entre aquesta i la última.

- A part d'aconseguir el *feedback* del tutor, a les reunions es mostren les dificultats que han sorgit durant les setmanes, si és que n'hi ha, i es busca la millor solució conjuntament.

L'ordre de la realització de les tasques del projecte de *Godot* ha vingut donada per les necessitats del projecte, és a dir, s'ha iniciat el projecte amb les mecàniques bàsiques del moviment i llavors s'ha construït a sobre per anar generant contingut, per així sempre tenir un mostra jugable de l'estat del projecte.

A part de la metodologia de treball general, els integrants del grup han estat en contacte en tot moment i han observat els progressos generats en l'interval entre reunions amb el tutor. D'aquesta manera s'afavoreix la cooperació el qual és el dia a dia en estudis petits de desenvolupament de videojocs.
A més a més la majoria de les decisions de disseny s'han comentat i decidit com a grup. Així s'aconsegueix la satisfacció del grup de treball en les decisions i aquestes es posen a prova per més d'una persona.
No obstant això, sempre s'ha tingut en compte la visió general de projecte i la direcció establerta prèviament.

4. Marc de treball i conceptes previs

4.1. Conceptes previs

Aquí explicarem conceptes previs inherents al gènere del videojoc que ens adrecem al treball, els *Roguelike,* i al buit que volem emplenar dins l'indústria.

- Permadeath

La mort permanent és un dels aspectes clau d'aquest gènere. Com bé indica, no es tenen una quantitat de vides limitades com a la majoria de jocs, sinó que quan es perd, es torna al principi de tot.

Aquesta mecànica, que als inicis del gènere era així es va anar explorant i donant quantitat de utilitats al fet de perdre partida, a part de la informació obtinguda durant la mateixa.

Enter the Gungeon, per exemple, utilitza unes monedes especials que s'aconsegueixen durant la partida per poder obtenir bonificacions o nous personatges al acabar partida.

Moonlighter, en canvi, utilitza la partida com a la fase d'obtenció de recursos per després passar a un espai de gestió de recursos on l'objectiu és vendre'n el màxim per obtenir equipatge millor per a la pròxima incursió i obtenció de recursos.

Per descomptat, cada vegada que es mor, es perden tot els recursos obtinguts i es torna a començar la incursió des del principi, si no s'ha marxat abans amb un objecte especial per no perdre els recursos.

Figura 4.1. Botiga de *Moonlighter*, s'observen les monedes que s'obtenenpels objectes a la venta, que llavors serveixen per millorar l'equipatge.

SIFU, en canvi, utilitza la mort com a progressió; cada vegada que el perd, el personatge es fa més vell i amb això perd la capacitat de obtenir habilitats, però els atacs fan més mal, fins a arribar un punt on s'acaben els anys i es perd de veritat i es torna a començar de jove.

Figura 4.2. Evolució del personatge a mesura que avancen els anys a *SIFU*.

Figura 4.3. Suma d'edat en morir a *SIFU*, a la dreta es veu com està pujant el mal mentre baixa la vida.

- Deck Building

La construcció de baralles es el nucli de la mecànica de combat en molts de videojocs de cartes. Es tracta de triar una sèrie de cartes per abordar les combats que hi haurà durant la partida.

Aquesta mecànica representa el progrés dins la partida i també és tota l'estratègia prèvia als combats, durant la construcció de la baralla es proposa un repte al jugador, la configuració d'aquesta perquè pugui superar qualsevol dels reptes que es proposen d'ara en endavant fins al final de partida.

També cal comentar que la fase on s'usen aquestes cartes i es roben de la baralla en ordre aleatori es una fase important, si s'està jugant a un joc de cartes, però en el nostre joc no té influència, ja que l'acte de combat és d'acció clàssica i no per torns o jugant cartes.

El *deck building* al nostre videojoc és representat per la construcció de l'espasa del protagonista, on cada carta és un tros de l'espasa que té un efecte en el combat del jugador. Llavors, és una innovació total ja que no hi ha cap videojoc actual que faci ús de components de *deck building* en relació al combat cos a cos.

En la tria de cartes se'n diferencien 2 tipus de jocs:

1. Totes les cartes es trien al primer moment abans de començar la partida, com per exemple als jocs multijugador com *Hearthstone* o de cartes físiques com *Magic: The Gathering*.

Figura 4.4. Deckbuilding a *Hearthstone*, tria de 30 cartes amb màxim de 2 còpies de carta individual

2. Les cartes es trien com a recompensa dels combats i altres events dins el videojoc, com per exemple als jocs *Slay the Spire* o *Loop Hero* (construcció d'escenari en comptes de baralla).

Figura 4.5. Deckbuilding a *Slay the Spire*, tria de cartes (1 de les 3 o bé cap) al haver acabat un combat

Figura 4.6. Deckbuilding a *Loop Hero*, aquí les cartes no es trien, sinó que s'aconsegueixen al final de cada combat aleatòriament, però sí es coloquen dins al tauler

- Generació procedural de nivells

La gran majoria de videojocs dins el gènere *Roguelikes* utilitzen algun tipus de generació procedural de nivells. Cadascun té una generació de nivells diferents, però els podem englobar en 2 categories: nivells basats en sales i generació de l'escenari en la seva totalitat.

1. Nivells de sales, és el cas de jocs com *The Binding of Isaac*, *Darkest Dungeon* i *Enter the Gungeon*, entre molts altres. Aquests basen els nivells en la disposició de les sales interconnectades (sales predefinides, tant enemics com objectes), aquesta connexió és diferent a cadascun dels jocs però compleix el mateix propòsit.

A *The Binding of Isaac* es pot veure les portes que conecten a les sales contigües. Les sales són de mida variada, però sempre utilitzen quadrats com a base per construir cada sala per poder disposar-les entre sí.

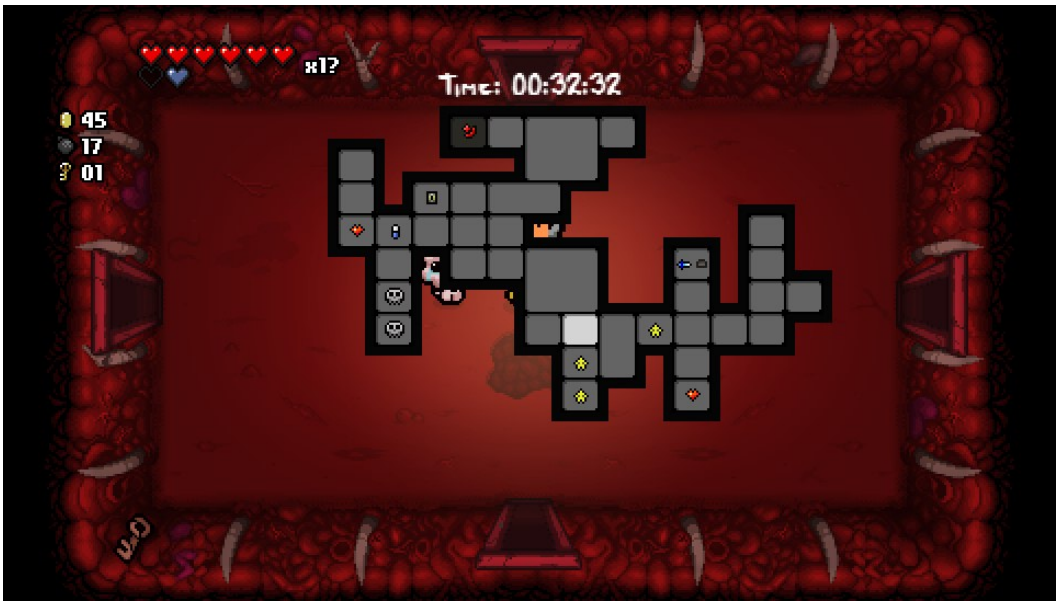

Figura 4.7. Mapa a *The Binding of Isaac*.

En canvi, a *Enter the Gungeon* es pot veure les sales de mida totalment variada i connectades a partir de passadissos. Aquests són més llargs o petits depenent de la distància que hi ha entre les sales.
A més a més, ens dona informació sobre la construcció de la sala, en canvi l'anterior joc no ho feia.

Figura 4.8. Mapa a *Enter the Gungeon*.

2. Tot el món està generat proceduralment, per exemple, en jocs com *Noita* o *Downwell*. Tot i així, els nivells d'aquest videojocs solen tenir zones d'interès que són introduïdes dins la generació de manera homogènia.

Figura 4.9. Generació procedural del món a *Noita*, s'observa una zona d'interès a la part esquerra

Les zones d'interès a *Noita* no són generades proceduralment, sinó que són moderades a mà. En canvi, sí que està colocada a la sobre la generació procedural que es veu a la part dreta.

Downwell és un altre exemple de generació de món, però amb una combinació dels 2 mètodes anteriorment explicats, aquí les sales són trossos de l'escenari que s'adjunten verticalment per crear un nivell de distància determinada sense cap mena de pausa, on l'objectiu del jugador és arribar al fons.

El canvi entre habitacions en comptes de ser un passadís o una porta, ara és simplement un espai buit de paret vertical sense res, i com totes les sales del nivell són començades i acabades d'aquesta manera, el resultat és que no hi ha canvi entre les solesi aparenta ser un nivell totalment generat proceduralment.

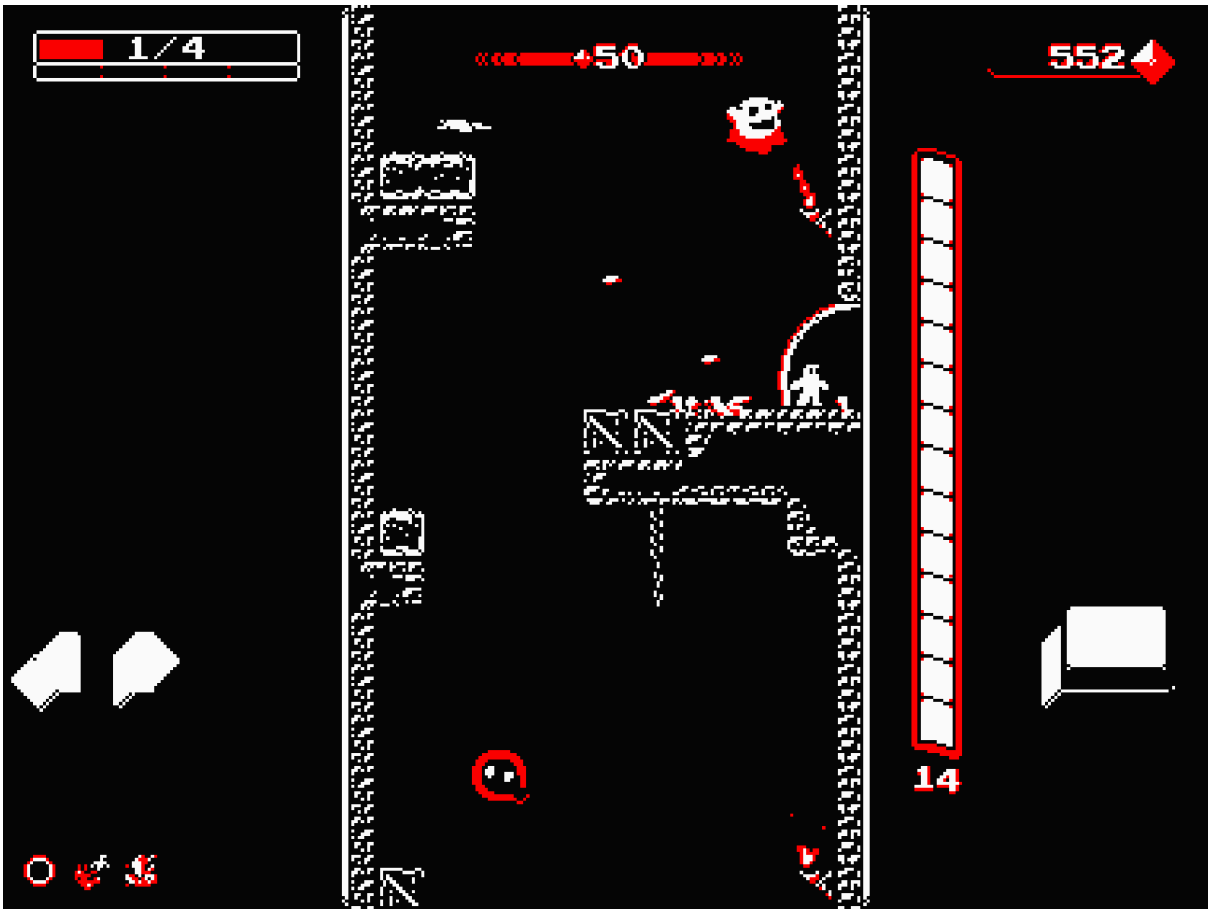

Figura 4.10. Generació procedural del món a *Downwell*.

4.2. Característiques pròpies de Godot

Aquí s'expliquen el vocabulari i les característiques de *Godot*, per poder començar amb la implementació de les diferents parts del projecte.

- Node

Els nodes són els blocs de construcció a *Godot*, poden ser assignats a fills d'altres nodes i cadascun d'ells pot tenir infinits fills, en una estructura d'arbre. Estan organitzats per categories en una jerarquia, però bàsicament s'usen nodes de la branca *CanvasItem*, que està separada en nodes de *Control* i *Node2D*, els primers més enfocats a interfícies i els segons a personatges.

- Scene

Una agrupació de nodes és una escena, es poden fer escenes de la més petita fins a la més gran. Per exemple, una fletxa es una escena per poder-la instanciar més fàcilment i no haver de posar-hi tots els components cada vegada que se'n vol crear una, però també una escena és un nivell sencer. Al ser senzillament una agrupació de nodes, una escena pot tenir altres escenes com a fills.

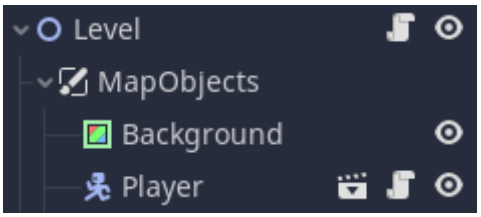

Figura 4.11. Jerarquia de nodes a *Godot*, s'observa el node *root* pare de totes els nodes de l'arbre i l'escena *Player*, que s'indica que es escena amb el símbol de claqueta

- GDScript

Aquest és el llenguatge propi de *Godot*. D'alt nivell i dinàmicament tipat, és molt semblant a *Python* ja que es basa en indentació per a la separació de blocs i moltes de les paraules clau també són iguals.

- Editor

L'editor de *Godot* és una eina interna del propi motor, no és necessària la instal·lació de un altre editor per a treballar en el programa, cosa que el fa més veloç al obrir-se i no comporta problemes de compatibilitat. Encara que sempre es pot decidir utilitzar un altre editor preferit, canviant-lo a les opcions del programa.

- Layers

El godot té un sistema d'estructuració d'elements molt interessant. Aquest sistema utilitza *Tags*, que serveixen per etiquetar els diferents fills que té cada escena del projecte. Per fer-ho, primer s'han de crear els grups dins de les opcions del projecte:

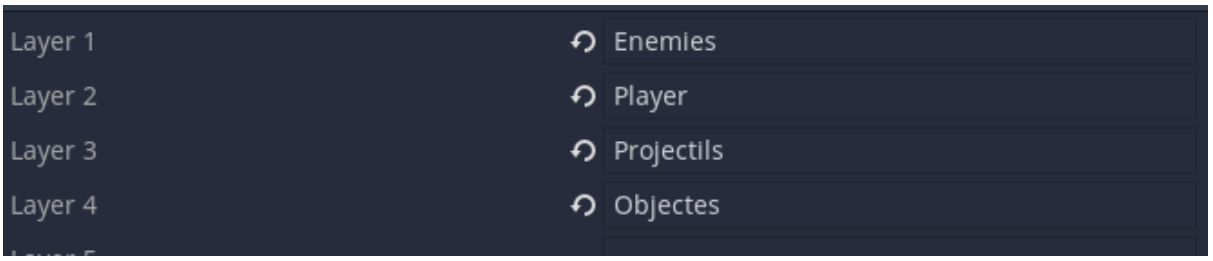

Figura 4.12. Les *layers* que té el projecte i a les que s'etiqueten els fills.

Com podem veure a la **Figura 4.12,** tenim quatre etiquetes al projecte: Enemics, Jugador, Projectils i Objectes. Cada una d'elles ha estat assignada al seu grup a través del seu inspector.

Tenir aquestes agrupacions és extremadament útil per evitar interaccions d'elements no desitjades, com per exemple que l'àrea de l'atac de l'espasa detecti un efecte com a element que rebi mal, cosa que no ha de passar mai.

Figura 4.13. Part de l'inspector referent a les col·lisions d'un element.

Per controlar aquests problemes potencials, Godot compta amb un inspector on es pot indicar a quin grup pertany el fill (apartat *Layer* de la **Figura 4.13**) i amb quins elements pot interactuar (apartat *Mask* de la **Figura 4.13**). A més, també es pot referir-se a aquestes *Layers* a través del codi de cada element, com es mostrarà més endavant en l'apartat d'enemics.

- Signal

Els senyals o *signals* són els missatges que els nodes emeten quan passen events en concret, aquestes són instanciades pels propis nodes de godot o es poden afegir per codi.

Per exemple un botó de Godot té un signal *pressed()* que s'activa quan el botó és pressionat i llavors es connecta a una funció de nom customitzable de un script.

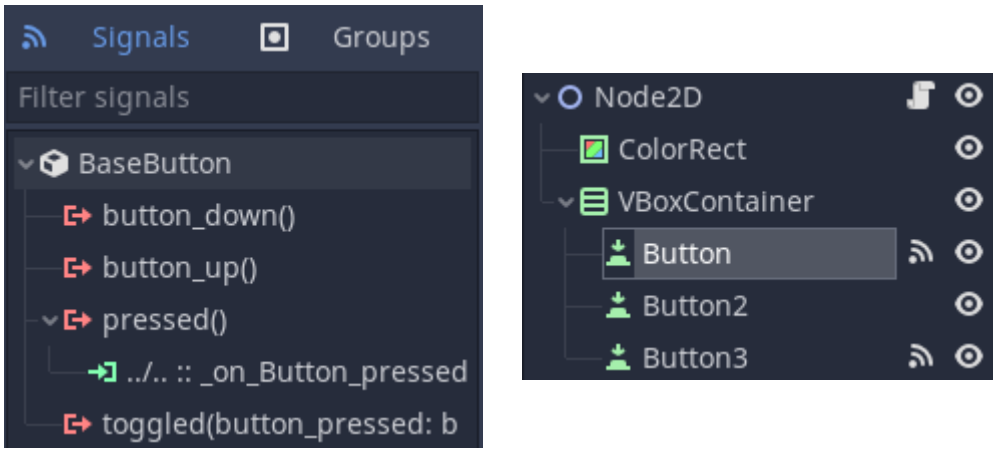

Figura 4.14. *Signals* de *Godot*, s'observa el senyal *pressed()* connectada a *_on_Button_pressed()* a l'*script* que conté *Node2D*

D'altre banda si es vol crear un signal per codi, es pot fer de la següent manera:

```
Script nº1 -> emisor del senyal

#Creació del signal
signal signalName

#Emissió del signal amb els arguments args
emit_signal("signalName", args)

#Connexió del signal per codi
connect(signalName, nodeName, 'functionName')
```

- Funció Process

La funció *_process(delta)* i *_physics_process(delta)* són els bucles a *Godot*, aquestes contenen la variable delta que indica el temps que ha passat entre execució i execució de la funció.

Per una banda *_process(delta)* s'executa una vegada per *frame* i en canvi *_physics_process(delta)* s'executa a una velocitat concreta 60 vegades per segon, per defecte.

4.3. Singleton

Tots els motors de videojocs necessiten una eina per referir-se a les variables globals, dins de *Godot*, aquesta eina és el Singleton. Per activar-lo, s'ha de modificar un *script* creat dins de les opcions del projecte. Un cop fet això, aquest *script* es podrà referenciar des de tots els altres codis creats al projecte. En el nostre cas, aquest *script* es diu Global.

Aquesta eina és molt útil per guardar elements que seran fixes en totes les escenes de *Godot*. Un exemple molt clar d'això seria la vida del jugador, una variable de tipus int que s'ha de controlar des de l'escena jugador però que, si s'ha de comprovar des d'alguna altre escena, l'hem de tenir guardada dins del Singleton.

Un altre part molt interessant del Singleton és que no només podem afegir variables, sinó que també podem cridar funcions des d'ell. Això vol dir que, si en algun moment s'ha de cridar una funció des de vàries escenes diferents, no s'haurà de repetir codi, sinó que només caldrà afegir-la al Singleton i cridar la funció en cada una de les escenes pertinents.

Global.gd

```
var Efectes = [["c_StableDamage",10], ["Double_Dash",20],
["root",20], ["vampire",15], ["retro",20], ["slow",15]]
var ProbTotal = 100
var PlayerPos = Vector2.ZERO # Posició del Jugador (dins la
sala)
var PlayerDamage = 1 # Dany del jugador
var MapPos = Vector2.ZERO # Posició del Jugador (dins el
minimapa)
var SwingTime = 0 # Temps que tarda el jugador a atacar
var PlayerHealth = 5 # Vida del jugador
var SwordSize = 70 #Tamany inicial de l'Espasa
var PlayerEffects = [] # Habilitats que té el jugador
var Disable = false
var rng # Llavor de l'aleatorietat del joc
var NumEnemics = 0 # Num d'enemics a pantalla
var RoomsVisited = Array() # Habitacions que el jugador ha
visitat
var BossRoom # Habitació on apareix el Boss final
```

Com podem veure en el codi anterior, dins del Singleton hi tenim variables de tot tipus, cada una utilitzada per varis elements del joc. Aquestes variables aniran variant durant les diferents partides que jugui l'usuari.

Dins d'aquest script també tenim algunes funcions que s'utilitzaran durant el desenvolupament de la partida. Aquestes s'explicaran en el seu punt propi seguidament.

4.4. Entorn de treball

La decisió de l'entorn de treball comú va ser ràpida i senzilla ja que sabíem les necessitats del projecte i l'abast que volia recollir. A part, els 2 estem familiaritzats amb el programari utilitzat i ens vam posar d'acord en poc temps.

Tenint en compte les següents característiques bàsiques del projecte hem decidit els entorns de treball:

- Videojoc en *2D Top-Down View*
- Algunes pròpies amb implementació original
- L'abast del projecte és un prototipus del videojoc final

4.4.1. Motor

Al analitzar les necessitats del projecte i les diferents possibilitats de motors, que hi ha disponibles de manera gratuïta i amb els que estem familiaritzats, les opcions eren: *Unreal Engine*, *Godot* i *Unity*.

Hem decidit *Godot* a causa de la fàcil implementació en *2D,* la qual eliminava *Unreal Engine* ja que és més enfocat a *3D* i el llenguatge a base de *Blueprints* no hi estem tan acostumats. Per últim, *Unity,* trobem que per projectes petits i amb poques persones a vegades dóna molts problemes amb *GitHub* i la velocitat d'obertura del programa. A més a més que les característiques de *Godot* explicades a l'**Apartat 6.1** ens atrauen més i hi tenim més experiència que amb les pròpies de Unity.

Figura 4.15. Logotip de *Godot*.

4.4.2. Disseny artístic

Photoshop ha estat la opció des del principi a causa que, en Miquel, l'encarregat principal de l'apartat artístic del projecte, és el programa amb el que més familiaritzat està i el que té una versatilitat per a generar els *Sprite Sheets* necessaris. Deixant de banda que ja el tenia en la seva possessió abans de començar el projecte.

Figura 4.16. Logotip de *Photoshop*.

4.4.3. Documentació

La documentació s'ha realitzat en la seva totalitat usant les eines gratuïtes de *Google*, *Google Drive* i *Google Documents*.

També s'ha utilitzat *Lucidchart* per tot el contingut de la memòria relacionat amb diagrames i esquemes visuals.

Figura 4.17. Logotip de *Google Documents* (esquerra) i *Lucidchart* (dreta).

5. Disseny del videojoc

5.1. Mecàniques del jugador

La principal característica que diferencia al joc de qualsevol altre és com gestionem les habilitats que pot anar obtenint el jugador a mesura que elimina a enemics. Aquest apartat del disseny va passar per molts passos fins arribar el estat final, ja que entre els membres del grup vam tenir vàries discussions sobre com aplicar la llargada de l'espasa al joc de manera que penalitza al jugador, i així poder jugar amb els efectes que l'usuari tenia a mà.

Inicialment, es volia fer que la llargada de l'arma fos un estat, és a dir, que el jugador tingués l'opció de allargar o disminuir l'arma en mig del combat (estil *Bloodborne*). Veure **Figura 5.1**.

Figura 5.1. Arma *Saw Spear* de *Bloodborne*, s'observa la comparació entre mida gran i petita, es canvia usant el botó *LB* del controlador

Aquesta era una idea interessant en la que indagar, ja que es faria que, al tenir una espasa més grossa, aquesta atacaria més lent, però amb molt més rang, mentre que l'espasa petita atacaria ràpid a menys distància. Però vam creure que haver de cambiar l'arma en mig del combat podria ser complicat amb tots els elements que es troben per pantalla. A causa d'això, vam creure necessari modificar aquest apartat del disseny.

Vam arribar a la conclusió que seria molt més senzill d'entendre i d'implementar que cada cop que s'agafa un efecte, sigui en un moment en el que no hi hauria enemics per pantalla. A causa d'aquest canvi, era impossible aplicar el canvi de la llargada estil *Bloodborne* que volíem fer.

Finalment, vam concluir que el millor seria tenir una llargada lligada a cada efecte, de manera que, com més efectes tingui el jugador, més lent ataca (cosa que pot anar bé o malament davant dels enemics que li surten).

El fet de fer el disseny de l'arma d'aquesta manera, va fer que també s'hagués de plantejar la gestió d'efectes, i una manera dins del joc de eliminar-los des d'un menú accessible durant la partida.

Un altre mecànica en la que es basen una gran majoria dels videojocs actuals és la d'utilitzar un *dash,* un desplaçament ràpid que pot fer el jugador i que s'utilitza per posicionar-se millor en una sala o esquivar algún atac debastador, ja que utilitzar aquest moviment el torna invulnerable durant uns segons. Ja que és una mecànica molt beneficiosa pel jugador, aquesta té un temps de recàrrega.

5.1.1. Espai de joc

5.1.1.1. Espai local

L'espai local, al ser totes les sales de la mateixa mida, es gestiona a partir dels enemics que són a cadascuna de les sales. A cause de l'aleatorietat dels enemics a cada sala, vam decidir afegir unes escenes que ens servissin com a plantilla del que pot sortir quan s'obre una porta. Aquestes plantilles tenen diferents enemics instanciats, i entrar, pot tocar una plantilla o una altre a partir d'un número aleatori generat per Godot.

5.1.1.2. Espai Global

L'espai de joc global estava clar des del principi que hauria de ser gestionat per una generació procedural del mapa amb diferents tipus de sales. Abans de veure els algoritmes, es va triar la mida aproximada del nivell generat, d'entre unes 20 i 30 sales incloent la sala inicial on no hi ha enemics i la sala de l'enemic final.

5.1.2. Accions

Aquí trobem les accions que pot portar a terme el jugador per superar els reptes que li presenta el videojoc:

- Moure's
- Atacar
- Usar el *Dash* (moviment ràpid)
- Recollir efecte
- Eliminar efecte

5.1.3. Jerarquia de reptes

La jerarquia de reptes és senzilla a causa dell gènere del joc Roguelike que es basa en la repetició per arribar a un objectiu millor.

Completar nivell

- Completar sales bàsiques
  - Eliminar enemics de la sala
  - Recollir efecte
- Completar sala de enemic final
  - Eliminar enemic final

5.1.4. Objectes, recursos i atributs

Personatge

- Vida [0, …, 10]
- Allargada espasa [70, …, ∞)
- Efectes [efecte1, …, efecteN]
- SwingTime [1, …, ∞)
- Mal [1, 2]
- Posició de sala [X: …, Y:...]
- Posició de mapa [X: …, Y:...]

Enemics

- Vida [0, …, 10]
- Posició de sala [X: …, Y:...]

Efecte

- Nom [0, …, 10]
- Augment de llargada [20, 60]
- Probabilitat [10, 15, 20]

5.1.5. Economia interna del joc

L'economia del joc és representada amb la vida i la mida de l'espasa.

Sources

- Vida

A l'*Spawn* del personatge, obté 5 punts de vida. Si s'obté l'efecte vampir, cada enemic eliminat farà augmentar la vida del personatge principal 1 punt.

- Mida de l'espasa

Quan s'obté un efecte, aquest té una llargada associada i fa créixer l'espasa.

Drain

- Vida

La vida es perd cada vegada que el jugador es colpejat per un enemic, es perd 1 punt de vida.

- Mida de l'espasa

Quan s'elimina un efecte, aquest té una llargada associada i fa decréixer l'espasa.

5.2. Disseny de nivells

5.2.1. Sales bàsiques

Les sales bàsiques contenen enemics escollits aleatòriament. Aquests van estar dissenyats segons com poden afectar més al jugador, basant-se també en els enemics més comuns d'aquesta tipologia de jocs.

5.2.1.1. Enemics bàsics

- *Archer:* L'enemic a distància del joc. El típic arquer que dispara fletxes cada dos segons. Aquest enemic s'ha creat per molestar al jugador amb l'atac a distància. Amb l'obligació d'acostar-se a atacar, el jugador ha d'esquivar les fletxes que dispara mentre ataca.

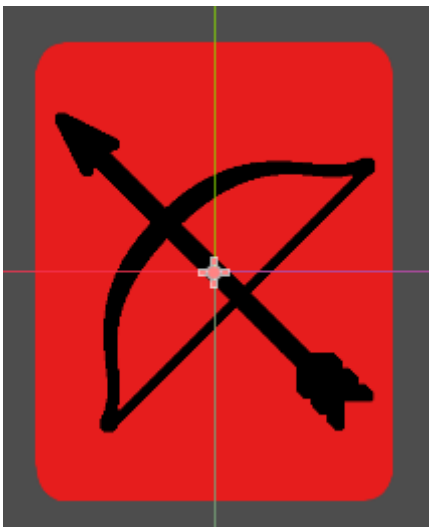

Figura 5.2. *Sprite* de *l'Archer*.

- *Tank*: L'enemic més simple del joc. És una massa que s'acosta al jugador per tocar-lo i eliminar-lo. És l'enemic més resistent del joc. Un enemic fàcil d'eliminar però tediós, útil per provar les habilitats del jugador.

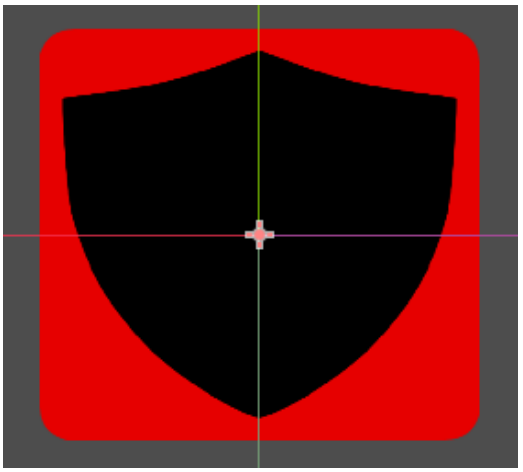

Figura 5.3. *Sprite* del *Tank*.

- *Slime*: Amb una naturalesa molt semblant a la del Tank, s'acosta al jugador per fer-li mal. L'*Slime* es duplica en parts més petites d'ell mateix quan el jugador l'ataca. Aquest enemic és un dels més comuns del món dels videojocs, pel que vam trobar oportú afegir-l'ho.

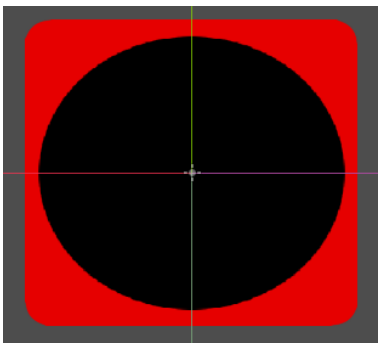

Figura 5.4. *Sprite* de l'*Slime*.

- *Dasher*: Un petit enemic molt veloç, el qual ataca ràpidament al jugador fent tres moviments devastadors. Després ha de descansar per un petit lapse de temps. Tots els enemics donen molt de temps per reaccionar, l'objectiu d'aquest és espantar al jugador amb la velocitat amb la que s'acosta.

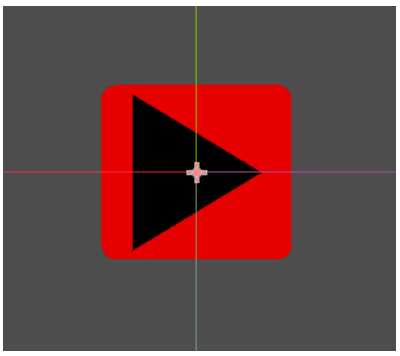

Figura 5.5. *Sprite* del *Dasher*.

- Cavaller: Un enemic gros que elimina a tothom qui se li acosta amb la seva gran espasa. El jugador se l'hi ha d'acostar perquè l'ataqui, i aprofitar els moments en que descansa per eliminar-lo.

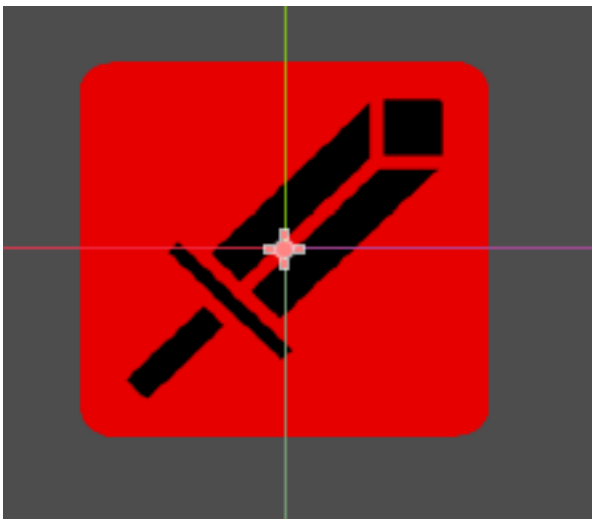

Figura 5.6. *Sprite* del Cavaller.

5.2.2. Sala amb enemic final

La sala més allunyada al centre del mapa, on inicia la partida el jugador, és on apareixerà l'enemic final. Vam creure que seria la manera més indicada de dissenyar-ho ja que, quan el jugador arribi a ella, ja serà suficientment poderós com eliminar-lo sense moltes dificultats.

5.2.2.1. Enemic final

- *Boss* : L'enemic final té diferents fases que s'activen quan perd vida. La primera dispara 20 bales aleatòriament en totes direccions. La segona controla zones on, si el jugador les trepitja, rep mal. La tercera és la suma de les dos fases anteriors. L'objectiu d'aquest enemic és ser el més difícil de tots; al ser el final, té moltes habilitats i es pot rebre mal fàcilment, pel que pot fer perdre la partida a molts jugadors.

Figura 5.7. *Sprite* del Boss.

5.3. Disseny d'objectes

Dins del projecte, els objectes són les habilitats que pot obtenir el jugador. Aquests efectes poden tenir diferents repercussions al jugador o als enemics que l'envolten. Aquestes han estat dissenyades basant-se en les habilitats que es solen adquirir en els *roguelikes,* ja que són efectes que es noten fàcilment pel jugador i li faciliten molt la partida.

Una vegada més, el tret distintiu dels *Roguelikes* són l'aleatorietat de cada partida. Per això, cada cop que el jugador agafa un efecte, aquest s'ha de escollir aleatòriament. Ja que hi ha efectes amb molts més beneficis que d'altres, aquesta aleatorietat s'ha de controlar de manera que els efectes més bons surtin menys i els més dolents surtin més, cosa que fa l'experiència de joc molt més interessant.

- *Root*: Si un enemic rep un atac amb aquesta habilitat, aquest es queda immobilitzat al terra durant uns segons, cosa que permet al jugador guanyar espai per evitar rebre mal. Aquesta habilitat té un 20% de possibilitats d'aparèixer.

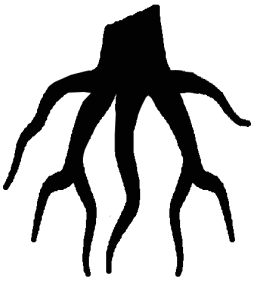

Figura 5.8. *Sprite* del *Root*.

- *Retro*: Quan un enemic rep mal del jugador amb aquest efecte, és desplaçat cap enrere, cosa que també serveix per guanyar distància. Si es té un jugador amb els dos efectes anteriors, es contraresten, pel que el més lògic és eliminar-ne un dels dos, ja que l'espasa es farà més lenta sense cap benefici. Aquesta habilitat té un 20% de possibilitats d'aparèixer.

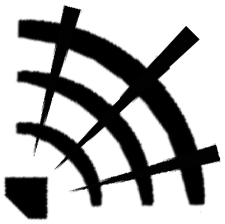

Figura 5.9. *Sprite* del *Retro*.

- *Vamp*: El jugador es cura una unitat de vida cada cop que mata a un enemic. Aquesta habilitat és molt interessant ja que beneficia molt als jugadors que exploren més sales, ja que podran arribar a l'enemic final amb molta més vida. Aquesta habilitat té un 15% de possibilitats d'aparèixer.

Figura 5.10. *Sprite* del *Vamp*.

- *Double_Dash:* Aquesta habilitat permet al jugador utilitzar el *dash* amb molta més freqüència, pel que tindrà l'oportunitat d'utilitzar-lo per esquivar atacs més ràpidament. Aquesta habilitat té un 20% de possibilitats d'aparèixer.

Figura 5.11. *Sprite* del *Double_Dash*.

- *Slow:* Quan un enemic és atacat pel jugador amb aquest efecte, aquest queda ralentitzat durant uns segons, cosa que permet al jugador atacar de nou i matar a l'enemic amb més seguretat. Aquesta habilitat té un 15% de possibilitats d'aparèixer.

Figura 5.12. *Sprite* del *Slow*.

- *Stable Damage:* Aquest efecte duplica el mal que fa el jugador, però per contra s'impedeix que es pugui moure mentre està atacant. Al crear aquest efecte, ens vam adonar que tenia molt de benefici a canvi de poc càstig, pel que vam decidir afegir-li una penalització de més. D'aquesta manera, eliminarà als enemics més ràpidament, però serà molt més vulnerable a atacs. Aquesta habilitat té un 10% de possibilitats d'aparèixer.

Figura 5.13. *Sprite* del *Stable Damage*.

5.4. Disseny estètic

El disseny estètic del personatges va dirigir tota l'estètica general del videojoc. Primerament es va començar pensant en estètica *Pixel Art 2D* però es va canviar al cap d'un temps per a una animació 2D de línies clares amb un shader de pixelació pel damunt.

El procés va començar amb l'exploració dels 12 principis de l'animació tradicional aplicats al videojoc, amb un vídeo que ho explica en profunditat i amb exemples [AlanBecker2017]. Aquí farem un petit resum dels mateixos:

1. Compressió i extensió

Aquest tècnia s'usa per donar flexibilitat a l'objecte, és a dir, més dur o més tou. Una pilota de goma o una bola de billar són uns bons exemples.

2. Anticipació

Anticipar una acció que passarà és una tècnica per deixar a la audiencia expectant i preveure el que passarà.

3. Posada en escena

Aquesta tècnica es refereix a la posada en escena quan es una cinemàtica o en una animació tradicional. Bàsicament, explica que sempre s'ha d'anar sequencialment i no aclaparar a l'audiència amb molts estímuls a la vegada, sinó que s'ha d'anar seqüencialment i de forma clara.

4. Animació directe o posició a posició

Aquest apartat és una decisió de com dur a terme el procés d'animació, és a dir, fer cada *frame* després del anterior (animació directe) o plantejar primer les posicions claus i anar emplenant els *frames* entremig (posició a posició).

5. Acció complementària i acció superposada

Acció complementària es basa en que les parts adjuntes a un objecte que són més lluny del moviment tarden més a seguir l'objecte i en arribar a destí. En canvi, l'acció superposada es tenir en compte com es mouen els diferents materials, per exemple el cabell i una gorra.

6. Accelerar i desaccelerar

El principi i el final de cada moviment són més lents, d'aquesta manera es dóna un valor més gran a les posicions claus.

7. Arcs

Els moviments naturals segueixen arcs, una mica de curvatura sempre dóna una sensació més orgànica. Això s'aplica també en l'altre direcció, si es vol que un moviment sembli artificial es fan els moviments més rectes. Per últim, dels moviments ràpids i forts es solen utilitzar línies més rectes.

8. Acció secundària

Enriquir amb accions secundàries per donar més enfasi a l'animació principal.

9. Temporització

La temporització té a veure amb quants frames per segon s'usen a l'animació, si s'usa animació en dosos o en uns. En dosos significa fer 1 frame de cada 2: en una animació de 24 frames, es fan 12 dibuixos per segon. En canvi, animar en uns es fer els 24 frames.

10. Exageració

Exagerar els moviments és una bona tècnica per donar èmfasi a una animació. Una bona manera per no passar-se amb aquest principi és exagerar-ho al màxim i llavors anar-ho baixant fins que quedi bé.

11. Dibuix sòlid

En animació de personatges que semblin en 3D s'ha de tenir en compte l'espai, és a dir, el volum i el pes del personatge.

12. Atractiu

La connexió amb el públic de qualsevol personatge és molt important. Per aquest motiu, l'animació s'ha d'adaptar a les seves característiques, i transmetre-les amb atractiu.

Al tenir clar aquest punt vam fer la búsqueda de com animar amb *Pixel Art 2D*, i quines eren les tècniques més importants i conegudes en aquest camp. A banda de que l'animació és exactament la mateixa, hi ha uns trucs que s'usen per donar més dinamisme i assemblar-se a l'animació clàssica, aquest són anomenats animació *Sub-Pixel*. Es tracta de animar a menys de 1 píxel de moviment el qual és impossible ja que només pots passar d'un píxel a un altre, no pots subdividir, ara bé pots assimilar-te a aquest efecte amb:

1. *Smearing*

La traducció literal és tacat, és a dir deixar un rastre del fram anterior abans de passar al frame següent, aquesta tècnica és una extracció de alguns dels principis de l'animació clàssica al *Pixel Art* concretament el principi 1 i el 5. En animació clàssica això dóna resultat a algunes imatges espectaculars que no tenen molt sentit per si soles però si en el context de l'animació. Veure **Figura 5.14**.

Figura 5.14. Exemplificació de *frames* que utilitzen *smearing* en un joc de lluita.

2. Interpolació de contorns

Com el nom indica el que fem és en comptes de moure tota una línia de píxels com a bloc, ho fem amb diferents passos, baixant cada vegada més línia, per exemple primer els contorns i llavors el centre de la línia fins a haver-la baixat tota a la següent línia.

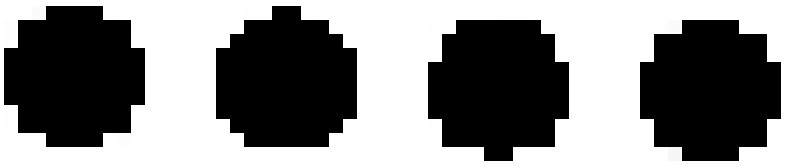

Figura 5.15. Exemplificació de *frames* que utilitzen *Edge Tweening*.

3. Interpolació de colors

En aquesta tècnica, en comptes de moure una línia amb diferents passos però utilitzant els mateixos colors, és a dir o un píxel amb color o píxel sense color, fem la interpolació tenint en compte els colors veïns amb els que s'interpola. Per això aquest métode es usat sobretot en interiors de models i no als contorns.

Bàsicament es va barrejant els colors amb el temps, fins que el píxel destitjat s'ha mogut on estava l'altre color i s'ha acabat la transició. En el vídeo de Luis Zuno, es fa una bona demostració de la tècnica explicada i es posen exemples [LuisZuno2019]. Veure **Figura 5.16**.

Figura 5.16. Exemplificació de interpolació de colors, extret del video de Luis Zuno. S'observa com el píxel caniva d'un color a l'altre amb 5 frames en comptes de fer la transició amb 1.

Després de fer varies proves amb animació pixel art, vam decidir canviar d'aquesta a animació clàssica amb línia concreta, i finalment al acabar de fer-la amb un shader de pixelació acabar de donar el toc final al videojoc.

L'animació es va fer pensant amb la demo del videojoc *Have a Nice Death*, presentada fa 8 mesos. A la **Figura 5.17** hi podem veure una imatge del trailer [GameSpot2021].

Figura 5.17. Captura del videojoc *Have a Nice Death*.

5.5. Interfícies

La única interfície que hi ha al videojoc són els menús i la barra de vida, que són quantes ànimes li queden al protagonista. L'altre interfície que s'ha dissenyat és la llista d'objectes. Aquesta es mostra al jugador a l'obrir l'inventari, on es mostren els diferents efectes obtinguts durant la partida i es poden eliminar, fent clic al que es vulgui.

5.6. Narrativa

5.6.1. Sinopsi

Després de molts anys d'entrenament, el jugador és escollit per portar l'espasa llegendària i adentrar-se en un món atemporal on tot el que es veu l'intenta assassinar. S'ha de sobreviure en aquesta dimensió desconeguda i tornar al teu regne amb la victòria.

5.6.2. Backstory

Fa 5 anys es va obrir una escletxa dimensional que va permetre l'entrada de monstres d'un altre món. Al mateix temps, va aparèixer l'empunyadura d'una espasa que semblava espantar a aquests monstres. Per això, es va protegir i mitificar com la representació d'una deïtat.

El culte més poderós de la regió realitza un ritual sagrat cada pocs mesos, en el que s'enviava un escollit amb l'empunyadura de l'espasa a l'escletxa per intentar tancar-la. Ells creien que fent això, algun dia apareixeria algú capaç de derrotar als enemics que hi havia a l'altre cantó i així derrotar al mal que apareixia al seu regne.

Després de molts anys enviant a cultistes, l'únic que veuen és com es forma de nou l'espasa dins del ritual, sense l'escollit. Tot i això, ells creuen que aquest sacrifici els acosta cada cop més a la victòria.

5.7. Món del joc

5.7.1. Dimensió física

La dimensió física de GreaterSword és representada per tres tipus atributs:

- Dimensió Espacial: La dimensió espacial del projecte és 2D, però amb vista des de planta. Això fa que els *Sprites* es representin com si quedéssin aixafats al terra.
- Escala: El joc està pensat per ser a escala humana, ja que els enemics i jugador tenen aquesta mida.
- Límits: Els límits del joc són la pròpia masmorra, on el jugador es pot moure dins de cada sala amb llibertat.

5.7.2. Dimensió temporal

La masmorra està en una altra dimensió, pel que el temps no flueix dins d'ella. Tot el que fa el jugador durant cada partida no segueix les lleis temporals del món exterior.

5.7.3. Dimensió ambiental

GreaterSword està ambientat en un món fantàstic sense cap relació al nostre. Tot i així, ens hem basat en els mons medievals màgics que existeixen en moltes de les històries i videojocs de ficció creats actualment, com ara El Senyor dels Anells, el *Divinity: Original Sin 2* o tota l'ambientació i regnes creats amb la saga de joc de rol *Dungeons and Dragons*.

5.7.4. Dimensió emocional

L'entreteniment del videojoc ve donat per les pròpies mecàniques, que comporten un combat difícil. L'adrenalina i el frenesí de cada sala són les emocions i la gestió de l'espasa per donar el moment de pausa entre sala i sala.

5.7.5. Aspectes ètics

En el videojoc els protagonistes canvien per narrativa cada vegada i són sacrificats per una causa major sense saber ben bé si serveix d'alguna cosa.

L'aspecte ètic més important és el tractament a partir de la narrativa del treball de les sectes i el que comporta formar-ne part, no preguntar-se el perquè i actuar en benefici del grup, en qualsevol situació.

A part d'això, hi ha el tema recurrent de la mort dels enemics que simplement estan fent el mateix que el jugador, protegir les seves vides o complir les imposicions del regne, igual que el protagonista.

També dóna una pinzellada sobre la mort, ja que l'espasa torna a aparèixer i tot està igual, l'únic que canvia es la falta del protagonista que ha entrat a la masmorra i està en un espai atemporal fins a acabar victoriòs o eliminat.

5.8. Elements a desenvolupar

Els elements a desenvolupar del videojoc són pocs en termes artístiques ja que ens hem centrat en el desenvolupament mecànic i tecnològic.

5.8.1. Objectes i escenaris

Els efectes han de tenir un identificatiu clar per poder-los diferenciar un de l'altre i un escenari base on l'acció succeeix a cada sala. Overwatch és un bon exemple de diferenciació: per exemple, una habilitat amb 3 accions de moviment s'indica amb el següent icona. Veure la **Figura 5.18**.

Figura 5.18. Icona de l'habilitat *Blink* del personatge *Tracer* a *Overwatch*.

L'escenari s'en formarà 1 que serà el mateix per totes les sales, amb estil semblant a The Binding of Isaac, tètric i amb subterrani. Veure **Figura 5.19**.

Figura 5.19. Escenari a *The Binding of Isaac* un dels pisos anomenat *Flooded Cav*e.

5.8.2. Animacions

Les animacions seran les de moviment i atac del personatge principal. Aquest es podrà moure en totes direccions, però per les animacions en 4 en fem prou. Veure **Figura 5.20**.

Figura 5.20. *Sprites* del protagonista Will a *Moonlighter* en 4 direccions.

També s'animarà els atacs del combat contra l'enemic final.

5.9. Decisions de disseny

5.9.1. Singletons i globals

S'usarà un *Singleton* per a emmagatzemar totes les variables i mètodes usats per escenes diferenciades. Així es pot accedir a aquest script des de qualsevol punt del videojoc i tenim un ancoratge pel més important.

5.9.2. Dispositius d'entrada

El videojoc es pot jugar amb 2 dispositius d'entrada, teclat i ratolí i també amb controlador, en el nostre cas controlador de *Xbox Series X*. *Godot* mapeja els *inputs* i els recull al mig d'execució, per molt que canviï de dispositiu d'entrada, així que no hi ha problema en el canvi de dispositiu.

Figura 5.21. *Inputs* afegits al videojoc, a part dels bàsics de moviment i acceptació i declinació a *Godot*.

6. Implementació i proves

6.1. Diagrama de classes

Aquest és el diagrama de classes reduït del projecte. La classe *Global*, el *Singleton*, es connecta amb totes. La classe *Level* és la principal, i engloba la majoria de la resta. Finalment les pantalles de fi de partida, *Screens* que tornen al *Main Menu*.

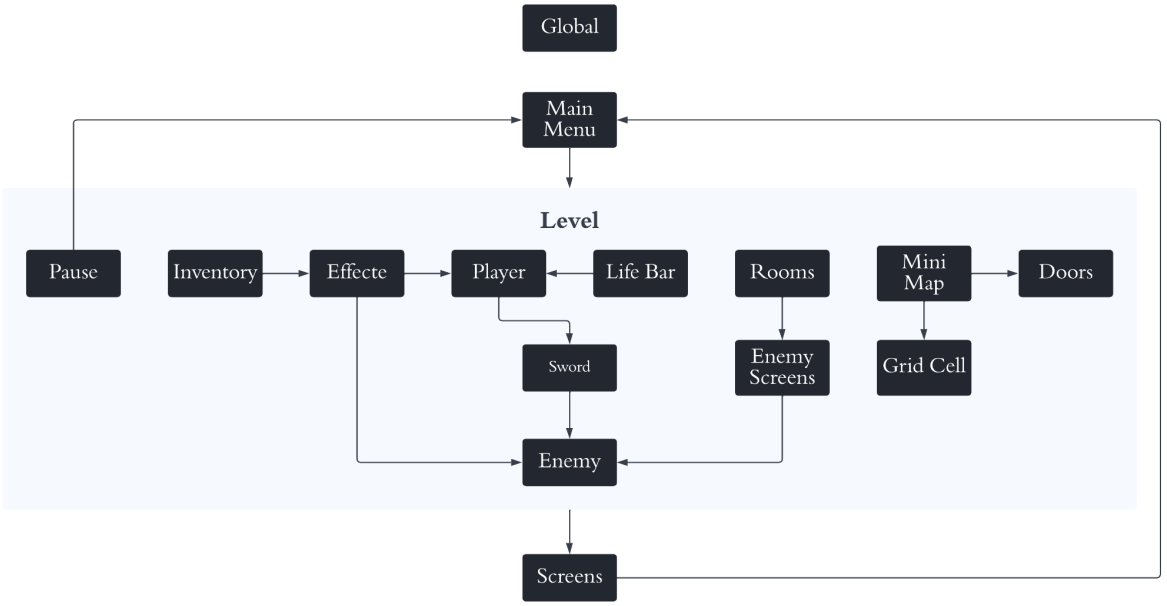

Figura 6.1. Diagrama de classes del projecte.

6.2. Implementació de la generació de nivells

La generació de nivells del videojoc està basada en l'algoritme Drunk Walk, què genera un nivell a partir de uns quants agents que es mouen per una garella i van marcant el seu camí. Tot el codi de l'algoritme està dins la funció *_ready()* de *Level.gd*.

Level.gd

```
const MAZE_SIZE = 11 # Nombre de sales per costat
const NUM_STEPS = 10 # Iteracions de l'algoritme

randomize()

mazeRooms = Array()

for i in range(MAZE_SIZE):
     mazeRooms.append(Array())
     for _j in range(MAZE_SIZE):
          mazeRooms[i].append("Info")
```

```
Info
{"room": '-', "doors": [false, false, false, false]}
```

Primerament al codi anterior es fa la instància de la graella 11x11, que és la mida que hem agafat, però es pot fer de la mida que es vulgui. Es un doble bucle que crea una *array* de *arrays* i cada posició s'hi inclou la informació necessària de cada sala:

`"room"`:
- Habitació buida: `'-'`
- Habitació plena: `'*'`

`"doors": [Nord, Est, Sud, Oest]`
- Porta tancada: `'false'`
- Porta oberta: `'true'`

Level.gd

```
var mazeCenter = mazeRooms.size()/2

var pos1 = Vector2(mazeCenter, mazeCenter)
var pos2 = Vector2(mazeCenter, mazeCenter)
var pos3 = Vector2(mazeCenter, mazeCenter)
var pos4 = Vector2(mazeCenter, mazeCenter)

mazeRooms[mazeCenter][mazeCenter]["room"] = '*'
```

Seguidament s'instancien els agents que es mouràn per aquesta graella *pos1*, *pos2*, *pos3* i *pos4*, tots al mateix punt, al centre de la graella *mazeRooms.size()/2* en aquest cas la posició P(5,5).

També es canvia aquesta posició a visitada (Habitació plena) i es dona pas a l'algoritme de moviment dels agents, 4 en aquest exemple, però també es podria automatitzar per a fer el nombre d'agents desitjat.

Level.gd

```
for i in NUM_STEPS:
     pos1 = newRandPos(pos1)
     pos2 = newRandPos(pos2)
     pos3 = newRandPos(pos3)
     pos4 = newRandPos(pos4)
     mazeRooms[pos1.x][pos1.y]["room"] = '*'
     mazeRooms[pos2.x][pos2.y]["room"] = '*'
     mazeRooms[pos3.x][pos3.y]["room"] = '*'
     mazeRooms[pos4.x][pos4.y]["room"] = '*'
```

S'executa un moviment de 1 casella aleatori de cada agent, el nombre de iteracions marcat per la constant NUM_STEPS i es posen les habitacions pertinents a visitades.

Level.gd

```
Global.MapPos = [mazeCenter, mazeCenter]
UpdateDoors()
show(mazeRooms)
```

Finalment es s'indica la sala inicial al Singleton (*Gloabl.MapPos*), s'activen les portes pertinents i es mostra el mapa al jugador.

- Funcions addicionals:

Level.gd

```
func newRandPos(pos):
     var possible = [false, false, false, false]
     if pos.y > 0:
          possible[0] = true
     if pos.x < MAZE_SIZE-1:
          possible[1] = true
     if pos.y < MAZE_SIZE-1:
          possible[2] = true
     if pos.x > 0:
          possible[3] = true

     var rand = randi()%4
     while (!possible[rand]):
          rand = randi()%4
```

A *newRandPos()*, primer es mira si l'agent està a un costat de mapa i es marca quiens posicions són vàlides per moure's i llavors es genera un moviment de 1 posició en direccions posibles (nord, est, sud o oest). Si no és possible, es torna a generar una altere posició, ja que com a mínim sempre té 2 direccions possibles.

Level.gd - Continuació newRandPos(pos)

```
if rand == 0:
     mazeRooms[pos.x][pos.y]["doors"][2] = true
     pos.y -= 1
     mazeRooms[pos.x][pos.y]["doors"][0] = true
elif rand == 1:
     mazeRooms[pos.x][pos.y]["doors"][1] = true
     pos.x += 1
     mazeRooms[pos.x][pos.y]["doors"][3] = true
elif rand == 2:
     mazeRooms[pos.x][pos.y]["doors"][0] = true
     pos.y += 1
     mazeRooms[pos.x][pos.y]["doors"][2] = true
elif rand == 3:
     mazeRooms[pos.x][pos.y]["doors"][3] = true
     pos.x -= 1
     mazeRooms[pos.x][pos.y]["doors"][1] = true

return pos
```

Per últim es comprova quin moviment ha pres l'agent i s'obre la porta que és la connexió entre aquesta sala i la següent.

Ìndex de portes:

- Nord = 0
- Est = 1
- Sud = 2
- Oest = 3

Amb aquesta tècnica de canvi irreversible en les variables d'informació de cada sala, aconseguim que, si un altre agent passa per la mateixa habitació, no es malmeti la graella o les portes anteriorment inicialitzades i, a la vegada, també permet que s'obrin altres portes de la mateixa habitació, aconseguit que cada sala tingui un set de portes obertes diferent.

Level.gd

```
func UpdateDoors():
     if Porta oberta:
          $MapObjects/DoorSouth.EnableDoor()
     else:
          $MapObjects/DoorSouth.DisableDoor()
```

```
Porta oberta
mazeRooms[Global.MapPos[0]][Global.MapPos[1]]["doors"][0]
```

A *UpdateDoors()*, es comprova si cadascuna de les portes és oberta. Al codi superior es mostra com amb les funcions *EnableDoor()* i *DisableDoor()* s'obren i es tanquen les portes (amaguen el node i fan desactiven la col·lisió).

La condició *Porta Oberta* escrita al calaix de sota, per claredat, mira si la porta Nord és oberta, les altres portes tenen el mateix codi canviant l'índex.

La funció *show(mazeRooms)* rep com a argument el laberint ja creat i crida la funció de generació del mapa a la interfície *CreateMinimap(maze)*.

L'escena MiniMap conté el component d'interfície del mapa, aquest és una graella construida a partir de un node *GridContainer* de *Godot*.

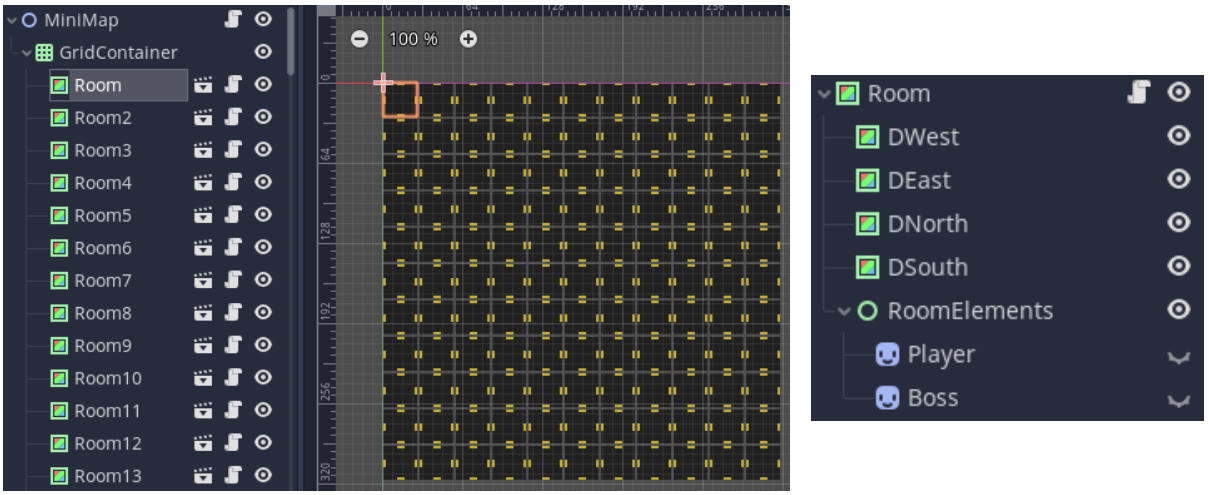

Figura 6.2. A l'esquerra arbre del mapa, s'observa la graella de 11x11 escenes *Room* filles del *GridContainer*, a la dreta arbre de l'escena *Room*.

Cada posició de la graella conté una escena *Room*, que té a l'interior uns rectangles grocs per indicar les porta i un rectangle gris pel fons. A més tenen *Sprites* per indicar: si hi ha el *Player* i un si és la sala amb *Boss*.

L'escena *Room* conté funcions per ensenyar portes (*openDoors()*) que rep el vector de booleans i obre les portes pertinents, i per amagar totes les portes (*hide()*) i la funció *BossRoom()* per indicar que hi ha el *Boss*.

MiniMap.gd

```
var furthestRoom = Vector2(Global.MapPos[1],Global.MapPos[0])
var dist = 0

for i in range(MAZE_SIZE):
     for j in range(MAZE_SIZE):
          if maze[i][j].room == "-":
               miniMapRooms[j][i].hide()
          else:
               miniMapRooms[j][i].openDors(args)
               var aux = euclideanDist(args)
               if aux > dist:
                    dist = aux
                    furthestRoom = Vector2(i,j)

miniMapRooms[furthestRoom.y][furthestRoom.x].BossRoom()
Global.BossRoom = furthestRoom
```

Com es pot observar a la funció *CreateMinimap(maze)*, es fa un doble bucle per totes les portes, primerament mira si l'habitació és buida, llavors executa *hide()*, en cas contrari, executa *openDoors()*, amb els arguments pertinents, *args*, explicats anteriorment per claredat de lectura

Llavors calcula la distància euclidiana, entre la posició actual que està processant P(i,j) i el centre del mapa. En cas que la distància sigui superior a la màxima enmagetzamada a la variable, es guarda com a *furthestRoom*, és a dir l'habitació més llunyana i la que serà hi haurà el *Boss*.

En darrer lloc s'executa la funció BossRoom().

6.3. Implementació de les sales

Figura 6.3. Interfície base de les sales.

Com podem veure a la **Figura 6.3**, la interfície base de les sales es basa en una superfície rectangular on s'ha afegit les parets utilitzant una de les eines del *Godot*, el *CollisionPolygon2D*, el qual ens resulta molt útil per utilitzar com a limitadors de l'espai de moviment del jugador.

A més d'això, també es pot veure uns rectangles blaus als límits de les parets. Aquests són uns elements anomenats *Area2D*, els quals ens serveixen com a detectors per comprovar cap a quina direcció vol moure's el jugador.

A la part superior dreta podem observar el minimapa abans de ser generat aleatòriament.

Al ser una escena fixa on s'intancien tots els elements a la mateixa (el jugador i els enemics), no hi ha una manera de diferenciar quina sala estem segons l'escena en la que el jugador està. Per això, des del

Singleton es té una llista de posicions on es guarda la sala en la que està el jugador cada vegada que entra a una habitació.

Global.gd

```
func AlreadyVisited():
     var found = false
     var i = 0

     while i < RoomsVisited.size() and !found:
          if RoomsVisited[i][0] == MapPos[0] and
RoomsVisited[i][1] == MapPos[1]:
               found = true
          i += 1

     return found


func VisitedRoom(ant):
     if !([ant[0],ant[1]] in RoomsVisited):
          RoomsVisited.append([ant[0],ant[1]])
```

Com podem veure en el codi mostrat, la gestió de les sales visitades pel jugador consta de dues funcions diferents, primerament tenim *AlreadyVisited()* que bàsicament comprova si ja s'ha entrat anteriorment a aquesta sala i retorna un booleà, i *VisitedRoom()* la qual afegeix la sala a la que s'ha entrat, en cas que aquesta no hi sigui a la llista de sales visitades.

La funció *VisitedRoom()* es crida al script de les portes, quan la *Area2D* detecta que ha entrat el jugador per una d'elles. L'altre funció, es crida al codi referent a l'escena *Level*, què és el codi principal que controla una gran part del joc.

Aquestes funcions són necessàries ja que si el jugador visita una sala que ja ha visitat anteriorment, en aquesta no l'hi hauran d'aparèixer enemics ni efectes per aconseguir.

Level.gd

```
func _process(_delta):
     if Global.NumEnemics <= 0:
          if !Instanciat and !Global.AlreadyVisited():
               roomEffect = Efect.instance()
               add_child(roomEffect)
               if roomEffect.id != "non":
                    roomEffect.position.x = 960
                    roomEffect.position.y = 540
               else:
                    roomEffect.queue_free()
               Instanciat = true

          UpdateDoors()


func SpawnEnemies():
     if is_instance_valid(roomEffect):
          roomEffect.queue_free()

     if !Global.AlreadyVisited():
          if Global.isBoosRoom():
               var instance = Boss.instance()
               add_child(instance)
          else:
               var pick =
Global.rng.randi_range(0,Pantalles.size()-1)
               var instance = Pantalles[pick].instance()
               add_child(instance)
```

Dins de *process()* (la funció que s'executa contínuament dins del joc) es necessita comprovar el número d'enemics que té la sala (una de les variables que tenim guardades al Singleton). Si la sala no té cap enemic viu, es crea una instància un efecte al centre de la sala que el jugador podrà obtenir si el toca. A més, també es comprova la possibilitat que el jugador ja tingui tots els efectes, cosa que farà que no es creï cap altre nou.

Per afegir els enemics s'han dissenyat de manera que siguin aleatoris. Per fer-ho, s'han creat vàries escenes amb diferents combinacions d' enemics (**Figura 6.4**), i s'han afegit a una llista d'on es triaran a la funció *SpawnEnemies()*. Aquesta controla primer de tot si el jugador no ha agafat l'efecte de la sala anterior, si és així l'elimina. Seguidament, es comprova que sigui la primera vegada que el jugador entra a la sala i es tria una de les escenes a través de la llavor aleatoria que hem creat al Singleton.

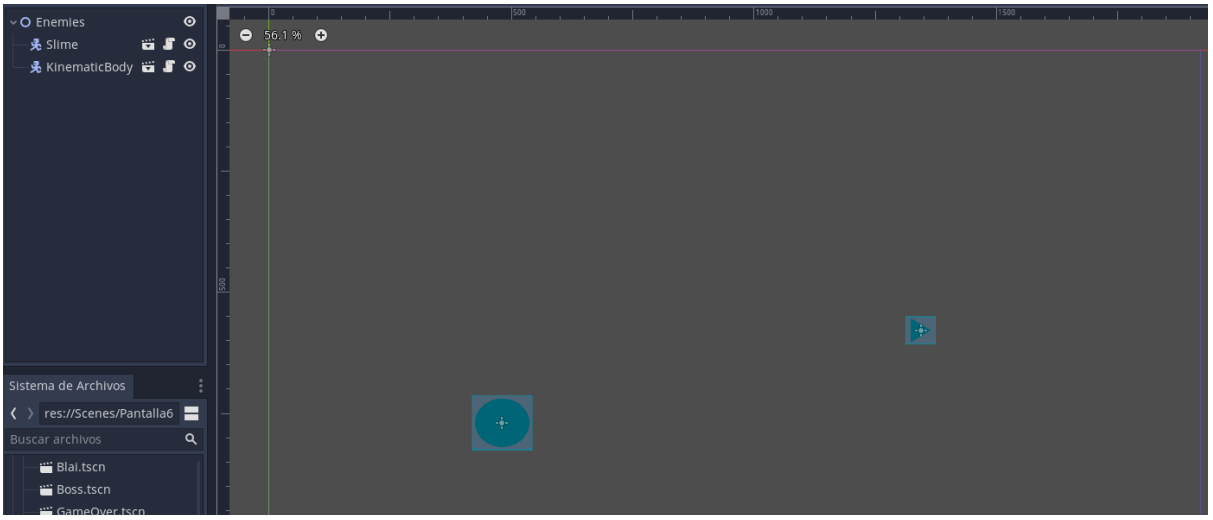

Figura 6.4. Interfície de la Pantalla 6, una de les combinacions d'enemics que pot tocar aleatòriament.

6.4. Implementació del personatge principal

El jugador és l'element més important del projecte, i a causa de la peculiaritat dels atacs amb la que comptem, té un dels codis més complexes del joc.

Figura 6.5. Arbre de nodes del *Player*.

Com es pot apreciar a la **Figura 6.5,** el jugador consta de un *AnimatedSprite*, que s'utilitza per les animacions de moviment; un *CollisionShape2D* per detectar les col·lisions del jugador; un temporitzador, per detectar el temps en el que es pot utilitzar el *Dash*; un *Sprite*, per indicar l'espasa del jugador i la instància de l'espasa.

Player.gd

```
const MAX_SPEED = 500 #Velocitat del jugador
var vida # Vida del jugador
var velocity = Vector2.ZERO
var Invulnerable = false # Booleà si el jugador és invencible
var multiplier = 1
var cursed = false
var is_attacking = false # Booleà si el jugador esta atacant
var can_dash = true # Booleà su el jugador pot utilitzar el
dash
var dashTimer = 3 # Retard per reutilitzar el dash
var isDashing = false # Booleà si el jugador esta utilitzant
el dash
```

6.4.1. Moviment

El moviment del personatge, com tots els *Inputs* del jugador, es poden entrar a través d'un controlador o amb el propi teclat. Per fer-ho, s'ha afegit al mapa d'entrada (vist al l'**Apartat 5.9.2**), els dos possibles *Inputs*, el *joystick* del controlador, i les tecles *WASD*.

Player.gd

```
func _movement():
     var inputVector = Vector2();
     inputVector.x = Input.get_action_strength("ui_right") -
Input.get_action_strength("ui_left")
     inputVector.y = Input.get_action_strength("ui_down") -
Input.get_action_strength("ui_up")
     inputVector = inputVector.normalized()

     control_move(inputVector)

     if inputVector != Vector2.ZERO:
          velocity = inputVector * (MAX_SPEED * multiplier)
     else:
          velocity = Vector2.ZERO

     velocity = move_and_slide(velocity)

     if Input.is_action_just_pressed("dash"):
          dash()
```

Com podem veure, la funció *movement()* s'encarrega de tot el moviment que pot fer el jugador. Gràcies a les funcions predeterminades de Godot tal com *get_action_strength()* i *normalized(),* podem detectar el moviment que vol fer el jugador i cridar el botó amb el que el fa.

A més, també es controla que el jugador deixi de moure's quan l'usuari deixa de prémer el botó o *joystick*. Aquí també es controla si el jugador vol utilitzar el *dash*, el qual té una funció diferent que es crida quan el jugador prem l'*Input* pertinent.

Player.gd

```
func dash():
     if can_dash:
          can_dash = false
          dashPlayer(facing)
          multiplier = 2
          $DashTime.start()
          Invulnerable = true
          isDashing = true
          yield(get_tree().create_timer(0.4), "timeout")
          Invulnerable = false
          isDashing = false
          yield(get_tree().create_timer(dashTimer),
"timeout")
          can_dash = true
```

El dash es basa en un condicional que controla si el personatge pot reutilitzar el dash (aquest té un temps de recàrrega), i seguidament es posa al jugador en estat d'invulnerabilitat i augmenta el valor de la variable *multiplier*, la qual fem servir per modificar la velocitat del jugador i fer-la més ràpida durant 0.4 segons.

Un cop fet això, s'inicia el temporitzador de nou el temporitzador que controla si el jugador pot tornar a utilitzar el *dash*.

6.4.2. Arma

En el videojoc l'arma pot variar de mida, llavors el balanceig del videojoc depèn de la velocitat a la qual l'arma fa l'escombrada de 90º al atacar. Aquesta efecte d'atac ha de variar segons el temps, ja que com més gran sigui l'arma, més lenta ha d'atacar.

Per aconseguir aquest efecte a part de l'animació, també s'ha de tenir en compte que l'atac ha de detectar la col·lisió amb els enemics d'una manera poc convencional.

Aquest mètode està influenciat en la implementació de la detecció de col·lisions, les *hitbox*, de diferents atacs al videojoc Overwatch.

Figura 6.6. Atac principal del personatge *Reinhardt* a *Overwatch*. Extret del vídeo de YouTube [KarQ2021], usant l'eina per visualitzar *hitboxs* [EloHellWorkshop2021].

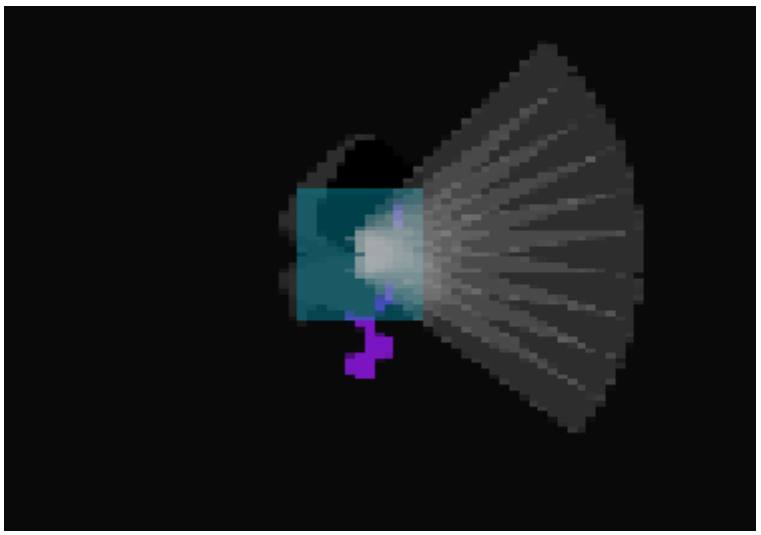

Figura 6.7. Jugador de GreaterSword *ingame*, amb les col·lisions de l'espasa visibles.

Com es pot observar a la **Figura 6.6**, el personatge té unes hitbox al davant i, a mesura que l'animació del martell passa pel davant d'aquestes, es van activant i va avançant la col·lisió. Al ser uns milisegons dins el videojoc, les animacions i les hitbox no cal que siguin sincronitzades. En canvi, en el nostre cas, l'atac és més lent per tant sí ha d'estar sincronitzat (**Figura 6.7**).

Figura 6.8. Arbre de nodes de *Sword*.

Com es mostra a la **Figura 6.8**, Sword simplement és un node *Position2D*, que marca una posició respecte a *Player* al ser fill del mateix en aquest node s'hi afegiran les diferents àrees.

Sword.gd

```
var fullSwordArea =
preload("res://ObjectScenes/PlayerObjectsScenes/FullSwordArea
.tscn")
var partialSwordArea =
preload("res://ObjectScenes/PlayerObjectsScenes/PartialSwordA
rea.tscn")
var swingTime = float(Global.SwordSize)/1000
var partitions
var canAttack
```

Al codi superior es mostren les variables de l'escena *Sword*. Trobem l'escena *fullSwordArea* que controla la mida de l'espasa de 0 a 50 i l'escena d'una sola àrea de col·lisió *partialSwordArea* de la qual se'n instanciaràn més d'una que controla la mida de l'arma de 51 fins ∞.

També trobem la variable *swingTime* que en aquesta escena controla quant temps passa entre cada activació de col·lisió, per això és proporcional a la mida de l'arma i partitions què ens indicarà quantes col·lisions s'han de usar per cada mida d'arma.

Per últim la variable *canAttack* què és un booleà que permet al jugador usar l'atac. Això ens serveix per no tenir conflictes a l'hora de fer l'arma més llarga o haver d'afegir efectes, mentre es fan aquestes operacions no es pot atacar.

Primerament al iniciar-se l'arma es crida la funció *UpdateSize()*.

Sword.gd

```
func updateSize():
     canAttack = false
     yield(get_tree().create_timer(Global.SwingTime),
"timeout")
     deleteSword()
     var swordSize = Global.SwordSize
     if swordSize <= 50:
          var areaInstance = fullSwordArea.instance()
          add_child(areaInstance)
     else:
          partitions = int(swordSize/7)
          var swordCollScale = float(swordSize)/50
          var swordCollRot = -45
          var rotStep = float(90)/partitions

          for _i in range(0,partitions):
               var areaInstance = partialSwordArea.instance()
               add_child(areaInstance)
               areaInstance.scale.x = swordCollScale
               areaInstance.rotation_degrees = swordCollRot
               swordCollRot += rotStep
     canAttack = true
```

La funció superior crea les *hitbox* necessàries per cadascuna de les mides de l'espasa, que extreu del *Singleton*. El primer pas és impedir que el jugador pugui atacar durant el procés de creació de l'arma, que intentaria accedir a col·lisions que encara no existeixen i donaria problemes.

Llavors es fa un *timeout* amb la funció *yield* propia de *Godot*, mentre s'acaba un possible atac que estigui fent el jugador, i llavors s'executa *deleteSword()* que elimina qualsevol instància de col·lisió que hi pugui haver prèviament.

Es comprova si la mida de l'espasa és més petita a 50, llavors només s'instancia una àrea *fullSwordArea* i l'atac passa a ser una implementació bàsica de qualsevol altre joc. En cas contrari es passa a la creació de les variables dependents de *swordSize*, com *partitions*, *swordCollScale* (indica l'escala a la qual les col·lisions han d'estar per ser de mida desitjada) i *rotStep* (indica quan s'ha de rotar cada instància de col·lisió per no deixar espais buits quan s'ataca). També s'instancia la variable *swordCollRot*, que indica a quants graus es comença a instanciar la primera àrea -45º, el mateix que 270º.

Finalment es fa un bucle amb el nombre de particions i s'instancien les diferents col·lisions. Aquí es fan diferents passos, primera es fa una instància de *partialSwordArea*, es fa un *add_child()* a Sword, es canvia l'escala de l'àrea i es canvien els seus *rotation_degrees*, per últim es suma *rotScale* a *swordCollRot* per poder indicar els graus de rotació de la següent àrea.

Sword.gd

```
var atkState = 1

func _process(_delta):
     if Input.is_action_just_pressed("Attack") and canAttack:
          if atkState == 1:
               basic(1)
               atkState = 2
          elif atkState == 2:
               basic(2)
               atkState = 1
```

Dins la funció *_process()* de l'espasa es troba la detecció de l'*input* d'atac, la primera vegada, s'executa la funció basic amb argument 1 i la segona vegada, en argument 2 i així es va intercalant l'atac, primer l'espasa escombra de dreta a esquerra i llavors al inrevés. Sempre tenint en compte que el jugador pugui atacar usant el booleà *canAttack*.

Sword.gd

```
func basic(combo):
     if canAttack:
          canAttack = false
          if combo == 1:
               for i in self.get_children():
                    i.get_node('Collision').disabled = false
                    yield(get_tree().create_timer(swingTime),
"timeout")
                    i.get_node('Collision').disabled = true
          elif combo == 2:
               var lastChild = get_child_count() - 1
               while lastChild >= 0:

get_child(lastChild).get_node('Collision').disabled = false
                    yield(get_tree().create_timer(swingTime),
"timeout")

get_child(lastChild).get_node('Collision').disabled = true
                    lastChild -= 1

          canAttack = true
```

Per últim l'atac és una successió de activacions de les col·lisions que té l'arma, instanciades amb la funció *UpdateSize()*.

Al codi superior es pot veure com es controla si el jugador pot atacar, en aquest cas es posa aquesta variable a *false* perquè no pugui tornar a atacar fins a acabar el primer atac. Llavors es comprova si l'atac és el primer o el segon de la seqüència. L'únic que canvia entre els 2 es l'ordre d'activació de les àrees, de dalt a baix en la jerarquia de fills de l'espasa, o a l'inrevés.

Per últim es fa un bucle que en el cas de *atkState* 1 s'utilitza un *for* del primer a l'últim fill de *Sword*, i en el cas de *atkState* 2 s'utilitza un *while* del últim al primer fill de *Sword*. S'accadeix a cadascun dels fills se'ls hi activa la col·lisió i al cap de un temps marcat per *swingTime* es desactiva, creant un efecte d'escombrat.

En *atkState* 1 els fills ja es tenen dins la variable *i*, llavors només fa falta usar la funció *get_node('Collision')* i accedir a la variable *disabled* de dins de cada fill. En canvi en *atkState* 2 s'ha de usar un índex per trobar a cada instància de l'arma ja que no es pot fer un *for* de últim a primer a *Godot*, usant la funció *get_child(childIndex)* i el procés descrit en l'altre estat.

6.4.2.1. Atacs

Les *Area2D* de l'espasa només detectarà enemics gràcies a les *Layers* explicades anteriorment, i, al activar una de les Signals de les Area2Ds de l'atac de l'espasa, es cridarà la funció *_Damage()* que té cada un dels enemics implementada.

Tank.gd

```
func _Damage():
     if !Invulnerable:
          vida -= Global.PlayerDamage
          if retro:
               _knockback()
          Invulnerable = true
          if vida > 0:
               yield(get_tree().create_timer(Time_Inv*1.1),
"timeout")
          Invulnerable = false
```

Com podem veure en aquest codi, l'enemic rep mal restant el *PlayerDamage* que té el jugador a la vida que té, i es torna invulnerable durant un temps, per a què no rebi dos danys del mateix atac.

6.5. Implementació dels enemics

Els enemics són la part més important del joc, ja que és amb els que ha d'interactuar el jugador i els que faran que aquest trii unes habilitats o unes altres davant dels enemics.

Per això, el joc conta amb 5 enemics bàsics i 2 enemics finals amb els que cada partida podrà ser completament diferent. L'implementació d'aquests varia segons el tipus d'enemic i la naturalesa donada per cada un.

Degut a la manera en la que està fet l'atac del jugador (explicat a l'**Apartat 6.5.2**) els enemics no poden tenir números enters com a vida, ja que un sol atac del jugador activaria vàries vegades les *Area2D* de l'espasa, cosa que farà que els enemics els hi baixi la vida més ràpid de l'esperat. Durant la fase de disseny del projecte, es va plantejar crear una llista que contingués els enemics que tocaves amb l'espasa per només tocar-los una vegada, però finalment es va descartar a causa de que es va trobar una manera menys costosa de fer-ho. Aquesta manera era que tots els enemics tinguéssin un temps en el que fossin invulnerables en el moment en el que els toca l'espasa del jugador. Així, només els tocaria una vegada i l'únic recurs que necessites és una variable de més.

El temps de invulnerabilitat és marcat per la variable *SwingTime* que conté el *Singleton*, un element que anirà variant durant la partida a mesura que s'agafin efectes.

6.5.1. Enemics bàsics

6.5.1.1. Tank

El *Tank* és l'enemic bàsic del joc, és el que s'ha utilitzat més per testejar les habilitats del jugador i és el més senzill d'implementar.

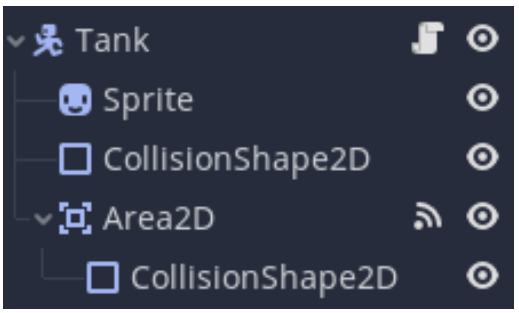

Figura 6.9. Arbre de nodes del *Tank*

Com podem observar en la **Figura 6.9**, el *Tank* està format per un *Sprite*, que li donarà una forma; una *CollisionShape2D*, per controlar amb què pot xocar; i una *Area2D*, la que indica si el jugador ha rebut mal per tocar el *Tank,* com es pot observar a la **Figura 6.10**.

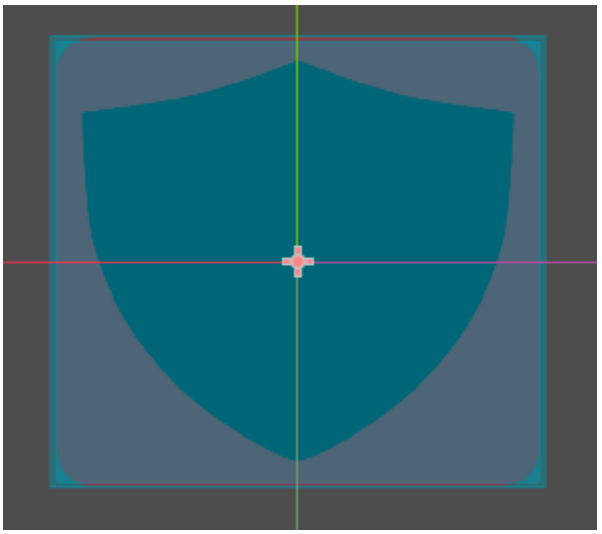

Figura 6.10. Interfície del *Tank*

Tank.gd

```
func _physics_process(delta):
     Delta = delta

     if move:
          velocity = position.direction_to(Global.PlayerPos)
* MAX_SPEED
          if slow:
               velocity = velocity.normalized() * 0.5
          else:
               velocity = velocity.normalized() * 1.5
          move_and_collide(velocity+knockback)

     knockback = lerp(knockback, Vector2.ZERO, 0.1)
```

Com podem veure en el codi del moviment del *Tank*, aquest està basat en seguir al jugador per tocar-lo i fer-li mal. Per fer-ho, s'utilitza la funció *direction_to()* per indicar al *Tank* a on ha d'anar utilitzant la variable global del Singleton que dona la posició del jugador. A més, també tenim unes línies de codi referent als efectes i com influeixen a l'enemic, cosa que s'explicarà amb més profunditat a l'**Apartat 6.7**.

Tank.gd

```
func _process(_delta):
     if vida <= 0:
          if vamp:
               Global.PlayerHealth += 1
          Global.NumEnemics -= 1
          queue_free()
     Time_Inv = Global.SwingTime

func _Damage():
     if !Invulnerable:
          vida -= Global.PlayerDamage
          if retro:
               _knockback()
          Invulnerable = true
          if vida > 0:
               yield(get_tree().create_timer(Time_Inv*1.1),
"timeout")
          Invulnerable = false
```

En aquest codi es pot observar la gestió de la vida del tank, amb una funció *Damage()* que es crida quan l'espasa del jugador el toca, la qual resta el mal que fa el jugador a la seva vida. També podem observar el temps en el que el Tank és invencible un cop ha rebut mal per primer cop. Aquesta funció *Damage()* no variarà molt en els diferents codis dels enemics.

6.5.1.2. Slime

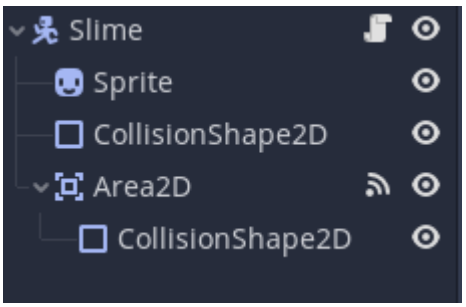

Figura 6.11. Arbre de nodes de *Slime*

L'enemic *Slime* té una naturalesa semblant a la del tank, però no té tanta vida ni és tan gran. Per això, l'inspector que podem veure a la **Figura 6.11,** és el mateix que al *Tank.* Aquest enemic té la peculiaritat que al morir es separa en dos unitats més petites d'ell mateix. Per fer-ho, es crida la funció Damage() a l'entrar a l'àrea de l'espasa i aquesta instància una escena del mateix enemic.

Slime.gd

```
func _Damage():
     if !Invulnerable:
          var Slime_petit =
load("res://Addons/Enemies/Slime_petit.tscn")
          var Slime2 = Slime_petit.instance()
          var Slime1 = Slime_petit.instance()
          get_parent().call_deferred("add_child", Slime2)
          get_parent().call_deferred("add_child", Slime1)
          Slime2.position.x = position.x + 70
          Slime2.position.y = position.y
          Slime2.scale = Slime2.scale/2
          Slime1.position.x = position.x
          Slime1.position.y = position.y
          Slime1.scale = Slime1.scale/2
          queue_free()
```

Com podem observar, Godot té molta facilitat per instanciar elements que hem creat amb anterioritat, ja que amb dues línies de codi es pot afegir fàcilment. Un cop instanciat, simplement donem la posició on està el Slime principal i els reduïm en tamany. Un cop fet tot això s'elimina l'escena Slime.

6.5.1.3. Archer

L'*Archer* és un dels dos enemics del joc que ataca a distància, pel que per implementar-l'ho s'han creat dos escenes noves, la del propi arquer i la del projectil que dispara:

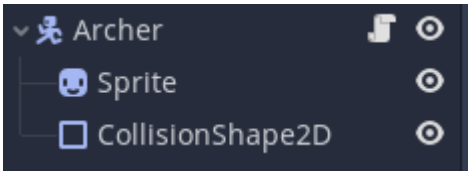

Figura 6.12. Arbre de nodes de *Archer*

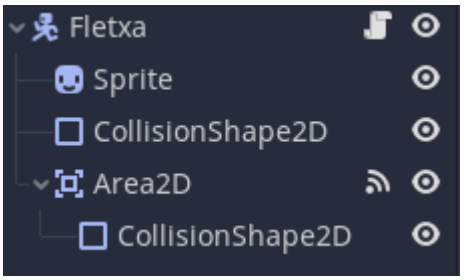

Figura 6.13. Arbre de nodes de *Fletxa*

Com podem veure a les **Figures 6.12** i **6.13,** mentre que l'arquer té un inspector bastant simple, ja que no necessita tocar al jugador per fer-li mal, la fletxa té una descendència més complexa, bastant similar a la del Tank.

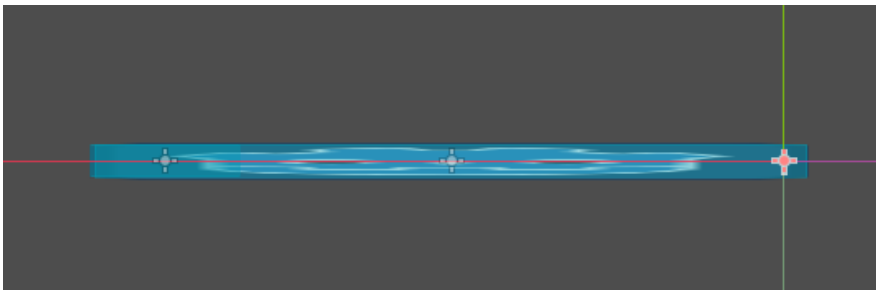

Figura 6.14. Interície de la fletxa

Tot hi tenir un arbre de nodes similar al Tank (**Figura 6.13**), també podem observar com la seva forma és molt més allargada (**Figura 6.14**), a més, l'àrea que controla si ha col·lisionat amb un enemic està a la punta esquerra, tal com ho seria en una fletxa normal.

Archer.gd

```
func _ready():
     Global.NumEnemics += 1
     yield(get_tree().create_timer(1.0), "timeout")
     while(true):
          var fletxa = fletxes.instance()
          get_parent().call_deferred("add_child", fletxa)
          fletxa.position = position
          fletxa.velocity = Global.PlayerPos - position
          fletxa.perseguir = Global.PlayerPos
          yield(get_tree().create_timer(2.0), "timeout")
```

El codi de l'arquer té una rutina que repetirà fins que s'elimini, pel que en aquest cas s'utilitza el clàssic *While(true)* per evitar que el bucle acabi en algún moment. Al ser eliminat per l'enemic al tocar l'espasa, aquest codi s'eliminarà, pel que el while s'acabarà acabant. Dins del bucle podem veure que el que es fa és crear una escena filla fletxa, a la qual se li aplica la posició de l'arquer i se li dóna valor a la variable perseguir, la que utilitzarà el codi de Fletxa.

Finalment, s'utilitza la funció *yield()* com a temporitzador per tornar a repetir el bucle als 2 segons, la variable que se li dona a la funció *create_timer()*.

Fletxa.gd

```
var perseguir = Vector2(0,0)
const MAX_SPEED = 300
var velocity = Vector2(0,0)


func _ready():
     look_at(perseguir)


func _physics_process(delta):
     move_and_collide(velocity.normalized() * delta *
MAX_SPEED)
```

La variable perseguir que se li dona al crear la fletxa la utilitzem per cridar la funció *look_at(),* que servirà per donar-li la direcció a la fletxa amb la qual avançarà l'element amb la funció pròpia de godot *move_and_collide()*.

Al donar valor a la variable perseguir quan es crea i no fer-ho des del codi de Fletxa, serveix perquè s'apunti i vagi recte al jugador en el moment que és disparada, com ho faria una fletxa normal, ja que si li donem el valor dins d'aquesta funció, perseguiria al jugador com si estigués dirigida.

6.5.1.4. Dasher

El dasher és l'enemic més ràpid del joc, i la seva naturalesa és fer tres atacs ràpids per descansar durant uns segons i tornar a començar. Aquesta dinàmica es pot implementar d'una manera molt similar a la de l'Arquer, ja que té una mecànica repetitiva que només s'atura en eliminar al propi enemic.

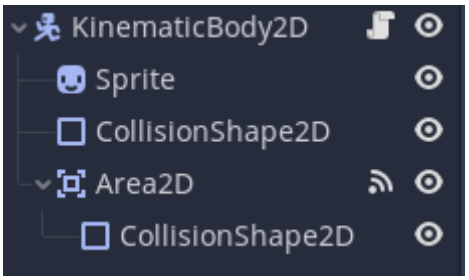

Figura 6.15. Arbre de nodes de *Dasher*

Al ser un enemic que es basa en contactar amb el jugador, té la mateixa estructura que el *Tank* o el *Slime.* Veure **Figura 6.15.**

Dasher.gd

```
func _ready():
     _updatedahs()
     Global.NumEnemics += 1


func _physics_process(delta):
     if move:
          move_and_collide(velocity.normalized() * delta *
MAX_SPEED)
```

Dasher.gd

```
func _updatedahs():
     while(true):
          for i in 3:
               var dir_change_x =
Global.rng.randi_range(-100, 100)
               var dir_change_y =
Global.rng.randi_range(-100, 100)

               var final_dir = Global.PlayerPos +
Vector2(dir_change_x, dir_change_y)
               velocity = final_dir - position

               look_at(Global.PlayerPos)
               yield(get_tree().create_timer(1.0), "timeout")
          move = false
          yield(get_tree().create_timer(2.0), "timeout")
          move = true
```

Com podem veure en el codi anterior, el moviment del *Dasher* es basa en tres funcions, *ready()*, la funció que es crida al instanciar l'escena, el *physics_process(),* la que indica quin moviment fa l'escena, i *updatedash(),* la qual es crida al ready i té un comportament similar a la de l'arquer, ja que té un *while(true)* que crida un petit bucle de 3 moviments ràpids, els quals es fan restant la posició a la que està l'element amb la que vol anar, afegint una petita probabilitat per què no vagin sempre cap al mateix jugador. Un cop fet això, es repeteix el moviment esperant un segon 3 vegades, i un cop fet, para el moviment amb un nou timer al final del bucle *for,* canviant un booleà que s'utilitza a la funció physics_process() per evitar que es pugui moure de nou.

6.5.1.5. Cavaller

El cavaller és l'enemic bàsic més complex de tots, ja que en comptes de basar-se en un atacant que ha de tocar al jugador, com el Tank, té àreas on atacarà quan se li acosti el jugador.

Figura 6.16. Arbre de nodes del Cavaller.

Com es pot veure en la **Figura 6.16**, aquest compte amb vàries Area2D, la que farà mal a l'enemic si el toca, la que detecta si l'enemic està en area d'atac, i la que ataca al jugador, tal com es pot observar a la **Figura 6.17**.

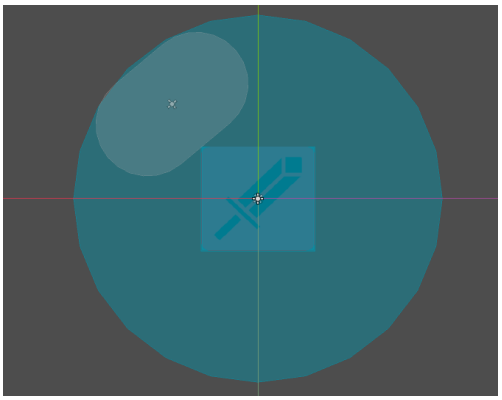

Figura 6.17. Interfície del Cavaller.

Cavaller.gd

```
func _on_Area2D_body_entered(body):
     if body.collision_layer == 2:
          body.Damage(body)

func _on_Deteccio_body_entered(body):
     if body.collision_layer == 2:
          if Atac:
               atkColl.set_deferred("disabled",false)
               yield(get_tree().create_timer(0.2), "timeout")
               atkColl.set_deferred("disabled",true)
               Cooldown()

func Cooldown():
     Atac = false
     yield(get_tree().create_timer(2.0), "timeout")
     Atac = true

func _on_Atac_body_entered(body):
     if body.collision_layer == 2:
          body.Damage(body)
```

El codi que controla els efectes és molt senzill, el primer *_on_Area2D_body_entered()* controla, tal i com fan tots els enemics de contacte, que el jugador rebi el mal cridant la funció damage del codi.

La funció *_on_Deteccio_body_entered()* és la que controla si l'enemic entra al radi d'atac del Cavaller, i fa una crida al *atkColl*(), l'àrea de la pròpia espasa, per activar i desactivar ràpidament l'àrea d'atac, el que, dins del joc, sembla que sigui com una estocada ràpida de l'espasa. Al activar-se a l'instant cada cop que el jugador s'acostava, això feia que fos impossible no rebre mal per part d'aquest enemic, pel que, per balancejar-lo una mica, se li ha afegit un retard cada cop que ataca amb l'espasa, utilitzant la funció *Cooldown()*.

La funció *_on_Atac_body_entered()* és la que fa referència a l'atac de l'espasa del cavaller, que si detecta al jugador dins de l'àrea aquest rebrà mal, així com si toca l'armadura del cavaller.

A més d'això, com ja hem vist a l'arbre de nodes d'aquest enemic (**Figura 6.16**) l'àrea de l'atac només conté un fill *Area2D*, ja que aquest s'utilitza de manera única i rota donant la volta al Cavaller controlant la posició que té el jugador:

Cavaller.gd

```
func PosAtac():
     if Global.PlayerPos.x > position.x and
Global.PlayerPos.y > position.y:
          get_node("Atac").rotation_degrees = 230 #180
     elif Global.PlayerPos.x < position.x and
Global.PlayerPos.y < position.y:
          get_node("Atac").rotation_degrees = 50 #0
     elif Global.PlayerPos.x < position.x and
Global.PlayerPos.y > position.y:
          get_node("Atac").rotation_degrees = 320 #270
     elif Global.PlayerPos.x > position.x and
Global.PlayerPos.y < position.y:
          get_node("Atac").rotation_degrees = 140 #90
```

Aquesta bateria de condicionals controla la posició del jugador i rota l'àrea de l'atac del cavaller, en conseqüència modificant la variable *rotation_degrees*.

6.5.2. Enemic final

L'enemic final representa el cim del viatge del jugador, pel que acaba resultant l'enemic més complicat del joc. Per fer-lo així, s'ha creat un enemic amb el doble de vida que el *Tank* i amb vàries fases a superar amb diferents atacs.

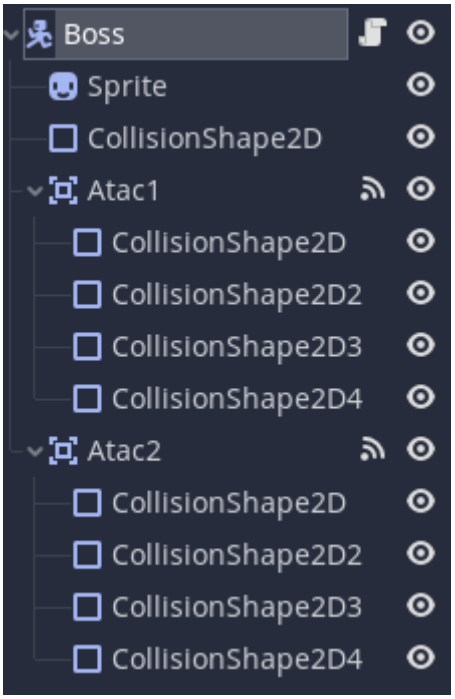

Figura 6.18. Arbre de nodes de *Boss*.

La **Figura 6.18** ens mostra l'arbre de nodes de l'enemic final. Com podem veure, és el més gran de tots, superant al cavaller. També podem apreciar que aquest enemic conta amb dos atacs diferents, els quals ocupen pràcticament la totalitat de l'entorn on el jugador es pot moure (**Figura 6.19**). Aquestes àrees es representen com a columnes de foc que, si el jugador toca, rebrà mal.

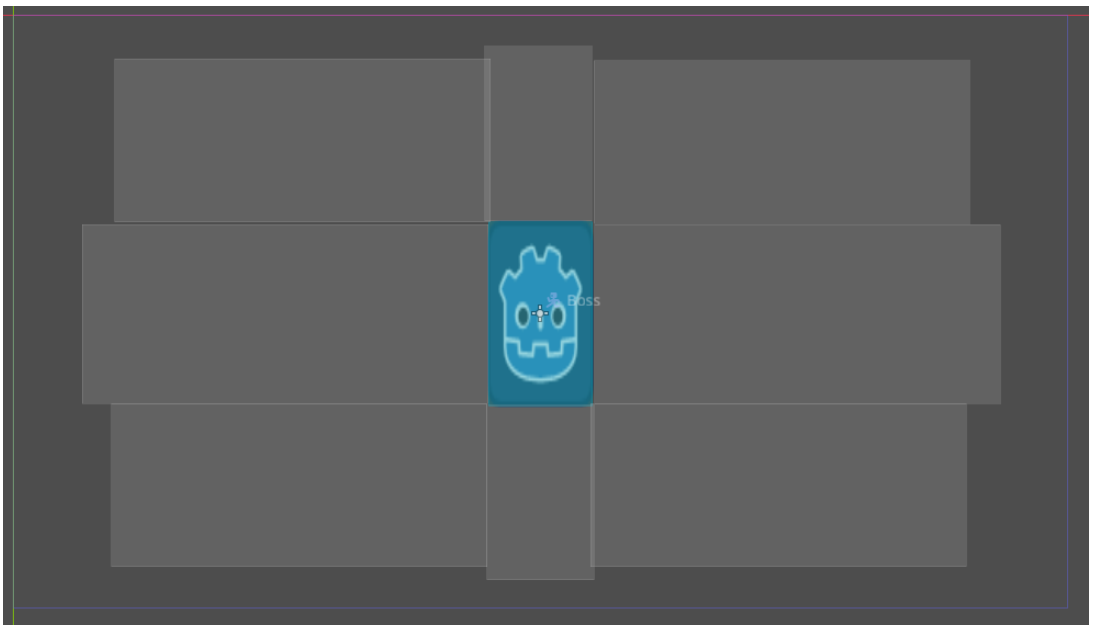

Figura 6.19. Interfície de *Boss.*

Boss.gd

```
var fletxes = preload("res://Addons/Enemies/Bala.tscn")
var move = true
var vida = 10
var Invulnerable = false
var Delta
var vamp = false

var canAttack1 = false
var canAttack2 = false
var fase = 1

func _ready():
     canAttack1 = true
     Fase1()

     Global.NumEnemics += 1
     yield(get_tree().create_timer(1.0), "timeout")

func _on_Area2D_body_entered(body):
     if body.collision_layer == 2:
          body.Damage(body)
```

Boss.gd

```
func _process(_delta):

     if vida <= 7 and vida > 4 and fase == 1:
          fase = 2
          canAttack2 = true
          canAttack1 = false
          Fase2()
     elif vida <= 3 and fase == 2:
          fase = 3
          canAttack2 = true
          canAttack1 = true
          Fase1()

     if vida <= 0:
          if vamp:
               Global.PlayerHealth += 1
          Global.NumEnemics -= 1
          queue_free()
```

En el codi vist anteriorment podem observar com les funcions *ready()* i *process()* controlen quan es fa cada atac. La primera cridant la fase 1 del codi al ser la que treballa al crear-se la escena, mentre que la segona controla quan canviar de fase a través de la vida que li falta a l'enemic final.

A més, al ser els atacs en bucle, cada cop que iniciem un o parem l'altre, hem de modificar els booleans que les aturen.

Boss.gd

```
func Fase2():
     while(canAttack2):
          for i in get_node("Atac1").get_children():
               i.disabled = true
          for i in get_node("Atac2").get_children():
               i.disabled = false
          yield(get_tree().create_timer(4.0), "timeout")
          for i in get_node("Atac1").get_children():
               i.disabled = false
          for i in get_node("Atac2").get_children():
               i.disabled = true
          yield(get_tree().create_timer(4.0), "timeout")

func Fase1():
     while(canAttack1):
          for i in 20:
               var fletxa = fletxes.instance()
               get_parent().call_deferred("add_child",
fletxa)
               fletxa.position = position
               var randDirX = Global.rng.randi_range(-90,90)
               var randDirY = Global.rng.randi_range(-90,90)
               fletxa.velocity = Vector2(randDirX,randDirY)

          yield(get_tree().create_timer(2.0), "timeout")
```

Aquí podem veure els dos tipus d'atacs que té l'enemic final. El primer, Fase1(), utilitza un bucle que crea 20 projectils cap a totes direccions aleatòriament, instanciant l'escena bala i donant-li una posició i una direcció. Això es repeteix cada dos segons i fins que no es passi a la fase 2.

A la fase 2, s'aturen els atacs de la fase 1 i s'activen les àrees comentades anteriorment, de manera que cada un dels fills de Atac1 i Atac2 (vistos a **Figura 6.19**) activen o desactiven les seves *Area2Ds* filles de forma intermitent, per donar-li espai de joc al jugador.

Un cop el jugador ha aconseguit baixar la vida del Boss fins a 3, l'enemic final activa la fase 3, que és la combinació de les dues mencionades anteriorment, cosa que dificulta al jugador atacar i moure's.

6.6. Implementació dels efectes

Els efectes o habilitats del jugador són la part més important del joc, ja que ens indiquen com han de interactuar els enemics quan el jugador els ataqui, o com ha d'actuar el propi personatge.

Per fer-los, s'ha creat una nova escena efecte amb un identificador que utilitzarem per saber quina habilitat s'ha d'activar dins el codi pertinent.

Figura 6.20. Arbre de nodes d'Efecte.

Com podem observar a la **Figura 6.20**, l'arbre de nodes d'efecte no és molt complex, es basa en una *Area2D* que el jugador tocarà si vol agafar l'habilitat.

```
Efecte.gd

var id = "non"
var efe = []
export var AugmentSize = 30

signal EffectAdded

func _ready():
     id = Global.EfecteRandom()
     AugmentSize = Global.rng.randi_range(20,60)
     connect('EffectAdded',
get_parent().get_parent().get_node("Inventory"), 'addEffect')
     connect('EffectAdded', get_parent().get_node("Player"),
'BoostEffects', [id])
```

Abans però que el jugador l'agafi, l'efecte ja ha escollit aleatòriament quina variable *id* tindrà, ja que serà l'identificador que s'utilitzarà per saber quina efecte ha d'actuar. Per escollir-lo, es crida una funció que té el Singleton anomenada *EfecteRandom()*.

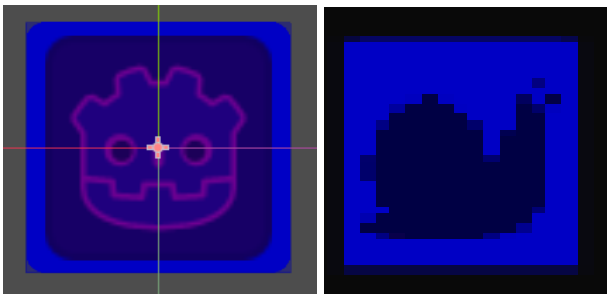

Figura 6.21. Efecte abans d'escollir id (esquerre) i després(dreta) escollint l'efecte *slow*.

A més, també se li indica un augment de mida aleatoria entre 20 i 60, per donar-li un estil més *Roguelike*.

Singleton.gd

```
     Efectes = [["c_StableDamage",10], ["Double_Dash",20] ,
["root",20], ["vampire",15], ["retro",20], ["slow",15]]

func EfecteRandom():
     if Efectes.size() > 0:
          var pick = rng.randi_range(1,ProbTotal)
          var sumaAux = 0
          var i = 0

          while i < Efectes.size() and sumaAux < pick:
               sumaAux += Efectes[i][1]
               if sumaAux < pick:
                    i += 1

          var aux = Efectes[i]
          ProbTotal -= aux[1]
          Efectes.erase(Efectes[i])

          return aux[0]
     return "non"
```

Aquesta funció escull aleatòriament un dels efectes de la llista de parelles que estan disponibles i les posa a una llista anomenada *PlayerEffects*, la qual, a l'estar al *Singleton*, es podrà accedir des de qualsevol escena del joc. La funció utilitza el segon element de la parella per veure la probabilitat que té de que surti, i utilitza una funció basada en CDF (Cumulative Distribution Function).

Efecte.gd

```
func _on_Area2D_body_entered(body):
     if body.collision_layer == 2:
          efe.append(id)
          efe.append(AugmentSize)
          Global.PlayerEffects.append(efe)
          Global.SwordSize += AugmentSize
          body.UpdateSword()
          emit_signal("EffectAdded")
          queue_free()
```

Aquesta és la funció que es crida quan el jugador toca l'efecte: primerament, es comprova que qui ha detectat l'*Area2D* sigui un objecte de la *layer* jugador. Seguidament, afegeix la mida i la id a la llista dels efectes del jugador, i augmenta la mida de l'espasa cridant la funció *UpdateSword()*. Finalment, s'elimina l'efecte, ja que ja s'ha enviat tota l'informació necessària al *Singleton*, pel que ja no ens serveix de res mantenir aquest efecte viu.

Un cop l'efecte és a la llista de *PlayerEffects*, aquest pot interactuar de dues maneres diferents: modificant alguna característica del jugador, o canviant algun comportament dels enemics als que toca.

6.6.1. Efectes dirigits al jugador

Quan el jugador rep un efecte, es crida una funció anomenada *BoostEffects(),* que actualitza les variables que ens permet activar les habilitats que toquin.

Player.gd

```
func BoostEffects(effect):
     if effect == "c_StableDamage":
          if Global.PlayerDamage == 2:
               Global.PlayerDamage = 1
               cursed = false
          else:
               Global.PlayerDamage = 2
               cursed = true

     if effect == "Double_Dash":
          if dashTimer == 3:
               dashTimer = 1
          else:
               dashTimer = 3
```

Aquesta funció detecta la *id* de l'efecte que li han passat i comprova si és una de les dos variables que afecten al jugador:

- Si és *c_StableDamage*, modificarà el mal del jugador i activarà el booleà que permet al jugador moure's o no mentre ataca.
- Si és *Double_Dash,* modificarà el temps que tarda el jugador en poder tornar a utilitzar el Dash.

Ja que aquesta funció es crida tant quan el jugador obté l'habilitat com quan la perd, s'ha de tenir en compte les dues opcions i fer que quan es cridi faci el contrari del que ja ha fet.

6.6.2. Efectes dirigits als enemics

Els efectes dirigits a l'enemic són més complicats de cridar ja que s'ha de fer referència al que s'està atacant en concret. Per fer-ho, hem d'utilitzar una particularitat que té Godot que resulta molt interessant: si utilitzem el *Signal* que té Godot per detectar, si ha entrat un element en una *Area2D,* es pot fer referència a ell utilitzant la variable *body,* donada per referència.

SwordArea.gd

```
func Basic_Atack(body):
     if body.collision_layer == 1:
          Global.SwordEffects(body)
          body._Damage()
```

Per això només hem de cridar la funció del Singleton *SwordEffects()*, passant-li per referència el *body*, que és l'enemic tocat.

Singleton.gd

```
func SwordEffects(body):
     for i in PlayerEffects:
          body.ActivateEffects(i[0])
```

Dins d'aquesta funció del Singleton, i tenim un bucle que passa per tots els efectes del jugador i, gràcies a *body,* els activa a l'enemic que s'ha tocat.

La funció *ActivateEffects()* és molt semblant a tots els enemics, ja que la funció és posar a *true* els booleans que activaràn les habilitats del jugador. Per això es mostrarà el codi del Tank, ja que és el que més habilitats li afecten (tenir en compte que hi ha efectes com el retrocés que no afecteran a un enemic com el *Dasher*, al tenir un sol punt de vida).

Tank.gd

```
func ActivateEffects(effects):
     if !Invulnerable:
          if effects == "root":
               move = false
               yield(get_tree().create_timer(1.0), "timeout")
               move = true
          if effects == "slow":
               slow = true
               yield(get_tree().create_timer(3.0), "timeout")
               slow = false
          if effects == "retro":
               retro = true
          if effects == "vampire":
               vamp = true
```

Aquí podem veure que aquesta funció no és el que fa els efectes com a tal, sinó que l'objectiu és, si l'enemic no és invulnerable, activar l'efecte que es doni per referència a través de l'identificador.

Cada booleà de cada efecte funciona diferent, però principalment s'activen amb un condicional en el moment de rebre mal:

Tank.gd

```
func _physics_process(delta):
     Delta = delta
     if move:
          velocity = position.direction_to(Global.PlayerPos)
* MAX_SPEED
          if slow:
               velocity = velocity.normalized() * 0.5
          else:
               velocity = velocity.normalized() * 1.5
          move_and_collide(velocity+knockback)

     knockback = lerp(knockback, Vector2.ZERO, 0.1)
```

- *Root:* posa a fals la variable move durant 1 segon, el que no permet moure a l'enemic.
- *Slow*: multiplica la velocitat del jugador per 0.5, per fer-lo més lent durant 3 segons.
- *Retro*: aquest efecte s'activa a la funció _Damage(), cridant una funció *knockback()* que aplica una força a l'enemic al costat contrari al que està el jugador

Tank.gd

```
func _knockback():
     if Global.PlayerPos.x < position.x:
          knockback = Vector2.RIGHT * 25
     elif Global.PlayerPos.x > position.x:
          knockback = Vector2.LEFT * 25
     knockback = knockback.move_toward(Vector2.ZERO, 200 *
Delta)
```

- *Vamp:* l'efecte vampir es crida al moment en el que la vida de l'enemic arriba a 0, i, si està actiu, suma 1 punt de vida al jugador.

```
Tank.gd

func _process(_delta):
     if vida <= 0:
          if vamp:
               Global.PlayerHealth += 1
          Global.NumEnemics -= 1
          queue_free()
     Time_Inv = Global.SwingTime
```

6.7. Implementació de la interfície

6.7.1. Barra de vida

La barra de vida del jugador es mostra en la cantonada superior esquerra i, al ser representat per un número enter, el personatge té el número de cors que mostra la variable *PlayerHealth* del *Singleton*.

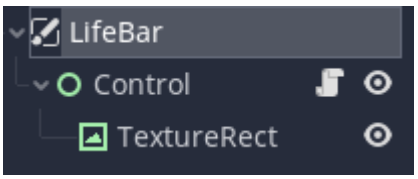

Figura 6.22. Arbre de nodes del *Boss*.

L'arbre de nodes (**Figura 6.22**) és molt senzill, però el TextureRect té una peculiaritat que ens és molt útil per mostrar la vida de la manera que volem: a l'inspector d'aquest node es pot afegir un *Sprite* (que seran els cors representant la vida del jugador), té una opció per fer que a l'estirar aquest *Sprite*, es mostri com un *Tile*, és a dir, una repetició de la mateixa foto, en comptes de fer-se més gran.

```
LifeBar.gd

func _process(_delta):
     if Global.PlayerHealth > 0:
          $TextureRect.rect_size.x = Global.PlayerHealth * 64
```

Com podem veure, el codi modifica la llargada per x del node multiplicant la vida del jugador per 64, la distància en x de cada cor.

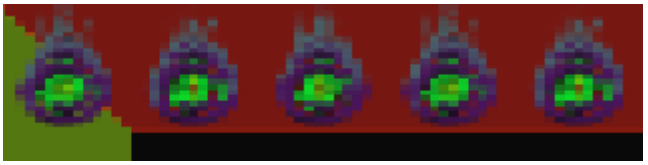

Figura 6.23. La vida del jugador mostrada dins el joc.

A la **Figura 6.23** es pot observar el resultat final d'aquest codi, amb el jugador tenint 5 vides.

6.7.2. Menú inicial

El menú principal del joc és l'encarregat de gestionar els canvis d'escenes a través dels botons que el jugador prem. Per fer-ho, s'ha de crear una escena nova, que es modificarà dins de les opcions de projecte per què sigui la primera en aparèixer al iniciar el joc.

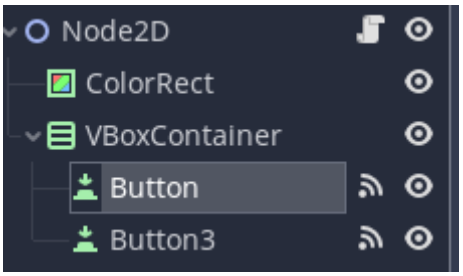

Figura 6.24. Arbre de nodes del Menú Principal.

Aquesta escena consta d'un *VBoxContainer*, que serveix per agrupar els fills per pantalla, i dos fills botons, el primer que iniciarà el joc, i el segon que en sortirà.

MenuPrincipal.gd

```
func _ready():
     $VBoxContainer/Button.grab_focus()
     get_tree().paused = false


func _on_Button_pressed():
     Global.reset()
     get_tree().change_scene("res://Scenes/LvL.tscn")


func _on_Button3_pressed():
     get_tree().quit()
```

Com es pot veure, el codi consta de dos *signals* connectats als botons a través dels nodes de Godot, els quals es cridaràn al moment en que el jugador prem un dels botons. Si el jugador prem iniciar partida, aquest cridarà la funció *reset(),* des del Singleton, per reiniciar tots els elements i tornar a començar la partida.

Això ho fem aquí perquè el jugador por accedir a aquest menú desde la pausa, havent de reiniciar el joc i totes les variables del Singleton que necessitin estar en l'estat inicial. Un cop fet això, es crida la funció *change_scene()*, pròpia de godot, la qual canvia l'escena a la que li donem.

Pel codi referent al botó Sortir, simplement s'ha de cridar la funció *quit()*.

Figura 6.25. Resultat final de Menú Principal.

6.7.3. Pantalla final de partida

La pantalla final del joc és molt semblant a la del menú principal, també conta amb dos botons, un per sortir del joc i l'altre per indicar que es vol tornar al menú principal (**Figura 6.26**).

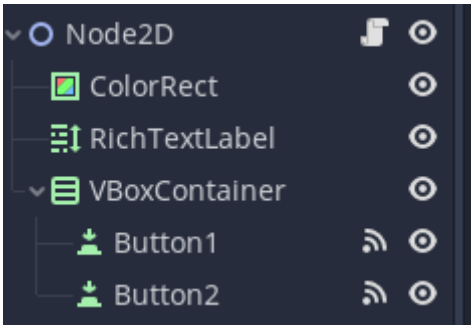

Figura 6.26. Arbre de nodes de la Pantalla Final.

A part d'això, també se li ha afegit un *RichTextLabel*, per posar un text a sobre dels botons que mostren al jugador el quadre de text: Game Over.

Player.gd

```
func _process(_delta):
     Global.PlayerPos = position
     vida = Global.PlayerHealth
     if vida == 0:

get_tree().change_scene("res://Scenes/GameOver.tscn")
          Global.reset()
          get_parent().queue_free()
     if Input.is_action_just_pressed("Attack"):
          is_attacking = true
          var timeout = Global.SwingTime*1.1
          yield(get_tree().create_timer(timeout), "timeout")
          is_attacking = false
```

Per entrar en aquesta escena, es crida la funció *change_scene()* des del *Script* jugador, a *process()*, ja que en el moment en què aquest tingui 0 de vida es canviarà a l'escena final.

6.7.4. Menú de Pausa

El menú de pausa, a diferència del menú principal o la pantalla de fi del joc, no consta d'una escena per ella mateixa, sinó que és una part de l'escena nivell. Això s'ha fet així perquè, si es creava una escena diferent, el Godot no realitzava la funció que permet pausar el joc correctament.

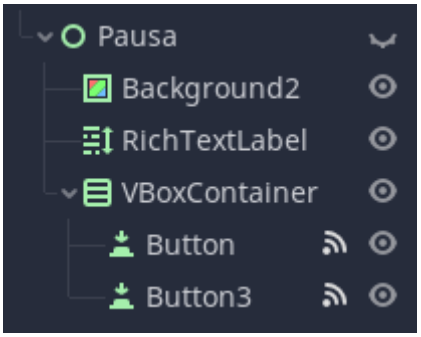

Figura 6.27. Arbre de nodes del Menú de Pausa.

L'arbre de nodes de la pausa és idèntic al del menú, però, al ser dins de l'escena Nivell, el *Background2* té una opacitat menor, per poder veure el joc pausat per darrera, com es veu en la **Figura 6.28**.

Figura 6.28. Arbre de nodes del Menú de Pausa.

Pausa.gd

```
func _ready():
     $Pausa/VBoxContainer/Button.grab_focus()

func _on_Button3_pressed():

get_tree().change_scene("res://Scenes/MenuPrincipal.tscn")

func _on_Button_pressed():
     get_tree().paused = false
     $Pausa.visible = false

func _process(_delta):
     if Input.is_action_just_pressed("Pause"):
          $Pausa/VBoxContainer/Button.grab_focus()
          $Pausa.visible = not $Pausa.visible
          get_tree().paused = not get_tree().paused
```

Com podem veure en el codi anterior, el menú de pausa consta de 2 botons, el primer canvia d'escena al menú principal (de la mateixa manera que es fa en la pantalla *GameOver* o al Menú Principal), mentre que l'altre utilitza una variable pròpia de Godot que ens serveix per pausar tots els processos que estan funcionant.

A més, aquí tenim *l'Input* que controla quan el jugador prem el botó de pausa, cosa que pausa o reanudació el joc i fa visible el menú que és ocult en un principi.

La variable *paused* és molt útil per aturar tots els processos del joc, però si es fa servir directament, deixa absolutament tot el joc pausat, pel que no es podrà interactuar amb el menú.

Per evitar això, Godot té una opció a l'inspector de cada node per indicar que, en cas de que es pari el joc, aquest element i tots els fills segueixin funcionant, com podem observar en la **Figura 6.29**.

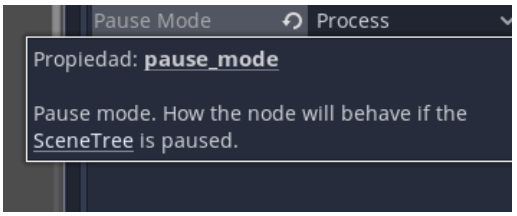

Figura 6.29. Estat de la propietat de pausa del node *PauseMenu.*

6.7.5. Gestió d'efectes

Com ja hem comentat anteriorment, una de les intencions al crear GreaterSword és que tingui un apartat de *DeckBuilding,* pel que és necessari que, en tot moment, el jugador tingui l'opció d'eliminar els efectes que no li convenen. Per fer-ho, s'ha creat un inventari al qual el jugador pot accedir mentre juga, amb una estructura molt similar a la del menú de Pausa. Veure **Figura 6.30.**

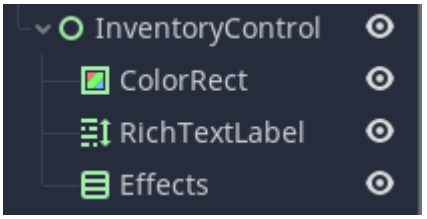

Figura 6.30. Arbre de nodes de l'Inventari.

L’inventari també està ocult a l’iniciar el joc, i al moment de prémer determinat botó, aquest es fa visible pel jugador.

Figura 6.31. Arbre de nodes de l’Inventari.

Com es pot observar en la **Figura 6.31**, el menú mostra l’efecte que ha obtingut el jugador, i li permet eliminar-lo al prémer el botó.

Inventory.gd

```
var inventory
var player

signal UpdateSize

func _ready():
     inventory = $InventoryControl
     player =
get_parent().get_node("MapObjects").get_node("Player")

     connect('UpdateSize', player.get_node("Sword"),
'updateSize')

func _process(delta):
     if Input.is_action_just_pressed("Inventory"):
          showInventory()

func showInventory():
     inventory.visible = not inventory.visible
```

En el codi anterior podem observar com es controla si s'ha pres el botó referent a l'inventari, i si ho fa, aquest mostra al jugador el menú. A més, també connecta a través de codi el signal *UpdateSize*, que farem servir més endavant per cridar una funció d'una escena molt llunyana a aquesta, la que s'encarrega d'actualitzar la mida de l'espasa.

Per gestionar tots els efectes que el jugador pot escollir, hem de tenir una funció que modifiqui l'*array* en la que estàn els efectes i els doni en alta i en baixa quan sigui necessari.

Inventory.gd

```
func addEffect():
     var button = Button.new()
     var newEffect = Global.PlayerEffects.size()-1
     button.text = "Effect:" +
Global.PlayerEffects[newEffect][0] + " | Augment Size:" +
str(Global.PlayerEffects[newEffect][1])
     inventory.get_node("Effects").add_child(button)
     button.connect("pressed", self, "removeEffect",
[Global.PlayerEffects[newEffect], button])
     button.connect("pressed", player, "BoostEffects",
[Global.PlayerEffects[newEffect][0]])
```

Aquí podem veure les dues funcions que s'encarreguen d'eliminar o treure els efectes. La funció *AddEffect()* es cridarà quan el jugador toqui un efecte, cosa es pot fer gràcies als *Signals* per codi. Aquesta funció es basa en instanciar un nou botó i posar-li com a text el nom de l'efecte i la llargada del mateix. Un cop creat aquest nou botó, el connecta a través de codi a la funció *removeEffect*, la qual ens servirà per poder cridar aquesta funció quan el jugador toqui el botó, eliminant l'efecte que s'ha pres.

A més, també es connectarà la funció *BoostEffects*, per eliminar els efectes que el jugador s'havia posat al tenir aquesta habilitat activa.

Inventory.gd

```
func removeEffect(effect, button):
     Global.SwordSize -= effect[1]
     emit_signal("UpdateSize")
     Global.PlayerEffects.erase(effect)
     button.queue_free()
```

La funció *removeEffect* crida el *Signal* connectat a la funció *ready()* per modificar la mida de l'espasa, es resta aquesta a la variable del Singleton i elimina l'efecte de la llista global d'efectes del jugador.

6.8. Implementacions artístiques

Dins aquest apartat és parlarà sobre tota la tecnologia implementada per a la gestió de les animacions i *Sprites* del videojoc.

6.8.1. Animacions

Les animacions a Godot es poden implementar de dues maneres, amb un node *AnimatedSprite*, o un node *AnimationPlayer*. Les dues compleixen propòsits diferents.

- El node *AnimationPlayer*, es un node que té tot el material que es necessita per fer animacions. Es poden modificar la posició d'objectes per a animar per codi o afegir animacions a triggers en concret i tot està organitzat per capes. És el compendi de tot el que es pot fer en animació *2D* a *Godot*.

- El node *AnimatedSprite* en canvi, és una petita funció del node *AnimationPlayer*, que simplement és un contenidor on organitzar i canviar la velocitat de les animacions que es tenen a partir de *Sprite Sheets*. Aquesta és la usada en el treball, ja que totes les animacions són en *2D* i no necessiten cap tractament, ja que els frames d'animació ja són tot el necessari. L'únic que es modifica és la velocitat de reproducció.

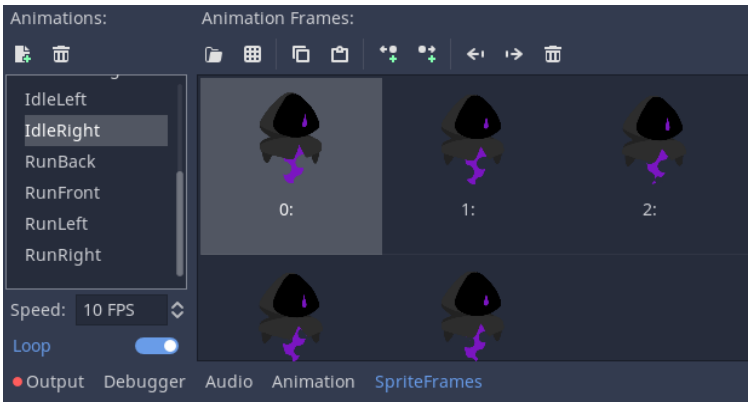

Figura 6.32. Interfície del node *AnimatedSprite* s'observa la llista d'animacions del personatge principal a l'esquerra, la velocitat de reproducció en FPS (Frames Per Second) i si es vol que sigui en Bucle (opció Loop).

Llavors, per a reproduir les animacions en el videojoc, es controla tot per codi en el node *AnimatedSprite*. Amb la simple utilització de la funció *animatedSprite.play("NomAnimació")* es reprodueixen les animacions i les que són en bucle continuen després de la primera passada.

Per últim, el control de quina és l'animació que es reprodueix tenint en compte la direcció, s'han usat variables que guarden aquesta informació i amb una bateria de condicionals es dóna pas a l'animació pertinent.

Player.gd

```
func dashPlayer(direction):
     if direction == "right":
          animatedSprite.play("DashRight")
     elif direction == "left":
          animatedSprite.play("DashLeft")
     elif direction == "front":
          animatedSprite.play("DashFront")
     else:
          animatedSprite.play("DashBack")
```

6.8.2. Sprite de l'Arma

Al tenir una mida canviant a l'arma, *l'Sprite* s'ha d'anar modificant, i al tenir mides d'arma controlades amb una variable i no vàries mides fixes, no es pot crear un nombre de imatges d'arma i indexar-les per mida, ja que se'n necessiten molts. Tampoc serveix canviar l'escalat de la imatge perquè llavors queda deformada i no és el resultat que volem.

Figura 6.33. *Sprite* de l'espasa.

Una solució senzilla a aquest problema és usar la informació de l'*Sprite*, és a dir, baixar a nivell de píxel i tallar la imatge on convingui segons la mida, tenint en compte que es fa una imatge molt allargada i així es pot tallar-la on convingui.

SwordTexture.gd

```
var swordSprite = load("res://Addons/Player/Sword/sword.png")
var imageTexture : ImageTexture = null
var dynImage : Image = null

func _ready():
     imageTexture = ImageTexture.new()
     dynImage = swordSprite.get_data()

     imageTexture.create_from_image(dynImage)
     self.texture = imageTexture

     update_image(400)
```

Amb la creació de l'arma també es crea la textura dins *SwordTexture.gd*, com es veu al codi superior, primer es guarda l'*Sprite* de l'arma i es creen variables necessàries per fer el canvi de mides:

- *imageTexture*, que és el contenidor de la imatge final, que s'iguala a la propietat de textura del node *Sprite* de *Godot*.
- *dynImage*, que és el contenidor de la informació interna de la imatge per poder canviar-la a mesura que avança el joc.

Llavors es torna a posar la informació a dins de la textura de la imatge i es crida la funció *update_image(swordSize)* per posar la mida inicial.

SwordTexture.gd

```
func update_image(swordSize) -> void:
     dynImage.lock()
     for i in range(swordSize*4, dynImage.get_width()):
          for j in range(dynImage.get_height()):
               dynImage.set_pixel(i, j, Color(0.0, 0.0, 0.0,
0.0))
     dynImage.unlock()

     imageTexture.set_data(dynImage)
     texture = imageTexture
```

Aquesta funció fa un bucle des del rang de l'espasa fins al final de l'*Sprite* i posa tots els píxels de *dynImage* transparents, aconseguit d'aquesta manera només es vegi la part inicial de la imatge. També s'usa les funcions pròpies de Godot de les variables Image lock() i unlock() per a permetre la edició de la informació de l'imatge. Finalment es torna a posar la informació a dins de la textura de la imatge.

6.8.3. Map

Figura 6.34. *Sprite* de *Background*.

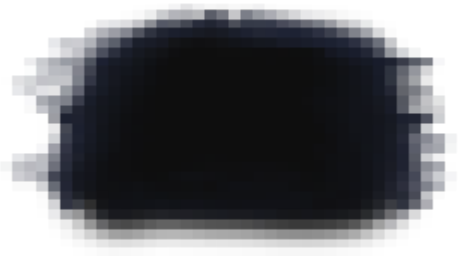

Figura 6.35. *Sprite* de les portes.

6.8.4. Filtre pixelat

El filtre de pixelació que s'ha usat és basada en la implementació d'un shader oficial de la documentació de Godot [godotshaders2021] que s'utilitza en nodes en concret, però adaptat a pantalla completa.

```
Shader Node

shader_type canvas_item;

void fragment() {
    vec2 pixelatedUV = round(SCREEN_UV * float(400)) /
float(400);
    vec4 screenColor = textureLod(SCREEN_TEXTURE,
pixelatedUV, 0.0);
    COLOR = screenColor;
}
```

Bàsicament agafa les textura que hi ha dins un rectangle, de la pantalla completa en el nostre cas, la multiplica per un escalar fent que cada píxels sigui una agrupació de píxels iguals, llavors arrodoneix el resultat. Finalment torna a dividir la textura pel mateix escalar, aconseguint així, en comptes de revertir la multiplicació, que es perdi resolució a la imatge que hi havia en pantalla inicialment.

Amb aquest procés s'aconsegueix un efecte semblant a la pèrdua de qualitat d'imatge, però es pot aplicar a les parts que es vulguin de l'arbre de nodes, canviant simplement a la canvas *layer* els nodes que no es vol, per exemple els menús.

6.9. Proves

Durant el desenvolupament del videojoc s'han realitzat proves usant la línia de comanda, sobretot de Godot, usant la funció *print()* en execució del videojoc per mostrar les variables i veure si el flux d'execució anava en la direcció esperada.

També *Godot* té un sistema d'errors que ens ha ajudat per trobar les referències correctes al buscar els errors a internet i a solucionar-los amb l'explicació proporcionada pel propi *Godot*. També té un inspector de variables al moment d'execució i quan es para la mateixa, que ens ha servit per veure quina variable no estava en el valor esperat en un moment donat. Veure **Figura 6.36**.

Figura 6.36. Sortida a *Godot*.

Per últim Godot té l'opció de observar les col·lisions en execució, que ha servit per a millorar l'atac del jugador i altres ajustaments. Veure **Figura 6.37.**

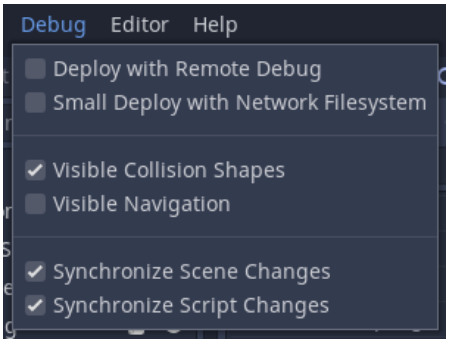

Figura 6.37. Opcions bàsiques de debugatge a *Godot*, *Visible Collison Shapes*.

Referent als contratemps, el pas més problemàtic ha set el procés de combinació de la feina dels 2 integrants del grup al final del desenvolupament. Per molt que la majoria de parts del videojoc són independents de les altres, també n'hi ha que al ajuntar ens hem trobat amb diferents problemes. Sobretot en la gestió de sales, ja que sempre s'està en un mateix escenari i s'instancien els objectes en aquell. Llavors, al passar per una porta, s'ha d'eliminar i crear els components de l'habitació, com els efectes al sortir i entrar.

D'altre banda, la gestió de sales que en principi era totalment independent ens va donar error al mostrar-les per la gestió de graelles a *Godot* canviant x per y.

Per últim la funció bàsica de Godot d'afegir elements dins una llista va fallar a l'utilitzar parelles d'elements, i a les comprovacions d'igualació, ja que vam assumir que funcionaven bé a causa de ser un llenguatge no tipat.

7. Resultats

7.1. Legislació i normativa vigent

El prototip del videojoc no presenta cap problema d'aspecte legislatiu. No es guarda cap tipus d'informació de caràcter personal per tant, no s'aplica la LOPD (Llei Orgànica de Protecció de Dades). Tampoc comporta cap activitat econòmica per tant, no s'aplica LSSICE (Llei de Serveis de la Societat de la Informació i Comerç Electrònic).

D'altre banda, el problemes relacionats amb drets d'autor *Copyright*, no ens han de suposar cap problema si el volguéssim comercialitzar.

En el cas que el joc acabi al mercat, *Godot* és creat i distribuït sota una llicència *MIT*, indicant que no imposa cap impediment ni compensació econòmica ja que és un iniciativa *Open Source*. L'única obligació és la inclusió del text de la llicència [JuanLinietskyArielManzur20072014]

7.2. Classificació PEGI

La classificació PEGI (*Pan European Game Information*) són uns descriptors i classificació d'edat oficials d'Europa. Aquesta classificació no és de caire obligatori sinó que té un objectiu purament informatiu i de recomanació al comprador del videojoc, llavors no impedeix que cap franja d'edat pugui jugar.

Figura 7.1. Etiquetes del sistema de classificació *PEGI*, part superior edats, part inferior aspectes del videojoc.

En el cas de GreaterSword, l'única acció que comporta problema és l'existència de **violència**, però tenir un estil enfocat a la fantasia i no al realisme i sense tractar directament el tema de la violència es qualifica el videojoc com a *PEGI* 7, com el videojoc Moonlighter.

7.3. Resultat final

Figura 7.2. Menú Principal.

Figura 7.3. Pantalla inicial amb el primer efecte que podem obtenir.

Figura 7.4. Menú de l'inventari *ingame*.

Figura 7.5. Combat the *GreaterSword*.

Figura 7.6. Menú de pausa.

Figura 7.7. Combat de l'enemic final.

Figura 7.8. Pantalla de derrota.

Figura 7.9. Pantalla de victòria.

8. Conclusions

Primerament, hem après a realitzar un projecte a més gran escala amb el motor Godot. Això ens permet que, si en algun moment tornem a aquest programari, tinguem més facilitat en veure la procedència dels errors i com solucionar-los, a part de obtenir lleugeres en la implementació i les característiques del motor.

El procés de disseny de mecàniques del videojoc és la part que més s'ha gaudit del projecte. Les reunions amb el tutor del projecte i els debats generats a partir de les decisions que s'anaven prenent a mesura que s'implementa el videojoc, han estat encertades en retrospectiva. Però al moment de posar-se a fer la feina és més complicat i menys entretingut a causa de la falta d'experiència i els continus problemes amb els que ens anàvem topant.

També comentar que l'experiència de la creació d'un videojoc major, sobretot en l'apartat tecnològic, genera un canvi de perspectiva a l'hora de la programació de diversos mètodes que dins anteriors assignatures de la carrera no s'haguessin donat per bones, ja que, tot i que és obvi que proporciona el millor resultat, és negligible dins les situacions que es poden trobar durant el desenvolupament d'un videojoc.

8.1. Assoliment del requisits

Tot que creiem que hem realitzat una bona feina, quan es dóna un temps límit per entregar un projecte d'aquestes proporcions, sempre s'acaben veient molts aspectes en el que es podria millorar i afegir contingut. Pel que fa a la sensació dels membres del grup és agredolça, ja que hem complert les tasques planificades, però creiem que ens hem quedat curts en alguns aspectes comentats a fases prèvies a l'inici del projecte.

Comentaris personals de cada integrant:

- Miquel Sangrador

En Miquel tenia la sensació inicial de treballar la part estètica del projecte amb més profunditat i realitzar més assets. A causa d'una falta de motivació, al acabar la recerca de l'apartat artístic, va deixar de banda més aquesta faceta i es va anar decantant a millorar implementacions i no a treballar en estética purament.

Això ha provocat una certa mancança d'estètica al videojoc, tenint en compte els objectius inicials. Però assolint els objectius de recerca que volia obtenir durant el projecte, sense assolir tots els pràctics que es volien prèviament.

En aspectes generals en Miquel creu que la feina feta ha estat més reduïda del que es volia abastar, però a l'acabar la redacció de la memòria, ha vist que ha estat una bona feina i un treball enriquidor pels integrants del grup.

- Blai Vilanova

En Blai s'ha encarregat d'implementar tots els enemics i com interactuen amb els efectes, i, tot hi que creu que falta algun enemic i algun efecte per donar-li més amplitud al joc, té la sensació que la feina feta és bona. La interacció dels elements que apareixen per pantalla funcionen bé amb tots els efectes posats, i els enemics tenen comportaments molt divertits d'implementar i debugar (sobretot els últims creats, el cavaller i l'enemic final).

Tot hi que en Blai creu que es podia haver fet millor la feina, també és cert que el joc presentat no és l'última versió que s'implementarà, sinó que és més aviat el tret de sortida del projecte, pel que segurament, a curt termini, afegirà més contingut al joc.

8.2. Desviació de la planificació inicial

Els blocs de treball planificats s'han assolit amb la temporització esperada. La gestió dins dels blocs descrits a l'**Apartat 3.3** s'ha portat a terme a mesura que s'anaven fent reunions amb el tutor del projecte, exponent la feina que s'havia dut a terme durant les setmanes abans de la reunió i decidint la que es faria a continuació.

Alguns dels *sprints* en els que s'havien plantejat qüestions d'implementaicó que generen problemes tecnològics. Si no es podia arribar a l'objectiu, la feina per a la següent setmana es duplicava o fins i tot retrassava el projecte, però entenem que és la norma en qualsevol projecte de programació. Està clar però, que en aquest casos es valora el problema i el grup de treball ajuda l'integrant per sortir de la situació el més ràpid possible.

9. Treball futur

Com hem vist durant tot el projecte, el gènere *Roguelike* és dels més interessants que s'han introduït dins el mon dels videojocs i ens han portat autèntiques obres d'art en forma de diferents tipus de videojocs. Nosaltres, com a jugadors, també hem pogut apreciar l'esforç i dedicació dels dissenyadors d'aquests projectes, alhora de veure com continuen amb les seves obres.

La gran majoria de jocs d'aquest estil opten per crear noves entregues o agregar *DLCs* per continuar afegint contingut a obres ja creades, ja que, per la naturalesa del propi gènere, és molt més senzill crear nous elements que no interrompen el que ja s'havia creat anteriorment.

Per això, creiem que, tot hi que el joc és completament jugable i gaudible, podem afegir-li molt més contingut a dins, ja sigui narrativa, afegint *NPC* que ajudin al jugador dins dels nivells; una cinemàtica, per entendre la història; jugabilitat, agregant nous enemics i efectes per fer l'experiència del jugador més enriquidora i interessant; o estètica, afegint més animacions als elements que es veuen per pantalla.

També podem plantejar-nos la possibilitat de augmentar el rang de dispositius jugables, ja que de moment només es pot jugar amb PC, i *Godot* té moltes eines que permeten que els projectes es puguin jugar per diferents plataformes.

10. Bibliografia

- [John Wordsworth2020] https://www.johnwordsworth.com/projects/photoshop-sprite-sheet-generator-script/

- [LuisAvilés2021] https://alfabetajuega.com/pc/revelado-el-porcentaje-de-jugadores-que-utiliza-pc-y-es-mas-alto-de-lo-que-imaginas

- [ManuDelgado2022] https://vandal.elespanol.com/reportaje/los-mejores-roguelike

- [AlanBecker2017] https://www.youtube.com/watch?v=uDqjIdI4bF4&list=PL-bOh8btec4CXd2ya1NmSKpi92U_l6ZJd&index=13&ab_channel=AlanBeckerTutorials

- [LuisZuno2019] https://www.youtube.com/watch?v=Wqd6epIWo6E&list=PL02ki_m1apvXdiOqDhgPxmvwVpJaw8eeQ&index=4&ab_channel=LuisZuno

- [GameSpot2021] https://www.youtube.com/watch?v=N9Vs79VcYx0&ab_channel=GameSpot

- [godotshaders2021] https://godotshaders.com/shader/pixelate/

- [KarQ2021] https://www.youtube.com/watch?v=1hLbceyuF_o

- [EloHellWorkshop2021] https://docs.google.com/document/d/1hHR-3VZ4_VIGc5skwslMBf3FLuhN8dZVygtwxwZVicY/edit

- [JuanLinietskyArielManzur20072014] https://docs.godotengine.org/en/stable/about/complying_with_licenses.html

11. Manual d'usuari i d'instal·lació

11.1. Execució del joc

Un cop exportada la carpeta amb tots els documents relacionats amb el TFG, s'ha de prémer el programa anomenat *GreaterSword.exe*, es reconeix fàcilment veient l'icona blava de *Godot* (primer document de la **Figura 11.1**). L'executable actual de *GreaterSword* és vàlid a la plataforma *Windows*, però *Godot* no posa cap impediment en exportar per altres plataformes.

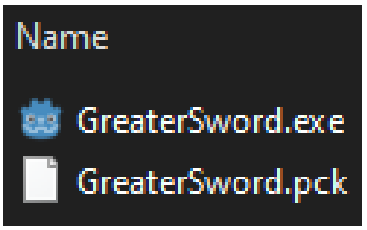

Figura 11.1. Fitxers necessaris per a l'execució del joc.

11.2. Esquemes de control

Com ja s'ha comentat anteriorment, el joc és jugable tan amb un controlador, com en teclat i ratolí. Els controls es mostraràn per teclat i per controlador respectivament utilitzant la **Figura 11.1** com a indicador del controlador:

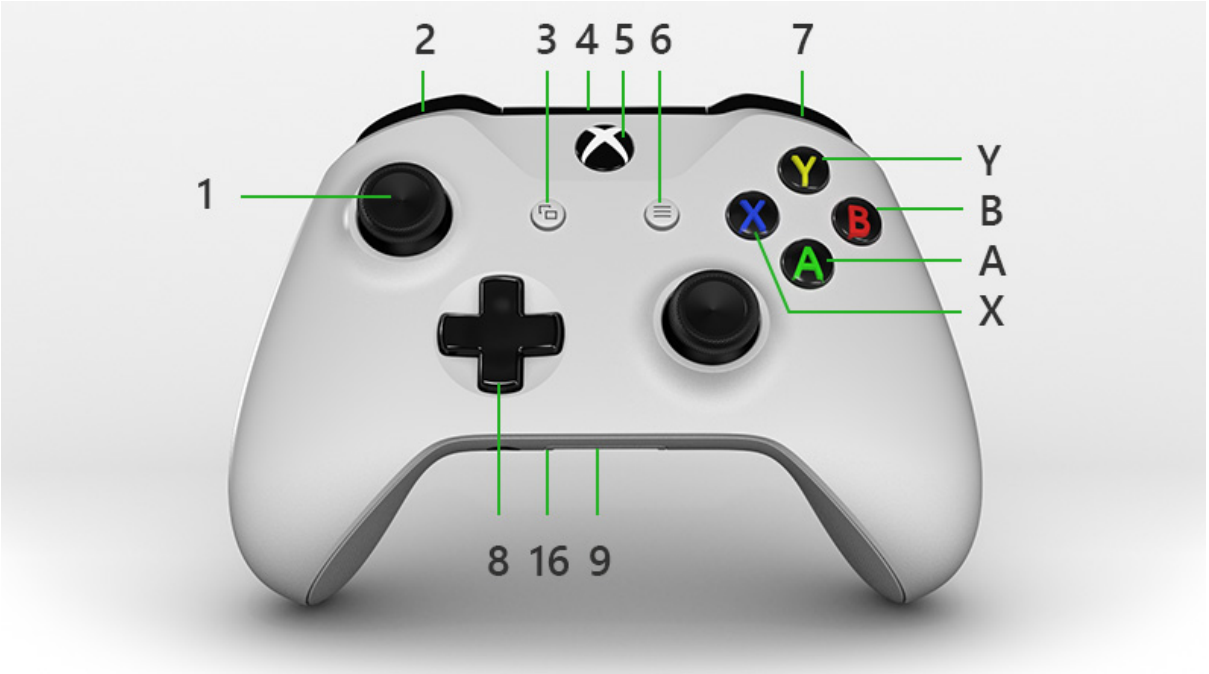

Figura 11.2. Controlador Xbox amb indicadors als *Inputs*.

- **Moviment del Personatge**: W,A,S,D / 1
- **Atac del Personatge:** Espai / X
- ***Dash* del Personatge:** Shift / B
- **Confirmar acció als menús**: Enter / A (o click amb el ratolí)
- **Menú de pausa:** Esc / 6
- **Inventari:** Tab / 3