Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau de Disseny i Desenvolupament de Videojocs

Títol: Disseny i desenvolupament d'un videojoc d'aventures i gestió de recursos

Document: Memòria

Alumnes: Sergio Rovira Gómez
Daniel Saavedra Martínez

Tutor: Gustavo Patow
Departament: Informàtica, Matemàtica Aplicada i Estadística
Àrea: Llenguatges i Sistemes Informàtics

Convocatòria (mes / any) Juny / 2021

Contingut

1. Introducció i objectius

1.1. Introducció

Durant els últims anys i a causa de l'augment de la disposició i l'accessibilitat a equips tecnològics, la popularitat del contingut relacionat amb l'entreteniment digital ha crescut exponencialment. Un dels mercats que més estímul ha rebut en els últims 10 anys és el dels videojocs, arribant a facturar la sorprenent quantitat de 174'9 mil milions de dòlars el passat 2020, un increment del 20% respecte al 2019. Quan parlem de la indústria dels videojocs som capaços de segmentar-la principalment en el hardware i el software. El hardware comprèn tot aquell dispositiu electrònic i tangible per l'usuari amb el quan podrà interactuar directament amb el videojoc i totes les possibilitats que els desenvolupadors, gràcies a la utilització de software específic, serán capaços de crear a fi d'obtenir un producte atractiu pel públic objectiu pel qual ha estat dissenyat.

Pel que fa al software, més específicament els videojocs, podem observar que els últims anys han aparegut una gran quantitat de títols impulsats per grans multinacionals però també per petits desenvolupadors que intenten fer-se un lloc en el mercat. Si explorem les possibilitats que els desenvolupadors han portat als diferents mercats i plataformes, serem capaços de categoritzar-los segons gèneres, estils, públics, durada, tecnologies implementades i sobretot la repercussió aconseguida durant la venta i després. El món digital en el que vivim actualment es mou a través de tendències seguides per la gran influència d'algunes figures públiques i que els propis jugadors decideixen impulsar mitjançant xarxes socials o el famós boca a boca. Per tant, podem trobar que al llarg del temps alguns gèneres han vist un increment de popularitat, arribant a trobar diverses opcions al mercat amb mecàniques molt similars.

Personalment, trobem que aquestes tendències neixen de projectes innovadors que aposten per crear noves experiències i que siguin capaços de recollir tot allò que un jugador promig espera. Per això apostem per la creació d'un producte que neix de les nostres pròpies necessitats com a jugadors i que, segons el nostre punt de vista, reuneix tot allò que actualment el mercat necessita.

1.2. Motivacions

A l'hora de portar endavant un projecte d'aquest tipus cal tenir molt clares les motivacions col·lectives i les individuals que ens guiaràn a l'hora del desenvolupament, proposant reptes i fent que el resultat final sigui encara més satisfactori. Algunes de les motivacions són les següents:

1. Exploració i implementació de metodologies de treball per equips.
2. Aprendre i millorar coneixements previs sobre les tecnologies que envolten el desenvolupament de videojocs.
3. Experimentar el procés de creació d'un producte propi sense la utilització d'elements gràfics o llibreries externes.
4. Desenvolupar mecàniques atractives pel jugador, iterant múltiples vegades fins aconseguir el resultat desitjat.
5. Ser capaços de repartir la feina segons els interessos propis per explorar un àrea específica (programació vs. dissenys visual).
6. Indagar més profundament sobre el programari específic usat pel desenvolupament.
7. Aconseguir un producte final que reflecteixi els coneixements adquirits i la capacitat de treball en equip, i que doni a lloc futures possibilitats i projectes més rics.

1.3. Propòsit i objectiu del projecte

En quant als propòsits del desenvolupament els podem separar en dos apartats que pretenden cobrir totes les necessitats del joc i dels propis com a desenvolupadors.

En primer lloc hem volgut desenvolupar un joc capaç d'atreure l'atenció del jugador, tant per diversitat mecànica com en l'apartat visual, fent especial èmfasi en la senzillesa i cohesió visual en tots els elements.
Per altra banda un dels propòsits com a desenvolupadors ha estat la capacitat d'implementar totes les mecàniques que ens vam proposar durant la pluja d'idees i que finalment donen lloc al producte final.
Alguns dels objectius que com a desenvolupadors volem cobrir són els següents:

1. Gestionar de manera eficient les tasques i objectius com a equip.
2. Dominar l'entorn de desenvolupament de Unity i conèixer més sobre aquesta eina.
3. Estimular l'àmbit creatiu generant tots els elements del joc, fent servir referències d'altres però sense incloure-les directament.

4. Ser capaç de plasmar totes les idees plantejades en el disseny i millorar-les al llarg del desenvolupament, fent-les més riques i interessants.
5. Aprendre a estructurar un projecte de grans dimensions amb tots els elements visuals i lògics correctament localitzats.
6. Procurar dissenyar tots els elements del joc de manera que aquest sigui fàcilment escalable.
7. Garantir un correcte funcionament de totes les mecàniques aplicades i la cohesió entre elles.
8. Aprendre a implementar un sistema de generació de mons amb una aleatorietat controlada que permet una fàcil expansió del món.

1.4. Distribució de tasques

A l'hora de distribuir les tasques a realitzar vam decidir enfocar-nos en els camps que més còmodes ens podíem sentir, i també en aquells en els que volíem indagar una mica més. En el nostre cas, i gràcies a experiències prèvies treballant de manera conjunta, vam tenir bastant clara com sería l'organització de l'equip.

Sergio Rovira Gómez

Estètica	65%
Narrativa	5%
Mecàniques	20%
Tecnologia	10%

En Sergio va decidir enfocar tots els esforços en treballar tots els apartats visuals del videojoc, tant els relacionats amb la jugabilitat com els menús i l'estil general. Una de les motivacions que va trobar en centrar-se en el contingut visual va ser el d'estimular el costat creatiu, millorant la tècnica i procurant mantenir un estil artístic al llarg del desenvolupament.

Pel que fa als percentatges reflectits i com s'indica anteriorment, la majoría del temps invertit en el desenvolupament ha estat el de la creació del tots els recursos visuals. Per altra banda, també s'hi ha dedicat temps a la planificació de tasques i a la presa de

decisions sobre mecàniques i el flux de joc. Un altre aspecte important ha estat el de la recerca de tècniques de modelat i l'aplicació de diverses metodologies a l'hora de realitzar tots els components del joc. Un cop trobat el mètode adequat, aquest s'aplicava a la resta d'elements.

Daniel Saavedra Martinez

Estètica	5%
Narrativa	5%
Mecàniques	20%
Tecnologia	70%

Durant tot el procés de desenvolupament en Dani va decidir enfocar-se completament en l'implementació a nivell tècnic de totes les mecàniques i l'estructura del món, per tant el seu balanç de temps està completament posicionat al costat tècnic. Una de les característiques principals del temps que s'hi ha invertit ha estat en el de la investigació de tot el relatiu al motor de videojoc que s'ha triat, ja que un cop parlem d'un projecte d'alta envergadura, la complexitat del codi i les estructures s'incrementa.

Pel que fa als percentatges, i com a equip de treball, ha intervingut sobre alguns elements visuals d'entre les propostes que es llençaven i també ha contribuït a l'implementació d'efectes visuals. Per altra banda i com s'esmenta anteriorment, la seva feina ha estat la d'investigar les possibilitats del motor de joc i la creació d'estructures lògiques que tenen com a principis la modularitat i la capacitat d'expansió del món d'una manera fàcil i entenedora.

2. Estudi de viabilitat

L'estudi de viabilitat és un procés que permet analitzar les necessitats econòmiques del projecte, la possibilitat d'entrada en el mercat i l'interès general per part del públic. Amb aquest estudi som capaços de realitzar una estimació del cost final del projecte i de realitzar pronòstics sobre l'acollida al mercat.

2.1. Recursos necessaris i viabilitat

Aquí s'esmenten tots aquells recursos materials i humans per la realització del projecte. Es calcularà una estimació del cost del projecte i si aquest resulta viable per a la realització.

2.1.1. Recursos tècnics

En quant a recursos tècnics, parlarem de tot el material que l'equip necessita per realitzar el desenvolupament. Aquest material tant pot ser software com hardware, i és el següent:

- **Ordinador de sobretaula equipat amb un Ryzen 5 3600 x, 16GB de ram i una tarjeta gràfica NVIDIA GTX 1060 6GB.** Eina principal feta servir per en Sergio a l'hora de desenvolupar el contingut visual del joc.
- **Ordinador de sobretaula equipat amb un Intel i7 7700, 16GB de ram i una tarjeta gràfica GTX 1060 3GB.** Eina principal feta servir per en Dani a l'hora de desenvolupar el codi del joc.
- **Taula gràfica Wacom Intuos M.** Eina feta servir per la creació de contingut visual.
- **Unity 3D.** Software d'ús lliure utilitzat per la creació de videojocs.
- **Photoshop.** Programa especialitzat en la creació i edició d'imatges utilitzat per generar contingut visual dins del joc.
- **Blender.** Software gratuït per la creació d'elements 3d pels diferents objectes que trobarem al joc.
- **Visual Studio 2019.** Editor de codi que, gràcies a la integració amb Unity, és capaç de reconèixer els objectes interns del motor i realitzar tasques de debug en temps real.
- **Discord.** Programa gratuït per realitzar trucades online i fet servir per coordinar el desenvolupament i realitzar reunions per discutir diversos aspectes del joc.
- **jsoneditoronline.org**. Página web gratuita que serveix per millorar la comprensió i la llegibilitat d'un fitxer JSON.

- **Autodesk FBX Converter.** Software gratuït fer servir per convertir models 3d antics a versions més recents i així poder obrir-los amb Blender.

Tots els recursos esmentats anteriorment no han tingut cap cost pel projecte ja que son programaris gratuïts o bé ja es disposaven amb anterioritat al projecte.

2.1.2. Recursos humans

Els recursos humans són aquells que fan referència a tot el personal necessari per a la realització d'un projecte de qualsevol àmbit. En el nostre cas existeixen uns rols especialitzats en la creació de continguts relacionats amb el món del videojcs.
Els perfils especialitzats en el nostre cas són els següents:

- **Dissenyador de joc:** Persona encarregada de definir com serà al joc a nivell visual, mecànic i narratiu. Es una de les figures més importants del projecte ja que moltes vegades s'encarrega de coordinar els altres rols.
- **Programador:** Encarregat de dur a terme totes les tasques d'implementació de regles i mecàniques del món. Aquest rol ha de tenir un enfoc purament tècnic per assolir tots els reptes de disseny que es proposen.
- **Artista:** Rol encarregat de portar a terme tota la representació visual del joc. Aquest rol és el més imaginatiu i tindrà la responsabilitat de crear tot l'entorn de joc mantenint sempre una bona cohesió entre tots els elements.
- **Dissenyador de so:** Aquell encarregat de crear o modificar sons o melodies per incloure-les al joc.

A banda d'aquests rols esmentats, existeixen una gran quantitat de professionals en un projecte o una empresa de gran envergadura. En el cas del projecte que es tracta, i com l'equip és prou petit, considerem que aquests són els més importants i aquells que han tingut un paper representatiu en el desenvolupament.

2.1.3. Viabilitat econòmica

La viabilitat econòmica tracta aquells temes que, pel desenvolupament del projecte, suposen un cost inicial i per tant s'han de tenir en compte abans d'iniciar-lo. En aquest cas i com ja disposem d'equip capaç de realitzar les tasques necessàries, el cost del projecte ha estat mínim. Per tant, es realitzarà una estimació del cost de la maquinària i del software fet servir.

El cost aproximat dels elements sería el següent:

Recurs	Cost (€)
Maquinaria (ordinadors)	2300
Complements (taules gràfiques, monitors, perifèrics...)	600
Software (photoshop, control de versións)	60
TOTAL	2960

Aquests costos seríen els que, en un cas hipotètic, hauríem d'adquirir per poder dur a terme les tasques de desenvolupament.

Per altra banda si decidissim llogar un local especialitzar per treballar en el projecte hauríem d'estimar el cost del lloguer i el mobiliari.

El cost aproximat sería el següent:

Element	Cost (€)
Lloguer d'oficina (35m2)	350 (mensual)
Compra de mobiliari (taules, cadires, armaris...)	600
Serveis (internet, llum, aigua...)	170
TOTAL	1120

Com es pot observar, les necessitats de treballar en un espai específic pot incrementar notablement el cost del projecte, sobre tot l'inicial, ja que en el supòsit que plantegem no disposem de cap tipus de mobiliari i només es pressuposta el bàsic per poder treballar. Ara bé, en aquest cas existeix la possibilitat de treballar des de casa i mitjançant programes de control de versió podem compartir els progressos de manera instantània.

Ara que ja tenim tots els elements físics, cal recalcar que el preu hora d'un desenvolupador també s'ha de comptabilitzar, tot i que els desenvolupadors no reben una retribució en el moment de la producció. Per cadascun dels rols esmentats anteriorment existeix un salari promig el qual sería el següent:

Game Designer: 14€/h
Game Programmer: 12,3€/h
Game Artist: 10,8 €/h
Game Audio: 11,3 €/h

Un cop tenim en ment el salari promig de cadascun d'aquests llocs de treball, haurem de fer un càlcul senzill per saber el cost anual en quant al cost econòmic en termes dels recursos humans. Si ho exemplifiquem en 6 mesos de feina i només tenim en compte que tant en Dani com en Sergio ha treballat com 1 dels rols, podem estimar que el salari aproximat de cadascun sería de 1.728€ per en Sergio com a Game Artist i 1.968€ per en Dani com a programador.

Amb totes les dades que conformen aquest apartat podem obtenir el cost mensual aproximat des de l'inici del projecte fins el teòric llançament. Evidentment, tots aquests costos dependràn de la situació dels desenvolupadors i de possibles factors externs que puguin intervenir.

2.2. Estudi de mercat

Entenem per estudi de mercat com una manera d'intentar trobar possibles tendències, productes similars o la manca de productes semblants per poder introduir de manera efectiva el producte que es desenvoluparà. A l'hora de realitzar l'estudi cal tenir en compte que el mercat pot variar molt ràpidament i que es mou per tendències que poden aparèixer en qüestió d'hores amb l'aparició de títols amb gèneres o mecàniques innovadores. Tenint en compte aquest fet, hem de ser capaços de fugir de les tendències per evitar quedar fora de mercat un cop el joc estigui desenvolupat, ja que és possible que amb un equip petit no s'arribi a aprofitar l'impuls del mercat un cop el joc estigui finalitzat. Per evitar això ens haurem de centrar en aquells gèneres que perduren en el temps, implementant noves maneres de jugar o variacions que permetin al jugador viure noves experiències.

2.2.1. Estat de l'art

A l'hora de realitzar l'estudi de mercat cal considerar la resta de títols per tal de posicionar-nos en una posició còmode per poder oferir un producte novedós. Per això cal estudiar els millors exponents del gènere o bé aquells que per l'apartat visual, la narrativa o les mecàniques s'assemblin a la nostra proposta. Tota aquesta recerca inicial servirà per veure si la proposta ofereix alguna cosa nova que pugui resultar atractiva i diferenciadora pel jugador.

A continuació s'esmenten algunes referències que han servit per inspirar algunes de les parts del projecte tant pel que fa als gràfics o la intenció del desenvolupador envers el jugador.

Stardew Valley

Stardew Valley es un joc comprès entre gèneres com el rol, la simulació i l'acció, que tot i portar una línia estètica totalment diferent a la que proposem, si que comparteix moltes de les mecàniques que hem volgut implementar al nostre títol. Les mecàniques més interessants que podem trobar són la de *farmeig*[1], l'exploració i el combat. La primera de les mecàniques permet al jugador plantar, recol·lectar i produir per poder generar l'economía que més tard necessita per poder progressar. La segona de les mecàniques la podem trobar en un dels escenaris, on també intervé el combat, i que és totalment desconegut pel jugador estàndard. Ara que ja coneixem les mecàniques principals, podem veure que té una gran similitud al nostre producte i per això el situem com a punt de referència principal.

Figura 2.2.1: Captura del joc Stardew Valley

[1] **Farmeig: Terme que comprèn totes les activitats relacionades amb la recol·lecció tant d'objectes del món com els propis generats pel jugador.*

Human fall flat

Human Fall Flat és un joc de plataformes que consisteix en superar un seguit d'obstacles o proves per arribar a la meta. Aquest joc té la capacitat de ser jugat cooperativament cosa que aporta un valor afegit molt gran de cares al públic al que va dirigit. Ara que ja sabem de que va el joc, cal dir que les mecàniques no han estat precisament la font d'inspiració, de fet tot el contrari ja que ha estat l'estil de l'art el que, gràcies a la seva senzillesa i, tot i la seva simplicitat, genera un entorn fàcil d'entendre pel jugador. La simplicitat de la geometria i la paleta de colors amb la manca de textures complexes provoca que el jugador es concentri totalment en l'experiència de joc.

Figura 2.2.2: Captura del joc Human Fall Flat

Don't starve

Don't starve és un títol un tant peculiar que tracta principalment el gènere d'aventura i supervivència recolzat per l'acció i l'exploració, uns factors que tots plegat ofereixen una experiència molt complerta pel jugador. En aquest títol també trobem la possibilitat de crear la teva pròpia base on el jugador serà capaç de plantar cultius i produir diferents recursos.

És, doncs, aquesta l'experiència que hem volgut portar cap al nostre projecte ja que combinat amb l'aleatorietat dels escenaris que podem trobar en tots dos títols, podem obtenir diferents resultats a cada partida. També es tracten temes com la recol·lecció i la lluita, un element que en el nostre cas volem que tingui més protagonisme apostant per oferir mecàniques que el facin més dinàmic pel jugador.

Figura 2.2.1: Captura del joc Don't starve

Astroneer

Astroneer es un joc de supervivència i món obert on el jugador haurà de completar un seguit de reptes mitjançant la recol·lecció i la creació d'elements. Quan comparem Astroneer amb el nostre projecte, podem veure algunes similituds pel que fa al camp artístic i algunes de les mecàniques, com la col·locació d'elements últils que ajuden en la producció o la possibilitat d'explorar un món general aleatòriament. Per tant, tot i que a Astroneer no trobem la component de combat, trobem que és una de les referències que han guiat el projecte.

Figura 2.2.1: Captura del joc Astroneer

2.2.2. Model de negoci

Actualment en el sector del videojoc existeixen diverses maneres de generar ingressos segons el contingut. En general els podem categoritzar de la següent manera:

- **Pay-to-play**: Aquest és el model tradicional de venta on el client compra el producte per una quantitat fixe, obtinguent accés a la totalitat del joc.
- **Free-to-play:** Model de negoci que consisteix en la posada a disposició al públic d'un producte totalment gratuït. Aquest model de negoci generarà benefici gràcies a la venda de béns virtuals dins del joc.
- **Freemium:** Categoría que consisteix en la posada a disposició al públic del producte parcial de manera gratuïta, fent que per desbloquejar la totalitat sigui necessari realitzar alguna compra.
- **Subscripció:** El terme subscripció fa referència a la necessitat del comprador de pagar de manera continuada per accedir a béns o serveis, en aquest cas s'aplica videojoc fent que el suscriptor necessiti mantenir el pagament per poder-hi accedir.

Alguns d'aquests models de negoci han aparegut a mesura que les tecnologies de la comunicació han avançat fins que finalment ens han permès jugar a videojocs en línia de manera regular. En el nostre cas, el producte que desenvolupem es tracta d'un videojoc tancat el qual no incorporarà la capacitat de jugar en línia o bé l'obtenció de béns dins del joc, per tant el model escollit ha estat el de tradicional *Pay-to-play*.

En quant a la venta del producte i la comercialització, el joc apunta principalment a la plataforma d'ordinador de sobretaula ja que, basant-nos en les propostes del mateix camp que existeixen en el mercat i el tipus de públic al qual va adreçat, creiem que és l'idoni. No es descarta una futura adaptació per a consoles com Nintendo Switch, Playstation o Xbox un cop el joc ja es trobi en el mercat. Per tal de mantenir el joc amb vida i gràcies a la seva estructura i tipología es podría plantejar, en un futur, la capacitat d'expandir el joc amb contingut extra (DLC) on el jugador pogués continuar les seves aventures.

2.2.3 Matriu de competència

Mitjançant una matriu de competència serem capaços de comprendre si realment el nostre projecte es tracta d'un producte prou interessant.

	Gràfics	**Gènere**	**Duració**	**Preu**
Stardew Valley	Pixel Art	Rol	60h~	13,99€
Human fall flat	3D	Plataforma	4h~	7,99€
Don't Starve	2D	Supervivència,acció i aventura	25h~	8,20€
Astroneer	3D	Supervivència	15h~	27,99€
Lost drift: the Kairu's tale	3D	Supervivència, acció i aventura	2h~	4,99€

Si analitzem la taula podem observar que, tot i no afegit gèneres nous, trobem un equilibri a l'hora de proposar el llançament. Cal destacar que el gènere no és l'únic diferenciador a l'hora de tenir en compte el mercat, ja que el que realment diferencia dos títols entre ells és l'aplicació de mecàniques, l'ambientació, el propòsit final... i tot un conjunt d'elements que fa que cada joc sigui diferent.

En conclusió, podem dit que *Lost Drift,* tot i ser de curta durada, és un títol interessant gràcies als gèneres que tracta i el preu assequible que farà que tingui un lloc en el mercat.

2.3. Públic objectiu i tipología de jugador

A l'hora de dissenya un videojoc és molt important tenir en compte cap a qui va dirigit i quina és l'experiència d'usuari que voldrem aconseguir. Aquests dos elements són els que, juntament amb l'estudi de mercat, encaminaran el nostre joc en la direcció correcta sempre i quan es compleixin les espectatives i el joc contingui tot l'esperat.

Lost Drift és un joc que pretén agradar tant a petits com a grans i per tant podem dir que el públic objectiu abarca un rang d'edats prou ampli. Per tant, considerem que l'edat mínima per entendre els conceptes del joc i les mecàniques per poder completar-lo sería de set

anys. En quant a l'edat máxima, considerem que no hi ha una edat màxima, ja que gràcies al gènere i l'estil de jugabilitat és un joc que es pot gaudir en qualsevol moment.

Amb l'objectiu d'analitzar de manera més objectiva el perfil del jugador al que el joc anirà dirigit ens podem recolzar en l'estudi de la taxonomía de Bartle, una categorització del perfil dels jugadors segons els seus interessos, les motivacions personals i el caràcter. Segons Bartle els podem categoritzar de la següent manera:

- **Achiever:** Aquell jugador que té com a objectiu superar tots els reptes que el joc proposa, complementant-lo en la seva totalitat i aconseguint totes les recompenses.
- **Explorer:** Jugador que gaudeix explorant el món, descobrint-lo i aprenent d'ell. Aquest sol ser un perfil que li agrada aprofundir sobre la història del joc i els esdeveniments passats.
- **Socializer:** Tal i com indica el nom, aquest és un perfil més social que té com objectiu explorar totes les capacitats socials que el joc ofereix per sobre de la obtenció de recompenses.
- **Killer:** Jugador que se centra en l'aspecte més competitiu del joc volent ser el millor obtinguen la major quantitat de punts. Aquest perfil el podem trobar més en jocs en línia on es crea un major sentiment de competitivitat.

Un cop descrits els perfils podem veure que el perfil que més s'adequa al contingut que oferim és el del *Explorer,* ja que tant el disseny del món com les mecàniques faran què aquest tipus de perfils gaudeixin d'una experiència molt satisfactòria. Així i tot, cal esmentar que, tot i que dins el joc existeix el component de combat, i que és prou important per poder completar el joc, aquest no té l'objectiu de ser competitiu i per tant no compliria les característiques d'un *Killer*.

3. Planificació

El procés de planificació del projecte ens ajudarà a organitzar les tasques, marcar un temps límit per cadascuna i controlar el temps de cares a l'entrega final. Com a equip és molt important mantenir un bon flux de comunicació per coordinar tota la feina a l'hora de generar el contingut.

Podem dir que l'inici del projecte es va dur a terme a principis d'Octubre del 2020 on es va començar a realitzar una pluja d'idees per encaminar el projecte. A partir d'aquí es van repartir les tasques i a mesura que ha anat avançant el projecte hem après a coordinar-nos de manera més efectiva, fent que la producció incrementés.

3.1. Diagrama de Gantt

El diagrama de Gantt ens permetrà conèixer quant temps s'ha invertit per realitzar les diferents fases que comprenen un projecte d'aquesta envergadura on intervenen diverses modalitats (lògica, artística, proves...).

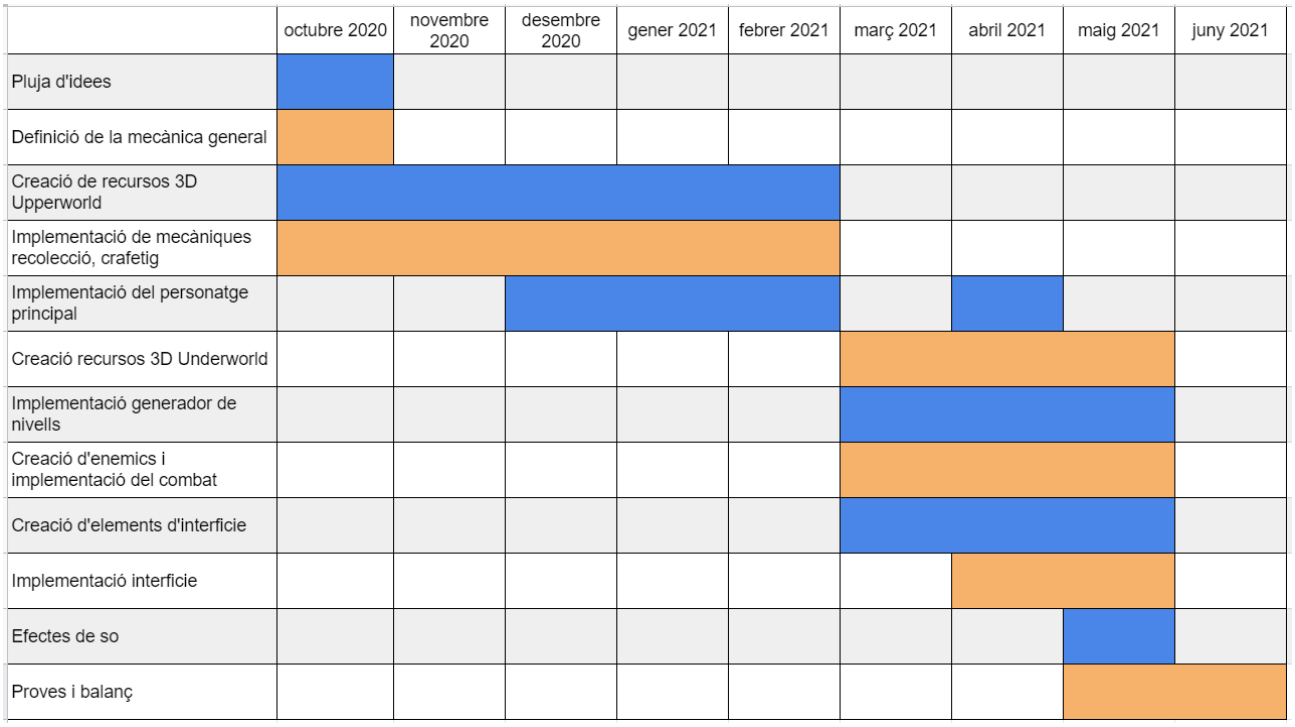


Figura 3.1.1:Diagrama de Gantt del projecte

Cal esmentar que, tot i la representació del gràfic, és realment dificil realitzar una línia del temps concreta per la realització de la tasca ja que durant tot el procés el projecte ha patit canvis a nivell logístic i visual que han repercutit directament en el temps de planificació, ja que qualsevol canvi, per menor que sigui, pot comportar una modificació d'altres aspectes del videojoc.

3.2. Eines de treball i programari

A l'hora de desenvolupar un projecte d'aquestes característiques és necessari la utilització de diversos software per generar el contingut necessari, tant artístic com lògic. A continuació s'esmenten categoritzats per tipología els diferents programes i la implicació en el projecte.

3.2.1. Motor de jocs

El pilar fundamental del desenvolupament d'un videojoc, sobre tot si es tracta d'un projecte independent, és la utilització d'un motor de videojocs capaç de proporcionar-nos eines per l'elaboració d'un joc. En aquest cas hem decidit fer servir Unity 3D per realitzar tot el procés de maquetació del joc ja que és el motor amb el que ens sentim més còmodes per treballar.

Unity 3D és un dels motors de videojoc més populars avui en día, que està disponible de manera gratuïta pel lliure desenvolupament i distribució, sempre i quan no superi una xifra de vendes determinada. Gràcies a aquesta eina un gran número d'estudis independents han estat capaços de portar jocs increibles com per exemple *Monument Valley* o *Cuphead*. També cal destacar que aquest motor no només el fan servir petits estudis, ja que jocs com *Pokemon Go* o *Hearthstone* també estan fets amb Unity 3D per empreses multinacionals.
Una de les característiques de Unity 3D es la possibilitat de generar contingut de diversos estils i plataformes, fent que el títol desenvolupat tingui la possibilitat d'arribar al major número de persones apliant enormement el mercat. Unity 3D proporciona eines per simular físiques, incorporar modelats tridimensionals, animacions i controlar interaccions de manera senzilla i eficient.

Per altra banda també és important conèixer l'alternatives com Unreal Engine, un altre motor de videojoc capaç de realitzar les mateixes tasques de Unity 3D, però que en el nostre cas no hi estem tant familiaritzats.

3.2.2. Editors de codi

Visual studio

Tot el codi del projecte ha estat realitzat amb Visual Studio. Vam decidir utilitzar Visual Studio ja que és l'editor per defecte del Unity. A més, té moltes funcionalitats, correccions i comentaris que estan enfocats a desenvolupar amb Unity. També permet debugar en temps real cosa que afavoreix a tenir un producte molt més estable i sense errors.

3.2.3. Programari artístic

Blender

Per la realització de tots els models 3D ens hem decantat per la utilització de Blender, un programa molt popular, potent i sobretot d'ús lliure que disposa de totes les eines necessàries per aquest projecte. Aquest programa rivalitza directament amb 3D Max ja que, sent d'ús lliure i gràcies a les seves possibilitats, ha generat un gran número de seguidors el quals han creat una gran quantitat de guies i continguts audiovisuals que ens han ajudat a l'hora de fer-lo servir.

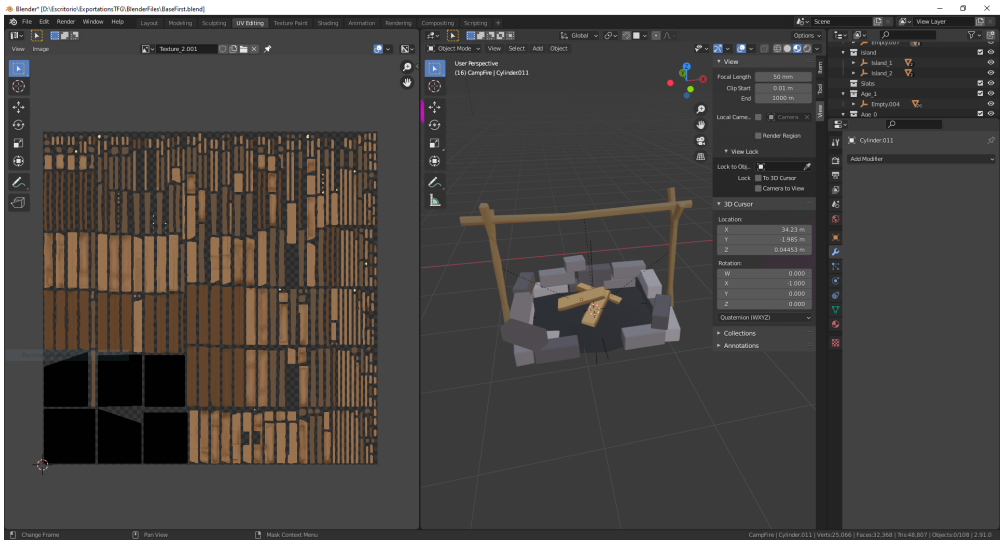


Figura 3.2.3.1: Entorn de desenvolupament de Blender

Photoshop

Una eina que ha tingut molt protagonisme en el projecte ha estat Adobe Photoshop el qual hem fet servir per generar textures i recursos gràfics per la creació de la interfície. Photoshop és una eina àmpliament coneguda en el món del disseny gràfic i la fotografía que té un recorregut històric molt ampli i una gran empresa darrere que proporciona suport i actualitzacions contínues. Per la nostra part, s'ha fet servir sobretot a l'hora de realitzar els elements que formen la interfície del joc.

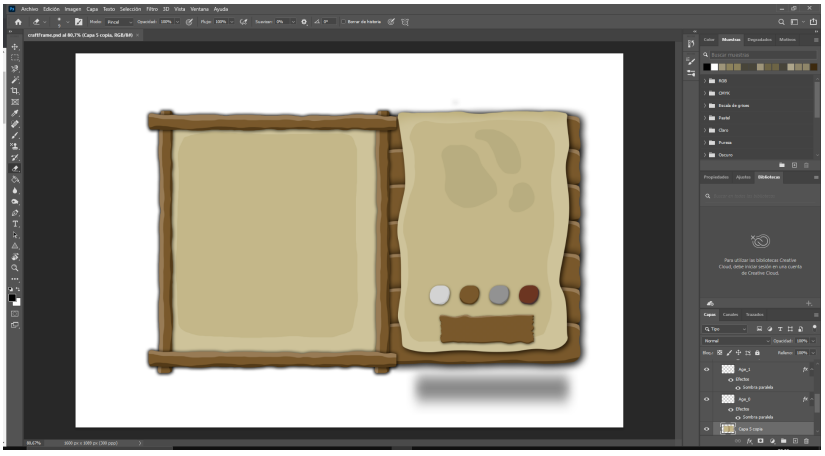


Figura 3.2.3.2: Entorn de desenvolupament de Photoshop

Autodesk FBX Converter

Un cop entrat en el projecte, vam trobar la necessitat d'inspeccionat un model extret d'internet per veure la seva composició, i com aquest estava en format antic, no el vam poder obrir. Amb aquest problema en mà, i després d'una breu cerca, vam trobar Autodesk FBX Converter que ens va permetre fer una conversió de format que finalment en va permetre obrir en model amb Blender.

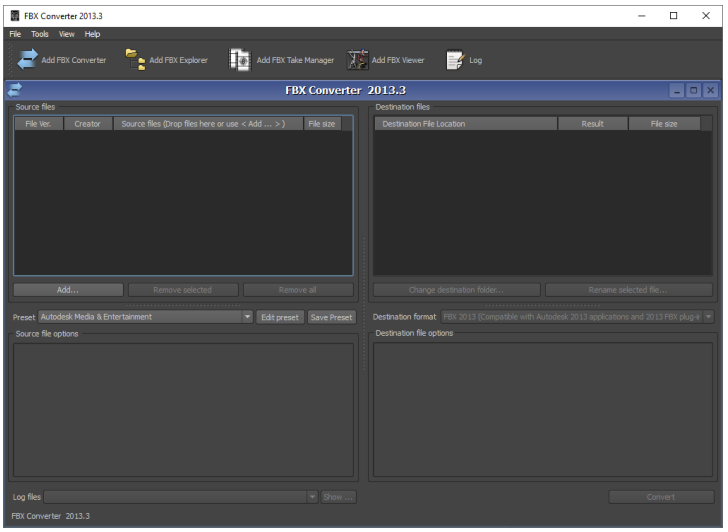


Figura 3.2.3.1: Entorn de desenvolupament de FBX Converter

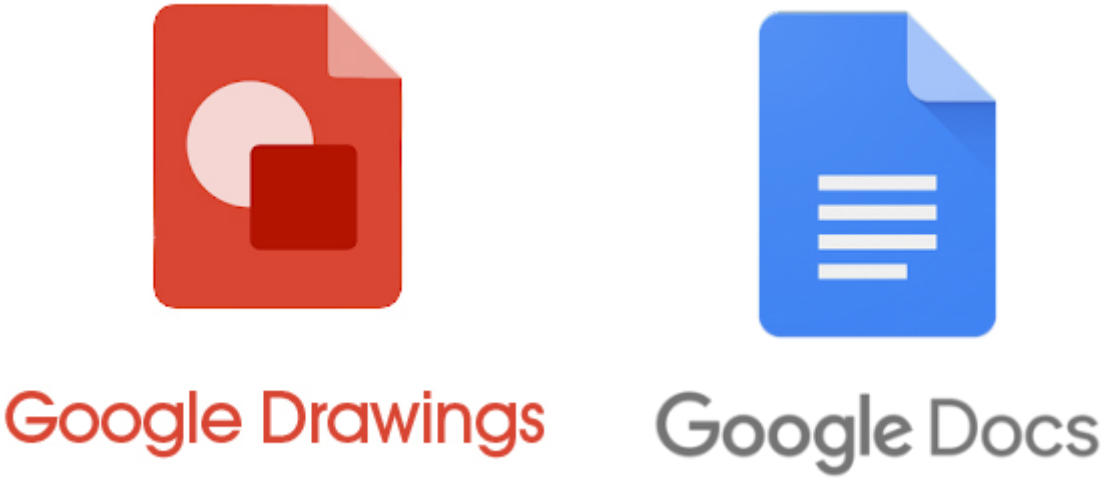
3.2.4. Altres

Discord

Discord ha estat per excelència el programa utilitzar per mantenir la comunicació entre l'equip ja que mitjançant les trucades voip i el sistema compartiment de pantalla hem estat capaços de resoldre dubtes, planificar i ajudar-nos mútuament. Aquest és un programa gratuït que permet fer trucades voip múltiples. Treballa de manera similar a Skype i la seva popularitat ha augmentat enormement els últims anys gràcies a la seva interfície, les funcions i la qualitat de les trucades.

Google Docs i Google Draw

Google ofereix una gran quantitat d'eines i serveis de manera gratuïta que en el nostre cas hem aprofitat per realitzar algunes de les tasques com la documentació i els múltiples gràfics que podem trobar al llarg del document. Per fer tot això hem fet servir concretament Google Docs que ens ha permès realitzar el document en viu d'una manera fàcil i eficient i Google Draw que conté les funcionalitats necessàries per realitzar petits esquemes UML i altres necessaris per entendre algunes de les estructures del projecte.



4. Marc de treball

El projecte que es tracta en aquest document té la intenció de ser presentat davant un jurat el qual evaluarà cadascún dels apartat que formes la creació d'un videojoc. Aquest projecte forma part d'una de les etapes més importants del creixement professional dels desenvolupadors, en Sergio Rovira i en Dani Saavedra, dos estudiants del grau de Disseny i Desenvolupament de Videojocs que s'han proposat creat un prototip de videojoc aprofitant tots els coneixements adquirits durant la carrera i apostant per plasmar una idea i exprimir tot el potencial que aquesta idea pot otorgar. El que començés sent una idea entre amics ha esdevingut en la formació d'un equip disposat a crear un producte que complís les seves pròpies expectatives, donant-los llibertat creativa per imaginar un món fictici.

Pel que fa al projecte, anomenat *Lost drift: the Kairu's tale*, ens situa en un món desconegut el qual està format per un conjunt d'illes flotants on es desenvoluparà una part important del joc. Per altra banda, existeix una terra alternativa habitada per criatures desconegudes i que serà un dels punts més importants pel que fa a la jugabilitat. En aquest món el jugador serà capaç d'interactuar amb diversos elements amb l'objectiu de superar l'objectiu final.
Durant tot el joc el jugador haurà de realitzar diverses accions, entre aquestes podem distingir les més importants explicades a continuació i que serviran per entendre futurs apartats:

- **Farmejar:** Capacitat del jugador de recollir recursos i crear-ne a partir de tècniques conegudes, com per exemple la recol·lecció, la tala, la minería o bé el cultiu i la caça. En aquest cas el terme es farà servir per recollir la matèria prima que més tard farà servir el jugador per construir o obtenir millores. Aquest concepte prové de l'anglès.
- **Craftejar:** Acció que farà el jugador per crear objectes per un ús concret. En aquest cas concret el jugador necessitarà construir eines i estructures per generar recursos com per exemple una espasa o un test per plantar-hi una llavor. Aquest concepte prové de l'anglès.
- **Època / etapa:** Una etapa defineix un perìode de temps concret. En el nostre cas fem referència a un període o un estat del món que limita el coneixement del jugador per poder crear objectes o realitzar certes accions. Aquestes etapes es representen dins del joc en l'estat de la casa del personatge (veure apartat 5).
- **Parring:** Aquesta és una expressió que fa referència a l'acció de parar alguna acció. En aquest cas trobarem aquesta acció en el mòdul de combat, ja que és una de les

mecàniques del joc que ens proporcionarà una experiència més rica a l'hora de lluitar contra els enemics i que ens permet obtenir avantatges com una millora temporal en l'atac i invulnerabilitat temporal (el mòdul de combat s'explicarà detalladament més endavant. Apartat 6). Aquest concepte prové de l'anglès.

- **Generació de món:** Pel que fa al món inferior que trobarem en el joc (informació en els apartats 5 i 6) aquest es genera de manera automàtica i amb un algoritme d'aleatorietat controlada dissenyat per obtenir petits recorreguts d'illes per aportar un element diferenciador entre partides.

Tots aquests són els elements que formen el joc i que, després de realitzar l'estudi de mercat, hem trobat que pot aportar un salt en quant a jugabilitat pels gèneres tractats. És evident que el mercat dels videojocs és molt gran i existeixen una gran quantitat de títols plens de contingut variat i és molt difícil tenir-los tots en compte, però estem convençuts que una idea original ben portada por arribar als ordinador i les consoles de molta gent.

5. Disseny de videojoc

En aquesta secció es defineixen tots els elements que formen el videojoc. Com podem observar a continuació els elements que formen un videojoc són molt diversos i necessiten estar connectats entre ells per finalment formar un producte que tingui sentit. Trobarem des d'aspectes tècnics relacionats amb l'estructura del món fins a referències artístiques i explicacions sobre la metodología seguida per la creació de contingut divers.

5.1. Mecàniques

Les mecàniques les podem definir com les regles que donen forma al món i descriuen el funcionament i les limitación que el jugador es troba durant l'experiència.

5.1.1. L'espai de joc

L'espai de joc defineix tota l'estructura del món on passa tota l'acció durant el joc. A l'hora de definir l'espai de joc vam tenir en compte com volíem que es desenvolupés el transcurs del videojoc per poder oferir un entorn innovador que atragués el públic.

Des del principi vam plantejar dos espais clarament diferenciats per l'estètica i l'efecte que volíem provocar sobre el jugador. En el primer dels entorns ens trobem amb un espai molt amigable i colorit sense cap element hostil. Per altra banda, el segon es tracta d'un entorn fosc i desconegut que té com intenció provocar certa tensió al jugador.

El mòn en el que ens situem està format per un conjunt d'illes flotants les quals hauran de ser explorades, ja que aquestes illes contindran diferents tipus de recursos que serviran al jugador per poder avançar. Per altra banda també existeix una zona on el jugador haurà de superar reptes combatint contra enemics per poder obtenir altres recursos.

Figura 5.1.1.1: Esbós d'espai de joc

5.1.1.1. Upperworld

El que anomenem *Upperworld* fa referència a aquell espai on el jugador serà capaç d'avançar passivament sense enfrontar-se a cap enemic en concret. Aquest espai està format per un conjunt d'illes que seran accessibles a mesura que el jugador obtingui recursos per poder avançar.

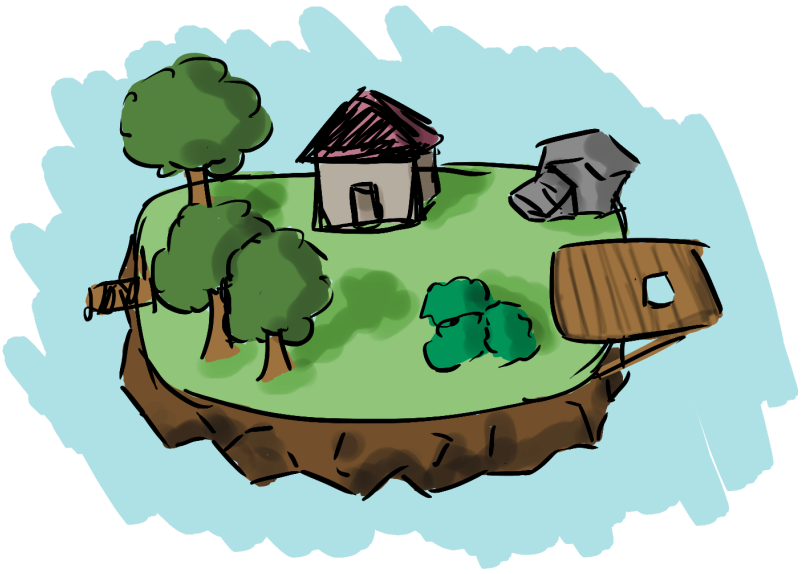


Figura 5.1.1.1.1: Esbós de Upperworld

Tal com es pot veure en la Figura 5.1.1.1.1, en aquesta escena apareixen elements pels quals el jugador podrà realitzar interaccions referents a la creació d'objectes, millores o reparacions. Un dels elements més importants és la casa, l'element cèntric del joc que marcarà el nivell de progrés del jugador, ja que aquesta anirà evolucionant i desbloquejarà l'accés a noves creacions.

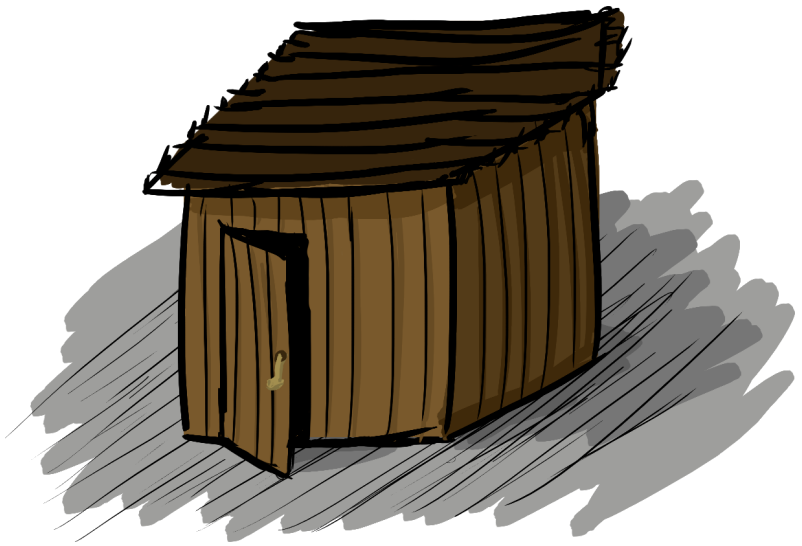


Figura 5.1.1.2: Esbós de casa de fusta

Un altre element característic que podem veure a la part dreta de la illa és la passarel·la que permetrà descendir al Underworld.

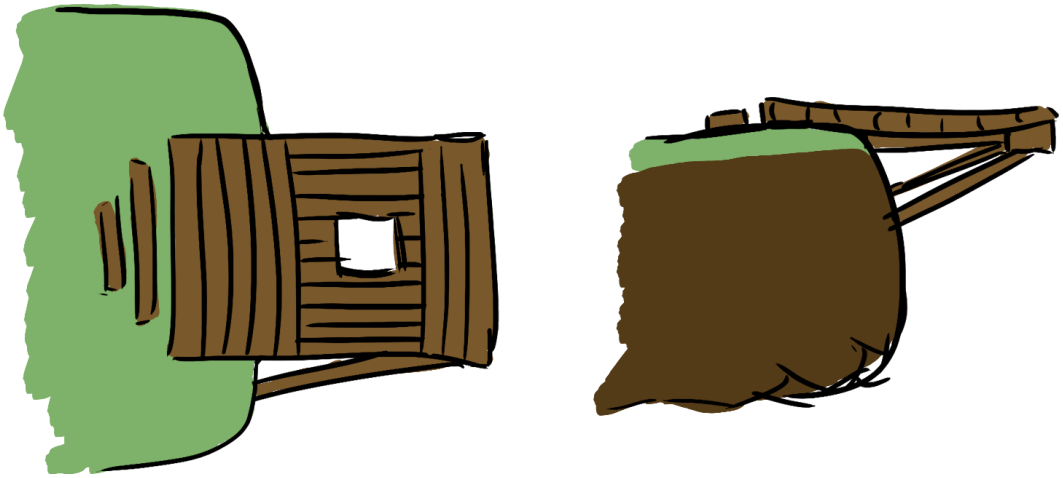


Figura 5.1.1.3: Esbós de passarel·la de fusta

Finalment podem observar un pont a la part esquerra que ens permet accedir a la segona illa del joc, la qual ens permet accedir a més recursos i augmentar l'espai de construcció.

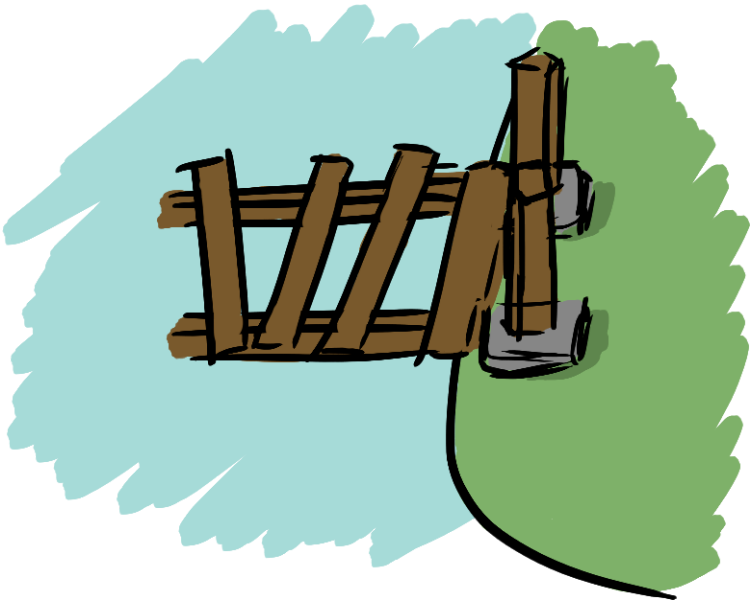


Figura 5.1.1.4: Esbós de pont de fusta

5.1.1.2. Underworld

El Underworld és la definició del que és desconegut, on el jugador haurà d'explorar per poder localitzar els recursos que hi trobem i on s'haurà d'enfrontar a enemics. Aquest entorn és probablement el més important, ja que serà aquí on realment el jugador avanci en el progrés del joc.

Figura 5.1.1.2.1: Esbós de Underworld

5.1.2. La mecànica del joc, els reptes que ha d'afrontar el jugador i les accions possibles

La definició de les mecàniques és un dels punts més important a l'hora de donar personalitat al joc, ja que són les regles que regeixen el món i que orientarà l'experiència del jugador cap a un gènere concret. En aquest cas, i a causa que el gènere pot tenir una gran varietat, podem trobar un gran número de mecàniques i accions. Tal i com s'esmenta en apartats anteriors, el joc està inspìrat en un seguit de mecàniques que, combinades entre si, donen lloc a una fórmula interessant i novedosa.

A l'inici del projecte ens vam plantejar aplicar mecàniques que permetessin al jugador interactuar amb el món i, sobretot, plantejar un entorn que se sentís viu, on el jugador hauria d'interactuar per poder avançar. Tot i que des del principi teníem clar el gènere, van aparèixer diverses idees sobre com portar-lo a la pràctica. Entre aquestes teníem molt clar que volíem que el jugador tingués la capacitat de produir recursos i generar objectes. Aquest és una de les mecàniques principals. Un altre aspecte que van tenir molt clara és la localització i l'espai de joc on se centrará l'acció. Parlem de les illes flotants, amb la intenció d'aportar un *toc* més místic i diferenciador.

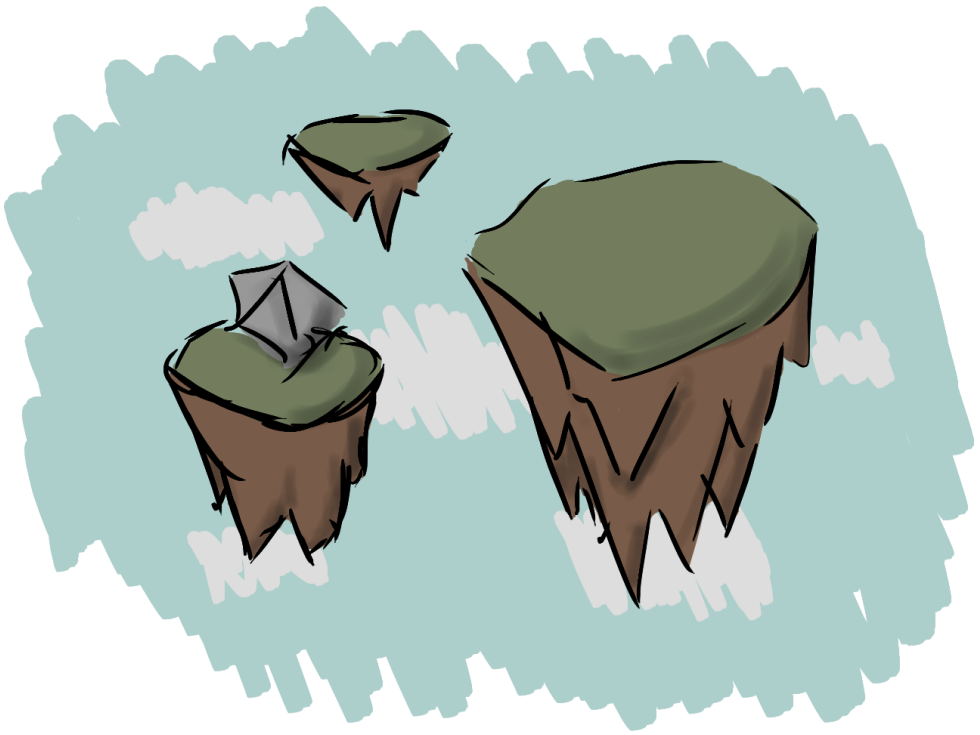


Figura 5.1.2.1: Esbós illes flotants

Figura 5.1.2.2: Imatge de referència de la pel·lícula Avatar

Els reptes que haurà de superar el jugador per tal de completar el joc serà el de completar els reptes que es presentaran al *Underworld*, enfrontar-se a enemics i eliminant el boss[2] final. Durant tot el recorregut el jugador haurà d'explorar el món inferior recol·lectant recursos i eliminant enemics que li tancaran el pas.

[2] Figura que representa un enemic final més poderòs i únic.

5.1.3. Objectes, recursos i interaccions que pot fer el jugador

Un dels pilars del joc és la capacitat de recol·lectar elements per poder produir d'altres, per tant, necessitem que existeixin recursos de diferents tipus per poder oferir varietat i afegir complexitat al joc.

Els elements que trobarem inicialment són:

- **Fusta:** Element que podem recol·lectar mitjançant una destral.
- **Pedra:** Element que podem recol·lectar mitjançant un pic.
- **Fibra:** Element que podem recol·lectar mitjançant l'espasa.
- **Bronze:** Element que trobarem en el *Underworld* i que haurem de recol·lectar amb un pic.

Cadascun dels elements esmentats anteriorment seran el resultat de l'extracció de la matèria dels objectes interactuables que podrà trobar el jugador al llarg del joc.

Figura 5.1.3.1: Esbós de materials del joc

A partir d'aquestes matèries podem crear objectes que ens ajuden a avançar en la trama. També els podrem fer servir per obtenir millores que, de la mateixa manera, ajudaran al jugador a enfrontar-se a futurs obstacles. Alguns d'aquests objectes els podem col·locar físicament a la nostra illa, en canvi, n'hi han d'altres que els manindrà el jugador (pic millorat, espasa millorada,...).

Entre els elements que podem crear inicialment trobem:

- **Foguera:** Element que el jugador pot col·locar en la illa i que farà servir per elaborar menjar.
- **Test:** Element que el jugador pot col·locar en la illa i que farà servir per plantar i generar aliments.
- **Espasa:** Una de les tres eines principals del jugador que farà servir per atacar enemics i recol·lectar recursos bàsics.
- **Pic:** Eina que farà servir el jugador per extreure minerals i pedra.
- **Destral:** Eina feta servir per extreure fusta i eliminar alguns elements decoratius de tipus orgànic (arbustos, soques...).

Figura 5.1.3.2: Esbós d'elements creables

Tots aquests elements els podrem crear mitjançant el panell de creació que trobarem a la casa.

5.1.4. Economia interna del joc

L'economía del joc és aquell element que moltes vegades va lligat al concepte d'objectiu i moneda de canvi. En el primer cas es tracta d'un valor que indica al jugador el pròxim que ha estat a assolir el següent nivell o simplement el resultat del seu esforç en la partida. Per altra banda el segon dels punts fa referència a aquell objecte que farem servir per obtenir-ne d'altres que necessitem per avançar en el joc. En aquest cas podem considerar economía del joc a la quantitat de matèria prima que té l'usuari, ja que aquesta serà la que

s'intercanvia per altres objectes més complexes que serviran a l'usuari per poder generar-ne més. Amb tot això no volem dir que a més matèria prima tingui l'usuari en l'inventari, més puntuació final tindrà, ja que l'objectiu no és aquest, sinó derrotar l'enemic final.

5.1.5. Estudi dels diferents nivells del joc

El punts més característic del videojoc és la divisió que existeix entre els dos móns als quals el jugador pot accedir i el propòsit de cadascun. Mentre que en el món superior, on apareix el jugador, es tracta d'un món pacífic on créixer investigant, recol·lectant i construint, en el segon es presentaran reptes per poder avançar en la trama principal.

El *Upperworld* està dissenyat de tal manera que el jugador se senti en un lloc tranquil i amigable, format per illes poblades per natura i on poder créixer de manera segura. En aquest entorn el jugador no trobarà cap element hostil amb el que combatir i el principal protagonista serà l'exploració.

Figura 5.1.5.1: Captura de Upperworld

En canvi, el *Underworld* representa totalment lo invers al *Upperworld* ja que aquest és un món fosc i perillós que ens farà plantejar-nos si volem endinsar-nos o no. En aquest entorn podem trobar recursos i enemics els quals haurem de derrotar per finalment lluitar contra el boss.

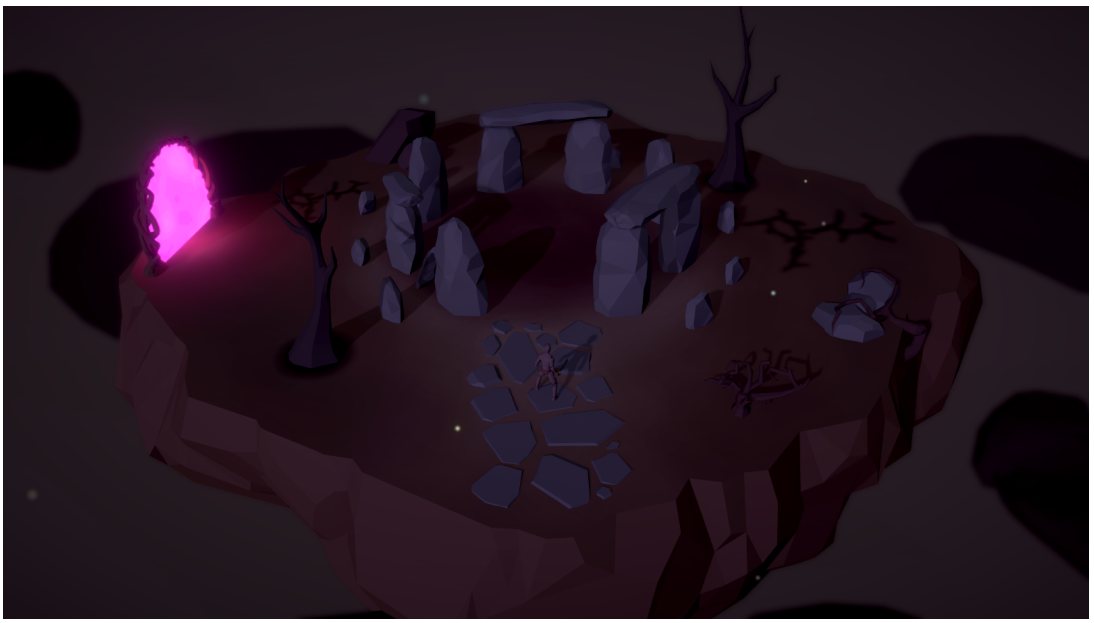


Figura 5.1.5.2: Captura de Underworld

5.2. Estudi i disseny de personatges

L'estudi i el disseny del personatge ens permet conèixer les motivacions que el porta a complir uns objectius concrets. En aquest cas, i com un dels objectius principals del projecte era la creació d'un món i l'implementació de mecàniques que interaccionen amb ell, tant el pes narratiu del personatge i el disseny han passat a un segon pla.
Tot i que no s'ha dut a terme una implementació de les idees, el personatge no és un simple ninot, aquest té una proposta narrativa i una primera idea de disseny que veurem a continuació.

5.2.1. Pes narratiu dels personatges

Tal i com s'esmenta al inici, el pes narratiu del personatge de cares al prototip presentat és pràcticament nul i per tant no té un paper protagonista en el projecte.
Una de les idees proposades durant la pluja d'idees de les primeres setmanes és la presentació d'un personatge el qual ha perdut la memòria i que es troba en aquest conjunt d'illes. Aquest personatge s'anomena Kairu, és un antic guerrer alat[3] que va caure durant la guerra entre els dos mons. L'objectiu d'en Kairu és recuperar els records. Per fer-ho necessita recuperar les plomes, que li van caure durant la batalla, i que contenen el seu esperit guerrer.

5.2.2. Els personatges com a base de la jugabilitat

Actualment el prototip de personatge que es presenta es una representació d'en Kairu. Aquesta representació pretén substituir temporalment el que serà el personatge final, amb l'objectiu d'implementar totes aquelles mecàniques que ens vam proposar que el joc tingués.
Una de les bases jugables que formen part del nostre full de ruta és la inclusió de millores per al personatge a mesura que obté les plomes perdudes, d'aquesta manera incrementarà les habilitats i les estadístiques, permetint-lo enfrontar-se a altres perills.
Un altre de les característiques d'aquest personatge és la capacitat de fer servir l'energía per realitzar parades. Al realitzar aquestes parades obté la capacitat de realitzar un atac potenciat. L'energía que l'envolta dependrà del número de plomes o les millores que hagi obtingut durant el joc i es recuperarà després d'un temps de consumir-la (Veure apartat 5.7.2 per conèixer la seva representació).

[3] S'entén d'aquella entitat amb la capacitat de volar. Solen tenir ales com les dels ocells.

5.2.3. Estil artístic (condicionat per la jugabilitat)

L'estil artístic del joc és un dels fonaments del projecte, ja que en aquest cas s'hi ha treballat expressament amb l'objectiu de generar tot el contingut visualment sense la utilització de recursos externs. En aquest cas, i com que cap dels dos desenvolupadors té un perfil artístic massa clar, hem optar per un estil artístic simplificat amb l'objectiu de potenciar els colors i aconseguir una armonía generalitzada que fos atractiva pel jugador.

Figura 5.2.3.1: Captura de Upperworld amb interfíce

Algunes de les referències artístiques ja les hem esmentat anteriorment (revisar secció 2.2.1). Gràcies a la "simplicitat" dels gràfics hem estat capaços de generar i iterar nombroses vegades fins a trobar un estil propi que identifiques el projecte i el fes més únic.

Una de les referències més importants del projecte la podem trobar en el Underworld, ja que es va realitzar una cerca amb la finalitat de recrear un monument antic anomenat "cromlec". Pensem que aquest element aporta el punt de misteri que volem crear en aquest ambient en específic.

Figura 5.2.3.2: Cromlec de Underworld

Figura 5.2.3.2: Cromlec de Stonehenge

Per altra banda, i a la mateixa escena, també hem fet servir una referència a un altre joc per crear un portal que ens portés a la zona d'acció. La referència pertany al joc "Hellblade: Senua's Sacrifice" on podem trobar portals similars.

Figura 5.2.3.3: Comparativa portal Underworld amb portal de Hellblade

5.2.4. Coherència, característiques físiques i psicologia

En Kairu és un personatge que està inspirat en diverses figures guerrers de diverses civilitzacions. Algunes de les referències són:

- **Guerrer azteca:** En quant a la referència presa d'aquest guerrer, volem destacar el seu caràcter feroç i les seves vestimentes. També volem fer èmfasi a la utilització de les plomes per part de la cultura per simbolitzar poder.

Figura 5.2.4.1: Referència guerrers aztecas

- **Guerrer maori:** D'aquest figura extraiem alguns dels tatuatges tradicionals d'aquesta tribu. Aquests tatuatges pretenen aportar un toc de personalitat al personatge i fer-los servir com a característica identificativa.

Figura 5.2.4.2: Personatge de la pel·lícula Vaiana i fotografia d'un tatuatge tradicional maori

Com es tracta d'una aventura en solitari, hem volgut fer que en Kairu es tracti d'un personatge fort i valent capaç de superar tots els reptes que se li presentin. Tot això combinat amb ingeni i coneixement de la matèria que el permetràn construïr diferents elements que l'ajudaràn en el camí.

5.2.5. Els enemics

Durant la fase d'exploració del *Underworld* ens trobarem amb l'enemic principal de baix rang, aquest és el tentacle. Aquesta figura prové del enemic final i aquests es distribueixen aleatòriament al llarg de totes les illes. El tentacle és la principal figura hostil del joc i ens permet aplicar de manera activa les mecàniques de combat que hem implementat.
Pel que fa a les característiques físiques del tentacle, aquestes son molt senzilles ja que només es tracta de l'extensió final i que no té cap cop adherit.

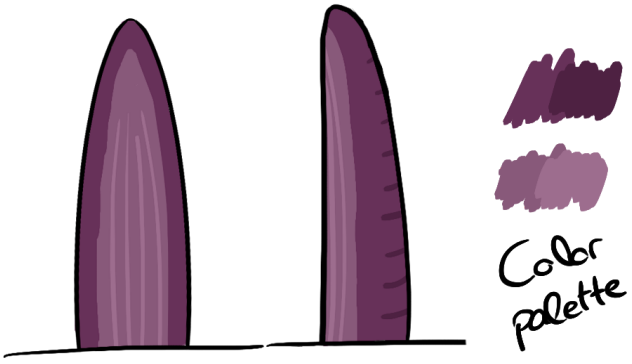


Figura 5.2.5.1: Esbós enemic

El tentacle té la capacitat de seguir i atacar l'objectiu realitzant una animació de balanceig que finalment acaba amb un impacte a terra. Si durant l'animació d'atac el jugador l'ataca, podrà interrompre el tentacle. Podem veure una representació del moviment d'atac a la Figura 5.2.5.2.

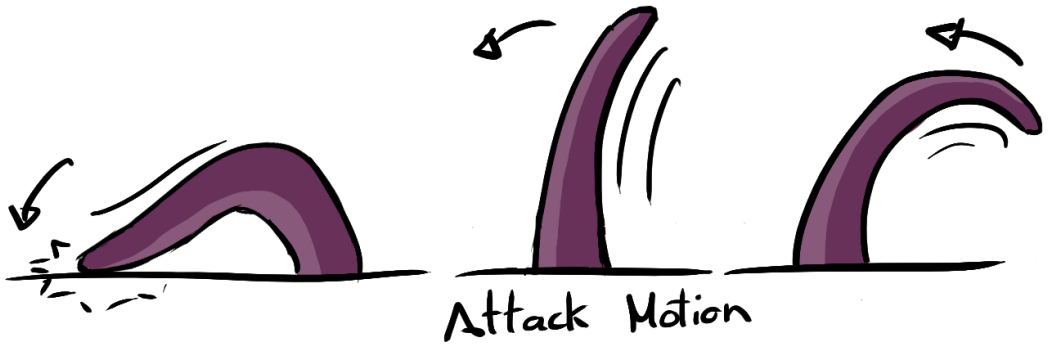


Figura 5.2.5.2: Esbós de l'atac del tentacle

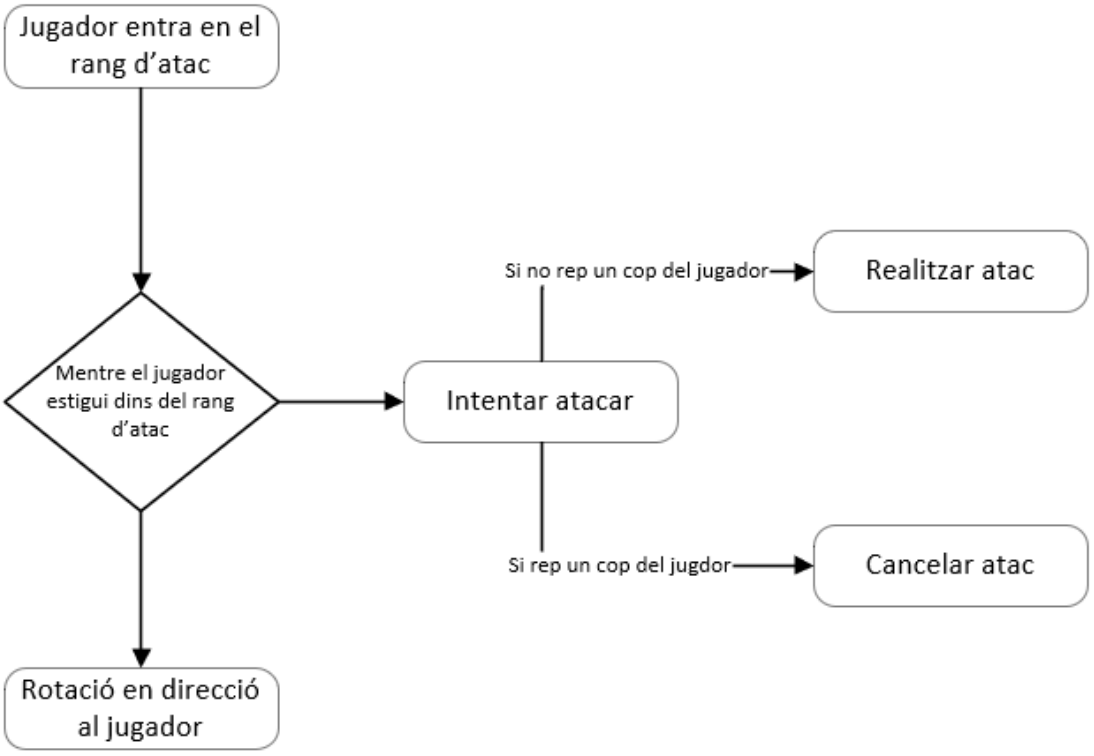


Figura 5.2.5.2: Diagrama d'estats del tentacle

En aquest punt el jugador tindrà la capacitat de parar l'atac per beneficiar-se de la seva habilitat pasiva, i realitzar un atac doble.

Algunes de les referències que hem fet servir per dissenyar el tentacle i la seva paleta de colors han estat de videojocs com World of Warcraft, il·lustracions diverses d'artistes independents o la película de disney Monstruos S.A.

Figura 5.2.5.3: Exemples de tentacles en diferents estils

5.2.6. El boss. Disseny de gameplay.

L'enemic final que podem trobar en el *Underworld* té les característiques d'un pop, el qual es troba en un medi diferent a l'habitual com és el terrestre. Aquest enemic controla les terres per on el jugador haurà d'explorar i serà l'objectiu a vèncer per superar l'escenari i, per tant, el joc.

Figura 5.2.6.1: Imatge boss

Per la realització d'aquest enemic hem agafat referències del propi animal marí, i els seus colors intenten generar un sentiment de perillositat i d'incertesa.

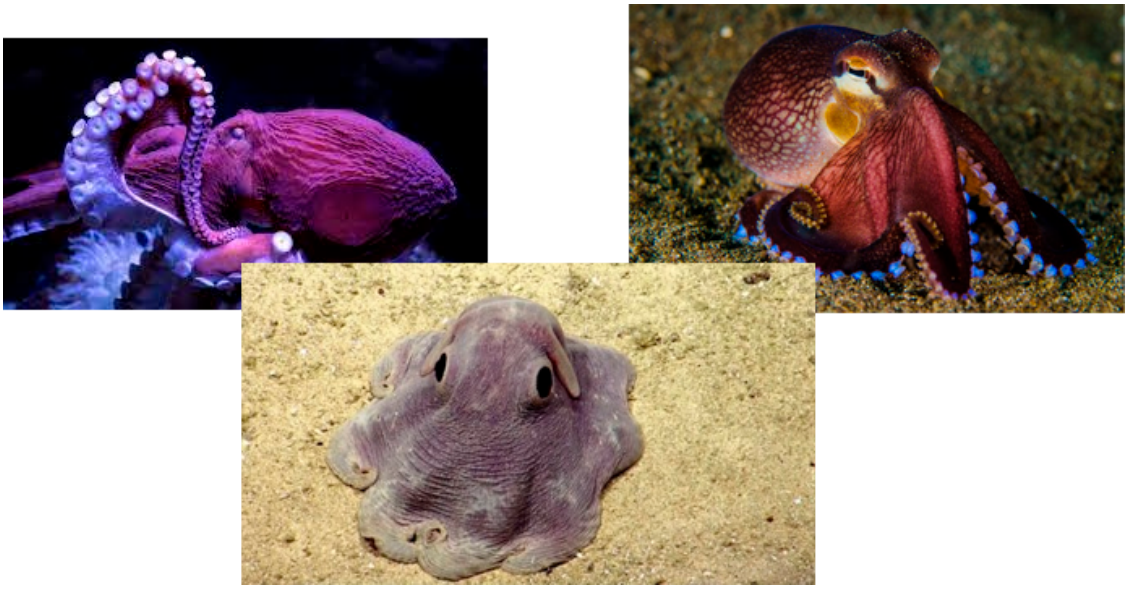


Figura 5.2.6.2: Fotografies de diferents espècies de pops

Pel que fa al comportament envers el jugador, aquest té un nivell de complexitat superior a un enemic normal com és el tentacle, i tot i no tenir atacs propis des del seu nucli, s'ajudarà de tentacles generats per ell mateix per ferir al jugador.

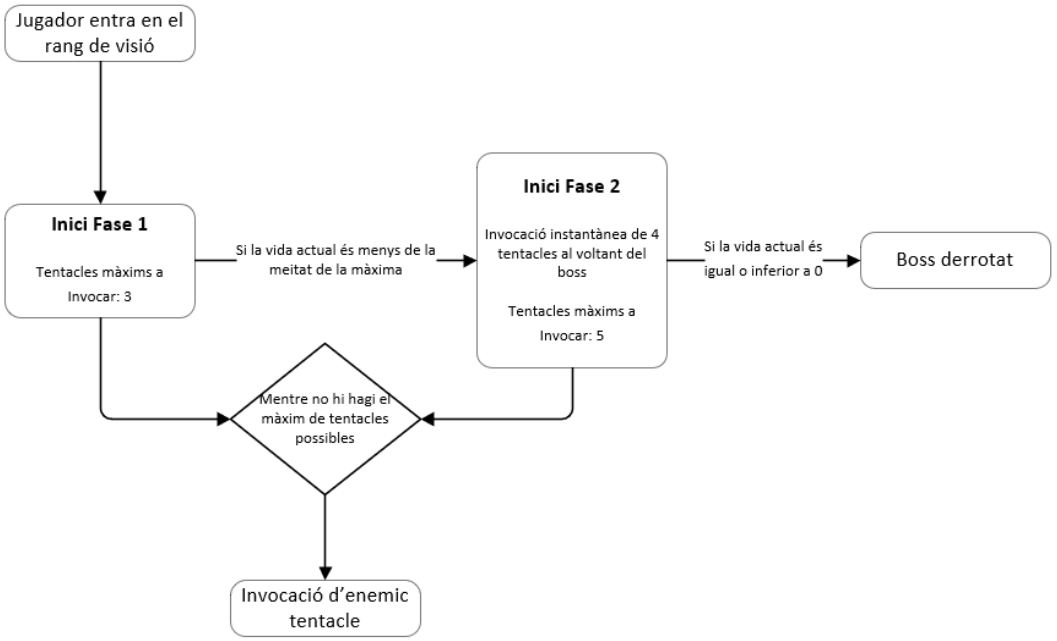


Figura 5.2.6.3: Diagrama de flux de les fases del boss

Com es pot veure en la Figura 5.2.6.2, l'enemic boss té 2 fases. La primera s'inicia en quan el jugador entra en el rang de visió del boss. En aquesta fase el boss invoacarà tentacles per a intentar acabar amb el jugador. Quan la vida actual del boss es inferior a la meitat de la seva vida màxima, iniciarà la fase 2. En aquesta fase podrà invocar més tentacles alhora, a més d'invocar 4 tentacles al seu voltant per a protegir-se del jugador. Un cop la seva vida arribi a 0, el boss serà derrotat i morirà juntament amb els tentacles que encara estiguessin vius.

5.3. Narrativa

La narrativa en el videojocs és un component que ha anat en augment amb el pas del temps, i és que una història amb un component fort, no necessita d'unes mecàniques extraordinàries i moltes vegades són aquestes les que perduren en el temps. Tot i això, el jocs que mantenen un fil narratiu més discret també tenen el seu lloc en el mercat i molt d'aquestes predominen en alguns gèneres. En aquest cas el fil argumental que segueix el projecte és mínim i la jugabilitat està centrada en la multitud de mecàniques i possibilitats que ofereixen.

5.3.1. Notes importants

Tal i com s'esmenta, l'argument és extern a la trama que es viu en el joc, i el jugador només influeix en el objectiu final, que és recuperar les plomes perdudes den Kairu derrotant a l'enemic final.

5.3.2. Sinopsi

En Kairu és un jove guerrer d'una tribu alada que ha estat lluitant contra les forces del món inferior des de fa milers d'anys. És tant el temps que porten batallant que el motiu s'ha anat difuminant i només els motiva la rivalitat.
Durant l'última batalla, les forces enemigues van dur a terme un experiment militar sobre el camp de batalla. Aquests van fer servir una màgia antiga per canalitzar un poder ancestral que arrabassa als éssers alats el seu esperit, sent fatal pels guerrers de la tribu. Quan l'esquadra d'en Kairu va rebre l'impacte van perdre les capacitats de mantenir-se en l'aire i van caure en picat. Mentre queia, i abans de desmaiar-se, va notar com una part d'ell se separava i llavors tot es va fer fosc. Ara en Kairu es troba en una illa desconeguda, sense records de la batalla ni qui és ell. A partir d'aquí haurà d'explorar l'entorn que l'envolta buscant d'on prové i qui és.

5.4 Mons de joc

5.4.1. Dimensió física

Tal i com s'esmenta en l'apartat de l'espai de joc (veure secció 5.1.1) l'entorn està format per dos mons, el primer format per un conjunt d'illes en la superfície dels nùvols i l'altre conjunt situat a sota.

La raó del món i la seva composició té diverses raons:

- La primera és l'originalitat que suposa pel projecte el fer que el món estigui situat en illes flotants sense cap explicació, i és que aquest fet aporta un misticisme que atrau l'atenció del jugador.
- El segon dels punts és la limitació de moviment del jugador i el compliment de reptes per augmentar el camp d'acció. Segons l'illa a la que vulguem accedir haurem de completar un repte. En el cas de la primera illa, haurà de reconstruir un pont mitjançant matèria primera que prèviament ha extret.
- Recolzat per la segona raó, la possibilitat d'oferir més reptes i recompenses a l'exploració.

En quant a la decisió de la forma i les dimensions, aquests paràmetres han estat establerts segons les necessitats d'incloure els elements que l'emplenen, es a dir, els recursos i els elements decoratius.

En la primera de les illes, on en Kairu apareix, trobem el nucli del joc a partir d'on es desenvolupa el creixement i properament l'acció mitjançant la plataforma. Per altra banda, el pont, que comunica l'illa principal amb les adjacents.
Si anem al *Underworld*, l'illa on apareixen és el nexe amb la resta d'illes que el formen, i haurem de travessar el portal que tenim a l'esquerra per poder accedir-hi.

5.4.2. Dimensió temporal

El temps en el que es situa l'acció no és important per la jugabilitat. Per altra banda, el temps que passa durant la jugabilitat sí que és important, ja que hi han elements que es veuen afectats pel temps, com per exemple les plantes que es poden cultivar amb els testos. Un altre exemple que si afecta és el fet d'entrar en el portal, ja que cada cop que entrem, el camí i la formació d'illes haurà canviat.

5.4.3. Dimensió ambiental

En quant a la dimensió ambiental, ens situem en un món estrany inclús pel propi protagonista de la història. Inicialment no necessitem posar-nos en context amb la història ja que no creiem que sigui un dels pilars del projecte.
En quant a referències que podem trobar dins del joc, i que poden resultar-nos familiars, veurem el cromlec del *Underworld*, que tampoc trobem que tingui un pes massa representatiu en cap aspecte. A priori, l'únic que pretén és afegir un toc místic a l'escena.

5.4.4. Dimensió emocional

Les motivacions del jugador estan guiades a l'exploració i el desig d'avançar mitjançant les mecàniques de creació i col·locació d'elements a la illa principal. D'aquesta manera es recompensan les hores dedicades amb el progrès, que es veurà reflectit en l'aspecte de l'illa principal. Per altra banda, una motivació prou important, és la de superar l'enemic final i finalitzar el joc.

5.4.5. Aspectes ètics

Com que el pes narratiu és molt dèbil, no creiem que existeixi un aspecte ètic massa clar en aquests moments del projecte. En el cas que la part narrativa agafes més pes, podríem considerar l'empatia com a eina principal per connectar amb el personatge i ajudar-lo a aconseguir l'objectiu de recuperar el seu esperit.

5.5 Game layout charts

El joc segueix la següent estructura:

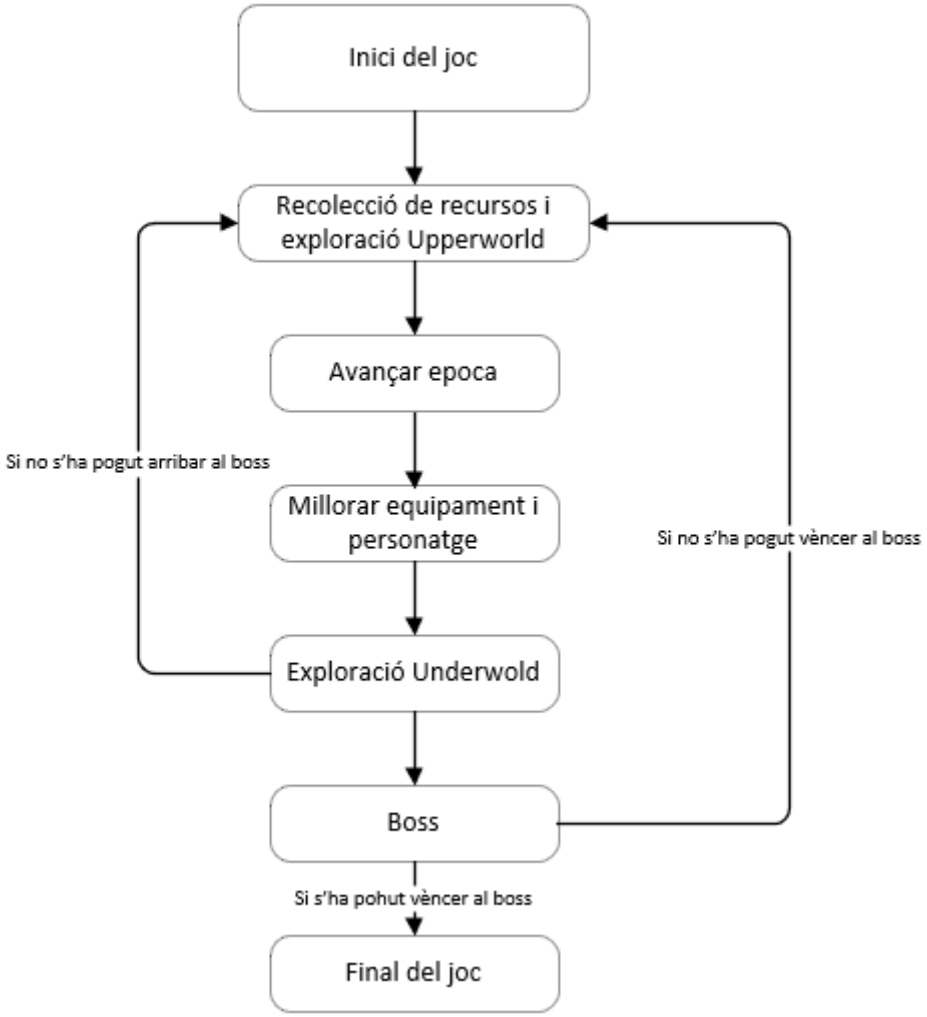


Figura 5.5.1: Estructura de les fases del joc

Com es pot veure en la Figura 5.5.1, el joc no té una estructura lineal, sinó que és una estructura repetitiva. En aquesta estructura el jugador haurà de tornar a una fase anterior si no és capaç de seguir avançant. En cada nova iteració el jugador podrà millorar, tant obtenint millors recursos com millorant les eines, per a que sigui més fàcil avançar. A més, sentirà una sensació de progrés, ja que tasques que anteriorment li resultaven mes complicades cada cop seran més fàcils gràcies a les millores que obtindrà amb les iteracions.

5.6 Disseny de nivells i creació d'ambients

Un dels reptes que ens hem proposat en aquest projecte és la creació de tots els elements visuals amb l'objectiu d'elaborar un producte propi en tots els sentits. Per això es va realitzar un estudi a nivell gràfic per determinar l'estil. Tail i com s'esmenta anteriorment (revisar Apartat 2.2.1), finalment l'estil que s'ha decidit aplicar té unes característiques que permeten a l'usuari gaudir de les gammes cromàtiques sense la intenció de generar contingut realista. Per fer tot això, primerament es va dissenyar l'escenari principal. Un cop es va decidir que seran illes flotants, va començar el procés de disseny proposant diferents estils d'illes.

Durant tot aquest procés vam observar que, a causa de la posició de la càmera i les seves característiques, algunes illes podien quedar deformades, inclús donar la sensació que no estaven alineades. Finalment es va decidir fer una illa de forma gairebé rectangular, en el cas de l'illa principal, per mostrar a l'usuari els camins a seguir.

5.6.1. Modelat i texturitzat d'illes

Per elaborar les illes s'han realitzat diferents procediments amb l'eina Blender per generar el model. Com que es tracta d'un element orgànic sense unes formes gaire definides, ha suposat un repte a nivell de disseny per aconseguir un resultat que no semblés gaire artificial i que complís amb tots els requisits que volíem.

El procés de creació d'illes s'ha realitzat a partir de una figura bàsica que serviria per donar forma a l'illa final, tal i com s'observa a la Figura 5.6.1.1.

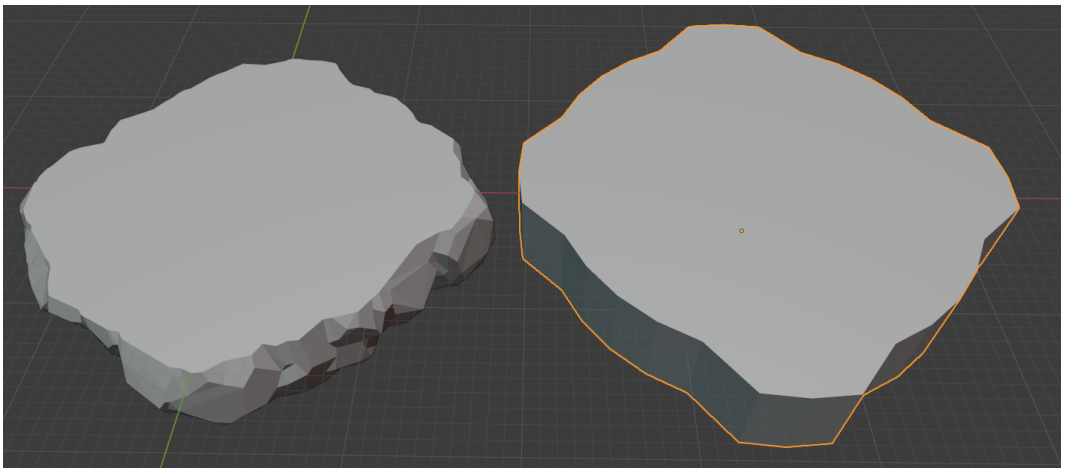


Figura 5.6.1.1. Forma bàsica per guiar el modelat en el següents passos. A l'esquerra el resultat final, a la dreta el model de referència.

Un cop ja tenim aquesta forma inicial, generarem un element amb forma de roca que farem servir per replicar-lo al voltant de la primera figura, i així formar un relleu. Veure el model fet servir a la Figura 5.6.1.2.

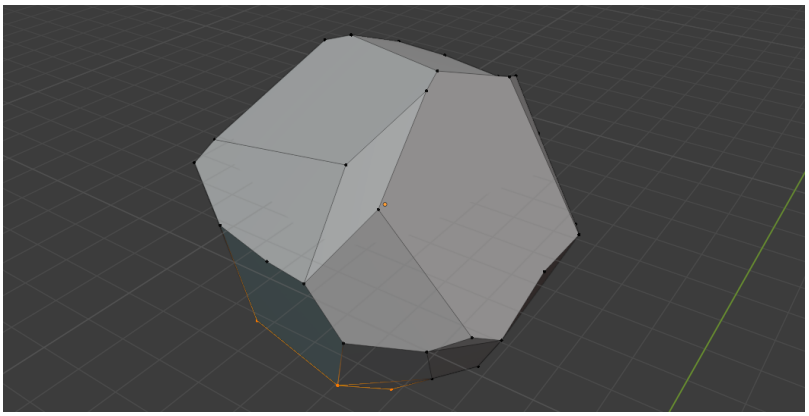


Figura 5.6.1.2. Model de pedra utilitzat en el procés de creació d'illes.

Un cop ja tenim la figura rocosa, aplicarem un modificador de Blender que comunament es fa servir per generar cabell a les superfícies. Gràcies a aquesta eina aconseguirem replicar la roca sobre la figura geomètrica inicial, repartint-la aleatòriament aplicant-hi una rotació i una densitat determinada. Els valor de densitat dependran de les característiques que vulguem que la illa tingui. De totes manres, més endevant s'optimitzarà per eliminar tota aquella informació geomètrica que mai es veurà. Podem veure un exemple d'implementació a la Figura 5.6.1.3.

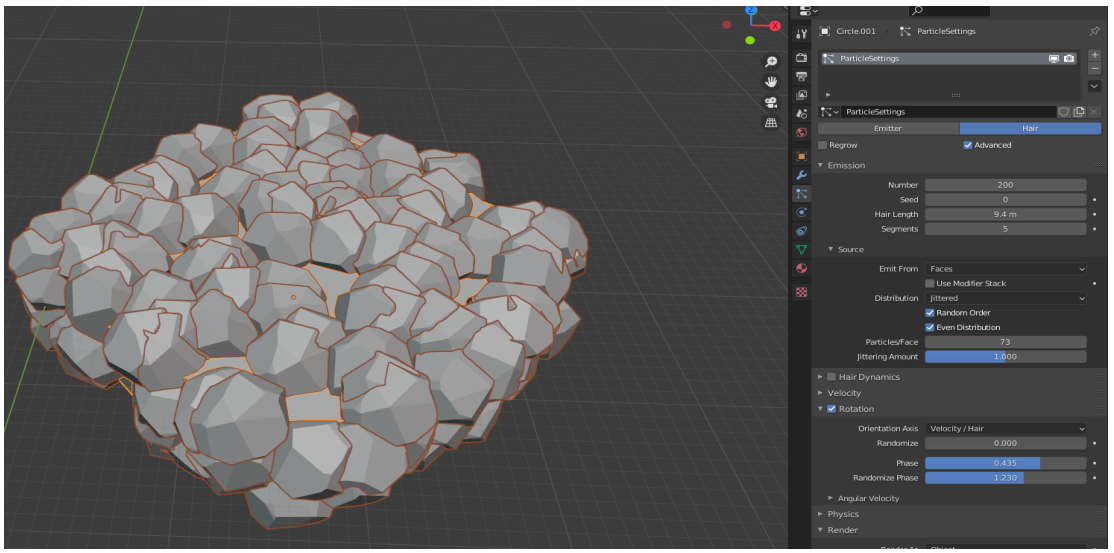


Figura 5.6.1.3. Procés de creació d'una illa, aplicant un efecte de partícules.

Tot seguit apliquem el modificador, fent efectiu el canvi, i aplicarem un seguit de moviment coordinats entre totes les roques per afegir encara més complexitat l'objecte. Aquests moviments es realitzaràn a petita escala per cadascuna de les roques que s'han generat. En aquest moment ja estem llestos per combinar totes les roques per formar-ne una i per finalment tallar el punt desitjat per formar la superfície jugable. Per "retallar-la" utilitzarem una operació booleana de diferència tal i com podem veure a la Figura 5.6.1.4.

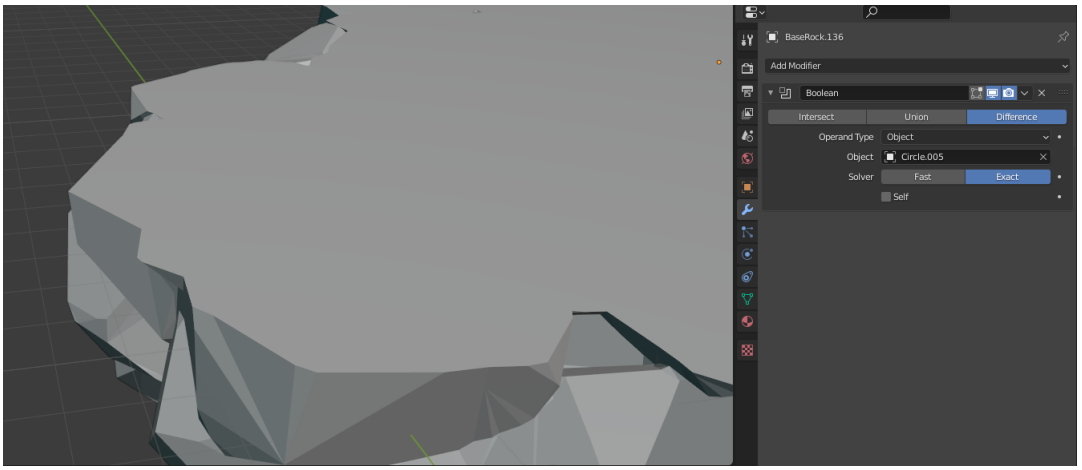


Figura 5.6.1.4. Ap licació d'una operació booleana.

Un altre element molt important per afegir les barreres físiques que comporten el món, és la creació d'unes parets invisibles. Aquestes parets no tindràn cap textura i es faran servir a Unity per aplicar un camp de col·lisió per evitar que el jugador caigui. Aquesta tècnica ens permetrà simplificar els límits de la illa i ens evitarà problemes de cares a la jugabilitat. També és molt important afegir forats o portes temporals per allà on el jugador pugui passar, sigui a l'inici o més endavant durant el gameplay.

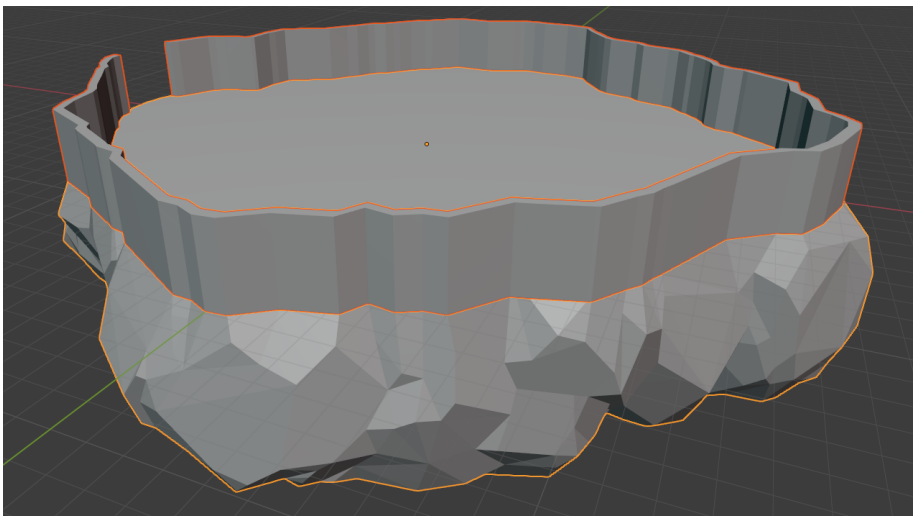


Figura 5.6.1.5. Model de l'illa amb les parets delimitants.

Ara que ja tenim la figura base, és molt important revisar tots els vertex que hagin pogut quedar malformats, eliminar els que siguin massa propers e innecessaris, mitjançant l'eina d'unió de vèrtex per distància, i finalment cal ajustar el contorn per obtenir la forma desitjada. Finalment el model de l'illa ja està llest i podrem continuar amb el procés de texturitzat. Per texturitzar l'illa, primer ens haurem d'assegurar que totes les cares del model es troben correctament orientats. Per fer-ho ens ajudarem de l'eina de visualització d'orientació de cares. A la Figura 5.6.1.6 podem veure que el model està completament pintat de color blau, això indica que l'orientació de les cares és correcte.

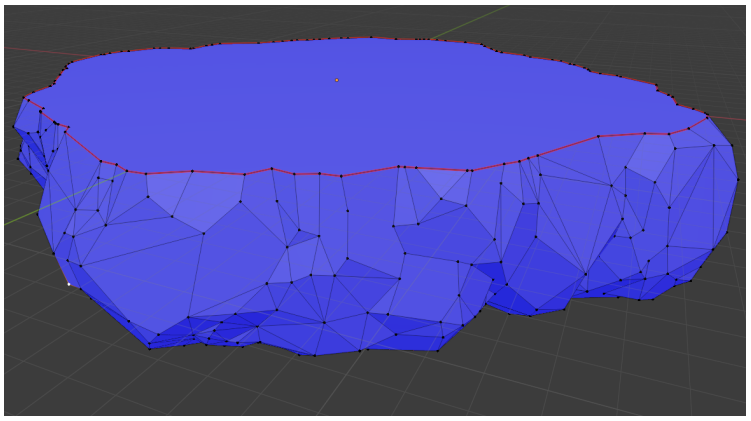


Figura 5.6.1.6. Mode de visualització de Blender per veure l'orientació de les cares d'un model.

Si totes les cares ja estan orientades correctament podrem continuar amb el procés de unwrapping[4]. Un cop aplicat el unwrap, accedim a la pestanya d'edició per intentar ajustar les cares i donar més prioritat a aquelles que tindran més detall, en aquest cas la cara superior. Podem observar amb més detall el resultat a la figura 5.6.1.7.

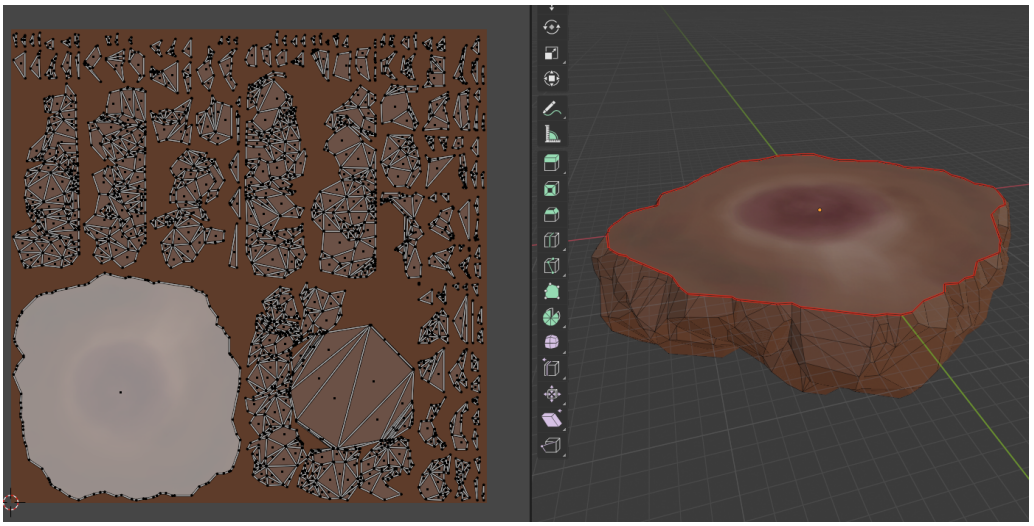


Figura 5.6.1.7. Illa principal del *Underworld* texturitzada.

[4] Procés pel qual s'extreuen les cares d'una figura geomètrica sobre un pla per aplicar una textura.

Un cop texturitzat ja podem exportar el model i la imatge de la textura i portar-la a Unity 3D, on haurem de generar un material amb la textura de l'illa i assignar-la al model. Ara ja podem afegir l'illa a l'escena. Podem veure el resultat final en la Figura 5.6.1.8.

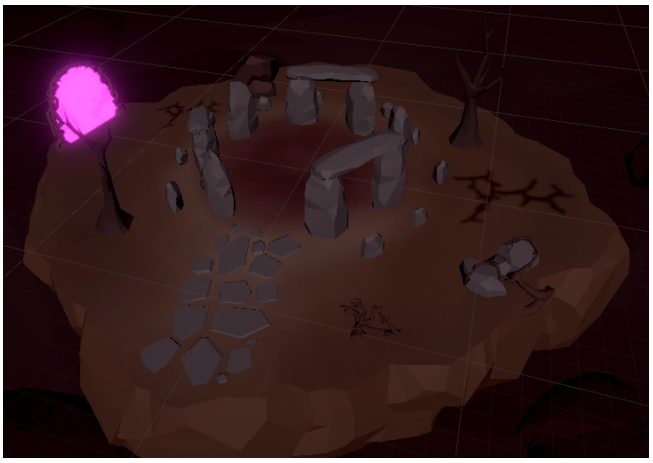


Figura 5.6.1.8. Resultat final de l'illa principal del *Underworld* amb els elements de decoració.

5.6.2 Modelat i texturitzat de la vegetació

La vegetació és un dels elements claus a l'hora de poblar un entorn natural. Com que aquests són uns elements orgànics, hem necessitat realitzar un gran nombre de proves fins a trobar una fórmula que permetés generar-los d'una manera prou senzilla a l'hora que fossin realistes i afins a l'estil escollit.

5.6.2.1. Arbres i les seves fases

Els arbres ha estat un dels elements que més hem iterat fins a aconseguir la fórmula que necessitàvem. Des de l'inici, vàrem determinar que seria un dels elements principals, ja que d'ell se n'extreia un element bàsic, la fusta.

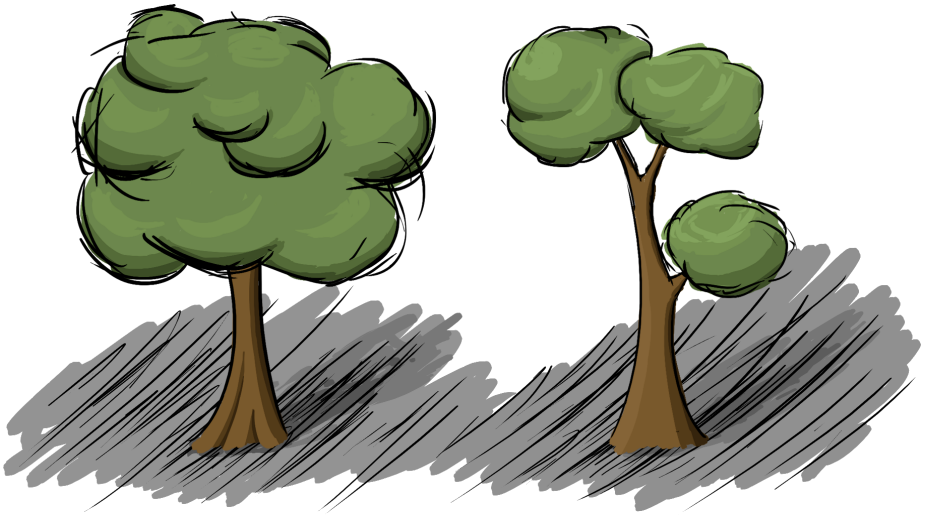


Figura 5.6.2.1. Esbossos dels arbres

En la Figura 5.6.2.1.1 podem veure com han anat evolucionant a mesura que el projecte ha anat avançant i l'estil del joc ha evolucionat:

Figura 5.6.2.1.1: Evolució de l'arbre

Com es pot veure en la Figura 5.6.2.1.2, vam passar d'un arbre purament poligonal, a un arbre més complex que conté conjunts de fulles i que aporta un volum molt més realista.

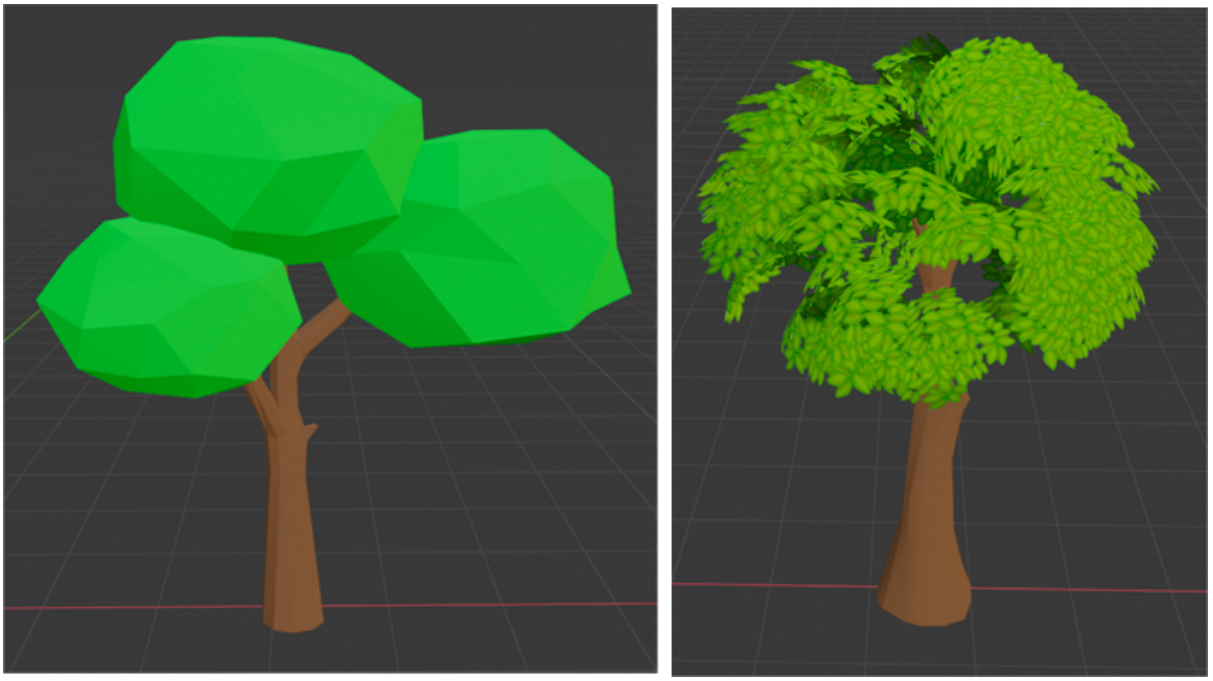


Figura 5.6.2.1.2: Comparativa d'evolució de l'arbre

Per realitzar el model final dels arbres, primer l'hem de dividir entre el tronc i les fulles. Per modelar el tronc, primer cal traçar un camí de vèrtex que haurem d'anar ajustant segons la forma que vulguem. A continuació generarem una malla fent servir el modificador anomenat "Skin", com es pot veure en la Figura 5.6.2.1.3.

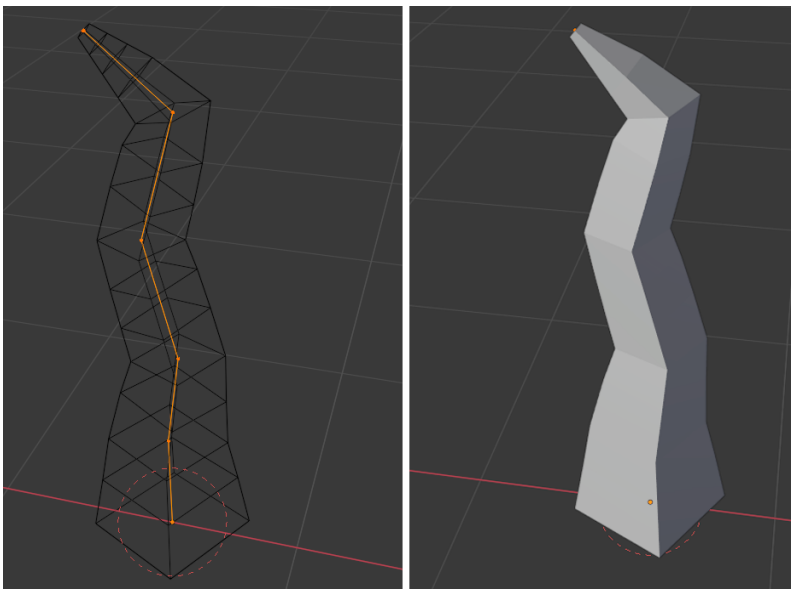


Figura 5.6.2.1.3: Malla generada amb el modificador Skin

Un cop generat el tronc, hem de generar la copa. Per començar, generem un parell d'esferes i les deformem fins obtenir la figura desitjada. Ara que tenim la forma de la copa, necessitarem generar el recurs que després integrarem en el model amb el modificador de cabell que ja hem fet servir en l'apartat de les illes. El recurs que necessitem està format per un conjunt de fulles, aquest conjunt s'ha generat mitjançant photoshop, i el resultat es pot veure en la Figura 5.6.2.1.4.

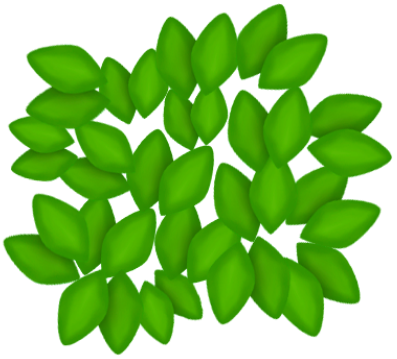


Figura 5.6.2.1.4: Conjunt de fulles generat amb Photoshop

Per preparar el component necessari primer haurem de generar un pla i col·locar la textura. Quan tinguem el recurs, executarem la mateixa metodología que vam seguir amb les illes, definint la densitat i l'orientació com es pot veure en la Figura 5.6.2.1.5.

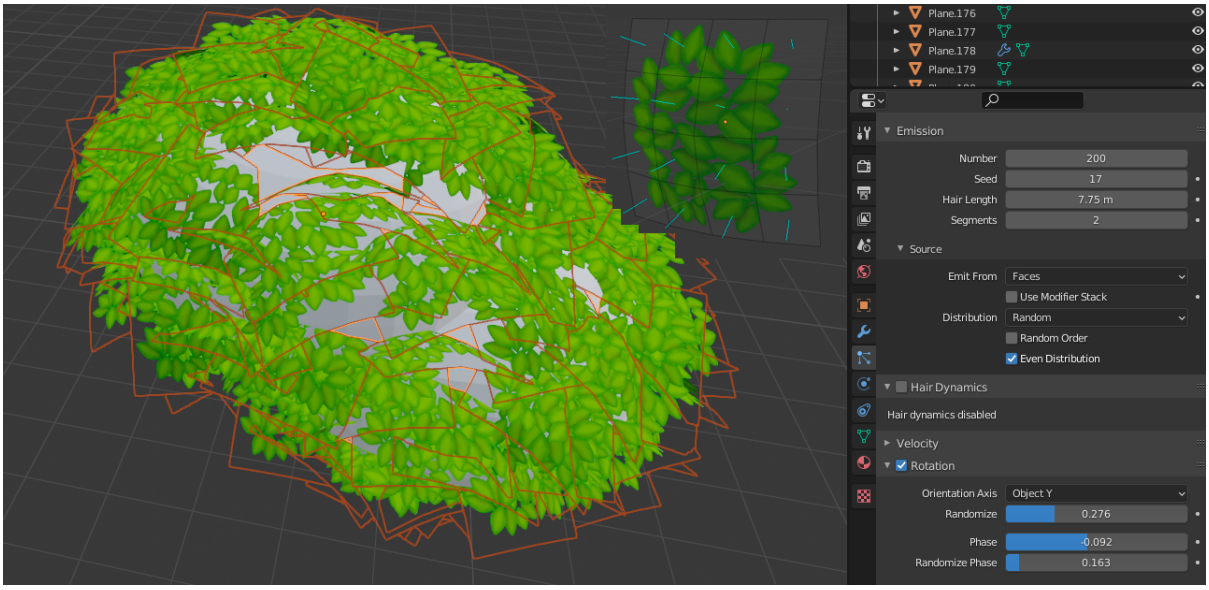


Figura 5.6.2.1.5: Procés de creació del fullatge d'un arbre.

Finalment ja tindrem el model final, només caldrà definir el punt d'origen i realitzar les exportacions adients per portar-lo a Unity. Cal recordar, que un cop exportat a Unity, haurem de generar el material i assignar les textures corresponents per obtenir el resultat desitjat com es pot veure en la Figura 5.6.2.1.6.

Figura 5.6.2.1.6: Escena amb els models finals.

Unes de les característiques del joc és la capacitat de poder generar alguns recursos de forma activa, en aquest cas podem plantar arbres. Per dur a terme la representació visual d'aquest creixement, cal dissenyar i modelar algunes de les etapes que ens trobarem durant

aquestes fases. Com es pot veure en la Figura 5.6.2.1.7, hem decidir segmentar-ho en 4 fases, essent l'última els models que podem observar a la Figura 5.6.2.1.6.

Figura 5.6.2.1.6: Fases de creixement d'un arbre

Les tècniques utilitzades han estat les mateixes que en la realització de l'última fase, ja explicada a l'inici del apartat.

5.6.2.2. Arbust

L'arbust és un altre del components essencials del *Upperworld* ja que és la font de la fibra, un recurs necessari pel progrés del joc.

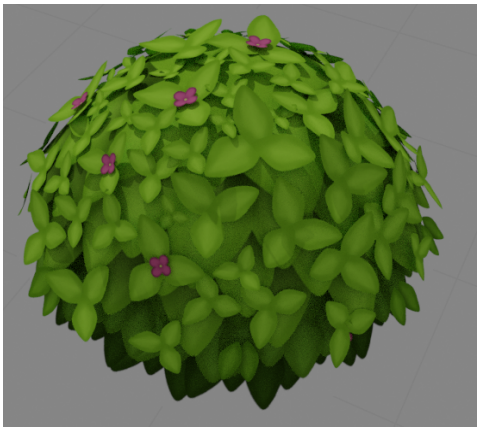


Figura 5.6.2.2.1: Imatge d'un dels arbustos finals del joc

Per generar un arbust, com el de la Figura 5.6.2.2.1, primer necessitarem crear uns recursos fent servir photoshop. Aquests recursos són els que aporten el color i les formes de fulles al modelat, ja que aquest realment serà mitja esfera texturitzada de tal manera que sembli con conjunt de fulles. El recurs s'ha generat a mà replicant la mateixa fulla i aplicant efectes d'ombres per generar la sensació de profunditat com es pot veure en la Figura 5.6.2.2.2.

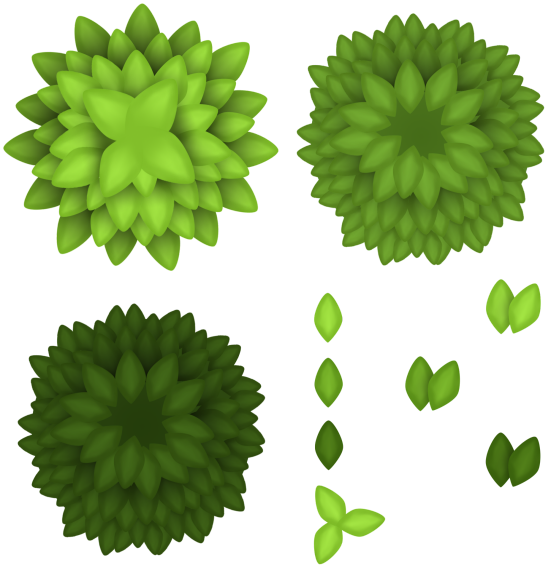


Figura 5.6.2.2.2: Recursos per a la creació de l'arbust

Ara que ja tenim el recurs, tornem a Blender per generar les mitges esferes, fer el unwrap de la malla i aplicar la textura. Crearem mitja esfera per cada conjunt de fulles, com es pot veure en la 5.6.2.2.3.

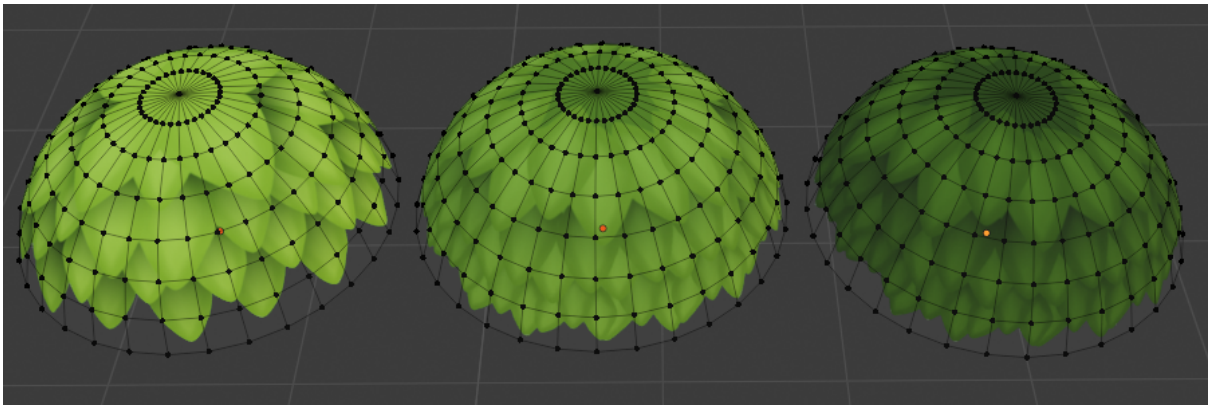


Figura 5.6.2.2.3: Mitges esferes del conjunt de l'arbust

Un cop ho tenim preparat, els posarem un sobre l'altre, deixant el més fosc a la part inferior. Com es pot observar ja tenim una sensació de realisme superior però encara es pot millorar. Per augmentar el nivell de detall, generarem diversos plans, i hi col·loquem les textures de les fulles de la Figura 5.6.2.2.2 per posar-les per sobre del conjunt. Totes aquestes fulles les haurem d'orientar en direcció a les normals de la superfície on les vulguem posar com es pot veure en la Figura 5.6.2.2.4.

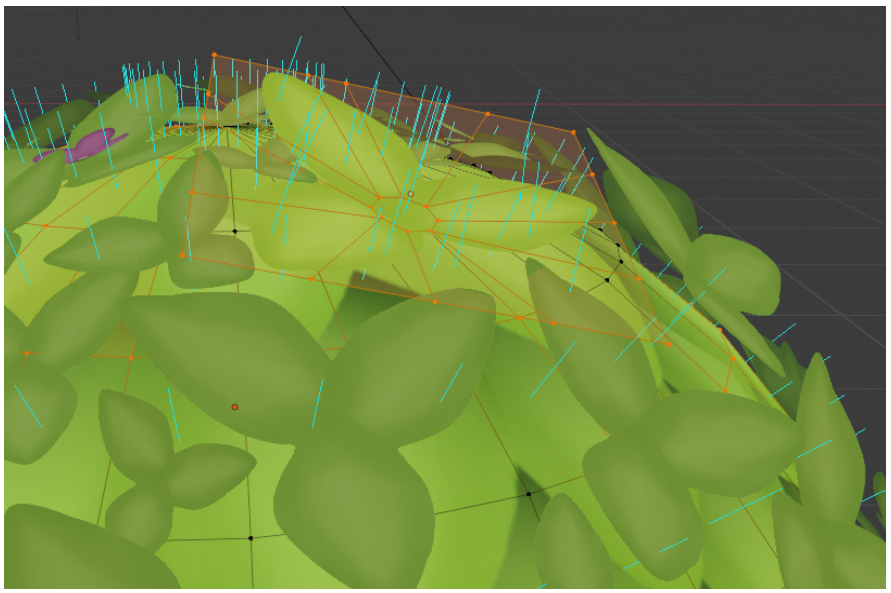


Figura 5.6.2.2.4: Orientació de les fulles de l'arbust

Finalment tindrem el model i ja el podem exportar cap a Unity juntament amb la textura com podem veure en la Figura 5.6.2.2.5.

Figura 5.6.2.2.5: Imatge de l'arbust al Unity

5.6.3. Modelat de roques i minerals

Les roques i minerals son elements de decoració i d'extracció de recursos que podem trobar tant al *Upperworld* com al *Underworld*. Aquests models estan generats a partir d'esferes amb un gran nombre de polígons que després han estat tractades amb una simplificació i finalment uns ajustament finals per obtenir les roques, tal i com les podem veure en la Figura 5.6.3.1.

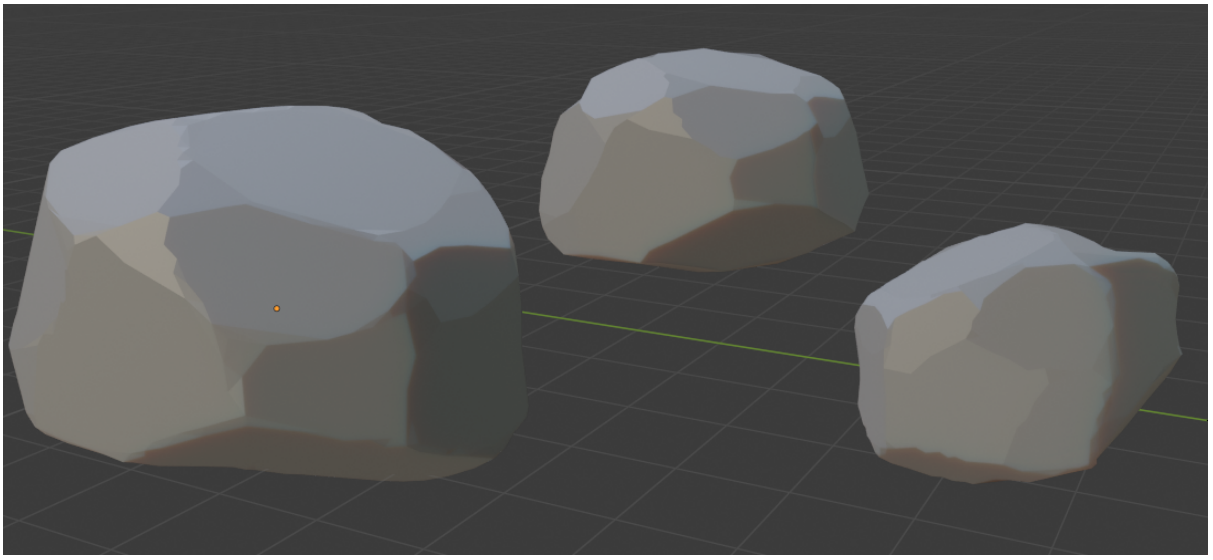


Figura 5.6.3.1: Roques que podem trobar al *Upperworld*

Per altra banda el procés de creació del mineral és una mica diferent ja que hem d'incloure l'element diferenciador que indica que es tracta d'una mena com es pot veure en la Figura 5.6.3.2.

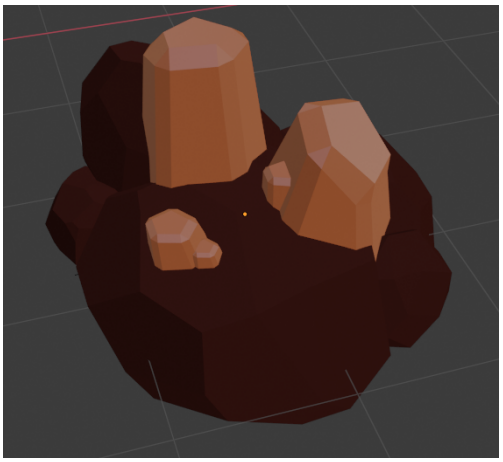


Figura 5.6.3.2: Mineral que podem trobar al *Underworld*

5.6.4. Modelat d'estructures

Al llarg dels escenaris podem veure algunes estructures immòbils com per exemple la passarel·la al *Underworld,* el cromlec i el pont inicial a la segona illa del *Upperworld*. Tots aquests elements formen part de l'escenari i compleixen funcions segons el moment en el que es necesiten.

5.6.4.1. Passarel·la al *Underworld*

La passarel·la és l'element que connecta el *Upperworld* i el *Underworld*. Aquest element cobra una gran importància en el referent al gameplay ja que és el connector entre el món del personatge i els objectius. A l'hora de dissenyar la passarel·la volíem deixar molt clara la direcció. Per això vam decidir deixar un forat a la part central que indiques que servía per baixar, com es pot veure en la Figura 5.6.4.1.1.

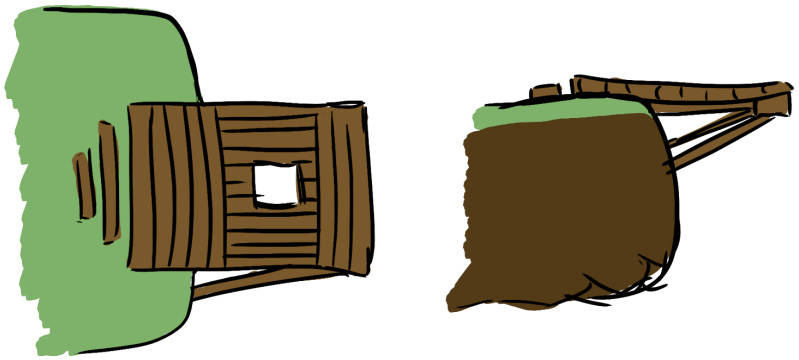


Figura 5.6.4.1.1: Esbós de la passarel·la de fusta

Per elaborar el disseny, primer es va generar un llistó de fusta, el qual més tard replicaríem mitjançant el modificador "Array". Gràcies a el modificador array podem generar estructures, sempre amb patrons repetitius, com es pot veure en la Figura 5.6.1.2.

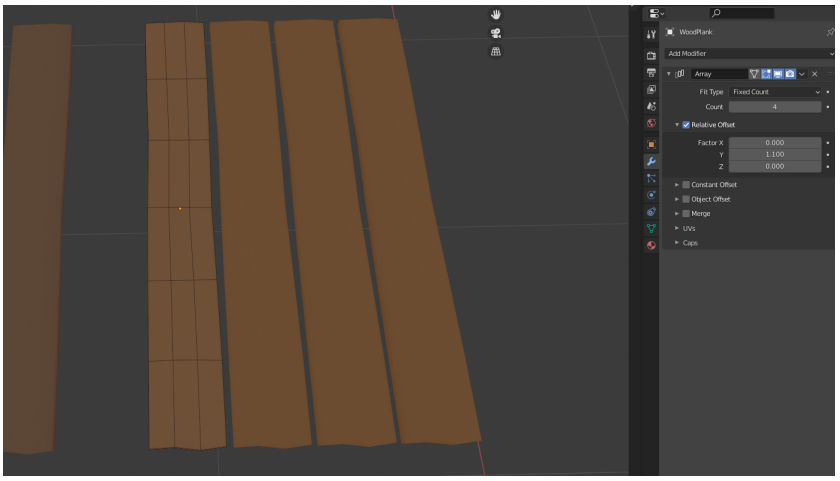


Figura 5.6.4.1.2: Llistó de fusta replicat

Finalment, gràcies al disseny previ i a les petites estructures generades, tindrem llest la plataforma, només quedarà afegir-hi els punts de suport amb l'illa per donar un toc més realista i ja tindrem el model. Un cop finalitzat, li assignarem un material amb un color similar a la fusta i ja podrem realitzar l'exportació. El resultat final es pot veure en la Figura 5.6.4.1.3.

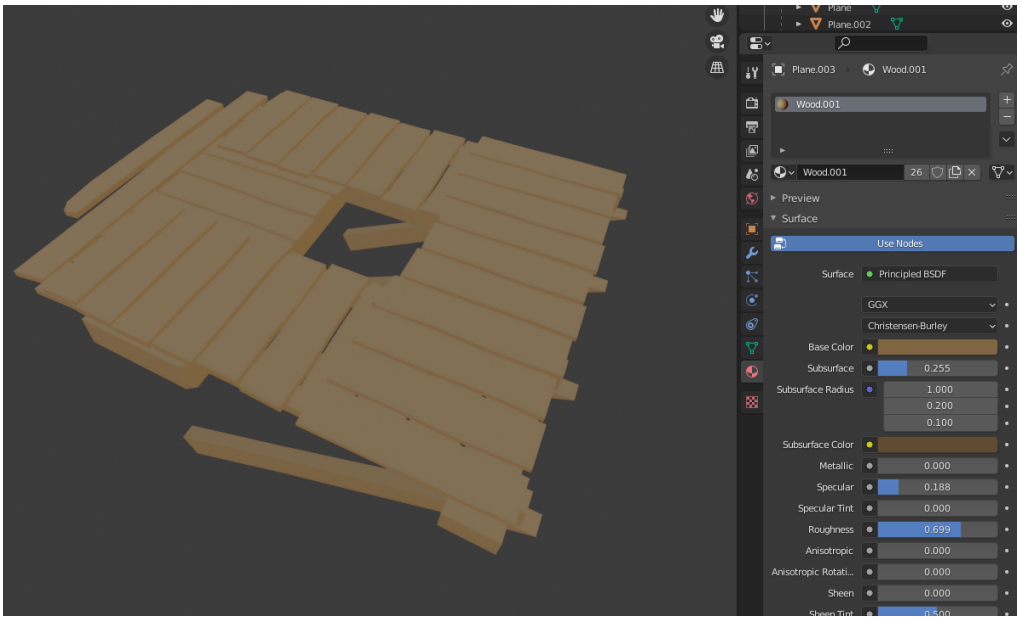


Figura 5.6.4.1.3: Passarel·la de fusta

5.6.4.2. Pont de connexió

El pont de connexió el podem trobar en primera illa del *Underworld*. Aquest pont es troba destruït i serà necessària una reparació per la utilització, com es pot veure en la Figura 5.6.4.2.1. Per reparar-lo candrà fusta i fibra, que trobarem en l'estat natural i els haurem de recol·lectar tant punt començem l'aventura.

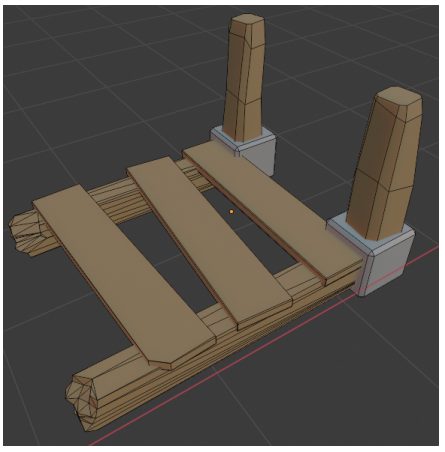


Figura 5.6.4.2.1: Model del pont en el editor de Blender

Per generar aquest model no hem fet servir cap tècnica específica, ja que només es tracta de combinar diferents polígons, modificant-los fins a obtenir les formes desitjades. Un cop ja tinguem tot el model, només caldrà assignar els materials i passar-ho a Unity 3D on més tard li assignarem els scripts encarregats d'afegir la interacció. En la Figura 5.6.4.2.2 podem veure el pont de fusta col·locat al Unity.

Figura 5.6.4.2.2: Pont integrat en la illa del *Upperworld*

5.6.4.3. Farola

La farola és un dels primers objectes que van formar part del joc. Es tracta d'un element púrament estètic que acompanya l'entrada de la passarel·la i que no té cap funcionalitat. Abans de generar-lo es van realitzar algunes proves per tal de decidir el disseny, com es pot veure en la Figura 5.6.4.3.1, ja que aquest podía influir en la resta d'elements.

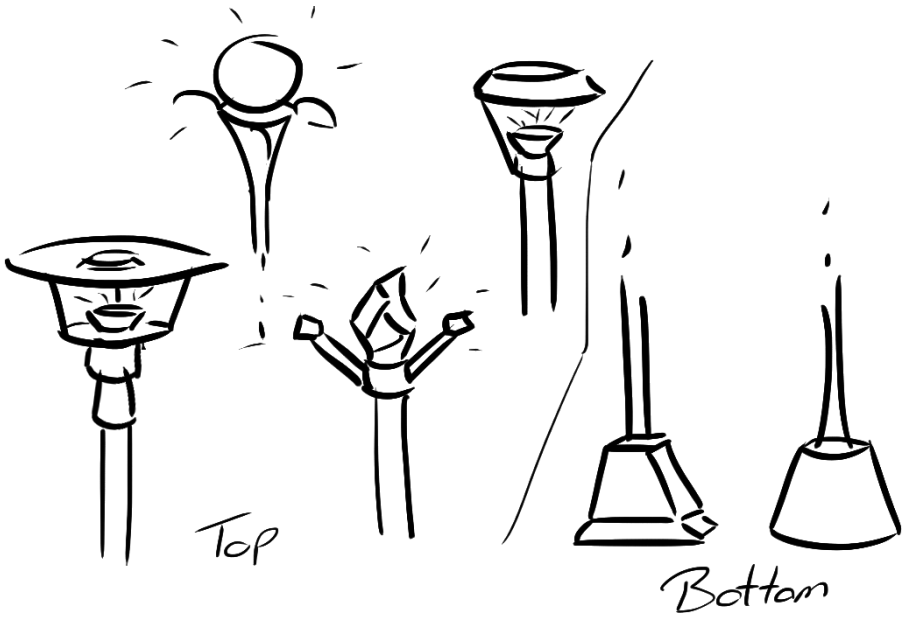


Figura 5.6.4.3.1: Esbós de la farola

La creació d'aquest model no incorpora cap tècnica complexa, ja que a partir d'un polígon, i amb la utilització de extrusions, translacions i escales, podem arribar a obtenir el model de la figura 5.6.4.3.2.

Figura 5.6.4.3.2: A l'esquerra el model de la farola en el editor i a la dreta el resultat a Unity 3D

5.6.4.4. Cromlec

Com un dels elements protagonistes del *Underworld* trobem el cromlec, un monument megalític de milers d'anys d'antiguitat. En aquest cas forma part de la decoració i té la funció d'aportar misticisme a l'escena.Durant el procés de creació de l'illa, el cromlec no va ser la primera opció, ja que el primer element que vam decidir colocar abans va ser un altre edificació antiga, un tolo grec, com es pot veure a la Figura 5.6.4.4.1.

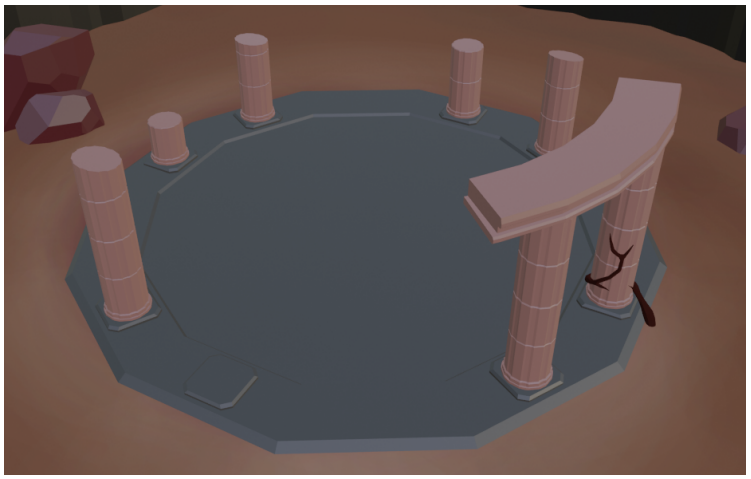


Figura 5.6.4.4.1: Model de tolo grec descartat durant el desenvolupament

Finalment es va canviar d'idea perquè el fet de posar el tolo orientaria el projecte a una era concreta, i per ara aquest element no tindria un objectiu clar. Un cop descartat, ens vam decantar per un element més atemporal com el cromlec que podem veure a la Figura 5.6.4.4.2.

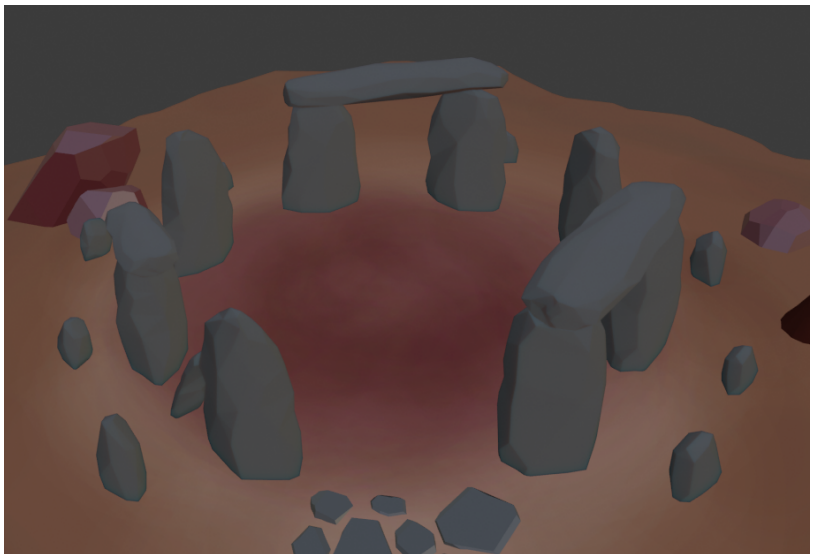


Figura 5.6.4.4.2: Modelat del cromlec fet servir en el *Underworld*.

Com es pot veure en la Figura 5.6.4.4.2 vam complementar el seu aspecte amb la textura de l'illa per tal d'afegir un efecte encara més misteriós, que combinat amb el joc de llums a l'escena de Unity dona com a resultat la Figura 5.6.4.4.3.

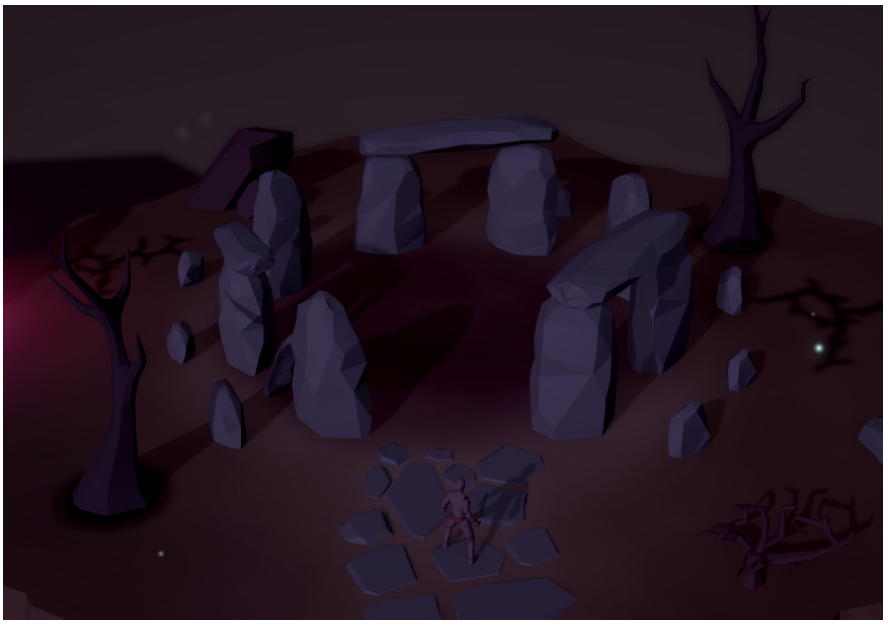


Figura 5.6.4.4.3: Escena *Underworld* a Unity amb el cromlec i altres elements decoratius

5.6.4.5. Portal

El portal és l'eina que ens endinsa en les profunditats del *Underworld* on es troba l'objectiu principal. Aquest portal és místic i el lloc on ens portarà és totalment desconegut, però un cop el fem servir no podrem tornar enrere.

A l'hora de dissenyar el portal hem fet servir una referència directa a l'estil dels portals que podem trobar a "Hellblade: Senua's Sacrifice", com es pot veure en la Figura 5.2.3.3 de la Secció 5.2.3.

El procés de modelatge ha estat clarament influït per l'experiència obtinguda de la creació dels arbres, ja que, per la formació de l'arc i les branques que l'envolten, s'ha fet utilitzant el modificador "Skin" sobre un seguit de eixos i vèrtex, tal i com es pot veure en la Figura 5.6.4.5.1.

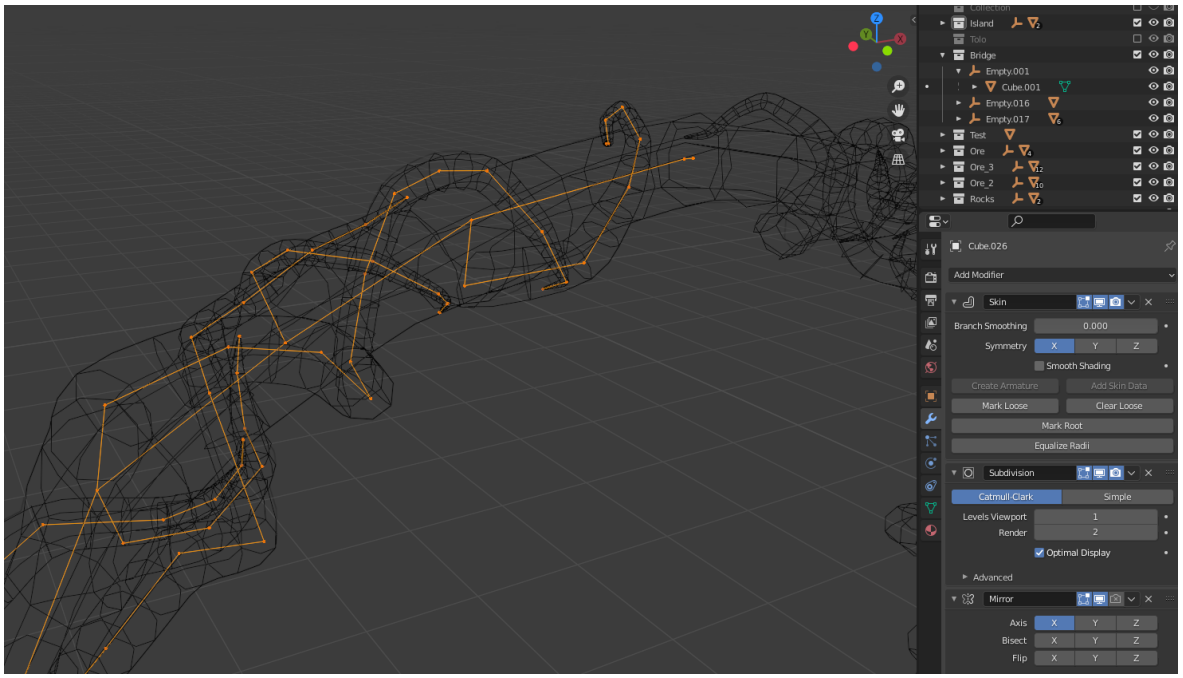


Figura 5.6.4.5.1: Captura del procés de creació de l'arc del portal

Un cop completat l'arc, vam necessitar afegir un polígon que cobrís la part interior per després, a Unity, afegir un material complementat per un *shader* (veure Apartat 5.12.7). Finalment, al Unity, s'ha complementat el model amb un punt de llum adjacent a l'entrada per il·luminar els objectes del voltant. El resultat es pot veure a la Figura 5.6.4.5.2.

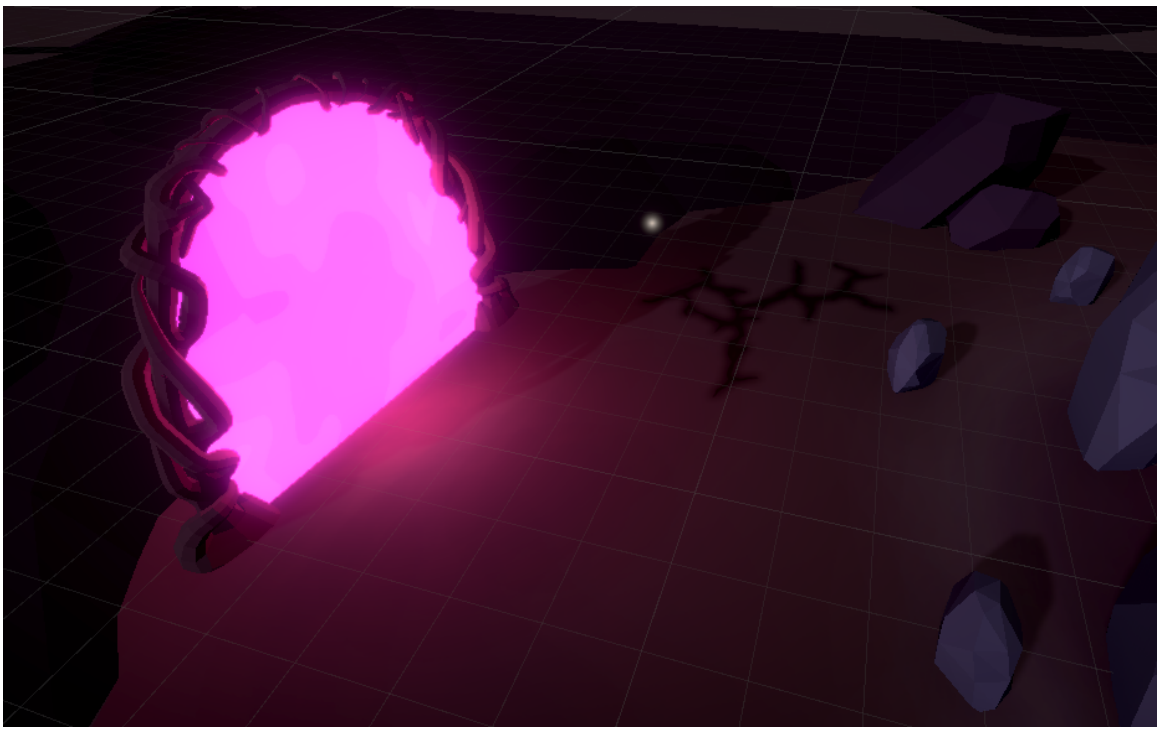


Figura 5.6.4.5.2: Resultat de la implementació del portal al joc

5.6.4.6. Altar de tornada

Un cop ens endinsem en les profunditats del *Underworld,* no existeix un altra manera de tornar que no sigui per l'altar de tornar. En aquest altar es realitza un ritual que permet al jugador tornar a les portes del *Underworld*. Tal i com s'esmenta, es realitza un ritual, pel que les referències a l'hora de dissenyar-lo van des de pel·lícules a il·lustracions, típicament de cultures antigues, com es pot veure a la Figura 5.6.4.6.1.

Figura 5.6.4.6.1: Altar de l'illa del sol a Bolivia

Pel que fa al model final, cal destacar la formació de les cadenes que connecten els pilars laterals. Per generar les cadenes s'ha fet servir el modificador de "Array" que permet duplicar un objecte. Un cop tenim la longitud desitjada, i mitjançant un camí guia, podem aplicar un modificador de seguiment de camins per col·locar-la de pilar a pilar, donant-li la forma desitjada. Un cop col·locat, podem veure el resultat a la Figura 5.6.4.6.2.

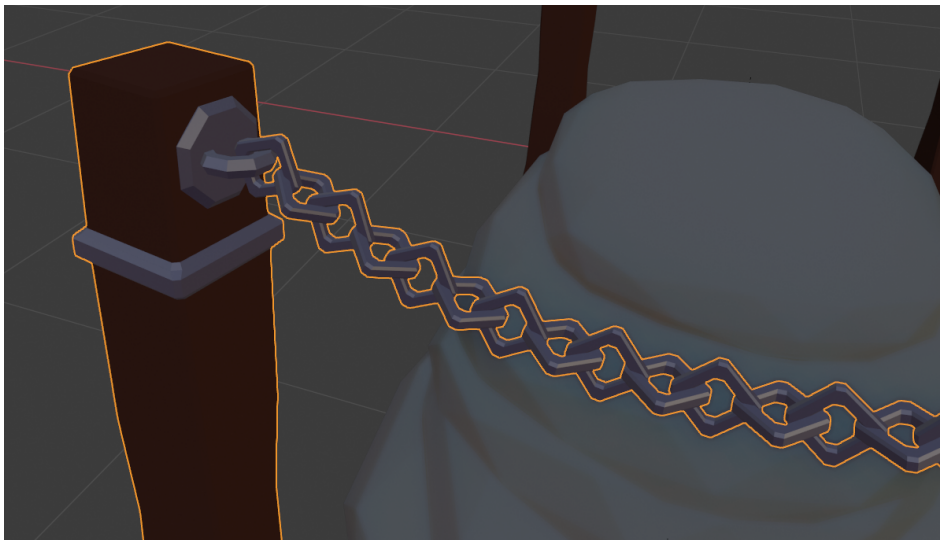


Figura 5.6.4.6.2: Resultat del modelatge de la cadena

Finalment al exportar el model a Unity, hem volgut complementar l'altar afegint una flama flotant utilitzant un shader (veure Apartat 5.12.7) tal i com es pot veure a la figura 5.6.4.6.3.

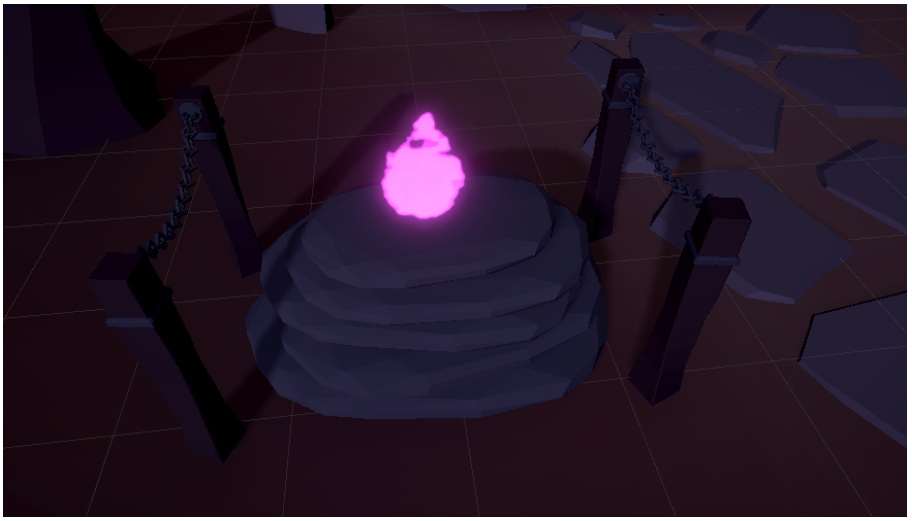


Figura 5.6.4.6.3: Resultat final de l'altar al joc.

5.6.4.7 Casa

La casa és l'element central del joc que ens permet evolucionar tecnològicament. Amb l'adquisició d'elements bàsics, serem capaços de millorar-la per poder accedir a nous objectes i millores que es faran servir per progressar en el joc. durant el disseny es van realitzar alguns esbossos per determinar l'estil de les possibles fases, com els de la Figura 5.6.4.7.1, que finalment han esdevingut en els models funcionals que trobem a l'escena del *Upperworld.*

Figura 5.6.4.7.1: Primers dissenys de les cases

En quant al modelat de les cases, hem utilitzat les tècniques bàsiques de redimensió, translació i escalat per generar tots els models. Al realitzar els models hem tingut molt en compte la quantitat de polígons, ja que encara que el target siguin ordinadors, cal realitzar petites millores en quant a reducció de geometria per aconseguir un rendiment superior i arribar a màquines menys potents. Com es pot veure a la figura 5.6.4.7.2, la tenda es tracta d'un model senzill, dotat de diferents materials que li aporten els colors.

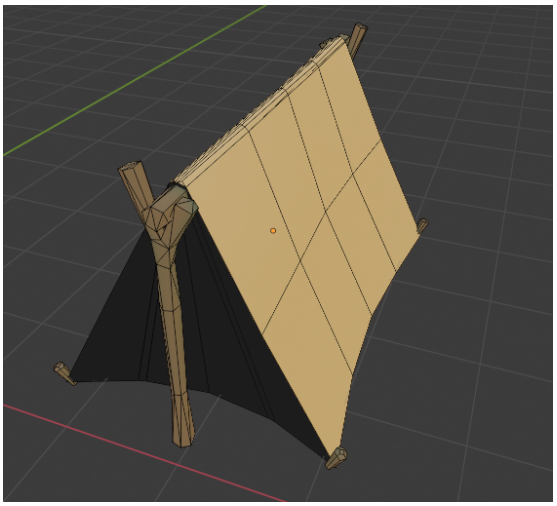


Figura 5.6.4.7.2: Model 3D de la tenda en el editor de Blender

De la mateixa manera que la tenda, la segona de les fases, la de fusta, també ha estat modelada seguint el mateix patró d'optimització. L'element que la diferencía de la primera fase és l'augment del detall, ja que s'ha treballat més en quant a la textura, tal i com es pot veure a la Figura 5.6.4.7.3.

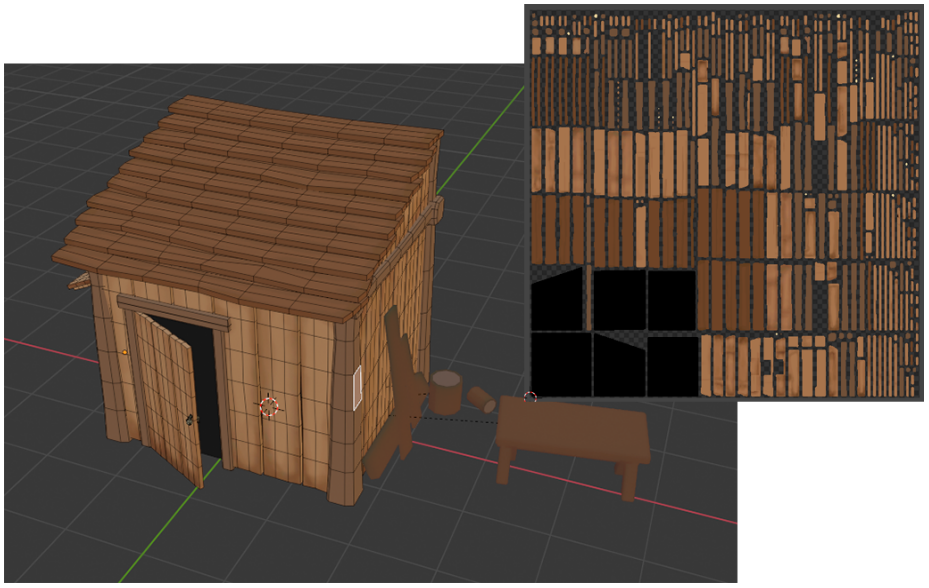


Figura 5.6.4.7.3: Modelat de la segona fase amb el mapa de textura utilitzat.

Amb tot això, exportarem els models a Unity i hi aplicarem tots els scripts necessaris per afegir la lògica que permeten realitzar la millora de les cases i el desbloqueig d'elements. Podem veure els models en escena en la Figura 5.6.4.7.4 i 5.6.4.7.5.

Figura 5.6.4.7.4: Fase 1 de la casa en el joc.

Figura 5.6.4.7.5: Fase 2 de la casa en el joc.

5.7 Interfícies

La interfície és un element a tenir molt en compte en aquest tipus de jocs, ja que el necessitarem per indicar al jugador les accions a realitzar o els objectes a construir. En aquest cas podem diferenciar diferents nivells d'interfície segons l'ús.

5.7.1. Menú principal

En aquest cas hem decidit elaborar un menú principal que representa el nucli de la jugabilitat i l'espai on el jugador haurà de dedicar un gran esforç per poder superar el joc, parlem del *Underworld*. Com es pot veure en la Figura 5.6.1.1, hem aprofitat un element del joc per integrar les diferents opcions inicials.

Figura 5.7.1.1: Menú principal

- **Continue:** Ens permetrà reanudar el joc en el últim punt on vam desar la partida.
- **New Game:** Ens permetrà començar una partida des de l'inici.
- **Exit:** Opció que servirà per tancar el joc.

5.7.2. Interfície de l'estat del jugador

Per tal de representar totes les accions que pot realitzar el jugador, cal dissenyar una interfície que ens permet, de manera senzilla, conèixer les possibilitats que ofereix una zona o objecte interactuable. En la Figura 5.7.2.1 es pot veure una captura de la interfície.

Figura 5.7.2.1: Captura de Upperworld amb interfície

En aquest cas existeixen diversos panells per indicar informació de l'usuari o de l'estat de la partida. El primer element d'interfície del joc que podem veure és el de la informació de l'usuari, que el podem veure a la part superior esquerre de la pantalla. En aquest panell es mostrarà la vida i l'energía com es pot veure en la Figura 5.7.2.2.

Figura 5.7.2.2: Barra de vida i energia del jugador

Com a segon protagonista trobem el panell de col·locació d'elements, que serà l'encarregat de permetre al jugador interactuar amb el món i col·locar objectes en la illa. Aquest panell tindrà dos estats:

Amagat: Representació visual. Aquest element de la interfície només es mostrarà quan sigui possible col·locar coses al món. En la Figura 5.7.2.3 podem veure aquest element.

Figura 5.7.2.3: Panell de col·locació amagat

Obert: Moment en el que el jugador podrà seleccionar l'element a posar a la illa. Un cop seleccionem l'element a posar, el panell s'amagarà per centrar-se en l'acció de col·locar i donar més visió al jugador.En la Figura 5.7.2.4 podem veure aquest element.

Figura 5.7.2.4: Panell de col·locació obert

El tercer dels panells que podem veure en pantalla es tracta de l'espai d'inventari que porta el jugador a sobre. Aquest inventari és limitat i serà necessari buidar el objectes que portem a la casa per poder seguir recollint-ne. En la Figura 5.7.2.5 podem veure aquest element.

Figura 5.7.2.5: Panell d'inventari

5.7.3. Interfícies d'interacció

Ara que ja tenim els elements bàsics d'interfície que ens trobem en iniciar el joc, seguirem amb aquells que farà servir el jugador per interactuar amb el món. Aquests seran panells que permetin realitzar accions més específiques.

En primer lloc parlarem de la interfície que ens permet interactuar amb el món. Aquesta interfície està dirigida a indicar a l'usuari l'acció que realitzarà, el botó d'acció i si existeix algun requeriment.

A continuació podem veure alguns exemples a les Figures 5.7.3.1, 5.7.3.2, 5.7.3.2 i 5.7.3.4:

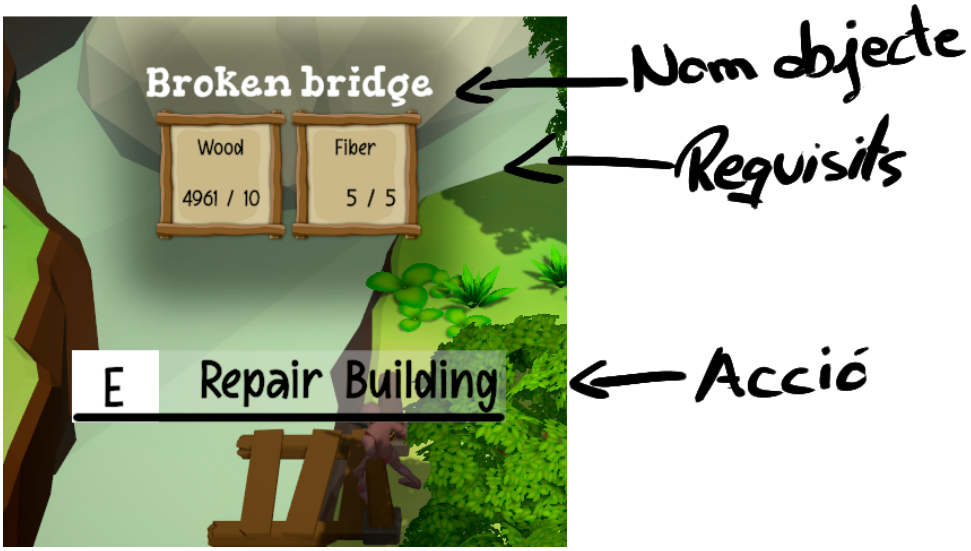


Figura 5.7.3.1: Panell d'interacció amb el pont de fusta

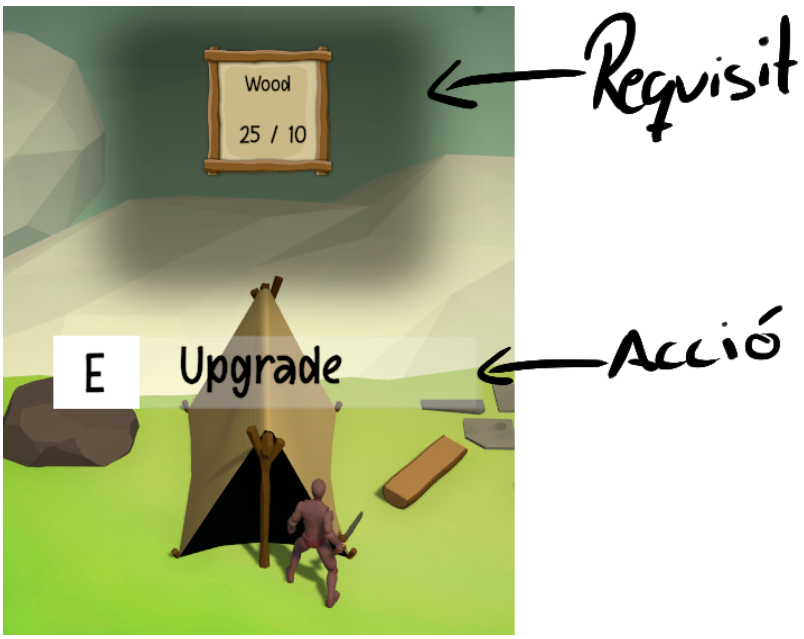


Figura 5.7.3.2: Panell d'interacció amb la tenda

Figura 5.7.3.3: Panell d'interacció amb el test

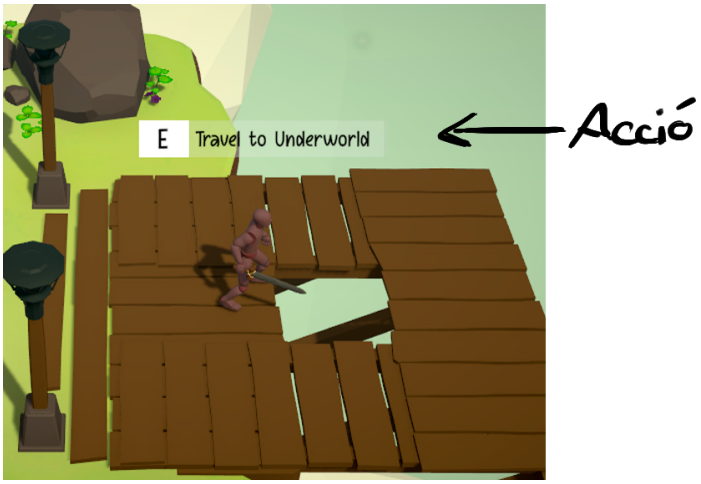


Figura 5.7.3.4: Panell d'interacció amb passarel·la de fusta

Com es pot observar, aquests panells d'interacció poden variar segons l'objecte al que ens apropem.Per altra banda, trobem panells que permeten realitzar accions com la creació d'elements. En aquests panells trobem un llistat d'elements a crear, els requisits, una imatge identificativa i el selector de nivell d'objecte. Podem veure aquest panell en la Figura 5.7.3.5.

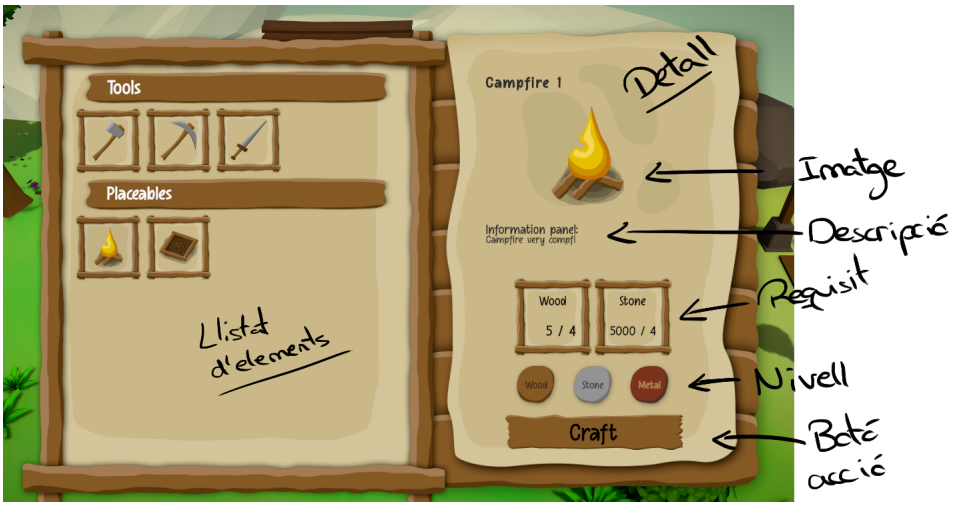


Figura 5.7.3.5: Panell de creació

5.7.4. Menú de pausa

Quan el jugador prem el botó de pausa, s'enfosca tota la pantalla i s'obre el menú de pausa en el que hi trobem 4 opcions:

- Continuar: Ocultar el menú de pausa i seguir jugant
- Guardar: Guardar l'estat actual de la partida
- Carregar: Carregar l'estat de l'última partida guardada
- Menú: Tornar a l'escena de menú principal

Es pot veure una imatge del menú en la Figura 5.7.4.

Figura 5.7.3.5: Menú de pausa

5.7.5. Font

La font que hem fet servir per tot el projecte s'anomena "Scratch Boys". Creiem que aquesta font encaixava amb tota la resta d'elements visuals gràcies a la seva irregularitat i les seves formes arrodonides, que aporten una sensació més dócil envers l'entorn en la que ens la trobarem majoritàriament, el *Upperworld*. En la Figura 5.7.4 podem veure el títol del joc amb la font.

Figura 5.7.5: Títol del joc amb la font Scratch Boys

5.8 Disseny d'objectes

La incorporació de diferents objectes i la capacitat de crear-los mitjançant altres materials és una de les característiques principals del joc. Amb l'objectiu de complir amb les mecàniques bàsiques que ens vam proposar en el planning, s'han dissenyat elements que, principalment, permetin generar recursos.

5.8.1 Test

El test es l'element del joc que ens permet cultivar diferents plantes que més tard farem servir per obtenir millores amb un altre objecte. Tal i com es pot veure a la Figura 5.8.1.1, el disseny del test és quadrat, ja que la intenció darrere la forma és la possibilitat de situar-los un al costat de l'altre per generar una quadrícula.

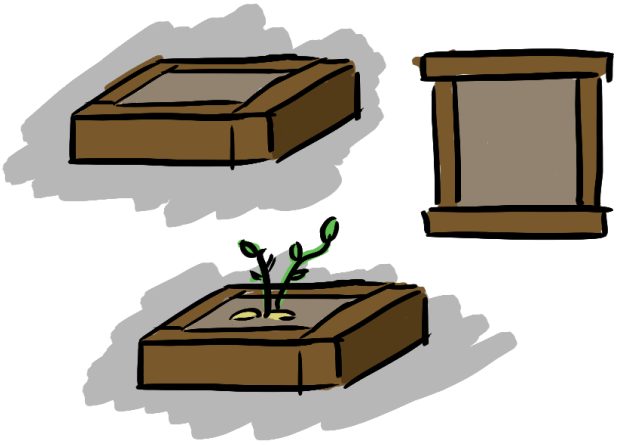


Figura 5.8.1.1: Disseny del test

A l'hora de modelar-lo hem tingut en compte els elements que després hi situaríem a sobre, per tant necessitàvem prou espai per afegir un segon model, respectant la forma i els marges per no superposar elements. En la Figura 5.8.1.2 podem veure el modelat final del test.

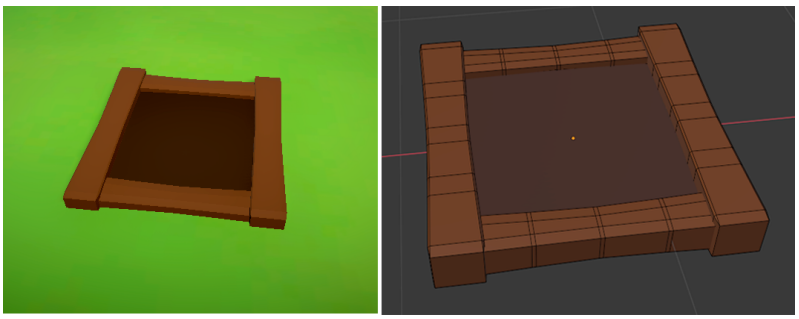


Figura 5.8.1.2: Model final del test col·locat en el joc a l'esquerra i el model 3d en el editor a la dreta

5.8.2 Plantes

Tot seguit del test, cal destacar les plantes, ja que sense explicar prèviament el test, no podrem entendre el seu disseny. A l'hora de dissenyar els tres tipus de plantes, vam triar fer-les de diferents formes per afegir varietat visual al joc. Els primer esbossos els podem veure en la figura 5.8.2.1.

Figura 5.8.2.1: Esbós de les plantes.

Abans de passar amb les plantes, cal esmentar que aquestes tenen fases de creixement, i tot i que la representació visual és la mateixa, la final correspondrà a la llavor plantada. Podem observar les primeres dues fases en la Figura 5.8.2.2.

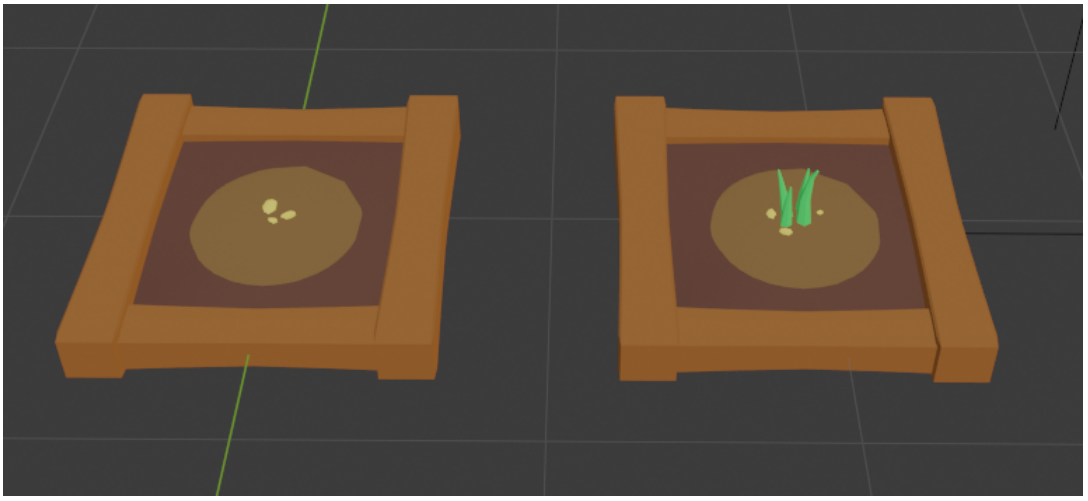


Figura 5.8.2.2: Modelats de les fases de creixement de les plantes.

La primera de les plantes representa el que generalment coneixem con un arbust de fruits. Aquesta planta té la peculiaritat de generar diverses collites abans de degradar-se, de manera que el jugador por beneficiar-se de de la seva última etapa diverses vegades. Podem veure el resultat final a la Figura 5.8.2.3.

Figura 5.8.2.3: Model de la planta de fruits al joc.

La segona de les plantes cobreix la família dels tubercles. En aquest cas serà una única collita i el seu aspecte el podem veure en la Figura 5.8.2.4.

Figura 5.8.2.4: Planta de la família dels tubercles en el joc.

Finalment, la última agafa el disseny dels típics camps de cultiu de panotxes. Aquesta serà de única collita i el model resultant el podem veure en la Figura 5.8.2.5.

Figura 5.8.2.5: Planta panotxa en el joc.

5.8.3 Cuina

Ara que coneixem el test i les plantes que hi podrem plantar, és hora veure la cuina. La cuina és un objecte que es pot col·locar en l'illa i que té la funció de preparar plats per aconseguir bonificacions en les estadístiques. Les cuines consten de tres fases, cadascuna d'elles les desbloquejarem a mesura que evolucionem la casa. Podem veure un esbós de cadascuna de les cuines en la Figura 5.8.3.1.

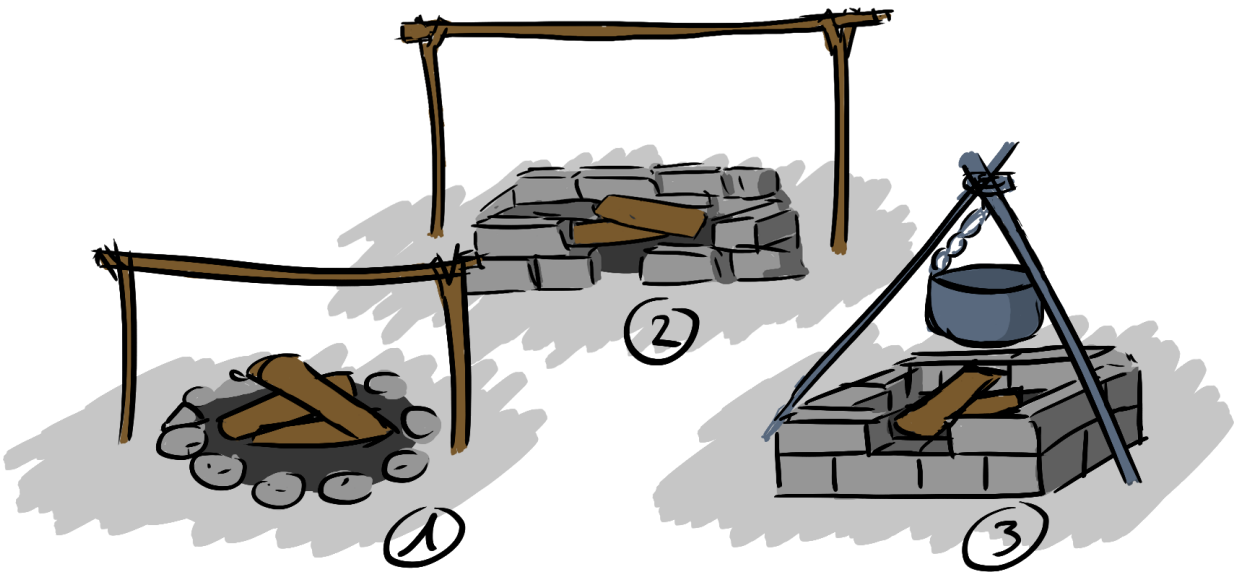


Figura 5.8.3.1: Esbós de les fases de la cuina.

En quant al modelat, no ha resultat complex, ja que gràcies al modificador “Array”, podem generar diversos maons per tal de fer les cuines 2 i 3. Cadascuna de les cuines està pensada i fabricada dels materials característics de cada etapa. A continuació podrem veure en les figures 5.8.3.2, 5.8.3.3 i 5.8.3.4 el resultat del modelatge de les cuines i com han quedat un cop texturitzades.

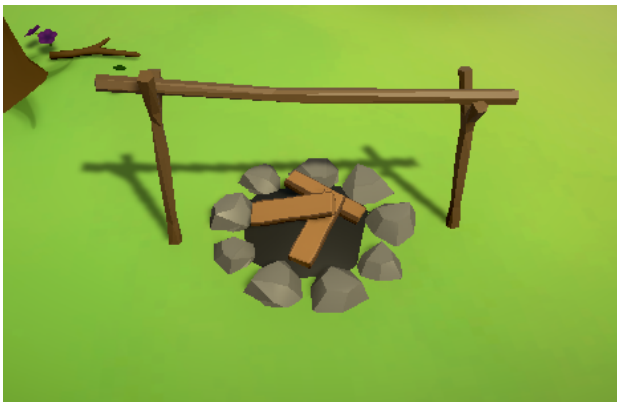


Figura 5.8.3.2: Cuina de nivell 1. Imatge del joc.

Figura 5.8.3.3: Cuina de nivell 2. Imatge del joc.

Figura 5.8.3.4: Cuina de nivell 3. Imatge del joc.

5.8.4 Eines

Amb la necessitat d'extreure recursos i combatre les hostilitats, hem dissenyat un conjunt d'eines bàsiques per cobrir aquestes accions. La primera de les eines que podem veure a la Figura 5.8.4.1 es tracta del pic. El pic el farem servir per extreure fragments de roca o minerals que puguem trobar en el món.

Figura 5.8.4.1: Modelat final del pic en el editor de Blender.

La segona de les eines, i que farem servir per l'etapa inicial del joc, és la destral. A la Figura 5.8.4.2 podem veure el resultat del modelatge.

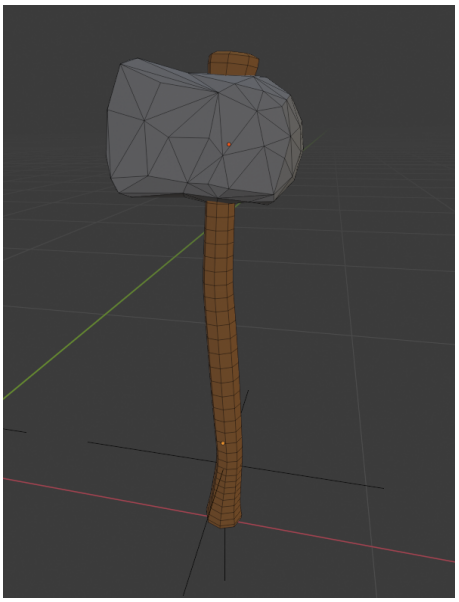


Figura 5.8.4.2: Modelatr final de la destral en el editor de Blender.

Per últim, i com a element defensiu, trobem l'espasa. El primer disseny de l'espasa és més rudimentari i per tant la fulla no és gaire refinada. Podem veure el modelat final a la Figura 5.8.4.3.

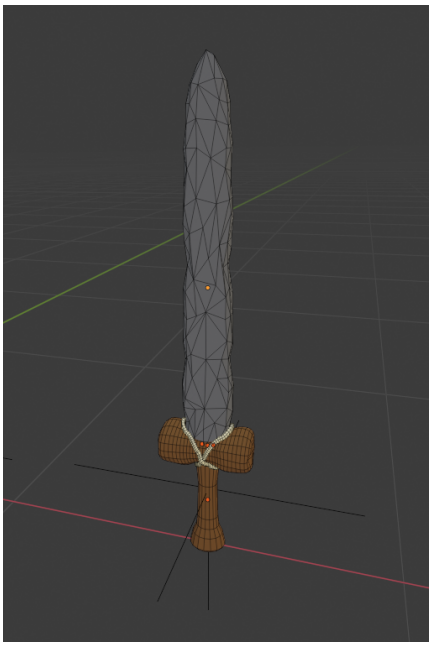


Figura 5.8.4.3: Modelat final de l'espasa en el editor de Blender.

5.9 Producció externa

Com a producció externa trobem l'element musical, ja que és l'únic element que no hem produit nosaltres. La producció musical l'ha portat a terme el germà d'un de nosaltres, en Josep Saavedra, també estudiant del grau en disseny i desenvolupament de videojocs, que té com a hobby la creació de peces musicals, normalment enfocades i altament inspirades en el món dels videojocs. Per a poder produir les peces, li vam proporcionar imatges del joc per a que tingués una referència de la cual partir. A més, vam especificar-li diferents sensacions que volíem transmetre amb les diferents peces musicals, com per exemple melodies més calmades i alegres per al UpperWorld o peces més misterioses i ténebres per al UnderWorld. D'aquesta manera, hem pogut afegir encara més personalitat a cada una de les escenes i donar sensacions molt diverses durant tot el joc.

5.10. L'entorn de Unity i disseny del projecte

Durant el període d'investigació i recerca, hem pogut observar que desenvolupar un joc en Unity es pot assolir de moltes maneres. Cada equip o empresa que fa servir aquest motor té aproximacions molt diverses per al que fa a estructurar un projecte o com utilitzar les eines que proporciona el motor. En aquest apartat descriurem les nostres decisions i la manera en la que hem desenvolupat el projecte.

5.10.1. Estructura del projecte

Per a estructurar el projecte sempre hem tingut present que sigui una estructura modular i fàcil d'expandir o modificar, ja que ens hem enfocat més en la creació d'un prototip per a un joc ambiciós que serveix com a base per a una posterior expansió. Per a fer-ho, hem dividit les diferents part més importants i amb més pes del joc en escenes. Es podria dir que cada escena és, i gestiona, una part d'un gran sistema complex, composat per la suma de nombroses escenes que es comuniquen entre elles. D'aquesta manera, aconseguim una modularitat molt gran, ja que cada escena pot existir en el joc per si sola. En l'Apartat 6.2.1 es poden veure aquestes escenes amb el seu funcionament.

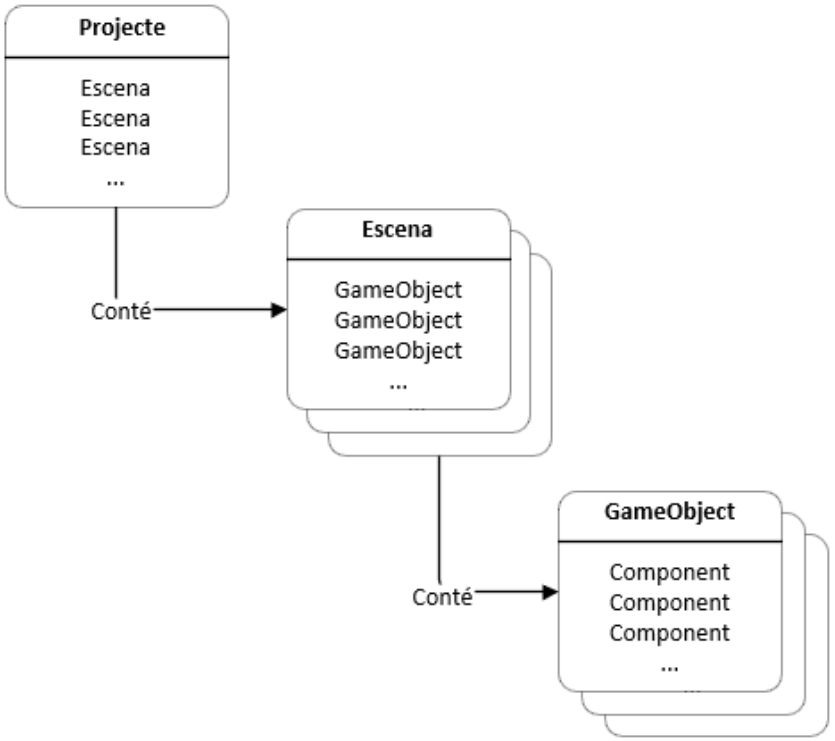


Figura 5.10.1: Estructura del projecte

Com es pot veure en la Figura 5.10.1, el projecte està format per escenes i cada escena té GameObjects que estan formats per components. Són aquests components els que creen els comportaments i sistemes de cada escena.

5.10.1.1 Escenes

Com hem comentat, cada escena del projecte està formada per GameObjects.

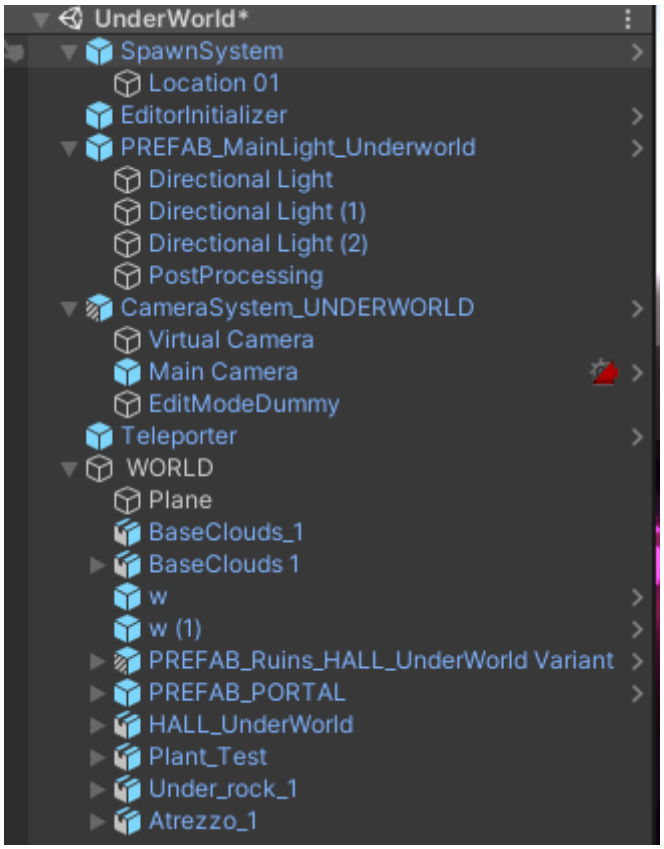


Figura 5.10.1..11: GameObjects de l'escena UnderWorld

Com es pot veure en la Figura 5.10.1.1.1, l'escena UnderWorld està formada per un llistat de GameObjects. Aquests objectes poden ser decoració (com Plant_Test o Uner_rock_1) o poden gestionar una funció en concret (com SpawnSystem, encarregat de gestionar la instanciació del jugador o CameraSystem_UNDERWORLD per a gestionar la càmera).

5.10.1.2 GameObjects

Els GameObjects són la clase bàsica per a totes les entitats del Unity. Per si sols no poden fer res. El seu comportament apareix a partir dels diferents components que els hi afegim, tant components propis del Unity com classes fetes per nosaltres.

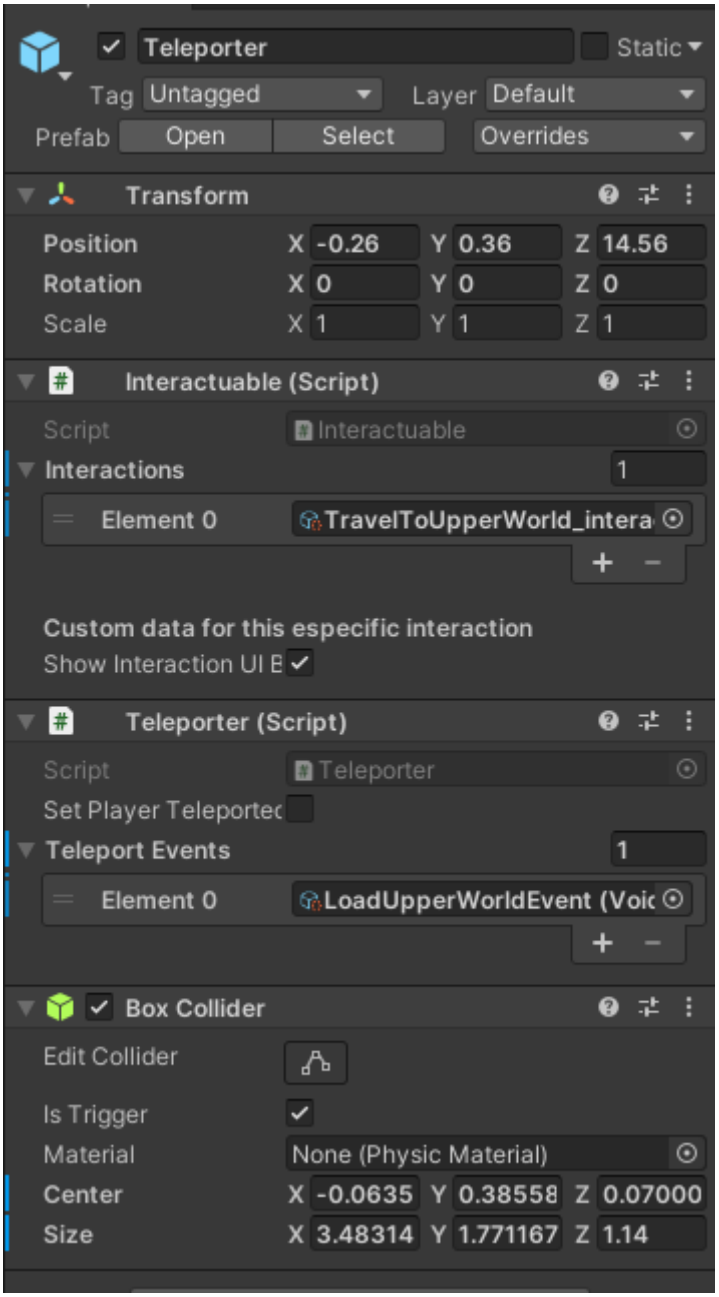


Figura 5.10.1.1.1..2: Components de l'objecte Teleport

Com es pot veure en la Figura 5.10.1.1.1.2, els components de l'objecte Teleport són els que afegeixen la lògica a l'objecte. Per exemple, amb el component Interactuable podem identificar que el jugador podrà interactuar amb l'objecte i alhora definir les possibles interaccions.

5.10.2. El motor de física

El motor de físiques integrat en Unity proporcionen components que permeten gestionar la simulació física d'acceleracions, col·lisions o gravetat. Depenent del projecte es pot utilitzar un motor 2D o un 3D.

5.10.2.1 Aplicació en el joc

Hem utilitzat el motor de física 3D, ja que el joc transcorre en un món en 3 dimensions. En el projecte, la física no és un punt gaire important, tots els objectes tenen un comportament molt estàtic. On sí que hem hagut de fer servir el motor de física ha estat en els moments on necessitem saber quan un objecte colisiona amb un altre o fer que el jugador no pogués travessar certs objectes. Per fer-ho hem utilitzat els components Collider amb diferents formes, com cubs, esferes, cilindres... En la Figura 5.10.2.1.1 podem veure un exemple de component Collider.

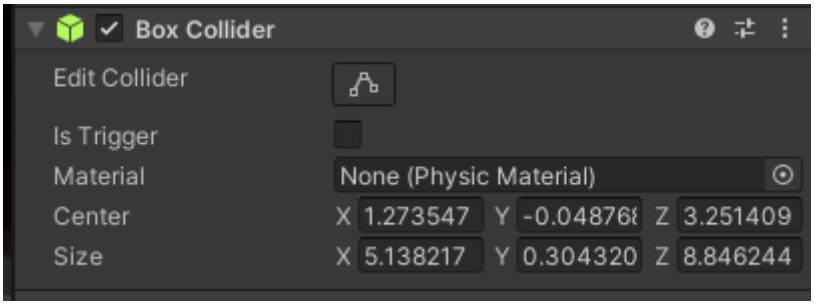


Figura 5.10.2.1.1: Component Box Collider

5.10.3. Dispositius d'entrada i controls

En aquests moments el joc està pensat per jugar-se amb teclat i ratolí, tot i que la base del sistema de dispositius d'entrada i controls està pensada per a poder utilitzar altres tipus de dispositius. En el Apartat 6.3 es descriu amb més detall la gestió i input del jugador.

5.10.5. Renderitzat

Per al renderitzat, hem fet servir el canal de renderitzat universal de Unity (Universal Render Pipeline). El canal de renderitzat universal és una solució eficient per a jocs amb gràfics d'un cert nivell de detall i que alhora busquin un rendiment i velocitat molt bons en tots els dispositius disponibles, ja sigui mòbils, ordinadors o consoles.

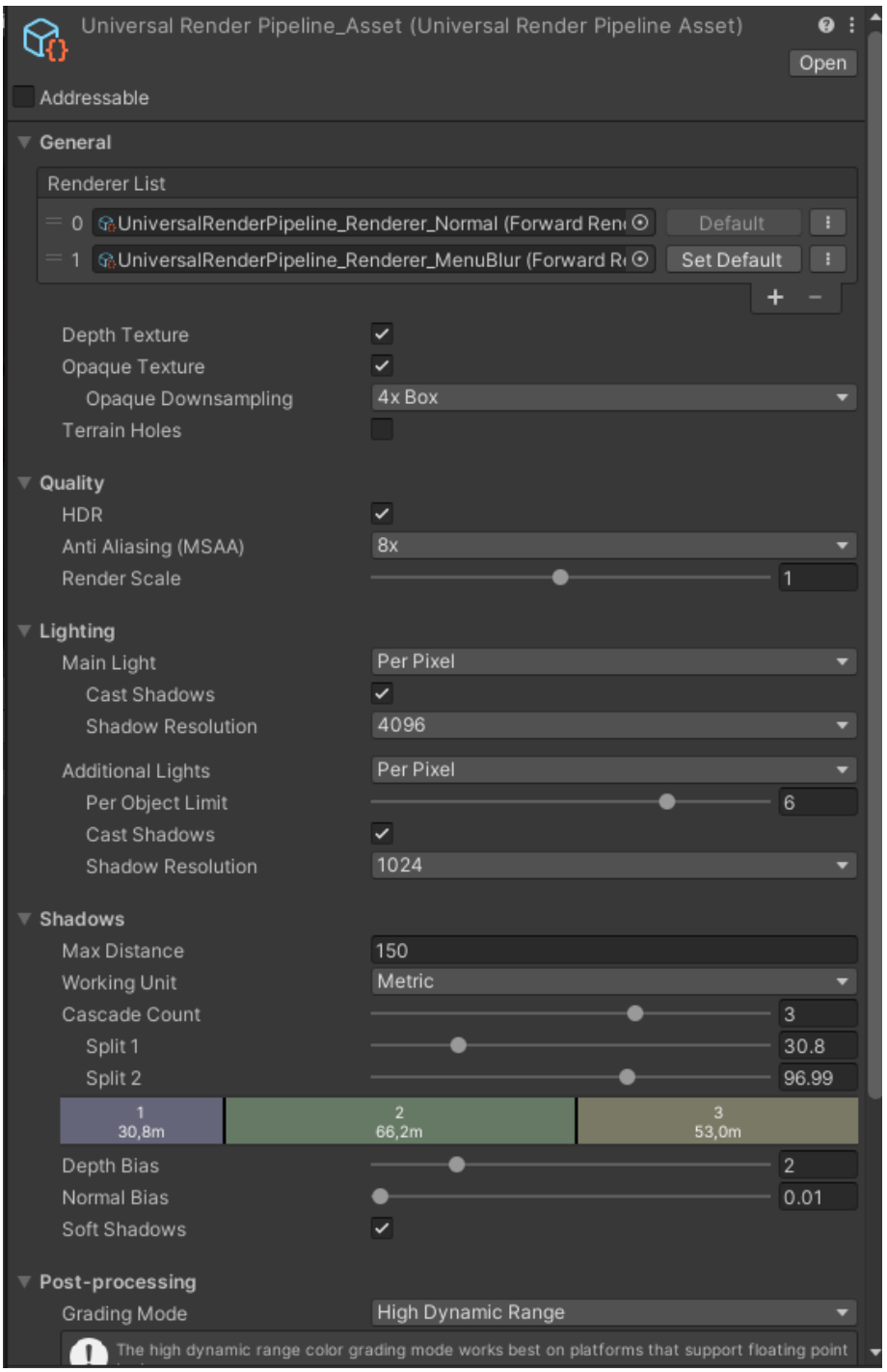


Figura 5.10.5.1: Opcions del canal universal de renderitzat

Com es pot veure en la Figura 5.10.5.1, el canal universal de renderitzat de Unity ens ofereix moltes opcions per a controlar el renderitzat, com ara les llums, ombres, la qualitat o altres atributs de post processat.

5.10.6. Importació dels recursos

La importació de recursos en el nostre joc ha sigut de 3 tipus:

- FBX:
- Imatges: les imatges s'han guardat totes com a .png i s'ha tingut cura de les imatges amb transparències.
- Audio: les pistes d'àudio s'ha procurat que siguin en format .wav, ja que és el format que Unity accepta.

5.11. Elements de feedback

5.11.1. Barra de vida i energia

Per a mostrar l'estat actual de la vida del jugador i la seva energia, hem fet servir barres. Aquestes barres s'encarreguen de proporcionar en tot moment la vida actual i l'energia del jugador, com es pot veure en la Figura 5.11.1.1.

Figura 5.10.5.1: Barres de vida i energia del jugador completes

Quan el jugador perd vida o energia, aquestes barres s'apaguen en proporció a la vida o energia que el jugador hagi perdut. Cal destacar que la vida no es recupera sola, en canvi l'energia sí. Per a que el jugador pugui veure el temps per a que un punt d'energia es recuperi, s'anirà omplint el següent punt d'energia fins que estigui complert. En aquest moment el jugador podrà veure que ha obtingut un punt d'energia. En la Figura 5.10.5.2 podem veure les dues barres mig buides, indicant que al jugador li queden 2 punts de vida i 2 d'energia, on la barra d'energia s'està carregant.

Figura 5.10.5.2: Barres de vida i energia del jugador mig buides

5.11.3. Botons d'interacció

Durant tot el joc el jugador podrà interacctuar amb diversos objectes de diferents maneres. Aquestes interaccions es mostren al jugador amb botons on es pot veure el nom de la interacció a realitzar i el botó que s'haurà de prémer per a realitzar-la. Aquests botons poden aparèixer a sobre el personatge, com es pot veure en la Figura 5.11.3.

Figura 5.11.3.1: Botons d'interacciósobre el personatge

A més, hi ha diferents objectes que tenen aquest botó d'interacció integrat en el seu propi panell d'informació, com es pot apreciar en la Figura 5.11.3.2.

Figura 5.11.3.2: Botó d'interacció integrat en el panell de l'objecte

Per últim, tenim interaccions que requereixen que el jugador mantingui el botó d'interacció apretat durant un període de temps. Per a que pugui saber quan temps ha d'estar polsant el botó, alguns botons d'interacció tenen una barra de càrrega a sota que indiquen quan la interacció serà exectuada. En la Figura 5.11.3.3 es pot veure un exemple. Per a més informació sobre les interaccions, veure Apartat 6.9.

Figura 5.11.3.3: Interacció amb barra de progrés

5.12 Disseny dels efectes i partícules

5.12.1. Llums

En un entorn tridimensional, el tractament de llums és un element imprescindible a l'hora de generar un entorn, ja que seran les encarregades de transmetre sensacions o representar el temps.

En aquest projecte hem decidit crear dos entorns prou diferents, tot i que hem fet servir el mateix esquema d'il·luminació. L'esquema que s'ha fet servir està basat en 3 punts orientats de diverses maneres per evitar punts foscos. L'esquema en qüestió es pot veure en la Figura 5.12.1.1.

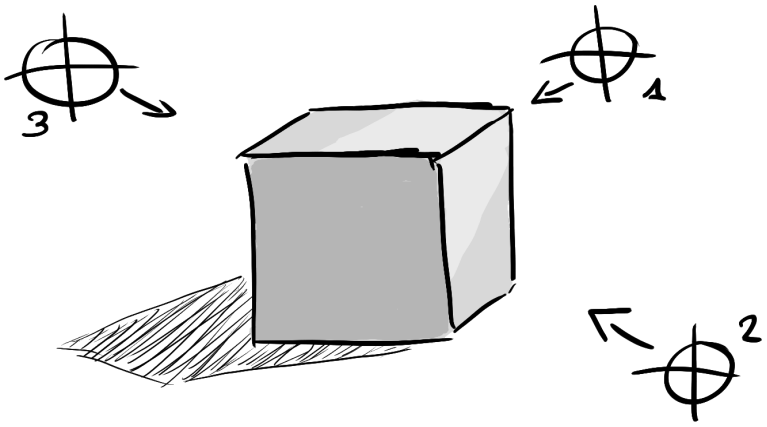


Figura 5.12.1.1: Representació de l'esquema de llums utilitzat en el projecte.

Si analitzem l'esquema de la Figura 5.12.1.1, podem veure que els punts de llums incideixen sobre la figura des d'orientacions diferents. Cal aclarir que es tracten de punts de llum global, i que la seva posició no afecta el resultat final, només la seva orientació. El punt número 1 és el més important, ja que s'encarrega de generar les ombres (complementar amb l'Apartat 5.12.6). Els punts 2 i 3 s'encarreguen de generar llum ambient per suavitzar l'efecte de les ombres provocat pel punt 1 i així obtenir una millor interpretació de les formes i colors dels objectes afectats.

En quant a la configuració feta servir per cadascuna de les escenes, trobarem una diferència en els colors i l'intensitat, ja que mentre en el *Upperworld* les tonalitats són clares i es fan servir colors càlids, a el *Underworld* es rebaixa la intensitat del llum i els colors fets servir són freds.

5.12.2. Postprocessat

Els efectes de postprocessat permeten al dissenyador realitzar ajustaments sobre la manera com es visualitza l'escena. El conjunt d'efectes de post processat permeten realitzar canvis en la interpretació dels colors de l'escena, el balanç de blancs, aplicar efectes de desenfoc i vinyeta entre altres. En aquest cas s'han aplicat diferents efectes amb l'objectiu d'aconseguir una millora visual el més similar possible a la imatge que el dissenyador té de l'escena.

Els efectes que s'han aplicat són els següents:

- **Bloom:** Aquest efecte permet magnificar les reflexions de la llum sobre els materials més brillants per provocar petits "halos" de llum al voltant del punt il·luminat.
- **Vinyetat:** Efecte que permet enmarcar la imatge captada per la càmera per enfosquir les cantonades. L'aplicació d'aquest efecte té la intenció de centrar l'atenció del jugador en la part central de la pantalla.
- **Balanç de blancs:** Efecte que permet ajustar la temperatura de l'escena i la tonalitat general. S'ha fet servir per incrementar la calidesa de l'escena.
- **Profunditat de camp:** Aquest element de postprocessat permet modificar la distància focal de la càmera, aconseguint un efecte de borrositat per distància. Se sol fer servir per augmentar la sensació de llunyania dels objectes, simulant l'ull humà.
- **Ajustament de la corba de colors:** Eina que permet ajustar l'intensitat d'un rang de colors específic.
- **Ajustament dels colors:** Paràmetre tradicional que permet modificar el contrast, l'exposició, la saturació i la tonalitat general de l'escena.

El resultat final de l'escena és la combinació de tots els efectes esmentats anteriorment, tal i com es pot veure a la Figura 5.12.2.1.

Figura 5.12.2.1. A l'esquerra l'escena amb els efectes de postprocessat activats, a la dreta sense.

5.12.3. Trails

Els trails són efectes utilitzats per mostrar el rastre que deixaría el moviment d'un objecte. Aquests trails, o cues en català, mostren la trajectòria de l'objecte i facilitan molt la interpretació del moviment, a més augmenten la sensació de velocitat. Hem utilitzat aquests trails en els orbes de recurs que el jugador podrà obtenir, com es pot veure en les Figures 5.12.3.1 i 5.12.3.2.

Figura 5.12.3.1: Trail de l'orbe de recurs de fusta

Figura 5.12.3.2: Trail de l'orde de recurs de fibra

5.12.4. Partícules

Per a poder emfatitzar accions, donar feedback i crear ambients, hem utilitzat el sistema de partícules integrat en el Unity. En les Figures 5.12.4.1 a 5.12.4.5 es mostren els efectes realitzats:

Figura 5.12.4.1: Fulla utilitzada per a les partícules de l'arbre i captura de les partícules de l'arbre

Figura 5.12.4.2: Partícules d'impacte de l'eina destral

Figura 5.12.4.3: Partícules d'obtenció del recurs fibra

Figura 5.12.4.4: Partícules ambientals del Underworld

Figura 5.12.4.5: Partícules d'impacte contra el terra de l'atac de l'enemic tentacle

5.12.5. Transicions i efectes de pantalla

En el joc trobem una única transició que ocorre entre canvis d'escenes. S'encarrega de mostrar al jugador visualment que l'escena està carregant i, quan acaba de carregar, aquesta pantalla desapareix. Podem veure la pantalla de càrrega en la Figura 5.12.5.1.

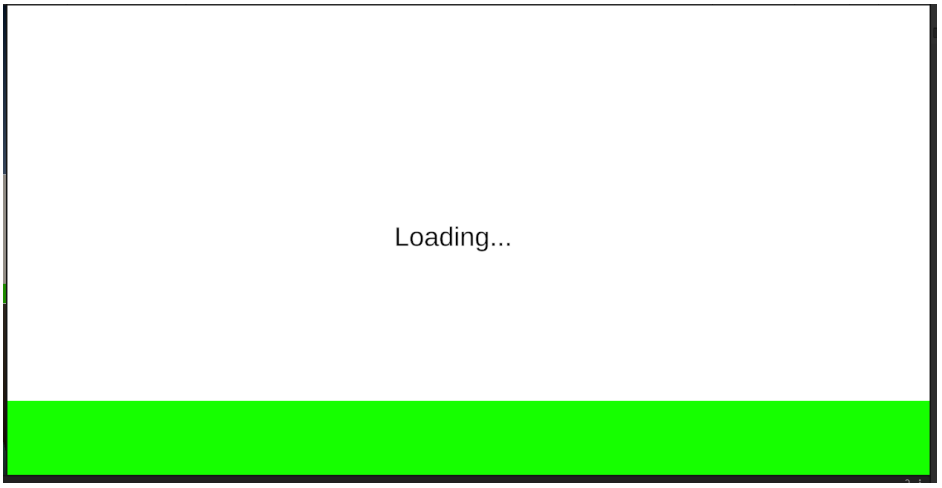


Figura 5.12.5.1: Pantalla de càrrega entre escenes

5.12.6. Ombres

De la mateixa manera que la il·luminació, les ombres prenen un paper molt important en quant a la composició de l'escena i els efectes que es volen transmetre al jugador. A l'hora de configurar un projecte, cal tenir en compte el tipus d'ombres i les característiques del projecte amb la finalitat d'ajustar el seu comportament. En aquest cas, s'ha volgut aconseguir una representació simplificada de la realitat en el joc, i per tant s'ha aplicat una configuració que permet generar ombres on tots els models afectats quedin vinculats amb la superfície que els sosté d'una manera suau i eficient.

Pel que fa a configuració de global de les ombre en el món, cal especificar la distància de generació i la qualitat. Aquests dos elements influeixen de manera significativa en el rendiment final del joc, ja que en aquest cas les ombres es generen en temps real. Una tècnica molt utilitzada en els mons de gran abast, és el renderitzat de les ombres en cascada. La tècnica de renderitzat en cascada permet al dissenyador ajustar la qualitat de les ombres i la definició en funció de la distància de la càmera. La configuració d'aquest efecte i la resta d'ajustaments fets servir en el projecte els podem veure a la Figura 5.12.6.1.

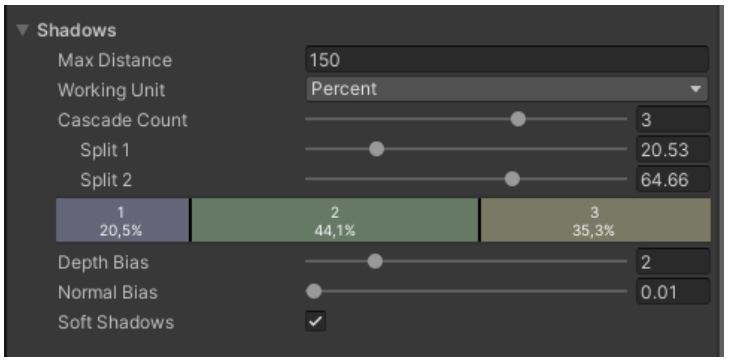


Figura 5.12.6.1: Secció de la configuració de les ombres de la pipeline de renderitzat.

5.12.7. Shaders

Un *Shader* és un algoritme matemàtic que serveix per calcular el color d'un píxel, basat en la llum i el material. En aquest cas, el nostre coneixement en shaders no és gaire ampli, i per tant ha estat necessari realitzar diferents cerques fins trobar l'efecte visual que el projecte requería. L'efecte a implementar, en aquest cas, ha estat el de la flama que podem trobar a l'altar, com es pot veure a la Figura 5.6.4.6.3 i al portal de la Figura 5.6.4.5.2.

Actualment a Unity podem treballar els shaders de dues maneres diferents, treballant amb scripts o bé amb un sistema de gràfics que porten un seguit de funcions ja implementades. En aquest cas hem decidit triar el segon ja que ens resulta més fàcil d'entendre i és un

sistema més modern amb una gran varietat d'exemples a Internet. Com es pot veure a la Figura 5.12.7.1, el gràfic resultant de la producció de la flama i el portal, està compost per diversos mòduls, on cadascun d'ells executa diferents operacions com representació de mapes de soroll, multiplicacions, restes, filtres de color, translacions de textures, rellotges…

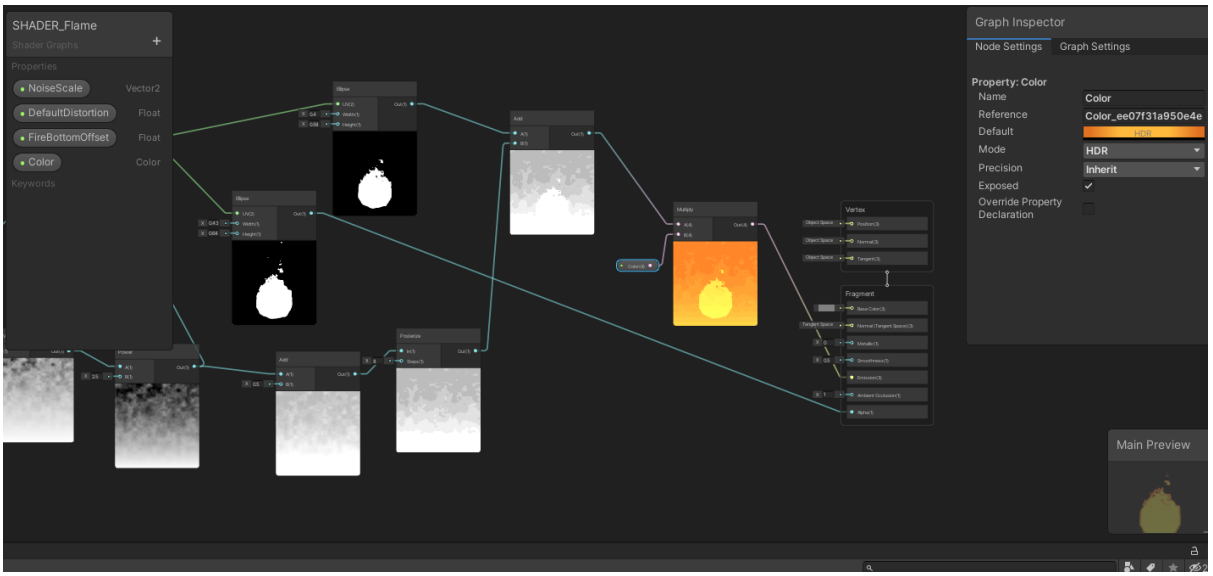


Figura 5.12.7.1: Detall de la formació d'un efecte mitjançant un shader a l'editor de Unity 3D.

Finalment, i gràcies a les variables públiques, som capaços d'ajustar el comportament de paràmetres com el soroll o el color a aplicar mitjançant l'inspector del motor. Com es pot veure a la Figura 5.12.7.2. Un cop modificat, només quedarà generar un material i col·locar-lo al model per aplicar l'efecte desitjat.

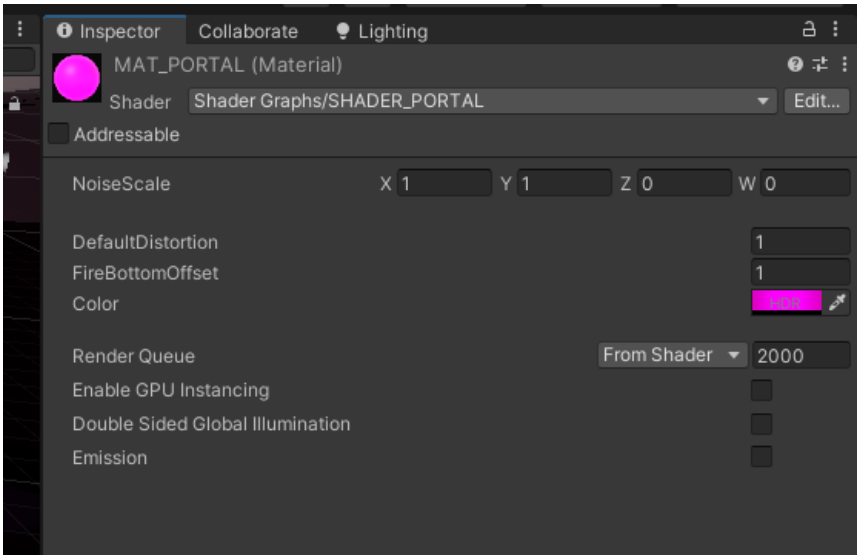


Figura 5.12.7.2: Panell d'inspecció amb les variables públiques del shader i els valors de la flama utilitzada per l'altar.

5.13. Disseny de so

A part de la producció musical esmentada en l'Apartat 5.9, hem utilitzat altres recursos de so de lliure ús, per emfatitzar accions, com un so per al moviment d'atac de les eines del personatge o un so de petjada per a quan el personatge camina. Per a fer-ho, hem utilitzat el sistema integrat de Unity que utilitza controladors de barreja d'audio (Audio MIxer Controller), per a controlar les diferents característiques de les pistes d'àudio. Com es pot veure en la Figura 5.13.1, tenim un controlador per a cada escena, encarregats de gestionar les pistes de so que reben per esdeveniments.

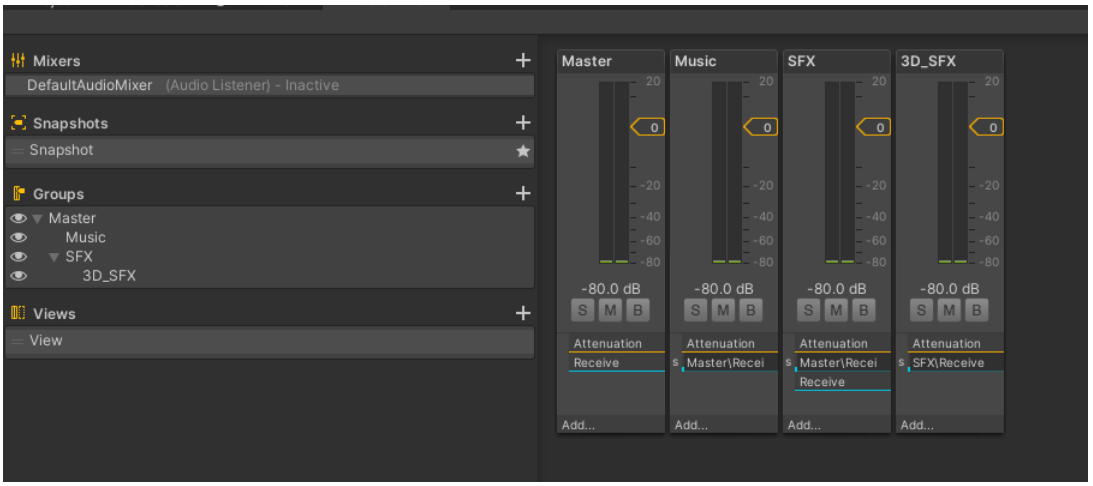


Figura 5.13.1: Controlador de barreja d'audio

A més, podem customitzar els valors per a cada canal d'audio depenent de l'objectiu, com podem veure en la Figura 5.13.2.

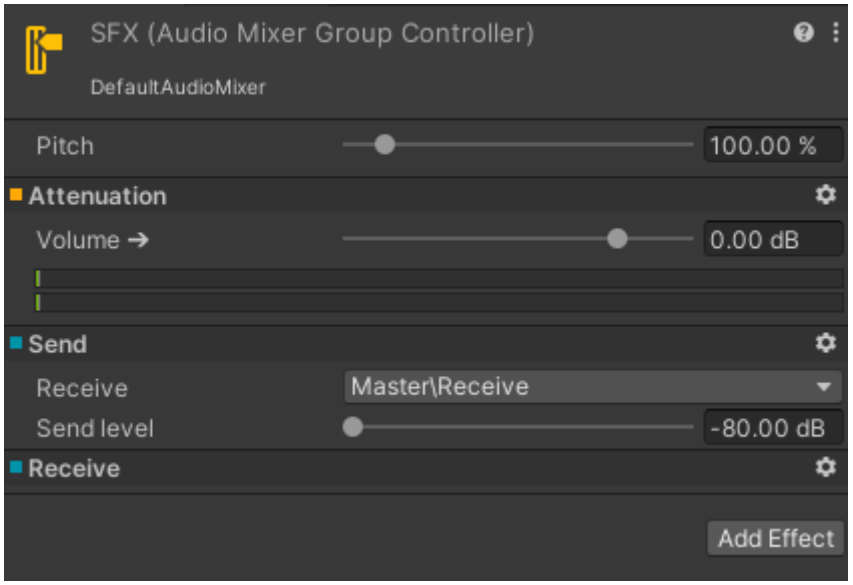


Figura 5.13.2: Atributs del canal SFX

5.14. Disseny de les càmeres

En el nostre joc trobem sempre una càmera fixa que segueix al personatge. Aquesta càmera no pot ser rotada i sempre estarà en la mateixa distància al personatge. Per a poder portar gestionar aquest comportament, hem utilitzat el paquet de Unity Cinemachine, encarregat de controlar la lògica de les diferents càmeras d'un joc.

5.14.1 Cinemachine

Per a poder utilitzar i configurar una càmera Cinemachine, s'ha d'assignar el component CinemachinBrain a la càmera principal de l'escena, com es pot veure en la Figura 5.14.1.1.

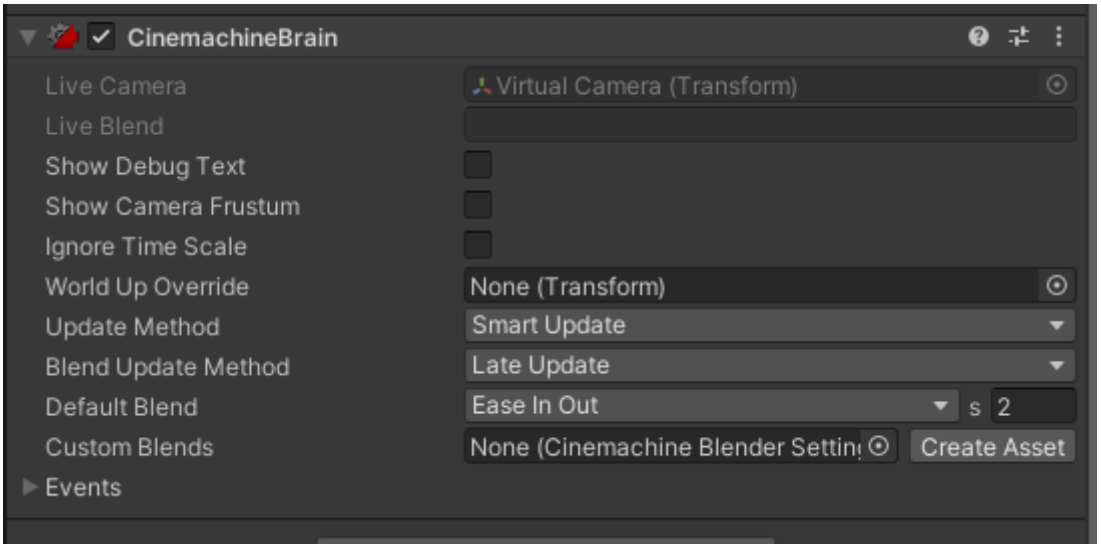


Figura 5.14.1.1: Component CinemachineBrain de l'objecte càmera principal

A més, hem de crear el què es denomina com a camera virtual. Aquest objecte camera virtual serà l'encarregat de gestionar els moviments i la lògica de la càmera principal que tingui el component CinemachineBrain. per tant, l'estructura del sistema de càmeras quedaria tal i com podem veure en la Figura 5.14.1..2.

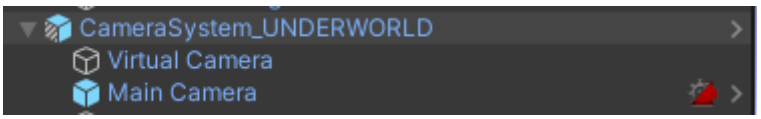


Figura 5.14.1..2: Estructura de l'objecte CameraSystem_UNDERWORLD

Un cop tinguem aquesta estructura, ja podem personalitzar el comportament de la càmera virtual que s'encarregarà de moure la càmera principal. Cinemachine vé amb molts modes i funcions disponibles però les nostres necessitats eren molt simples. Per tant, hem agafat el mode de seguiment i, com a target li hem assignat la transformació del personatge, com es pot veure en la Figura 5.14.1.3.

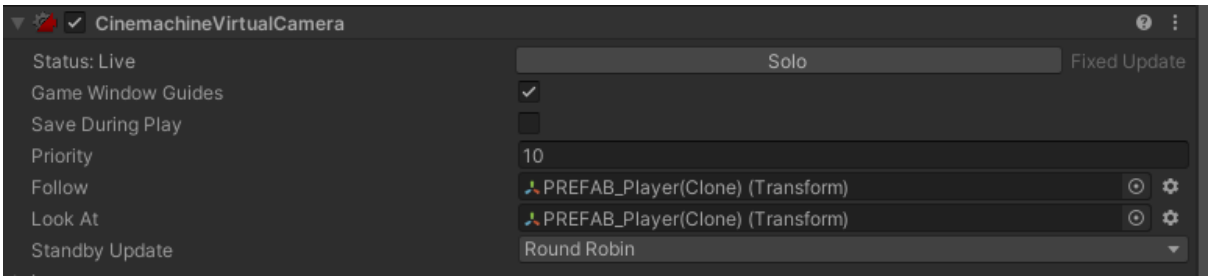


Figura 5.14.1.3: Atributs follow i look at del component CinemachineVirtualCamera assignats al jugador

Per últim, hem configurat la posició de la càmera i una *dead zone.* Aquesta zona el que fa és que, mentre el jugador no surti d'aquesta zona, la càmera no es moura. Un cop el jugador surti la càmera el seguira fins que torni a estar dins de la *dead zone.*

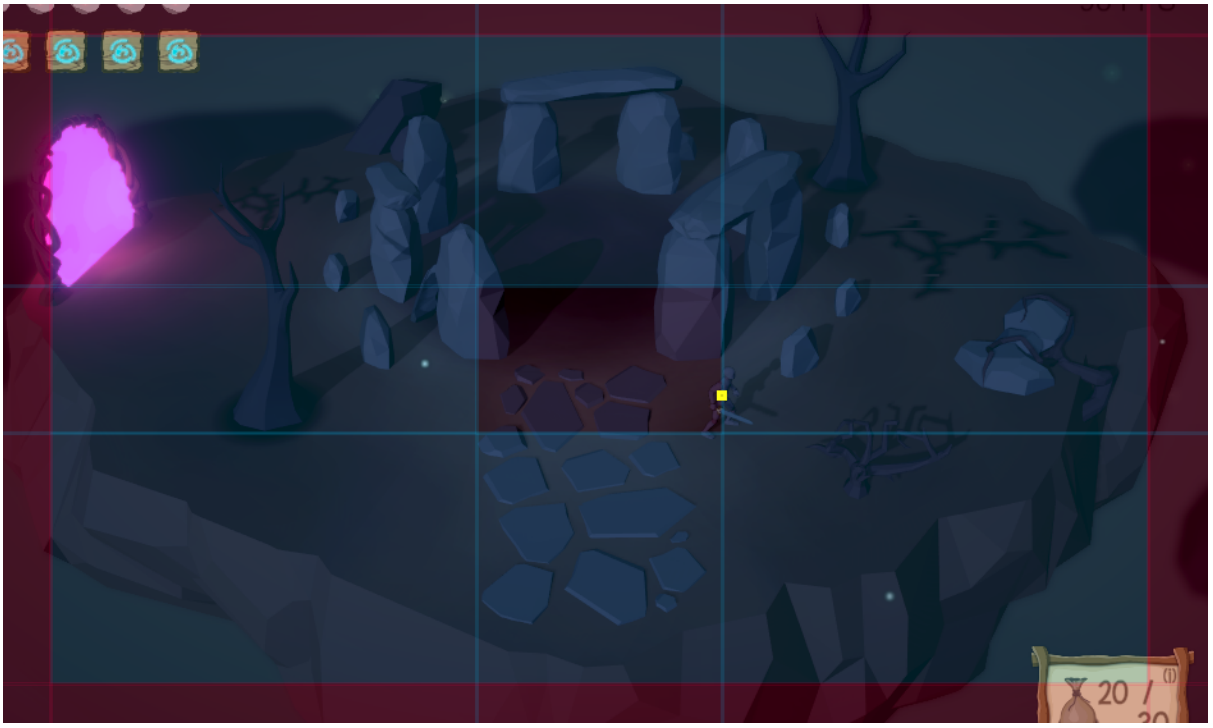


Figura 5.14.1.4: *Dead zone* de la càmera virtual

Com es pot veure en la figura 5.14.1.4, la *dead zone* és el quadre central sense pintar. El punt groc representa l'objectiu, en aquest cas és la transformació del jugador. Mentre el punt estigui dins, la càmera no es mourà. Això fa que la càmera passi de tenir un moviment estàtic i "robòtic", a un moviment més fluid i satisfactori. Per a definir els paràmetres de la *dead zone*, hem configurat els valors dels atributs com podem veure en la Figura 5.14.1.5.

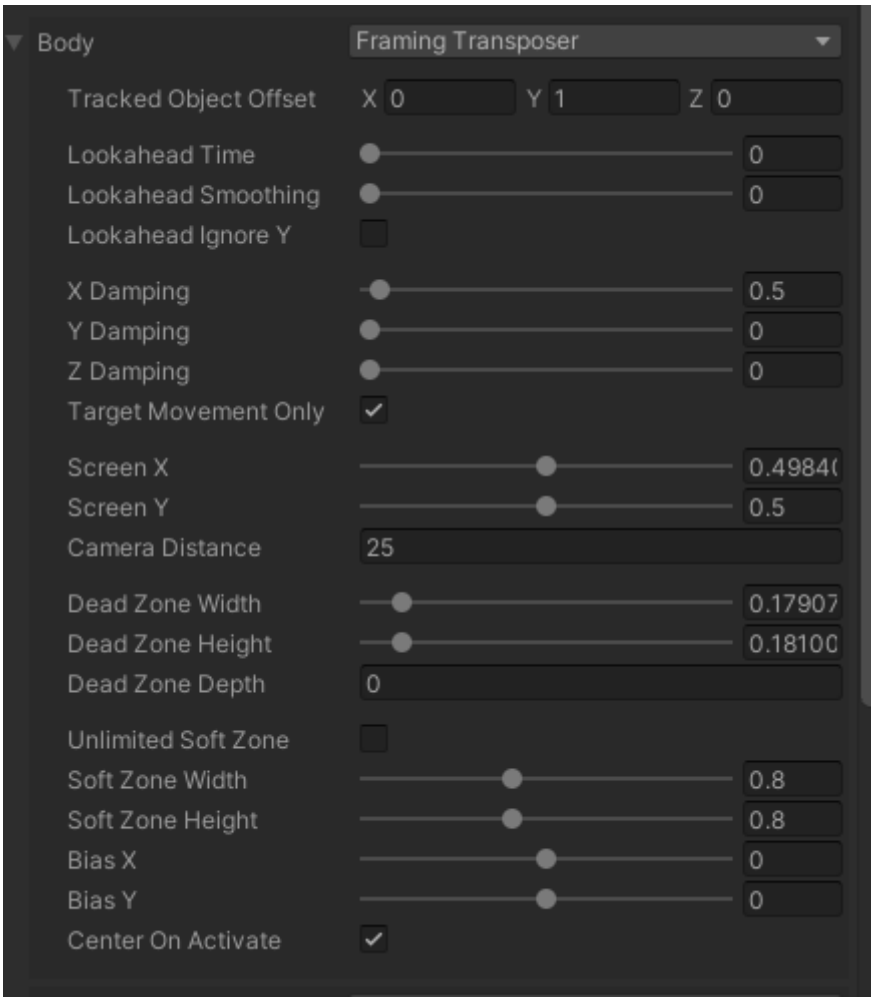


Figura 5.14.1.5: Atributs de la *dead zone* de la càmera virtual

6. Implementació i proves

6.1. Estructura del projecte i conceptes de programació

6.1.1 Patró de programació

Per aquest projecte hem decidit utilitzar un patró de programació basat en esdeveniments, sense clases estàtiques ni controladors que requereixin estar presents en tots moments en cada una de les escenes del joc. Ho hem fet d'aquesta manera per tenir un primer prototip modular i que sigui extremadament ampliable en el futur desenvolupament. Per poder assolir-ho, hem fet molt d'ús del tipus de classe ScriptableObject que ens proporciona Unity.

6.1.2. ScriptableObjects

Un ScriptableObject és, per definició, un contenidor d'informació el qual no està adjunt a cap objecte en l'escena, sino que resideix dins els arxius del joc com un recurs més. El seu principal ús és reduir la memòria del projecte evitant repeticions de valors. Per exemple tots els enemics d'un tipus donat aniran a buscar la informació que necessiten en un sol punt dins els arxius de joc. Aquesta característica principal l'hem fet servir també per a guardar la informació dels diferents ítems que té el joc.

L'enfocament principal que els hi hem donat però, és utilitzar-los com a llançadors d'esdeveniments. Totes les classes que requereixen rebre o enviar informació a altres classes, independentment de que siguin presents en la seva propia escena, utilitzen aquests objectes. D'aquesta manera aconseguim eliminar completament el requeriment de crear instàncies estàtiques de controladors globals (patró singleton) que requereixen existir en tot moment dins l'escena de joc per a poder rebre tota la informació que necessiten. Per tant, qualsevol esdeveniment d'un ScriptableObject que sigui llençat per qualsevol classe serà rebut per qualsevol altra classe que estigui escoltant aquell ScriptableObject específic. En la Figura 6.1.2.1 podem veure un esquema del funcionament d'aquests esdeveniments.

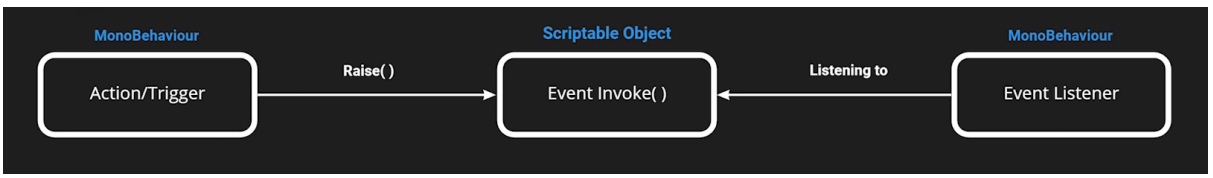


Figura 6.1.2.1: Esquema d'esdeveniments amb ScriptableObjects

Partint d'aquesta premisa, vam fer servir herència per crear un objecte base que hereta de la clase ScriptableObject i un seguit de classes personalitzades que hereten d'aquesta classe base, com podem veure en la Figura 6.1.2.2.

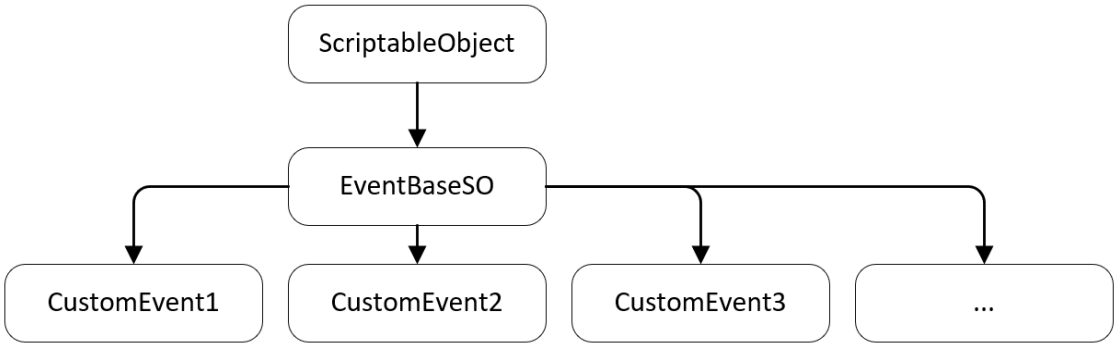


Figura 6.1.2.2: Jerarquía de les classes d'esdeveniment.

La classe base simplement té un atribut descripció per poder tenir una definició clara de quina és la funció de cada esdeveniment personalitzat, com podem veure en la Figura 6.1.2.3.

```
Script de Unity | 13 referencias
public class EventBaseSO : ScriptableObject
{
    [TextArea] public string description;
}
```

Figura 6.1.2.3: Classe base esdeveniment

Pel que fa als esdeveniments personalitzats, són simplement classes que hereten de EventBaseSO i que cadascun d'ells s'encarrega de llançar un esdeveniment amb diferents tipus de paràmetres.

```
[CreateAssetMenu(menuName = "Scriptable Objects/Events/Int Event")]
Script de Unity | 7 referencias
public class IntEventSO : EventBaseSO
{
    public UnityAction<int> OnEventRaised;

    3 referencias
    public void RaiseEvent(int value)
    {
        if (OnEventRaised != null)
            OnEventRaised.Invoke(value);
    }
}
```

Figura 6.1.2.4: Classe esdeveniment d'enter

En la Figura 6.1.2.4 podem veure un exemple d'esdeveniment personalitzat. Tots els esdeveniments personalitzats tenen un delegat públic i una funció pública amb els paràmetres concordants. D'aquesta manera, els objectes a l'escena que vulguin llençar un esdeveniment, cridaran a la funció RaiseEvent del ScriptableObject que vulguin, passant el paràmetre de valor requerit. Aquesta funció serà l'encarregada de comprobar si algún altre objecte està escoltant l'esdeveniment i, si és així, invocarà el delegat amb el paràmetre.
A continuació llistem tots els esdeveniments personalitzats que són classes idèntiques a l'exemple de la Figura 6.1.2.4, però simplement canviant el tipus de paràmetre:

- **BoolEventSO**: Esdeveniment que passa com a paràmetre un booleà
- **DishEventSO**: Esdeveniment que passa com a paràmetre un objecte de la classe Dish
- **DishListEventSO**: Esdeveniment que passa com a paràmetre un llistat d'objectes de la classe Dish
- **GameObjectEventSO**: Esdeveniment que passa com a paràmetre un objecte del tipus GameObject
- **InteractionEventSO**: Esdeveniment que passa com a paràmetre un objecte de la classe InteractionSO
- **InteractionUIEventSO**: Esdeveniment que passa com a paràmetres un booleà i una llista d'objectes de la classe InteractionSO
- **IntEventSO**: Esdeveniment que passa com a paràmetre un enter
- **InventorySlotEventSO**: Esdeveniment que passa com a paràmetre un objecte de la classe InventorySlot
- **ItemContainerEventSO**: Esdeveniment que passa com a paràmetre un objecte de la classe ItemContainer
- **SeedEventSO**: Esdeveniment que passa com a paràmetre un objecte de la classe SeedSO
- **ShakeCameraEventSO**: Esdeveniment que passa com a paràmetres 2 floats (amplitud i temps del moviment de la càmera)
- **ToolEventSO**: Esdeveniment que passa com a paràmetre un objecte de la classe ToolSO
- **TransformEventSO**: Esdeveniment que passa com a paràmetre un objecte de tipus Transform
- **UsableEventSO**: Esdeveniment que passa com a paràmetre un objecte de la classe UsableSO
- **VoidEventSO**: Esdeveniment que no passa cap paràmetre, utilitzat per cridar a una funció qualsevol en un moment donat.

Aquest últim tipus d'esdeveniment (VoidEventSO) és el que més hem fet servir, ja que en molts casos simplement volem escoltar un esdeveniment i, quan s'activa, du a terme una determinada resposta. A continuació podem veure exemples de VoidEventSO on cadascun d'ells s'activa en diferents moments de la partida. En aquest cas s'activen quan un objecte requereix que es canvi d'escena.

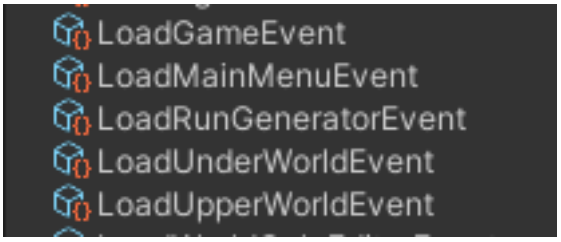


Figura 6.1.2.5: Llistat de VoidEventSO activats quan es requereix que es carregui una escena especifíca

En la Figura 6.1.2.5 podem veure com a exemple els ScriptabelObjects utilitzats com a esdeveniments que hem requerit per la gestió de la interfície. Cadascun d'ells és escoltat per elements de la interfície i ser llançats per altres objectes del joc amb o sense paràmetres, sense importar la seva localització ni en quina escena hi siguin.

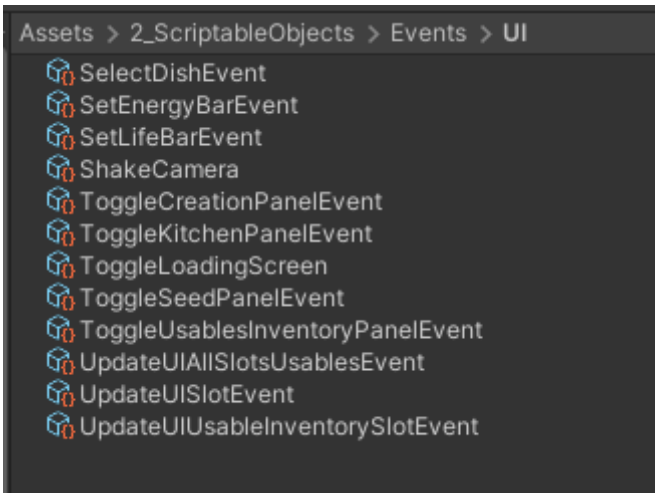


Figura 6.1.2.6: Exemple de llistat d'esdeveniments d'interfície a partir de ScriptableObjects

6.1.3. Gestió de recursos

Pel que fa a la gestió de recursos, hem fet servir el paquet Addressable Asset System de Unity que proporciona, de manera senzilla, la gestió de càrrega i guardat de recursos. D'aquesta manera em assolit un dels requeriments del desenvolupament, que era tenir un món persistent amb guardat de partida i també la possibilitat d'afegir objectes dinámicamente durant el transcurs de la partida per aconseguir un entorn ric i viu.

Aquest paquet proporciona un sistema per poder identificar els assets que fem servir i carregar-los només quan siguin necessaris, evitant molta càrrega de memòria i fent que el projecte sigui molt més lleuger. Així podem guardar l'identificador de qualsevol asset (marcat com a addressable), que estigui present en la escena i, posteriorment utilitzar aquest mateix identificador per a poder tornar a carregar-ho.

En la Figura 6.1.3.1 podem veure els assets marcats com a addressables diferenciats per grups. Aquests assets podran ser carregats a escena ja sigui pel identificador que tenen, pel grup que formen o pels tags que els defineixen. Mentre no es requereixin, no formaran part de la memòria i al destruir-los també deixaran lliure l'espai de memòria que ocupaven.

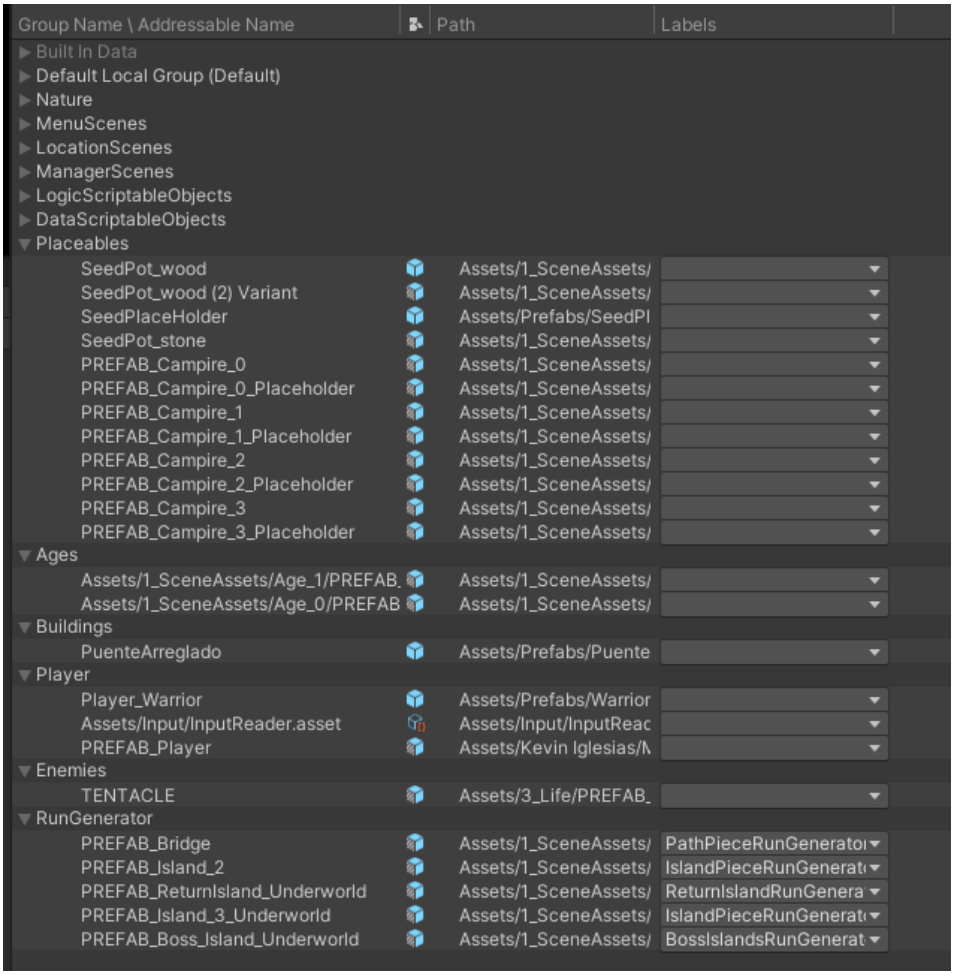


Figura 6.1.3.1: Vista del gestor del Addressable System de Unity

Un problema que ens hem trobat durant el desenvolupament és el conflicte que hi ha entre els objectes Addressables y els ScriptableObjects. Per una banda, els ScriptableObjects utilitzats com a magatzem d'informació havien d'estar guardats com a Addressables per poder guardar-los en l'arxiu de guardat i, al carregar la partida que tornessin al estat original (per exemple en el cas de l'inventari del jugador). Fins aquest punt tot funciona sense cap misteri. El problema arriba quan utilitzem els ScriptableObjects com a lligam entre diferents objectes o escenes. Si no guardem aquests "lligams" com a Addressables, cada objecte o escena que en referenií un d'ells, crearà la seva pròpia còpia. Aixó fá que un mateix esdeveniment estigui duplicat i per tant, quan un objecte el llenci, ningú l'escoltara perquè l'objecte que hauria d'estar escoltant-lo, ho està fent però d'una còpia del propi esdeveniment. Per a solucionar-ho, simplement ens hem d'assegurar que tots els esdeveniments també siguin Addressables i, així el propi sistema ja comprovarà si existeix i, en cas afirmatiu, agafa la referència correcta i no crearà un nou objecte. Aquest problema és pràcticament impossible de debugar pel que s'ha d'anar amb molta cura en aquests casos.

6.2. Implementació dels nivells

6.2.1. Estructura de les escenes

El joc està estructurat en diferents tipus d'escenes: les escenes controladores i les escenes de món. D'escenes de món en tenim 4:

1. MainMenu: Escena del menú principal on el jugador podrà escollir si començar una nova partida, continuar o sortir.
2. UpperWorld: Escena del món superior on el jugador passarà gran part de l'aventura.
3. UnderWorld: Escena principal del món inferior. Punt intermig entre el món superior i la part del món inferior generada aleatoriamente.
4. RunGenerator: Escena del món inferior generada aleatoriament.

Com a escenes controladores en tenim 3:

1. Inizalization: Escena inicial del joc, encarregada de carregar l'escena de controladors persistents i el menú principal.
2. PersistantManagers: Escena que, un cop inicialitzada, sempre existirà en el joc. La funció es portar el control de gestors globals que siguin necessaris en tot moment.
3. Gameplay: Escena que existirà semrpe que el jugador estigui en una escena de món, excepte la escena MainMenu. Encarregada dels elements de la interfície i del spawn de recursos.

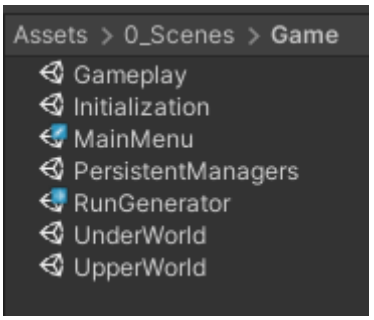


Figura 6.2.1.1: Estructura d'escenes dins els arxius

Totes les escenes estan marcades com a Addressables, es a dir, que per defecte no apareixen a la build final del joc fins que no se les carregui en memoria. Això afavoreix el pes inicial del joc i fa que l'aplicació, un cop instalada, pesi molt poc i, a mesura que es requereixi s'augmenta el consum de memòria. D'aquesta manera, com a estructura d'escenes finals de la build del projecte només apareix la escena inicialitzadora, ja que ella s'encarregara de carregar les escenes inicials, com podem veure en la Figura 6.2.1.2

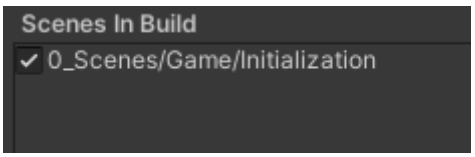


Figura 6.2.1.2: Escenes en la build final del joc

Com a aclariment, el mètode convencional de Unity es colocar en la jerarquía d'escenes de la build final del joc totes les escenes que apareixen durant tota la partida i, normalment ordenades. Aquest aproximament és correcte però el que provoca és que tots els arxius referenciats dins de cada una de les escenes (prefabs, imatges, text...) es carreguin també simplement en iniciar el joc. És a dir, Unity ja té preparat en memòria tot el que s'indiqui en la jerarquia d'escenes, encara que es tracti de l'últim nivell del joc en el que el jugador no hi podrà accedir fins més endavant. Per tant, el pes inicial de l'aplicació serà molt més elevat que no pas amb el mètode que hem utilitzat.

6.2.2. Gestió de les escenes

La metodologia seguida per a la estructura i gestió d'escenes es basa també amb ScriptableObjects. Cada escena té un ScriptableObject de clase GameSceneSO en el que hi apareix la informació específica de cada escena, com podem veure en la Figura 6.2.2.1.

```
Script de Unity | 5 referencias
public class GameSceneSO : ScriptableObject
{
    [Header("Information")]
    public string sceneName;
    public string shortDescription;
    public AssetReference sceneReference;

    [Header("Sounds")]
    public AudioCueSO musicTrack;

}
```

Figura 6.2.2.1: Classe GameSceneSO

Aquesta classé té com a atributs el nom de l'escena, una breu descripció, una referència a la adreça de l'escena, i una classe AudioCueSO per si hi ha música de fons. D'aquesta manera podem referenciar i obtenir informació de cada escena a partir dels ScriptableObjects creats, els quals podem veure en la Figura 6.2.2.2.

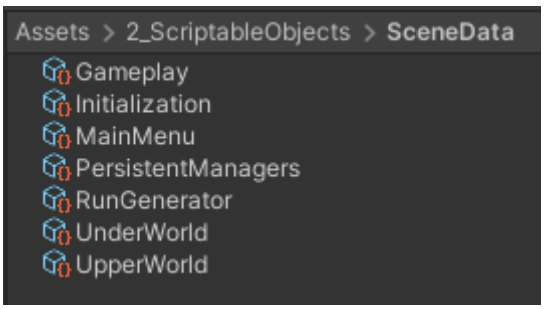


Figura 6.2.2.2: Llistat de GameSceneSO, un per cada escena del joc

La gestió d'escenes, que comporta la càrrega i descàrrega de les diferents escenes del joc, es du a terme en 2 punts que s'explicaran en els següents apartats.

6.2.2.1 Initialization

L'escena Initialization està formada únicament per l'objecte Initializer, com podem veure en la Figura 6.2.2.1.1.

Figura 6.2.2.1.1: Estructura de l'escena Initializer

Aquest objecte simplement té la classe Initialization Loader que és l'encarregada de inicialitzar 2 escenes: l'escena PersistantManagers i MainMenu.

```
public class InitializationLoader : MonoBehaviour
{
    [Header("Persistent managers Scene")]
    [SerializeField] private GameSceneSO _persistentManagersScene = default;
    [SerializeField] private AssetReference _loadMainMenu = default;
    Mensaje de Unity | 0 referencias
    private void Start(){
        _persistentManagersScene.sceneReference
            .LoadSceneAsync(LoadSceneMode.Additive, true).Completed += LoadEventChannel;
    }
    1 referencia
    private void LoadEventChannel(AsyncOperationHandle<SceneInstance> obj){
        _loadMainMenu.LoadAssetAsync<VoidEventSO>().Completed += LoadMainMenu;
    }
    1 referencia
    private void LoadMainMenu(AsyncOperationHandle<VoidEventSO> obj){
        VoidEventSO loadEventChannelSO = (VoidEventSO)_loadMainMenu.Asset;
        loadEventChannelSO.RaiseEvent();

        SceneManager.UnloadSceneAsync(0);
    }
}
```

Figura 6.2.2.1.2: Classe InitializationLoader

Com podem veure en la Figura 6.2.2.1.2, la classe InizalizationLoader té 2 atributs privats serialitzats a través del inspector:

- _persistantManagerScene: Objecte de la classe GameSceneSO amb la informació sobre l'escena PersistantManager.
- _loadMainMenu: Referència a l'esdeveniment de càrrega del menú principal.

Al tractar-se de la primera escena, hem de carregar els esdeveniments que vulguem llançar amb el sistema d'Addressables, com hem comentat en l'Apartat 6.1.3 per abolir duplicaciones i no trencar el fluxe. Per això tenim la referència del esdeveniment loadMainMenu i no el propi esdeveniment en sí.

El mètode **Start** té la funció de carregar l'escena PersistantManagers i cridar a la funció LoadEventChannel quan l'escena estigui carregada.

LoadEventChannel obté l'esdeveniment loadMainMenu de memòria i crida a la funció LoadMainMenu quan l'esdeveniment estigui carregat.

LoadMainMenu rep com a paràmetre un objecte carregat de memòria de tipus VoidEventSO i llença l'esdeveniment i, posteriorment descarrega l'escena actual, ja que no es tornarà a fer servir.

6.2.2.2 SceneLoader

SceneLoader és un objecte amb la classe SceneLoader dins de l'escena Persistentmanagers i la funció és gestionar la càrrega i descàrrega d'escenes durant el transcurs de la partida.

Model de classe SceneLoader:

SceneLoader
-_gameplayScene: AssetReference -_menuScene: AssetReference -_upperWorldScene: AssetReference -_underWorldScene: AssetReference -_runGeneratorScene: AssetReference -_startGameEvent: VoidEventSO -_newGameEvent: VoidEventSO -_loadMenuEvent: VoidEventSO -_loadUpperWorldEvent: VoidEventSO -_loadUnderWorldEvent: VoidEventSO -_loadRunGeneratorEvent: VoidEventSO -_gameplayManagerLoadingOpHandle: AsyncOperationHandle<SceneInstance> -_mainMenuManagerLoadingOpHandle: AsyncOperationHandle<SceneInstance> -_upperWorldManagerLoadingOpHandle: AsyncOperationHandle<SceneInstance> -_underWorldManagerLoadingOpHandle: AsyncOperationHandle<SceneInstance>

-_runGeneratorManagerLoadingOpHandle: AsyncOperationHandle<SceneInstance> -_gameplayManagerSceneInstance: SceneInstance -_mainMenuManagerSceneInstance: SceneInstance -_upperWorldManagerSceneInstance: SceneInstance -_underWorldManagerSceneInstance: SceneInstance -_runGeneratorManagerSceneInstance: SceneInstance -_onSceneReady: VoidEventSO -_onLoadGame: VoidEventSO -_onSceneLoadingComplete: VoidEventSO +loadingScreen: GameObject -loadingScreenState: bool
-OnEnable(): void -OnDisable(): void -ToggleLoadingScreen: void -LoadMainMenu: void -ContinueGame: void -NewGame: void -GoToUpperWorld: void -GoToUnderWorld: void -GoToRunGenerator: void -WaitForLoadingUpperWorld: IEnumerator -WaitForLoadingNewGame: void -WaitForLoadingMainMenu: void -WaitForLoadingUnderWorld: void -WaitForLoadingRunGenerator: void -LoadGameplay: void -LoadMenu: void -LoadUpperWorld: void -LoadUnderWorld: void -LoadRunGenerator: void -UnloadGameplay: void -UnloadMainMenu: void -UnloadUpperWorld: void -UnloadUnderWorld: void -UnloadRunGenerator: void -LoadingGameplay(): void -LoadingUpperWorldt(): void -LoadingUnderWorld(): void -LoadingRunGenerator(): void -LoadingMainMenu(): void

- **_gameplayScene**: Referència a l'escena Gameplay
- **_menuScene**:Referència a l'escena MainMenu
- **_upperWorldScene**: Referència a l'escena UpperWorld
- **_underWorldScene**: Referència a l'escena UnerWorld
- **_runGeneratorScene**: Referència a l'escena RunGenerator
- **_startGameEvent**: objecte VoidEventSO per l'esdeveniment d'inici de joc
- **_newGameEvent**: objecte VoidEventSO per l'esdeveniment de nou joc
- **_loadMenuEvent**: objecte VoidEventSO per l'esdeveniment de càrrega del menu principal
- **_loadUpperWorldEvent**: objecte VoidEventSO per l'esdeveniment de càrrega de

l'escena UpperWorld

- **_loadUnderWorldEvent**: objecte VoidEventSO per l'esdeveniment de càrrega de l'escena UnderWorld
- **_loadRunGeneratorEvent**: objecte VoidEventSO per l'esdeveniment de càrrega de l'escena RunGenerator
- **_gameplayManagerLoadingOpHandle**: Resolució del operador asíncron de càrrega de l'escena gameplay
- **_mainMenuManagerLoadingOpHandle**: Resolució del operador asíncron de càrrega de l'escena menu
- **_upperWorldManagerLoadingOpHandle**: Resolució del operador asíncron de càrrega de l'escena upperWorld
- **_underWorldManagerLoadingOpHandle**: Resolució del operador asíncron de càrrega de l'escena underWorld
- **_runGeneratorManagerLoadingOpHandle**: Resolució del operador asíncron de càrrega de l'escena runGenerator
- **_gameplayManagerSceneInstance**: Instancia actual de l'escena Gameplay
- **_mainMenuManagerSceneInstance**: Instancia actual de l'escena MainMenu
- **_upperWorldManagerSceneInstance**: Instancia actual de l'escena UpperWorld
- **_underWorldManagerSceneInstance**: Instancia actual de l'escena UnderWorld
- **_runGeneratorManagerSceneInstance**: Instancia actual de l'escena RunGenerator
- **_onSceneReady**: objecte VoidEventSO per l'esdeveniment de escena preparada
- **_onLoadGame**: objecte VoidEventSO per l'esdeveniment de carregar l'arixu de guardat
- **_onSceneLoadingComplete**: objecte VoidEventSO per a l'esdeveniment de escena carregada i llesta

+ **loadingScreen**: Pantalla de càrrega

- **loadingScreenState**: Estat de la pantalla de càrrega

Mètodes de SceneLoader:

- **OnEnable**, **OnDisable**: Funcions cridades pel Unity quan l'objecta s'activa i es desactiva. Utilitzades per subscriure's i desubscriure's als esdeveniments.

```
private void OnEnable(){
    _startGameEvent.OnEventRaised += ContinueGame;
    _newGameEvent.OnEventRaised += NewGame;
    _loadMenuEvent.OnEventRaised += LoadMainMenu;
    _loadUpperWorldEvent.OnEventRaised += GoToUpperWorld;
```

```
    _loadUnderWorldEvent.OnEventRaised += GoToUnderWorld;
    _loadRunGeneratorEvent.OnEventRaised += GoToRunGenerator;
    _onSceneLoadingComplete.OnEventRaised +=ToggleLoadingScreen;
}

private void OnDisable(){
    _startGameEvent.OnEventRaised -= ContinueGame;
    _newGameEvent.OnEventRaised -= NewGame;
    _loadMenuEvent.OnEventRaised -= LoadMainMenu;
    _loadUpperWorldEvent.OnEventRaised -= GoToUpperWorld;
    _loadUnderWorldEvent.OnEventRaised -= GoToUnderWorld;
    _loadRunGeneratorEvent.OnEventRaised -= GoToRunGenerator;
    _onSceneLoadingComplete.OnEventRaised -= ToggleLoadingScreen;
}
```

- **ToggleLoadingScreen**: Funcióm per activar o desactivar la pantalla de càrrega

```
private void ToggleLoadingScreen(){
    loadingScreenState = !loadingScreenState;
    loadingScreen.SetActive(loadingScreenState);
}
```

- **LoadMainMenu**: Funció que carrega el menú principal, descarregant si es necessari les altres escenes

```
private void LoadMainMenu(){
    ToggleLoadingScreen(); StartCoroutine(WaitForLoadingMainMenu());
    UnloadUpperWorld(); UnloadGameplay(); UnloadUnderWorld(); UnloadRunGenerator();
    LoadMenu();
}
```

- **ContinueGame, NewGame**: Funcion que carreguen l'escena UpperWorld. Tenen la mateixa estructura però en ContinueGame carreguem l'estat del món amb l'arxiu de guardat i NewGame deixa el món per defecte. A conrtinuació es mostra ContinueGame com a exemple.

```
private void ContinueGame(){
    ToggleLoadingScreen();
    if (_upperWorldManagerSceneInstance.Scene.isLoaded){
        _onLoadGame.RaiseEvent(); _onSceneReady.RaiseEvent(); return;
    }
    StartCoroutine(WaitForLoadingUpperWorld());
    UnloadMainMenu(); UnloadUnderWorld(); UnloadRunGenerator();
    LoadGameplay(); LoadUpperWorld();
}
```

- **GoToUpperWorld**, **GoToUnderWorld**, **GoToRunGenerator**: Funcions que carreguen les 3 diferents escenes de món del joc. Tenen la mateixa estructura però canviant l'escena a carregar i les escenes a descarregar. A continuació es mostra GoToUpperWorld com a exemple.

```
private void GoToUpperWorld(){
    ToggleLoadingScreen();
    StartCoroutine(WaitForLoadingUpperWorld());UnloadUnderWorld();
    LoadGameplay(); LoadUpperWorld();
}
```

- **WaitForLoadingNewGame, WaitForLoadingUpperWorld, WaitForLoadingMainMenu, WaitForLoadingUnderWorld, WaitForLoadingUnderWorld, WaitForLoadingRunGenerator**:

  Corrutines encarregades d'esperar a que la escena escollida es carregui. Un cop carregada la deixa com a escena activa, activa les llums de l'escena i llença l'esdeveniment de carregar arxiu de guardat (només en el cas de l'escena UpperWorld) i l'esdeveniment d'escena llesta. A continuació es mostra WaitForLoadingUpperWorld com a exemple.

```
private IEnumerator WaitForLoadingUpperWorld(){
    while (!_upperWorldManagerSceneInstance.Scene.isLoaded)yield return null
    SceneManager.SetActiveScene(_upperWorldManagerSceneInstance.Scene);
    LightProbes.TetrahedralizeAsync();
    _onLoadGame.RaiseEvent();
    _onSceneReady.RaiseEvent();
}
```

- **LoadGameplay, LoadMenu, LoadUpperWorld, LoasUnderWorld, LoadRunGenerator**:

  Funcions que activen la càrrega asíncrona de l'escena triada, emmagatzemen l'estat dins la variable de resolució de l'escena i llencen la corrutina de carregant l'escena. Totes tenen la mateixa estructura amb les diferents variables assiganades a l'escena en concret. A continuació es mostra LoadUpperWorld com a exemple.

```
private void LoadUpperWorld(){
   _upperWorldManagerLoadingOpHandle =
   _upperWorldScene.LoadSceneAsync(LoadSceneMode.Additive, true, 0);
              StartCoroutine(LoadingUpperWorld());
}
```

- **UnloadGameplay, UnloadMainMenu, UnloadUpperWorld, UnloadUnderWorld, UnloadRunGenerator**:

  Funcions que descarreguen (si estan carregades), cada una de les escenes del joc. A continuació es mostra UnloadUpperWorld com a exemple.

```
private void UnloadUpperWorld(){
    if (_upperWorldManagerSceneInstance.Scene != null
        && _upperWorldManagerSceneInstance.Scene.isLoaded)
        Addressables.UnloadSceneAsync(_upperWorldManagerLoadingOpHandle, true);
}
```

- **LoadingGameplay, LoadingUpperWorld, LoadingUnderWorld, LoadingMainMenu LoadingRunGenerator**:

  Corrutines encarregades d'esperar mentre l'escena no s'hagi carregat. Un cop carregada guarda l'instància en la variable d'instància de la escena escollida. Totes tenen la mateixa estructura. A continuació es mostra LoadingUpperWorld com a exemple.

```
private IEnumerator LoadingUpperWorld(){
    while (_upperWorldManagerLoadingOpHandle.Status !=AsyncOperationStatus.Succeeded)
        yield return null;
        _upperWorldManagerSceneInstance =_upperWorldManagerLoadingOpHandle.Result;
}
```

6.3. Gestió input del jugador

6.3.1. Unity Input System

Pel que fa a la gestió del input del jugador, hem fet servir el nou sistema de Unity Input System. Aquest sistema proporciona una manera de maquetar diferents esquemes de control depenent el dispositiu amb el qual s'estigui jugant. Dins de cada esquema, s'estableixen accions que el jugador podrà realitzar amb els diferents botons. Aquestes accions poden tenir diferents comportaments, com per exemple apretar el botó, mantenir polsat durant uns segons, fer doble toc, etc. D'aquesta manera podem tenir diferents esquemes per a que estiguin enllaçats amb els possibles estats del jugador, modificant el seu input fent que s'adapti a cada situació.
En la Figura 6.3.1.1 podem veure els 3 esquemes que hem fet servir.

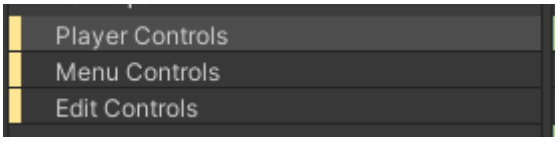


Figura 6.3.1.1: Esquemes de control del input

6.3.2. Input reader

A partir dels diferents esquemes mencionats en l'Apartat 6.2.1, hem definit un controlador de input que serà l'encarregat de rebre les accions efectuades pel jugador en cada moment. Un cop s'efectui una acció, aquest controlador la rebrà i llençarà un esdeveniment específic per aquesta acció donada. Cada una de les classes que necessitin alguna informació sobre el input del jugador, tindran una referència a aquest controlador, el qual és un ScriptableObject seguint amb l'estàndard del patró mencionat en l'Apartat 6.1.1.

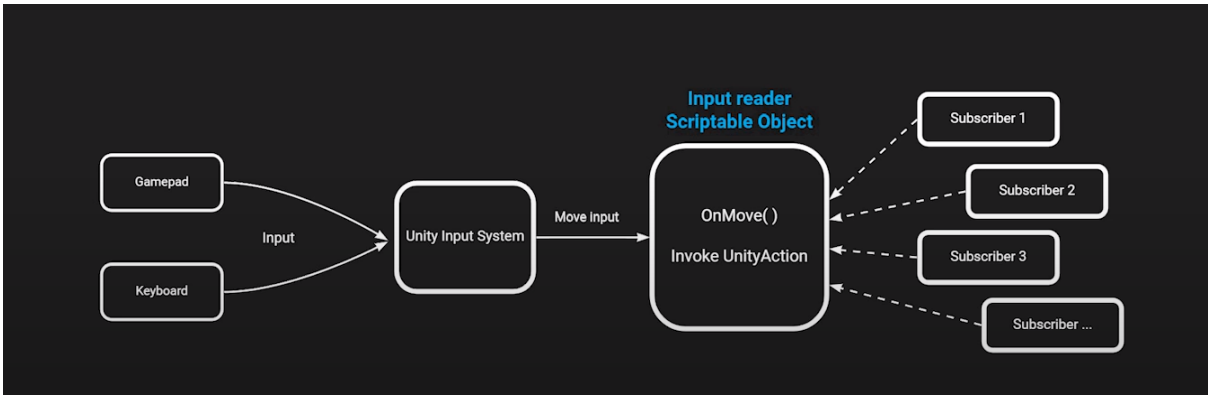


Figura 6.3.2.1: Esquemes de control del input

Model de la classe InputReader:

InputReader
+moveEvent: UnityAction<Vector2> +pauseMenuEvent: UnityAction +attackEvent: UnityAction +parryEvent: UnityAction +consumeEnergyButtonPressEvent: UnityAction +primaryInteractionEvent: UnityAction +secondaryInteractEvent: UnityAction +tertiaryInteractEvent: UnityAction +holdInteractEventStart: UnityAction +holdInteractEventCanceled: UnityAction +sprintEvent: UnityAction +usablesMenuEvent: UnityAction +useUsableEvent: UnityAction +plantMenuEvent: UnityAction +cancelEvent: UnityAction +upPositionEditMode: UnityAction +downPositionEditMode: UnityAction +leftPositionEditMode: UnityAction +rightPositionEditMode: UnityAction +snapPlaceable: UnityAction +placePlaceable: UnityAction +equipSword: UnityAction +equipAxe: UnityAction +equipPickaxe: UnityAction +playerStats: PlayerStats -gameInput: GameInput
-OnEnable(): void +OnAttack(InputAction.CallbackContext context): void +OnHoldInteraction(InputAction.CallbackContext context): void +OnTogglePause(InputAction.CallbackContext context): void +OnPressInteraction(InputAction.CallbackContext context): void +OnPressSecondaryInteraction(InputAction.CallbackContext context): void +OnPressTertiaryInteraction(InputAction.CallbackContext context): void +OnSprint(InputAction.CallbackContext context): void +OnUsablesMenu(InputAction.CallbackContext context): void +OnUseUsable(InputAction.CallbackContext context): void +OnParry(InputAction.CallbackContext context): void +OnConsumeEnergy(InputAction.CallbackContext context): void +OnPlantMenu(InputAction.CallbackContext context): void +OnCancel(InputAction.CallbackContext context): void +OnMovement(InputAction.CallbackContext context): void +OnEquipSword(InputAction.CallbackContext context): void +OnEquipAxe(InputAction.CallbackContext context): void +OnEquiPickaxe(InputAction.CallbackContext context): void +OnUp(InputAction.CallbackContext context): void +OnDown(InputAction.CallbackContext context): void +OnLeft(InputAction.CallbackContext context): void +OnRight(InputAction.CallbackContext context): void +OnSnapPlaceable(InputAction.CallbackContext context): void +OnPlace(InputAction.CallbackContext context): void +EnableGameplayInput(): void +EnableEditModeInput(): void +EnableUIInput(): void

+DisableAllInput(): void

Atributs de InputReader:

- **moveEvent**: delegat per l'esdeveniment de moviment
- **pauseMenuEvent**: delegat per l'esdeveniment d'activar o desactivar el menu de pausa
- **attackEvent**: delegat per l'esdeveniment d'atacar
- **parryEvent**: delegat per l'esdeveniment de retornar un atac
- **consumeEnergyButtonPressEvent**: delegat per l'esdeveniment de consumir energia
- **primaryInteractionEvent**: delegat per l'esdeveniment de prémer el botó primari d'interacció
- **secondaryInteractEvent**: delegat per l'esdeveniment de prémer el botó secundari d'interacció
- **tertiaryInteractEvent**: delegat per l'esdevenimentt de prémer el botó terciar d'interacció
- **holdInteractEventStart**: delegat per l'esdeveniment de començar a mantenir el botó primari d'interacció
- **holdInteractEventCanceled**: delegat per a la cancelació de l'esdeveniment de mantenir el botó primari d'interacció
- **sprintEvent**: delegat per l'esdeveniment de esprintar
- **usablesMenuEvent**: delegat per l'esdeveniment d'activar o desactivar el menu de utilitzables
- **useUsableEvent**: delegat per l'esdeveniment d'usar un utilitzable
- **plantMenuEvent**: delegat per l'esdevenimentt d'activar o desactivar el menu de plantació
- **cancelEvent**: delegat per l'esdeveniment de cancelar
- **upPositionEditMode**: delegat per l'esdeveniment de posicionar un colocable
- **downPositionEditMode**: delegat per l'esdeveniment de posicionar un colocable
- **leftPositionEditMode**: delegat per l'esdeveniment de posicionar un colocable
- **rightPositionEditMode**: delegat per l'esdeveniment de posicionar un colocable
- **snapPlaceable**: delegat per l'esdeveniment de posicionar un colocable enganxat a un altre
- **placePlaceable**: delegat per l'esdeveniment de colocar un colocable
- **equipSword**: delegat per l'esdeveniment d'equipar l'espasa
- **equipAxe**: delegat per l'esdeveniment d'equipar la destral

- **equipPickaxe**: delegat per l'esdeveniment d'equipar el pic
- **playerStats**: punter al ScriptableObject d'estadístiques del jugador
- **gameInput**: classe GameInput per rebre tots els callbacks del input

Mètodes de InputReader:

- **OnEnable**: Funció cridada pel Unity quan l'objecta s'activa. En aquesta funció ens asegurem que existeix un GameInput. Sii no és el cas, el creem i escoltem els Callbacks. Seguidament habilitem l'esquema d'input del jugador per defecte.

```
private void OnEnable(){
    if (gameInput == null){
        gameInput = new GameInput();
        gameInput.PlayerControls.SetCallbacks(this);
        gameInput.MenuControls.SetCallbacks(this);
        gameInput.EditControls.SetCallbacks(this);
    }
    EnableGameplayInput();
}
```

- **EnableGameplayInput, EnableEditModeInput, EnableUIInput, DisableAllInput** : Aquestes funcions activen o desactiven un esquema d'input en concret i totes tenen la mateixa estructura. A continuació veiem la estructura de EnableGameplayInput com a exemple.

```
public void EnableGameplayInput(){
    gameInput.PlayerControls.Enable(); gameInput.MenuControls.Disable();
    gameInput.EditControls.Disable();
}
```

- **OnAttack, OnTogglePause, OnPressSecondaryInteraction, OnPressTertiaryInteraction, OnSprint, OnUsablesMenu, OnParry, OnConsumeEnergy, OnPlantMenu, OnCancel, OnEquipSword, OnEquipAxe, OnEquiPickaxe, OnUp, OnDown, OnLeft, OnRight, OnSnapPlaceable, OnPlace**: Totes aquestes funcions tenen el mateix propòsit i estructura. Escolten els callbacks específics de l'input i, si l'acció s'ha completat, invoquen el delegat al que estan associats mentre hi hagi alguna classe escoltant. A continuació veiem l'estructura OnAttack com a exemple.

```
public void OnAttack(InputAction.CallbackContext context){
    if (context.phase == InputActionPhase.Performed) attackEvent?.Invoke();
}
```

- **OnHoldInteraction**: Funció encarregada d'invocar dos delegats, un quan comença l'acció i l'altre quan es cancela

```
public void OnHoldInteraction(InputAction.CallbackContext context){
    if (context.phase == InputActionPhase.Started) holdInteractEventStart?.Invoke();
    if (context.phase == InputActionPhase.Canceled) holdInteractEventCanceled?.Invoke();
}
```

- **OnUseUsable**: Funció encarregada d'invocar dos delegats depenent de l'estat del jugador. Un per fer servir un objecte utilitzable i l'altre per activar-lo si prèviament s'ha fet servir

```
public void OnUseUsable(InputAction.CallbackContext context){
    if (context.phase == InputActionPhase.Performed && !playerStats.isPlacingPlaceable)
            useUsableEvent?.Invoke();
    else if (context.phase == InputActionPhase.Performed
             && playerStats.isPlacingPlaceable)
            placePlaceable?.Invoke();
    }
```

- **OnMovement**: Funció encarregada d'invocar el delegat de moviment amb el valor de l'acció de moure que representa la direcció

```
public void OnMovement(InputAction.CallbackContext context){
    moveEvent?.Invoke(context.ReadValue<Vector2>());
}
```

6.4. Sistema de guardat

El sistema de guardat està format per 3 parts:

1. Classes d'informació. Aquestes classes són scripts bàsics, els quals només tenen atributs que seran els valors que necessitarem per guardar la informació del món (posicions, estats, temporitzadors...)
2. World Saver. Classe que portarà el control de tots els objectes que tenen estats variants i dinàmics i que es necessita guardar la situació actual en el món (recursos, época actual, inventari...)
3. Save System: Classe ScriptableObject encarregada tant de registrar com de llegir la informació dins dels fitxers de guardat.

L'objectiu principal d'aquesta estructura és poder ampliar-la tant com es vulgui en un futur desenvolupament del joc.

6.4.1. Classes d'informació

Hem creat un seguit de classes d'informació bàsiques per a poder guardar l'estat i la informació d'un tipus d'objecte donat. Aquestes classes només tenen atributs públics i un constructor, ja que serveixen com a classes d'informació. A més, totes han de tenir l'atribut System.Serializable per a que el Unity entengui que les pot llegir i serialitzar en altres estructures, com ara fitxers de text en el nostre cas.

- ObjectPositionData

```
[System.Serializable]
7 referencias
public class ObjectPositionData
{
    public string assetGUID;
    public Vector3 pos;
    public float yRotation;

    3 referencias
    public ObjectPositionData(string assetGUID, Vector3 pos, float yRotation)
    {
        this.assetGUID = assetGUID;
        this.pos = pos;
        this.yRotation = yRotation;
    }
}
```

Figura 6.4.1.1: Classe ObjectPositionData

Com podem veure en la Figura 6.4.1.1, la classe té 3 atributs:

- assetGUID: String Id de l'adreça del objecte que proporciona l'AddressableSystem mencionat en l'Apartat 6.1.3.
- pos: Vector3 de la posició actual en el món de l'objecte
- yRotation: float de la rotació en l'eix y, ja que en el joc no es podrà rotar en cap altre eix. Per tant només guardem el que tindrà un valor dinàmic.

A més tenim el constructor per crear cada objecte per cada element en el món. Tots els elements que requereixin guardar la posició i rotació, heretaran d'aquesta classe.

- SerializedItemStack

```
[System.Serializable]
12 referencias
public class SerializedItemStack
{
    public string itemGuid;
    public int amount;

    3 referencias
    public SerializedItemStack(string itemGuid, int amount)
    {
        this.itemGuid = itemGuid;
        this.amount = amount;
    }
}
```

Figura 6.4.1.2: Classe SerializedItemStack

Com podem veure en la Figura 6.4.1.2, la classe té 2 atributs:

- itemGUID: String Id de l'adreça de l'objecte que proporciona l'AddressableSystem mencionat en l'Apartat 6.1.3.
- amount: Int per determinar la quantitat actual de l'objecte

A més, tenim el constructor per crear cada objecte per cada element. Aquesta classe s'utilitza en els objectes que no existeixen en el món, com per exemple els items que té actualment el jugador a l'inventari.

- SeedData

```
[System.Serializable]
8 referencias
public class SeedData : ObjectPositionData
{
    public string seedGUID;
    public int actualGrowthLevel;
    public float timeGrowing;
    public int timesRecolected;
    public float timeGrowingNextrecolect;
    public bool canRecolect;

    2 referencias
    public SeedData(string assetGUID, Vector3 pos, float yRotation, string seedGUID, int actualGrowthLevel,
        float timeGrowing, int timesRecolected, float timeGrowingNextRecolect, bool canRecolect )
        : base(assetGUID, pos, yRotation)
    {
        this.assetGUID = assetGUID;
        this.pos = pos;
        this.yRotation = yRotation;
        this.seedGUID = seedGUID;
        this.actualGrowthLevel = actualGrowthLevel;
        this.timeGrowing = timeGrowing;
        this.timesRecolected = timesRecolected;
        this.timeGrowingNextrecolect = timeGrowingNextRecolect;
        this.canRecolect = canRecolect;
    }
}
```

Figura 6.4.1.3: Classe SerializedItemStack

Com podem veure en la Figura 6.4.1.3, la classe té 6 atributs:

- seedGUID: String Id de l'adreça de l'objecte que proporciona l'AddressableSystem mencionat en l'apartat 6.1.3.
- actualGrowthLevel: Int per guardar el nivell de creixement de la llavor
- timeGrowing: Float per guardar el temps que porta creixent la llavor
- timesRecolected: Int per guardar quants cops s'ha recol·lectat la planta
- timesGrowingNextRecolection: Float per guardar el temps que porta creixent la següent recol·lecció del fruit
- canRecolect: Bool per determinar si la la planta o fruit pot ser collita

A més tenim el constructor per crear cada objecte per cada element. Aquesta classe s'utilitza en els objectes llavors per emmagatzemar l'estat actual.

- SeedPotData

```
public class SeedPotData : ObjectPositionData
{
    public SeedData seedData = null;
    2 referencias
    public SeedPotData(string assetGUID, Vector3 pos,
        float yRotation, SeedData seedData = null) : base(assetGUID, pos, yRotation){
        this.assetGUID = assetGUID;
        this.pos = pos;
        this.yRotation = yRotation;
        this.seedData = seedData;
    }
}
```

Figura 6.4.1.4: Classe SeedPotData

Com podem veure en la Figura 6.4.1.4, la classe té un atribut que és de classe SeedData. Tots els altres atributs que necessita venen per l'herència de la classe ObjectPositionData. Aquesta classe s'utilitza en els objectes de classe test en els que hem de guardar la posició i, alhora la informació de la llavor que contenen.

- InventoryData

```
[System.Serializable]

1 referencia
public class InventoryData
{
    public List<SerializedItemStack> playerResourcesInventory = new List<SerializedItemStack>();
    public List<SerializedItemStack> globalResourcesInventory = new List<SerializedItemStack>();
    public List<SerializedItemStack> globalUsablesInventory = new List<SerializedItemStack>();

    1 referencia
    public void ResetData()
    {
        playerResourcesInventory.Clear();
        globalResourcesInventory.Clear();
        globalUsablesInventory.Clear();
    }
}
```

Figura 6.4.1.5: Classe InventoryData

Com podem veure en la Figura 6.4.1.5, la classe té una llista d'objectes de tipus SerializedItemStack per a cada inventari que té el jugador. A més té una funció per a eliminar la informació donada.

- IslandData

```
[System.Serializable]
4 referencias
public class IslandData
{
    public int IslandID = -1;
    public bool globalInventoryAcces = false;
    public List<ObjectPositionData> dynamicObjects = new List<ObjectPositionData>();
    public List<SeedData> dynamicSeeds = new List<SeedData>();
    public List<SeedPotData> dynamicSeedPot = new List<SeedPotData>();
}
```

Figura 6.4.1.6: Classe IslandData

Com podem veure en la Figura 6.4.1.6, la classe té 5 atributs:

- IslandId: Int per guardar l'identificador de l'illa. Aquest identificador el posem a mà per cada illa, per així poder saber en quina illa hem de carregar la informació donada.
- globalInventoryAcces: Bool per determinar si el jugador té accés a l'inventari global dins l'illa.
- dynamicObjects: Llista d'objectes de tipus ObjectpositionData per guardar la informació de tots els objectes dinàmics dins l'illa, majoritàriament els recursos que el jugador podrà recolectar.
- dynamicSeeds: Llistat d'objectes de tipus SeedData per guardar la informació de tots els objectes llavors dinàmics de l'illa, majoritàriament les llavors dels arbres.
- dynamicSeedPot: Llistat d'objectes de tipus SeedPotData per guardar la informació dels objectes tests dinàmics de l'illa.

- Save

```
[Serializable]
1 referencia
public class Save
{
    public InventoryData inventoryData = new InventoryData();
    public List<IslandData> upperWorldislandsData = new List<IslandData>();
    public List<int> upperWorldBuildingsStates = new List<int>();

    public int WorldAge = 0;

    public Vector3 PlayerPosition = Vector3.one  * -1;
    public float PlayerRotation = 0;
    public int PlayerCurrentLife = 0;
    public int PlayerCurrentEnergy = 0;

    1 referencia
    public string ToJson()
    {
        return JsonUtility.ToJson(this);
    }

    1 referencia
    public void LoadFromJson(string json)
    {
        JsonUtility.FromJsonOverwrite(json, this);
    }
}
```

Figura 6.4.1.7: Classe Save

Com podem veure en la Figura 6.4.1.7, la classe té 8 atributs:

- InventoryData: variable de tipus InventoryData on s'emmagatzema tota la infromació de l'inventari del jugador
- upperWorldIslandsData: Llistat d'objectes de tipus IslandData que correspon a la informació de totes les illes de l'escena UpperWorld.
- upperWorldBuildingStates: Llistat d'enters per cada una de les estructures reparables de l'escena, per saber si s'ha reparat o no.
- WorldAge: int de l'época actual de la partida.
- PlayerPosition:Vector3 de la posició actual del jugador.
- PlayerRotation: float de rotació en l'eix y del jugador.
- PlayerCurrentLife: int de la vida actual del jugador.
- PlayerCurrentEnergy: int de la energía actual del jugador.

A més tenim les funcions ToJson, encarregada de retornar la classe amb tots els atributs públics formatats a JSON, i LoadFromJson, encarregada de construir una instància de classe a partir d'un string JSON.

Aquesta classe és la que conté tota la informació que es guardarà a través de les classes de dades mencionades anteriorment en aquest apartat.

6.4.2. World Saver

World Saver és la classe encarregada tant de guardar tota la informació dinàmica de l'estat actual de l'escena UpperWorld, com de carregar la informació de dins dels arxius de guardat i portar a l'escena al mateix estat.

Model de la classe WorldSaver:

World Saver
-saveSystem: SaveSystem -playerStats: PlayerStats -islandNode: GameObject -age: AgeManager -spawnSystem: SpawnSystem -inventoryManager: InventroyManager -buildings: List<World_BuildingToRepair> -onSeceneReadyEvent: VoidEventSO -saveGameEvent: VoidEventSO -loadGameEvent: VoidEventSO -updateUIAllSlotsUsables: VoidEventSO -playerTransform: TransformAnchor -loadWordlOnlyEditor: VoidEventSO -onSceneLoadingComplete: VoidEventSO -loadGame: bool -inventoryLoaded: bool
-OnEnable(): void -OnDisable(): void -ToggleLoadGame(): void -Initialization(): void +Save(): void +Load(): void -SaveBuildings(): void -SaveIslands(): void -SaveInventory(): void -LoadIslands(): void -Loadbuildings(): void -OnSceneLoaded(): IEnumerator -LoadInventory(): IEnumerator

Atributs de WorldSaver:

- **saveSystem**: Classe de dades per guardar tota la informació
- **playerStats**: ScriptableObject on hi ha la informació actual del jugador
- **islandNode**: GameObject de l'escena que té com a fills tots els objecte illes del món
- **age**: ScriptableObject on hi ha la informació de l'época actual
- **spawnSystem**: Objecte de la classe SpawnSystem encarregat de instanciar al jugador
- **inventoryManager**:ScriptableObject encarregat de la gestió del inventari del jugador
- **buildings**: Llista d'objectes de la clase World_BuildingToRepair per a poder guardar els estats
- **onSceneReadyEvent**: Objecte VoidEventSO per l'esdeveniment de escena llesta
- **saveGameEvent**:Objecte VoidEventSO per l'esdeveniment de guardar l'estat de la partida
- **loadGameEvent**:Objecte VoidEventSO per l'esdeveniment de carregar l'estat de la partida
- **updateUIAllSlotsUsables**:Objecte VoidEventSO per l'esdeveniment de re-pintar els objectes al panell del inventari d'objectes usables.
- **playerTransform**: Objecte TransformAnchor on s'hi emmagatzema la transformació del jugador
- **loadWorldOnlyEditor**:Objecte VoidEventSO per l'esdeveniment de carregar l'estat de la partida només en el editor
- **onSceneLoadingComplete**::Objecte VoidEventSO per l'esdeveniment de carregat d'escena completat
- **loadGame**: Bool per determinar si la partida s'ha de carregar o no
- **inventoryLoaded**: Bool per determinar si l'inventari del jugador ja ha estat carregat

Mètodes de WorldSaver:

- **OnEnable**, **OnDisable**: Funcions cridades per el Unity quan l'objecte s'activa i es desactiva. Utilitzades per subscriure's i desubscriure's als esdeveniments.

```
private void OnEnable(){
    loadGameEvent.OnEventRaised += ToggleLoadGame;
    saveGameEvent.OnEventRaised += Save;
    onSceneReadyEvent.OnEventRaised += Initialization;
    loadWordlOnlyEditor.OnEventRaised += Load;
}
```

```
private void OnDisable(){
    loadGameEvent.OnEventRaised -= ToggleLoadGame;
    saveGameEvent.OnEventRaised -= Save;
    onSceneReadyEvent.OnEventRaised -= Initialization;
    loadWordlOnlyEditor.OnEventRaised -= Load;
}
```

- **Initialization:** Funció encarregada de cridar a la funció de càrrega o de spawnejar al jugador en el punt per defecte i notificar que l'escena està llesta

```
private void Initialization(){
    if (loadGame) Load();
    else{ spawnSystem.Load(0); onSceneLoadingComplete.RaiseEvent(); }
}
```

- **Save:** Funció encarregada de cridar a totes les funcions de guardat de les diferents estructures

```
public void Save(){
    SaveIslands(); SaveBuildings(); SaveInventory();
    saveSystem.saveData.WorldAge = age.Age.AgeID;
    saveSystem.saveData.PlayerPosition = playerTransform.Transform.position;
    saveSystem.saveData.PlayerRotation =playerTransform.Transform.eulerAngles.y;
    saveSystem.saveData.PlayerCurrentLife = playerStats.currentlLifes;
    saveSystem.SaveDataToDisk();
}
```

- **Load:** Funció encarregada de cridar a totes les funcions de càrrega de les diferents estructures

```
public void Load(){
    saveSystem.LoadSaveDataFromDisk(); age.Load(saveSystem.saveData.WorldAge);
    playerStats.currentlLifes = saveSystem.saveData.PlayerCurrentLife;
    spawnSystem.Load(saveSystem.saveData.PlayerPosition,
                     saveSystem.saveData.PlayerRotation);
    LoadIslands(); LoadBuildings();
    StartCoroutine(LoadInventory()); StartCoroutine(OnSceneLoaded());
}
```

- **SaveBuildings:** Funció encarregada de guardar l'estat del edificis reparables del joc

```
private void SaveBuildings(){
        saveSystem.saveData.upperWorldBuildingsStates.Clear();
        foreach (var build in buildings){
        if (build.IsRepaired()) saveSystem.saveData.upperWorldBuildingsStates.Add(1);
        else saveSystem.saveData.upperWorldBuildingsStates.Add(0);
    }
}
```

- **SaveIslands:** Funció encarregada de guardar l'estat de cada illa filla del GameObject islandNode

```
void SaveIslands(){
    saveSystem.saveData.upperWorldislandsData.Clear();
    foreach (Transform transform in islandNode.transform){
        Island island = transform.GetComponent<Island>();
        if (island){
           island.Save();
           saveSystem.saveData.upperWorldislandsData.Add(island.islandData);
        }
    }
}
```

- **SaveInventory**: Funció encarregada de guardar l'estat de l'inventari del jugador

```
void SaveInventory(){
    saveSystem.saveData.inventoryData.ResetData();
    foreach (InventorySlot item in inventoryManager.PlayerResourcesInventory())
        if (item.item != null)
            saveSystem.saveData.inventoryData.playerResourcesInventory
            .Add(new SerializedItemStack(item.item.Guid, item.amount));
        foreach (InventorySlot item in inventoryManager.GlobalResourcesInventory())
            if (item.item != null)
                saveSystem.saveData.inventoryData.globalResourcesInventory
                .Add(new SerializedItemStack(item.item.Guid, item.amount));
        foreach (InventorySlot item in inventoryManager.GlobalUsablesInventory())
            if (item.item != null)
                saveSystem.saveData.inventoryData.globalUsablesInventory
                .Add(new SerializedItemStack(item.item.Guid, item.amount));
    }
```

- **LoadIslands**: Funció encarregada de carregar la informació de cada illa

```
private void LoadIslands(){
    foreach (Transform transform in this.transform){
        Island island = transform.GetComponent<Island>();
        if (island){
            int i = 0;
            while (i < saveSystem.saveData.upperWorldislandsData.Count && island.GetID()
                != saveSystem.saveData.upperWorldislandsData[i].IslandID) i++;
                if (i < saveSystem.saveData.upperWorldislandsData.Count)
                 island.Load(saveSystem.saveData.upperWorldislandsData[i]);
            }
        }
    }
```

- **LoadBuildings**: Funció encarregada de carregar la informació de cada edifici reparable

```
void LoadBuildings(){
    int i = 0;
    foreach (var item in buildings){
        if (i < saveSystem.saveData.upperWorldBuildingsStates.Count){
            item.Load(saveSystem.saveData.upperWorldBuildingsStates[i]); i++;
        }
    }
}
```

- **OnSceneLoaded**: Corrutina que espera a que l'inventari del jugador s'ha carregat per notificar-ho

```
IEnumerator OnSceneLoaded(){
    yield return !inventoryLoaded; yield return new WaitForSeconds(0.5f);
    onSceneLoadingComplete.RaiseEvent();
}
```

- **LoadInventory**: Corrutina que carrega l'estat de l'inventari del jugador, ho notifica a la interfície i posa l'estat de la variable inventoryLoaded a cert.

```
IEnumerator LoadInventory(){
    inventoryManager.ClearAll();
    foreach (SerializedItemStack item in
             saveSystem.saveData.inventoryData.playerResourcesInventory){
        var loadItemOperationHandle =
            Addressables.LoadAssetAsync<ItemBase>(item.itemGuid);
        yield return loadItemOperationHandle;
        if (loadItemOperationHandle.Status == AsyncOperationStatus.Succeeded){
            var itemSo = loadItemOperationHandle.Result;
            inventoryManager.PlayerResourcesInventory()
                .Add(new InventorySlot( itemSo ,item.amount));
        }
    }
    foreach (SerializedItemStack item in
             saveSystem.saveData.inventoryData.globalResourcesInventory)
        ...
    foreach (SerializedItemStack item in
             saveSystem.saveData.inventoryData.globalUsablesInventory)
        ...
    updateUIAllSlotsUsables.RaiseEvent();
    inventoryLoaded = true;
}
```

6.4.3. Save System

La classe Save System és un ScriptableObject encarregat de llegir o guardar en el fitxer tota la informació de la classe Save. La classe simplement té 3 atributs:

- **saveFileName**: string pel arxiu principal de guardat
- **backupSaveFileName**: string per a un possible arxiu secundari per seguretat
- **saveData**: clase Save on hi ha tota la informació a guardar

A més té 3 funcions:

- **SaveDataToDisk**: void per escriure la informació en un fitxer de text
- **LoadDataFromDisk**: void per llegir la informació del fitxer de text de guardat i passar-la en format json a la classe Save.
- **WriteEmptySaveFile**: void per eliminar tots els arxius de guardat

```
public class SaveSystem : ScriptableObject
{
    public string saveFilename = "save.data";
    public string backupSaveFilename = "save.data.bak";
    public Save saveData = new Save();

    1 referencia
    public void SaveDataToDisk()
    {
        if (FileManager.MoveFile(saveFilename, backupSaveFilename))
        {
            if (FileManager.WriteToFile(saveFilename, saveData.ToJson()))
                {
                 Debug.Log("Save successful");
            }
        }
    }

    1 referencia
    public bool LoadSaveDataFromDisk()
    {
        if (FileManager.LoadFromFile(saveFilename, out var json))
        {
            saveData.LoadFromJson(json);
            return true;
        }
        WriteEmptySaveFile();
        return false;
    }

    [ContextMenu("Remove Save")]
    1 referencia
    public void WriteEmptySaveFile()
    {
        FileManager.WriteToFile(saveFilename, "");
        FileManager.WriteToFile(backupSaveFilename, "");
    }
}
```

Figura 6.4.3.1: A la Figura podem veure la classe SaveSystem

6.5. Controladors del jugador

Per portar el control de les mecàniques que pot fer el jugador i de com interactua amb el món, hem dividit les diferents mecàniques del joc en diferents controladors que tenen una funció específica. D'aquesta manera, aïllem components i millorem la llegibilitat del codi.
Adicionalment, l'objecte jugador esta composat per diferents parts que són les encarregades d'interactuar amb el món, com la pròpia malla, partícules o diferents colisionadors. En la Figura 6.5.1 es pot veure l'esquema de la composició mencionada.

Figura 6.5.1: Estructura de l'objecte Player

6.5.1. Player Controler

Controlador base encarregat d'inicialitzar els altres controladors del jugador. A més, és l'encarregat d'escoltar els inputs base del jugador i enviar la informació als altres components.

Model de la classe PlayerControler:

Player Controler
-inputReader: InputReader -playerStats: PlayerStats -playerMovementBehaviour: PlayerMovementBehaviour -playerAnimationtBehaviour: PlayerAnimationtBehaviour -playerCombatController: PlayerCombatController +movementSmoothingSpeed: float -rawInputMovement: Vector3 -smoothInputMovement: Vector3 -playerDie: VoidEventSO -restartGame: VoidEventSO +walkingParticles: ParticleSystem
-OnEnable(): void -OnDisable(): void

+PlayerHit(int damage)t: bool
+SetUpPlayer(): void
+OnMove(Vector2 movement): void
+OnAttack(): void
+OnParry(): void
+OnConsumeEnergyButtonPress(): void
-Update(): void
-StopMovement(): void
-CalculateMovementInputSmoothing(): void
-UpdatePlayerMovement(): void
-UpdatePlayerAnimationMovement(): void
-StartAttack(): void
-FinishAttack(): void
-FinishParry(): void
-Die(): void
-RestartGameOnDie(): IEnumerator

Atributs de PlayerControler:

- **inputReader**: ScriptableObject encarregat de llençar els esdeveniments de l'input
- **playerStats**: ScriptableObject on hi ha la informació actual del jugador
- **playerMovementBehaviour**: Referència al component de la classe PlayerMovementBehaviour encarregat de portar el control del moviment del jugador
- **playerAnimationBehaviour**: Referència al component de la classe PlayerAnimationBehaviour encarregat de portar el control de les animacions del jugador
- **playerCombatController**: Referència al component de la classe PlayerCombatController encarregat de portar el control del combat
+ **movementSmoothingSpeed**: Float per determinar la fluïdesa de la rotació del objecte en el moviment del jugador
- **rawInputMovement**:Vector3 per emmagatzemar el valor del moviment que arriba des de l'inputReader
- **smoothInputMovement**:Vector3 per emmagatzemar el valor del moviment final, després de modificar el valor del moviment per defecte
- **playerDie**: Objecte VoidEventSO per l'esdeveniment de mort del jugador
- **restarGame**: Objecte VoidEventSO per l'esdeveniment de reinici del joc
- **walkingParticles**: ParticleSystem assignades al moviment del jugador

Mètodes de PlayerControler:

- **OnEnable, OnDisable**:Funcions cridades per el Unity quan l'objecte s'activa i es desactiva. Utilitzades per subscriure's i desubscriure's als esdeveniments, e inicialitzar dependències.

```
private void OnEnable(){
    playerDie.OnEventRaised += Die; _inputReader.moveEvent += OnMove;
    _inputReader.attackEvent += OnAttack; _inputReader.parryEvent += OnParry;
    _inputReader.sprintEvent += OnSprint;
    _inputReader.consumeEnergyButtonPressEvent += OnConsumeEnergyButtonPress;
    playerAnimationBehaviour = GetComponent<PlayerAnimationBehaviour>();
    playerCombatController = GetComponent<PlayerCombatController>();
    playerMovementBehaviour = GetComponent<PlayerMovementBehaviour>();
}

private void OnDisable(){
    playerDie.OnEventRaised -= Die; _inputReader.moveEvent -= OnMove;
    _inputReader.attackEvent -= OnAttack; _inputReader.parryEvent -= OnParry;
    _inputReader.sprintEvent -= OnSprint;
    _inputReader.consumeEnergyButtonPressEvent -= OnConsumeEnergyButtonPress;
}
```

+ **PlayerHit**: Funció amb damage com a paràmetre d'entrada. Encarregada d'enviar l'informació al controlador del combat i, si s'ha pogut atacar al jugador, avisar al controlador d'animacions. Retorna cert si s'ha pogut atacar al jugador, altrament fals.

```
public bool PlayerHit(int damage){
    if (playerCombatController.PlayerHit(damage)) {
        playerAnimationBehaviour.PlayHitAnimation();
        return true;
    }
    return false;
}
```

+ **SetUpPlayer**: Funció encarregada de inicialitzar els diferents controladors i reiniciar les estadístiques globals del jugador.

```
public void SetupPlayer(){
    _playerStats.Reset(); playerAnimationBehaviour.SetupBehaviour();
    playerCombatController.SetBarsUIStatusBars();
}
```

- **OnMove, StopMovement:** Funcions encarregades d'escoltar els esdeveniments d'input. OnMove emmagatzema el valor del moviment en una variable de classe. StopMovement modifica el valor del moviment actual a zero.

```
public void OnMove(Vector2 movement){
    rawInputMovement = new Vector3(movement.x, 0, movement.y);
}

void StopMovement(){
    smoothInputMovement = Vector3.zero;
    playerMovementBehaviour.UpdateMovementData(smoothInputMovement);
    playerAnimationBehaviour.UpdateMovementAnimation(smoothInputMovement.magnitude);
}
```

+ **OnAttack, OnParry:** Funcions encarregades d'escoltar els esdeveniments d'input i notificar als altres controladors si es pot efectuar l'acció.

```
public void OnAttack(){
    if (_playerStats.isAttacking == false && _playerStats.isParring == false){
        StopMovement(); playerAnimationBehaviour.PlayAttackAnimation();
        _playerStats.isAttacking = true;
    }
}

public void OnParry(){
    if (_playerStats.isParring == false && _playerStats.isAttacking == false){
        StopMovement(); playerAnimationBehaviour.PlayParryAnimation();
        _playerStats.isParring = true;
    }
}
```

- **Update**: Funció del Unity cridada cada frame i encarregada de portar el control del moviment del jugador, si es pot moure.

```
void Update(){
    if (!_playerStats.isDead && !_playerStats.isAttacking && !_playerStats.isParring){
        CalculateMovementInputSmoothing();
        UpdatePlayerMovement(); UpdatePlayerAnimationMovement();
    }
}
```

- **CalculateMovementInputSmoothing, UpdatePlayerMovement, UpdatePlayerAnimationMovement**:

  Funcions encarregades de tractar el valor del moviment.

```
void CalculateMovementInputSmoothing(){
    smoothInputMovement = Vector3.Lerp(smoothInputMovement, rawInputMovement,
                                       Time.deltaTime * movementSmoothingSpeed);
    }

void UpdatePlayerMovement(){
    playerMovementBehaviour.UpdateMovementData(smoothInputMovement);
    if (rawInputMovement.magnitude == Vector3.zero.magnitude) walkingParticles.Stop();
    else if (!walkingParticles.isPlaying) walkingParticles.Play();
}

void UpdatePlayerAnimationMovement(){
    playerAnimationBehaviour.UpdateMovementAnimation(smoothInputMovement.magnitude);
}
```

+ **StartAttack**, **FinishAttack**, **FinishParry**: Funcions cridades per events d'animacions. Encarregades de modificar l'estat del jugador depenent del moment de l'acció. Utilitzem aquestes funcions per poder afinar el feedback al jugador i evitar cops duplicats.

```
    public void StartAttack(){ _playerStats.canHit = true; }

    public void FinishAttack() {
        _playerStats.isAttacking = false; _playerStats.canHit = false;
    }

    public void FinishParry() { _playerStats.isParring = false; }
```

+ **Die**, **RestartGameOnDie**: Funcions per a la mort del jugador. Die s'encarrega d'activar l'animació de mort. RestartGameOnDie és una corrutina que espera uns segons a que s'hagi completat l'animació i, seguidament llença l'esdeveniment per a tornar a carregar el joc.

```
public void Die(){
    StopMovement(); playerAnimationBehaviour.PlayerDieAnimation();
    StartCoroutine(RestartGameOnDie());
}

IEnumerator RestartGameOnDie(){
    yield return new WaitForSeconds(2); restartGame.RaiseEvent();
}
```

6.5.2. Player Movement Behaviour

Classe simple encarregada de moure el component RigidBody del jugador per el món.

```
public Rigidbody playerRigidbody;
public float movementSpeed = 3f;
public float turnSpeed = 0.1f;
[SerializeField] private TransformAnchor _cameraTransformAnchor = default;
private Vector3 movementDirection;
```

Figura 6.5.2.1: Atributs de la classe PlayerMovementBehaviour

Com es pot veure en la Figura 6.5.2.1, la classe té simplement una referència al component RigidBody del jugador, 2 variables per a poder modificar la manera de moure l'objecte, la transformació actual de la càmera i una variable privada per emmagatzemar la direcció actual.

```
void MoveThePlayer()
{
    Vector3 movement = CameraDirection(movementDirection) * movementSpeed * Time.deltaTime;
    playerRigidbody.MovePosition(transform.position + movement);
}
1 referencia
void TurnThePlayer()
{
    if(movementDirection.sqrMagnitude > 0.01f)
    {
        Quaternion rotation = Quaternion.Slerp(playerRigidbody.rotation,
            Quaternion.LookRotation (CameraDirection(movementDirection)),turnSpeed);
       playerRigidbody.MoveRotation(rotation);
    }
}
2 referencias
Vector3 CameraDirection(Vector3 movementDirection)
{
    var cameraForward = _cameraTransformAnchor.Transform.forward;
    var cameraRight = _cameraTransformAnchor.Transform.right;
    cameraForward.y = 0f; cameraRight.y = 0f;
    return cameraForward * movementDirection.z + cameraRight * movementDirection.x;
}
```

Figura 6.5.2.2: Mètodes de la classe PlayerMovementBehaviour

En la Figura 6.5.2.2 podem veure els 3 mètodes utilitzats per moure al jugador. CameraDirection és un mètode que retorna un Vector3 amb la direcció actual del moviment depenent de la direcció de la càmera. Hem hagut d'implementar aquest mètode per culpa de la peculiar posició de la càmera del nostre joc. MoveThePlayer i TurnThePlayer són funcions que es criden a cada fotograma. MoveThePlayer mou el RigidBody del jugador amb el moviment actual i TurnThePlayer simplement el rota amb la rotació del moviment actual.

6.5.3. Player Animation Behaviour

Classe encarregada únicament d'activar cada animació del jugador.

```
[Header("Component References")]
public Animator playerAnimator;

private int playerMovementAnimationID;
private int playerAttackAnimationID;
private int playerPrryAnimationID;
private int playerTakeDamageID;
private int playerDeathID;
```

Figura 6.5.3.1: Atributs de la classe PlayerAnimationBehaviour

En la Figura 6.5.3.1, podem veure els atributs de la classe, que són una referencia al component Animator assignat l'objecte del jugador, i un enter per cada animació que tingui.
El funcionament és molt senzill: tenim una funció inicialitzadora encarregada d'emmagatzemar en les variables privades l'identificador de cada animació disponible. A més, una funció per cada animació que, simplement, activa l'animació específica. A continuació en la Figura 6.5.3.2 podem veure la funció inicialitzadora i, seguidament en la Figura 6.5.3.3 podem veure un exemple de funció activadora d'animacions.

```
void SetupAnimationIDs()
{
    playerMovementAnimationID = Animator.StringToHash("Movement");
    playerAttackAnimationID = Animator.StringToHash("Attack");
    playerPrryAnimationID = Animator.StringToHash("Parry");
    playerTakeDamageID = Animator.StringToHash("TakeDamage");
    playerDeathID = Animator.StringToHash("Death");
}
```

Figura 6.5.3.2: Mètode inicialitzador de la classe PlayerAnimatorBehaviour

```
public void PlayAttackAnimation()
{
    playerAnimator.SetTrigger(playerAttackAnimationID);
}
```

Figura 6.5.3.3: Mètode PlayAttackAnimation de la classe PlayerAnimatorBehaviour

Totes les funcions activadores tenen la mateixa estructura que en la figura 6.1.20, però activan una animació en concret.

6.5.4. Player Items Behaviour

Classe simple encarregada d'obtenir el punt vàlid per a la col·locació d'objectes.

```
[SerializeField] public GameObjectEventSO _placeObjectEvent = default;
[SerializeField] public GameObject placeObjectPoint;
[SerializeField] public float groundCheckerDistance;
[SerializeField] public LayerMask whatIsGround;
```

Figura 6.5.4.1: Atributs de la classe PlayerItemsBehaviour

Com es pot veure en la Figura 6.5.4.1, la classe té un atribut GameObjectEventSO per escoltar a l'esdeveniment de col·locar un objecte. A més, placeObjectPoint és simplement un objecte buit que serveix per determinar el punt de col·locació d'objectes (davant del jugador).GroundCheckerDistance és la distància del raig que es llença des de placeObjectPoint cap a vall per obtenir el punt de col·lisió amb el terra. WhatIsGround és la capa on està el terra de l'escena.

```
void PlaceObject(GameObject prefab)
{
    Vector3 pos = PlacePoint();
    if (pos != Vector3.zero)
        Instantiate(prefab, pos, Quaternion.identity);
}
4 referencias
public Vector3 PlacePoint()
{
    RaycastHit hit;
    if (Physics.Raycast(placeObjectPoint.transform.position, Vector3.down, out hit,
        groundCheckerDistance, whatIsGround))
        return new Vector3(hit.point.x, hit.point.y, hit.point.z);
    return Vector3.zero;
}
```

Figura 6.5.4.2: Mètodes de la classe PlayerItemsBehaviour

Per últim, tenim els 2 mètodes de la classe que es poden veure en la figura 6.5.4.2. PlaceObject és la funció encarregada d'instanciar l'objecte que li arriba des de l'esdeveniment placeObjectEvent en el punt de col·locació. PlacePoint és la funció encarregada de retornar un Vector3 amb la posició del punt de col·locació, utilitzant un raig des del objecte placeObjectPoint cap a vall i comprova quan col·lisiona amb la capa del terra.

6.5.5. Player Seed Planter

Classe encarregada de gestionar la plantació de llavors.

Model de la classe PlayerSeedPlanter:

Player Seed Planter
-inputReader: InputReader -playerStats: PlayerStats -seedPlaceHolder: AssetReference -toggleSeedPanel: VoidEventSO -seedToPlantEvent: SeedEventSO -seedPotForPlanting: GameObjectSO -inventoryManager: InventoryManager -interactingSeedPot: SeedPot
-OnEnable(): void -OnDisable(): void +GetInteractingSeedPot(GameObject seedPot): void +PlantSeed(SeedSo seed): void

Atributs de PlayerSeedPlanter:

- **inputReader**: ScriptableObject encarregat de llençar els esdeveniments de l'input
- **playerStats**: ScriptableObject on hi ha la informació actual del jugador
- **seedPlaceHolder**: Referència a l'objecte seedPlaceHolder encarregat de gestionar la llavor que es planta.
- **toggleSeedPanel**: Esdeveniment encarregat d'activar o desactivar el panell de les llavors disponibles
- **seedToPlantEvent:** Esdeveniment encarregat de enviar la llavor que es vol plantar
- **seedPotForPlanting**: Esdeveniment encarregat d'enviar l'objecte del test amb el que el jugador està interaccionant
- **inventoryManager**: ScriptableObject encarregat de gestionar l'inventari del jugador
- **interactingSeedPot**: Atribut de la classe SeedPot per emmagatzemar la referència al test amb el que el jugador està interaccionant.

Mètodes de PlayerSeedPlanter:

- **OnEnable, OnDisable**:Funcions cridades per el Unity quan l'objecte s'activa i es desactiva. Utilitzades per subscriure's i desubscriure's als esdeveniments.

```
private void OnEnable(){
    seedPotForPlanting.OnEventRaised += GetInteractingSeedPot;
    SeedToPlantEvent.OnEventRaised += PlantSeed;
}

private void OnDisable(){
    seedPotForPlanting.OnEventRaised -= GetInteractingSeedPot;
    SeedToPlantEvent.OnEventRaised -= PlantSeed;
}
```

- **GetInteractingSeedPot**: Funció que rep el objecte test amb el que el jugador està interaccionant i guarda la referència.

```
public void GetInteractingSeedPot(GameObject seedPot){
    interactingSeedPot = seedPot.GetComponent<SeedPot>();
}
```

- **PlantSeed**: Funció que rep la llavor a plantar. Depenent de si la llavor es pot plantar al terra o no, actuarà d'una manera o una altre. Si la llavor no es pot plantar a terra, mirem si estem interactúan amb un test i si en aquest test, es pot plantar una llavor. Si és així, simplement la plantem al test, amaguem la pantalla de llavors, eliminem la llavor del inventari i activem l'esquema gameplay de l'input.

  Si només es pot plantar a terra, primer obtindrem el punt de plantament. Seguidament comprovem si col·lisiona amb algun objecte que no sigui *attrezzo* i, si és així, la plantarem. Per plantar-la, instanciem un objecte SeedPlaceholder i li plantem la llavor. A més, eliminem tots els objectes d'*attrezzo* col·lisionats.

```
public void PlantSeed(SeedSO seed){
    List<ItemContainer> items = new List<ItemContainer>();
    items.Add(new ItemContainer(seed, 1));
    if (seed.plantOnlyInSeedPot){
        if (interactingSeedPot != null && interactingSeedPot.CanPlant()){
            interactingSeedPot.Plant(seed); interactingSeedPot = null;
            inventoryManager.RemoveItems(items); ToggleSeedPanel.RaiseEvent();
            inputReader.EnableGameplayInput(); return;
        }
    }
    else if (seed.plantOnlyInFloor){
        Vector3 pos = this.GetComponent<PlayerItemsBehaviour>().PlacePoint();
        if (pos != Vector3.zero){
            Collider[] hitColliders = Physics.OverlapSphere(pos, 1);
            foreach (var hits in hitColliders){
                if (!hits.gameObject.CompareTag("Untagged")
```

```
                    && !hits.gameObject.CompareTag("Player")
                    && !hits.gameObject.CompareTag("Island")
                   && !hits.CompareTag("Atrezzo"))
                    return;
            }
            foreach (var hits in hitColliders) {
                if (hits.CompareTag("Atrezzo"))
                   if (!Addressables.ReleaseInstance(hits.gameObject))
                       Destroy(hits.gameObject);
            }
            Addressables.InstantiateAsync(seedPlaceHolder, pos, Quaternion.identity,
                playerStats.currentIsland.islandDynamicObjects.transform)
                .Completed += (obj) => { obj.Result.GetComponent<Seed>().Plant(seed);
            };
            inventoryManager.RemoveItems(items); ToggleSeedPanel.RaiseEvent();
            inputReader.EnableGameplayInput();
        }
    }
}
```

6.5.6. Player Tools Controller

Classe per portar el control de les eines. El funcionament principal es basa en portar el control de l'eina equipada, tant internament com visualment.

```
[SerializeField] InputReader inputReader;
public VoidEventSO energyConsumedEvent, endEnergyStateEvent;
[SerializeField] PlayerStats playerStats;
public GameObject sword, axe, pickaxe, energyModeSword;
```

Figura 6.5.6.1: Atributs de la classe PlayerToolsController

Com es pot veure en la Figura 6.5.6.1, en els atributs de classe tenim inputReader (per escoltar els esdeveniments de canvi d'eina), energyConsumedEvent (per escoltar l'esdeveniment de consumició d'energia) i endEnergyStateEvent (per escoltar l'esdeveniment de fi de l'estat d'energia). Després, el ScriptableObject PlayerStats que és on hi ha tota la informació de les eines del jugador. Per últim tenim un seguit de GameObjects que són les representacions visuals de cada eina, més l'estat d'energia de l'espasa.

A continuació es pot veure l'estructura dels mètodes de classe que simplement escolten als inputs de canvis d'eines i activen o desactiven els objectes que representen visualment cada eina.

```
public void EquipSword() => ChangeTool(playerStats.sword);

public void EquipAxe() => ChangeTool(playerStats.axe);

public void EquipPickaxe() => ChangeTool(playerStats.pickaxe);

public void ChangeTool(ToolSO tool){
    switch (tool.type){
        case ToolType.Axe:
            axe.SetActive(true); pickaxe.SetActive(false); sword.SetActive(false);
            break;
        case ToolType.Pickaxe:
            pickaxe.SetActive(true); axe.SetActive(false);sword.SetActive(false);
            break;
        case ToolType.Sword:
            sword.SetActive(true); axe.SetActive(false); pickaxe.SetActive(false);
            break;
            default: break;
        }
    }

public void ChangeEnergyMode() => energyModeSword.SetActive(true);

public void EndEnergyMode() => energyModeSword.SetActive(false);
```

6.5.7. Player Stats

La classe PlayerStats és un ScriptableObject de control. En aquest objecte és on emmagatzemem l'estat actual del jugador amb diferents variables per a varies situacions possibles. Aquest objecte serveix de lligam per a totes les classes que necessiten informació sobre el jugador. Al tractar-se d'un ScriptableObject, simplement necessitem una referència com a atribut en cada classe que depengui de la informació de l'estat del jugador. Aquesta seria una aproximació molt semblant al patró singleton on, normalment, aquesta classe seria una única instancia estàtica que hauria d'estar sempre present en l'escena. Gràcies al patró que fem servir, aquesta dependencia desapareix fent que qualsevol objecte des de qualsevol lloc pugui accedir a aquesta informació. A més, hi ha la possibilitat de crear varis objectes PlayerStats amb atributs amb valors diferents per si l'estat de la partida requereix un canvi dràstic en l'estat del jugador.

Model de la classe Player Stats:

Player Stats
+live: int +energy: int +attackStrength: int +currentLifes: int +currentEnergy: int +currentIsland: Island

+accesToGlobalInventary: bool +isPlacingPlaceable: bool +isCrafting: bool +isCooking: bool +isPlanting: bool +teleported: bool +hasEaten: bool +isDead: bool +sword: ToolSO +axe: ToolSO +pickaxe: ToolSO +noHitDelay: float +canRecibeDamage: bool +isAttacking: bool +isParring: bool +energyState: bool +canHit: bool +playerDie: VoidEventSO +ChangeIsland: delegate +OnChangeIsland: ChangeIsland
-OnEnable(): void -OnDisable(): void +Reset(): void +HasTool(ToolSO seed): bool +UpgradeTool(ToolSO seed): void +ReciveDamage(int damage): void -Die(): void +ChangeCurrentIsland(Island newCurrentIsland, bool globalInventoryAccess): void

Atributs de PlayerStats:

- **live**: Int que determina els punts de vida màxims del jugador
- **energy**: Int que determina els punts d'energia màxims del jugador
- **attackStrength**: Int que determina els punts de vida màxims que traurà als enemics cada cop del jugador
- **currentLifes**: Int que determina els punts de vida actuals del jugador
- **currentEnergy**: Int que determina els punts d'energia actuals del jugador
- **currentIsland**: Referència de tipus Island a l'illa actual en la que es troba el jugador
- **accesToGlobalInventary**: Bool que determina si el jugador té accés a l'inventari global
- **isPlacingPlaceable**: Bool que determina si el jugador està col·locant un col·locable
- **isCrafting**: Bool que determina si el jugador está en el panell de creació
- **isCooking**: Bool que determina si el jugador está en el panell de cuina
- **isPlanting**: Bool que determina si el jugador está en el panell de plantació
- **teleported**: Bool que determina si el jugador ha estat teleportat a la localització
- **hasEaten**: Bool que determina si el jugador ha menjat
- **isDead**: Bool que determina si el jugador ha mort

- **sword**: Referència de tipus ToolSO on s'hi troba l'inforamció actual de l'eina espasa equipada
- **axe**: Referència de tipus ToolSO on s'hi troba l'inforamció actual de l'eina destral equipada
- **pickaxe**: Referència de tipus ToolSO on s'hi troba l'inforamció actual de l'eina pic equipada
- **noHitDelay**: Float que determina el temps d'invulnerabilitat al rebre un cop del jugador
- **canRecibeDamage**:Bool que determina si el jugador pot rebre un cop
- **isAttacking**: Bool que determina si el jugador està atacant
- **isParring**: Bool que determina si el jugador està fent un contraatac
- **energyState**: Bool que determina si el jugador està en l'estat d'energia
- **canHit**: Bool que determina si el jugador pot golpejar
- **playerDie**: Objecte VoidEventSO per l'esdeveniment de mort del jugador
- **ChangeIsland**: Delegat per al canvi d'illa
- **OnChangeIsland**: Esdeveniment de tipus ChangeIsland

Mètodes de PlayerStats:

- **OnEnable, OnDisable**:Funcions cridades per el Unity quan l'objecte s'activa i es desactiva. Utilitzades per inicialitzar els atributs.

```
private void OnEnable() => Reset();

private void OnDisable(){
        Reset(); hasEaten = false; currentIsland = null;
}
```

- **Reset**: Funció per reiniciar tots els atributs de la classe al valor per defecte.

```
public void Reset(){
    currentlLifes = live; currentEnergy = energy;
    isDead = false; isPlanting = false; isPlacingPlaceable = false;
    isCrafting = false; isCooking = false; isAttacking = false;
    isParring = false; energyState = false; canRecibeDamage = true;
    teleported = false; accesToGlobalInventory = false;
}
```

- **HasTool**: Funció que determina si l'eina equipada és igual o superior a l'eina entrada per paràmetre.

```
public bool HasTool(ToolSO tool){
    switch (tool.type){
        case ToolType.Axe:
            if (axe.material >= tool.material) return true;
        case ToolType.Pickaxe:
            if (pickaxe.material >= tool.material) return true;
        case ToolType.Sword:
            if (sword.material >= tool.material) return true;
        default:
            return false;
    }
    return false;
}
```

- **UpgradeTool**: Funció que *seteja* el valor de l'eina actual amb el valor de l'eina entrada per paràmetre.

```
public void upgradeTool(ToolSO tool){
    switch (tool.type){
        case ToolType.Axe:
            axe = tool; break;
        case ToolType.Pickaxe:
            pickaxe = tool; break;
        case ToolType.Sword:
            sword = tool;break;
        default: break;
    }
}
```

- **ReciveDamage**: Funció que modifica la vida actual del jugador, restant-li el valor entrat per paràmetre. A més, determina si el jugador mor o no depenent de la vida actual restant.

```
public void ReciveDamage(int damage){
    currentlLifes -= damage; if (currentlLifes <= 0) Die();
}
```

- **Die**: Funció encarregada de controlar els estats i esdeveniments entorn a la mort del jugador.

```
private void Die(){
    canRecibeDamage = false; isDead = true; playerDie.RaiseEvent()
}
```

- **ChangeCurrentIsland**: Funció encarregada de gestionar el moviment del jugador per les diferents illes i portar constància a quina illa es troba. A més, s'encarrega de llençar l'esdeveniment de canvi d'illa.

```
public void ChangeCurrentIsland(Island newCurrentIsland, bool globalInventoryAccess){
    currentIsland = newCurrentIsland; accesToGlobalInventory = globalInventoryAccess;
    if (OnChangeIsland != null) OnChangeIsland();
}
```

6.6. Sistema d'items e inventari

Una de las parts fundamentals del joc són els recursos del món. El jugador podrà tant recolectar, emmagatzemar, utilitzar i crear recursos que li permitirán progressar en l'aventura.

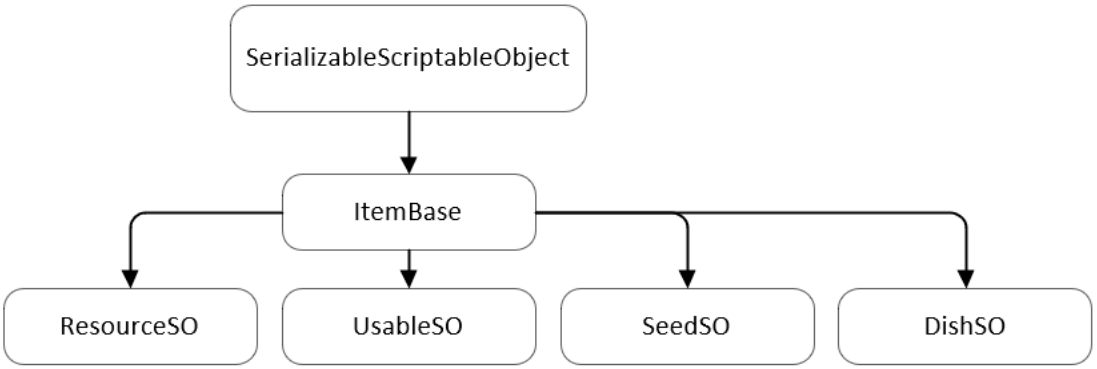


Figura 6.6.1: Jerarquia de les classes d'Ítems

Com es pot veure en la Figura 6.6.1, la jerarquia dels items està formada per una classe pare SerializableScriptableObject, que hereta de ScriptableObject. Aquesta jerarquía però, l'utilitzem per emmagatzemar la informació de cada item. La lògica forma part d'altres classes que fan servir aquests objectes per obtenir l'infromació.

```
public class SerializableScriptableObject : ScriptableObject
{
    [SerializeField] private string _guid;
    5 referencias
    public string Guid => _guid;

#if UNITY_EDITOR
    Mensaje de Unity | 0 referencias
    void OnValidate()
    {
        var path = AssetDatabase.GetAssetPath(this);
        _guid = AssetDatabase.AssetPathToGUID(path);
    }
#endif
}
```

Figura 6.6.2: Classe SerializableScriptableObject

Com es pot veure en la Figura 6.6.2, la classe SerializableScriptableObject és una classe utilitzada només per assignar a cada ScriptableObject l'identificador que Unity defineix per defecte. Aquest identificador és el mateix que fa servir el sistema d'Addressables vist en l'Apartat 6.1.3. D'aquesta manera, podem guardar l'estat de llistes d'ítems, com en el cas de l'inventari, i popularitzar el contingut carregant des de memòria.

```
public class ItemBase : SerializableScriptableObject
{
    [Space(10)]
    [Header("Item Info")]
    public string Name;
    public Color Color;
    public Sprite uiSprite;

}
```

Figura 6.6.3: Classe ItemBase

Per altra banda, com podem veure en la figura 6.6.3, la classe ItemBase està enfocada a centralitzar atributs que comparteixen tots els tipus d'ítems. A partir d'aquesta jerarquia, trobem els diferents tipus d'ítems que conformen el joc i que explicarem en els següents apartats.

6.6.1 Resources

Els ítems de tipus ResourceSO són els més bàsics i els que tenen un pes més gran en les mecàniques de joc.

```
public class ResourceSO : ItemBase
{
    [Space(10)]
    [Header("Resource Info")]
    public GameObject orbResource;
    public ParticleSystem getResource;
}
```

Figura 6.6.1.1: Classe ResourceSO

Com podem veure en la Figura 6.6.1.1, la classe ResourceSO té els atributs heretats de la classe ItemBase, afegint-hi atributs que defineixen el comportament de recol·lecció de cada recurs. OrbResource fa referència a l'objecte que apareixerà quan un recurs de món sigui

recolectat i getResource són les partícules que s'instanciaran quan l'orbe del recurs arribi al jugador. Aquest comportament es pot veure explicat amb més detall en l'Apartat 6.7.

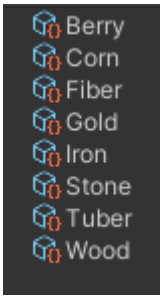


Figura 6.6.1.2: Llistat dels recursos actuals del joc

6.6.2 Usables

Els ítems de tipus UsableSO tenen un gran pes en el game flow, ja que són petits objectius que el jugador pot anar assolint a mesura que recolecta recursos. La diferencia principal entre els objectes usables i els recursos, és que els usables s'han de crear a partir de recursos i, a més cada un té un ús en específic.

```
public enum Action
{
    Activate,
    Place,
}
```

Figura 6.6.2.1: Enumerador d'accions dels objectes usables

Com podem veure en la Figura 6.6.2.1, hem definit 2 tipus d'accions per als diferents usables del joc. Els usables amb una acció de tipus Activate seran els que proporcionin un esdeveniment instantani que canviï l'estat del joc. Els usables amb una acció Place seran els objectes que el jugador pugui col·locar en el mon.

```
public class UsableSO : ItemBase
{
    [Space(10)]
    public Action action;
    [Space(10)]
    [Header("Placeable Info")]
    public AssetReference placePrefab;
    public AssetReference placeHolderPrefab;
    [Space(10)]
    [Header("Creation Info")]
    public List<ItemContainer> itemsNeeded = new List<ItemContainer>();
}
```

Figura 6.6.2.2:Classe UsableSO

Com podem veure en la Figura 6.6.2.2, la classe UsableSO té els atributs heretats de la classe ItemBase, a més de 4 atributs per definir el comportament. L'atribut action defineix el tipus d'ús que tindrà l'objecte. PlacePrefab és la referència a l'objecte que es col·locarà i, placeHolderPrefab és l'objecte de vista prèvia de l'objecte a col·locar. Per últim itemsNeeded és un llistat dels recursos amb la quantitat necessària per a crear l'usable.

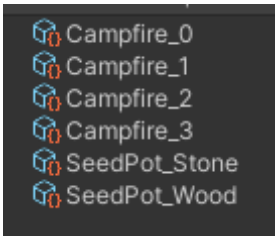


Figura 6.6.2.3: Llistat dels usables actuals del joc

6.6.3 Seeds

Els ítems de tipus SeedSo són els encarregats de guardar tota la informació sobre els tipus de llavors del joc.

```
public class SeedSO : ItemBase
{
    [Space(10)]
    [Header("Seed Info")]
    [SerializeField] public bool plantOnlyInFloor;
    [SerializeField] public bool plantOnlyInSeedPot;
    [SerializeField] public List<AssetReference> growthLevelPrefabs;
    [SerializeField] public List<AssetReference> finalLevelPrefabs;
    [SerializeField] public float minutesToGrowth = 0;
    [SerializeField] public float secondsToGrowth = 60f;
    [SerializeField] public bool randomRotation = false;
    [Space(10)]
    [Header("Harvest Info")]
    [SerializeField] public WorldResourceSO worldResource;
    [SerializeField] public UsableSO usable;
    [Space(10)]
    [Header("Multiple Recolection Seed Info")]
    [SerializeField] public bool multipleRecolection = false;
    [SerializeField] public AssetReference fruitsToRecolectPrefab;
    [SerializeField] public float minutesToNextRecolect = 0;
    [SerializeField] public float secondsToNextRecolect = 60f;
    [SerializeField] public int totalRecolections = 0;
}
```

Figura 6.6.3.1:Classe SeedSO

Com podem veure en la Figura 6.6.3.1, la classe SeedSO té els atributs heretats de la classe ItemBase més un llistat d'atributs per a definir el comportament de les llavors:

- **plantOnlyInFloor**: bool que determina si la llavor només es pot plantar a terra
- **plantOnlyInSeedPot**: bool que determina si la llavor només es pot plantar en un test
- **growthLevelPrefabs**: Llistat de referències als diferents objectes dels nivells de creixement de la llavor
- **finalLevelPrefabs**: Llistat de referències als diferents objectes de la llavor quan ha crescut completament.
- **minutesToGrowth**: Float que determina els minuts que necessita la llavor per créixer
- **secondsToGrowth**: Float que determina els segons que necessita la llavor per créixer

- **randomRotation**: Bool que determina si els diferents nivells de la llavor apareixen amb una rotació aleatoria
- **worldResource**: ScriptableObject de la classe WorldResourceSO on hi ha la informació del recurs que obtindrem en collir la llavor
- usable: ScriptableObject de la classe UsableSO on hi ha la informació del recurs que obtindrem en collir la llavor
- **multipleRecolection**: Bool que determina si la llavor es podrà collir varis cops
- **fruitsToRecolectPrefab**: Referència a l'objecte de fruits de la llavor
- **minutesToNextRecolect**: Float que determina els minuts que necessita la llavor per créixer la següent collita
- **secondsToNextRecolect**: Float que determina els segons que necessita la llavor per créixer la següent collita
- **totalRecolections**: Int que determina el nombre total de collites de la llavor

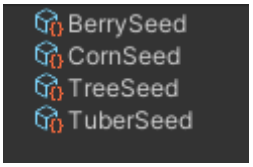


Figura 6.6.3.2: Llistat de les llavors actuals del joc

6.6.4 Dishes

Els ítems de tipus DishSO són els encarregats de guardar tota la informació sobre els tipus de plats del joc.

```
public class DishSO : ItemBase
{
    [Header("Needs")]
    public List<ItemContainer> itemsNeeded = new List<ItemContainer>();
    public List<ItemContainer> fuel = new List<ItemContainer>();
    [Header("Buffs")]
    public int extraLives = 0;
    public int extraStamina = 0;
    public string description;
}
```

Figura 6.6.4.1: Classe DishSO

Com podem veure en la Figura 6.6.4.1, la classe DishSO té els atributs heretats de la classe ItemBase més un llistat d'atributs per a definir el comportament dels plats:

- **itemsNeeded**: Llistat d'objectes ItemContainer on hi ha la informació i les quantitats dels recursos requerits per a crear el plat
- **fuel**: Llistat d'objectes ItemContainer on hi ha la informació i les quantitats dels recursos requerits per a cuinar el plat
- **extraLives**: Int que determina la vida que obtindra el jugador al menjar el plat
- **extraStamina**: Int que determina l'energia que obtindra el jugador al menjar el plat
- description: String que determina la descripció de l'objecte

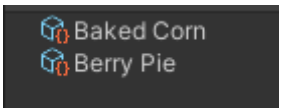


Figura 6.6.4.2: Llistat de plats actuas del joc

6.7. Inventari del jugador

6.7.1 Estructura bàsica

L'inventari del jugador és una part molt important en el joc, ja que és l'encarregada de gestionar els recursos que el jugador anirà recolectant, fent que aparegui una progressió i que el jugador senti que està avançant.

L'inventari i recol·lecció de recursos és una de les parts del projecte que ha patit més canvis e iteracions. Després de totes les proves i canvis, es va decidir deixar l'estructura d'inventaris que comentarem a continuació:

- Inventari del jugador de recursos: Inventari que el jugador portarà a sobre en tot moment, té límit de quantitat i el podem definir com a motxilla. Encarregat d'emmagatzemar recursos
- Inventari global de recursos: Inventari que forma part de la zona de la casa del jugador, no té límit de quantitat i és on el jugador dipositarà els seus recursos quan la seva motxilla estigui plena. El podem definir com a un cofre. Encarregat d'emmagatzemar recursos.
- Inventari global d'usables: Inventari que també forma part de la zona de la casa del jugador, no té límit de quantitat i s'encarrega d'emmagatzemar els objectes usables del jugador.

Per tant, el jugador tindrà una motxilla per guardar recursos i un cofre per guardar tan recursos com usables. Els usables, un cop creats, s'emmagatzemarán directament al cofre i, quan el jugador entri en l'estat de col·locació d'objectes, podrà veure el llistat de tots els usables de col·locació que tindrà disponibles.

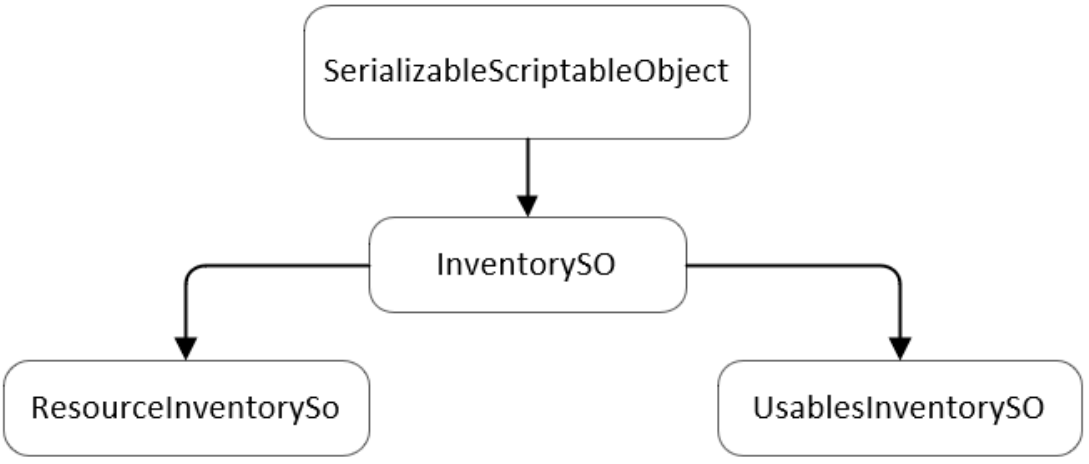


Figura 6.7.1.1: Jerarquía de les classes d'inventari

Cada un dels inventaris mencionats anteriorment són ScriptableObjects com podem veure en la Figura 6.7.1.1, i els dividim en dos depenent si guarden recursos o usables, ja que el funcionament varia.
Per últim tenim la classe InventoryManager que és l'encarregada de, amb la informació de cada inventari, gestionar-ho tot.

6.7.2 InventorySO

La classe InventorySO és la classe base dels inventaris. Hereta de la classe ScriptableObject i té tots els mètodes bàsics per a tots els inventaris.
Com a atribut aquesta classe base només té l'atribut inventory que és un llistat d'objectes de la classe InventorySlot.

```
public class InventorySlot
{
    public ItemBase item;
    public int amount;
    5 referencias
    public InventorySlot(ItemBase item, int amount)
    {
        this.item = item;
        this.amount = amount;
    }
}
```

Figura 6.7.2.1: Classe InventorySlot

Com podem veure en la Figura 6.7.2.1, la classe InventorySlot simplement és una classe per emmagatzemar un item i la quantitat. Aquesta classe es farà servir per portar el control dels objectes que es guardin en l'inventari, ja que cadascun dels objectes de l'inventari tindrà la quantitat que el jugador té actualment.

Model de la classe InventorySO:

InventorySO
+inventory: List<InventorySlot>
+SlotAmount(ItemBase item): int +SlotNumber(ItemBase item): int +ContainsAmount(ItemBase item, int amount = 1): bool +GetSlot(int i): InventorySlot

```
+FirstEmptySlot(): int
+IsEmpty(): bool
+Remove(int actualSlot, int amount): void
+ClearInventory(): void
```

Atributs de InventorySO:

- **inventory**: Llista d'objectes de tipus InventorySlot encarregada d'emmagatzemar l'informació de tots els ítems amb les seves quantitats que el jugador ha obtingut o creat

Mètodes de InventorySO:

+ **SlotAmount:** Funció que retorna la quantitat actual de l'item entrat per paràmetre

```
public virtual int SlotAmount(ItemBase item){
    int actualSlot = SlotNumber(item);
    if (actualSlot != -1) return inventory[actualSlot].amount;
    return 0;
}
```

+ **SlotNumber**: Funció que retorna la posició actual a l'inventari de l'ítem entrat per paràmetre

```
public virtual int SlotNumber(ItemBase item){
    for (int i = 0; i < inventory.Count; i++)
        if (item == inventory[i].item) return i;
    return -1;
}
```

+ **ContainsAmount**: Funció que determina si l'inventari té igual o més de la quantitat de l'ítems entrats per paràmetre.

```
public virtual bool ContainsAmount(ItemBase item, int amount = 1){
    return SlotAmount(item) >= amount;
}
```

+ **GetSlot**: Funció que retorna l'objecte InventarySlot de la posició de l'inventari entrada per paràmetre

```
public virtual InventorySlot GetSlot(int i){
    if (i >= inventory.Count) return null;
    return inventory[i];
}
```

+ **FirstEmptySlot**: Funció que retorna la primera posició buida de l'inventari

```
public virtual int FirstEmptySlot(){
    int i = 0;
    while (i < inventory.Count){
        if (inventory[i].item == null) return i;
        i++;
    }
    return -1;
}
```

+ **IsEmpty**: Funció que determina si l'inventari està buit

```
public virtual bool IsEmpty(){
    if (inventory.Count == 0) return true;
    int i = 0;
    while (i < inventory.Count){
        if (inventory[i].item != null) return false;
        i++;
    }
    return true;
}
```

+ **Remove**: Funció que retira de la posició entrada per paràmetre, la quantitat entrada per paràmetre

```
public virtual void Remove(int actualSlot, int amount = 1){
    inventory[actualSlot].amount -= amount;
    if (inventory[actualSlot].amount <= 0) inventory[actualSlot].amount = 0;
}
```

+ **ClearInventory**: Funció que neteja el contingut del inventari

```
public virtual void ClearInventory() => inventory.Clear();
```

6.7.3 ResourceInventorySO

La classe ResourceInventorySO és l'inventari encarregat de gestionar els ítems de tipus recurs. Hereta els atributs base de la classe InventorySlot.

Model de la classe ResourceInventorySO:

ResourceInventorySO
+totalResourceAmount: int +inventoryCapacity: int
+Add(ItemBase item, int amount = 1): void +Add(ResourceInventorySo inventory): void +Remove(ItemContainer resourceContainer): int +HasResources(List<ItemContainer> resources): bool +RemoveResources(List<ItemContainer> resources): bool +ClearInventory(): void

Atributs de ResourceInventorySO:

- **totalResourceAmount**: Int per determinar la quantitat total d'ítems de l'inventari
- **inventoryCapacity**: Int per determinar la quantitat màxima de l'inventari

Mètodes de ResourceInventorySO

+ **Add:** Funció encarregada d'afegir un recurs a l'inventari, comprovant si l'inventari no està ple. També comprova si el recurs existeix per augmentar la seva quantitat. A més, tenim també la funció Add, però entrant per paràmetre un inventari de tipus recurs i tots els recursos s'emmagatzemen a l'inventari.

```
public void Add(ItemBase item, int amount = 1){
    if (amount <= 0) return;
    if (inventoryCapacity != -1) if (totalRescourceAmount >= inventoryCapacity) return;
    int actualSlot = SlotNumber(item);
    if (actualSlot != -1){
        inventory[actualSlot].amount += amount; totalRescourceAmount += amount; return;
    }
    inventory.Add(new InventorySlot(item, amount)); totalRescourceAmount += amount;
}

public void Add(ResourceInventorySO inventory){
    if (inventory.IsEmpty()) return;
    for (int i = 0; i < inventory.inventory.Count; i++)
        Add(inventory.inventory[i].item, inventory.inventory[i].amount);
}
```

+ **Remove**: Funció que rep un objecte ItemContainer on hi ha la informació del recurs i la quantitat a eliminar. Elimina la quantitat de recurs d'entrada i, si el jugador no té la suficient quantitat, retorna la quantitat que queda per eliminar

```
public int Remove(ItemContainer resourceContainer){
    if (resourceContainer.amount <= 0) return -1;
    int actualSlot = SlotNumber(resourceContainer.item);
    int amountPendentingToRemove = resourceContainer.amount;
    if (actualSlot != -1){
        if (inventory[actualSlot].amount < resourceContainer.amount){
            // Si el jugador té menys quantitat de recurs del que s'ha de treure
            totalRescourceAmount -= inventory[actualSlot].amount;
            amountPendentingToRemove = resourceContainer.amount-inventory[actualSlot].amount;
            inventory[actualSlot].amount = 0;
        }
        else{ // Si el jugador té més quantitat de recurs del que s'ha de treure
            inventory[actualSlot].amount -= resourceContainer.amount;
            totalRescourceAmount -= resourceContainer.amount;
        }
        if (inventory[actualSlot].amount <= 0)inventory.Remove(inventory[actualSlot]);
    }
    return amountPendentingToRemove;
}
```

+ **HasResources**: Funció que determina si l'inventari té els recursos i quantitats de la llista d'objectes ItemContainer entrada per paràmetre

```
public bool HasResources(List<ItemContainer> resources){
    foreach (var resource in resources)
        if (!ContainsAmount(resource.item, resource.amount)) return false;
     return true;
}
```

+ **RemoveResources**: Funció que elimina els recursos de la llista d'objectes de tipus ItemContainer

```
public bool RemoveResources(List<ItemContainer> resources){
    if (HasResources(resources)){
        foreach (var resource in resources)Remove(resource);
        return true;
    }
    return false;
}
```

+ **ClearInventory**: Funció que crida a la funció de la classe pare InventorySO per a netejar l'inventari, a més reinicia el total de recursos a 0

```
public override void ClearInventory(){base.ClearInventory(); totalRescourceAmount = 0;}
```

6.7.4 UsablesInventorySO

La classe UsablesInventorySO és l'inventari encarregat de gestionar els ítems de tipus usable i hereta els atributs base de la classe InventorySlot. A diferència de l'inventari de recursos vist en l'Apartat 6.7.3, aquesta classe és molt més simple, ja que l'inventari és global i no hi ha tanta lògica de comprovacions, simplement afegirem un objecte o l'utilitzarem.

```
public class UsablesInventorySO : InventorySO
{
    1 referencia
    public int Add(ItemBase item)
    {
        InventorySlot slot = new InventorySlot(item, 1);
        int actualSlot = SlotNumber(item);
        if (actualSlot > -1) inventory[actualSlot].amount++;
        else
        {
            inventory.Add(slot); actualSlot = SlotNumber(item);
        }
        return actualSlot;
    }

    1 referencia
    public InventorySlot Use(UsableSO usable)
    {
        int actualSlot = SlotNumber(usable);
        if (actualSlot > -1)
            Remove(actualSlot);
        return inventory[actualSlot];
    }

    [ContextMenu("ClearInventory")]
    8 referencias
    public override void ClearInventory() => base.ClearInventory();
}
```

Figura 6.7.4.1: Classe UsablesInventorySO

Com podem veure en la Figura 6.7.4.1, el mètode Add afegeix un ítem al inventari i retorna la posició de l'objecte dins l'inventari. Per altra banda, tenim el mètode Use que simplement elimina 1 quantitat de l'ítem entrat per paràmetre.

6.7.5 InventoryManager

InventoryManager és una classe de tipus ScriptableObject. És l'encarregada de gestionar tots els inventaris del jugador mencionats en els Apartats 6.7.2, 6.7.3 i 6.7.4. La funció principal és escoltar tots els diferents esdeveniments que modifiquin l'estat dels diferents inventaris.

Model de la classe InventoryManager:

Inventory Manager
-playerStats: PlayerStats -playerInventoryResources: ResourceInventorySO -globalInventoryResources: ResourceInventorySO -globalInventoryUsables: UsablesInventorySO -addResourceEventChannel: ItemContainerEventSO -addUsableEventChannel: UsableEventSO -useUsableEventChannel: UsableEventSO -depositePlayerInventoryChannel: VoidEventSO -updateUIUsablesInventorySlotEvent: InventorySlotEventSO
-OnEnable(): void -OnDisable(): void +AddResource(ItemContainer resourceContainer): void +AddUsable(UsableSO usable): void +UseUsable(UsableSO usable): void +DepositePlayerInventoty(): void +ItemAmount(ItemBase item): int +HasItems(List<ItemContainer> items) bool +RemoveItems(List<ItemContainer> items) bool +CraftItem(ItemBase item): bool +PlayerResourcesInventory(): List<InventorySlot> +GlobalrResourcesInventory(): List<InventorySlot> +GlobalUsablesInventory(): List<InventorySlot> +ClearAll(): void

Atributs de InventoryManager:

- **playerStats**: ScriptableObject on hi ha la informació actual del jugador
- **playerInventoryResources**: Objecte de la classe ResourcesInventorySO que defineix l'inventari de recursos del jugador
- **globalInventoryResources**: Objecte de la classe ResourcesInventorySO que defineix l'inventari de recursos global de la partida
- **globalInventoryUsables**: Objecte de la classe UsablesInventorySO que defineix l'inventari global d'usables de la partida.
- **addResourceEventChannel**: Objecte ItemContainerEventSO per a l'esdeveniment d'obtenció de recurs
- **addUsableEventChannel**: Objecte UsableEventSO per a l'esdeveniment d'obtenció

d'usable

- **useUsableEventChannel**: Objecte UsableEventSO per a l'esdeveniment d'utilització d'usable
- **depositePlayerInventoryChannel**: Objecte VoidEventSO per a l'esdeveniment de dipositar l'inventari de recursos del jugador al inventari de recursos global
- **updateUIUsablesInventorySlotEvent**: Objecte InventorySlotEventSO per a l'esdeveniment d'actualitzar la interfície encarregada de mostrar l'inventari d'usables.

Mètodes de InventoryManager:

- **OnEnable, OnDisable:** Funcions cridades per el Unity quan l'objecte s'activa i es desactiva. Utilitzades per subscriure's i desubscriure's als esdeveniments.

```
private void OnEnable(){
    _addResourceEventChannel.OnEventRaised += AddResource;
    _addUsableEventChannel.OnEventRaised += AddUsable;
    _useUsableEventChannel.OnEventRaised += UseUsable;
    _depositePlayerInventoryChannel.OnEventRaised += DepositePlayerInventory;
}

private void OnDisable(){
    _addResourceEventChannel.OnEventRaised -= AddResource;
    _addUsableEventChannel.OnEventRaised -= AddUsable;
    _useUsableEventChannel.OnEventRaised -= UseUsable;
    _depositePlayerInventoryChannel.OnEventRaised -= DepositePlayerInventory;
}
```

+ **AddResource**: Funció que afegeix un recurs a l'inventari del jugador

```
public void AddResource(ItemContainer resourceContainer){
    playerInventoryResources.Add(resourceContainer.item,resourceContainer.amount);
}
```

+ **AddUsable**: Funció que afegeix un usable a l'inventari d'usables i ho comonica a la interfície

```
public void AddUsable(UsableSO usable){
    int slot = globalInventoryUsables.Add(usable);
    _updateUIUsablesInventorySlotEvent.
        RaiseEvent(globalInventoryUsables.Inventory[slot]);
}
```

+ **UseUsable**: Funció que utilitza l'usable i ho notifica a la interfície

```
public void UseUsable(UsableSO usable){
    _updateUIUsablesInventorySlotEvent.RaiseEvent(globalInventoryUsables.Use(usable));
}
```

+ **DepositePlayerInventory**: Funció que diposita l'inventari de recursos del jugador a l'inventari global de recursos

```
public void DepositePlayerInventory(){
    globalInventoryResources.Add(playerInventoryResources);
    playerInventoryResources.ClearInventory();
}
```

+ **ItemAmount**: Funció que retorna la quantitat actual de l'ítem entrat per paràmetre depenent del tipus d'ítem i de si el jugador pot accedir a l'inventari global de recursos i usables.

```
public int ItemAmount(ItemBase item){
    if (item is UsableSO usable) return globalInventoryUsables.SlotAmount(usable);
    else if (item is ResourceSO resource){
       if (playerStats.accesToGlobalInventory)
            return playerInventoryResources.SlotAmount(resource)
                   + globalInventoryResources.SlotAmount(resource);
            return playerInventoryResources.SlotAmount(resource);
        }
    else if (item is SeedSO seed){
        if (playerStats.accesToGlobalInventory)
            return playerInventoryResources.SlotAmount(seed)
                + globalInventoryResources.SlotAmount(seed);
        return playerInventoryResources.SlotAmount(seed);
    }
    return 0;
}
```

+ **HasItems**: Funció que determina si el jugador té igual o més de la quantitat de la llista d'ítems entrada per paràmetre

```
public bool HasItems(List<ItemContainer> items){
    for (int i = 0; i < items.Count; i++){
        if (playerStats.accesToGlobalInventory) {
            if (!((playerInventoryResources.SlotAmount(items[i].item)
             + globalInventoryResources.SlotAmount(items[i].item)) >= items[i].amount))
                return false;
            }
            else if (!(playerInventoryResources.SlotAmount(items[i].item)
                     >= items[i].amount))
                return false;}
    return true;}
```

+ **RemoveItems**: Funció que elimina la quantitat del llistat d'ítems entrats per paràmetre, si el jugador té els ítems.

```
public bool RemoveItems(List<ItemContainer> items){
    if (HasItems(items)){
            if (!playerStats.accesToGlobalInventory)
              playerInventoryResources.RemoveResources(items);
        else{
            for (int i = 0; i < items.Count; i++){
                int amountPendingToRemove = playerInventoryResources.Remove(items[i]);
                if ( amountPendingToRemove  > 0)
                    globalInventoryResources.Remove(new ItemContainer(items[i].item,
                                                    amountPendingToRemove));
            }
        }
        return true;
    }
    return false;
}
```

+ **CraftItem**: Funció per crear l'ítem entrat per paràmetre depenent del tipus, retornant cert o fals depenent de si l'ha pogut crear

```
public bool CraftItem(ItemBase item){
    if (item is UsableSO usable)
        if (RemoveItems(usable.itemsNeeded)){ AddUsable(usable); return true; }
    if (item is DishSO dish)
        if (RemoveItems(dish.itemsNeeded)) return true;
    return false;
}
```

+ **PlayerResourcesInventory**,**GlobalResourcesInventory**, **GlobalUsablesInventory**:

  Funcions que retornen el llistat d'objectes de tipus InventorySlot de cada inventari

```
public List<InventorySlot> PlayerResourcesInventory() =>
                          playerInventoryResources.Inventory;
public List<InventorySlot> GlobalResourcesInventory() =>
                          globalInventoryResources.Inventory;
public List<InventorySlot> GlobalUsablesInventory() =>
                           globalInventoryUsables.Inventory;
```

+ **ClearAll**: Funció per a netejar tots els inventaris del jugador

```
public void ClearAll(){
    playerInventoryResources.ClearInventory();
    globalInventoryResources.ClearInventory();
    globalInventoryUsables.ClearInventory();
}
```

6.8. Estructura i manteniment dels recursos del món

Durant el joc, el jugador podrà recolectar, plantar, millorar, reparar, eliminar, i crear diferentes recursos i ítems, els quals denominem recursos de món, ja que tenen una representació física en l'entorn. Aquests recursos de món tenen la seva lógica i interactúen amb el jugador de diverses maneres, però la informació interna del ítem al que corresponen, l'obtenen dels ítems explicats en l'Apartat 6.6.
A més, tots els recursos del món que puguin ser modificats o recolectats tindran un script Addressable.

```
public class Addressable : MonoBehaviour
{
    public AssetReference assetRefenernce;
    4 referencias
    public string GetGUID()
    {
        return assetRefenernce.AssetGUID;
    }
}
```

Figura 6.8.1: Classe Addressable

Com podem veure en la Figura 6.8.1, aquesta classe s'utilitza en tots els recursos de món i simplement té una referència al propi objecte. D'aquesta manera, al guardar o carregar l'estat del món, podem desar aquesta referència per a poder tornar a l'estat actual.

6.8.2 WorldResourceManager

Cada cop que un recurs de món es recolecta, deixa anar un orbe per cada recurs que s'obté al recolectar-ho. Per exemple, un arbre podria deixar, al recolectar-se, un orbe de recurs fusta i un orbe de recurs fibra. Aquests orbes s'instancien en la posició del recurs de món i viatgen cap al jugador fins desaparèixer en quan entren en contacte amb ell. Per portar una gestió de tots els orbes que s'instancien i tenir-ho separat del propi recurs, vam crear el gestor WorldResourceManager que viu dins l'escena gameplay i està present en totes les escenes de món. D'aquesta manera, qualsevol recurs que es recolecti, independentment d'on es trobi, farà aparèixer els orbes de recurs que el jugador podrà obtenir.

```
[SerializeField] TransformEventSO _resourceDestroyedChannel = default;
[SerializeField] ItemContainerEventSO _resourceToSpawn = default;
[SerializeField] private TransformAnchor playerTransform;
[SerializeField] private Vector3 lastWorldResourceDestroyed;
```

Figura 6.8.2.1: Atributs de la classe WorldResourcesManager

Com es pot veure en la Figura 6.8.2.1, la classe WorldResourceManager simplement té 4 atributs:

- **resorceDestroyedChannel**: Esdeveniment que passa per paràmetre una transformació. En aquest cas és la transformació en el món del recurs recolectat
- **resorceToSpawn**: Esdeveniment que passa per paràmetre un objecte del tipus ItemContainer. En aquest cas és el recurs que s'obté al recolectar el recurs de món
- **playerTransform**: Objecte de tipus TransformAnchor on hi ha la transformació de món actual del juagador
- **lastWorldResourceDestoyed**: Vector3 per emmagatzemar la posició de l'últim recurs de món recolectat

Per tant, la classe escolta els esdeveniments i instancia un orbe de recurs, com es pot veure en la Figura 6.8.2.1.2, que viatjarà cap al jugador i, al col·lisionar, instanciarà les partícules determinades pel recurs.

```
public void CreateOrb(ItemContainer item)
{
    ResourceSO resource = item.item as ResourceSO;
    // Mirem si l'item que ens entar es de tipus Resource al fer el cast
    if (resource != null)
    {
        GameObject prefab = resource.orbResource;
        ParticleSystem particleSystem = resource.getResource;
        Vector3 spawnPosition = new Vector3(lastWorldResourceDestroyed.x,
            lastWorldResourceDestroyed.y + 3, lastWorldResourceDestroyed.z);
        ResourceOrb newOrb = Instantiate(prefab, spawnPosition,
            Quaternion.identity).GetComponent<ResourceOrb>();
        newOrb.target = playerTransform.Transform;
        newOrb.particles = particleSystem; newOrb.resource = item;
    }
}
```

Figura 6.8.2.1.2: Mètode CreateOrb de la clase WorldResourceManager

6.8.3 Recol·lecció

Tots els recursos del món que el jugador podrà recol·lectar tenen la infromació bàsica emmagatzemada en la classe WorldResourceSO, que és un ScriptableObject.

```
public class WorldResourceSO : ScriptableObject
{
    public string worldResourceName;
    [SerializeField]
    public ItemContainer[] resources;
    [Header("Gathering Needs")]
    public ToolSO toolNeeded;
    public int live = 3;
    public bool canBeHittedToGather = true;
}
```

Figura 6.8.3.1: Classe WorldResourceSO

Com es pot veure en la Figura 6.8.3.1, la classe WorlResourceSO té un seguit d'atributs per definir el recurs de món i com es podrà recolectar:

- **worldResourceName**: String per definir el nom del recurs
- **resources**: Llistat d`objectes container on trobem la informació de l'ítem i la quantitat que el recurs de món generarà al ser recolectat
- **toolNeeded**: Objecte de tipus ToolSO per determinar quina és l'eina requerida per a poder recolectar el recurs.
- **live**: Int per emmagatzemar la vida màxima del recurs per determinar els cops que podrà rebre abans de ser recolectat
- **canBeHittedToGather**: Bool per determinar si el recurs es podrà golpejar amb l'eina per a ser recolectat o no (els casos negatius normalment són característics dels fruits de les plantes)

A més, tenim la classe WorldResource encarregada de gestionar tota la lògica depenent de la informació de la classe WorldResourceSO.

Model de la classe WorldResource:

World Resource
+data: WorldResourceSO -resourceDestroyedTransform: TransformEventSO -itemsToSpawnEvent: ItemContainerEvenetSO -newObjectOnDestroy: GameObject

-actualLive: int
-Start(): void +Gather(bool destroyObject): void +Hit(int damage): void

Atributs de WorldResource:

- **data**: Objecte de tipus WorldResourceSO on hi ha la informació del recurs
- **resourceDestroyedTransform**: Esdeveniment de tipus TransformEventSO per a poder informar de la transformació de l'objecte per instanciar els orbes de recurs
- **itemsToSpawnEvent**: Esdeveniment de tipus ItemContainerEventSO per determinar tots els recursos que s'han d'instanciar al recolectar el recurs de món
- **newObjectOnDestroy**: Objecte que s'instanciarà un cop s'hagi recolectat el recurs de món
- **actualLive**: Int que determina la quantitat de vida actual del recurs de món

Mètodes de WorldResource:

- **Start**: Funció cridada per el Unity abans del primer frame. Utilitzada per a reiniciar la vida actual de l'objecte

```
private void Start() => actualLive = data.live;
```

- **Gather**: Funció encarregada de llençar tots els esdeveniments de recol·lecció i si el valor entrat per paràmetre és cert, destruieix l'objecte i instancia el nou objecte si newObjectOnDestroy es diferent de nul

```
public void Gather(bool destroyObject){
    resourceDestroyedTransform.RaiseEvent(gameObject.transform);
    foreach (var item in data.resources) itemsToSpawnEvent.RaiseEvent(item);
    if (destroyObject){
        if (OnDestroyEvnt != null) OnDestroyEvnt(gameObject);
         if (newObjectOnDestroy != null)
             Instantiate(newObjectOnDestroy, transform.position, transform.rotation);
             if (!Addressables.ReleaseInstance(gameObject)) Destroy(this.gameObject);
    }

}
```

- **Hit**: Funció encarregada de gestionar els cops del jugador restant la quantitat entrada per paràmetre a la vida actual i cridant a la funció de recol·lecció si el recurs de món s'ha quedat sense vida

```
public void Hit(int damage){
    actualLive -= damage; if (actualLive <= 0) Gather(true);
    gameObject.transform.DOShakeScale(0.2f, 0.2f, 2, 90f, true);
}
```

6.8.4 Plantació

La plantació de recursos es pot dur a terme tant en el terra com en tests, depenent de la llavor que es vulgui plantar. Sigui quin sigui el cas, el procediment és el mateix: instanciar un objecte amb la classe Seed que serà l'encarregat de gestionar la llavor (SeedPlaceHolder). Aquest objecte és un objecte del món buit en el que s'aniran instanciant les diferents fases de la llavor i que serà destruït un cop la llavor sigui recolectada. A més, en el cas dels testos, es modificarà el comportament de l'objecte seedPlaceHolder que tingui en aquell moment.

6.8.4.1 Seed

Model de la classe Seed;

Seed
-playerStats: PlayerStats +seed: SeedSO +actualGrowthLevel: int +buffGrowthSpeed: int -seedPrefab: GameObject +timeGrowingActualLevel: float +secondsToGrwth: float -growthSpeed: float -maxLevel: int -spawnLastPrefab: bool +timesRecolected: int +secondsToNextRecolect: float +fruitsToRecolectPrefab: GameObject +timeGrowingNextRecolect: float +canRecolect: bool -worldResource: WorldResource
-OnEnable(): void -IsMaxLevel(): bool -CanGrowt(): bool -CanGrowFruits(): bool -GrowingPlant(): IEnumerator -GrowingFruits(): IEnumerator +Recolect(): void

+Plant(): void
+Load(string seed, int actualGrowthLevel, float timeGrowing, int timesReclected, float timeGrowingNextRecolect, bool canRecolect, float buff = 0): void
+Grow(): void
-NewRecolection(): void
-InstantiateNextLevel(): void
+Gather(): void
-StartGrowingCorrutines(): void

Atributs de Seed:

- **playerStats**: ScriptableObject on hi ha la informació actual del jugador
- **seed**: Objecte del tipus SeedSO on hi ha la informació de la llavor plantada
- **actualGrowthLevel**: Int per determinar el nivell actual de creixement de la llavor
- **buffGrowthSpeed**: Int per determinar l'augment de la velocitat de creixement de la llavor
- **seedPrefab**: GameObject de l'actual representació visual del nivell de la llavor
- **timeGrowingActualLevel**: Float per determinar el temps que porta creixent la llavor en el nivell actual
- **secondsToGrowth**: Float per determinar el temps que necessita la llavor per créixer un nivell
- **growthSpeed**: Float per determinar la velocitat de la llavor en créixer
- **maxLevel**: Int per determinar el nivell màxim de creixement de la llavor
- **spawnLastPrefab**: Bool per determinar si el següent objecte a instanciar és l'últim de la llavor
- **timesRecolected**: Int per determinar els cops que la llavor ha estat recolectada
- **secondsToNextRecolect**: Float per determinar el temps que necessita la llavor per créixer la següent collita
- **fruitsToRecolectPrefab**: GameObject de la representació visual dels fruits de la collita
- **timeGrowingNextRecolect**: Flat per determinar el temps que porta creixent la següent collita
- **canRecolect**: Bool per determinar si la collita es pot recol·lectar
- **worldResource**: Referència al component WorldResource de l'objecte

Mètodes de Seed:

- **OnEnable:** Funció cridada per el Unity quan l'objecte s'activa. Utilitzada per obtenir la referència del component WorldResource de l'objecte.

```
private void OnEnable(){
    worldResource = this.gameObject.GetComponent<World_Resource>();
}
```

- **IsMaxLevel**, **CanGrow**, **CanGrowFruits**: Funcions d'ajuda. IsMaxLevel determina si el nivell actual és l'últim. CanGrow determina si la llavor pot créixer més. CanGrowFruits determina si la llavor pot créixer més fruits.

```
private bool IsMaxLevel() => actualGrowthLevel >= ( maxLevel -1);
private bool CanGrow() => seedPrefab != null && !IsMaxLevel();
private bool CanGrowFruits() => seed.MultipleRecolection()
                              && IsMaxLevel()
                              && timesRecolected < seed.totalRecolections
                              && !canRecolect;
```

- **GrowingPlant**, **GrowingFruits**: Corrutines per anar fent créixer la llavor tant de nivell com fer créixer fruits depenent de la velocitat de creixement i del temps de creixement.

```
private IEnumerator GrowingPlant(){
    while (timeGrowingActualLevel < secondsToGrowth){
        timeGrowingActualLevel++;
        yield return new WaitForSeconds(growthSpeed - buffGrowthSpeed);
    }
    Grow();
}

private IEnumerator GrowingFruits(){
    while (timeGrowingNextRecolect < secondsToNextRecolect){
        timeGrowingNextRecolect++;
        yield return new WaitForSeconds(growthSpeed - buffGrowthSpeed);
    }
    NewRecolection();
}
```

- **Recolect**: Funció de recol·lecció de la llavor. Si no pot créixer fruits, la llavor es recolecta. SI pot créixer fruits, es comença la corrutina de creixement de fruits.

```
public void Recolect(){
    if (canRecolect){
        SeedPot seedPot = GetComponentInParent<SeedPot>();
        if (seedPot) seedPot.CanRecolect(false);
        canRecolect = false; timesRecolected++;
```

```
        if (CanGrowFruits()){
            if (worldResource != null) worldResource.Gather(false);
            fruitsToRecolectPrefab.gameObject.SetActive(canRecolect);
            StartGrowingCorrutines(); return;
        }
    }
    Gather();
}
```

- **Plant**: Funció per plantar una llavor. S'encarrega de llegir la informació de la llavor a plantar i establir els valors dels atributs.

```
public void Plant(SeedSO seed, float buff = 0){
    if (seed.worldResource!= null && worldResource!= null)
        worldResource.data = seed.worldResource;
    maxLevel = seed.growthLevelPrefabs.Count; this.seed = seed;
    secondsToGrowth = (int)(seed.minutesToGrowth * 60 + seed.secondsToGrowth);
    buffGrowthSpeed = buff / 100;
    if (seed.MultipleRecolection()) secondsToNextRecolect = seed.secondsToNextRecolect;
    InstantiateNextLevel();
}
```

- **Load**: Funció que gestiona la càrrega de les llavors. Estableix els valors entrats per paràmetres i instancia el nivell actual de la llavor.

```
public void Load(string seed, int actualGrowthLevel, float timeGrowing,
    int timesRecolected, float timeGrowingNextRecolect, bool canRecolect, float buff = 0){
    Addressables.LoadAssetAsync<SeedSO>(seed).Completed += (seedSO) => {
        if (seedSO.Result.worldResource != null && worldResource != null)
            worldResource.data = seedSO.Result.worldResource;
            this.seed = seedSO.Result;
            this.maxLevel = this.seed.growthLevelPrefabs.Count;
            if (actualGrowthLevel >= this.maxLevel) this.actualGrowthLevel = this.maxLevel-1;
            else this.actualGrowthLevel = actualGrowthLevel;
            this.secondsToGrowth = (int)(seedSO.Result.minutesToGrowth * 60
                                        + seedSO.Result.secondsToGrowth);
    Addressables.InstantiateAsync(seedSO.Result.growthLevelPrefabs[this.actualGrowthLevel],
        this.transform.position, Quaternion.identity, this.transform)
        .Completed += (prefab) => {
            seedPrefab = prefab.Result;
            this.timeGrowingActualLevel = timeGrowing;
            this.timesRecolected = timesRecolected;
            this.timeGrowingNextRecolect = timeGrowingNextRecolect;
            this.canRecolect = canRecolect;
            this.actualGrowthLevel = actualGrowthLevel;
            if (seedSO.Result.MultipleRecolection()) secondsToNextRecolect
                                                    = seedSO.Result.secondsToNextRecolect;
            if (this.canRecolect) NewRecolection();
            buffGrowthSpeed = buff / 100;
            StartGrowingCorrutines();
        };
    };
}
```

- **Grow**: Funció de creixement de la planta. Encarregada d'augmentar el nivell actual de la llavor, reiniciar el contador del temps de creixement e instanciar el nou nivell.

```
public void Grow(){
    timeGrowingActualLevel = 0f;
    actualGrowthLevel++;
    InstantiateNextLevel();
}
```

- **NewRecolection**: Funció encarregada de gestionar una nova collita de fruits. Instancia l'objecte fruits de la llavor si no existeix i estableix els diferents valors de recollida. Si l'objecte llavor existeix, el deixa visible.

```
private void NewRecolection(){
    timeGrowingNextRecolect = 0;
    SeedPot seedPot = GetComponentInParent<SeedPot>();
    if (fruitsToRecolectPrefab == null){
        Addressables.InstantiateAsync(seed.fruitsToRecolectPrefab,
            seedPrefab.transform.position, Quaternion.identity, this.transform)
            .Completed += (prefab) => {
                fruitsToRecolectPrefab = prefab.Result;
                if (seedPot) seedPot.CanRecolect(true);
                canRecolect = true;
            };
    }
    else{
        fruitsToRecolectPrefab.gameObject.SetActive(true);
        canRecolect = true;
        if (seedPot) seedPot.CanRecolect(true);
    }
}
```

- **Gather**: Funció de recollida de la llavor. És la funció encarregada de destruir tant l'objecte com l'objecte fruits i eliminar la relació amb el test, si existeix.

```
public void Gather(){
    SeedPot seedPot = GetComponentInParent<SeedPot>();
    if (seedPot) seedPot.RemoveSeed();
    if (fruitsToRecolectPrefab != null)
    if (!Addressables.ReleaseInstance(fruitsToRecolectPrefab))
        Destroy(fruitsToRecolectPrefab);
    if (!Addressables.ReleaseInstance(seedPrefab)) Destroy(seedPrefab);
    if (worldResource!= null) worldResource.Gather(true);
    if (!Addressables.ReleaseInstance(this.gameObject)) Destroy(this.gameObject);
}
```

- **StartGrowingCorrutines**: Funció encarregada de iniciar les corrutines de creixement, depenent si s'ha de crèixer la llavor o els fruits.

```
private void StartGrowingCorrutines(){
    if (!CanGrowFruits() && CanGrow()) StartCoroutine(GrowingPlant());
    else if (CanGrowFruits()) StartCoroutine(GrowingFruits());
    else {
        SeedPot seedPot = GetComponentInParent<SeedPot>();
        if (seedPot) seedPot.CanRecolect(true);
    }
}
```

- **InstantiateNextLevel**: Funció encarregada de instanciar el següent nivell de la llavor i destruir l'anterior. Si és l'últim nivell de la llavor i no està en un test, instancia l'objecte i destrueix el SeedPlaceHolder. També inicia les corrutines de creixemnt si la llavor pot seguir creixent. Tots els objectes instanciats es carreguen a traves del sistema d'Addressables.

```
private void InstantiateNextLevel(){
    if (spawnLastPrefab){
        playerStats.currentIsland.PlaceItem(seed.finalLevelPrefabs
            [Random.Range(0, seed.finalLevelPrefabs.Count)],
            seedPrefab.gameObject.transform.position,
        Quaternion.Euler(seedPrefab.gameObject.transform.rotation.x,
        Random.Range(0f, 360f),seedPrefab.gameObject.transform.rotation.z));
        if (!Addressables.ReleaseInstance(this.gameObject)) Destroy(this.gameObject);
        return;
    }
    if (IsMaxLevel()){
        if (!seed.multipleRecolection){
            if (seedPrefab != null)
            if (!Addressables.ReleaseInstance(seedPrefab)) Destroy(seedPrefab);
            if (seed.finalLevelPrefabs.Count > 0){
            Addressables.InstantiateAsync(seed.growthLevelPrefabs[actualGrowthLevel],
                this.transform.position,
                Quaternion.identity, this.transform).Completed += (prefab) =>{
                    if (seedPrefab != null)
                    if (!Addressables.ReleaseInstance(seedPrefab))
                        Destroy(seedPrefab);
                    seedPrefab = prefab.Result;
                    spawnLastPrefab = true;  StartCoroutine(GrowingPlant());
                };
            }
            else{
               Addressables.InstantiateAsync
               (seed.growthLevelPrefabs[actualGrowthLevel],
                    this.transform.position, Quaternion.identity,
                    this.transform).Completed += (prefab) =>{
                        if (seedPrefab != null)
                        if (!Addressables.ReleaseInstance(seedPrefab))
                            Destroy(seedPrefab);
                        seedPrefab = prefab.Result;
                        SeedPot seedPot = GetComponentInParent<SeedPot>();
                        if (seedPot) seedPot.CanRecolect(true);
                    };
```

```
            }
        }
        else{
            Addressables.InstantiateAsync(seed.growthLevelPrefabs[actualGrowthLevel],
                this.transform.position, Quaternion.identity, this.transform)
                .Completed += (prefab) =>{
                    if (seedPrefab != null)
                    if (!Addressables.ReleaseInstance(seedPrefab))
                       Destroy(seedPrefab);
                    seedPrefab = prefab.Result;  StartGrowingCorrutines();
                };
        }
    }
    else{
        Addressables.InstantiateAsync(seed.growthLevelPrefabs[actualGrowthLevel],
            this.transform.position, Quaternion.identity, this.transform)
        .Completed += (prefab) =>{
            if (seedPrefab != null)
            if (!Addressables.ReleaseInstance(seedPrefab)) Destroy(seedPrefab);
            seedPrefab = prefab.Result;  StartGrowingCorrutines();
            };
    }
}
```

6.8.4.2 SeedPot

Dins de la gestió de llavors tenim també els testos. Aquests interactuen directament amb la llavor que es planta en ells. Per gestionar el comportament dels testos, tenim la classe SeedPot que s'encarrega de interactuar amb la llavor plantada.

Model de la classe SeedPot:

SeedPot
+seedPlaceHolder: AssetReference +buffGrowthSpeed: float +actualSeed: Seed -plantInteraction: InteractionSO -pickUpInteraction: InteractionSO -removeInteraction: InteractionSO -upgradeInteraction: InteractionSO -interactuable: Interactuable -upgradeable: Upgradeable
-OnEnable(): void -OnDisable(): void +Plant(SeedSo): void +Load(SeedData seedInfo): void +CanPlant(): bool +CanRecolect(bool can): void +PickUp(): void +RemoveSeed(): void +ResetInteractions(): void

Atributs de SeedPot:

+ **seedPlaceHolder**: Referència a l'objecte SeedPlaceHolder
+ **buffGrowthSpeed**: Float que determina l'augment de velocitat de creixement que tindrà la llavor plantada
+ **actualSeed**: Objecte de la classe Seed de la llavor actual plantada
- **plantInteraction**: Objecte de la classe InteractionSO de tipus plantació
- **pickUpInteraction**: Objecte de la classe InteractionSO de tipus recol·lectar
- **upgradeInteraction**: Objecte de la classe InteractionSO de tipus millorar
- **removeInteraction**: Objecte de la classe InteractionSO de tipus eliminar
- **interactuable**: Referència al component Interactuable de l'objecte
- **upgradeable**: Referència al component Upgradeable de l'objecte

Mètodes de SeedPot:

- **OnEnable**, **OnDisable**: Funcions cridades per el Unity quan l'objecte s'activa i es desactiva. Utilitzades per subscriure's i desubscriure's als esdeveniments, reiniciar valors i obtenir referències als components dels objectes.

```
private void OnEnable(){
    advanceAgeEvent.OnEventRaised += AdvanceAge;
    upgradable = GetComponent<Upgradeable>();
    interactuable = GetComponent<Interactuable>();
    interactuable.interactions.Clear();
    interactuable.interactions.Add(plantInteraction);
    interactuable.interactions.Add(removeInteraction);
    if (upgradable && upgradable.CanUpgrade())
        interactuable.interactions.Add(upgradeInteraction);
}

private void OnDisable(){
    advanceAgeEvent.OnEventRaised -= AdvanceAge;
}
```

- **Plant**, **Load**: Funcions encarregades de plantar la llavor entrada per paràmetre instanciant l'objecte SeedPlaceHolder i plantant-li la llavor. A més, reinicia les interaccions disponibles. Les dos tenen la mateixa estructura però Load crida a la funció Load de la classe Seed. A continuació es mostra la funció Plant com a exemple.

```
public void Plant(SeedSO seedInfo){
    Addressables.InstantiateAsync(seedPlaceHolder, transform.position,
        Quaternion.identity, this.transform).Completed += (obj) =>{
            actualSeed = obj.Result.GetComponent<Seed>();
            actualSeed.Plant(seedInfo, buffGrowthSpeed);
    };
```

```
    ResetInteractions();
 }
```

- **CanPlant**: Funcio que determnia si es pot plantar al test.

```
public bool CanPlant() => actualSeed == null;
```

- **CanRecolect**: Funció que estableix que la llavor ja pot ser collida, establint les noves interaccions.

```
public void CanRecolect(bool can){
    interactuable.interactions.Clear();
    if (can) interactuable.interactions.Add(pickUpInteraction);
    interactuable.ChangeInteraction();
}
```

- **PickUp**: Funció per a recollir la llavor actual

```
public void PickUp() => actualSeed.Recolect();
```

- **RemoveSeed**: Funció que canvia les interaccions possibles després de recolectar la llavor.

```
public void RemoveSeed(){
    interactuable.interactions.Clear();
    interactuable.interactions.Add(plantInteraction);
    interactuable.interactions.Add(removeInteraction);
    if (upgradable && upgradable.CanUpgrade())
       interactuable.interactions.Add(upgradeInteraction);
    interactuable.ChangeInteraction();
}
```

- **ResetInteraction**: Funció encarregada de resetejar les interaccions possibles

```
public void ResetInteractions(){
    interactuable.interactions.Clear();
    interactuable.ChangeInteraction();
}
```

6.8.5 AgeManager

L'època actual de la partida és una part molt important del joc, ja que interacciona indirectament amb molts sistemes. A més, afecta directament a l'estat de la casa del jugador. Per gestionar les diferents èpoques tenim un seguit d'objectes AgeSo que són ScriptableObjects encarregats d'emmagatzemar la informació específica de cada època.
L'època actual es gestiona amb la classe AgeManager, que també és un ScriptableObject encarregat d'avançar d'època en època. Per últim, WorldAgeManager és la classe encarregada de gestionar les modificacions visuals depenent de l'época actual (com la casa del jugador).

6.8.5.1 AgeSO

Classe de tipus ScriptableObject per emmagatzemar la informació de cada època del joc.

```
public class AgeSO : ScriptableObject
{
    [SerializeField] public Age Age = Age.None;
    [SerializeField] public List<ItemContainer> itemsNeeded;
    [SerializeField] public AssetReference prefab;
    [SerializeField] public List<ItemBase> ItemsCanCraft;
}
```

Figura 6.8.5.1: Classe AgeSO

Com es pot veure en la Figura 6.8.5.1, la classe té 4 atributs:

+ **Age**: Tipus de l'Enum Age (Figura 6.8.5.2)
+ **itemsNeeded**: Llistat d'objectes ItemContainer on hi ha la informació dels recursos necessaris per avançar d'època
+ **prefab**: Referència a l'objecte de l'època (la casa del jugador)
+ **ItemsCanCraft**: Llistat d'objectes ItemBase. Objectes que el jugador podrà crear en aquesta època

```
public enum Age
{
    None,
    Wood,
    Stone,
    Iron
}
```

Figura 6.8.5.2: Enumerador Age

6.8.5.2 AgeManager

Classe ScriptableObject encarregada de gestionar l'època actual de la partida i és l'objecte el qual totes les altres classes que necessiten obtenir informació de l'època actual hauran de referenciar.

Model de la classe AgeManager:

AgeManager
-age: AgeSo -ages: AgeSO[] -advanceAgeEvent: VoidEvenetSO -loadAgeEvent: VoidEventSO -inventoryManager: InventoryManager -actualAgeIndex: int
-OnDisable(): void +Load(int loadedAge): void +CanUpgrade(): bool +AdvanceAge(): void +NextAge(): AgeSO

Atributs de AgeManager:

- **age**: Objecte de tipus AgeSO que representa l'època actual
- **ages**: Llistat d'objectes de tipus AgeSO que representen totes les èpoques possibles del joc
- **advanceAgeEvent**: Objecte de tipus VoidEventSO per a l'esdeveniment d'avançar d'èopca
- **loadAgeEvent**: Objecte de tipus VoidEventSO per a l'esdeveniment de carregar l'època actual
- **inventoryManager**: Objecte de tipus InventoryManager per a la comunicació amb l'inventari del jugador
- **actualAgeIndex**: Int que determina l'índex de l'època en el llistat d'èpoques

Mètodes de AgeManager:

- **OnDisable**: Funció cridada per el Unity quan l'objecte es desactiva. Utilitzada per reiniciar els atributs.

```
private void OnDisable(){
    actualAgeIndex = 0;  _age = ages[actualAgeIndex]; }
```

+ **Load**: Funció que assigna l'època de la partida a l'època del llistat d'èpoques amb l'index entrat per parametre.

```
public void Load(int loadedAge){
    actualAgeIndex = loadedAge; _age = ages[actualAgeIndex]; loadAgeEvent.RaiseEvent();
}
```

+ **CanUpgrade**: Funció que determina si el jugador pot avançar d'època depenent dels ítems que tingui

```
public bool CanUpgrade(){
    if (actualAgeIndex < ages.Length-1)
    return inventoryManager.HasItems(ages[actualAgeIndex+1].itemsNeeded);
    return false;
}
```

+ **AdvanceAge**: Funció qua avança l'època actual de la partida, si es pot.

```
public void AdvanceAge(){
    if (actualAgeIndex < ages.Length && CanUpgrade()){
        if (inventoryManager.RemoveItems(ages[actualAgeIndex + 1].itemsNeeded)){
                actualAgeIndex++;
                _age = ages[actualAgeIndex];
                advanceAgeEvent.RaiseEvent();
        }
    }
}
```

+ **NextAge**: Funció que retorna l'objecte AgeSo de la següent època

```
public AgeSO NextAge({
        if (actualAgeIndex < ages.Length) return ages[actualAgeIndex + 1];
        else return null;
}
```

6.8.5.3 WorldAgeManager

Classe que s'encarrega de la gestió de l'objecte que representa visualment la casa del jugador.

```
[SerializeField] private GameObject home;
[SerializeField] private AgeManager actualAge;
[SerializeField] private VoidEventSO loadAge = default;
[SerializeField] private VoidEventSO advanceAge = default;
private GameObject actualAgePrefab;
```

Figura 6.8.5.3.1: Atributs de la classe WorldAgeManager

Com podem veure en la Figura 6.8.5.3.1, la classe WorldAgeManager té 5 atributs:

- **home**: GameObject de la jerarquía de l'escena on d'instància la casa del jugador
- **actualAge**: Objecte AgeManager amb la informació de l'època
- **loadAge**: Objecte VoidEventSO per a l'esdeveniment de càrrega de l'època
- **advanceAge**: Objecte VoidEventSO per a l'esdeveniment de avançar d'època
- **actualAgePrefab**: GameObject de l'actual casa del jugador

I com atributs tenim: Load i AdvanceAge. Cada un d'ells té la mateixa estructura i, simplement, eliminen tots els objectes fills dins de l'atribut home i instancien la nova època. Load instancia l'època actual que hi ha a la variable actualAge, ja que ja ha sigut carregada. AdvanceAge instancia la següent época. En la Figura 6.8.5.3.2 podem veure la funció Load com a exemple.

```
public void Load()
{
    foreach (Transform transform in home.transform)
    {
        if (!Addressables.ReleaseInstance(transform.gameObject))
            Destroy(transform.gameObject);
    }
    Addressables.InstantiateAsync(actualAge.Age.prefab,
        home.transform.position, Quaternion.identity, home.transform)
        .Completed += (obj) => { actualAgePrefab = obj.Result.gameObject; };
}
```

Figura 6.8.5.3.2: Funció Load de la classe WorldAgeManager

6.9. Sistema d'interacció i esdeveniments

Per portar el control de les interaccions que el jugador pot fer amb els diversos objectes del joc, hem creat un sistema per a gestionar totes les interaccions i a l'hora, que sigui molt modular per a poder afegir o modificar interaccions en un futur.
El sistema està dividit en varies parts:

- ScriptableObjects amb la informació de cada interacció
- Interfícies d'interacció per a les diferents classes d'objectes interactuables
- Zona d'interacció del jugador encarregada de gestionar els objectes amb els quals el jugador pot interactuar
- Gestor d'interaccions per a poder portar el control de totes les interaccions possibles que tingui el jugador i d'interactuar amb elles

A més, hem definit 3 enumeradors:

- **InteractionType**: Enumerador per determinar el tipus d'interacció
- **InteractionBeaheaviour**: Enumerador per determinar el tipus d'acció a l'interactuar (prémer el botó o mantenir pulsat).
- **InteractionButton**: Enumerador per determinar el botó que s'utilitzarà per interactuar (Primari, secundari o terciari)

```
public enum InteractionType {
    None = 0, Plant, Upgrade,
    Repair, Craft, PickUp,
    Cook, Teleport, Remove
};
```

Figura 6.9.1: Enumerador de tipus d'interacció

```
public enum InteractionBeaheaviour
{
    PressButton, HoldButton
}
21 referencias
public enum InteractionButton
{
    None, Primary, Secondary, Tertiary
}
```

Figura 6.9.2: Enumerador de comportament i botons d'interaccions

6.9.1 InteractionSO

Classe ScriptableObject per emmagatzemar la informació de cada interacció.

```
public class InteractionSO : ScriptableObject
{
    public string interactionName;
    public InteractionType interactionType;
    public InteractionBeaheaviour interactionBeaheaviour;
    public float progressBarSeconds = 2f;
    public InteractionButton interactionButton = InteractionButton.Primary;
}
```

Figura 6.9.1.1: Classe InteractionSO

Com es pot veure en la Figura 6.9.1.1, la classe InteractionSO està formada simplement per 5 atributs:

- **interactionName**: String per determinar el nom de l'interacció
- **interactionType**: Enum de tipus InteractionType per determinar el tipus de l'interacció
- **InteractionBehaviour**: Enum de tipus InteractionBehaviour per determinar el tipus d'acció a l'interactuar
- **progressBarSeconds**: Float per determinar els segons que s'ha de mantenir pulsat el botó si l'interacció té l'acció de mantenir pulsat
- **interactionButton**: Enum de tipus InteractionButton per determinar el botó que tindrà assignat l'interacció

D'aquesta manera podem crear diferents tipus d'interacció d'una manera senzilla i escalable. En la Figura 6.9.1.2 es poden veure totes les animacions que s'han creat al llarg del desenvolupament, algunes s'han descartat o no s'han arribat a fer servir en la versió actual del projecte.

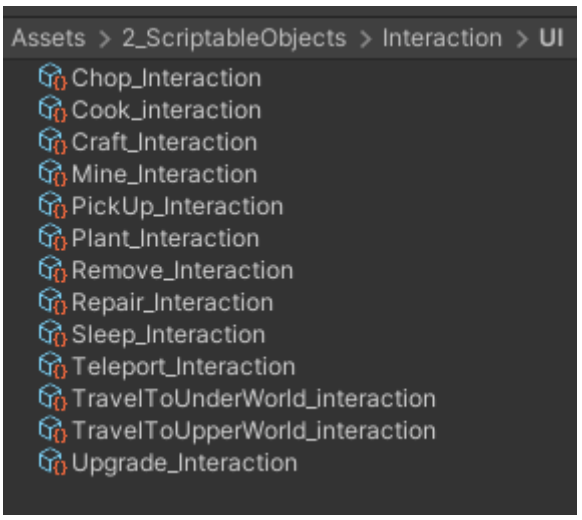


Figura 6.9.1.2: Llistat d'objectes de tipus InteractionSO del projecte

6.9.2 Interfícies d'interacció

Tot objecte que, a l'interaccionar amb ell, modifiqui l'estat o el del jugador, tindrà una interfície específica de l'interacció. Així doncs, tots els objectes amb les interaccions de tipus Remove, PickUp, Repair i Teleport hauran d'implementar la interfície corresponent.

6.9.2.1 IDestructible

Tot objecte que es pugui destruir o tingui l'interacció de tipus Remove, implementarà la interfície IDestructible. Com es pot veure en la Figura 6.9.2.1.1, la interfície només té un delegat que serà l'encarregat d'avisar als objectes que es subscriuin que l'objecte ha estat destruit, passant-li el propi objecte.

```
public interface IDestructible
{
    public event OnDestroyDelegate OnDestroyEvnt;
    public delegate void OnDestroyDelegate(GameObject obj);
}
```

Figura 6.9.2.1.2: Interfície IDestructible

6.9.2.2 IPickable

Tots els objectes que es puguin recolectar o que tinguin una interacció de tipus PickUp implementaran la interfície IPickable. Com es pot veure en la Figura 6.9.2.2.1, la interfície només té 1 mètode PickUp, en el que cada classe aplicarà la lògica necessària.

```
public interface IPickable
{
    5 referencias
    void PickUp();
}
```

Figura 6.9.2.2.1: Interfície IPickable

6.9.2.3 IReparable

Tots els objectes que es puguin reparar o que tinguin una interacció de tipus Repair implementaran la interfície IReparable. Com es pot veure en la Figura 6.9.2.3.1, la interfície té 3 mètodes:

- **Repair**: Funció que es cridarà per realitzar l'acció de reparar. Aquí és on cada objecte aplicarà la seva pròpia lògica.
- **TryToRepair**: Funció que determina si el jugador pot reparar l'objecte depenent dels recursos requerits.
- **IsRepaired**: Funció que determina si un objecte ja està reparat.

```
public interface IReparable
{
    1 referencia
    void Repair();

    2 referencias
    bool TryToRepair();

    3 referencias
    bool IsRepaired();

}
```

Figura 6.9.2.3.1: Interfície IReparable

6.9.2.4 ITeleport

Tots els objectes que tinguin una interacció de tipus Teleport implementaran la interfície ITeleport. Com es pot veure en la Figura 6.9.2.4.1, la interfície només té 1 mètode teleport, en el que cada classe aplicarà la lògica necessària.

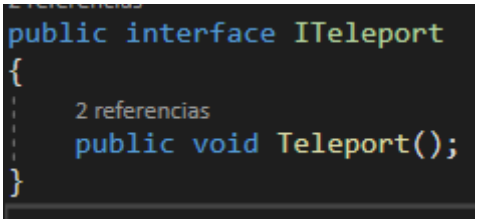


Figura 6.9.2.4.1: Interfície ITeleport

Amb aquestes interfícies agilitzem la gestió global de les interaccions fent que sigui molt més ràpid i fàcil interactuar sobre un objecte.

6.9.3 Colliders de interacció

En aquest Apartat trobem la gestió dels objectes amb els que pot interactuar el jugador. Per determinar-ho, el objecte jugador sempre tindrà davant seu una zona on tot objecte que tingui el component Interactuable serà enviat al gestor global d'interaccions.
En els següents apartats veurem amb més detall la zona d'interacció del jugador (ZoneTriggerController) i el component Interactuable.

6.9.3.1 ZoneTriggerController

ZoneTriggerController és la classe encarregada de sel·leccionar els objectes que el jugador está interactuant. Ho fa gràcies al component Box Collider que té l'objecte.

```
public class BoolEvent : UnityEvent<bool, GameObject>
{

}

Script de Unity | 0 referencias
public class ZoneTriggerController : MonoBehaviour
{
    [SerializeField] private BoolEvent _enterZone = default;
    Mensaje de Unity | 0 referencias
    private void OnTriggerEnter(Collider other)
    {
        Interactuable interaction = other.gameObject.GetComponent<Interactuable>();
        if (interaction)
        {
            _enterZone.Invoke(true, other.gameObject);
        }
    }
    Mensaje de Unity | 0 referencias
    private void OnTriggerExit(Collider other)
    {
        Interactuable interaction = other.gameObject.GetComponent<Interactuable>();
        if (interaction)
        {
            _enterZone.Invoke(false, other.gameObject);
        }
    }
}
```

Figura 6.9.3.1.1: Classe ZoneTriggerController

Com es pot veure en la Figura 6.9.3.1.1, la classe escolta els events llençats pel Unity OnTriggerEnter i OnTriggerExit. Aquests events es llencen quan un objecte amb component Collider entra o surt de la zona. Així doncs, quan un objecte entri o surti, la zona llençarà l'esdeveniment públic amb 2 paràmetres: l'objecte que ha entrat o sortit i un bool per determinar si ha entrat o ha sortit. Aquest esdeveniment públic (atribut enterZone) està relacionat amb el controlador global InteractionManeger (component de l'objecte player) via inspector.

6.9.3.2 Interactuable

Tot objecte interactuable tindrà el component Interactuable. Aquesta classe va sorgir de la necessitat de tenir interaccions dinàmiques. Per exemple el test on, quan no hi ha cap llavor, té unes interaccions, quan la llavor està creixent, no en té cap i quan la llavor ha crescut, té unes interaccions que depenen de la llavor. Per tant, aquesta classe s'encarrega de gestionar els canvis d'interaccions, a més de tractar el comportament on el jugador està interactuant amb un objecte i les interaccions canvien just en aquell moment.

```
public class Interactuable : MonoBehaviour
{
    [SerializeField] public List<InteractionSO> interactions = new List<InteractionSO>();
    [HideInInspector] public InteractionManager interactionManager;
    public bool showInteractionUIButton = true;
    4 referencias
    public void ChangeInteraction()
    {
        if (interactionManager != null)
            interactionManager.ChangePotentialInteraction(interactions);
    }

    1 referencia
    public void SetManager(InteractionManager interactionManager)
    {
        this.interactionManager = interactionManager;
    }

    1 referencia
    public void RemoveManager()
    {
        this.interactionManager = null;
    }
}
```

Figura 6.9.3.2.1:: Classe Interactuable

Com es pot veure en la Figura 6.9.3.2.1, la classe Interactuable té 3 atributs:

- **interactions**: Llistat d'objectes InteractionSO. Interaccions actuals que el jugador podrà realitzar en aquest objecte.
- **interactionManager**: Atribut de tipus InteractionManager encarregat de guardar la referència al controlador global d'interaccions. Aquesta referència es fa servir per saber si el jugador està interaccionant amb aquest objecte.
- **showInteractionUIButton**: Bool per definir com es mostrarà visualment la interacció (sobre el jugador o dins el panell de l'objecte a interactuar)

A part dels mètodes d'assignar o eliminar la referència a l'InteracionManager, també té el mètode ChangeInteraction. En aquest mètode simplement es canvien les interaccions possibles que es poden efectuar a l'InteractionManager amb el llistat actual d'interaccions. D'aquesta manera, si el jugador està just interaccionant amb l'objecte, podrà veure les noves interaccions possibles.

6.9.5 InteractionManager

La classe InteractionManager és el controlador global de les interaccions del jugador. Forma part dels components del jugador i s'encarrega de portar el recull de totes les possibles interaccions que pugui fer el jugador amb els objectes que està interaccionant. A més, és la classe encarregada d'escoltar i llençar tots els esdeveniments relacionats amb les interaccions i comunicar a la interfície l'estat per poder donar un feedback al jugador.

Model de la classe InteractionManager:

Interaction Manager
-interactionPerformed: InteractionEventSO -playerStats: PlayerStats -ageManager: AgeManager -inputReader: InputReader -seedPotForPlanting: GameObjectEventSO -toggleInteractionUI: InteractionUiEventSO -toggleCraftPanelUI: VoidEventSO -toggleSeedPanelUI: VoidEventSO -removeInteraction: VoidEventSO -potentialInteractions: LinkedList<Interaction>
-OnEnable(): void -OnDisable(): void +OnTriggerChangeDetected(bool entered, GameObject obj) -AddPotentialInteraction(GameObject obj) -RemovePotentialInteraction(GameObject obj) -OnInteractionPerformed(InteractionSO interaction) -RequestUpdateUI(bool visible) +ChangePotentialInteraction(List<InteractionSO> newInteractions) +RemoveFirstInteraction

Atributs de InteractionManager:

- **InteractionPerformed**: Objecte de tipus InteractionEventSO per a l'esdeveniment d'interacció executada
- **playerStats**: ScriptableObject on hi ha la informació actual del jugador
- **ageManager**: Objecte de tipus AgeManager per a la comunicació amb l'època actual
- **inputReader**: ScriptableObject encarregat de llençar els esdeveniments de l'input
- **seedPotForPlanting**: Objecte del tipus GameObjectEventSO per a l'esdeveniment

d'interacció amb un test

- **toggleInteractionUI**: Objecte del tipus InteractionUIEventSO per a l'esdeveniment de mostrar el panell d'interaccions actuals
- **toggleCraftPanelUI**: Objecte del tipus VoidEventSO per a l'esdeveniment d'ensenyar o amagar el panell de creació
- **toggleSeedPanelUI**: Objecte del tipus VoidEventSO per a l'esdeveniment d'ensenyar o amagar el panell de plantació
- **removeInteraction**: Objecte del tipus VoidEventSO per a l'esdeveniment d'eliminar una interacció
- **potentialInteractions**: Llistat d'objectes de tipus Interaction amb les possibles interaccions que pot fer el jugador en un moment donat

Mètodes de InteractionManage:

- **OnEnable, OnDisable**:Funcions cridades pel Unity quan l'objecte s'activa i es desactiva. Utilitzades per subscriure's i desubscriure's als esdeveniments.

```
private void OnEnable(){
    interactionPerformed.OnEventRaised += OnInteractionPerformed;
    removeInteraction.OnEventRaised += RemoveFirstInteraction;
}

private void OnDisable(){
    interactionPerformed.OnEventRaised -= OnInteractionPerformed;
    removeInteraction.OnEventRaised -= RemoveFirstInteraction;
}
```

+ **OnTriggerChangeDetected**: Funció cridada per la classe ZoneTriggerControler esmentada en l'Aapartat 6.9.3.1. Afegeix o elimina una interacció amb l'objecte depenent del bool entrat

```
public void OnTriggerChangeDetected(bool entered, GameObject obj){
    if (entered)AddPotentialInteraction(obj);
    else RemovePotentialInteraction(obj);
}
```

- **AddPotentialInteraction:** Funció encarregada d'afegir una nova interacció potencial amb l'objecte entrat per paràmetre. Crea un objecte de tipus Interacton amb totes les possibles interaccions de l'objecte i l'afegeix al llistat de potentialInteractions com a primer valor.

```
private void AddPotentialInteraction(GameObject obj){
    Interaction newPotentialInteraction = new Interaction(obj);
    Interactuable interactuable = obj.GetComponent<Interactuable>();
    foreach (var interaction in interactuable.interactions){
        if (interaction.InteractionType == InteractionType.Repair)
            if (!obj.GetComponent<IReparable>().IsRepaired())
                newPotentialInteraction.interactions.Add(interaction);
        else if (interaction.InteractionType == InteractionType.Upgrade)
                if (ageManager.CanUpgrade())
                    newPotentialInteraction.interactions.Add(interaction);
        else newPotentialInteraction.interactions.Add(interaction);
    }
    if (newPotentialInteraction.HasInteractions()){
        _potentialInteractions.AddFirst(newPotentialInteraction);
        interactuable.SetManager(this);
        if (interactuable.showInteractionUIButton)RequestUpdateUI(true);
    }
}
```

- **RemovePotentialInteraction:** Funció que elimina la possible interacció del jugador amb l'objecte entrat per paràmetre.

```
private void RemovePotentialInteraction(GameObject obj){
    LinkedListNode<Interaction> currentNode = _potentialInteractions.First;
    while (currentNode != null){
        if (currentNode.Value.interactableObject == obj){
            currentNode.Value.interactableObject
            .GetComponent<Interactuable>().RemoveManager();
            _potentialInteractions.Remove(currentNode);
            break;
        }
        currentNode = currentNode.Next;
    }
    RequestUpdateUI(_potentialInteractions.Count > 0);
}
```

- **OnInteractionPerformed:** Funció encarregada de gestionar totes les interaccions que es duen a terme. Depenent del tipus d'interacció entrada per paràmetre, comprobarà si el primer objecte amb el que el jugador pot interactuar té la interacció. Després, efectuarà la interacció modificant l'estat de l'objecte o notificant mitjançant esdeveniments que la interacció ha estat realitzada.

```
private void OnInteractionPerformed(InteractionSO interaction){
    if (_potentialInteractions.Count == 0) return;
    foreach (var potential in _potentialInteractions.First.Value.interactions){
       if (potential == interaction){
           switch (potential.InteractionType){
               case InteractionType.None: break;
               case InteractionType.Plant:
                   _potentialInteractions.First.Value.interactableObject
                       .GetComponent<SeedPot>().ResetInteractions();
                   seedPotForPlanting.RaiseEvent
                       (_potentialInteractions.First.Value.interactableObject);
                   toggleSeedPanelUI.RaiseEvent();
```

```
                    inputReader.EnableUIInput();
                    return;
                case InteractionType.Upgrade:
                    Upgradeable upgradeableObject = _potentialInteractions.First.Value
                       .interactableObject.GetComponent<Upgradeable>();
                    if (upgradeableObject && upgradeableObject.CanUpgrade())
                        upgradeableObject.Upgrade();
                    else ageManager.AdvanceAge();
                    RemovePotentialInteraction(_potentialInteractions
                        .First.Value.interactableObject);
                   break;
               case InteractionType.Repair:
                    if (_potentialInteractions.First.Value.interactableObject
                        .GetComponent<IReparable>().TryToRepair())
                        RemovePotentialInteraction(_potentialInteractions
                        .First.Value.interactableObject);
                    break;
               case InteractionType.Craft:
                   if (!playerStats.isCrafting) toggleCraftPanelUI.RaiseEvent();
                    break;
               case InteractionType.Cook:
                    if (!playerStats.isCooking)
                        _potentialInteractions.First.Value.interactableObject
                        .GetComponent<Campfire>().CookDishes();
                    break;
                case InteractionType.PickUp:
                        _potentialInteractions.First.Value.interactableObject
                        .GetComponent<IPickable>()?.PickUp();
                    if (!_potentialInteractions.First.Value.
                         interactableObject.CompareTag("SeedPot"))
                         RemovePotentialInteraction(_potentialInteractions.
                                                    First.Value.interactableObject);
                    return;
                case InteractionType.Teleport:
                    _potentialInteractions.First.Value.interactableObject
                        .GetComponent<ITeleport>()?.Teleport();
                    if (_potentialInteractions.First.Value.interactableObject
                        .GetComponent<Teleporter>().setPlayerTeleportedState)
                        playerStats.teleported = true;
                    break;
                case InteractionType.Remove:
                    _potentialInteractions.First.Value.interactableObject
                        .GetComponent<Removable>()?.Remove();
                    break;
                default: break;
                }
            }
        }
    }
```

- **RequestUpdateUI:** Funció encarregada de llençar l'esdeveniment de mostrar el panell d'interacció o d'amagar-ho depenent de la variable bool entrada per paràmetre

```
private void RequestUpdateUI(bool visible){
    if (visible) toggleInteractionUI.RaiseEvent(true,
                 _potentialInteractions.First.Value.interactions);
    else toggleInteractionUI.RaiseEvent(false, Interaction.NoneInteraction());
}
```

- **ChangePotentialInteraction:** Canvia les interaccions disponibles del primer objecte de la llista de potentialInteractions amb el llistat d'interaccions entrades per paràmetre

```
public void ChangePotentialInteraction(List<InteractionSO> newInteractions){
    _potentialInteractions.First.Value.interactions.Clear();
    foreach (var newInteraction in newInteractions)
        _potentialInteractions.First.Value.interactions.Add(newInteraction);
    RequestUpdateUI(true);
}
```

- **RemoveFirstInteraction**: Funció que elimina el primer objecte del llistat potentialInteractions

```
public void RemoveFirstInteraction(){
  RemovePotentialInteraction(_potentialInteractions.First.Value.interactableObject);
}
```

6.10. Sistema de combat i enemics

El combat comprèn tant tot el que el jugador pot atacar, com tot el que pot atacar al jugador. Per ara, el jugador podrà ser atacat pels enemics, en canvi el jugador podrà atacar tant a enemics com els recursos de món mencionats en l'Apartat 6.8.2. Per a fer-ho, utilitzarà les diferents eines que té a l'abast i, depenent l'eina equipada i l'entitat que es vol atacar, s'efectuarà el cop.

6.10.1 Damageable

Tot enemic del joc té un component de la classe Damageable. Aquest component s'encarrega de rebre els cops del jugador.

```
public int health;
public int currentLife = default;
public event OnGetHitDelegate OnGetHit;
public delegate void OnGetHitDelegate();
public event OnDieDelegate OnDie;
public delegate void OnDieDelegate();
```

Figura 6.10.1.1: Atributs de la classe Damageable

Com es pot veure en la Figura 6.10.1.1, la classe Damageable te:

+ **health**: Int, que determina la vida màxima del objecte
+ **currentLife**: Int, que determina la vida actual de l'objecte
+ **OnGetHit**: Delegat per a l'esdeveniment de rebre un cop
+ **OnDie**: Delegat per a l'esdeveniment de mort

```
public void GetHit(int damage){
    currentLife -= damage;
    if (OnGetHit != null) OnGetHit();
    if (currentLife <= 0)
        if (OnDie != null) OnDie();
}
```

Figura 6.10.1.2: Mètode GetHit de la classe Damageable

A més, la classe Tool té únicament el mètode GetHit que es pot veure en la Figura 6.10.1.2. La funció és rebre els cops del jugador i restar el valor del cop entrat per paràmetre a la vida actual. A més, notifica als subscriptors dels delegats quan es rep un cop i quan es mor.

6.10.2 Tool

La classe Tool és l'encarregada de gestionar les col·lisions de les eines amb els altres objectes.

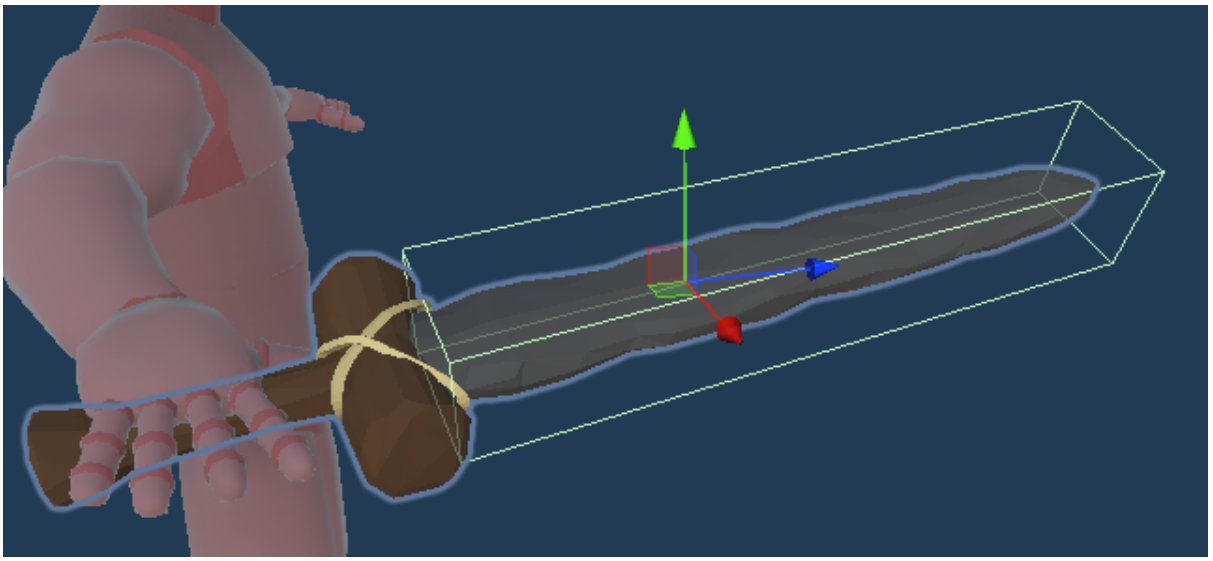


Figura 6.10.2.1: Espasa del jugador amb colisionador

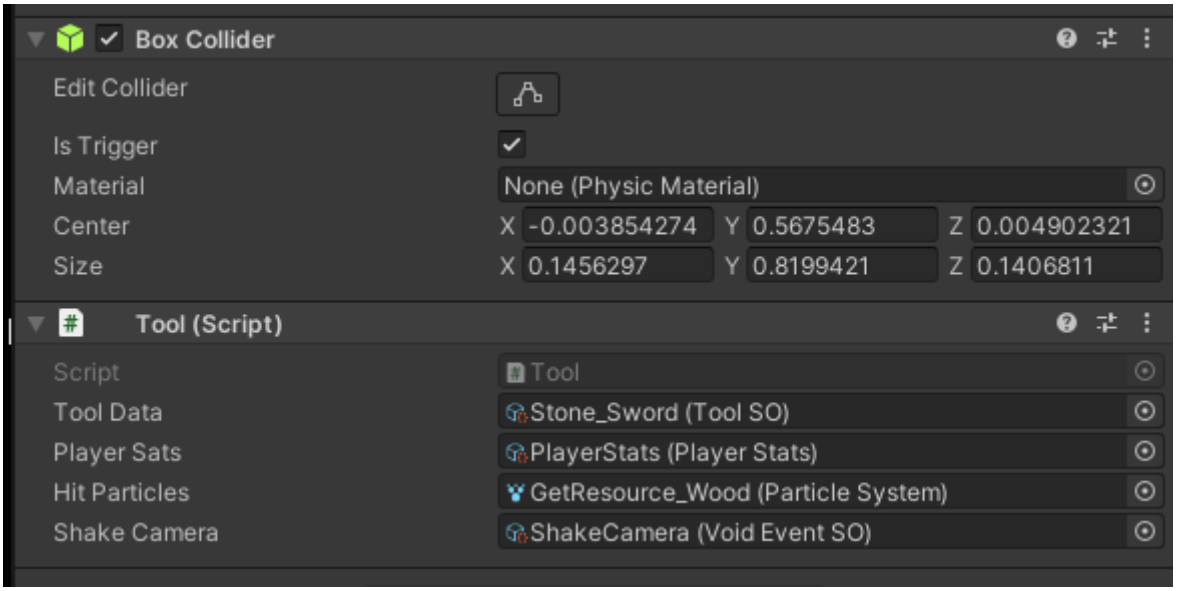


Figura 6.10.2.2: Components del objecte espasa

Com es pot veure en les Figures 6.10.2.1 i 6.10.2.2, Tool és un component que utilitza el component Box Collider del objecte per a escoltar les col·lisions. Aquesta mateixa estructura es repeteix per les diferents eines que el jugador té disponibles, variant la forma del colisionador per a que s'adapti a la forma per on l'eina pot efectuar un cop.

```
public ToolSO toolData;
public PlayerStats playerSats;
public ParticleSystem hitParticles;
[SerializeField] VoidEventSO ShakeCamera = default;
```

Figura 6.10.2.3: Atributs de la classe Tool

Com es pot veure en la Figura 6.10.2.3, la classe Tool té els següents atributs:

+ **toolData**: ScriptableObject de tipus ToolSO on hi ha emmagatzemada la informació de l'einea
+ **playerStats**: ScriptableObject de tipus PlayerStats on hi ha emmagatzemada la informació del jugador
+ **hitParticles**: Sistema de partícules per a quan es produeix el cop
- **shakeCamera**: Objecte de tipus VoidEventSO per a l'esdeveniment de sacsejament de la càmera.

```
private void OnTriggerEnter(Collider other){
    if (playerSats.canHit){
        World_Resource worldResource = other.GetComponent<World_Resource>();
        if (worldResource && worldResource.data.canBeHittedToGather){
            Instantiate(hitParticles, transform.position, transform.rotation);
            ShakeCamera.RaiseEvent();
            if (toolData.type == worldResource.data.toolNeeded.type
                && toolData.material >= worldResource.data.toolNeeded.material)
                worldResource.Hit(toolData.damagePerHit);
        }
        Damageable damageable = other.GetComponent<Damageable>();
        if (damageable && toolData.type == ToolType.Sword && playerSats.isAttacking){
            Instantiate(hitParticles, transform.position, transform.rotation);
            if (playerSats.energyState)
                damageable.GetHit(toolData.damagePerHit * 2);
            else
                damageable.GetHit(toolData.damagePerHit);
        }
    }
}
```

Figura 6.10.2.4: Mètode OnTriggerEnter de la classe Tool

L'únic mètode de la classe Tool és OnTriggerEnter que es pot veure en la Figura 6.10.2.4. Aquest mètode es cridat pel Unity en quan un altre objecte amb un component Collider entra dins de l'àrea del Collider del objecte en el que hi ha el component Tool. La funció del mètode és veure si el jugador pot atacar, o no a l'objecte amb el que l'eina ha col·lisionat. Per fer-ho, comprova si l'objecte té el component World_Resource o Damageable. En el cas

de que algún sigui cert, s'instanciaran les particules, es llençarà l'esdeveniment de sacsejament de la càmera i es notificarà a l'objecte que ha estat atacat si l'eina equipada compleix els requisits per poder atacar-lo.

6.10.3 Enemy

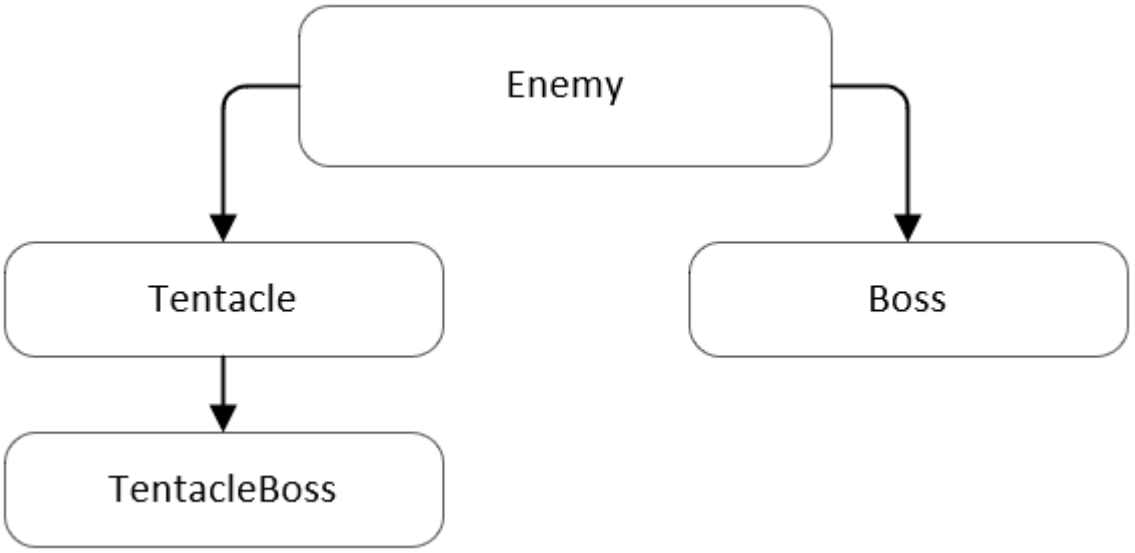


Figura 6.10.3.1: Jerarquía de les classes d'enemic

Com es pot veure en la Figura 6.10.3.1, la classe Enemy és la classe pare per a tots els enemics. La seva funcionalitat és unificar comportaments característics per a tots els enemics els quals són: la detecció del jugador i la posterior rotació cap a ell. A més guarda els atributs que són compartits per a tots els enemics.

```
public float rotationSpeed = 0.5f;
protected bool playerInside = false;
protected Transform playerTransform = null;
protected Damageable damageable;
[SerializeField] public ShakeCameraEventSO shakeCameraEvent = default;
```

Figura 6.10.3.2: Atributs de la clase Enemy

Com es pot veure en la Figura 6.10.3.2, la classe Enemy té els atributs:

+ **rotationSpeed**: Float per determinar la velocitat en la que l'objecte rotarà cap al jugador
+ **playerInside**: Bool per determinar si el jugador està dins del rang de visió
+ **playerTransform**: Transform del jugador

+ **damageable**: Variable de tipus Damageable per guardar la referència al component Damageable de l'objecte
+ **shakeCameraEvent**: Objecte de tipus VoidEventSO per a l'esdeveniment de sacsejament de la càmera.

```
protected virtual void OnTriggerEnter(Collider other){
    if (other.CompareTag("Player")){
        playerInside = true;
        playerTransform = other.gameObject.transform;
        PlayerEnter();
    }
}
Mensaje de Unity | 2 referencias
protected virtual void OnTriggerExit(Collider other){
    if (other.CompareTag("Player")){
        playerInside = false;
        PlayerExits();
    }
}
```

Figura 6.10.3.3: Mètodes OnTriggerEnter i OnTriggerExit de la classe Enemy

Per detectar quan el jugador entra o surt del rang de visió, la classe Enemy té els mètodes OnTriggerEnter i OnTriggerExit que podem veure en la Figura 6.10.3.3. Depenent de si el jugador entra o surt, es llencen les funcions PlayerEnter o PLayerExists, que són funcions virtuals buides per a que cada enemic les pugui sobreescriure amb el comportament en específic.

```
protected virtual void FixedUpdate(){
    if (playerInside) RotateToPlayer();
}
2 referencias
protected virtual void RotateToPlayer(){
    if (playerTransform != null){
        Quaternion targetRotation = Quaternion.LookRotation(
            playerTransform.position - transform.position);
        transform.rotation = Quaternion.Slerp(
            transform.rotation, targetRotation, rotationSpeed * Time.deltaTime);
    }
    else playerInside = false;
}
```

Figura 6.10.3.4: Mètodes FixedUpdate i RotateToPlayer de la classe Enemy

Pel que fa a la rotació cap al jugador, la classe Enemy utilitza les funcions FixedUpdate i RotateToPlayer que podem veure en la Figura 6.10.3.4. FixedUpdate és una funció del Unity cridada cada cada frame amb una taxa fixa, i la seva funció és cridar a la funció de rotació si el jugador està dins el rang de visió. RotateToPlayer utilitza la transformació del jugador per a rotar l'objecte en la direcció adequada.

6.10.4 Tentacle i TentacleBoss

La classe Tentacle hereta de la classe enemic i s'encarrega de donar vida al enemic principal del joc. També implementa la interfície IDestructible.

Model de la classe Tentacle:

Tentacle
+health: int +waitTimeBtwAttacks: float +attackDamage: int +groundSlamParticles: ParticleSystem +tentacleAnimation: Animator +canAttack: bool +CR_running: bool +isAttacking: bool +isParried: bool +canRotate: bool +appearing: bool +canBeInterrupted: bool +OnDestroyEvnt: OnDestroyDelegate
-OnEnable(): void -Start(): void +OnGetHit(): void -FixedUpdate(): void -OnTriggerEnter(Collider other): void -OnTriggerExit(Collider other): void +PlayerHit(): void +Die(): void -DestroyObject(): void -CheckCanAttack(): IEnumerator -Attack(): void +FinishAttack(): void +ShakeCamera(): void +StartAttackAnimation(): void +PlayGroundSlamParticles(): void -SetAppearAnimation(bool state): void -StopAppearAnimation(): void

Atributs de Tentacle:

+ **health**:Int per determinar la vida actual del tentacle
+ **waitTimeBtwAttacks**: Float per determinar el temps d'espera entre atacs
+ **attackDamage**: Int per determinar el valor del cop inflingit
+ **groundSlamParticles**: ParticleSystem per a les partícules d'impacte contra el terra
+ **tentacleAnimation**: Component Animator de l'objecte
+ **canAttack**: Bool per determinar si el tentacle pot atacar
+ **CR_running**: Bool per determinar si la corrutina CheckAttack està activa
+ **isAttacking**: Bool per determinar si el tentacle està atacant
+ **isParried**: Bool per determinar si l'atac del tentacle ha estat contraatacat
+ **canRotate**: Bool per determinar si el tentacle pot rotar
+ **appearing**: Bool per determinar si el tentacle està apareixent
+ **canBeInterrupted**: Bool per determinar si el tentacle pot ser interumpit pels atacs del jugador
+ **OnDestroyEvnt**: Delegat per a l'esdeveniment de mort

Mètodes de Tentacle:

- **OnEnable:** Funció cridada pel Unity quan l'objecte s'activa. Encarregada de inicialitzar les variable, obtenir referències a components i iniciar l'animació Idle en un moment aleatori.

```
private void OnEnable(){
    playerTransform = null; playerInside = false;
    damageable = GetComponentInChildren<Damageable>();
    tentacleAnimation = GetComponent<Animator>();
    AnimatorStateInfo state = tentacleAnimation.GetCurrentAnimatorStateInfo(0);
    tentacleAnimation.Play(state.fullPathHash, -1, Random.Range(0f, 1f));
}
```

- **Start**: Funció cridada pel Unity en el primer frame. Utilitzada per obtenir l'informació i subscriure's als esdeveniments del component Damageable

```
protected override void Start(){
    base.Start();
    if (damageable != null){
        damageable.OnGetHit += OnGetHit;  damageable.OnDie += Die;
        damageable.health = health; damageable.currentLife = health;
    }
}
```

- **OnGetHit**: Funció cridada quan el tentacle rep un atac del jugador. Si el tentacle pot ser interromput, activa l'animació de rebre un atac i assigna els estats corresponents.

```
protected void OnGetHit(){
    if (canBeInterrupted){
        tentacleAnimation.SetTrigger("TakeDamage");
        isAttacking = false; isParried = false;  canRotate = true;
    }
}
```

- **FixedUpdate**: Funció cridada pel Unity en cada frame amb taxa fixa. Encarregada de rotar el tentacle i atacar si es pot.

```
protected override void FixedUpdate(){
    if (!appearing && playerInside){
        if (canRotate)RotateToPlayer();
        if (!CR_running) StartCoroutine(CheckCanAttack());
        if (canAttack) Attack();
    }
}
```

- **OnTriggerEnter**, OnTriggerExit: Funcions que executen el funcionament de les de la classe pare. A més, OnTriggerExit para la corrutina de comprovar si es pot atacar.

```
protected override void OnTriggerExit(Collider other){
    base.OnTriggerExit(other);
    if (other.CompareTag("Player")) StopCoroutine(CheckCanAttack());
}

protected override void OnTriggerEnter(Collider other){
    base.OnTriggerEnter(other);
}
```

- **Die**: Funció de mort del tentacle. Activa l'animació d'aparició i mou l'objecte negativament en l'eix vertical. Un cop arribada a una certa altura, crida a la funció DestroyObject per acabar de destruir l'objecte.

```
public void Die(){
    SetAppearAnimation(true);
    transform.DOMoveY(-10, 3f).onComplete += DestroyObject;
}
```

- **DestroyObject**: Funció que destrueix l'objecte i llença l'esdeveniment OnDestroyEvnt

```
protected void DestroyObject(){
    if (this.OnDestroyEvnt != null) OnDestroyEvnt(this.gameObject);
    if (!Addressables.ReleaseInstance(this.gameObject)) Destroy(this.gameObject);
}
```

- **CheckCanAttack**: Corrutina encarregada d'esperar el temps de la variable waitTimeBtwAttacks entre atacs

```
protected IEnumerator CheckCanAttack(){
    CR_running = true;
    yield return new WaitForSeconds(waitTimeBtwAttacks);
    canAttack = true; CR_running = false;
}
```

- **Attack**: Funció que activa l'animació d'atac i canvia els estats del tentacle

```
protected void Attack(){
    isAttacking = true; canAttack = false;
    tentacleAnimation.SetTrigger("attackTrigger");
}
```

+ **FinishAttack**, **ShakeCamera**, **StartAttackAnimation**, PlayGrounSlamParticles: Funcions cridades des de les animacions del tentacle. S'utilitzen per a que, en un moment donat de l'animació, els estats del tentacle canviïn, o per activar efectes.

```
public void FinishAttack(){
    isAttacking = false; isParried = false; canRotate = true; canBeInterrupted = true;
}

public void ShakeCamera(){
    shakeCameraEvent.RaiseEvent(2f, 0.5f);
}

public void StartAttackAnimation(){
    canRotate = false; canBeInterrupted = false;
    }
public void PlayGroundSlamParticles(){
    groundSlamParticles.Play();
}
```

- **SetAppearAnimation**, **StopAppearAnimation**: Funcions per a controlar l'aparició i desaparició del tentacles en escena. Gestionen els diferents estats, animacions i colliders.

```
protected void SetAppearAnimation(bool state){
    appearing = true; isAttacking = false;
    isParried = false; canRotate = false;
    tentacleAnimation.SetBool("isAppearing", state);
}

protected void StopAppearAnimation(){
    this.GetComponent<CapsuleCollider>().enabled = true;
    SetAppearAnimation(false); appearing = false; canRotate = true;
}
```

A més tenim la classe TentacleBoss. Aquesta classe hereta de Tentacle, com es pot veure en la Figura 6.1.58. El comportament és exactament igual al de la classe pare Tentacle, però variant certs punts tant al aparèixer com al desaparèixer d'escena.

```
public class TentacleBoss : Tentacle{
    Mensaje de Unity | 0 referencias
    private void OnEnable(){
        playerTransform = null;playerInside = false;
        damageable = GetComponentInChildren<Damageable>();
        tentacleAnimation = GetComponent<Animator>();
        AnimatorStateInfo state = tentacleAnimation.GetCurrentAnimatorStateInfo(0);
        SetAppearAnimation(true);
        transform.DOMoveY(0, 1.5f).onComplete += StopAppearAnimation;
    }
    Mensaje de Unity | 3 referencias
    protected override void FixedUpdate(){
        base.FixedUpdate();
        if (playerTransform != null
            && Vector3.Distance(transform.position, playerTransform.position) > 10)
            Die();
    }
}
```

Figura 6.10.4.1: Classe TentacleBoss

Com es pot veue en la Figura 6.10.4.1, els objectes de la classe TentacleBoss varian el comportament al entrar en escena (al instanciar-se surten del terra) i al sortir d'escena (es moren automàticament si el jugador està lluny).

6.10.5 Boss

La classe Boss hereta de la classe Enemy, com es pot veure en la Figura 6.1.58, i és la encarregada de gestionar l'enemic final del joc. A diferencia dels tentacles, aquest enemic no és capaç d'atacar al jugador, sino que la principal mecànica és invocar enemics tentacles per a que lluitin per ell. D'aquesta manera, el jugador haurà d'anar eliminant els tentacles que el *boss* invoqui, alhora que intenta atacar-lo. A més, el boss tindrà una segona fase quan la seva vida caigui per sota d'un llindar, en aquesta fase invocarà 4 tentacles instantàniament al seu voltant i augmentarà el màxim de tentacles que podrà invocar en total.

Model de la classe Boss:

Boss
+health: int +tentaclePrefab: GameObject +whatIsGround: LayerMask +spawnTentacleRatio: float +maxTentacles: int +actualState: int +healthForSecondState: int +secondFaceFixSpawners: List<GameObject> -tentacles: List<TentacleBoss> -dynamicObjectsNode: GameObject
-OnEnable(): void -Start(): void +Die(): void -DestroyObject(): void -OnGetHit(): void -NextState(): void -SpawnFixedTentacles(): void -PlayerEnter(): void -PlayerExits(): void -SpawnTentacle(): IEnumerator -SpawnTentacleRandomPoint(): void -SpawnTentacleInSpawner(Vector3 spawner): void -TentacleDestroyed(GameObject obj): void -RandomPointInPlayerZone(): void

Atributs de Boss:

+ **health**:Int per determinar la vida actual del boss
+ **tentaclePrefab**: GameObject que determina el prefab del tentacle per a instanciar
+ **whatIsGround**: LayerMask per determinar quina capa de l'escena és la de terra
+ **spawnTentacleRatio**: Float per determinar el rati d'invocació de tentacles
+ **maxTentacles**: Int per determinar els màxims tentacles que el boss podrà invocar

alhora.

+ **actualState**: Int per determinar l'estat actual del boss
+ **healthForSecondState**: Int per determinar la vida mínima del boss per canviar al segon estat
+ **secondFaceFixSpawners**: Llistat de GameObjects que actuen com a punts per determinar les posicions on el boss invocarà els tentacles de la segona fase.
- **tentacles**: Llistat d'objectes de tipus TentacleBoss per portar constància del tentacles invocats que hi ha a l'escena
- **dynamicObjectsNode**: Referència al GameObject on s'instanciaran els tentacles

Mètodes de Boss:

- **OnEnable:** Funció cridada pel Unity quan l'objecte s'activa. Encarregada de inicialitzar les variable i obtenir referències a components.

```
private void OnEnable(){
    playerTransform = null; playerInside = false;
    damageable = GetComponentInChildren<Damageable>();
    dynamicObjectsNode = transform.parent.gameObject;
    if (dynamicObjectsNode != null)
        islandMeshCollider = dynamicObjectsNode.
             GetComponent<DynamicObjectsNode>().islandMeshCollider;
}
```

- **Start**: Funció cridada per el Unity abans del primer frame. Utilitzada per gestionar la informació del component Damageable.

```
protected override void Start(){
    base.Start();
    if (damageable != null){
        damageable.OnGetHit += OnGetHit; damageable.OnDie += Die;
        damageable.health = health; damageable.currentLife = health;
    }
}
```

- **Die**: Funció per gestionar la mort del boss. S'encarrega d'eliminar tots els tentacles presents i fer una animació de mort.

```
private void Die(){
    foreach (TentacleBoss tentacle in tentacles) tentacle.Die();
    transform.DOShakeScale(3f, 0.7f, 3, 0);
    transform.DOScale(0, 3f).onComplete += DestroyObject;
    shakeCameraEvent.RaiseEvent(2f, 6f);
}
```

- **DestroyObject**: Funció per a destruir l'objecte de l'escena.

```
private void DestroyObject(){
    if (!Addressables.ReleaseInstance(this.gameObject)) Destroy(this.gameObject);
}
```

- **OnGetHit**: Funció encarregada de gestionar els cops del jugador i comprova si s'ha de canviar a la segona fase depenent de la vida actual.

```
private void OnGetHit(){
    if (actualState == 0 && damageable.currentLife <= healthForSecondState) NextState();
}
```

- **NextState**: Funció encarregada de passar a la segona fase. Activa una animació, fa tremolar la càmera, augmenta els màxims tentacles i invoca a tentacles instantàniament al voltant del boss.

```
private void NextState(){
    transform.DOShakeScale(3f, 0.3f, 3, 0);
    shakeCameraEvent.RaiseEvent(2f, 6f);
    actualState++; maxTentacles = 10; SpawnFixedTentacles();
}
```

- **SpawnFixedTentacles**: Funció encarregada de instanciar els tentacles de la segona fase en el llistat de punts de secondFaceFixSpawners

```
private void SpawnFixedTentacles(){
    foreach (GameObject gameObject in secondFaceFixSpawners)
        SpawnTentacleInSpawner(gameObject.transform.position);
}
```

- **PlayerEnter** i **PlayerExists**: Funcions encarregades de comprobar quan el jugador entra o surt del rang de visió del boss. Activen o desactiven l'invocació de tentacles.

```
protected override void PlayerEnter(){
    base.PlayerEnter(); StartCoroutine(SpawnTentacle());
}

protected override void PlayerExits(){
    base.PlayerExits(); StopCoroutine(SpawnTentacle());
}
```

- **SpawnTentacle**: Corrutina per anar invocant tentacles al voltant del jugador.

```
IEnumerator SpawnTentacle(){
    while (playerInside){
        if (tentacles.Count < maxTentacles){
            SpawnTentacleRandomPoint();
            yield return new WaitForSeconds(spawnTentacleRatio);
        }
        yield return null;
    }
    yield return null;
    }
```

- **SpawnTentacleRandomPoint**: Funció encarregada de instanciar els tentacles en un punt aleatori al voltant del jugador. A més, guarda la referència del tentacle instanciat i es subscriu a l'esdeveniment de destrucció.

```
private void SpawnTentacleRandomPoint(){
    RaycastHit hit;
    if (Physics.Raycast(RandomPoinInPlayerZone(), Vector3.down,
        out hit, 20, whatIsGround)){
        Vector3 pos = new Vector3(hit.point.x, hit.point.y - 10, hit.point.z);
        bool canPlace = true;
        Collider[] hitColliders = Physics.OverlapSphere(hit.point, 2);
        foreach (var hits in hitColliders)
            if (hits.gameObject.CompareTag("UnPlaceable")) canPlace = false;
        if (canPlace){
            TentacleBoss newTentacle = Instantiate(tentaclePrefab, pos,
                Quaternion.Euler(tentaclePrefab.transform.rotation.x,
                Random.Range(0f, 360f),
                tentaclePrefab.transform.rotation.z),
                dynamicObjectsNode.transform)
                .GetComponent<TentacleBoss>();
            tentacles.Add(newTentacle);
            newTentacle.OnDestroyEvnt += TentacleDestroyed;
        }
    }
}
```

- **SpawnTentacleInSpawner**: Funció encarregada de instanciar un tentacle en un punt en específic. A més, guarda la referència del tentacle instanciat i es subscriu a l'esdeveniment de destrucció.

```
private void SpawnTentacleInSpawner(Vector3 spawner){
    Vector3 pos = new Vector3(spawner.x, spawner.y-10, spawner.z);
    TentacleBoss newTentacle = Instantiate(tentaclePrefab, pos,
        this.transform.rotation, dynamicObjectsNode.transform)
            .GetComponent<TentacleBoss>();
    tentacles.Add(newTentacle);
    newTentacle.OnDestroyEvnt += TentacleDestroyed;
}
```

- **TentacleDestroyed**: Funció cridada quan un tentacle és destruït. Encarregada d'eliminar la referència del tentacle en el llistat de tentacles.

```
private void TentacleDestroyed(GameObject obj){
    tentacles.Remove(obj.GetComponent<TentacleBoss>());
}
```

- **RandomPointInPlayerZone**: Funció que retorna un Vector3 d'un punt aleatori al voltant del jugador

```
private Vector3 RandomPoinInPlayerZone(){
    return new Vector3(
        Random.Range(playerTransform.position.x + 5, playerTransform.position.x - 5), 20,
        Random.Range(playerTransform.position.z + 5, playerTransform.position.z - 5));
}
```

6.11. Sistema de generació de nivells aleatoris

L'escena de món RunGenerator és una escena on la seva estructura es genera aleatoriament. Des d'un primer moment del desenvolupament del projecte volíem una part aleatòria rejugable, molt semblant als jocs de l'estil *roguelike* o *roguelite*. L'implementació d'aquest sistema va passar per diverses fases, ja que es tracta d'una part més complexa en l'apartat tècnic. A més, la manca d'informació sobre generació aleatoria d'entorns 3D va dificultar l'implementació. Després de diverses probes e iteracions, vam arribar a un resultat que s'adapta bastant al que buscàvem en un principi: una generació aleatòria del món amb un cert punt de control.

6.11.1 Primeres versions i dificultats

L'objectiu principal era crear un entorn d'illes col·locades aleatòriament i connectades per ponts. A més, poblar aquestes illes amb objectes i recursos de món, com arbres, minerals, enemics, *attrezzo...* Per a poder aconseguir aquest objectiu, vam pensar en utilitzar una metodologia recursiva. El sistema consistía en generar recursivament un camí per a cada illa creada. Aquesta aproximació no ens donava el resultat que necessitàvem, ja que per la pròpia essència de la recursivitat es creava una estructura fractal.

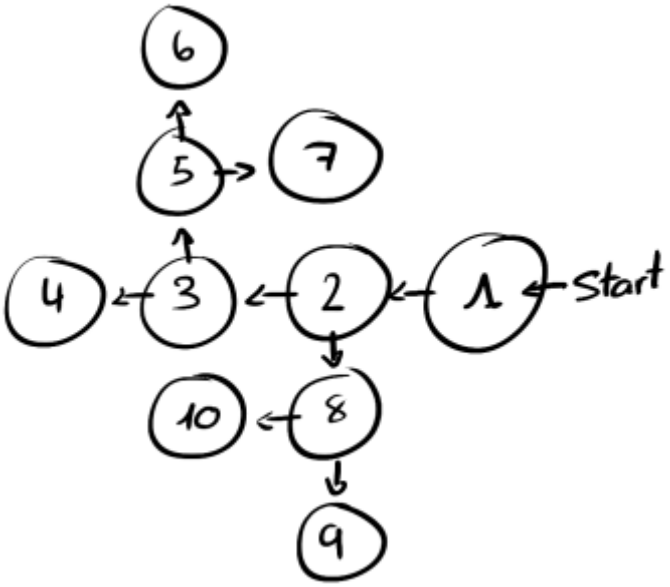


Figura 6.11.1.1. Exemple de creació recursiva del generador aleatori

En la Figura 6.11.1.1 podem veure un exemple d'aquesta estructura fractal que es generava. El problema d'aquesta estructura era que el posicionament de les illes quedava massa quadriculat i la nostra idea era tenir un camí principal i que d'aquest camí sortissin

altres camins com a ramificacions. A més, un gran problema que ens vam trobar amb aquesta aproximació completament recursiva era en la part de col·locar cada illa mirant de que no xoques amb cap altra illa. Aquest problema va ser molt difícil de debugar, ja que apareixia per culpa del propi motor i de com gestionava l'instanciament d'objectes a l'escena. El que passava era que, com el instanciament d'illes es duia a terme en la funció recursiva, fins que aquesta funció no acabava, Unity no instanciava els objectes en l'escena. Així doncs, quan una illa s'instanciava, mai col·lisionava amb cap altra illa, ja que encara que en aquell punt s'hagués especificat que hi anava una illa, encara no existía en l'escena.

Per tant, després de comprovar aquests dos problemes, vam veure que una generació completament recursiva no s'ajustava a les nostres necessitats. El plantejament que vam acabar adoptant i que s'explica en els següents Apartats va acabar sent de caràcter semi recursiu. Aquest plantejament utilitza la recursivitat per a crear camins però cada creació d'un camí no es crida recursivament, sinó que es fa a través d'una corrutina que, a més, espera un temps entre creació i creació per assegurar-se de que el Unity instancia cada camí en l'escena abans de generar el nou.

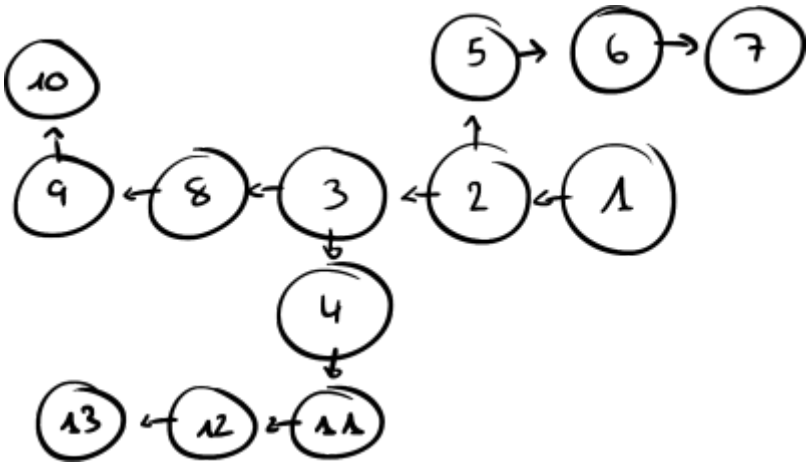


Figura 6.11.1.2: Exemple de creació semi recursiva del generador aleatori

Com es pot veure en la Figura 6.11.1.2, un cop creat el camí principal representats pels nodes 1,2,3,4, es crea un nou camí per cada un d'ells. D'aquesta manera, aconseguim una estructura menys "laberíntica" que amb un enfoc completament recursiu. A més, la personalització augmenta molt fent que no en totes les illes del camí principal es pugui crear un nou camí o que un camí tingui una distància aleatòria. Amb aquesta estructura donem al jugador una direcció a seguir, però amb la possibilitat de explorar altres camins. També ens dóna la possibilitat de portar un registre de l'estructura del camí creat per a poder col·locar

illes de caràcter més estàtic en certs punts. Per exemple, la col·locació de l'illa del boss que normalment la voldrem col·locar en la posició més allunyada possible de l'illa inicial.

6.11.2 Sistema de generació aleatori

El sistema de generació aleatori està gestionat per la classe Run_generator, ja que determinem una nova generació com a *run.* Cada generació estara composada per objectes que denominem peces.

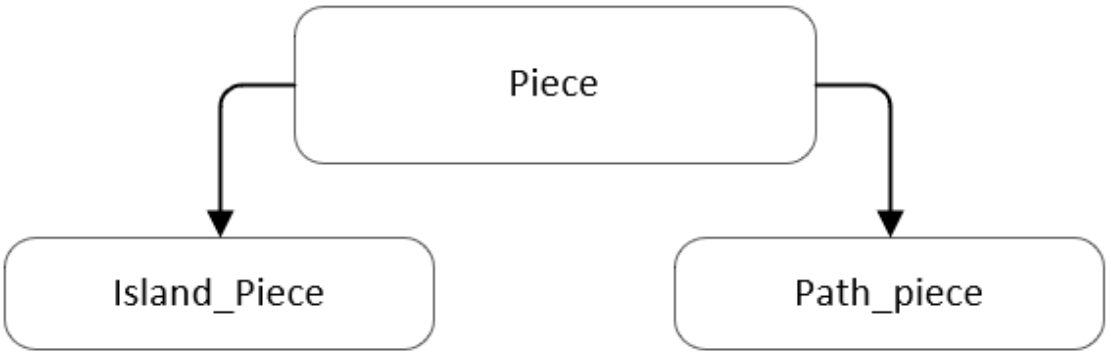


Figura 6.11.2.1: Jerarquia de les classes peça

Com es pot veure en la Figura 6.11.2.1, cada peça pot ser de tipus illa o de tipus camí (ponts). Una peça illa es connectarà amb la següent peça illa mitjançant una peça camí. Cada peça camí tindrà una entrada i una sortida i cada peça illa tindrà una entrada però podria no tenir cap sortida.

A més, hem definit un Enum per a determinar les possibles direccions en les que un camí es podrà anar formant

```
public enum Direction
{
    None, Up, Down, Left, Right
};
```

Figura 6.11.2.2: Enum de direccions

Com es pot veure en la Figura 6.11.2.2, tenim 5 direccions:

- **None**: Direcció nula. Aquesta direcció serveix per determinar quan ja no es pot avançar més
- **Up**: Direcció cap a munt

- **Down**: Direcció cap avall
- **Left**: Direcció cap a l'esquerra
- **Right**: Direcció cap a la dreta

Gràcies a les direccions aconseguim poder col·locar totes les peces de la manera correcte, depenent de la direcció de sortida de l'anterior peça.

6.11.1.1 Piece

Com es pot veure en la Figura 6.1.62, la classe Piece és la classe pare de la jerarquía de peces. En aquesta classe hi trobem els atributs compartits per totes les peces.

```
public class Piece : MonoBehaviour
{
    public GameObject entrance = null;
    public GameObject exit = null;
    public Direction entranceDirection;
    public bool hasExits = true;
}
```

Figura 6.11.1.1.1: Classe Piece

En la Figura 6.11.1.1.1 trobem els 4 atributs de la classe piece:

+ **entrance**: GameObject que determina la posició del punt d'entrada de la peça
+ **exit**: GameObject que determina la posició del punt de sortida de la peça
+ **entranceDirection**: Atribut de tipus Direction per determinar quina és la direcció per on s'entra en la peça depenent de l'anterior peça col·locada
+ **hasExits**: Bool que determina si la peça té o no sortidas

6.11.1.2 Estructura d'una peça

Cada peça està formada per diferents parts que determinen els diferents punts possibles d'entrades i sortides.

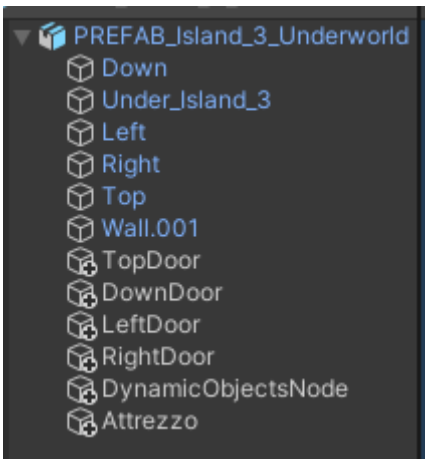


Figura 6.11.1.2.1: Exemple d'estructura d'un objecte de tipus peça illa

Com es pot veure en la Figura 6.11.1.2.1, l'estructura d'una peça illa compren:

- Under_island_3: És l'objecte malla que representa visualment l'illa. Com a exemple es pot veure aquest objecte seleccionat en la Figura 6.11.1.2.2.

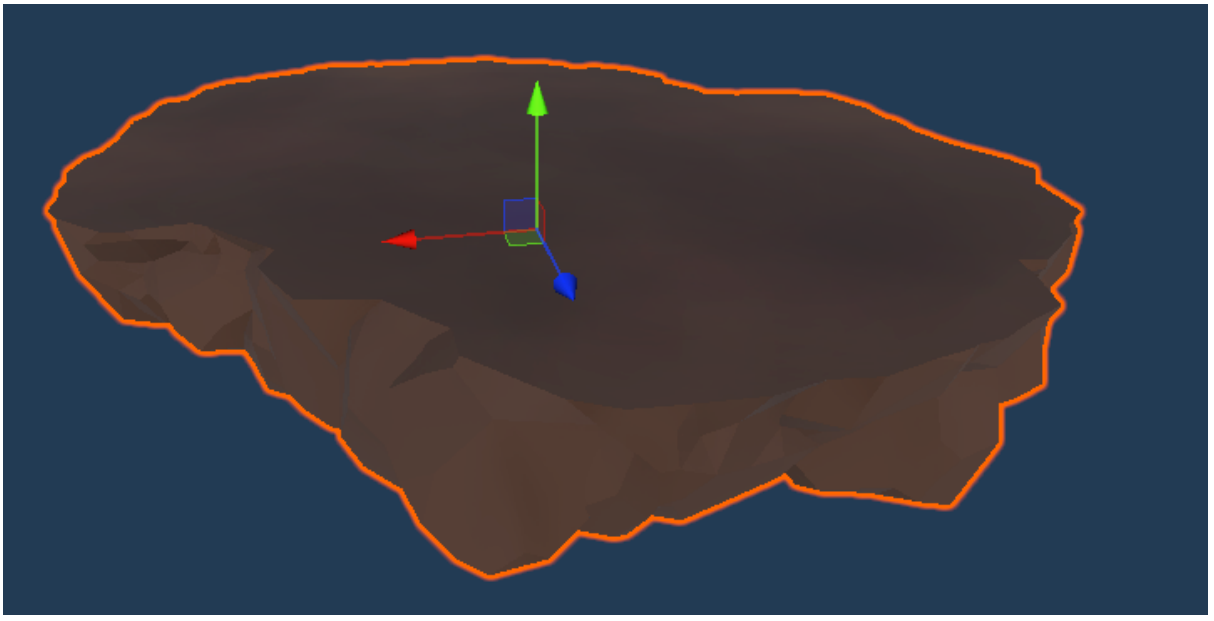


Figura 6.11.1.2.2: Illa amb el seu objecte malla seleccionat

- Wall.001: És el mur invisible que recobreix tot el borde de l'objecte, a excepció dels talls que determinen les possibles entrades i sortides de l'illa. Té un component collider i s'encarrega de que el jugador no pugui sortir dels límits. Com a exemple es pot veure aquest objecte seleccionat en la Figura 6.11.1.2.3

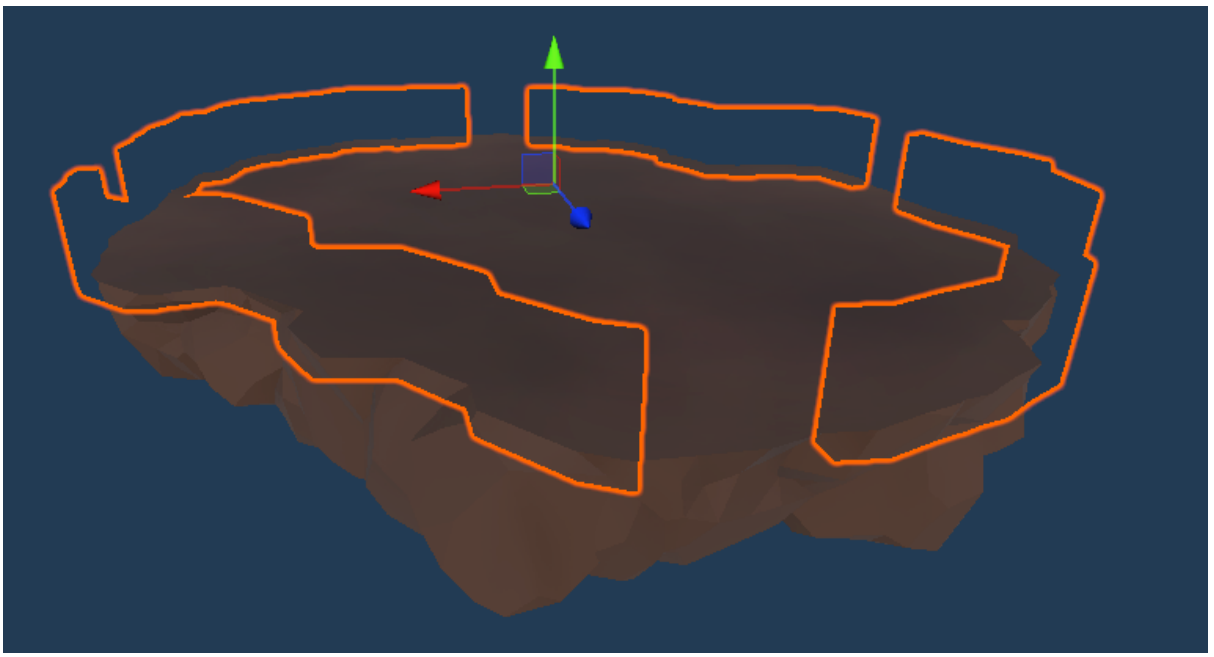


Figura 6.11.1.2.3: Illa amb el seu objecte mur seleccionat

- Down, Left, Right i Top: Són els objectes dels talls de mur que representen les posicions d'entrades i sortides de l'illa. Aquests objectes tenen un component Collider i actuen com a portes. Depenent de la col·locació de l'illa, aquests objectes activan o desactivan els colliders per a que el jugador pugui avançar. Com a exemple es pot veure aquests objecte seleccionat en la Figura 6.11.1.2.4

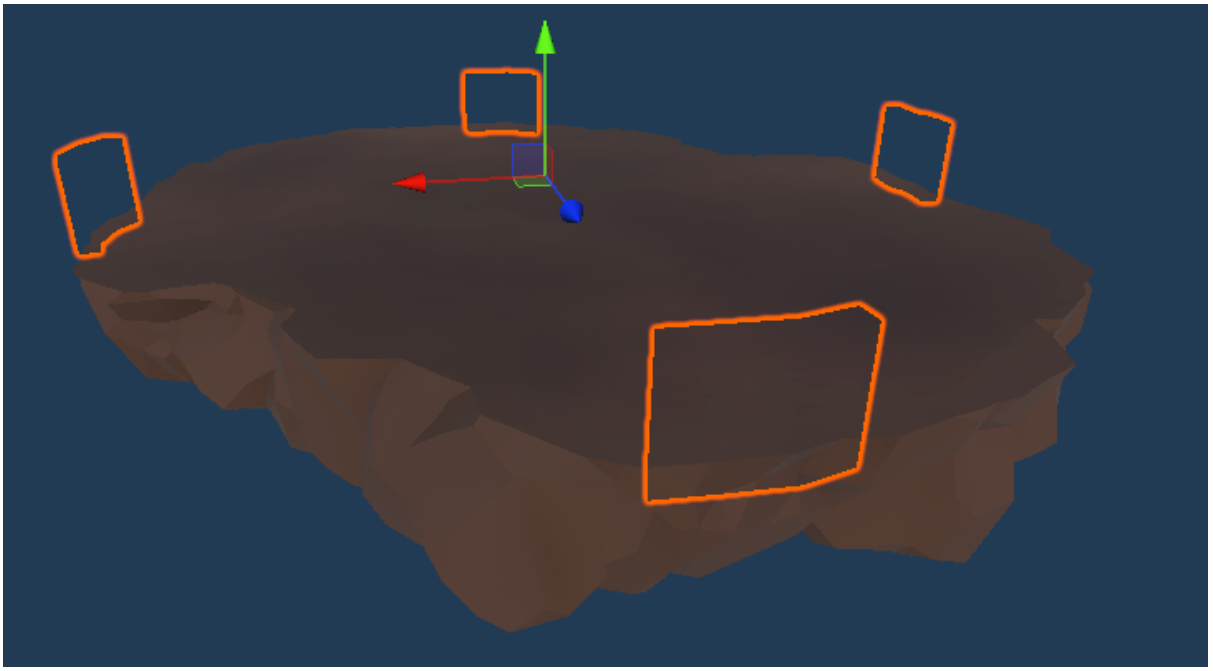


Figura 6.11.1.2.4: Illa amb els objectes Top, Down, Left, Right

- TopDoor, LeftDoor, RightDoor i DownDoor: Són els objectes portes. Aquests objectes estan posicionats cada un en la mateixa posició que Top, Left, Right, Down però tenen l'origen en altura 0. Tenen el component Door i s'utilitzen per a col·locar les illes. Aquests són els punts que s'utilitzen per "enganxar" una peça amb la següent, col·locant l'entrada de la nova peça en la posició del punt de sortida de la peça anterior. No es pot mostrar una Figura d'exemple, ja que aquests objectes no tenen representació visual, ja que son simplement punts amb el component Door.

- DynamicObjectsNode, Attrezzo: Objectes que s'utilitzen com a pare de tots els objectes dinàmics i de decoració que s'instanciaran en aquesta illa.

Cal destacar que els objectes peça de tipus camí o ponts tenen una estructura similar a l'anterior però més senzilla, amb només 2 objectes Left i Right. Aquests objectes simplement són els punts d'unió i un sempre actuarà com a entrada i l'altre com a sortida.

6.11.1.2 Door

La classe Door és l'encarregada de gestionar el component collider de les sortides i entrades d'una illa

```
public class Door : MonoBehaviour
{
    public Collider doorCollider;
    public bool isExit = false;
    public bool isEntrance = false;

    2 referencias
    public void Open()
    {
        if (doorCollider!=null)doorCollider.enabled = false;
    }
}
```

Figura 6.11.1.2.1: Classe Door

Com es pot veure en la Figura 6.11.1.2.1, la classe té:

+ **doorCollider**: Referència al component collider de l'objecte entrada o sortida amb el que està relacionat
+ **isExit**, **isEntrance**: Booleans que determinen si la porta és de sortida o d'entrada

Per últim el mètode Open desactiva el collider per a que el jugador pugui passar.

6.11.1.3 Path_piece

La classe Path_piece hereta de la classe Piece, com es pot veure en la Figura 6.1.62. La funció és gestionar la col·locació de tots els camins que uneixen les illes.

```
public class Path_piece : Piece
{
    [Header("Union points")]
    public GameObject left, right;
    2 referencias
    public void setEntranceDirection(Direction dir) {
        entranceDirection = dir;
        switch (dir) {
            case Direction.Up:
                entrance = right; exit = left;
                gameObject.transform.rotation = Quaternion.Euler(0, 180, 0);
                break;
            case Direction.Down:
                entrance = left; exit = right;
                gameObject.transform.rotation = Quaternion.Euler(0, 180, 0);
                break;
            case Direction.Left:
                entrance = right; exit = left;
                gameObject.transform.rotation = Quaternion.Euler(0, 90, 0);
                break;
            case Direction.Right:
                entrance = left; exit = right;
                gameObject.transform.rotation = Quaternion.Euler(0, 90, 0);
                break;
            default: break;
        }
    }
}
```

Figura 6.11.1.3.1: Classe Path_Piece

Com es pot veure en la Figura 6.11.1.3.1, la clase Path_Piece té 2 (left, right) atributs que són els punts d'unió que les illes faran servir per a col·locar-se. A més, té un mètode (setEntranceDirection) que s'encarrega de rotar l'objecte i assignar l'entrada i la sortida. Ho fa depenent de la direcció entrada per paràmetre, la qual és la direcció de sortida de la última illa.

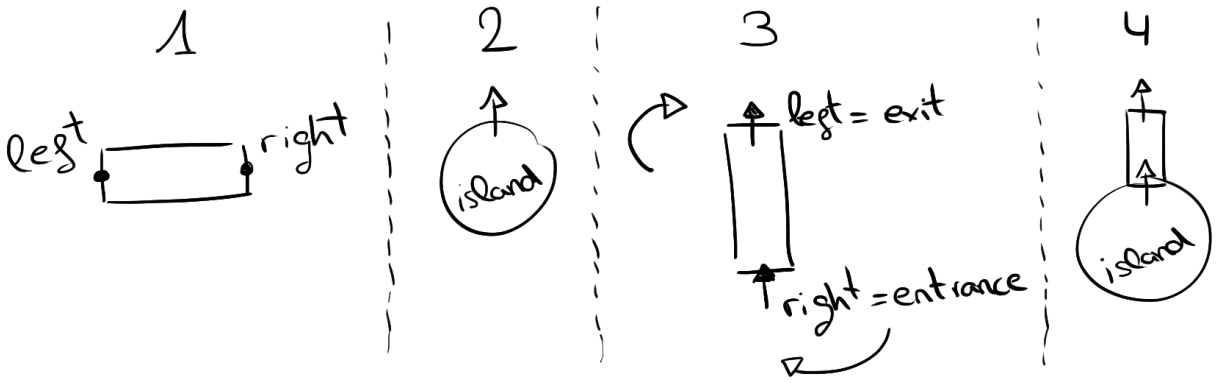


Figura 6.11.1.3.2: Exemple de rotació d'un camí en direcció cap a dalt

Com es pot veure en l'exemple de la Figura 6.11.1.3.2, en la part 1 es pot veure l'estructura que sempre tindrà un pont. En la part 2 es pot veure com una illa vol generar un camí cap a dalt, per tant la direcció de sortida de l'illa sera cap a dalt. En la figura 3 el pont es rota en l'eix de les y a partir de la direcció de sortida de l'illa. A més, direm que el punt d'unió right serà el punt d'entrada al pont i el punt de sortida el left. En la figura 4 es pot veure com quedaria la unió de l'illa amb el pont, on el punt d'entrada del pont es col·loca en la mateixa posició que el punt de sortida de l'illa.

6.11.1.3 Island_piece

La classe Island_path hereta de la classe Piece, com es pot veure en la Figura 6.1.62. La seva funció és gestionar la col·locació de totes les illes a més de calcular les possibles noves direccions per a nous camins a generar.

Model de la classe Island_piece:

Island_piece
+up, down, left, right: Door +exitDirection: Direction +visitedDirection: Direction +whatIsPiece: LayerMask -upVisited, downVisited, leftVisited, rightVisited: bool
+SetEntranceDirection(Direction dir): void +SetExitDirection(): Direction +CanPlacePiecesInDirection(Direction exitDirection): bool +AllVisited(): bool +Populate(): void +Open(): void

Atributs de Island_piece:

+ **up, down, left, right**: Objectes de tipus Door on es guarda la referència a les portes de l'illa
+ **exitDirection**: Direcció de sortida de l'illa
+ **visitedDirection**: Direcció contrària a la d'entrada (una direcció visitada es refereix a que en aquella direcció ja s'ha comprovat si es pot generar un nou camí o no)
+ **whatIsPiece**: LayerMask per determinar en quina capa es troben els objectes peça per poder col·lisionar amb ells
- **upVisited, downVisited, leftVisited, rightVisited**: Booleans per determinar si la direcció ha estat visitada

Mètodes de IslandPiece:

- **SetEntranceDirection:** Funció inicialitzadora. S'encarrega d'assignar a la direcció d'entrada la direcció entrada per paràmetre. Posteriorment, si l'illa té sortides, assigna les entrades, sortides i les direccions visitades depenent de la direcció d'entrada. Si l'illa no té sortides, rota l'objecte depenent de la direcció d'entrada.

```
public void setEntranceDirection(Direction dir){
    entranceDirection = dir;
    if (hasExits){
        switch (dir){
            case Direction.Up:
                entrance = down.gameObject; exit = up.gameObject;
                visitedDirection = Direction.Down; downVisited = true; break;
            case Direction.Down:
                entrance = up.gameObject; exit = down.gameObject;
                visitedDirection = Direction.Up; upVisited = true; break;
            case Direction.Left:
                entrance = right.gameObject; exit = left.gameObject;
                visitedDirection = Direction.Right; rightVisited = true; break;
            case Direction.Right:
                entrance = left.gameObject; exit = right.gameObject;
                visitedDirection = Direction.Left; leftVisited = true; break;
            default: break;
        }
    }
    else{
        switch (dir){
            case Direction.Up: break;
            case Direction.Down:
                gameObject.transform.rotation = Quaternion.Euler(0, 180, 0); break;
            case Direction.Left:
                gameObject.transform.rotation = Quaternion.Euler(0, -90, 0); break;
            case Direction.Right:
                gameObject.transform.rotation = Quaternion.Euler(0, 90, 0); break;
            default: break;
        }
    }
    entrance.GetComponent<Door>().isEntrance = true;
}
```

- **SetExitDirection:** Funció que calcula i retorna la direcció de sortida de l'illa. Per calcular-la visitarà totes les direccions possibles de l'illa de manera aleatòria calculant si es pot generar un nou camí, en la direcció. Si no pot trobar cap direcció on es pugui generar un nou camí retornarà una direcció nula. En canvi, si ha trobat una direcció bona, assignarà les variables relacionades i la retornarà. A més, existeix una petita probabilitat de que encara que trobi una bona direcció, retorni una direcció nula, per fer que l'estructura final d'illes sigui més variada.

```
public Direction SetExitDirection(){
    bool canPLacePiece = false;
    exitDirection = visitedDirection;
    while (!canPLacePiece && !AllVisited()){
        exitDirection = (Direction)Random.Range(1, 5);
        canPLacePiece = CanPlacePiecesInDirection(exitDirection);
    }
    if (!canPLacePiece){
        exitDirection = Direction.None; return Direction.None;
    }
    else{
        if (Random.Range(0, 90) > 1){
            exit.GetComponent<Door>().isExit = true;
            exit.GetComponent<Door>().Open(); return exitDirection;
        }
        else{
            exitDirection = Direction.None;  return Direction.None;
        }
    }
}
```

- **CanPlaceInDirection**: Funció per determinar si es pot crear un camí nou en la direcció entrada per paràmetre. Si la direcció no ha estat visitada, llençarà un raig i, si no col·lisiona amb cap altre objecte peça, la direcció serà vàlida.

```
public bool CanPlacePiecesInDirection(Direction exitDirection){
    switch (exitDirection){
        case Direction.Up:
            if (upVisited) return false;
            upVisited = true; exit = up.gameObject; break;
        case Direction.Down:
            if (downVisited) return false;
            downVisited = true; exit = down.gameObject; break;
        case Direction.Left:
            if (leftVisited) return false;
            leftVisited = true; exit = left.gameObject; break;
        case Direction.Right:
            if (rightVisited) return false;
            rightVisited = true; exit = right.gameObject; break;
        default: break;
    }
    Vector3 direction = exit.transform.position - transform.position;
    RaycastHit hit;
    if (Physics.Raycast(exit.transform.position, direction.normalized * 5,
        out hit, 6000, whatIsPiece)) return false;
```

```
    else return true;
}
```

+ **AllVisited**: Funció per determinar si totes les direccions han estat visitades

```
public bool AllVisited()=>upVisited && downVisited && leftVisited && rightVisited;
```

+ **Populate**: Funció que crida al mètode de poblar del component IslandPopulator

```
public void Populate()=>
    gameObject.GetComponentInChildren<IslandPopulator>().Populate();
```

+ **Open**: Funció que obre la portada de l'entrada

```
public void Open() => entrance.GetComponent<Door>().Open();
```

6.11.1.3 Run_generator

Run_Generator és la clase encarregada de gestionar la creació aleatòria de l'escena. La funció bàsica és crear recursivament un primer camí. Un cop el camí principal s'ha generat, itera les illes i, aleatòriament, crea nous camins amb el mateix mètode recursiu (fem que no en totes les illes es creïn nous camins per a evitar una estructura cuadriculada). Seguidament, genera i col·loca parts estàtiques de l'estructura, com l'illa de teleportació o illa de l'enemic final.

Model de la classe Run_generator:

Run_generator
-islandsLoaded: List<GameObjects> -pathsLoaded: List<GameObjects> -returnalIslandsLoaded: List<GameObjects> -bossIslandsLoaded: List<GameObjects> +maxLevel: int +directionToGenerate: DIrection +actualDirection: Direction +OnSceneReady: VoidEventSO +OnSceneLoadingComplete: VoidEventSO -mainPathGenerated: List<Island_piece> -secondaryPathGenerated: List<Island_piece> -generatingRecursivity: bool

-OnEnable(): void
-OnDisable(): void
+GenerateRun(): void
-Populate(): void
-WaitForGenerateRecursivity(): IEnumerator
-GenerateRecursivity(int deep, int level, Direction direction, GameObject recursiveExit, List<island_piece> saveIslands): void
-AddNewPath(Island_piece island): Path_piece
-AddNewIsland(Direction direction, GameObject exit):Island_piece
-AddReturnalIsland(Direction direction, GameObject exit): Island_piece
-AddBossIsland(Direction direction, GameoObject exit): Islan_piece
-ReturnalIsland(): void
-BossIsland(): void
-OpenAll(): void

Atributs de Run_generator:

- **islandsLoaded**: Llistat dels diferents tipus d'illes per a generar l'escena
- **pathsLoaded**: Llistat dels diferents tipus de ponts per a generar l'escena
- **returnalIslandsLoaded**: Llistat dels diferents tipus d'illes de retorn per a generar l'escena
- **bossIslandsLoaded**: Llistat dels diferents tipus d'illes de boss per a generar l'escena
+ **maxLevel**: Int per determinar el nivell màxim de profunditat on podrà arribar la funció recursiva de generació de camins
+ **directionToGenerate**: Direcció inicial del camí a generar
+ **actualDirection**: Direcció actual del camí
+ **OnSceneReady**: Objecte del tipus VoidEventSO per a l'esdeveniment d'escena llesta
+ **OnSceneLoadingComplete**: Objecte de tipus VoidEventSO per a l'esdeveniment d'escena carregada completament
- **mainGeneratedPath**: Llistat d'objectes de tipus Island_piece que pertanyen al camí principal generat en la primera crida recursiva
- **secondaryPathGenerated**: Llistat d'objectes de tipus Island_piece que pertanyen a tots els camins generats posteriorment del camí principal
- **generatingRecursivity**: Bool per determinar si actualment s'està generant el camí recursivament

Mètodes de Run_generator:

- **OnEnable, OnDisable**: Funcions cridades pel Unity quan l'objecte s'activa i es desactiva. Utilitzades per subscriure's i desubscriure's als esdeveniments.

```
private void OnEnable() => OnSceneReady.OnEventRaised += GenerateRun;
private void OnDisable() => OnSceneReady.OnEventRaised -= GenerateRun;
```

+ **GenerateRun**: Funció inicialitzadora encarregada d'instanciar un primer pont depenent de la direcció inicial a generar. Seguidament comença la funció recursiva a partir del primer pont generat i inicia la corrutina per a esperar a la funció recursiva.

```
public void GenerateRun(){
    actualDIrection = directionToGenerate;
    var pathAddress = pathsLoaded[Random.Range(0, pathsLoaded.Count)];
    var op = Addressables.InstantiateAsync(pathAddress.name, this.transform);
    if (op.IsDone){
        GameObject prefab = op.Result;
        prefab.GetComponent<Path_piece>().setEntranceDirection(directionToGenerate);
        generatingRecursivity = true;
        GenerateRecursivity(maxLevel, 0, actualDIrection,
            prefab.GetComponent<Path_piece>().exit, mainPathGenerated);
        StartCoroutine(WaitForGenerateRecursivity());
    }
}
```

- **Populate**: Funció que crida al mètode de poblar per cada illa creada.

```
private void  Populate(){
    foreach (Island_piece island in mainPathGenerated) island.Populate();
    foreach (Island_piece island in secondaryPathGenerated) island.Populate();
}
```

- **WaitForGenerateRecursivity**: Corrutina que espera mentres el camí principal es crea recursivament. Un cop creat, per cada illa creada, intentara generar un nou camí a partir de la sortida de cada illa. A més, hi ha una possibilitat de que no es generi cap camí en una illa donada, per fer que l'estructura final sigui més variada. Un cop ha passat per totes les illes, afegirà tant l'illa de teleport com l'illa del boss, cridarà a la funció Populate i obrirà totes les entrades. Finalment, avisarà que l'escena ha estat carregada.

```
IEnumerator WaitForGenerateRecursivity(){
    while (generatingRecursivity) yield return null;
    yield return new WaitForSeconds(0.2f);
    foreach (Island_piece island in mainPathGenerated){
```

```
        if (Random.Range(0, 10) > 4){
            island.SetExitDirection();
            if (island.exitDirection != Direction.None){
                Path_piece path = AddNewPath(island);
                if (path != null)
                    GenerateRecursivity(maxLevel - 3, 0, island.exitDirection,
                        path.GetComponent<Piece>().exit, secondaryPathGenerated);
            }
        }
        yield return new WaitForSeconds(0.1f);
    }
    generatingRecursivity = false;
    ReturnIsland(); BossIsland(); Populate(); OpenAll();
    OnSceneLoadingComplete.RaiseEvent();
}
```

- **GenerateRecursivity**: Funció recursiva encarregada de crear camins. Instancia una illa depenent de la direcció i posició entrades per paràmetre. Si s'ha instanciat i no és el nivell màxim, genera un pont a partir de la posició i direcció de la sortida de l'illa instanciada, i es crida a sí mateixa amb la direcció i sortida del pont instanciat.

```
private void GenerateRecursivity(int deep, int level, Direction direction,
    GameObject recursiveExit, List<Island_piece> saveIslands){
    if (level > deep) return;
    Island_piece island = AddNewIsland(direction, recursiveExit);
    if (island != null) {
        island.GetComponent<Island_piece>().SetExitDirection();
        saveIslands.Add(island);
        if ((level + 1) > deep) return;
        if (island.exitDirection == Direction.None) return;
        Path_piece path = AddNewPath(island);
        if (path!=null) {
            GenerateRecursivity(deep, level + 1, island.exitDirection, path.exit,
                                saveIslands);
            generatingRecursivity = false;
        }
    }
}
```

- **AddNewPath**: Funció que instància i col·loca un pont a partir de l'illa entrada per paràmetre.

```
private Path_piece AddNewPath(Island_piece island){
    var pathAddress = pathsLoaded[Random.Range(0, pathsLoaded.Count)];
    var op = Addressables.InstantiateAsync(pathAddress.name, this.transform);
    if (op.IsDone){
         GameObject path = op.Result;

    path.GetComponent<Path_piece>().setEntranceDirection(island.exitDirection);
        Vector3 offset = path.GetComponent<Piece>().entrance.transform.position
                         - path.transform.position;
        path.transform.position = island.exit.transform.position - offset;
        return path.GetComponent<Path_piece>();
    }
```

```
    else return null;
}
```

- **AddNewIsland**, **AddReturnalIsland**, **AddBossIsland**: Funcions encarregades de instanciar i col·locar un objecte del tipus Islan_piece. Totes tenen la mateixa estructura, però varia el tipus d'illa que instancian. A partir de la direcció i la posició entrada per paràmetre es col·loca la illa. Com a exemple es mostra AddNewIsland.

```
private Island_piece AddNewIsland(Direction direction, GameObject exit){
    var islandAddress = islandsLoaded[Random.Range(0, islandsLoaded.Count)];
    var op = Addressables.InstantiateAsync(islandAddress.name, this.transform);
    if (op.IsDone) {
        GameObject island = op.Result;
        island.GetComponent<Island_piece>().setEntranceDirection(direction);
        Vector3 offset = island.GetComponent<Piece>().entrance
            .transform.position - island.transform.position;
        island.transform.position = exit.transform.position - offset;
        return island.GetComponent<Island_piece>();
    }
    else return null;
}
```

- **ReturnalIsland**, **BossIsland**: **Fucnions** encarregades de trobar la posició en el camí per a poder generar tant l'illa de teleportació com l'illa del boss. Pot no trobar cap posició viable i que l'escena no tingui alguna d'aquestes illes. Com a exemple es mostra BossIsland

```
private void BossIsland(){
    int actualIsland = mainPathGenerated.Count - 1;
    Island_piece bossIslandGenerated = null;
    while (actualIsland > 0 && bossIslandGenerated == null){
        while (!mainPathGenerated[actualIsland].AllVisited()
                && !bossIslandGenerated){
            if (mainPathGenerated[actualIsland].SetExitDirection()
            != Direction.None){
                Path_piece path = AddNewPath(mainPathGenerated[actualIsland]);
                bossIslandGenerated = AddBossIsland(mainPathGenerated[actualIsland]
                    .exitDirection, path.GetComponent<Piece>().exit);
            }
        }
        actualIsland--;
    }
    if (bossIslandGenerated == null){
    }
    else bossIslandGenerated.Open();
}
```

- **OpenAll**: Funció encarregada d'obrir les entrades de totes les illes generades

```
private void OpenAll(){
    foreach (Island_piece island in mainPathGenerated) island.Open();
    foreach (Island_piece island in secondaryPathGenerated) island.Open();
}
```

6.11.1.4 IslandPopulator

Classe encarregada de instanciar el contingut de les illes (recursos, enemics, decoració...).

```
private List<GameObject> spawnedObjects = new List<GameObject>();
private MeshCollider meshCollider;
public LayerMask whatIsGround;
public GameObject[] prefabs;
public GameObject dynamicObjectNode;
```

Figura 6.11.1.4.1: Atributs de la classe IslandPopulator

Com es pot veure en la Figura 6.11.1.4.1, Islandpopulator té:

- **spawnedObjects**: Llistat de GameObjects amb els objectes instanciats
- **meshCollider**: Component MeshCollider de l'illa per poder obtenir posicions aleatòries dins d'ella
- **whatIsGround**: LayerMask que determina la capa del terra
- **prefabs**: Llistat d'objectes que es podran instanciar
- **dynamicObjectNode**: GameObject on s'instanciaran tots els objectes com a fills

Figura 6.1.73: Mètodes Populate i TryToSpawn de la classe IslandPopulator

```
public void Populate(){
    int tries = 0;
    while(spawnedObjects.Count < 10 && tries < 100) {
        if (!TryToSpawn()) tries++;
    }
}
1 referencia
private bool TryToSpawn(){
    RaycastHit hit;
    bool canPlace = false;
    if (Physics.Raycast(RandomPointInBounds(meshCollider.bounds), Vector3.down, out hit, 20, whatIsGround)){
        Vector3 pos = new Vector3(hit.point.x, hit.point.y, hit.point.z);
        canPlace = true;
        Collider[] hitColliders = Physics.OverlapSphere(hit.point, 2);
        foreach (var hits in hitColliders) if (hits.gameObject.CompareTag("UnPlaceable")) canPlace = false;
        if (canPlace)
        {
            GameObject prefab = prefabs[Random.Range(0, prefabs.Length)];
            spawnedObjects.Add(Instantiate(prefab, pos, Quaternion.Euler(prefab.transform.rotation.x,
                Random.Range(0f, 360f), prefab.transform.rotation.z)));
            spawnedObjects[spawnedObjects.Count - 1].transform.parent = dynamicObjectNode.transform;
        }
    }
    return canPlace;
}
```

Figura 6.11.1.4.2: Mètodes Populate i TryToSpawn de la classe islandPopulator

A més, com es pot veure en la Figura 6.11.1.4.2, la classe IslandPopulator té:

+ **Populate**: Funció que intenta instanciar 10 objectes a l'illa mentre no s'ha intentat instanciar més de 100 cops.
- **TryToSpawn**: Funció encarregada de trobar una posició viable per a un objecte sense que col·lisioni amb altres. Utilitza raigs per col·lisionar amb el terra de l'illa i trobar els punts. Seguidament mira si en aquell punt es col·lisiona amb altres objectes i, si no es col·lisiona, instancia un nou objecte.

6.12. Proves realitzades

Les proves han estat sempre presents en el desenvolupament del projecte, tant en la part artística com en la tècnica. Cada nou objecte o mecànica que creavem l'havíem de provar i posar en context amb l'estat actual del joc. Així doncs, moltes mecàniques i objectes s'han anat canviant, modificant o fins i tot eliminant, gràcies al resultat d'anar provant-los.

Per a poder provar en més profunditat els aspectes tècnics, hem utilitzat el mòdul "Debug" tant del Unity com del editor Visual Studio. Aquest mòdul ens permet monitoritzar totes les variables amb el seu flux i atributs per saber en tot moment l'estat de la partida. A més ens permet parar l'execució en els punts que desitgem per poder observar amb claredat tot el que està passant. Alhora, hem fet ús de la consola del Unity per poder escriure diferents missatges que ens indiquen apartats importants, com quina funció s'acabava d'executar o possibles missatges d'error, entre altres. També hem utilitzat les funcions del Unity OnGizmosDraw(), que ens permet veure els rajos que llençavem en situacions on volíem comprobar colliders.
Per últim, hem fet ús d'un lector online de fitxers de format json per poder veure i provar els diferents resultats en les primers fases del sistema de guardat.

7. Resultats

7.1. Legislació i normativa vigent

Pel que fa als aspectes legislatius, el projecte no presenta cap problema, donat que no s'aplica en cap situació la LOPD (Llei Orgànica de Protecció de Dades), ja que en cap cas guardem informació de caràcter personal del jugador. Tampoc apliquem la LSSICE (Llei de Serveis de la Societat de la Informació i Comerç Electrònic), ja que el projecte no constitueix cap activitat econòmica.

Pel que fa a problemes amb el Copyright, hem utilitzat algunes eines d'ús lliure, com:

- Lllibreria DoTween (http://dotween.demigiant.com/index.php), per animar objectes d'una manera simple i ràpida.
- El paquet del Unity Asset Store, Melee Warrior animations per al personatge i animacions (https://assetstore.unity.com/packages/3d/animations/melee-warrior-animations-free-165785#content).

Per tant, no hauríem de tenir cap problema a l'hora de comercialitzar-lo. Tot i així, si amb la comercialització del joc tinguéssim uns certs ingressos definits pels propietaris de Unity, hauríem de comprar llicències professionals del programa per a evitar problemes legals

7.2. Pegi

El sistema Pan European Game Information, o PEGI, és un sistema europeu per a cal·lificar els videojocs per edats recomanades. Aquest sistema descriu el contingut sensible que pot aparèixer en el joc, com ara la presència de sexe, drogues, violència, paraules malsonants… Aquestes classificacions només són informatives, i en cap moment són de compliment obligatori. Per exemple, a una persona menor d'edat no se li pot prohibir la compra d'un joc amb una edat recomanada de més de divuit anys. En la Figura 7.2.1 podem veure les diferents etiquetes de la classificació PEGI.

Figura 7.2.1:Llistat d'etiquetes PEGI

7.3. Resultat final

En aquest apartat mostrarem diferents captures del joc per poder veure el resultat final del projecte. A més, en el següent enllaç podem veure un vídeo d'una partida completa del joc. https://youtu.be/1t3LIzRvFBY


38 FPS
20 / 30


2 FPS
20 / 30


42 FPS
Wood
4891 / 10
E
Upgrade
20 / 30


40 FPS
E Craft
20 / 30


42 FPS
Tools
Placeables
Campfire 1
Information panel:
Campfire very compfi
Wood
4881 / 4
Stone
4977 / 4
Craft
20 / 30

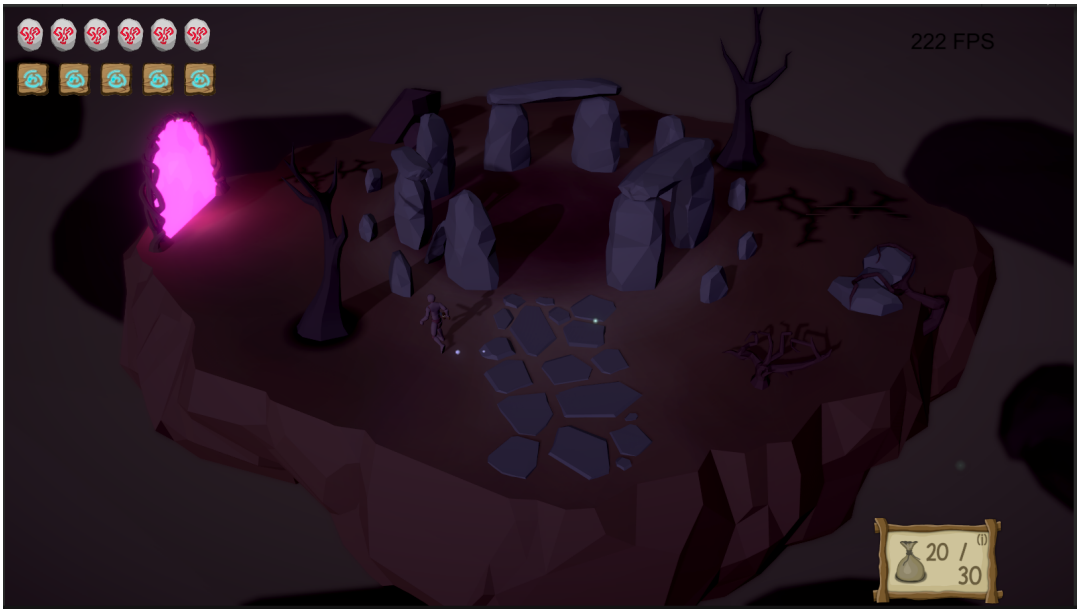

44 FPS
E Cook
R Upgrade
X Remove
20 / 30

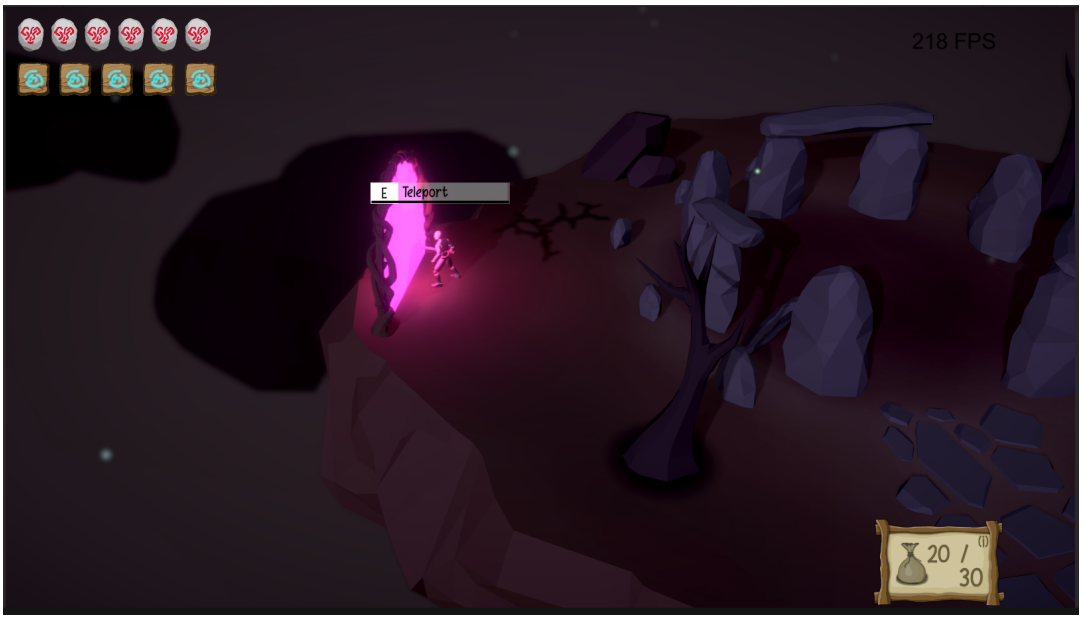

42 FPS
Campfire 3 1
Campfire 1 3
Maceta 4
20 / 30


30 FPS
20 / 30


2 FPS
20 / 30


46 FPS
E Travel to Underworld
20 / 30


222 FPS
20 / 30

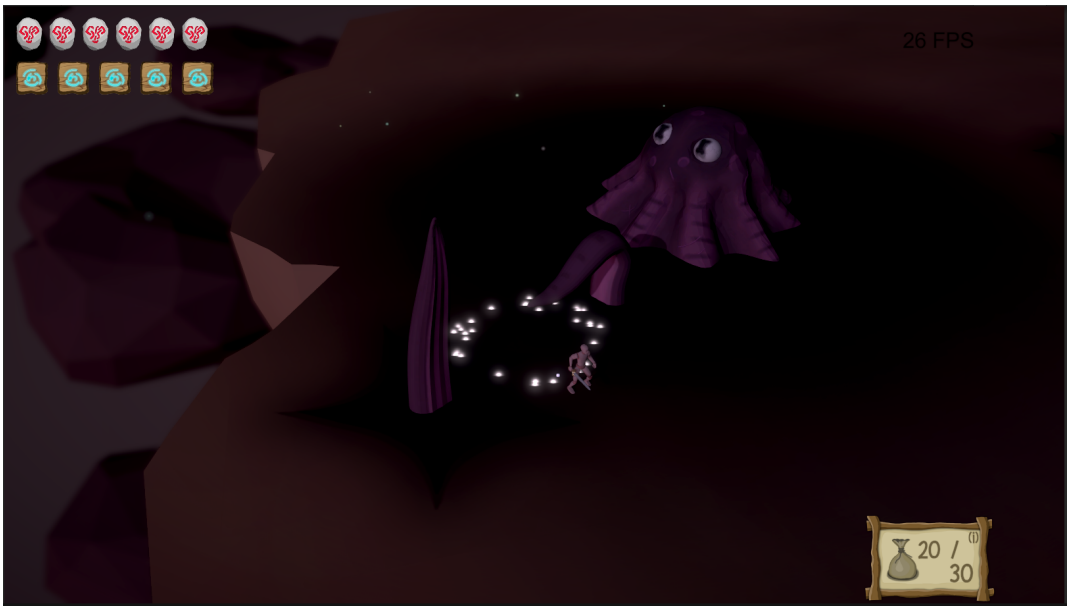

218 FPS
E Teleport
20 / 30

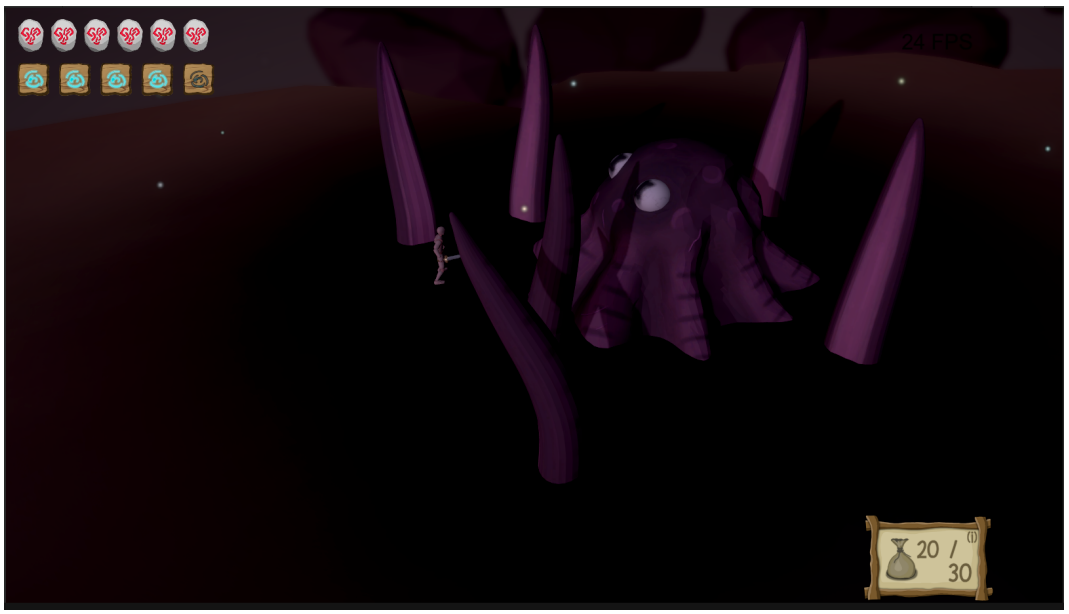

200 FPS
20 / 30


186 FPS
20 / 30

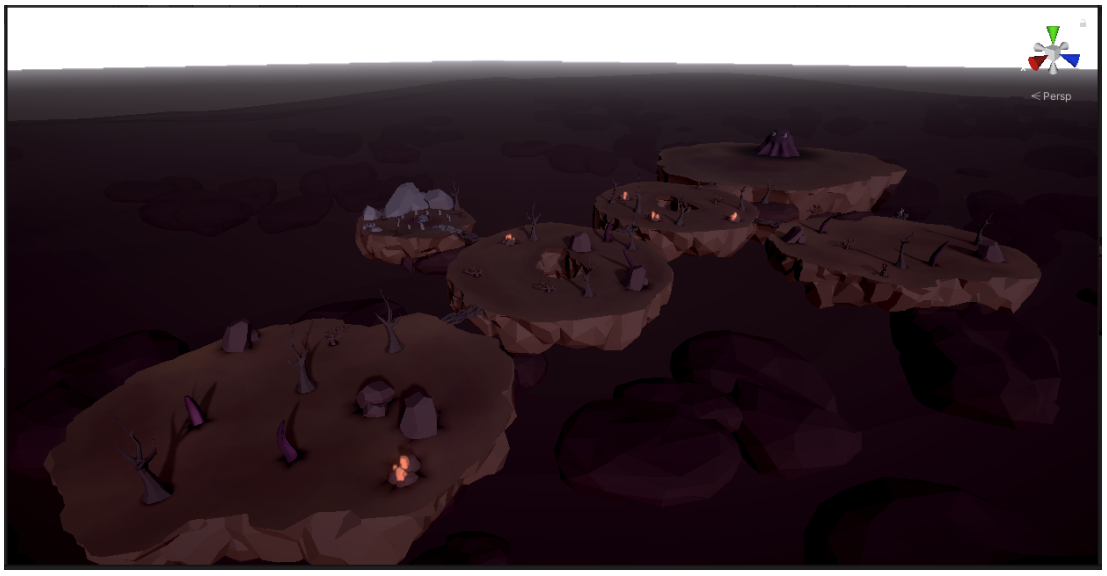

182 FPS
20 / 30

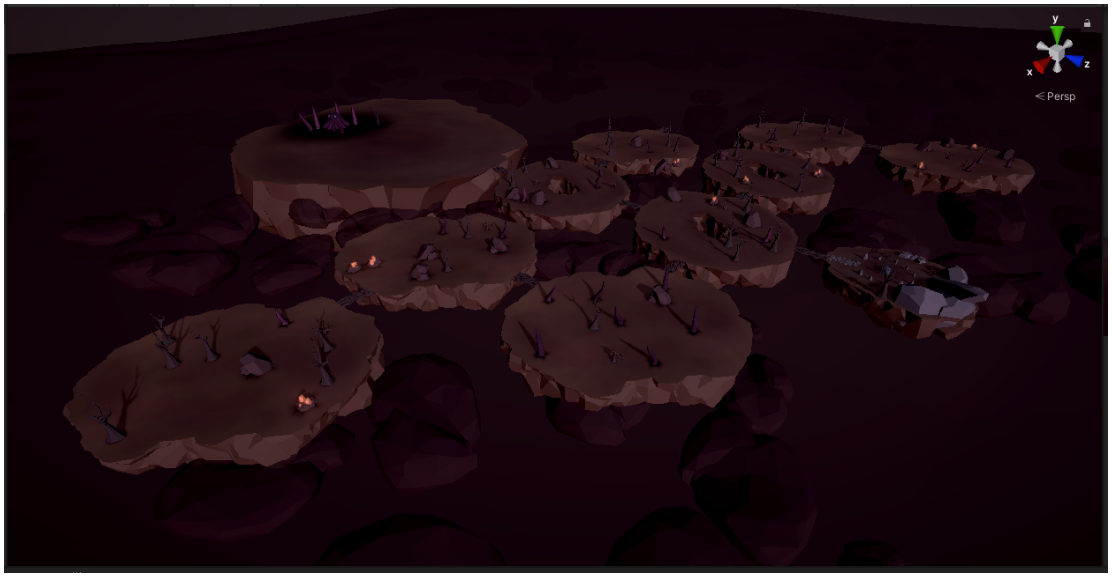

182 FPS
E Teleport
20 / 30

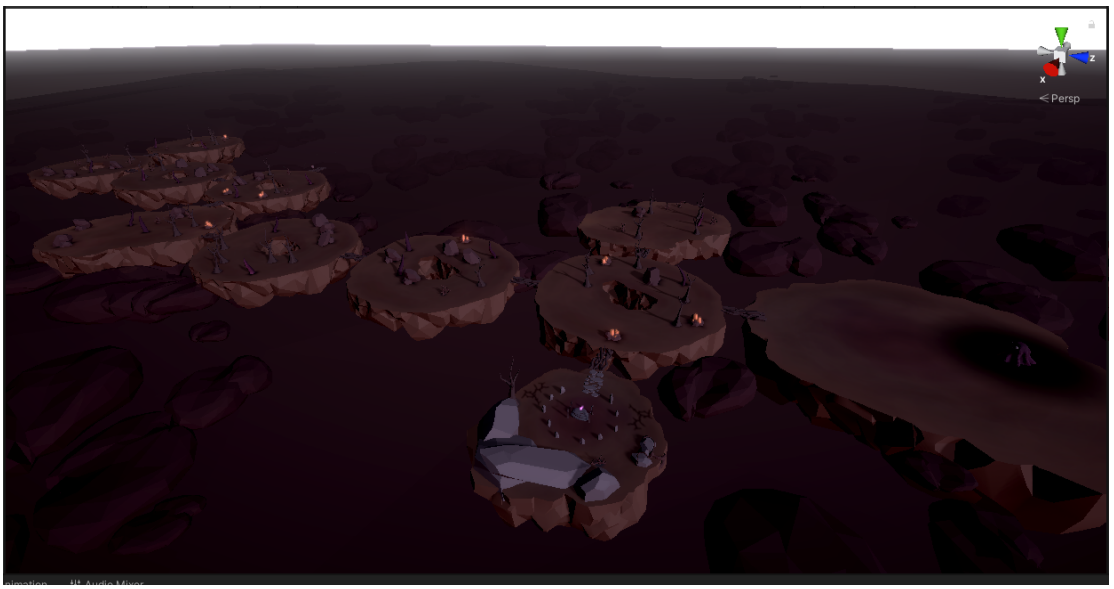

26 FPS
20 / 30


24 FPS
20 / 30


186 FPS
20 / 30


Persp


y
x
z
Persp


Persp

8. Conclusions

8.1 Valoració del treball

En quant a la valoració final del projecte, podem dir que el balanç final és positiu, tot i que finalment el resultat no ha complert totes les nostres expectatives. Aquest projecte és ambiciós i ataca un gènere exigent a nivell de contingut, i és precisament això el que ens fa pensar que encara hi ha molta feina per fer. Cal esmentar que tots dos membres apostem per aquest projecte per seguir desenvolupant-lo, ja que creiem que un projecte d'aquestes característiques encara en podem aprendre molt més.

Un aspecte molt important que ens ha motivat durant tot el desenvolupament ha estat l'augment de l'experiència en la realització de cadascuna de les tasques, tant el disseny i l'art, com la programació. Considerem que la realització d'aquest projecte ens ha fet créixer com a dissenyadors i desenvolupadors de videojocs, ja que treballar en equip en un projecte d'aquestes característiques necessita una planificació i coordinació que altres tasques durant el grau no ens ha portat, a causa de l'exigència personal que s'hi ha aplicat. Durant tot el desenvolupament el nivell de compromís no ha baixat, i creiem que a l'hora de treballar conjuntament ha estat molt important per portar un bon ambient i no perdre en cap moment la comunicació.

En quant a l'art, ha estat un repte, ja que cap dels dos té un perfil artístic molt marcat, i la decisió de realitzar tot l'apartat visual per nosaltres mateixos ha dificultat les coses i ha enlentit tot el procés de desenvolupament. Tot i això, considerem que ha estat el millor i no ho faríem de cap altre manera. Des del nostre punt de vista, crear un projecte des de l'inici amb la filosofia de generar tot contingut és el que més s'apropa a la realització d'un producte real, i és també el que dota al projecte d'un estil propi i únic d'acord amb les idees dels desenvolupadors. Tot aquest procés de generació de contingut ens ha fet veure la quantitat de disciplines que existeixen durant la creació d'un videojoc, i lo difícil que és dominar un camp concret. Bé és cert que la corba d'aprenentatge a nivell artístic no és lineal, i que abans d'aconseguir els primers dissenys s'han descartat nombroses idees, però és aquest el punt on el dissenyador ha de buscar l'estil propi el que el projecte demanda per, finalment, obtenir el resultat desitjat.

Per altra banda pel que fa a l'apartat més tècnic del projecte, ha estat necessari realitzar un estudi previ de les estructures bàsiques del motor a utilitzar, ja que són necessàries per

desenvolupar un projecte de tal envergadura. Per obtenir tot aquest coneixement ha estat necessari invertir una bona quantitat de temps durant les primeres fases del projecte, però és cert que cada hora invertida a l'inici s'ha guanyat a mesura que el projecte ha anat creixent. Una de les coses que teníem clares és que, donades les característiques, era necessari tenir un codi ben estructurat i amb una capacitat modular molt elevada per seguir integrant mecàniques i contingut.

Un aspecte a millorar ha estat la planificació del contingut i la definició de mecàniques, ja que en diverses ocasions ens hem trobat amb la necessitat de modificar o adaptar parts del codi per satisfer noves funcions. Aquest fet també ens ha comportat una pèrdua de temps en quant a la implementació del contingut general del joc i la seva finalització. Doncs, amb una planificació més organitzada i una definició més precisa de les mecàniques i com interactuen entre elles, podríem haver desenvolupat el projecte d'una manera més eficient d'acord als objectius.

En conclusió podem dir que, tot i que la realització del projecte ha estat molt gratificant, el producte que es presenta no es tracta d'un videojoc pròpiament dit, i que s'apropa més a un prototip amb aspiracions d'en un futur convertir-se en un joc capaç d'atraure l'atenció del públic al qual va dirigit. Finalment podem dir que ha estat tota una experiència i esperem que les bases que s'han format durant aquest projecte serveixin pels que vindran.

8.2 Desviacions de la planificació original

Durant tot el període de creació del projecte, hem hagut de compaginar el treball, la creació del TFG i el temps lliure. Això ha fet que pràcticament la totalitat del projecte es realitzes els caps de setmana, ja que era quan teniem més temps. Sí que és cert que durant els primers mesos en Dani no tenia treball i va poder avançar bastant la part tècnica del projecte. Però a principis d'any va començar a treballar a la mateixa empresa que en Sergio i els horaris per a poder realitzar el projecte es van igualar. Com els dos treballem en una empresa informàtica, estar treballant 8 hores davant un ordinador feia molt complicat poder treballar en el projecte per les tardes. Per això, entre setmana aprofitavem per fer esport i utilitzar els caps de setmana per centrar-nos únicament en el projecte.
En la figura 8.2.1 podem veure el diagrama de Gantt de la planificació final del projecte.

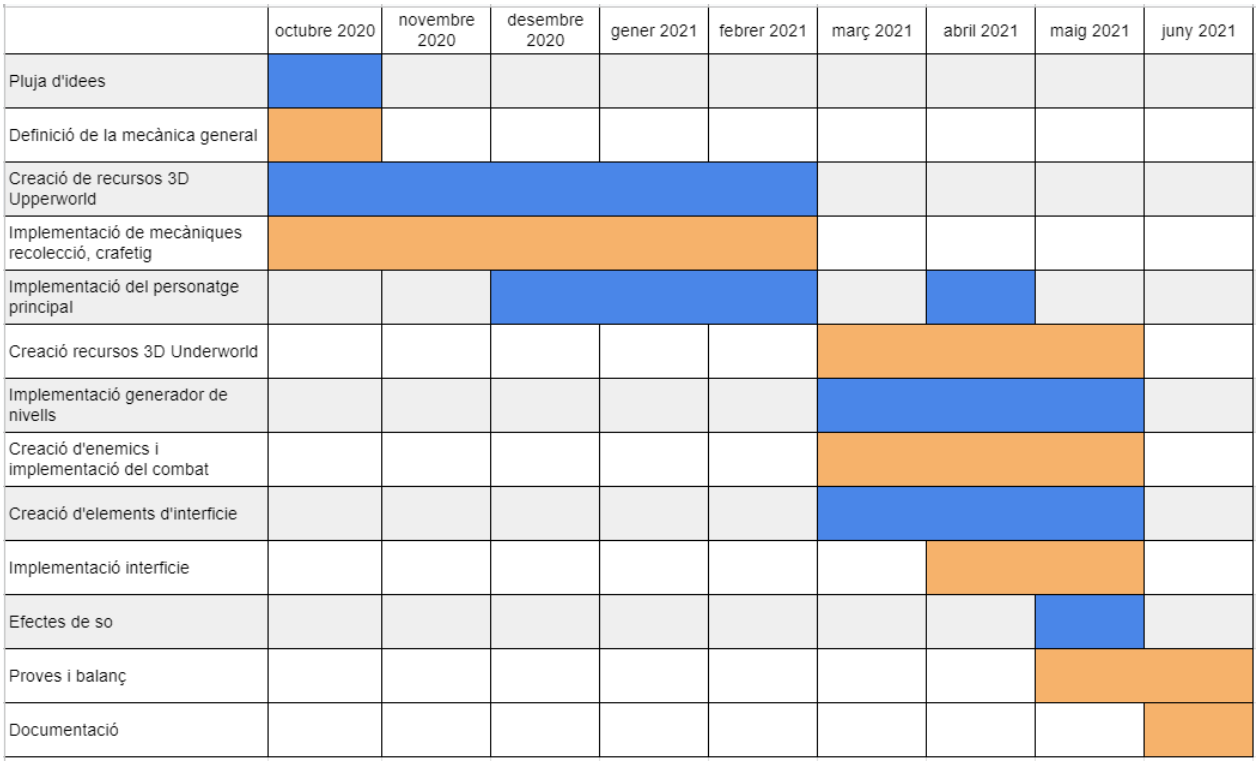


Figura 8.2.1: Diagrama de Gantt del projecte

9. Treball futur

Al llarg del desenvolupament del projecte hem realitzat canvis a nivell mecànic que han acabat afectant la planificació inicial. Tot això ha esdevingut en un problema a l'hora d'assolir tots els objectius que ens vam marcar en un inici. A causa d'aquests contratemps, trobem que el projecte no té tot el contingut que ens hauria agradat. Com a treball futur considerem que hi ha un seguit de tasques a realitzar que enriqueixen molt l'experiència, i de cares a la continuació del desenvolupament, seran els primers a ser desenvolupats.

A continuació el llistat d'imprescindibles de cares a noves versions:

- **Expansió del món:** Tal i com s'esmenta durant el projecte, l'objectiu del joc és atrapar el jugador en l'entorn que es presenta. Per fer-ho creiem que és molt important ampliar el món de tal manera que el jugador "s'hi pugui perdre", afegint diferents biomes, recompenses i proposant petits puzzles per fer l'exploració molt més entretinguda. Per altra banda, també serà necessari ampliar la quantitat d'elements decoratius que poblen cadascuna de les illes, afegint diversitat als escenaris.

  En quant al món inferior, el *Underworld*, també tenim intenció d'ampliar els reptes que suposa l'exploració, i aconseguir una experiència una mica més desafiant pel jugador. Per fer-ho més entretingut, volem afegir més enemics amb diferents comportaments i altres perills estàtics que realment forcin al jugador a exprimir les mecàniques de combat.
- **Millora en el combat:** Tot i que s'han aplicat mecàniques de combat més enllà de l'atac simple, creiem que es pot millorar notablement. Per millorar aquest aspecte es realitzarà una actualització, un cop tinguem definit el comportament de nous enemics, per tal d'ajustar el funcionament. Un altre aspecte a tenir molt en compte és el feedback, ja que creiem que és molt important en quant al combat es refereix, i ja que una gran part del joc se centra en l'acció del *Underworld*. Volem que sigui el més precís i informatiu possible.
- **Augment d'objectes:** En quant als objectes, i ens referim tant a aquells que es poden col·locar a l'illa com els que hi pots interactuar directament, sabem que en aquests moments no hi ha gaire varietat en quan a les possibilitats que aquests ofereixen. Un dels següents pasos és realitzar una pluja d'idees que ens doni l'oportunitat d'exprimir aquesta mecànica del joc, ja que creiem que és un dels pilars

fonamentals i donen un plus de llibertat al jugador a l'hora d'aconseguir l'objectiu final.

- **Nous objectius:** L'ampliació dels objectiu va directament enllaçat amb el punt de l'expansió del món, ja que en el moment que el món creix, les possibilitats també, i volem aprofitar aquest fet per proposar diferents reptes al jugador. Un dels punts principals sería la inclusió de nous enemics finals i, per tant, les recompenses, estimulant al jugador a seguir explorant i avançant en la trama.
- **Inclusió d'una història:** Encara que personalment creiem que no és l'objectiu del joc, explicar una història, per bàsica que sigui, és necessària, ja que en moltes ocasions es fa servir per justificar l'objectiu del personatge o els successos del món. Basant-nos en la història que podem veure en el Apartat 5.3.2, podem elaborar els elements bàsics per explicar la trama del personatge i d'aquesta manera enriquir l'experiència de joc.

10. Bibliografia

1. Unity technologies. Unity Manual. https://docs.unity3d.com/
2. Unity technologies. Unity Blog. https://blogs.unity3d.com/
3. Unity technologies. Unity Forum. https://forum.unity.com/
4. Unity technologies. Unity Answers. https://answers.unity.com/
5. Unify. Unity Wiki. http://wiki.unity3d.com/
6. Unity. Canal de youtube “Unity” https://www.youtube.com/user/Unity3D
7. Stack Exchange, Inc. Stack Overflow. https://stackoverflow.com/
8. CodeMonkey. Canal de youtube “Code Monkey” https://www.youtube.com/channel/UCFK6NCbuCIVzA6Yj1G_ZqCg
9. Brackeys. Canal de youtube “Brackeys”. https://www.youtube.com/brackeys
10. Flow Studio. Canal de youtube “Flow Studio” https://www.youtube.com/channel/UCeaSgd_wuEWpqINS9nLfBxQ
11. Game Dev Guide. Canal de youtube “Game Dev Guide” https://www.youtube.com/channel/UCR35rzd4LLomtQout93gi0w
12. Design reference models. http://www.stoolfeather.com/low-poly-series-landscape.html
13. brainchildpl. Canal de youtube. https://www.youtube.com/c/brainchildpl
14. Grant Abbitt. Canal de youtube. https://www.youtube.com/c/GrantAbbitt
15. Referències visuals. Pinterest. https://www.pinterest.es/
16. Low poly reference. https://cgi.tutsplus.com/es/tutorials/secrets-to-creating-low-poly-illustrations-in-blender--cg-31770

11. Anexos

Com a Annex, hem adjuntat el projecte sencer de Unity.
Per a poder explorar els arxius del projecte cal accedir a la carpeta Assets. En la carpeta Assets trobem la jerarquía de carpetes que hem fet servir per al projecte. Les més importants referents a la part técnica són les carpetes Scripts i 2_ScripatbleObjects. En aquesta última es poden trobar tots els tipus d'objectes ScriptableObjects utilitzats en el projecte, que són el "motor" central de tots els sistemes. Per la part artística les carpetes més interessants són 1_SceneAssets (amb els objectes principals que conformen els escenaris), 3_Life (on hi trobem tots els objectes relacionats amb els enemics), i 4_UI (on hi trobem les imatges i icones fetes servir per a la interfície del joc).

Per últim, hem adjuntat un vídeo on es poden veure totes les principals mecàniques del joc.
https://youtu.be/1t3LIzRvFBY

12. Manual d'usuari

12.1. Iniciar el joc

Per a poder iniciar el joc s'ha d'executar el fitxer executable "TFG_Project.exe" que resideix dins la carpeta "TFG_Project_Build". En iniciar el joc es mostrarà el logotip del Unity i seguidament es veurà el menú principal. Per poder jugar simplement s'haurà de sel·leccionar el botó New Game o Continue en cas d'existir un arxiu de guardat.

12.2. Controls

- **Moviment:** el jugador es podrà moure en totes direccions utilitzant les tecles WASD
- **Atac:** tecla F. El jugador realitzarà un atac amb l'eina equipada.
- **Contraatac:** tecla G. El jugador realitzarà un contraatac.
- **Consumir energia:** tecla . El jugador consumirà un punt d'energia.
- **Pausa:** tecla ESC. S'obre el menú de pausa.
- **Menú de plantació:** tecla TAB. S'obre el menú de plantació.
- **Menú de col·locació:** tecla Q. S'obre el menú de col·locació.
- **Equipar espasa:** tecla 1. El jugador s'equipa l'eina de tipus espasa.
- **Equipar destral:** tecla 2. El jugador s'equipa l'eina de tipus destral.
- **Equipar pic:** tecla 3. El jugador s'equipa l'eina de tipus pic.
- **Interacció primaria:** tecla E. El jugador realitza la interacció primària amb l'objecte.
- **Interacció secundaria:**tecla R. El jugador realitza la interacció secundària amb l'objecte.
- **Interacció terciaria:** tecla X. El jugador realitza la interacció terciària amb l'objecte.

12.3. Objectiu del joc

L'objectiu principal del joc és derrotar al enemic final que resideix al *Underworld*. Per poder arribar-hi, el jugador haurà de recolectar recursos i millorar les seves eines. El joc acaba quan el jugador derrota a l'enemic final.