Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Disseny i Desenvolupament de Videojocs

Títol: Eines per a la creació de videojocs RPG 2D a Unity

Document: Memòria

Alumne: Arnau Alberti Serra

Tutor: Gustavo Ariel Patow

Departament: Informàtica, matemàtica aplicada i estadística (IMAE)

Àrea: Llenguatges i sistemes informàtics

Convocatòria: Setembre 2022

Índex

1. Introducció

La indústria dels videojocs és una indústria molt dinàmica, canviant i ràpida, sobretot a les primeres etapes de prototip d'un videojoc. A l'hora de crear un videojoc, és important començar creant diversos prototips per a tenir una idea de com serà el videojoc final, això ens dona una idea de si serà divertit a l'hora de jugar o si té altres errors importants que puguin fer inviable el videojoc. És per aquest motiu que, per als productors de videojocs, la rapidesa i l'agilitat és un factor rellevant per l'èxit.

1.1 Motivacions

La motivació principal per a dur a terme aquest projecte ha estat la identificació de la necessitat d'un conjunt d'eines que faciliti la creació de projectes futurs. És a dir, un dels meus principals objectius, tant en l'àmbit personal com en l'àmbit professional, és poder dedicar-me a crear videojocs. És per aquest motiu que tenir un conjunt d'eines que m'ajudin a complir això pot ser molt beneficiós.

Aquesta necessitat la vaig detectar l'estiu del 2021, quan, en participar en diverses GameJams, en les quals la finalitat és crear un joc en molt poc temps, m'adonava que hi havia processos que podia crear de forma genèrica, ja que són comuns en molts de jocs, i que podien estalviar molt de temps.

1.2 Propòsit

El propòsit d'aquest treball, és proporcionar eines de creació de videojocs per a videojocs de rol 2d. D'aquesta manera, gràcies a les eines proporcionades, els desenvolupadors poden crear prototips de videojocs més ràpidament.

1.3 Objectius del projecte

Crear una sèrie d'eines altament personalitzables per a facilitar la feina als creadors de videojocs. Concretament, en aquest treball s'han creat diverses eines: genèriques, per a crear el moviment de personatge, per a crear menús, per a crear el diàleg, per a crear una IA personalitzable, per a crear els atacs, i finalment, per a crear l'inventari.

1.4 Estructura del document

Aquest treball consta d'11 apartats, explicats a continuació:

1. **Introducció:** en aquest apartat, s'explica en què consisteix el treball, les motivacions que han portat a escollir aquest projecte en concret, i els propòsits i objectius del projecte.
2. **Estudi de viabilitat:** en aquest apartat, es detallen quins han estat els recursos (tècnics, humans i tecnològics) que s'han necessitat per a dur a terme el treball, així com el seu cost total i la viabilitat del projecte.
3. **Planificació:** en aquest apartat, s'exposa quins han estat els passos que s'han dut a terme per a poder assolir els objectius, s'explica a través d'un FlowChart i d'un Gantt Chart.
4. **Marc de treball i conceptes previs**: en aquest apartat s'expliquen els conceptes bàsics necessaris per a poder seguir el projecte.
5. **Disseny de les llibreries:** en aquest apartat es fa una descripció i explicació de les diferents parts principals dels elements de les llibreries.
6. **Implementació:** en aquest apartat es mostra el codi de les llibreries amb explicacions sobre la seva implementació i funcionament.
7. **Resultats:** en aquest apartat, hi ha una demo on s' utilitzen totes les llibres.
8. **Conclusions:** en aquest apartat s'expliquen quins han estat els coneixements adquirits gràcies a aquest projecte, i el grau de compliment dels objectius proposats anteriorment.
9. **Treball futur:** en aquest apartat s'expliquen quines són les tasques que es poden dur a terme com a continuació d'aquest projecte
10. **Bibliografia:** en aquest apartat queden reflectits totes les fonts d'informació utilitzades per a aquest projecte
11. **Manual d'usuari i instal·lació:** en aquest apartat s'explica com utilitzar totes les eines proporcionades en aquest projecte, així com la manera com s'han d'instal·lar.

1.5 Quadre d'avaluació

En la Figura 1-1, queden reflectits els pesos que corresponen a la part estètica, narrativa, mecànica i tecnològica.

Estètica	5%
Narrativa	0%
Mecàniques	30%
Tecnologia	65%

Figura 1-1 Quadre d'avaluació

Es pot observar com el principal objectiu d'aquest treball ha estat crear les bases per a poder crear, posteriorment, les mecàniques dels jocs.

Pel que fa a les mecàniques, algunes de les llibreries inclouen elements prefabricats que permetran a l'usuari aplicar directament aquestes llibreries sense haver de fer treball previ.

Pel que fa a la part artística, en aquest treball, concretament hi ha dissenyada la interfície de l'inventari i els menús.

Finalment, no hi ha narrativa a causa de la pròpia naturalesa del projecte.

2. Estudi de viabilitat

2.1 Recursos tècnics

Per a poder dur a terme el projecte, s'ha necessitat el següent hardware:

Ordinador portàtil:

- Processador: Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
- Ram: 16,0 GB
- Grafica : Nvidia GeForce RTX 2060

Cost dels recursos tècnics

Ordinador portàtil: 1300

2.2 Recursos humans

Per a poder dur a terme el projecte, s'han necessitat els següents recursos humans:

- Programador: implementar les llibreries dissenyades.
- Cap de projecte: és el responsable de revisar que el treball compleixi amb els requisits i les normes que indica la Normativa Reguladora del Treball de Final de Grau de la UDG.

Cost dels recursos humans

- Cap de projecte: 120 dedicades a la documentació més 230 hores dedicades al disseny, a una ràtio de 28 euros/hora
- Programador: 220 hores dedicades a la implementació, a una ràtio de 15 euros/hora

En la Figura 2-1 es pot veure el sumatori del cost dels recursos humans:

Tasca	Perfil	Hores dedicades	Cost
Documentació	Cap de projecte	120	28 euros/hora
Disseny	Cap de projecte	230	28 euros/hora
Implementació	Programador	220	15 euros/hora
Total			**13100**

Figura 2-1 Cost total dels recursos humans

2.3 Requisits tecnològics

Per a dur a terme el treball, s'ha necessitat el següent software:

- Windows 10
- Unity
- VisualStudio

Cost dels requisits tecnològics

Llicència Windows 10: té un cost de 105 euros

2.4 Cost econòmic total

Tenint en compte el cost dels recursos tècnics, els recursos humans i els tecnològics, el projecte ha tingut un cost total de 1300+120*28+230*28+220*15+105 = 14505

2.5 Estudi de mercat i tecnologies existents fins ara

Actualment, un actiu com el creat en aquest projecte, a preu de mercat, val uns 55 euros, adquirint-lo a la Unity Asset Store (Unity Technologies, 2022), tal com es pot veure en les Figures 2-2 i 2-3.

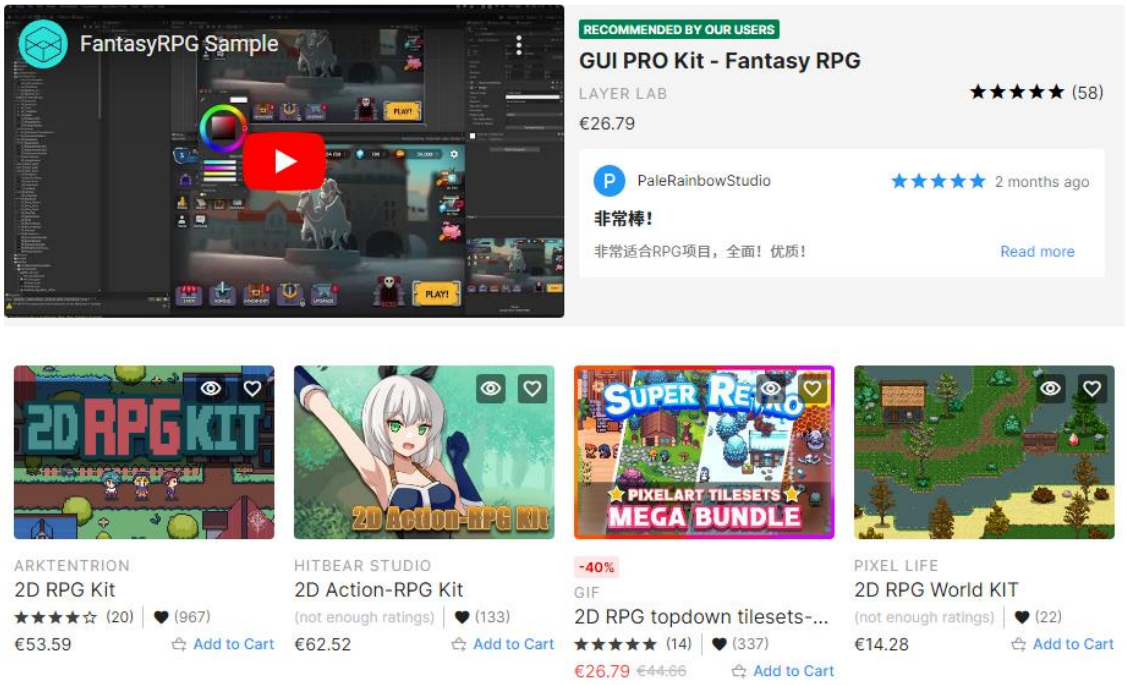


Figura 2-2 Resultats de la Unity Store al buscar 2D RPG

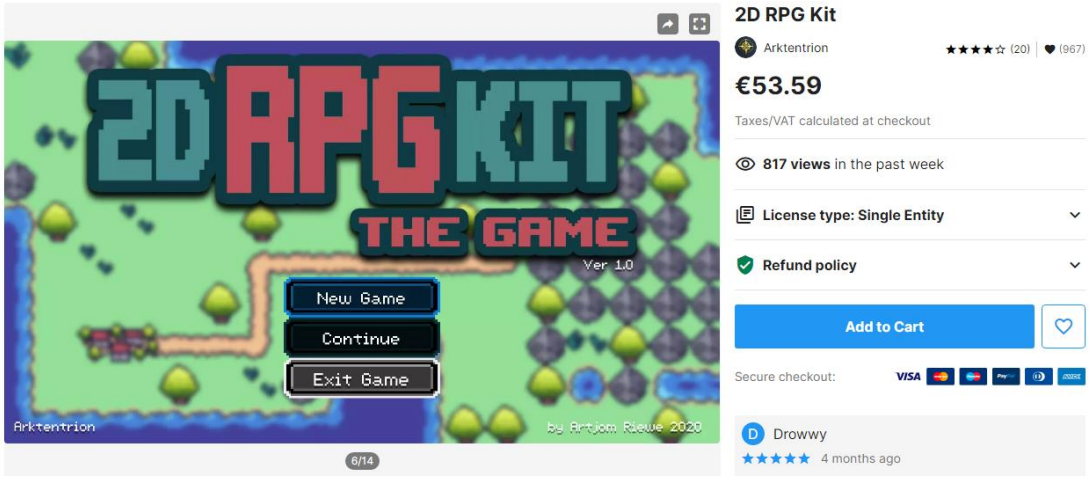


Figura 2-3 Resultats de la Unity Store al buscar 2D RPG

Càlcul del llindar de rendibilitat

Sense tenir en compte tots els costos associats a posar a la venda el producte creat en aquest projecte, calcularem quantes unitats s'haurien de vendre a preu de mercat per tal que aquest projecte fos viable econòmicament.

Per fer això, utilitzarem el càlcul del llindar de rendibilitat del negoci.

Costos fixos: 14505 euros
Costos variables per cada unitat venuda: cap
Preu per unitat: 55

$$\frac{14505}{55-0} = 263{,}73$$

2.6 Conclusions de viabilitat

Sense tenir en compte que tant el hardware com el software de pagament utilitzats poden utilitzar-se per altres projectes, es podria considerar que aquest projecte seria factible econòmicament si, posat a la venda, es venguessin més de 264 unitats.

3. Planificació

3.1 Metodologia

Per a dur a terme aquest treball, s'ha necessitat una metodologia molt concreta a causa de la grandària. En la Figura 3-1 queda representada la metodologia usada mitjançant un Flowchart.

Descripció

El projecte ha consistit en els següents passos:

1. Elecció del projecte: decidir de quin tema es vol fer el projecte i estudiar com funcionen les eines i els recursos necessaris per dur a terme el projecte
2. Elecció de les diferents llibreries: triar quines parts del videojoc de rol requereix una llibreria
3. Disseny d'una llibreria: disseny d'una de les llibreries triades anteriorment
4. Personalització: en cas que no tingui prou elements de personalització, tornar a dissenyar-la.
5. Implementar llibreries: crear la llibreria
6. Generalització: si hi ha elements que es puguin generalitzar, crear aquests elements per separat.
7. Crear elements genèrics: crear l'element genèric
8. Buscar a les llibreries ja implementades llocs on es pugui afegir aquests elements genèrics.
9. Documentació: consisteix a documentar les llibreries creades.

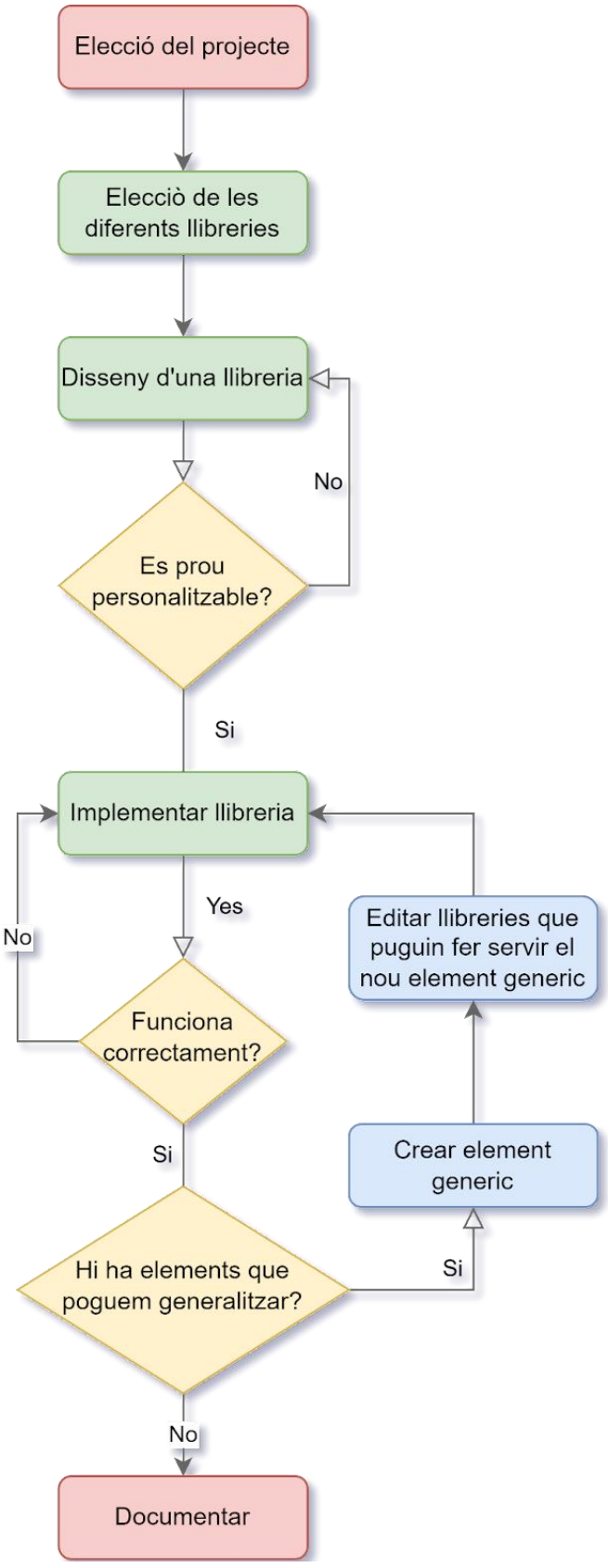


Figura 3-1 Metodologia

3.2 Pla de treball

Mitjançant aquest Gantt Chart (Figura 3-2) es representa la planificació i el temps estimat per a cada tasca.

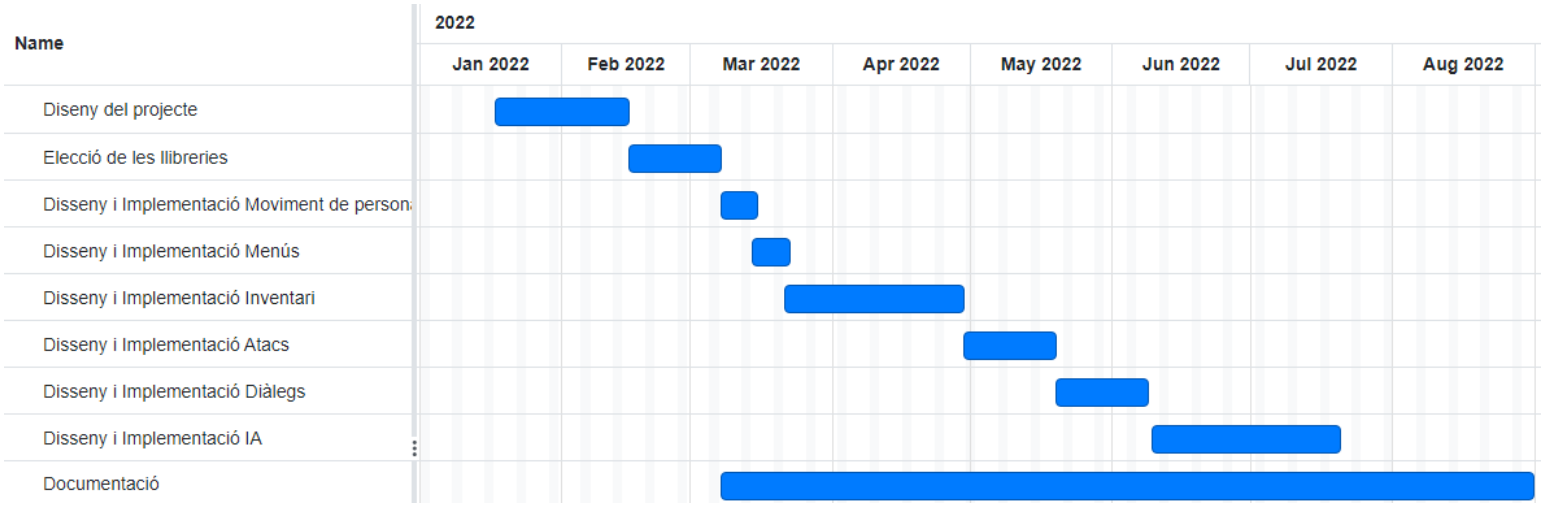


Figura 3-2 Pla de treball

4. Marc de treball i conceptes previs

4.1 Unity

Unity és un entorn de treball per a la creació de videojocs que permet la creació de jocs a un gran nombre de plataformes. Hi ha altres motors gratuïts i bons com pot ser el Godot o l'UnrealEngine, però s'ha escollit aquest motor per la gran quantitat de desenvolupadors que el fan servir i la facilitat de crear i vendre plugins i llibreries (Figura 4-1).

Figura 4-1 Unity

Llenguatge de programació.
Tot el codi del projecte està escrit en el llenguatge de programació C#. És un llenguatge orientat a objectes i amb seguretat de tipus, que té les seves arrels en el llenguatge C.

4.2 Conceptes bàsics

Hi ha una sèrie de conceptes sobre l'entorn de treball de Unity que son necessaris per a entendre el treball realitzat.

Escenes
Les escenes són l'espai de Unity que contenen tots els elements del joc, en un joc hi poden haver diverses escenes, però amb una sola es podria fer un joc sencer. A les escenes hi haurà tant els objectes i personatges del joc com les interfícies d'usuari.

GameObjects
Son els elements que hi ha a les escenes, poden ser objectes, personatges, càmeres, llums o imatges, per posar un exemple.

Prefabs

Són GameObjects que podem crear i Unity els guarda amb totes les característiques, per exemple, podem guardar un enemic com a prefab per fer que apareguin còpies d'aquest on vulguem. A més a més fent servir prefabs si es fa un canvi al Gameobject, aquest canvi també s'aplica a totes les instàncies del prefab.

Classes

Són scripts que descriuen comportaments dels objectes que els tenen. Aquestes classes poden servir per fer que certs objectes tinguin colliders o permeten utilitzar algunes funcions pròpies de cada classe.

GameObject

Són la classe bàsica d'Unity, a partir dels GameObject hi podem posar la resta de components que necessítem.

Component

És la base de tots els components.

Transform

És un component bàsic que tenen tots els objectes que guarda la posició, rotació i escala de l'objecte.

ScriptableObjects

Classe que s'encarrega d'asssentar les bases perquè objectes que s'encarreguen de guardar informació puguin ser creats des de l'editor.

MonoBehaviour

Classe assignable a un objecte que s'encarrega de realitzar funcions cada fotograma i detectar col·lisions entre d'altres.

Colliders

S'encarreguen de crear una zona en la qual, al detectar contacte amb un altre collider, activen funcions del MonoBehaviour (Figura 4-2).

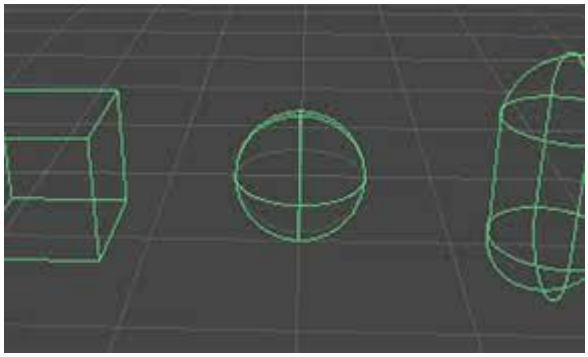


Figura 4-2 Colliders

Triggers

Són similars als colliders, però no tenen col·lisió. Igualment activen funcions al MonoBehaviour al detectar colliders que passin per sobre.

RigidBody

Permet que un GameObject tingui físiques, respon a la gravetat i a impactes d'altres colliders.

Interfície

És la part d'Unity on es veuen diferents finestres perquè l'usuari vegi l'escena en la qual treballa, informació sobre els objectes de l'escena, accés a les carpetes on es guarden, entre d'altres, scripts i prefabs, entre altres coses (Figura 4-3).

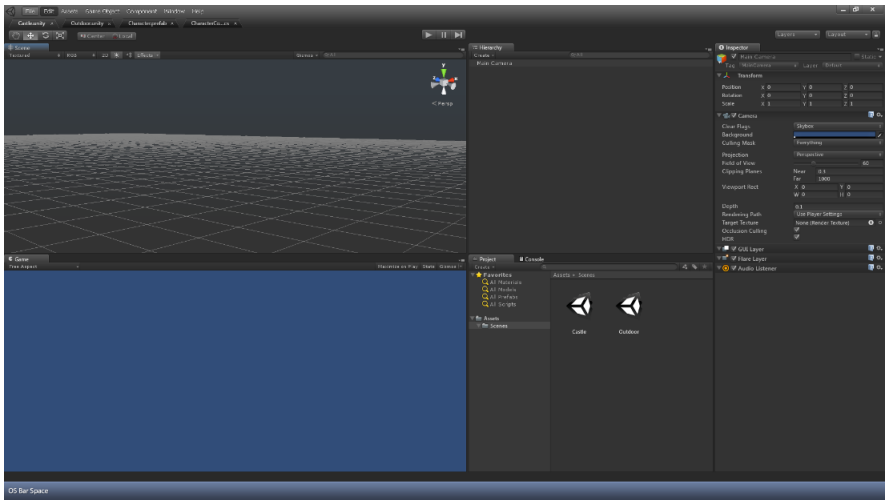


Figura 4-3 Interfície de Unity

(Unity Technologies, 2022)

5. Disseny de les llibreries

5.1 Global

Hi ha scripts que serveixen per a ajudar a fer que la resta de funcions i scripts funcionin correctament. Alguns d'aquests tenen funcions tan genèriques que no té sentit explicar-los junt amb la classe que els fa servir, per això els explicarem en aquesta categoria a part.

Podem dividir aquestes categories en:

- Detectors
- Interaccions
- Tags
- Temporitzador

5.1.1 Detectors

Són components que a l'activar-se retornen els Colliders que han trobat a l'àrea de cerca. Els Triggers poden fer una funció similar, però la diferencia és que els detectors no estan actius constantment, això fa que siguin més eficients en els casos en els que nomes es vulgui detectar els Colliders d'una zona en moments determinats.

5.1.2 Interaccions

Són components que permeten al jugador interactuar amb diferents objectes de l'escena. Per posar algun exemple, aquests elements es poden fer servir per interactuar amb botons que obren portes o cartells que ensenyen diàlegs

5.1.3 Temporitzador

És una classe que serveix per a activar funcions quan passa un temps determinat. També es pot fer servir en bucle.

5.1.4 Tags

Un Tag és una paraula de referència que es pot assignar als objectes. Aquests serveixen per identificar-los sense que faci falta veure res més sobre l'objecte, estalviant molt de temps de processament. A més a més, serveixen per agrupar objectes en categories evitant la feina de crear condicions per cada objecte.

Tags a Unity:

A Unity hi ha un sistema de tags ja creat, tal com es pot veure a la Figura 5-1. El sistema funciona de la següent forma:

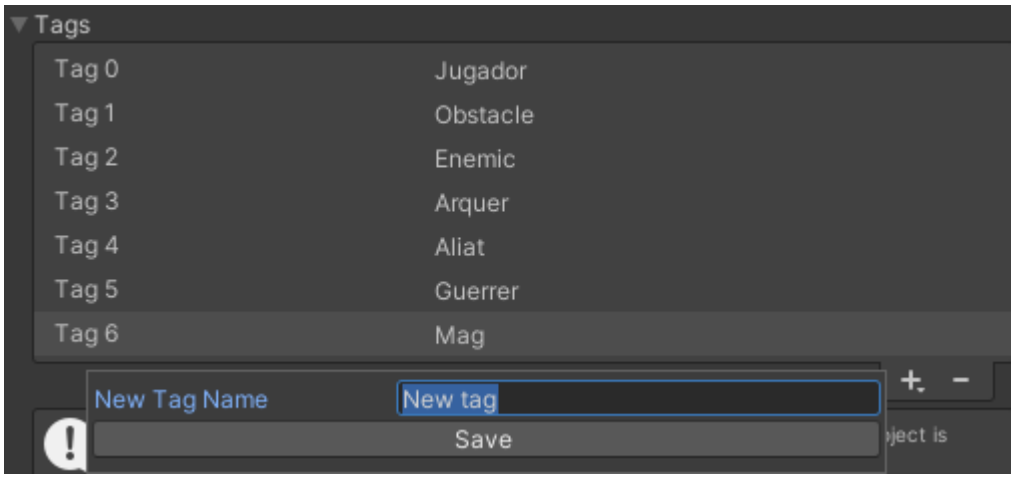


Figura 5-1 Sistema de Tags ja creat

Unity guarda una llista de tags que fa servir per identificar els objectes del joc. Aquesta llista es pot ampliar i afegir noves tags amb altres noms en cas de necessitar-les.

El sistema de tags d'Unity té una mancança i és que només es pot assignar una tag per objecte. Aquesta característica del sistema d'Unity fa que, en certs casos, s'hagi de crear moltes més tags de les inicialment necessàries. Per posar un exemple, si volguéssim posar el tag d'arquer en un objecte i també la tag d'enemic, hauríem de crear una tag d'arquer enemic, una d'arquer aliat i una d'arquer neutre.

Es pot veure com, en cas de necessitar més d'una tag per objecte, no només creix exponencialment el nombre de tags a crear, sinó que també s'han de fer casos especials per cadascuna d'aquestes tags, fent que el principal avantatge dels tags desaparegui. Per aquest motiu s'ha creat un sistema de tags alternatiu que permet la creació de conjunts de tags per a resoldre aquest problema.

Tags creades:

Donades les restriccions dels tags a Unity, s'ha creat un nou sistema de tags que permet assignar més d'una tag a un objecte.

5.2 Moviment del personatge

Definició

En un joc de rol sol ser essencial que el personatge principal es pugui moure, és necessari tant per avançar, arribar als objectius que se'ns han marcat o esquivar enemics, entre d'altres.

Decisions a l'hora de fer el moviment de personatge

Unes de les característiques més distintives entre els diferents jocs de rol és el moviment del personatge. Hi ha moltíssimes formes diferents de fer-lo i és dels elements que li sol donar més personalitat al joc.

Per aquest motiu, en comptes de fer molts scripts amb moviments molt complicats, com podria ser tenir en compte doble salts, esprint, enganxar-se a les parets i ajupir-se entre d'altres, s'ha decidit fer dos tipus de moviment de personatge senzills, un per a jocs en horitzontal i l'altre per a jocs amb vista des de dalt. D'aquesta manera els usuaris podran utilitzar aquesta llibreria per a fer les primeres iteracions del joc que estiguin fent i, si volen perfeccionar el moviment del personatge un cop tinguin el joc més polit, ho puguin fer.

Personalització

Els dos tipus de moviments d'aquesta llibreria estan dissenyats de forma que no s'hagi d'entrar al codi per a editar el moviment final del personatge. D'aquesta manera, hi ha paràmetres que activen, desactiven i ajusten diferents comportaments del personatge.

5.3 Menú

Definició

Els menús són una part fonamental, que no només és important pels jocs de rol sinó que és essencial per a gairebé tots els jocs. Els menús permeten canviar opcions del joc com la qualitat i el volum. També permeten fer pausa, avisen si es guanya o es perd i permeten canviar de nivell.

Elements generals dels menús

Com que volem fer un conjunt de menús de forma genèrica, el més important és donar el màxim d'eines a l'usuari, així com una base funcional que pugui canviar al seu gust. Els punts que podem trobar en comú en la gran majoria de menús són:

- Menú principal, on hi ha opcions, la possibilitat de sortir del joc així com la de començar el joc.
- Menú de pausa, on hi ha opcions, la possibilitat de continuar jugant i anar al menú principal
- Menú d'opcions, on hi ha diferents opcions sobre el joc.
- Pantalla que surt quan es guanya i permet anar al següent nivell.
- Pantalla que surt quan es perd i permet tornar a començar el nivell.

Personalització

Perquè l'usuari tingui control sobre els menús, totes les funcions estan a un sol objecte, que serà el MenuController. Aquest té tot el necessari perquè la lògica dels menús funcioni. Permet a l'usuari crear altres menús com podria ser selecció de nivells, entre d'altres, sense haver de tocar codi.

5.4 Inventari

Definició

En la gran majoria de jocs de rol hi podem trobar inventaris. Els inventaris ens poden permetre guardar objectes que ens deixin els enemics o que aconseguim d'altres formes. Els inventaris poden tenir moltes altres opcions a part de guardar els objectes que ens trobem, també es poden fer servir per intercanviar objectes amb algun contenidor, com pot ser un cofre, o ens poden permetre equipar-nos d'objectes que millorin les nostres característiques o armes per atacar.

Què té un inventari genèric?

Perquè l'inventari que creem sigui el més útil possible, ha de ser genèric. Això vol dir que hauria de tenir moltes funcionalitats comunes dels inventaris i adaptar-se a les necessitats de l'usuari. Els punts que podem trobar en comú en la gran majoria d'inventaris són:

- Elements o ítems, son els diferents objectes que guardem.
- Ranures, que són llocs on deixar i extreure ítems.
- Possibilitat de posar més d'un ítem a una ranura.
- Moure ítems d'un inventari a un altre.
- Diferents formes d'agafar i deixar ítems per agafar o deixar diferents quantitats.
- Recollir ítems i que vagin a l'inventari.

Personalització

Perquè l'usuari tingui control sobre com serà l'inventari final, podrà canviar variables com poden ser la mida de l'inventari, la posició de les ranures i tota la informació sobre els ítems, des de les imatges, els noms i les descripcions, fins al màxim d'ítems per ranura.

5.5 Sistema d'atac

Definició

En un joc de rol hi ha enemics, trampes, entre d'altres que poden ferir el personatge. El personatge també podrà atacar a cretes entitats com per exemple, els enemics. Per això, dissenyar un sistema d'atacs és important per a un joc de rol.

Què té un sistema d'atacs?

Hi ha una sèrie d'elements comuns que podem trobar a la majoria de jocs sobre com s'ataca i com es rep l'atac. Els elements genèrics més comuns podrien ser:

- Tipus d'atac (atac directe, atacs que s'apliquen en un interval, com el verí, atacs que treuen un percentatge de la vida, etc.).
- Tipus de dany (foc, aigua, verí).
- Defenses (reducció del mal).
- Vida.
- Formes d'atacar (a l'entrar en una àrea, quan ho activa algun element com podria ser un cop o tret).

Personalització

Tenint en compte tota aquesta varietat d'opcions que tenim a l'hora de fer diferents atacs, hem creat un sistema que permet crear atacs, defenses, objectes que fan dany i objectes que el reben, amb la possibilitat de personalitzar-los amb tots els elements explicats anteriorment.

5.6 Diàlegs

Definició

Per a donar informació al jugador o per realitzar converses entre personatges, els jocs solen tenir un sistema de diàleg. Aquest permet al jugador llegir cartells o mantenir converses amb els altres personatges.

Què necessitem?

A l'hora de crear un sistema de diàleg hi ha un seguit de premisses que hem de tenir en compte perquè sigui el més útil possible. El sistema hauria de permetre a l'usuari canviar la disposició dels elements de la interfície d'usuari que mostrarà el diàleg, així com el tipus de lletra i l'estètica de la interfície.

També fa falta que sigui un sistema molt dinàmic, de forma que puguem, en tot moment, canviar l'ordre de parts de la conversa o afegir noves parts.

Així mateix, volem un sistema que ens permeti tenir diverses opcions a triar, si així ho volem, per a contestar a la conversa. Això permetrà, per exemple, que el personatge pugui fer preguntes o triar entre opcions per respondre. Aquestes opcions han de tenir la qualitat de poder saltar d'un punt de la conversa a un altre, i aquest sistema no s'hauria de trencar si edítem la conversa i afegim o esborrem parts del diàleg.

5.7 Context Stearing AI

Per què una IA

Tots els jocs de Rol, i una gran part de la resta de jocs, tenen alguna forma d'intel·ligència artificial, des de la més simple, com podria ser detectar el jugador i avançar cap a ell, fins a la més complexa, on els personatges tenen rutines, personalitat o altres coses que els fan com a éssers intel·ligents. Per a facilitar als creadors de jocs la creació d'una IA, s'ha creat un sistema de creació altament personalitzable.

Per què aquesta IA?

Unity consta d'un sistema de pathfinding que es pot fer servir per fer una IA senzilla que vagi cap al Player. Per aquest motiu, crear una IA del mateix tipus no servia de gaire, ja que ja està implementada.

Per l'altra banda, com que la IA és una gran part de la creació d'un joc de rol, és important que cada creador de jocs la pugui personalitzar de la forma que més li convingui. Per tant, la IA que s'havia de fer havia de ser molt personalitzable. Finalment, s'ha arribat a la conclusió que un dels sistemes d'IA que encaixava es el Context Stearing AI. Aquesta intel·ligència artificial fa servir els elements del seu voltant per decidir en quina direcció es vol moure.

Limitacions de la IA creada

Aquest sistema pot fer moltes coses a partir d'una sèrie de condicions fàcils d'implementar.

Amb poques condicions, es creen comportaments senzills, com pot ser perseguir el jugador, o allunyar-se de les parets.

Quan aquestes condicions s'ajunten, creen comportaments més orgànics i intel·ligents com agrupar-se amb els de la seva espècie, però allunyar-se si estan massa a prop, mentre s'allunyen de certs perills o passen pel seu voltant en el cas de voler arribar a qualsevol cosa que estigui a l'altre costat del perill.

Funcionament

El funcionament del Context Stearing AI és relativament senzill. Cada personatge té un seguit de condicions assignades per a cada element del seu voltant que els hi interessi, per exemple, la condició que si estan a prop d'una paret, s'haurien d'allunyar una mica. Cada una d'aquestes condicions té una reacció associada. Aquesta reacció consta d'un pes que serveix per decidir si la reacció és important o no, un conjunt de vectors que serviran per decidir la direcció en la qual s'aplica aquesta reacció i finalment una forma de determinar si és una reacció a alguna cosa a la qual ens volem apropar o una que volem evitar. A l'hora de decidir la direcció final es tenen en compte totes les reaccions a tots els objectes del seu voltant, per decidir quina és la millor direcció a seguir.

6. Implementació

6.1 Global

En aquesta secció explicarem la implementació dels sistemes descrits a l'apartat 5.1. Són sistemes que es poden fer servir tant als elements principals de les llibreries, com a sistemes independents que podrà fer servir el desenvolupador.

6.1.1 Lògica interna dels Detectors

Detector

L'estructura interna dels detectors parteix de la classe abstracta Detector que té les funcions abstractes Detect i DetectAll, tal com es pot veure a la Figura 6-1.

```
public abstract Collider2D[] DetectAll(int? layerMask = null);
4 referencias
public abstract Collider2D Detect(int? layerMask = null);
```

Figura 6-1 Funcions DetectAll i Detect

També té la versió amb Tags implementada. Aquesta versió fa servir el mètode DetectAll i busca entre els resultats si hi ha objectes que tinguin les Tags que busquem, tal com es pot veure a les Figures 6-2 i 6-3.

```
public Collider2D[] DetectAllTag(TagList tl = null)
{
    if (tl == null) return DetectAll();
    List<Collider2D> colliders = new List<Collider2D>();
    foreach (Collider2D c in DetectAll())
    {
        if (c.GetComponent<Tagged>())
        {
            if (tl.isValid(c.GetComponent<Tagged>().TagList))
            {
                colliders.Add(c);
            }
        }
    }
    return colliders.ToArray();
}
```

Figura 6-2 Funció DetectAllTag

```
public Collider2D DetectTag(TagList tl = null)
{
    if (tl == null) return Detect();
    foreach (Collider2D c in DetectAll())
    {
        if (c.GetComponent<Tagged>())
        {
            if (tl.isValid(c.GetComponent<Tagged>().TagList))
            {
                return c;
            }
        }
    }
    return null;
}
```

Figura 6-3 DetectTag

CircleDetector/RayDetector/BoxDetector

Aquestes tres classes funcionen gairebé de la mateixa forma, per això només mostrarem el RayDetector. Aquestes funcions (Figures 6-4 i 6-5) implementen els mètodes Detect i DetectAll fent un cast, en el cas del RayDetector un Raycast, fent servir les variables donades com a paràmetres.

```
public override Collider2D Detect(int? layerMask = null)
{
    TransformVariables();
    Collider2D collider;
    if (layerMask != null)
    {
        collider = Physics2D.Raycast(_position, _direction,
            _detectionDistance, (int)layerMask).collider;
    }
    else
    {
        collider = Physics2D.Raycast(_position, _direction,
            _detectionDistance).collider;
    }
    return collider;
}
```

Figura 6-4 Funció Detect

```
public override Collider2D[] DetectAll(int? layerMask = null)
{
    TransformVariables();
    Collider2D[] colliders;
    RaycastHit2D[] hits;
    if (layerMask != null)
    {
        hits = Physics2D.RaycastAll(_position, _direction,
            _detectionDistance, (int)layerMask);
    }
    else
    {
        hits = Physics2D.RaycastAll(_position, _direction,
            _detectionDistance);
    }
    colliders = new Collider2D[hits.Length];
    for(int i = 0; i < hits.Length; i++)
    {
        colliders[i] = hits[0].collider;
    }
    return colliders;
}
```

Figura 6-5 Funció Detect all

Finalment, el RayDetector té una funció més que els altres detectors, que serveix per inicialitzar les variables de posició i direcció a partir dels paràmetres donats (vegeu Figura 6-6).

```
private void TransformVariables()
{
    if (_origin != null)
        _position = _origin.position;
    if (_target != null)
        _direction = _target.position - _origin.position;
}
```

Figura 6-6 Funció TransformVariables

6.1.2 Implementació Interaccions

Interacter/Interactuable

L'Interacter consta d'un Detector i una tecla que activa la detecció (Figura 6-7).

```
public Detector _detector;
public KeyCode _interactKey = KeyCode.E;
```

Figura 6-7 Variables Detector

Cada frame mira si s'ha clicat la tecla i, en el cas que s'hagi clicat, activa el detector i tots els Interactuables que s'hagin detectat (vegeu Figura 6-8).

```
private void Update()
{
    if (Input.GetKeyDown(_interactKey))
    {
        foreach(Collider2D c in _detector.DetectAll())
        {
            c.GetComponent<Interactuable>()?.onInteraction();
        }
    }
}
```

Figura 6-8 Funció Update de Detector

L'Interactuable activa un UnityEvent quan l'Interacter hi interacciona (vegeu Figura 6-9)

```
public UnityEvent _interacted;
1 referencia
public void onInteraction()
{
    if(Time.timeScale != 0)
    _interacted.Invoke();
}
```

Figura 6-9 Interactuable

OnCollision2D / OnTriggerer2D

L'script OnCollision2D funciona de la següent forma: cada cop que les funcions OnCollisionEnter2D, OnCollisionExit2D o OnCollisionStay2D es criden, activem els UnityEvents corresponents, tal com es pot veure a la Figura 6-10.

```
public UnityEvent<Collision2D> collisionEnter;
public UnityEvent<Collision2D> collisionExit;
public UnityEvent<Collision2D> collisionStay;

Mensaje de Unity | 0 referencias
private void OnCollisionEnter2D(Collision2D collision)
{
    collisionEnter.Invoke(collision);
}
Mensaje de Unity | 0 referencias
private void OnCollisionExit2D(Collision2D collision)
{
    collisionExit.Invoke(collision);
}
Mensaje de Unity | 0 referencias
private void OnCollisionStay2D(Collision2D collision)
{
    collisionStay.Invoke(collision);
}
```

Figura 6-10 OnCollision2D

En el cas del OnTriggerer2D, fem el mateix però canviant les funcions OnCollision per a funcions OnTrigger (vegeu Figura 6-11)

```
public UnityEvent<Collider2D> triggerEnter;
public UnityEvent<Collider2D> triggerExit;
public UnityEvent<Collider2D> triggerStay;

Mensaje de Unity | 0 referencias
private void OnTriggerEnter2D(Collider2D collision)
{
    triggerEnter.Invoke(collision);
}
Mensaje de Unity | 0 referencias
private void OnTriggerExit2D(Collider2D collision)
{
    triggerExit.Invoke(collision);
}
Mensaje de Unity | 0 referencias
private void OnTriggerStay2D(Collider2D collision)
{
    triggerStay.Invoke(collision);
}
```

Figura 6-11 OnTrigger2D

ClickableObject

El ClickableObject hereta de IPointerClickHandler i detecta tots els elements de la interfície que estan a sota del ratolí quan el jugador fa clic i activa un UnityEvents depenent del clic.

També té un booleà estàtic que, en cas de ser cert, hi deixen d'haver interaccions, tal com es pot veure a la Figura 6-12.

```
public UnityEvent leftClick;
public UnityEvent middleClick;
public UnityEvent rightClick;
public UnityEvent<int> click;

public static bool endRay = false;
```

Figura 6-12 Variables ClickableObject

La funció OnPointerClick (Figura 6-13) forma part de IPointerClickHandler i s'activa quan hi ha un clic. La seva funció és cridar la funció letRayThrough que té la lògica que activa tots els ClickableObject que estan sota el ratolí.

```
public void OnPointerClick(PointerEventData eventData)
{
    letRayThrough(eventData);
}
```

Figura 6-13 Funció OnPointerClick

La funció letRayThrough (Figura 6-14) fa un rayCast per a seleccionar tots els ClickableObject que estiguin sota el ratolí i crida per cadascú d'ells la funció OnClick.

```
private static void letRayThrough(PointerEventData eventData)
{
    List<RaycastResult> raycastResults = new List<RaycastResult>();
    EventSystem.current.RaycastAll(eventData, raycastResults);
    IEnumerable<RaycastResult> newTarget = raycastResults
        .Where(rcr => rcr.gameObject.GetComponent<ClickableObject>() != null);
    endRay = false;
    foreach (RaycastResult r in newTarget)
    {
        if (endRay) break;
        r.gameObject.GetComponent<ClickableObject>()?.doClick(eventData);

    }
}
```

Figura 6-14 Funció letRayThrough

Finalment, la funció OnClick (Figura 6-15) activa els UnityEvents corresponents.

```
public void doClick(PointerEventData eventData)
{
    click.Invoke((int)eventData.button);

    if (eventData.button == PointerEventData.InputButton.Left)
        leftClick.Invoke();

    else if (eventData.button == PointerEventData.InputButton.Middle)
        middleClick.Invoke();

    else if (eventData.button == PointerEventData.InputButton.Right)
        rightClick.Invoke();
}
```

Figura 6-15 Funció OnClick

6.1.3 Implementació Temporitzador

Perquè el temporitzador funcioni fan falta una sèrie de variables. Aquestes són el mètode a cridar quan s'acabi el temps, el temps que tardarà a activar-se, si està actiu o no, si està en bucle i, en cas que el bucle sigui limitat, el nombre de vegades que s'ha de fer el bucle.

Tenim setters del mètode a cridar i el temps a esperar així com getters de la duració (vegeu Figura 6-16)

```
public delegate void MethodToCall();
private MethodToCall s_method;

private float s_time;// time
private bool isRunning = false;// if timer is runing
private int? nLoop = null;// n of loops
private bool isLoop = false;// if is looping
```

Figura 6-16 Variables Temporitzador

Primer caldrà inicialitzar el temporitzador (vegeu Figura 6-17)

```
public Timer(MethodToCall method,float duration = 1f)
{
    s_time = duration;
    s_method = method;
}
```

Figura 6-17 Inicialitzador del temporitzador

La funció start (Figura 6-18) fa que el temporitzador comenci.

```
public void start()
{
    isRunning = true;
    beginTimer();
}
```

Figura 6-18 Funció Start

La funció stop (Figura 6-19) para el temporitzador.

```
public void stop()
{
    isRunning = false;
}
```

Figura 6-19 Funció Stop

La funció loopN (Figura 6-20) inicialitza el bucle n vegades.

```
public void loopN(int n)
{
    if (n <= 0) return;
    nLoop = n;
    isLoop = n > 0;
}
```

Figura 6-20 Funció loopN

La funció loop (Funció 6-21) inicialitza el bucle de forma indefinida.

```
public void loop()
{
    isLoop = true;
}
```

Figura 6-21 Funció Loop

La funció timeEnded (Figura 6-22) s'activa un cop acaba el temps i s'encarrega d'activar el mètode i de tornar a començar el cicle en cas d'haver-hi bucle.

```
private void timeEnded()
{
    if (!isRunning) return;
    s_method();
    if (!isLoop) { isRunning = false; }
    else if (nLoop != null)
    {
        nLoop -= 1;
        if (nLoop <= 0)
        { isLoop = false; isRunning = false; }
    }
}
```

Figura 6-22 Funció timeEnded

La funció beginTimer (Figura 6-23) busca un MonoBehaviour perquè faci servir la corutina que servirà per a activar el temporitzador.

```
private void beginTimer()
{
    UnityEngine.MonoBehaviour _mb =
        UnityEngine
        .Object
        .FindObjectOfType<UnityEngine.MonoBehaviour>();
    if (_mb != null)
    {
        _mb.StartCoroutine(MyEvent());
    }
    else {
        UnityEngine.Debug.LogError(
            "Timer not active: There is no MonoBehaviour in the scene");
    }
}
```

Figura 6-23 Funció beginTimer

La funció MyEvent (Figura 6-23) és la que s'encarrega d'activar la funció WaitForSeconds d'Unity que, un cop acabi el temps, activarà la funció timeEnded.

```
private System.Collections.IEnumerator MyEvent()
{
    while (isRunning)
    {
        yield return new UnityEngine.WaitForSeconds(s_time);
        timeEnded();
    }
}
```

Figura 6-24 Funció MyEvent

6.1.4 Implementació Tags

La implementació dels Tags és força senzilla La classe Tag és un ScriptableObject amb un string i un getter (vegeu Figura 6-25).

```
[CreateAssetMenu(menuName = "Tags/Tag")]
Script de Unity | 3 referencias
public class Tag : ScriptableObject
{
    [SerializeField] private string _tag;
    0 referencias
    public string getTag => _tag;

}
```

Figura 6-25 Classe Tag

La classe TagList (Figura 6-26) consta d'una llista de tags i una funció isValid que rep una TagList i retorna un booleà depenent de si tenen tags en comú o no.

```
public class TagList : ScriptableObject
{
    [SerializeField]private List<Tag> _tagList;
    1 referencia
    public List<Tag> tagList => _tagList;

    1 referencia
    public bool isValid(TagList tl)
    {
        foreach (Tag t1 in _tagList)
        {
            if (tl.tagList.Contains(t1)) return true;
        }
        return false;
    }
}
```

Figura 6-26 Classe TagList

La classe Tagged (Figura 6-27) és un MonoBehaviour amb una Taglist i un getter.

```
public class Tagged : MonoBehaviour
{
    [SerializeField]private TagList tagList;
    1 referencia
    public TagList TagList => tagList;
}
```

Figura 6-27 Classe Tagged

6.2 Lògica interna del moviment

En un sistema de moviment, el més important és transformar els inputs de l'usuari al moviment que farà el personatge. També és rellevant tenir control sobre el grau d'efecte dels inputs.

La lògica interna de moviment ens permetrà modificar com afecta l'usuari al moviment del personatge, i en alguns casos en particular també podrem controlar com afecta l'entorn al moviment.

PlayerController2DTopDown

PlayerController2DTopDown: aquest script implementa tota la lògica de moviment d'un joc vist amb vista des de dalt. Permet tant el moviment del personatge com la funcionalitat d'esprint. Les variables que fa servir com a entrada són: velocitat, duració i tecla predeterminada de l'esprint, junt amb un booleà per saber si està actiu (vegeu Figura 6-28). També fa servir una variable per a la velocitat del personatge.

```
public bool dash = true;
public float dashSpeed = 15;
public float dashDuration = 0.3f;
public KeyCode dashKey = KeyCode.LeftShift;
public float speed = 5;
```

Figura 6-28 Variables de PlayerController2DTopDown

Hi ha altres variables que fa servir per a guardar l'estat intern, com pot ser un temporitzador, o si s'està fent esprint en aquest moment (vegeu Figura 6-29)

```
Timer endDashTimer;
bool dashing = false;
```

Figura 6-29 Temporitzador

El programa, primer de tot, inicialitza el temporitzador amb les dades donades, vegeu Figura 6-30:

```
void Start()
{
    endDashTimer = new Timer(EndDash, dashDuration);
}
```

Figura 6-30 Inicialitzador del Temporitzador

I per cada fotograma actualitza l'input i l'estat de l'esprint (vegeu Figura 6-31)

```
void Update()
{
    horizontal = Input.GetAxisRaw("Horizontal");
    vertical = Input.GetAxisRaw("Vertical");

    if (dash && Input.GetKeyDown(dashKey))
    {
        dashing = true;
        endDashTimer.start();
    }
    if (dash && Input.GetKeyUp(dashKey))
    {
        EndDash();
    }
}
```

Figura 6-31 Funció Update de PlayerController2DTopDown

La funció FixedUpdate (vegeu Figura 6-32) actualitza la posició de l'objecte 50 vegades per segon, amb les opcions de sèrie (Unity Technologies, 2022).

```
private void FixedUpdate()
{
    this.gameObject.transform.position += new Vector3(
        horizontal * (dashing? dashSpeed: speed),
        vertical * (dashing ? dashSpeed : speed),
        0)
        *Time.fixedDeltaTime;
}
```

Figura 6-32 Funció FixedUpdate

Finalment, tenim la funció Moving (vegeu Figura 6-33) que retorna cert si el personatge s'està movent activament.

```
public bool Moving()
{
    return horizontal != 0 || vertical != 0;
}
```

Figura 6-33 Funció Moving

PlayerController2DHorizontal

PlayerController2DHorizontal: aquest script implementa tota la lògica de moviment d'un joc vist de forma que la gravetat vagi cap a sota la pantalla.

Tenim una sèrie de variables (vegeu Figura 6-34) per a modificar el moviment, el salt, l'esprint i el comportament que té en estar a prop d'una paret del personatge.

```
[Header("Move")]
public KeyCode leftKey = KeyCode.A;
public KeyCode rightKey = KeyCode.D;
public float speed = 5;
[Space(10)]
[Header("Wall")]
public bool wallStick = true;
public bool wallRefreshOneJump = true;
public float wallFallSpeed = 0.5f;
[Space(10)]
[Header("Jump")]
public int airJump = 1;
public KeyCode jumpKey = KeyCode.Space;

public float jumpForce = 6;
public float gravitiOnFall = 2.5f;
[Space(10)]
[Header("Dash")]
public bool dash = true;
public KeyCode dashKey = KeyCode.LeftShift;
public float dashSpeed = 10;
```

Figura 6-34 Variables de PlayerController2DHorizontal

També tenim variables auxiliars (vegeu Figura 6-35) que ens informen de l'estat del personatge.

```
int isOnWall = 0;
bool isGrounded = false;
int airJumpsRemaning = 0;
int direction = 0;
```

Figura 6-35 Variables Auxiliars

Quant a funcions, primer de tot, s'inicialitzen les variables de RigidBody i la gravetat inicial (vegeu Figura 6-36).

```
void Start()
{
    rb = GetComponent<Rigidbody2D>();
    initialGravity = rb.gravityScale;
}
```

Figura 6-36 Inicialització variables PlayerController2DHorizontal

Amb la funció Update (vegeu Figura 6-37) s'actualitzen la direcció i l'estat de l'esprint i el salt.

```
void Update()
{
    //Movement
    CalculateDirection();

    //Dash
    CalculateDash();

    //Jumps
    CalculateJump();
}
```

Figura 6-37 Funció Update de PlayerController2DHorizontal

I amb el FixedUpdate es mou el personatge (vegeu Figura 6-38).

```
private void FixedUpdate()
{
    this.transform.position += transform.right
        * direction
        * speed
        * Time.fixedDeltaTime;
}
```

Figura 6-38 FixedUpdate de PlayerController2DHorizontal

Tenim tres funcions que s'encarreguen d'actualitzar el moviment del personatge:

- La funció CalculateDirection (vegeu Funció 6-39) s'encarrega del moviment basic depenent de la tecla que premem.

```
private void CalculateDirection()
{
    if (Input.GetKeyDown(leftKey))
        { direction = -1; rb.velocity = new Vector2(0, rb.velocity.y); }

    else if (Input.GetKeyDown(rightKey))
        { direction = 1; rb.velocity = new Vector2(0, rb.velocity.y); }

    if (Input.GetKeyUp(leftKey) && direction < 0)
        { direction = 0; rb.velocity = new Vector2(0, rb.velocity.y); }

    else if (Input.GetKeyUp(rightKey) && direction > 0)
        { direction = 0; rb.velocity = new Vector2(0, rb.velocity.y); }
}
```

Figura 6-39 Funció CalculateDirection

- La següent funció (vegeu Figura 6-40), s'encarrega d'aplicar el esprint quan rep l'input correcte.

```
private void CalculateDash()
{
    if (dash && Input.GetKeyDown(KeyCode.LeftShift) && direction != 0)
    {
        rb.AddForce(new Vector2(dashSpeed * 1000 * direction, 1));
    }
}
```

Figura 6-40 Funció CalculateDash

- Finalment CalculateJump (vegeu Figura 6-41) té la lògica que regula els salts així com el comportament del personatge en estar tocant una paret.

```
private void CalculateJump()
{
    if (Input.GetKeyDown(jumpKey))
    {
        rb.gravityScale = initialGravity;

        //onWall
        if (isOnWall != 0 && wallStick && direction == 0)
        {
            rb.velocity = new Vector2(
                jumpForce / 1.5f * -isOnWall, jumpForce / 1.5f);
        }
        //not onWall
        else
        {
            if (isGrounded || (airJumpsRemaning != 0))
            {
                rb.velocity = new Vector2(rb.velocity.x, jumpForce);
                if (!isGrounded) { airJumpsRemaning -= 1; }
            }
        }
    }
    else if (!isGrounded && Input.GetKeyUp(jumpKey))
    {
        rb.gravityScale = gravitiOnFall;
    }
}
```

Figura 6-41 Funció CalculateJump

Les funcions hasGrounded (vegeu Figura 6-42) i hasWalled (vegeu Figura 6-43) s'encarreguen d'actualitzar les variables "isGrounded" i "airJumpsRemaining" si el personatge està a terra o contra una paret.

```
void hasGrounded()
{
    isGrounded = true;
    airJumpsRemaning = airJump;
}
```

Figura 6-42 Funció hasGrounded

```
void hasWalled(float x)
{
    isOnWall = x>0?1:-1;
    if(rb.velocity.y < -wallFallSpeed)
    {
        rb.velocity = new Vector2(rb.velocity.x, -wallFallSpeed);
    }
    if (wallRefreshOneJump && airJumpsRemaning == 0)
        airJumpsRemaning = 1;
}
```

Figura 6-43 Funció hasWalled

Finalment, tenim dues funcions que detecten col·lisions i, depenent d'aquestes i de la direcció amb la qual xoca el personatge, determinen si és una paret o el terra i fan els canvis necessaris.

En cas de xocar s'activa la funció OnCollisionStay2D (vegeu Figura 6-44)

```
private void OnCollisionStay2D(Collision2D collision)
{
    Vector2 direction = (collision.contacts[0].point
        - new Vector2(
            this.gameObject.transform.position.x,
            this.gameObject.transform.position.y))
            .normalized;

    if(direction.y < -0.6)
    {
        hasGrounded();
    }
    if (wallStick && Mathf.Abs(direction.x) > 0.6)
    {
        hasWalled(direction.x);
    }
}
```

Figura 6-44 Funció OnCollisionStay2D

I en cas de sortir d'una col·lisió (vegeu Figura 6-45)

```
private void OnCollisionExit2D(Collision2D collision)
{
    isGrounded = false;
    isOnWall = 0;
}
```

Figura 6-45 Funció OnCollisionExit2D

6.3 Lògica interna del MenuController

Perquè el sistema de menús funcioni, fa falta un controlador que s'encarregui de tota la lògica que farà falta. Per això totes les funcions que cridaran els botons dels menús estan al mateix objecte, el MenuController.

MenuController

El MenuController (Figura 6-46) és una classe que té una sèrie de variables per a gestionar quins menús es mostren per pantalla, algunes opcions de gràfics i àudio i la lògica per posar en pausa el joc.

```
public static MenuController menuController;

public int MainMenu = 0;
public GameObject[] menus;

public AudioMixer GeneralAudioMixer;
public Dropdown resolutionDropdown;
Resolution[] resolutions;

public static bool gameIsPaused = false;
public KeyCode pauseKey = KeyCode.Escape;
public GameObject PauseMenuUI;
```

Figura 6-46 Classe MenuController

- Start – Per inicialitzar algunes de les variables fem servir la funció d'start (vegeu Figura 6-47).

```
private void Start()
{
    UpdateResolutionDropdown();
    if (!menuController) menuController =
            this.gameObject.GetComponent<MenuController>();
}
```

Figura 6-47 Funció Start

- UpdateResolutionDropdown – S'encarrega d'inicialitzar les variables relacionades amb la resolució de la pantalla (vegeu Figura 6-48)

```
private void UpdateResolutionDropdown()
{
    if (!resolutionDropdown) return;
    resolutions = Screen.resolutions;
    resolutionDropdown.ClearOptions();

    List<string> options = new List<string>();

    int currentResolutionIndex = 0;
    for (int i = 0; i < resolutions.Length; i++)
    {
        string option = resolutions[i].width + " x "
            + resolutions[i].height;
        options.Add(option);
        if (resolutions[i].width == Screen
                .currentResolution.width
           && resolutions[i].height == Screen
                .currentResolution.height)
        {
            currentResolutionIndex = i;
        }
    }

    resolutionDropdown.AddOptions(options);
    resolutionDropdown.value = currentResolutionIndex;
    resolutionDropdown.RefreshShownValue();
}
```

Figura 6-48 Funció UpdateResolutionDropdown

- Update – Per obrir el menú de pausa és necessari comprovar si s'ha premut la tecla que tenim assignada a obrir el menú, això es mira a la funció Update (vegeu Figura 6-49).

```
private void Update()
{
    if (Input.GetKeyDown(pauseKey))
    {
        ActivatePauseMenu();
    }
}
```

Figura 6-49 Funció Update

- ActivatePauseMenu – Canvia l'estat del menú depenent de la variable gameIsPaused (vegeu Funció 6-50).

```
private void ActivatePauseMenu() {
    if (gameIsPaused)
    {
        ResumeGame();
    }
    else
    {
        PauseGame();
    }
}
```

Figura 6-50 Funció UpdateResolutionDropdown

- ResumeGame / PauseGame – Com el nom indica s'encarreguen de posar en pausa o continuar el joc (vegeu Funció 6-51).

```
public void ResumeGame()
{
    PauseMenuUI.SetActive(false);
    Time.timeScale = 1f;
    gameIsPaused = false;
}
1 referencia
public void PauseGame()
{
    ShowUI(PauseMenuUI);
    Time.timeScale = 0f;
    gameIsPaused = true;
}
```

Figura 6-51 Funcions ResumeGame / PauseGame

- ShowUI – Amaga tots els menús excepte el que li diguem (vegeu Figura 6-52).

```
public void ShowUI(GameObject toShow)
{
    GameObject[] a = { toShow };
    ChangeUI.Activate(menus, a);
}
```

Figura 6-52 Funció ShowUI

- GoToMainMenu – Ens fa tornar a l'escena de menú principal (vegeu Figura 6-53)

```
public void GoToMainMenu()
{
    GoToScene(MainMenu);
}
```

Figura 6-53 Funció GoToMainMenu

- GoToScene – Ens porta a l'escena que li diguem (vegeu Figura 6-54)

```
public void GoToScene(int i)
{
    SceneManager.LoadScene(i);
}
0 referencias
public void GoToScene(string name)
{
    SceneManager.LoadScene(SceneManager.GetSceneByName(name).buildIndex);
}
```

Figura 6-54 Funió GoToScene

- NextScene – Ens porta a la següent escena (vegeu Figura 6-55)

```
public void NextScene()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene()
        .buildIndex + 1);
}
```

Figura 6-55 Funció NextScene

- ResetScene – Ens porta a l'escena a la qual estem amb els valors inicials (vegeu Figura 6-56)

```
public void ResetScene()
{
    SceneManager.LoadScene(SceneManager.GetActiveScene()
        .buildIndex);
}
```

Figura 6-56 Funció ResetScene

- QuitGame – Ens fa fora del joc (vegeu Figura 6-57).

```
public void QuitGame()
{
    Application.Quit();
}
```

Figura 6-57 Funció QuitGame

- SetVolume – Canvia el volum al valor donat (vegeu Figura 6-58).

```
public void setVolume(float volume)
{
    GeneralAudioMixer.SetFloat("volume", volume);
}
```

Figura 6-58 Funció SetVolume

- SetQuality – Canvia el nivell de qualitat de gràfics del joc (vegeu Figura 6-59).

```
public void SetQuality(int qualityIndex)
{
    QualitySettings.SetQualityLevel(qualityIndex);
}
```

Figura 6-59 Funció SetQuality

- SetFullscreen – Canvia la pantalla de joc a pantalla completa (vegeu Figura 6-60)

```
public void SetFullscreen(bool isFullScreen)
{
    Screen.fullScreen = isFullScreen;
    Debug.Log("isFullScreen = " + isFullScreen);
}
```

Figura 6-60 Funció SetFullscreen

- SetResolution – Canvia la resolució de la pantalla (vegeu Figura 6-61)

```
public void SetResolution(int resolutionIndex)
{
    Resolution resolution = resolutions[resolutionIndex];
    Screen.SetResolution(
        resolution.width,
        resolution.height,
        Screen.fullScreen);
}
```

Figura 6-61 Funció SetResolution

ChangeUI

La classe ChangeUI (vegeu Figura 6-62) ens ajuda a tenir més flexibilitat en canviar entre interfícies i canviar d'una interfície a una altra sense la necessitat de tocar codi. Aquesta classe té dues variables que són els objectes que ha d'amagar i els que ha de mostrar.

```
public GameObject[] toHide;
public GameObject[] toShow;
```

Figura 6-62 Classe ChangeUI

Com a funcions simplement té la funció Activate (vegeu Figura 6-63) que amaga els objectes que hagi d'amagar i mostra els que s'han de mostrar. Activate està programada de forma que altres classes puguin fer servir una versió d'aquesta funció on es passa per paràmetre directament els objectes a ensenyar o amagar.

```
public void Activate()
{
    Activate(toHide, toShow);
}
2 referencias
public static void Activate(GameObject[] toHide, GameObject[] toShow)
{
    foreach(GameObject go in toHide)
    {
        go?.SetActive(false);
    }
    foreach (GameObject go in toShow)
    {
        go?.SetActive(true);
    }
}
```

Figura 6-63 Funció Activate

6.4 Lògica interna de l'inventari

Perquè el sistema d'inventari funcioni, fa falta guardar informació sobre els ítems que s'hi guarden, la quantitat d'ítems per ranura, en quina posició de l'inventari estan i tota la lògica per agafar ítems, treure'ls, intercanviar-los i altres funcions per saber l'estat de l'inventari.

InventoryÍtemData

InventoryÍtemData és la unitat més petita dins l'inventari, simplement guarda la informació de l'ítem que es guarda a l'inventari com la màxima quantitat d'ítems en una ranura o la imatge que representa l'ítem (vegeu Figura 6-64).

```
public class InventoryItemData : ScriptableObject
{
    #region Variables
    public ItemData ItemData;
    public Sprite ICON;
    public int STACK_SIZE;
    #endregion
}
```

Figura 6-64 InventoryÍtemData

InventorySlot

InventorySlot és el primer nivell que té mètodes, s'encarrega de guardar l'ítem i quantitat que hi ha en una ranura.

Els mètodes principals són:

- Crear/Inicialitzar InventorySlot.
- Posar Ítem.
- Retirar Ítem.

Crear/Inicialitzar

Inicialitza la ranura amb l'Ítem i la quantitat donats.

```
public InventorySlot(InventoryItemData data, int amount)
{
    itemData = data;
    slotAmount = amount;
}
4 referencias
public InventorySlot()
{
    ClearSlot();
}
```

Figura 6-65 Funció Inventory Slot

Posar Ítem

La funció PutÍtem (vegeu Figura 6-66), posa els Ítems donats a la ranura i retorna el sobrant, en cas que l'ítem sigui diferent al que ja hi ha els intercanvia.

```
public InventorySlot PutItem(InventorySlot invSlot)
{
    return PutItem(invSlot.itemData, invSlot.slotAmount);
}
3 referencias
public InventorySlot PutItem(InventoryItemData i_itemData, int i_slotAmount)
{
    InventorySlot extraItems = new InventorySlot();
    if (itemData == i_itemData)
    {
        extraItems.FillSlot(itemData, AddToSlot(i_slotAmount));
    }
    else
    {
        extraItems.FillSlot(itemData, slotAmount);
        FillSlot(i_itemData, i_slotAmount);
    }
    extraItems.UpdateSlot();
    return extraItems;
}
```

Figura 6-66 Funció PutÍtem

La funció FillSlot (vegeu Figura 6-67) simplement s'encarrega de buidar la ranura si la quantitat és zero i igualar el tipus d'ítems.

```
private void FillSlot(InventoryItemData data, int amount)
{
    if (amount == 0) ClearSlot();
    itemData = data;
    AddToSlot(amount);
}
```

Figura 6-67 Funció FillSlot

Finalment AddToSlot (vegeu Figura 6-68) fa els càlculs que facin falta per emplenar l'slot i retornar el sobrant, si és necessari.

```
private int AddToSlot(int amount)
{
    int extra_amount = 0;
    if(RoomLeftInSlot < amount)
    {
        extra_amount = amount - RoomLeftInSlot;
        slotAmount += RoomLeftInSlot;
    }
    else
    {
        slotAmount += amount;
    }
    return extra_amount;
}
```

Figura 6-68 Funció AddToSlot

Retirar Ítem

S'encarrega de treure la quantitat d'elements demanada, i retorna la quantitat que realment s'hagi tret, ja que podria ser que se'n demanés més de la que hi ha (vegeu Figura 6-69)-

```
public InventorySlot RemoveItem(int amount) // Returns removed items
{
    return new InventorySlot(itemData, RemovefromSlot(amount));
}
1 referencia
private int RemovefromSlot(int amount) // Returns amount really taken
{
    if (amount >= slotAmount)
    {
        amount = slotAmount;
        ClearSlot();
    }
    else
    {
        slotAmount -= amount;
    }
    return amount;
}
```

Figura 6-69 Funció Remove Ítem i RemovefromSlot

Altres Funcions

Hi ha mètodes no tan importants que s'encarreguen de facilitar-nos la feina per no haver de repetir algunes operacions.

- UpdateSlot – En cas que la ranura estigui massa plena, treu el sobrant, i si està buida borra l'ítem (vegeu Figura 6-70)

```
public void UpdateSlot()
{
    if (itemData != null && RoomLeftInSlot < 0) slotAmount += RoomLeftInSlot;
    if (slotAmount <= 0) ClearSlot();
}
```

Figura 6-70 Funció UpdateSlot

- ClearSlot – Deixa la quantitat a zero i l'Ítem a nul (vegeu Figura 6-71).

```
public void ClearSlot()
{
    itemData = null;
    slotAmount = 0;
}
```

Figura 6-71 Funció ClearSlot

- RoomLeftInSlot – Retorna la quantitat que falta per emplenar la ranura (vegeu Figura 6-72).

```
public int RoomLeftInSlot => itemData?
            itemData.STACK_SIZE - slotAmount : 0 - slotAmount;
```

Figura 6-72 Funció RoomLeftInSlot

- SlotEmpty – Retorna cert si la ranura esta buida (vegeu Figura 6-73)

```
public bool SlotEmpty => itemData == null || slotAmount <= 0;
```

Figura 6-73 Figura SlotEmpty

Inventory

Inventory és la classe més important, ja que és la que conté les dades de l'inventari i els mètodes per posar-hi ítems.

Les variables de l'inventari són una llista de InventarySlots que són els que tindran la informació del que hi ha a l'inventari i un UnityAction que servirà en un futur per a actualitzar la part de la interfície (vegeu Figura 6-74)

```
[SerializeField] protected List<InventorySlot> inventorySlots;
public UnityAction OnInventorySlotChanged;
```

Figura 6-74 Variables de l'Inventari

Els mètodes més importants són:

- Crear/Inicialitzar Inventory.
- Posar Ítem.
- Retirar Ítem.
- Altres mètodes per saber l'estat de l'inventari

Crear/Inicialitzar

Inicialitza l'inventari depenent de la mida donada i crea els InventorySlots necessaris per a inicialitzar-lo (vegeu Figura 6-75).

```
public Inventory(int size)
{
    inventorySlots = new List<InventorySlot>(size);
    for (int i = 0; i < size; i++)
    {
        inventorySlots.Add(new InventorySlot());
    }
}
```

Figura 6-75 Creació Inventory

Posar Ítem

Posa els ítems donats a l'inventari de forma que primer ompli les ranures no plenes que tinguin el mateix ítem, i si no n'hi ha prou, de forma recursiva al primer lloc buit. Si no hi ha lloc, retorna el contingut sobrant.

Quan canvia una ranura activa l'acció d'Unity per a avisar a qui faci falta del canvi de l'inventari.

```
public InventorySlot AddToInventory(InventorySlot slotToAdd)
{
    return new InventorySlot (slotToAdd.ItemData,
            AddToInventory(slotToAdd.ItemData, slotToAdd.SlotAmount));
}
2 referencias
public int AddToInventory(InventoryItemData itemToAdd, int amountToAdd)
{
    if (WhereToAdd(itemToAdd, out InventorySlot invSlot))
    {
        InventorySlot extra = invSlot.PutItem(itemToAdd, amountToAdd);
        OnInventorySlotChanged?.Invoke();
        if (!extra.SlotEmpty) return AddToInventory(itemToAdd, extra.SlotAmount);

    }
    return amountToAdd;
}
```

Figura 6-76 Funció AddToInventory

Retirar Ítem

S'encarrega de buidar una quantitat d'un ítem de l'inventari. En cas que no es pugui retirar el nombre necessari d'ítems, retorna la quantitat que no s'ha pogut treure (vegeu Figura 6-77).

```
public InventorySlot RemoveFromInventory(InventorySlot slotToRemove)
{
    return new InventorySlot(slotToRemove.ItemData,
            RemoveFromInventory(slotToRemove.ItemData, slotToRemove.SlotAmount));
}
2 referencias
public int RemoveFromInventory(InventoryItemData itemToRemove, int amountToRemove)
{
    if(ContainsItem(itemToRemove, out InventorySlot invSlot))
    {
        InventorySlot itemLeft = invSlot.RemoveItem(amountToRemove);
        OnInventorySlotChanged?.Invoke();
        if (itemLeft.SlotAmount != 0)
        {
            return RemoveFromInventory(itemLeft.ItemData, itemLeft.SlotAmount);
        }
        else return 0;
    }
    return amountToRemove;
}
```

Figura 6-77 Funció RemoveFromInventory

Mètodes per saber l'estat de l'inventari

Per fer servir els mètodes anteriors és necessari tenir informació sobre l'inventari, com pot ser la posició on es pot començar a afegir o retirar ítems, la quantitat d'ítems d'un tipus, si hi ha ranures buides, entre d'altres. Aquestes són les funcions que se n'encarreguen.

- AmountOf – Retorna la quantitat total d'un ítem a l'inventari (vegeu Figura 6-78).

```
public int AmountOf(InventoryItemData item)
{
    return InventorySlots.Where(Slot => Slot.ItemData == item)
        .Sum(Slot => Slot.SlotAmount);
}
```

Figura 6-78 Funció AmountOf

- CanBeRemoved – Retorna cert si hi ha prou quantitat d'un ítem a l'inventari (vegeu Figura 6-79)

```
public bool CanBeRemoved(InventoryItemData itemToRemove, int amountToRemove)
{
    return AmountOf(itemToRemove) >= amountToRemove;
}
```

Figura 6-79 Funció CanBeRemoved

- ContainsÍtemAndSpace – Retorna cert en el cas que contingui un ítem i la ranura a la qual està no estigui plena. També retorna la ranura a la que està (vegeu Figura 6-80)

```
public bool ContainsItemAndSpace(InventoryItemData item, out InventorySlot invSlot)
{
    invSlot = InventorySlots.FirstOrDefault(Slot => Slot.ItemData == item
          && Slot.RoomLeftInSlot > 0);
    return invSlot != null;
}
```

Figura 6-80 Figura ContainsÍtemAndSpace

- ContainsÍtem – Retorna cert si conté l'ítem en qüestió (vegeu Figura 8-81).

```
public bool ContainsItem(InventoryItemData item, out InventorySlot invSlot)
{
    invSlot = InventorySlots.FirstOrDefault(Slot => Slot.ItemData == item);
    return invSlot != null;
}
```

Figura 6-81 Funció ContainsÍtem

- HasFreeSlot – Retorna cert i la ranura en qüestió si hi ha una ranura buida (vegeu Figura 8-82).

```
public bool HasFreeSlot(out InventorySlot freeSlot)
{
    freeSlot = InventorySlots.FirstOrDefault(Slot => Slot.ItemData == null);
    return freeSlot != null;
}
```

Figura 6-82 Funció HasFreeSlot

- WhereToAdd – Retorna cert si hi ha algun lloc on es pot afegir l'ítem en qüestió, també retorna la ranura en qüestió (vegeu Figura 6-83).

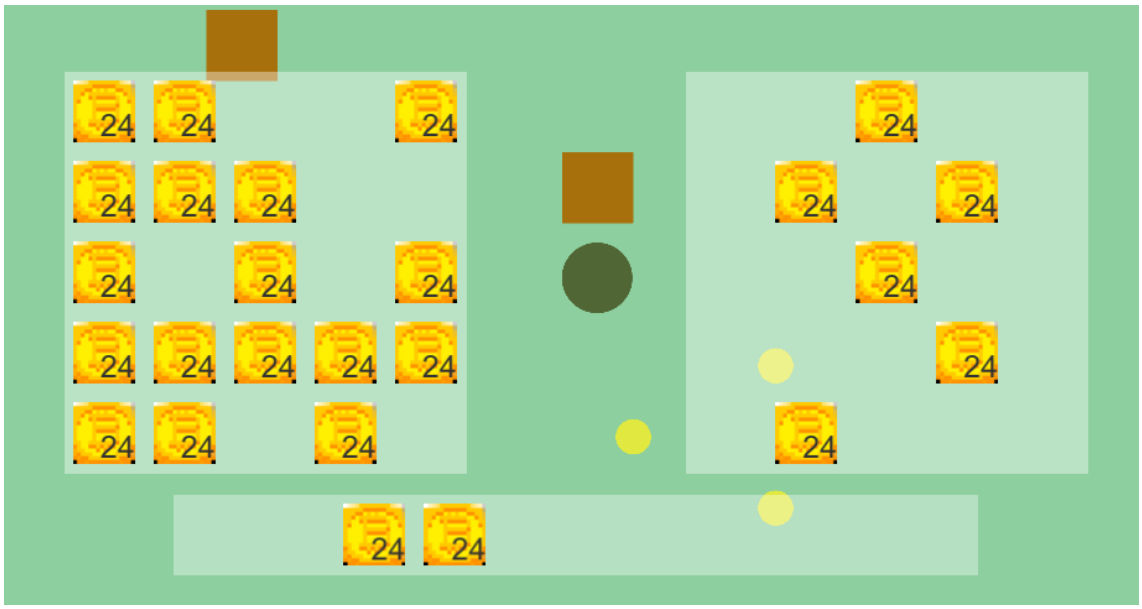
```
public bool WhereToAdd(InventoryItemData item, out InventorySlot invSlot)
{
    if(!ContainsItemAndSpace(item, out invSlot))
    {
        return HasFreeSlot(out invSlot);
    }
    return true;
}
```

Figura 6-83 Figura WhereToAdd

Interfície d'usuari de l'inventari

Un cop fet l'inventari i la seva lògica, ara farà falta que es pugui veure, per fer-ho necessitarem dues grans classes: la interfície de la ranura i la interfície de l'inventari.

A més a més, farem servir dues classes més perquè, en el cas que es vulgui, es pugui mostrar per pantalla diferents inventaris que seleccionem, com per exemple si volguéssim posar cofres en el joc, aquestes servirien per mostrar només un cofre en comptes de tots els del joc (Il·lustració 6-1).



Il·lustració 6-1 Exemple de múltiples inventaris

UI_InventorySlot

UI_InventorySlot fa la funció de guardar l'InventorySlot que li pertoca, així com actualitzar la informació de la imatge i el text que té assignats. També té assignat l'script ClickableObject, això permetrà que s'hi pugui interactuar encara que no sigui un botó.

Aquestes són les variables (Figura 6-84).

```
[SerializeField] protected Image itemSprite;
[SerializeField] protected Text itemCount;
[SerializeField] protected InventorySlot inventorySlot;
private ClickableObject cObj;
2 referencias
public InventoryDisplay ParentDisplay { get; private set; }
3 referencias
public InventorySlot InventorySlot => inventorySlot;
```

Figura 6-84 Variables de la funció WhereToAdd

Té quatre funcions:

- Inicialitzar.
- Esborrar la ranura.
- Actualitzar la interfície de la ranura.
- Notificar l'InventoryDisplay en cas que s'hi interactuï.

Inicialitzar

Actualitza el contingut de la interfície i busca el ClickableObject i l'InventoryDisplay que tindrà el pare (vegeu Figura 6-85).

```
private void Start()
{
    UpdateUISlot();

    cObj = GetComponent<ClickableObject>();
    cObj?.click.AddListener(OnUISlotClick);

    ParentDisplay = transform.parent.GetComponent<InventoryDisplay>();
}
```

Figura 6-85 Funció Inicialitzar

Esborrar

Elimina la informació de la ranura així com la imatge i el text de la interfície (vegeu Figura 6-86).

```
public void ClearSlot()
{
    inventorySlot?.ClearSlot();
    itemSprite.sprite = null;
    itemSprite.color = Color.clear;
    itemCount.text = "";
}
```

Figura 6-86 Funció Esborrar

Actualitzar

Depenent del contingut de la ranura canviarà la imatge i el text associats (vegeu Figura 6-87).

```
public void UpdateUISlot(InventorySlot slot)
{
    if (slot.ItemData != null)
    {
        itemSprite.sprite = slot.ItemData.ICON;
        itemSprite.color = Color.white;
        if (slot.SlotAmount > 1)
        {
            itemCount.text = slot.SlotAmount.ToString();
        }
        else
        {
            itemCount.text = "";
        }
    }
    else
    {
        ClearSlot();
    }
}
```

Figura 6-87 Funció Actualitzar

Notificar l'InventoryDisplay

En cas que s'interactuï amb la ranura envia la informació a l'InventoryDisplay (vegeu Figura 6-88).

```
public void OnUISlotClick(int button)
{
    ParentDisplay?.SlotClicked(this, button);
}
```

Figura 6-88 Funció OnUISlotClick

MouseÍtemData

MouseÍtemData hereda de UI_InventorySlot i s'encarrega de guardar l'ítems que pertany al ratolí, en cas de que es faci clic a les ranures de l'inventari. La difèrencia principal entre aquesta classe i la d'UI_InventorySlot és que aquesta es mou a on estigui el ratolí quan té algun ítem. Els mètodes que implementa són els següents.

Moure a la posició del ratolí

Mou la ranura a on estigui el ratolí, en cas de no estar a sobre cap element d'interfície d'usuari i fer clic esquerre, buida el contingut que hi havia (vegeu Figura 6-89).

```
private void Update()
{
    if (inventorySlot.ItemData != null)
    {
        transform.position = Input.mousePosition;
        if (LeftClick && !IsPointerOverUIObject())
        {
            DropSlot();
        }
    }
}
```

Figura 6-89 Funció Update de MouseÍtemData

Deixar l'objecte

Buida el contingut de l'slot (vegeu Figura 6-90).

```
private void DropSlot()
{
    ClearSlot();
}
```

Figura 6-90 Funció DropSlot

Hi ha una interfície a on apunta el ratolí?

Retorna cert si hi ha una interfície d'usuari a la posició del ratolí (vegeu Figura 6-91).

```
public bool IsPointerOverUIObject()
{
    PointerEventData eventDataCurrentPosition =
        new PointerEventData(EventSystem.current);
    eventDataCurrentPosition.position =
        new Vector2(Input.mousePosition.x, Input.mousePosition.y);
    List<RaycastResult> results =
        new List<RaycastResult>();
    EventSystem.current.RaycastAll(eventDataCurrentPosition, results);
    return results.Count > 0;
}
```

Figura 6-91 Funció IsPointerOverUIObject

InventoryDisplay

InventoryDisplay: el paper principal és guardar tots els UI_InventorySlot que pertanyen a un inventari i actualitzar els valors tant de l'inventari com de l'UI_InventorySlot quan hi hagi cap canvi. També s'encarrega de la interacció entre el ratolí i l'inventari.

Les seves variables són les següents, les més importants són l'inventari i el diccionari que relaciona cada ranura amb la interfície (vegeu Figura 6-92).

```
[SerializeField] protected MouseItemData mouseInventoryItem;
[SerializeField] protected Inventory inventory;
[SerializeField] protected Vector3 position = new Vector3(0, 0, 0);
protected Dictionary<InventorySlot, UI_InventorySlot> slotDictionary;
```

Figura 6-92 Variables InventoryDisplay

Els mètodes més rellevants que implementa són:

- Actualitzar totes les interfícies de les ranures.
- Assignar a cada interfície el seu slot.
- Reaccionar als clics.

També implementa un mètode abstracte que serveix per a les classes que heretin d'aquesta, que fa que creïn les interfícies de les ranures (vegeu Figura 6-93).

```
public abstract void MakeInventory();
```

Figura 6-93 Mètode abstracte MakeInventory

Actualitzar ranures

Actualitza l'aspecte de la interfície de tots els slots del diccionari (vegeu Figura 6-94).

```
protected void UpdateSlots()
{
    foreach(KeyValuePair< InventorySlot, UI_InventorySlot > kis in slotDictionary)
    {
        kis.Value.UpdateUISlot();
    }
}
```

Figura 6-94 Funció UpdateSlots

Assignar ranures

A partir d'un Array d'interfícies de ranures, crea el diccionari i l'inventari i els actualitza amb els valors donats. També assigna la funció d'actualitzar ranures amb l'acció que és crida quan l'inventari canvia (vegeu Figura 6-95).

```
protected void AssignSlot(UI_InventorySlot[] slots)
{
    slotDictionary = new Dictionary<InventorySlot, UI_InventorySlot>();
    inventory = new Inventory(slots.Length);
    inventory.OnInventorySlotChanged += UpdateSlots;
    for (int i = 0; i < inventory.InventorySize; i++)
    {
        slotDictionary.Add(inventory.InventorySlots[i], slots[i]);
        inventory.InventorySlots[i] = slots[i].InventorySlot;
        slots[i].UpdateUISlot();
    }
}
```

Figura 6-95 Funció AssignSlot

Reaccionar als clics

Aquí hi ha tota la lògica relacionada amb els clics que rep l'inventari, depenent del que hi hagi en el ratolí i el que hi ha a la ranura clicada es fan accions diferents (Figura 6-96).

```
public void SlotClicked(UI_InventorySlot clickedSlot, int button)
{
    if (!InventoryController.interactuable) return;

    InventorySlot mouse = mouseInventoryItem.InventorySlot;
    InventorySlot clicked = clickedSlot.InventorySlot;

    if (button == 0) // Left click
    {
        Interchange(clicked, mouse);
    }
    else if(button == 1)// Right click
    {
        if (mouse.ItemData == null && clicked.ItemData != null)
        {
            GrabHalf(clicked, mouse);
        }
        else if (mouse.ItemData != null &&
                (clicked.ItemData == null || (
                clicked.ItemData == mouse.ItemData &&
                clicked.RoomLeftInSlot > 0)))
        {
            DropOne(clicked, mouse);
        }
    }

    clickedSlot.UpdateUISlot();
    mouseInventoryItem.UpdateUISlot();
}
```

Figura 6-96 Funció SlotClicked

Per a les accions d'intercanviar ítems, agafar la meitat dels ítems i deixar-ne un, hi ha les següents funcions.

Intercanviar

S'activa quan el clic correspon al botó esquerra del ratolí. Canvia el contingut entre el ratolí i la ranura clicada (vegeu Figura 6-97).

```
private void Interchange(InventorySlot clicked, InventorySlot mouse)
{
    InventorySlot aux = clicked.PutItem(mouse);
    mouse.ClearSlot();
    mouse.PutItem(aux);
}
```

Figura 6-97 Funció Intercanviar

Agafar la meitat

S'activa quan es fa clic dret i el ratolí està buit. Agafa la meitat d'ítems de l'slot (vegeu Figura 6-98).

```
private void GrabHalf(InventorySlot clicked, InventorySlot mouse)
{
    int half = (clicked.SlotAmount + 1) / 2;
    mouse.PutItem(clicked.RemoveItem(half));
}
```

Figura 6-98 Funció GrabHalf

Deixar-ne un

Quan es fa clic dret, l'ítem del ratolí correspon a l'ítem clicat i hi ha espai suficient o la ranura clicada està buida, es deixa un ítem del ratolí i es posa a la ranura (vegeu Figura 6-99).

```
private void DropOne(InventorySlot clicked, InventorySlot mouse)
{
    clicked.PutItem(mouse.ItemData, 1);
    mouse.RemoveItem(1);
}
```

Figura 6-99 Funció DropOne

RectangleInventoryDisplay

RectangleInventoryDisplay hereta d'inventory display, la funció principal és crear un conjunt de ranures de forma que creïn un rectangle, per fer-ho fa servir les variables d'alçada, amplada, mida de la ranura i l'espai entre ranures així com un prefab d'una ranura (vegeu Figura 6-100).

```
public int width=5;
public int height=5;
public int padding = 10;
public int cellSize = 50;
public GameObject SlotUIPrefab;
private GridLayoutGroup layout;
```

Figura 6-100 Funció RectangleInventoryDisplay

Com a mètodes simplement implementa el MakeInventory de InventoryDisplay. Que crea les ranures i l'espai on posar-les (vegeu Figura 6-101).

```
public override void MakeInventory()
{
    SetUpObject();
    SetLayout();

    if (this.gameObject.GetComponent<RectTransform>())
    {
        this.gameObject.GetComponent<RectTransform>().sizeDelta =
            new Vector2(width * (cellSize + padding), height * (cellSize + padding));
        if (this.gameObject.GetComponent<RectTransform>().localPosition ==
                new Vector3(0, 0, 0) && position != new Vector3(0, 0, 0))
            this.gameObject.GetComponent<RectTransform>().localPosition = position;
    }
    List<UI_InventorySlot> slots = new List<UI_InventorySlot>() ;
    for (int i = 0; i < width * height; i++)
    {
        GameObject aux = Instantiate<GameObject>(SlotUIPrefab);
        aux.transform.SetParent(this.transform);
        slots.Add(aux.GetComponent<UI_InventorySlot>());
    }
    AssignSlot(slots.ToArray());
}
```

Figura 6-101 Funció MakeInventory

També té la funció SetUpObject (Figura 6-102) que inicialitza el GridLayoutGroup on es posen les ranures i en cas de no ser-hi, el crea.

```
private void SetUpObject()
{
    layout = this.gameObject.GetComponent<GridLayoutGroup>();
    if (!layout)
    {
        layout = gameObject.AddComponent(typeof(GridLayoutGroup)) as GridLayoutGroup;
    }
}
```

Figura 6-102 Funció SetUpObject

Finlament la funció SetLayout s'encarrega d'inicialitzar el GridLayoutGroup amb els parametres donats.

```
private void SetLayout()
{
    if (!layout)
    {
        Debug.Log("No Layout");
        return;
    }
    layout.padding.left = padding/2;
    layout.padding.right = padding / 2;
    layout.padding.top = padding / 2;
    layout.padding.bottom = padding / 2;
    layout.spacing = new Vector2(padding, padding);
    layout.cellSize = new Vector2(cellSize, cellSize);
}
```

Figura 6-103 Funció SetLayout

GenericInventoryDisplay

GenericInventoryDisplay hereta d'inventory display, en comptes de crear ranures com fa RectangleInventoryDisplay, aquesta classe rep les ranures com a paràmetre. D'aquesta forma es té molta més flexibilitat a l'hora de crear l'inventari, ja que pots posar les ranures a la posició que vulguis.

Només té una variable que és l'Array de ranures i la implementació del mètode MakeInventory (vegeu Figures 6-104 i 6-105).

```
[SerializeField] protected UI_InventorySlot[] _slots;
```

Figura 6-104 Variable GenericInventoryDisplay

```
public override void MakeInventory()
{
    AssignSlot(_slots);
}
```

Figura 6-105 Funció MakeInventory

InventoryShower

InventoryShower té la funció de mostrar l'inventari de l'objecte que se li passa per paràmetre i posar-lo com a fill, d'aquesta forma podem fer que es mostri un inventari concret a la posició de l'objecte que tingui aquest script.

Com a variables té l'objecte que té l'inventari i una variable privada que guarda el pare d'aquest objecte per tornar-lo al seu lloc quan s'hagi de deixar de mostrar (vegeu Figura 6-106).

```
public GameObject inventoryObject;
private GameObject auxParent;
```

Figura 6-106 Variables de InventoryShower

Les funcions són mostrar l'inventari i deixar de mostrar l'inventari.

Mostrar l'inventari

Posa l'inventari com a fill de l'objecte que té l'script de InventoryShower i el mostra (vegeu Figura 6-107).

```
public void ShowInventory()
{
    if (!inventoryObject) return;
    inventoryObject.SetActive(true);
    auxParent = inventoryObject.transform.parent.gameObject;
    inventoryObject.transform.SetParent(this.transform);
    inventoryObject.transform.localPosition = Vector3.zero;
}
```

Figura 6-107 Funció ShowInventory

Deixar de mostrar l'inventari

Posa l'inventari com a fill de l'objecte que el tenia com a fill originalment i el deixa de mostrar (vegeu Funció 6-108).

```
public void RemoveInventory()
{
    if (!inventoryObject) return;
    inventoryObject.transform.SetParent(auxParent.transform);
    inventoryObject.SetActive(false);
}
```

Figura 6-108 Funció RemoveInventory

InventoryController

Finalment tenim l'InventoryController la funció és mostrar o deixar de mostrar els inventaris que tenen assignats els InventoryShower que li donguem a les variables. També fa que els inventaris siguin interactuables i atura el joc. Com a variables li passem els objectes que tenen InventoryShower que vulguem mostrar (vegeu Figura 6-109).

```
[SerializeField] private GameObject PersonalInventory;
[SerializeField] public GameObject OtherInventory;
```

Figura 6-109 Variables InventoryController

Finalment, els mostrem o amaguem amb la funció ShowHideInventory (vegeu Figura 6-110).

```
public void ShowHideInventory()
{
    interactuable = !interactuable;
    Time.timeScale = interactuable?0f:1f;
    if (interactuable)
        PersonalInventory.gameObject.
            GetComponent<InventoryShower>()?.ShowInventory();
    else
        PersonalInventory.gameObject.
            GetComponent<InventoryShower>()?.RemoveInventory();
    if (OtherInventory)
    {
        if (interactuable)
            OtherInventory.gameObject.
                GetComponent<InventoryShower>()?.ShowInventory();
        else
            OtherInventory.gameObject.
                GetComponent<InventoryShower>()?.RemoveInventory();

    }

}
```

Figura 6-110 Funció ShowHideInventory

Cofres

Ara ja tenim tot el que està relacionat amb la part d'interfície dels inventaris, el que falta són els objectes que tindran aquests inventaris i com obrir-los. Aquí entren els cofres, que seran els objectes del món que tindran un inventari, i els obridors de cofres, que seran els objectes que tindran la capacitat d'obrir cofres, per exemple el personatge principal (Il·lustració 6-1).

Chest

Chest és l'script que li assignem a l'objecte que té un inventari. Simplement serveix per deixar-nos veure que l'objecte amb el que tractem és un cofre i com a variable ens dona l'objecte que té l'InventoryDisplay (vegeu Figura 6-111).

```
public class Chest : MonoBehaviour
{
    [SerializeField] public GameObject inventory;
}
```

Figura 6-111 Script Chest

ChestOpener

ChestOpener és l'script que li posarem a l'objecte que obri els cofres. Quan un objecte cofre estigui a una certa distància de l'obridor de cofres i premem la tecla assignada a obrir cofres l'inventari del cofre que estigui a prop s'enviarà a l'InventoryShower de l'InventoryController i s'activarà la funció ShowHideInventory.

Com a variables tenim la tecla a prémer per activar la detecció de cofres i que es mostri l'inventari, el Detector que ha de trobar el cofre, la TagList amb la que hem marcat l'objecte cofre i l'InventoryController per activar la funció de mostrar inventari (vegeu Figura 6-112).

```
[SerializeField] protected KeyCode interactKey;
[SerializeField] protected Detector detector;
[SerializeField] protected TagList chestTag;
[SerializeField] protected InventoryController inventoryController;
```

Figura 6-112 Variables funció ChestOpener

La funció per activar l'inventari, un cop es premi la tecla en qüestió, és la següent (vegeu Figura 6-113).

```
void Update()
{
    if (Input.GetKeyDown(interactKey))
    {
        GameObject chest = detector.DetectTag(chestTag)?.gameObject;
        if (inventoryController.OtherInventory?.
            GetComponent<InventoryShower>())
        {
            inventoryController.OtherInventory
                .GetComponent<InventoryShower>()
                .inventoryObject = chest?.GetComponent<Chest>()?.inventory;
        }
        inventoryController.ShowHideInventory();
    }
}
```

Figura 6-113 Update de ChestOpener

Collectables

Aquesta és l'última part relacionada amb l'inventari que veurem. Es tracten d'objectes que trobarem pel mapa que, quan hi colisionem, posaran l'ítem que tenen a l'inventari.

Collectable

Collectable és l'script que tindrà l'objecte que ha de ser recollit, aquest també haurà de tenir un collider2D en forma de trigger. Les variables que té són les següents (Figura 6-114).

```
public bool destroyOnCollected = true;
public InventorySlot inventorySlot = new InventorySlot();
```

Figura 6-114 Variables funció Collectable

La funció que s'encarrega de comprovar que l'objecte que ha activat el trigger és de tipus Collector i de posar l'ítem l'inventari d'aquest Collector és la següent (Figura 6-115).

```
private void OnTriggerEnter2D(Collider2D collision)
{
    if (collision.gameObject.GetComponent<Collector>() != null)
    {
        if (!inventorySlot.SlotEmpty)
            inventorySlot = collision
                .gameObject
                .GetComponent<Collector>()
                .inventory?
                .Inventory?
                .AddToInventory(inventorySlot);
        if(inventorySlot.SlotEmpty || destroyOnCollected)
            Destroy(this.gameObject);
    }

}
```

Figura 6-115 Funció OnTriggerEnter2D

Depenent de la variable destroyOnCollected l'objecte es destruirà o no en el cas de que no hi hagi prou espai a l'inventari i encara quedin ítems a l'objecte després d'haver estat recollit.

Collector

Collector és l'script que tindrà l'objecte encarregat de recollir Collectables. Aquest simplement té guardat l'inventari al qual anirà l'ítem agafat (Figura 6-116).

```
public class Collector : MonoBehaviour
{
    [SerializeField] public InventoryDisplay inventory;
}
```

Figura 6-116 Script Collector

6.5 Lògica interna del sistema d'atac

Per a aconseguir un sistema d'atacs que té en compte tant tipus d'atac, com tipus de defensa, com atacs que duren un cert període de temps, fa falta un sistema que tingui en compte totes aquestes variables i tingui una sèrie de funcions que facin possible el correcte funcionament dels atacs.

AttackTypeCombinations

En aquesta classe es guarda tota la informació relacionada amb el tipus de dany, conté les variables isGeneric, typeOfAttack i NOTTypeOfAttack (Figura 6-117).

```
[SerializeField] protected bool _isGeneric = true;
8 referencias
public bool isGeneric => _isGeneric;
[SerializeField] protected string[] _typeOfAttack;
6 referencias
public string[] typeOfAttack => _typeOfAttack;
[SerializeField] protected string[] _NOTtypeOfAttack;
4 referencias
public string[] NOTTypeOfAttack => _NOTtypeOfAttack;
```

Figura 6-117 Classe AttackTypeCombinations

També conté tota la lògica per veure si hi ha solapament entre dos AttackTypeCombinations. Aquesta lògica està separada en quatre parts, depenent de si els tipus de dany són genèrics amb excepcions o no.

En cas que els dos atacs siguin genèrics retornem cert (Figura 6-118.1).

```
public bool isOverlap(AttackTypeCombinations atc)
{
    if (isGeneric && atc.isGeneric)
    {
        return true;
    }
```

Figura 6-118.1 Funció isOverlap

Si cap dels dos és genèric retornem cert només si hi ha coincidència amb els tipus d'atac (Figura 6-118.2).

```
if (!isGeneric && !atc.isGeneric)
{
    foreach (string s in typeOfAttack)
    {
        foreach (string s2 in atc.typeOfAttack)
        {
            if (s == s2) return true;
        }
    }
    return false;
}
```

Figura 6-118.2 Funció isOverlap

Si només un d'ells és genèric contem quants dels atacs que pot fer el no genèric no pot fer el genèric, si aquest nombre és més petit que el nombre d'atacs que pot fer el no genèric, és a dir hi ha com a mínim un tipus d'atac del no genèric que el genèric no té a la llista d'atacs que no pot fer, llavors retornem cert (Figura 6-118.3 i Figura 6-118.4).

```
if (isGeneric && !atc.isGeneric)
{
    int nOfOverlap = 0;
    foreach (string s in NOTTypeOfAttack)
    {
        foreach (string s2 in atc.typeOfAttack)
        {
            if (s == s2) { nOfOverlap++; break; };
        }
    }
    return nOfOverlap < atc.typeOfAttack.Length;
}
```

Figura 6-118.3 Funció isOverlap

```
if (!isGeneric && atc.isGeneric)
{
    int nOfOverlap = 0;
    foreach (string s in typeOfAttack)
    {
        foreach (string s2 in atc.NOTTypeOfAttack)
        {
            if (s == s2) { nOfOverlap++; break; };
        }
    }
    return nOfOverlap < typeOfAttack.Length;
}
return true;
}
```

Figura 6-118.4 Funció isOverlap

Attack

En aquesta classe es guarda la informació necessària per a fer els atacs. Això inclou el tipus d'atac, el mal que fa i si és instantani o no (Figura 6-119).

```
[SerializeField] protected AttackTypeCombinations _type;
0 referencias
public AttackTypeCombinations Type => _type;
[SerializeField] protected float _damage = 1;
1 referencia
public float Damage => _damage;
[SerializeField] protected float _damagePercent = 0;
1 referencia
public float DamagePercent => _damagePercent;
[SerializeField] protected bool _isInstant = true;
1 referencia
public bool IsInstant => _isInstant;
[SerializeField] protected float _timeLength = 0;
1 referencia
public float TimeLength => _timeLength;
[SerializeField] protected float _attackFrequency = 0;
3 referencias
public float AttackFrequency => _attackFrequency;
```

Figura 6-119 variables Attack

També té tres versions de la funció de solapament del tipus de mal, per a tenir en compte els casos en els quals com a paràmetre rep: un atac, una defensa o un tipus d'atac (Figura 6-120).

```
public bool isOverlap(Attack atc)
{
    return _type.isOverlap(atc._type);
}
1 referencia
public bool isOverlap(Defense dfs)
{
    return _type.isOverlap(dfs.Type);
}
0 referencias
public bool isOverlap(AttackTypeCombinations atc)
{
    return _type.isOverlap(atc);
}
```

Figura 6-120 Funció isOverlap

Defense

En aquesta classe es guarda la informació que té cada peça de defensa: el tipus de dany i la quantitat de mal defensat (Figura 6-121).

```
[SerializeField] protected AttackTypeCombinations _type;
1 referencia
public AttackTypeCombinations Type => _type;
[SerializeField] protected float _protection = 1;
1 referencia
public float Protection => _protection;
[SerializeField] protected float _protectionPercent = 0;
1 referencia
public float ProtectionPercent => _protectionPercent;
```

Figura 6-121 Variables classe Defense

Attacable

En aquesta és la classe més complexa del sistema d'atac, ja que conté tota la lògica necessària per rebre mal i defensar-se d'aquest mal. Les variables no privades són: la vida total, la vida actual i les defenses que té (Figura 6-122).

```
[SerializeField] protected float _healthTotal;
0 referencias
public float HealthTotal => _healthTotal;
[SerializeField] protected float _health;
0 referencias
public float Health => _health;
[SerializeField] protected List<Defense> _defenses;
0 referencias
public List<Defense> Defenses => _defenses;
```

Figura 6-122 Variables classe Attacable

A més a més, també té una variable privada que guarda els atacs que fan mal en un període de temps, per fer-ho, té una llista amb l'atac i el temporitzador que fa servir aquest atac per a fer mal (Figura 6-123).

```
private List<Tuple<Attack, Timer>> ongoingAttacks =
    new List<Tuple<Attack, Timer>>();
```

Figura 6-123 Variable ongoing Attacks

Quan comença el joc la vida actual s'inicialitza a la vida màxima en el cas que sigui zero (Figura 6-124).

```
private void Start()
{
    if(_health == 0)_health = _healthTotal;
}
```

Figura 6-124 Funció Start Classe Attacable

Després tenim dues funcions que ens permeten posar i treure defenses (Figura 6-125).

```
public void AddDefense(Defense defense)
{
    _defenses.Add(defense);
}
0 referencias
public void RemoveDefense(Defense defense)
{
    _defenses.Remove(defense);
}
```

Figura 6-125 Funcions AddDefense/RemoveDefense

La següent funció és la de rebre atacs. S'encarrega de cridar la funció de rebre dany. En cas que l'atac rebut es repeteixi en el temps, també s'encarrega d'inicialitzar els temporitzadors necessaris perquè l'atac es repeteixi i per eliminar-lo un cop s'acabi la duració de l'atac (Figura 6-126).

```
public void ReciveAttack(Attack attack)
{
    if (attack.IsInstant) ReciveDamage(attack);
    else
    {
        RemoveAttack(attack);
        Timer t = new Timer(
            delegate { ReciveDamage(attack); },
            attack.AttackFrequency);
        t.loopN((int)(attack.TimeLength / attack.AttackFrequency));
        t.start();
        Tuple<Attack, Timer> to =
            new Tuple<Attack, Timer>(attack, t);

        ongoingAttacks.Add(to);

        Timer t2 = new Timer(
            delegate { RemoveAttack(attack); },
            attack.TimeLength);
        t2.start();
    }
}
```

Figura 6-126 Funció ReciveAttack

Junt amb la de rebre atacs, tenim la funció de treure atacs, aquesta serveix per eliminar els atacs de la llista d'atacs i parar els temporitzadors que tenen associats perquè no facin més mal (Figura 6-127).

```
public void RemoveAttack(Attack attack)
{
    foreach (Tuple<Attack, Timer> ta in ongoingAttacks
        .FindAll(
            ta =>
            ta.Item1 == attack
            && ta.Item2.getDuration() == attack.AttackFrequency))
    {
        ta.Item2.stop();
        ongoingAttacks.Remove(ta);
    }

}
```

Figura 6-127 Funció RemoveAttack

Per acabar, tenim la funció per a calcular el mal que fa cada atac. Per fer-ho, té en compte tant el mal directe, com el mal en percentatge de l'atac, com la defensa que tenim que el tipus de dany solapa amb el de l'atac rebut (Figura 6-128).

```
private void ReciveDamage(Attack attack)
{
    float defense = 0;
    float defensePercent = 0;

    foreach (Defense d in _defenses)
    {
        if (attack.isOverlap(d))
        {
            defense += d.Protection;
            defensePercent += d.ProtectionPercent;
        }
    }

    float totalDamage = attack.Damage +
        (attack.DamagePercent / 100) * _healthTotal;
    float actualDamage = (totalDamage - defense)
        * (1 - (0.01f * defensePercent));

    _health = _health - actualDamage;
    hasBeenHurt.Invoke(actualDamage);
    if (_health < 0) Death();
}
```

Figura 6-128 Funció ReciveDamage

Attacker

Aquesta classe s'encarrega d'atacar. Hi ha dues classes que hereten d'aquesta que són AttackOnTrigger i AttackOnDetector. Aquesta classe simplement té una variable on hi guarda l'atac, el UnityEvent que s'activa quan ataca i la funció doAttack que simplement ataca a l' Attackable que rep per paràmetre i activa el UnityEvent (Figura 6-129).

```
public class Attacker : MonoBehaviour
{
    [SerializeField] protected Attack _attack;
    2 referencias
    public Attack Attack => _attack;
    [SerializeField] protected UnityEvent hasAttacked;
    2 referencias
    protected void doAttack(Attackable attackable)
    {
        hasAttacked.Invoke();
        attackable.ReciveAttack(Attack);
    }
}
```

Figura 6-129 Classe Attacker

AttackOnDetector

Aquesta classe hereta d'Attaker, té una variable més que és el Detector i implementa una funció per atacar que ataca a tot el que detecta el Detector (Figura 6-130).

```
public class AttackOnDetector : Attacker
{
    [SerializeField] protected Detector _detector;
    0 referencias
    public Detector Detector => _detector;

    0 referencias
    public void doAttack()
    {
        Collider2D[] colliders = _detector.DetectAll();
        foreach(Collider2D c in colliders)
        {
            if (c.GetComponent<Attackable>())
            {
                doAttack(c.GetComponent<Attackable>());
            }
        }
    }
}
```

Figura 6-130 Funció AttackOnDetector

AttackOnTrigger

Aquesta classe hereta d'Attacker i té un booleà que determina si, en cas que el mal que fa és "no instantani", en cas de l'objecte atacat surti de l'àrea del trigger, també deixarà de rebre dany. També té dues funcions que s'activen quan un objecte entra o surt del trigger. En els dos casos primer mira si l'objecte pot ser atacat i després fa l'atac o el treu depenent de si l'objecte entra o surt de l'àrea (Figura 6-131).

```
public class AttackOnTrigger : Attacker
{
    [SerializeField] protected bool _stopOnExit = false;
    0 referencias
    public bool stopOnExit => _stopOnExit;
    Mensaje de Unity | 0 referencias
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.gameObject.GetComponent<Attackable>())
        {
            doAttack(collision.gameObject.GetComponent<Attackable>());
        }
    }
    Mensaje de Unity | 0 referencias
    private void OnTriggerExit2D(Collider2D collision)
    {
        if (_stopOnExit && collision.gameObject.GetComponent<Attackable>())
        {
            collision.gameObject
                .GetComponent<Attackable>()
                .RemoveAttack(Attack);
        }
    }
}
```

Figura 6-131 Classe AttackOnTrigger

6.6 Lògica interna del sistema de diàlegs

Internament el sistema de diàlegs té tres components principals, els paràgrafs, els diàlegs i el DialogueManager. També té un objecte DialogueBox que serveix per assignar un diàleg a un objecte.

Per posar un exemple de com s'utilitzarien, si volguéssim crear el diàleg d'un cartell primer crearíem el diàleg amb els seus paràgrafs com a ScriptableObject i l'assignaríem a un DialogueBox que tindrà el cartell. Després crearíem un objecte que s'encarregaria de contenir tots els elements per mostrar el diàleg on hi posaríem el DialogueManager. Finalment quan vulguem mostrar el diàleg del cartell cridaríem la funció "ShowDialogue" de DialogueBox.

Paragraph

Paragraph serveix per guardar tota la informació de cada part del diàleg. Guarda tant el text a mostrar les diferents opcions que té com a resposta (Figura 6-132).

```
public class Paragraph
{
    [SerializeField] public string id;
    [TextArea(4, 4)]
    [SerializeField] public string text;
    [Serializable]
    3 referencias
    public struct Option
    {
        [SerializeField] public string text;
        [SerializeField] public bool next;
        [SerializeField] public string goTo;
    }
    [SerializeField] public List<Option> options;
}
```

Figura 6-132 Classe paragraph

Dialogue

Dialogue conté un llistat de Paragraph i un punter, també conté la lògica de buscar el pròxim paràgraf o el paràgraf al qual es refereix una opció, així com el mètode per mostrar-lo al DialogueManager (Figura 6-133).

```
[SerializeField]public List<Paragraph> content;
private int pointer = 0;
```

Figura 6-133 Variables Dialogue

La funció GetParagraph retorna el paràgraf actual (Figura 6-134).

```
public Paragraph GetParagraph()
{
    if (pointer >= content.Count ||pointer < 0) return null;
    return content[pointer];
}
```

Figura 6-134 Funció GetParagraph del Dialogue

La funció NextParagraph (Figura 6-135) retorna el paràgraf que toca a continuació, en cas de donar una Option com a paràmetre fa la recerca del paràgraf depenent de les variables de la Option.

```
public Paragraph NextParagraph()
{
    pointer++;
    return GetParagraph();
}
1 referencia
public Paragraph NextParagraph(Paragraph.Option option)
{
    if (option.next)
        pointer++;
    else
        pointer = content.FindIndex(p => p.id == option.goTo);
    return GetParagraph();
}
```

Figura 6-135 Funció NextParagraph

Finalment, la funció Show (Figura 6-136), inicialitza el punter i fa que el DialogueManager mostri el diàleg.

```
public void Show()
{
    pointer = 0;
    DialogueManager.dialogueManager.ShowDialogue(this);
}
```

Figura 6-136 Funció Show

DialogueManager

Aquesta classe s'encarrega de passar la informació del Dialogue a la interfície designada. Com a paràmetres té, el Text i Canvas on es mostra el diàleg, els prefabs dels botons que farà servir per mostrar les opcions, la tecla per passar de paràgraf i el diàleg que ha de mostrar. També té un singleton perquè tots els diàlegs es puguin referir al que hauria de ser l'únic DialogueManager de l'escena (Figura 6-137).

```
public Text text;
public Canvas buttonsCanvas;
public Canvas globalCanvas;
private static DialogueManager _dialogueManager = null;
1 referencia
public static DialogueManager dialogueManager
    => _dialogueManager ?
        _dialogueManager
        : Resources.FindObjectsOfTypeAll<DialogueManager>()?[0];
public GameObject buttonPrefab;
public GameObject nextButtonPrefab;
public KeyCode nextParagraph = KeyCode.Space;
private Dialogue _dialogue;
```

Figura 6-137 Variables DialogueManager

En començar l'escena el DialogueManager desactiva el Canvas principal perquè no es vegi la interfície de diàleg (Figura 6-138).

```
private void Start()
{
    globalCanvas.gameObject.SetActive(false);
}
```

Figura 6-138 Funció Start DialogueManager

Per a fer que funcioni la tecla per passar pàgina fa falta que cada fotograma comprovem si s'ha premut la tecla, també comprovarem si en aquest moment s'està mostrant el diàleg i si el paràgraf actual té opcions o no (Figura 6-139).

```
private void Update()
{
    if(globalCanvas.gameObject.activeSelf
        && Input.GetKeyDown(nextParagraph)
        && _dialogue.GetParagraph().options.Count == 0)
    {
        ShowParagraph(_dialogue.NextParagraph());
    }
}
```

Figura 6-139 Funció Update DialogueManager

La funció ShowDialogue (Figura 6-140) s'encarrega de mostrar el diàleg que li arriba per paràmetre i pausa el joc.

```
public void ShowDialogue(Dialogue dialogue)
{
    Time.timeScale = 0;
    _dialogue = dialogue;
    globalCanvas.gameObject.SetActive(true);
    ShowParagraph(_dialogue.GetParagraph());
}
```

Figura 6-140 Funció ShowDialogue

La funció EndDialogue (Figura 6-141) reprèn el joc i amaga la interfície de diàleg.

```
public void EndDialogue()
{
    Time.timeScale = 1;
    globalCanvas.gameObject.SetActive(false);
}
```

Figura 6-141 Funció EndDialogue

ShowParagraph és la funció principal del DialogueManager s'encarrega de mostrar els paragrafs. Primer de tot destrueix tots els fills del Canvas de botons, després posa el text del paràgraf al text de la interfície, i finalment hi posa els botons (Figura 6-142)

```
public void ShowParagraph(Paragraph paragraph)
{
    if (paragraph == null)
    {
        EndDialogue();
        return;
    }
    foreach (Transform child in buttonsCanvas.transform)
    {
        GameObject.Destroy(child.gameObject);
    }
    text.text = paragraph.text;
    if (!TestButton(buttonPrefab))
    {
        Debug.LogError("buttonPrefab is not a Button");
        return;
    }
    if (paragraph.options.Count == 0)
    {
        AddButton(nextButtonPrefab);
    }
    foreach(Paragraph.Option option in paragraph.options)
    {
        AddButton(buttonPrefab,option);
    }
}
```

Figura 6-142 Funció ShowParagraph

La funció AddButton (Figura 6-143) s'encarrega de crear un botó depenent d'una Option i posar-lo com a fill del Canvas de botons.

```
private void AddButton(GameObject button, Paragraph.Option? option = null)
{
    GameObject auxButton = Instantiate(button);
    auxButton.transform.SetParent(buttonsCanvas.transform);
    auxButton.GetComponent<Button>()
        .onClick
        .AddListener(
            delegate {
                if (option == null)
                    ShowParagraph(_dialogue.NextParagraph());
                else
                    ShowParagraph(
                        _dialogue.NextParagraph(
                            (Paragraph.Option)option));
            });
    if (option != null)
        auxButton.GetComponentInChildren<Text>().text = option?.text;
}
```

Figura 6-143 Funció AddButton

Finalment tenim una funció per assegurar que els GameObjects que ens passen com a prefab de botó siguin botons (Figura 6-144).

```
private bool TestButton(GameObject button)
{
    return button.GetComponent<Button>() != null
        && button.transform.GetComponentInChildren<Text>() != null;
}
```

Figura 6-144 Funció TestButton

DialogBox

Aquesta és una classe molt simple que té la funció de relacionar un objecte amb un diàleg. També té una funció per mostrar aquest diàleg (Figura 6-145).

```
public class DialogBox : MonoBehaviour
{
    public Dialogue dialogue;
    0 referencias
    public void ShowDialogue()
    {
        dialogue.Show();
    }
}
```

Figura 6-145 Funció DialogBox

6.7 Lògica interna IA

VectorCircle

És la base per al funcionament de la Context Stearing AI. Consta de 8 vectors que apunten a 8 direccions separades per 45º i es poden fer operacions amb ells, com pot ser sumar, multiplicar, restar i girar.

Aquest element serà el que farà servir la IA per decidir en quina direcció anar. La part principal del Vector circle és un Array de floats de mida 8, que guarden la informació de cada direcció.

Per a poder fer operacions amb aquest Array, hi ha les següents funcions:

- Calcular força màxima i direcció:
- Calcular Força total
- Girar
- Sumar
- Multiplicar

La més important de les funcions és la de calcular força i direcció. La seva finalitat és trobar el punt del cercle de vectors on els vectors són més grans i en quina direcció està. Inicialment, sembla fàcil, ja que podríem agafar el vector més gran, però això porta diversos problemes, perquè el moviment final només seria en 8 direccions saltant d'una a l'altra. Per tant, necessitem una forma de saber la força entre vectors. La solució que s'ha fet servir és la proposada per Rabin. Quan volem buscar el valor més gran entre vectors agafem dos vectors d'un costat d'aquest espai i dos de l'altre costat, i fem servir aquests vectors per a crear dues rectes utilitzant els punts on acaben aquests vectors com es pot veure a la primera part de la Figura 6-146. Si la força del punt on fan intersecció aquestes rectes és més gran que la dels vectors que l'envolten podem determinar que la direcció d'aquest punt és la que hem de seguir, com es veu a la segona part de la Figura 6-146.

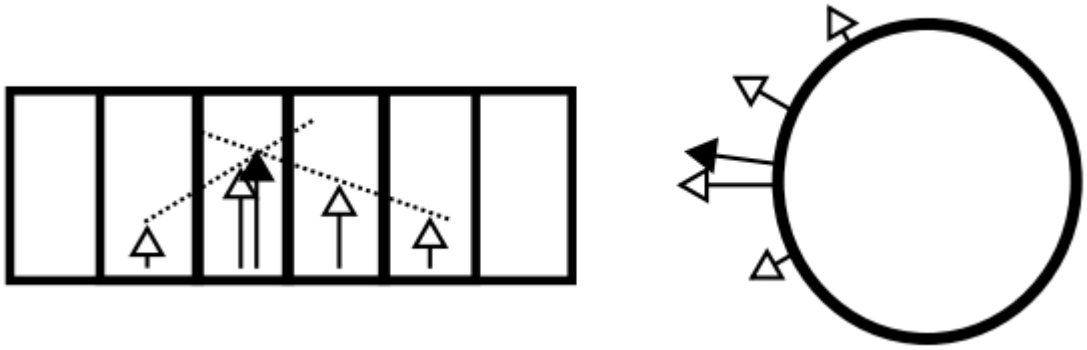


Figura 6-146 Direcció a partir d'un array de vectors (Rabin 2015)

Amb la funció Midpoint (Figura 6-147) trobem aquest punt.

```
private float Midpoint(float a, float b, float c, float d, out float auxPos)
{
    auxPos = 0;
    if (a > b || d > c) return 0;

    float B = b - a;
    float D = d - c;
    float C = c - D * 2;

    float x = (C - a) / (B - D); // pos
    float y = a + x * B; // str

    auxPos = x - 1;
    return y;
}
```

Figura 6-147 Funció Midpoint

Si fem sevir aquesta funció per cada espai entre vectors i mirem quin és el més gran, trobarem la direcció i força del VectorCircle (Figura 6-148).

```
private void BiggerPoint(out float place, out float strength)
{

    strength = MaxPos(out place);
    for(int i = 0; i < size; i++)
    {
        float auxPos;
        float aux = Midpoint(_vCircle[(i + size - 1) % size], _vCircle[i],
            _vCircle[(i + 1) % size], _vCircle[(i + 2) % size], out auxPos);
        if(aux > strength)
        {
            strength = aux;
            place = i + auxPos;
        }
    }
}
```

Figura 6-148 Funció BiggerPoint

Com que aquest càlcul podria ser molt lent si s'ha de fer moltes vegades, el resultat es guarda per si fes falta en un futur a les variables de savedStrength i savedDirection.

La funció per calcular la força total retrona la mitjana de força del VectorCircle tenint en compte que els vectors que apunten en direccions diferents es cancel·len, amb aquest valor podrem decidir amb la IA si, en cas d'haver-hi poca força, farem que la direcció retornada sigui zero i evitar "jittering", ja que, en els casos en els quals la força és molt petita, és molt possible que d'un fotograma al següent la direcció s'inverteixi i el personatge comenci a tremolar (Figura 6-149).

```
public float Strength() // Fet per 8
{
    float[] c = { 1, 0.70710678118f, 0, -0.70710678118f, -1,
        -0.70710678118f, 0, 0.70710678118f };
    float[] s = { 0, 0.70710678118f, 1, 0.70710678118f, 0,
        -0.70710678118f, -1, -0.70710678118f };
    float x = 0;
    float y = 0;
    for (int i = 0; i < size; i++)
    {
        x += _vCircle[i] * c[i];
        y += _vCircle[i] * s[i];
    }
    return (x > 0 ? x : -x) + (y > 0 ? y : -y);
}
```

Figura 6-149 Funció Strength

Per girar el VectorCircle fem servir la funció Turn (Figura 6-150) que fent servir un angle d'entrada fa que cada vector canviï de valor al corresponent a la posició del vector més l'angle de rotació.

```
public VectorCircle Turn(float angle)
{
    VectorCircle cAux = new VectorCircle();
    for(int i = 0; i < size; i++)
    {
        cAux._vCircle[i] = this.ValueAt((i * (360 / size)) + angle);
    }
    return cAux;
}
```

Figura 6-150 Funció Turn

També hi ha funcions per a fer operacions, com pot ser sumar o multiplicar (Figures 6-151 i 6-152).

```
public static VectorCircle operator +(VectorCircle a, VectorCircle b)
{
    float[] cAux = new float[size];
    for (int i = 0; i < size; i++)
    {
        cAux[i] = a._vCircle[i] + b._vCircle[i];
    }
    return new VectorCircle(cAux);
}
```

Figura 6-151 Operació suma VectorCircle

```
public static VectorCircle operator *(VectorCircle a, float b)
{
    float[] cAux = new float[size];
    for (int i = 0; i < size; i++)
    {
        cAux[i] = a._vCircle[i] * b;
    }
    return new VectorCircle(cAux);
}
```

Figura 6-152 Operació multiplicació VectorCircle

Finalment tenim altres funcions més senzilles com les següents:

- Inicialització/Creació del VectorCircle
- Normalitzar
- Valor i posició del vector més gran
- Restar
- Retornar VectorCircle com a String

VectorCircleMaker

VectorCircleMaker és una classe abstracta que serveix com a base per als dos tipus de creadors de VectorCircles. Aquests creadors serveixen per a construir VectorCircles sense tocar el codi en cap moment (Figura 6-153).

```
public abstract class VectorCircleMaker : ScriptableObject
{
    [SerializeField] protected VectorCircle _vectorCircle;
    1 referencia
    public VectorCircle VectorCircle => _vectorCircle;

}
```

Figura 6-153 Classe VectorCircleMaker

BasicVectorCircle

BasicVectorCircle és una classe que té un conjunt de vectors bàsics en totes direccions, junt amb una variable de classe enum, que serveix per crear una llista d'on triar vectors al menú de creació d'Unity, i una funció per passar de l'enum a VectorCircle

BasicVectorCircleMaker

BasicVectorCircleMaker és un ScriptableObject que hereta de VectorCircleMaker que fa servir els vectors bàsics del BasicVectorCircle per a crear VectorCircles. Ho fa barrejant dos arrays de VectorCircles amb pesos, uns que sumen i uns que resten per crear VectorCircles més complexos. Aquest procés el fa cada vegada que hi ha canvis, fins i tot abans que comenci el joc. Per tant, no consumirà recursos durant la partida (Figura 6-154).

```
[SerializeField] private BasicVectorsWheighted[] aditiveVectors;
[SerializeField] private BasicVectorsWheighted[] substractiveVectors;

Mensaje de Unity | 0 referencias
private void OnValidate()
{
    VectorCircle additiveV = new VectorCircle();
    VectorCircle substractiveV = new VectorCircle();

    foreach (BasicVectorsWheighted aV in aditiveVectors)
    {
        additiveV = additiveV +
            (BasicVectorCircle.enumToVector(aV.vectorType) * aV.wheight);
    }
    foreach (BasicVectorsWheighted sV in substractiveVectors)
    {
        substractiveV = substractiveV +
            (BasicVectorCircle.enumToVector(sV.vectorType) * sV.wheight);
    }

    _vectorCircle = additiveV - substractiveV;
}
```

Figura 6-154 BasicVectorCircleMaker

EditableVectorCircleMaker

EditableVectorCircleMaker és un ScriptableObject que hereda de VectorCircleMaker en el qual l'usuari pot construir un VectorCircle posant directament els valors a cada direcció (Figura 6-155).

```
private void OnValidate()
{
    float[] f = new float[VectorCircle.Size];
    f[0] = forward;
    f[1] = forward_left;
    f[2] = left;
    f[3] = back_left;
    f[4] = back;
    f[5] = back_right;
    f[6] = right;
    f[7] = forward_right;

    _vectorCircle = new VectorCircle(f);
}
```

Figura 6-155 EditableVectorCircleMaker

Condicions

Les condicions es fan servir per activar o desactivar certs VectorCircle o per donar-los un pes depenent de factors com la distancia.

Condition

És una classe abstracta que serveix com a base per fer altres condicions. Està formada per una TagList i una funció que retorna un float que serà el pes (Figura 6-156)

```
public abstract class Condition : ScriptableObject
{
    [SerializeField] protected TagList _tag;
    1 referencia
    public TagList tagList => _tag;
    4 referencias
    public abstract float Wheight(float condition = 0);
}
```

Figura 6-156 Classe Condition

ConditionedDistance

És una classe que hereta de Condition que implementa la funció Wheight. Per fer-ho, fa servir 3 variables que s'han d'inicialitzar des del menú de Unity, que són el tipus de relació entre el pes que retornarem, i la distancià i entre quines distàncies s'aplicarà el pes (Figura 6-157 i Funció 6-158)

```
[SerializeField] private float _min_distance;
[SerializeField] private float _max_distance;
[SerializeField] private wheightTipe _wheight_tipe;
```

Figura 6-157 Variables funció ConditionedDistance

```
public override float Wheight(float distance)
{
    if (distance < _min_distance || distance > _max_distance) return 0;
    float aux_d = (distance - _min_distance) / _max_distance;
    if (_wheight_tipe == wheightTipe.Constant) return 1;
    if (_wheight_tipe == wheightTipe.LiniarNegative) return aux_d;
    if (_wheight_tipe == wheightTipe.LiniarPositive) return 1 - aux_d;
    if (_wheight_tipe == wheightTipe.Exponential) return 1 - (aux_d * aux_d);
    if (_wheight_tipe == wheightTipe.InverseExponential) return aux_d * aux_d;

    return 0;
}
```

Figura 6-158 Funció Weight

ConditionalVectorCircle (CVC)

És la classe que s'encarrega d'ajuntar la part de Condicions amb els VectorCircle i si s'afegirà al vector final o es restarà. Només té tres variables que serien: la condició, el cercle de vectors, i si se suma o es resta, i una funció per comprovar si la condició es vàlida o no (Figura 6-159).

```
public class ConditionalVectorCircle : ScriptableObject
{
    [SerializeField] private Condition _condition;
    4 referencias
    public Condition Condition => _condition;
    [SerializeField] private VectorCircleMaker _vectorCircleMaker;
    2 referencias
    public VectorCircle VectorCircle => _vectorCircleMaker.VectorCircle;
    [SerializeField] private bool _isAditive = true;
    2 referencias
    public bool IsAditive => _isAditive;

    2 referencias
    public bool isValid(TagList tl)
    {
        return _condition.tagList.isValid(tl);
    }
```

Figura 6-159 Funció ConditionalVectorCircle

BasicDirectionAI

La classe BasicDirectionAi és una classe abstracta, que marca la base per a IAs que retornen simplement la direcció a la qual s'ha d'anar, com és el cas de la ContextStearingAI. D'aquesta forma és molt més fàcil barrejar diferentes IAs en el cas que l'usuari ho necessités.

Simplement, consta de dues funcions:

- Direcció
- Actualitzar direcció

(Vegeu Figura 6-160).

```
public abstract class BasicDirectionAi : MonoBehaviour
{
    protected Vector2 _direction;
    2 referencias
    public Vector2 Direction => _direction;
    Mensaje de Unity | 0 referencias
    private void Update()
    {
        UpdateDirection();
    }
    2 referencias
    protected abstract void UpdateDirection();
}
```

Figura 6-160 Classe BasicDirectionAi

ContextStearingAI

Aquesta és la part que s'ocupa del tema de deteccions i càlculs per retornar la direcció a la qual es mourà el personatge. Podríem dir que és el cervell de la IA. Té quatre funcions principals i una classe interna que serveix per guardar els objectes que estan dins el trigger que vindria a ser el camp de visió de la IA.

Aquesta classe interna anomenada ObjectCVC guarda el següent (Figura 6-161):

- L'objecte al qual s'apliquen els CVC.
- La llista de CVCs que s'apliquen a aquest objecte.
- Els VectorCircleja calculats depenent de la distància separats per si són additius o subtractius.
- L'angle al qual està l'objecte per poder girar els VectorCircle cap a la direcció a la qual està aquest objecte.

```
public struct ObjectCVC{
    public GameObject go;
    public List<ConditionalVectorCircle> cvcs;
    public VectorCircle additiveVC;
    public VectorCircle substractiveVC;
    public float angle;
```

Figura 6-161 Classe ObjectCVC

També té dues funcions, la funció de creació/inicialització i la funció que calcula els vectors additius i subtractius i inicialitza l'angle (Figura 6-162).

```
public void CalculateVectors(float distance)
{
    int nAdd = 0;
    int nSub = 0;
    additiveVC = new VectorCircle();
    substractiveVC = new VectorCircle();
    foreach (ConditionalVectorCircle cvc in cvcs)
    {
        if(cvc.Condition.Wheight(distance) != 0)
        {
            VectorCircle aux = cvc.VectorCircle *
                (cvc.Condition.Wheight(distance));
            if (cvc.IsAditive)
            {
                additiveVC += aux;
                nAdd++;
            }
            else
            {
                substractiveVC += aux;
                nSub++;
            }
        }
    }
    if(nAdd>0)additiveVC *= 1 / nAdd;
    if(nSub>0)substractiveVC *= 1 / nSub;

}
```

Figura 6-162 Funció CalculateVectors

Les quatre funcions importants de la IA són les següents:

- Afegir nous ObjectCVC.
- Treure un ObjectCVC en concret.
- Actualitzar els ObjectCVC.
- Calcular la direcció final.

La funció d'afegir nous ObjectesCVC es crida amb la funció OnTriggerEnter2D() (Figura 6-163) i, com a paràmetre d'entrada, té l'objecte que ha entrat a l'àrea de "trigger" i la distancia a la que està l'objecte.

```
private void OnTriggerEnter2D(Collider2D collision)
{
    Vector2 closestPoint = collision.ClosestPoint(this.gameObject.transform.position);
    float distance = Vector2.Distance(closestPoint, this.gameObject.transform.position);
    AddObjectCVC(collision.gameObject,distance);
}
```

Figura 6-163 Funció OnTriggerEnter2D

Després la funció AddObjectCVC (Figura 6-164) busca per totes les condicions que té guardades la IA i afegeix a l'ObjectCVC totes les que coincideixin amb les Tags de l'objecte i l'afegeix a la llista d'ObjectCVC.

```
private void AddObjectCVC(GameObject go, float distance)
{
    TagList tl = go.GetComponent<Tagged>()?.TagList;
    if (tl == null) return;
    List<ConditionalVectorCircle> auxCVCs = new List<ConditionalVectorCircle>();
    _cvcs.ForEach(x =>
    {
        if (x.isValid(tl))
        {
            auxCVCs.Add(x);
        }
    });
    OCVCs.Add(new ObjectCVC(go, auxCVCs, distance));
}
```

Figura 6-164 Funció AddObjectCVC

La funció per esborrar elements de la llista d'ObjectCVC s'activa amb la funció OnTriggerExit2D() (Figura 6-165).

```
private void OnTriggerExit2D(Collider2D collision)
{
    RemoveObjectCVC(collision.gameObject);
}
```

Figura 6-165 Funció OnTriggerExit2D

Aquesta simplement elimina l'ObjectCVC que conté l'objecte entrat per paràmetre (Figura 6-166).

```
private void RemoveObjectCVC(GameObject go)
{
    foreach(ObjectCVC ocvc in OCVCs)
    {
        if (ocvc.go.Equals(go))
        {
            OCVCs.Remove(ocvc);
            return;
        }
    }
}
```

Figura 6-166 Funció RemoveObjectCVC

Les dues últimes funcions s'encarreguen d'actualitzar la informació dels ObjectCVC i fer-la servir per determinar la direcció. Primer, per actualitzar la informació fem servir la funció UpdateCVC (Figura 6-167), la seva funció és calcular la distància i l'angle a la que està l'objecte, i actualitzar tots els CVC que té associats.

```
private void UpdateCVCs()
{
    for(int i = 0; i<OCVCs.Count; i++)
    {
        ObjectCVC ocvc = OCVCs[i];
        Collider2D c2d = ocvc.go.GetComponent<Collider2D>();
        if (c2d == null) return;
        //Calcular distancia
        float distance = Vector2.Distance(this.gameObject.transform.position,
            c2d.ClosestPoint(this.gameObject.transform.position));
        //Actualitzar CVC
        ocvc.CalculateVectors(distance);

        //Calculat angle
        Vector3 dir = this.gameObject.transform.position - ocvc.go.transform.position;
        dir = this.gameObject.transform.InverseTransformDirection(dir);
        float angle = (Mathf.Atan2(dir.y, dir.x) * Mathf.Rad2Deg + 180) % 360;
        ocvc.angle = angle;
        //Actualitzar OCVC
        OCVCs[i] = ocvc;
    }
}
```

Figura 6-167 Funció UpdateCVC

Finalment, amb la informació actualitzada només queda crear el VectorCircle que uneix tots els CVC i calcular la direcció d'aquest amb la funció "CalculateStrengthDirection" del VectorCircle (Figura 6-168)

```
private void CalculateDirection()
{
    VectorCircle additiveVC = new VectorCircle();
    VectorCircle substractiveVC = new VectorCircle();

    // Calcular VectorCircle additiu i substractiu
    foreach (ObjectCVC ocvc in OCVCs)
    {
        additiveVC += ocvc.additiveVC.Turn(ocvc.angle);
        substractiveVC += ocvc.substractiveVC.Turn(ocvc.angle);
    }

    // Calvular VectorCircle final
    _vectorCircle = additiveVC - substractiveVC;

    _vectorCircle.CalculateStrenghtDirection();
    _angle = _vectorCircle.savedDirection;

    // Crear Vector2D de direcció
    float rad = _angle * Mathf.Deg2Rad;
    _direction = new Vector2(Mathf.Cos(rad), Mathf.Sin(rad)).normalized;

    // Aplicar llindar de força
    if (_vectorCircle.Strength() < _stopThreshold)
        _direction = new Vector2(0, 0);
    //Mostrar direcció en Debug
    Debug.DrawRay(this.transform.position
        ,_direction,Color.green,Time.deltaTime,false);
}
```

Figura 6-168 Funció CalculateDirection

Enemy

Finalment, per a demostrar com es faria servir, s'ha creat la classe Enemy qu implementa de forma senzilla la IA de forma que desplaci el personatge depenent de la direcció que retorna la IA i la velocitat (Figura 6-169).

```
public class Enemy : MonoBehaviour
{
    [SerializeField] BasicDirectionAi AI;
    public float speed = 5;
    Mensaje de Unity | 0 referencias
    void Update()
    {
        this.transform.position = new Vector3(
            this.transform.position.x + (AI.Direction.x * speed *Time.deltaTime),
            this.transform.position.y +(AI.Direction.y * speed * Time.deltaTime),
            this.transform.position.z);
    }
}
```

Figura 6-169 Classe Enemy

Com podem veure, el codi és senzill i elegant, gràcies a les funcions de la llibreria desenvolupada.

7. Resultats

En aquest apartat es mostraran els resultats del treball en forma de "demo", es podrà veure com quedarien moltes de les llibreries explicades anteriorment si les féssim servir en un videojoc. Cal comentar que la majoria de llibreries es poden utilitzar sense tocar codi en cap moment, això ha permès que per fer la "demo" no hagi fet falta programar res.

Gran part d'aquest treball es basa en la facilitat que tindrà la persona encarregada de crear el videojoc per implementar aquestes llibreries, per tant, podem considerar que part dels resultats seria el manual d'usuari de l'apartat 12.1.

7.1 Demo Inventari

Recol·lecció d'objectes

A les Figures 7-1 i 7-2 es pot veure el comportament de recol·lecció d'objectes. A la primera figura l'objecte no s'ha recollit, i a la segona ja s'ha recollit i col·locat a l'inventari.

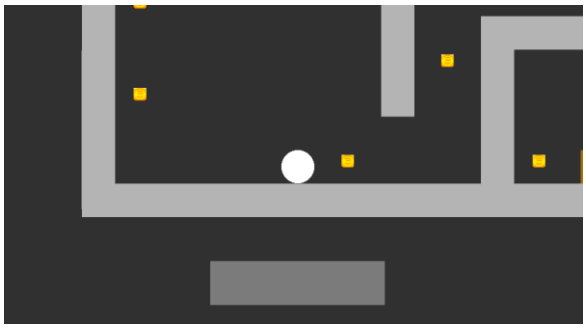


Figura 7-1 Comportament de recol·lecció d'objectes

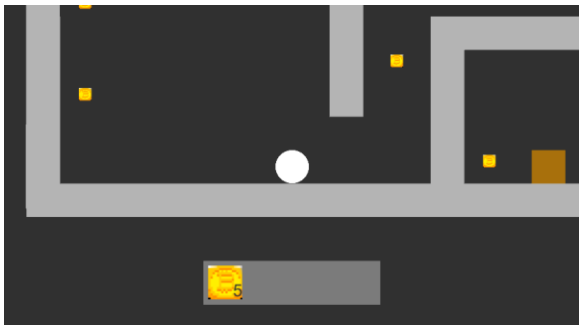


Figura 7-2 Comportament de recol·lecció d'objectes

Aquest comportament és el dels components Collectable, que té la moneda, i Collector, que té el jugador.

Gestió d'inventari

A les Figures 7-3 i 7-4 es pot veure el comportament d'intercanviar ítems d'un inventari a un altre. A la primera figura els ítems estan al cofre i a la segona a l'inventari personal.

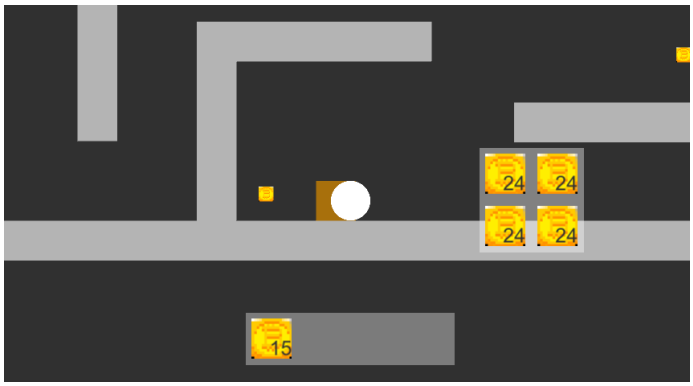


Figura 7-3 Gestió d'inventari

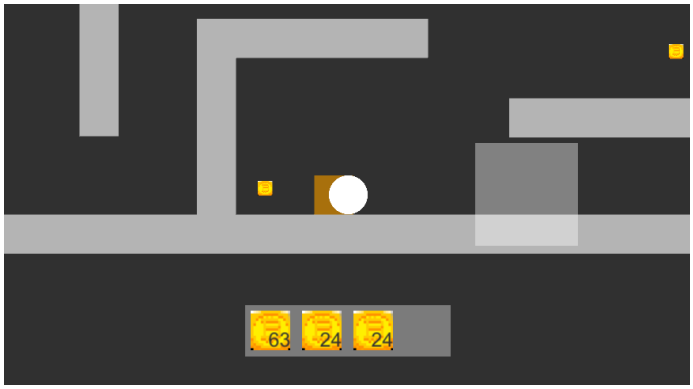


Figura 7-4 Gestió d'inventari

Aquest comportament utilitza els components Chest, InventoryDisplay, ChestOpener i InventoryController. El component Chest guarda l'InventoryDisplay que s'ha de mostrar. El component ChestOpener detecta el cofre i envia la informació a l'InventoryDisplay perquè ho mostri. Un cop mostrat es poden intercanviar objectes d'un lloc a l'altre amb el ratolí.

7.2 Diàlegs

A les Figures 7-5 i 7-6 podem veure un exemple de diàleg amb dues parts.

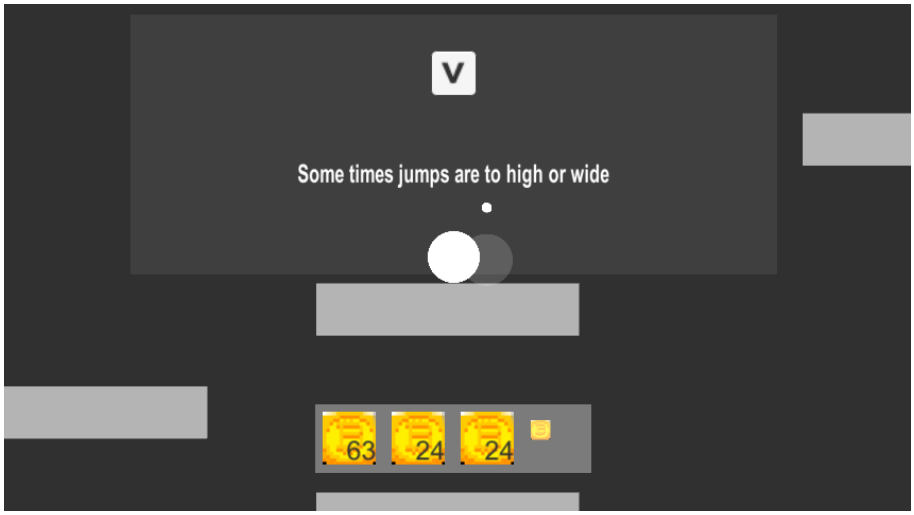


Figura 7-5 Diàlegs

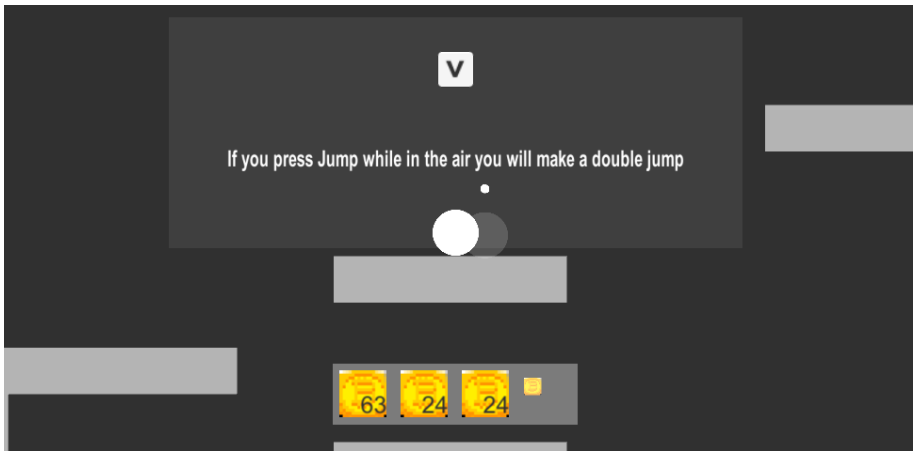


Figura 7-6 Diàlegs

Aquests diàlegs han estat creats com a ScriptableObject de tipus Dialogue, es guarden en un objecte que té un Interactuable i quan s'hi interacciona envia el diàleg al DialogueManager que s'encarrega de mostrar-lo.

7.3 Menús

Menú victòria

El menú de victòria (Figura 7-7) ens permet escollir entre anar al nivell següent o tornar al menú principal.

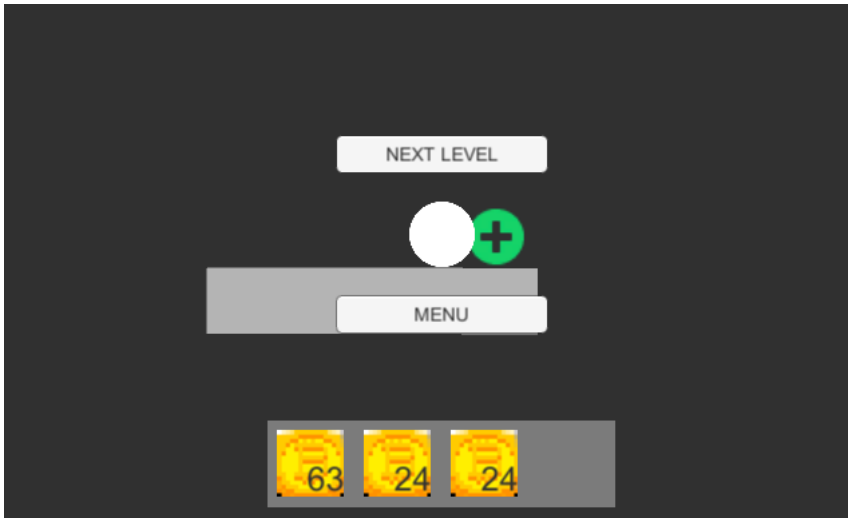


Figura 7-7 Menú victòria

Menú de pausa

El menú de pausa (Figura 7-8) ens permet reprendre el joc, obrir el menú d'opcions i tornar al menú principal.

Figura 7-8 Menú de pausa

Menú d'opcions

Ens permet canviar la resolució, la qualitat dels gràfics, posar pantalla completa i canviar el volum del joc (Figura 7-9).

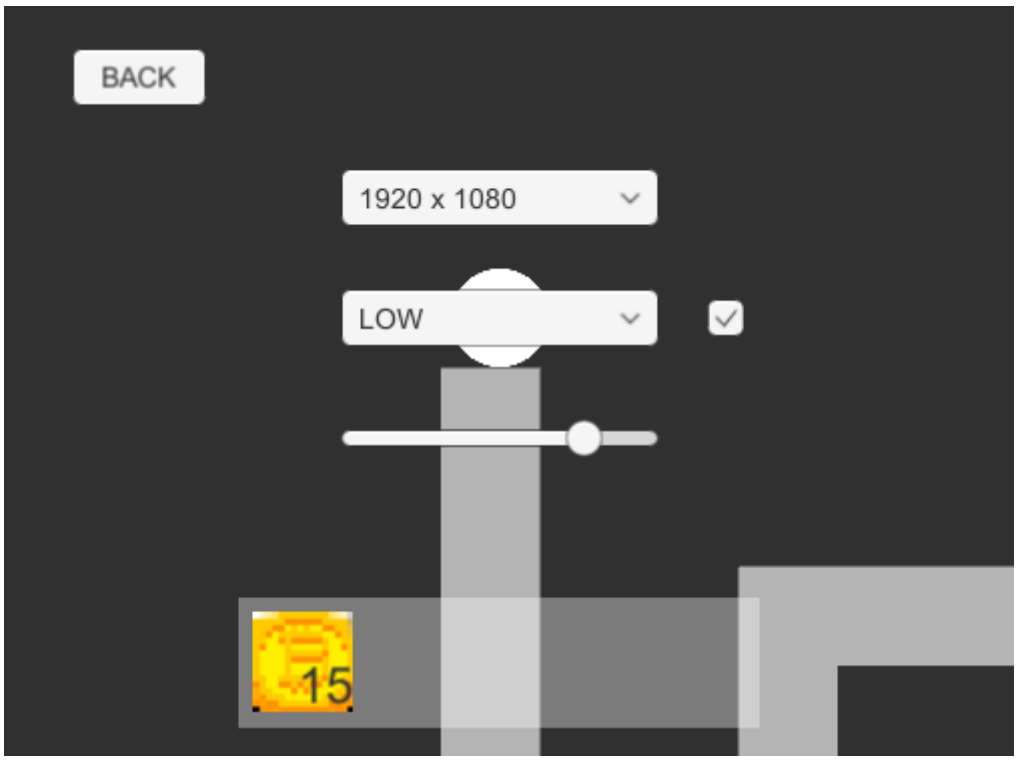


Figura 7-9 Menú d'opcions

Tots aquests menús són editables en un prefab, de forma que es pot canviar l'aspecte i els botons que té cadascun d'ells. Les funcions que fa cada botó les processa el MenuController.

7.4 Interactuables

Els interactuables permeten que el personatge interactuï amb diversos objectes quan estan a dins el detector. A l'exemple següent es fa servir un interactuable per prémer un botó que fa obrir una porta (Figures 7-10, 7-11 i 7-12). A més a més, el botó fa servir el component OnTrigger per a mostrar un indicador del fet que el personatge està en rang per a interactuar amb l'interactuable.

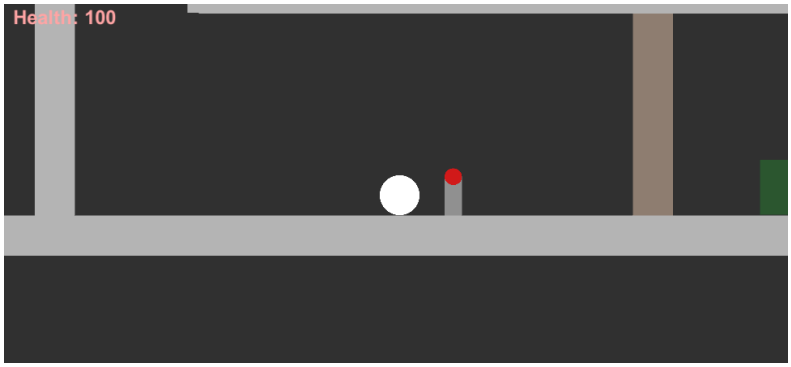


Figura 7-10 Interacció amb porta

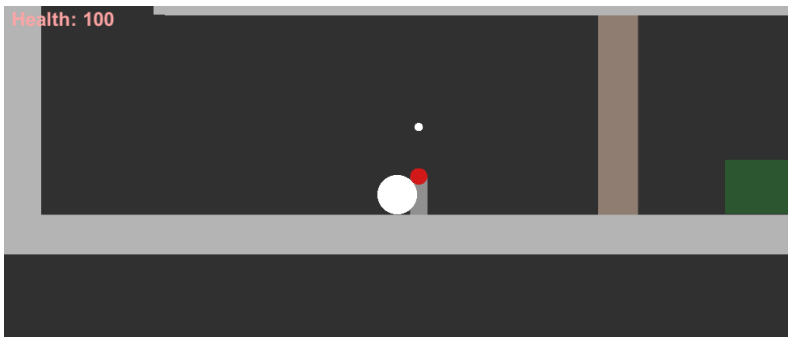


Figura 7-11 Interacció amb porta

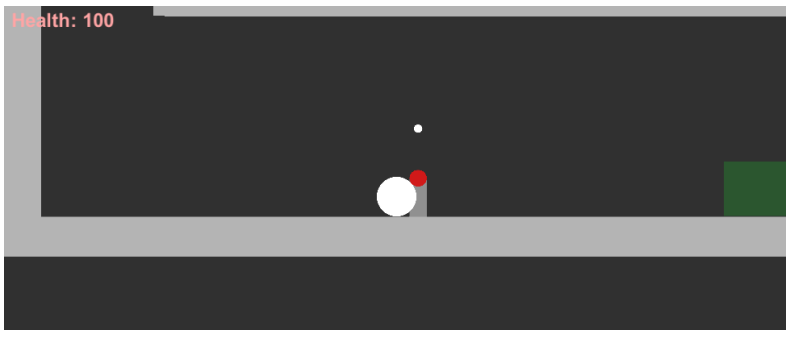


Figura 7-12 Interacció amb porta

7.5 Atacs

A les següents figures es pot veure com funciona el sistema d'atacs. La primera zona fa mal mentre el personatge hi està, un cop surt ja no li fa mal (Figures 7-13 i 7-14).

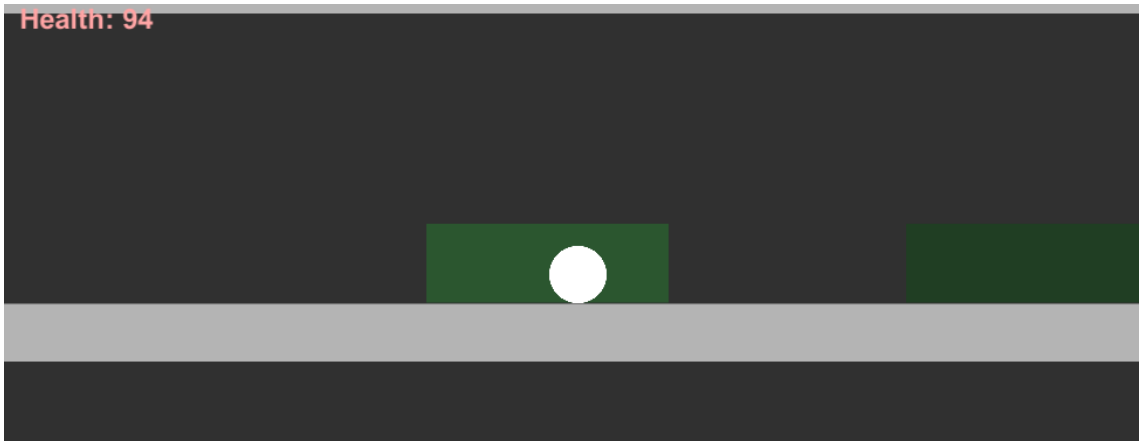


Figura 7-13 Primera zona atacs

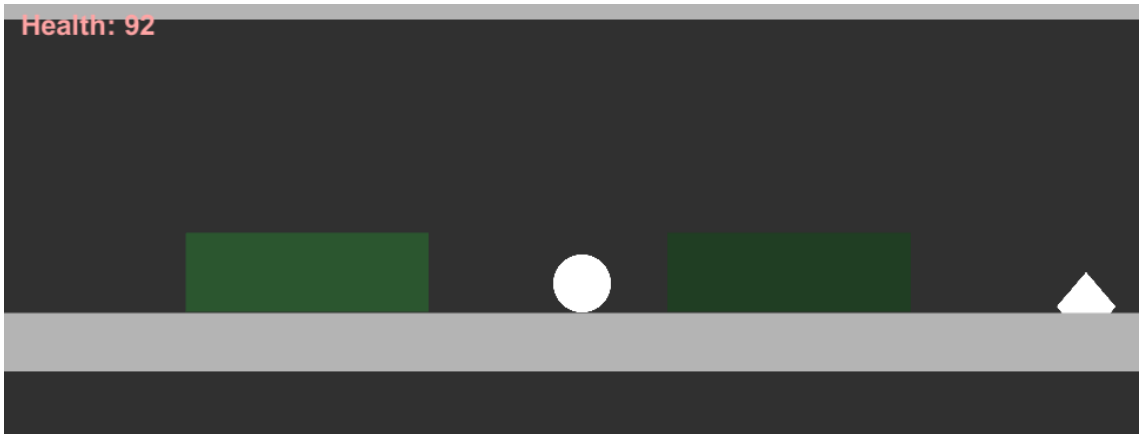


Figura 7-14 Primera zona atacs

Això ho aconseguim configurant el component AttackOnTrigger perquè deixi de fer mal en sortir de la zona.

La segona zona un cop hi entrem el mal que et fa dura un cert temps (Figures 7-15 i 7-16).

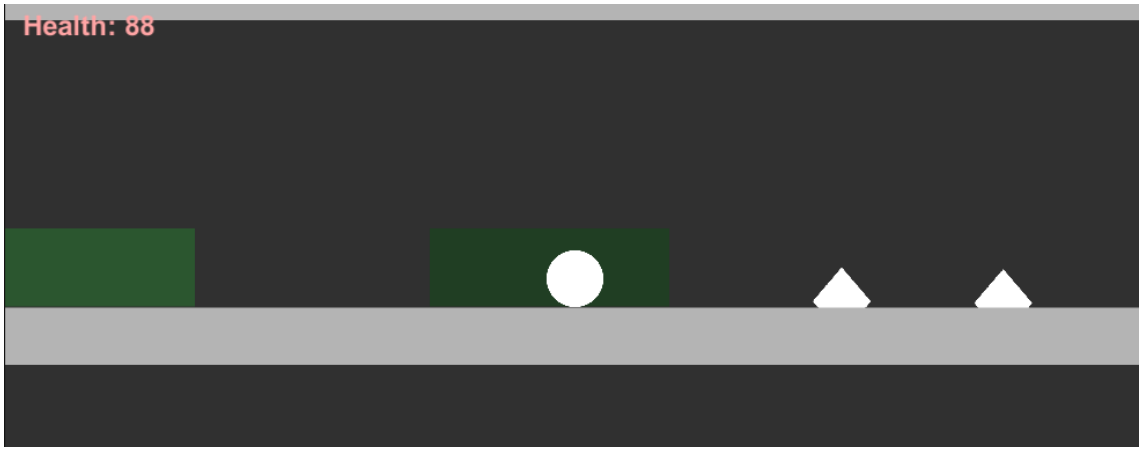


Figura 7-15 Segona zona atacs

Figura 7-16 Segona zona atacs

En aquest exemple l'AttackOnTrigger no té seleccionada l'opció de deixar de fer mal quan surts del trigger.

La tercera zona que serien les punxes et fa mal només l'instant en el qual hi entrem (Figures 7-17 i 7-18).

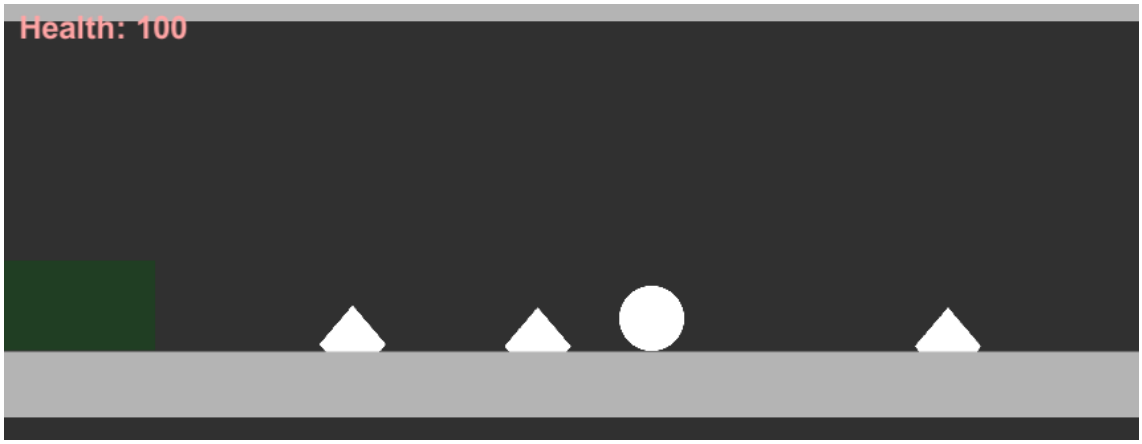


Figura 7-17 Tercera zona atacs

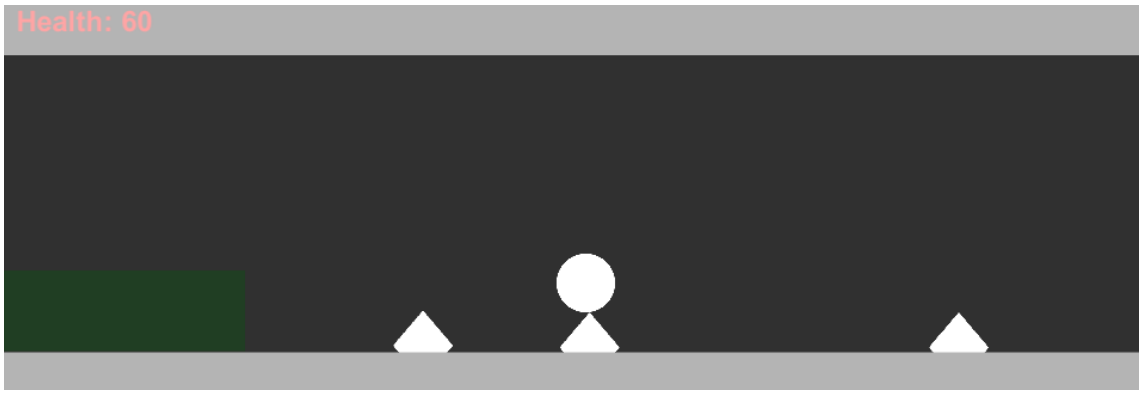


Figura 7-18 Tercera zona atacs

En aquest exemple l'atac que fa a l'entrar al trigger està configurat de forma que sigui instantani i no tingui durada.

7.6 Context Stearing AI

Els següents exemples són de dos comportaments diferents que es poden aconseguir amb la intel·ligència artificial d'aquesta llibreria. En el primer cas tenim un enemic amb una intel·ligència que li permet esquivar certs obstacles i perseguir el personatge principal (Figures 7-19 i 7-20).

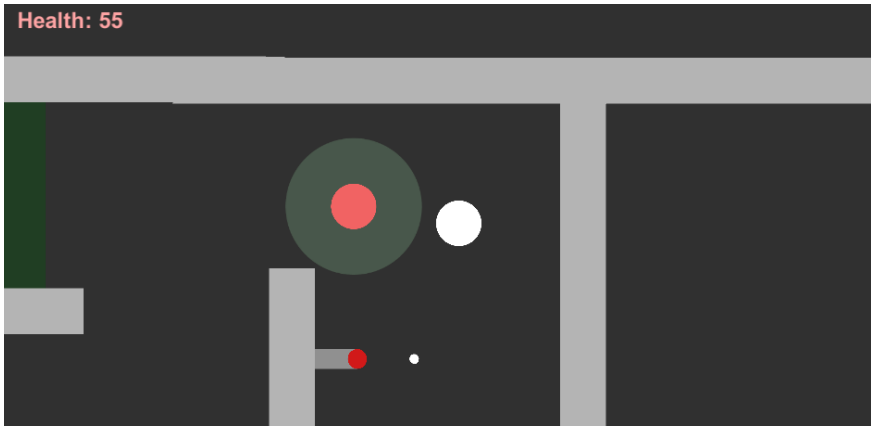


Figura 7-19 Context Stearing AI perseguint personatge

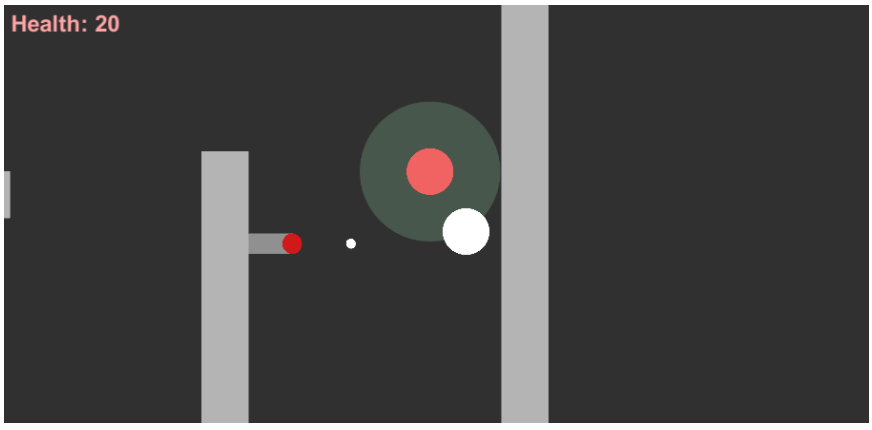


Figura 7-20 Context Stearing AI perseguint personatge

Per a aconseguir aquest afecte hem posat dos ContextVectorCircle, un que detecta parets i resta el VectorCircle que apunta a la paret i l'altra que detecta el jugador i suma el VectorCircle que l'apunta.

El segon cas és el d'un objecte que té por del personatge principal i intenta fugir d'aquest (Figures 7-21 i 7-22)

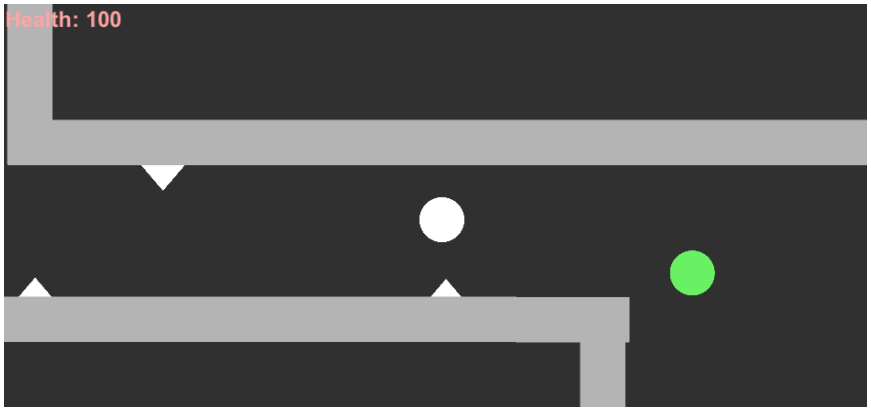


Figura 7-21 Context Stearing AI fugint del personatge

Figura 7-22 Context Stearing AI fugint del personatge

Aquest comportament l'aconseguim fent que en comptes que el VectorCitcle apunti cap al personatge com en el cas anterior, ara apunta en direcció contrària.

Finalment, alguns dels comportaments del Context Stearing AI queden recollits a l'Annex en forma de vídeo.

7.7 Moviment de personatge

El resultat del moviment del personatge queda recollit a l'Annex en forma de vídeo.

8. Conclusions

Per a aconseguir el propòsit d'aquest treball, es van fixar prèviament un llistat de tasques i passos a dur a terme (vegeu apartat 1.3 Objectius del projecte). Al llarg de la realització d'aquest treball, s'ha après a dissenyar i crear sistemes genèrics que es puguin personalitzar fàcilment canviant certs paràmetres. També s'ha après a utilitzar les opcions de la interfície d'Unity per a facilitar l'edició i la creació de certs elements. S'ha après sobre el funcionament d'alguns elements dels jocs de rol, com ara els inventaris o el sistema d'atac. I finalment s'ha après sobre la intel·ligència artificial de tipus Context Stearing. Tot i això, un dels aprenentatges més valuosos que s'han obtingut gràcies a aquest treball ha estat el disseny d'elements per videojocs, no des d'un punt de vista tant específic per a una funció en concret, sinó des d'un punt de vista més flexible, que permet fer canvis de forma molt més fàcil en un futur així com la creació i disseny d'elements genèrics que es poden fer servir a diversos llocs d'un projecte, com poden ser: temporitzadors, detectors o sistemes d'interacció entre d'altres.

Pel que fa a resultats, en aquest projecte s'ha assolit la creació de sis elements clau dels jocs de rol (moviment de personatge, menús, atacs, diàlegs, inventari i IA) amb diverses opcions de personalització. A més a més, tots aquests elements poden ser implementats sense tocar res del codi, ni crear codi nou, de forma que algú que no sap res de programació podria implementar aquestes llibreries.

També s'ha aconseguit la creació d'altres sistemes per a accions més genèriques, com pot ser la interacció amb objectes, sistemes de detecció i identificació, així com eines com ara un temporitzador.

Finalment, cal esmentar que els resultats han estat els que s'esperaven: aconseguir crear un conjunt d'eines per a facilitar als usuaris la creació de videojocs.

9. Treball futur

Propostes i millores de cara al futur

- Afegir nous elements a les llibreries, com ara:
  - Sistema de guardar la partida
  - Sistema de batalles per torns
  - Control global dels controls del personatge que pugui ser canviat al menú d'opcions
  - Interfície d'usuari per l'estat del personatge
  - Sistema de mana o stamina
  - Sistema de missions
  - Sistemes més genèrics com pot ser: parallax, música adaptativa o post processat d'imatge
- Afegir més opcions als menús
- Fer que els ítems de l'inventari puguin afectar als atacs i defensa
- Millorar el sistema d'atac perquè sigui condicional
- Afegir descripcions pels ítems de l'inventari

10. Bibliografia

Rabin, S. (2015). *Game AI Pro 2: Collected Wisdom of Game AI Professionals.* A K Peters.

Unity Technologies. (2022, August 01). *Unity Asset Store*. Retrieved from https://assetstore.unity.com/packages/templates/packs/2d-rpg-kit-163910: https://assetstore.unity.com/packages/templates/packs/2d-rpg-kit-163910

Unity Technologies. (2022, July 28). *Unity User Manual 2021.3 (LTS)*. Retrieved from https://docs.unity3d.com/Manual/index.html

11. Annex

Tots els vídeos mencionats en els apartats anteriors queden recollits a la seguent carpeta:
https://drive.google.com/drive/folders/1VA6Mt4B3BNB-rRz32PPmySiQtPfAhz7C?usp=sharing

12. Manual d'usuari i d'instal·lació

12.1 Manual d'usuari

12.1.1 Exemple d'ús d'un detector

Els detectors són molt fàcils de crear, simplement anem a un objecte, anem a l'opció d'afegir component i triem un dels detectors, n'hi ha de tres tipus (Funció 11-1).

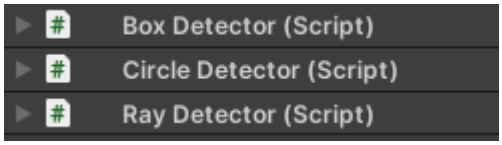


Figura 12-1 Tipus de detectors

El Box Detector detecta els Colliders que hi ha en una àrea rectangular a la qual nosaltres li donarem les mides i l'angle, en el cas de voler girar el rectangle, i també li donarem un origen (Figura 11-2).

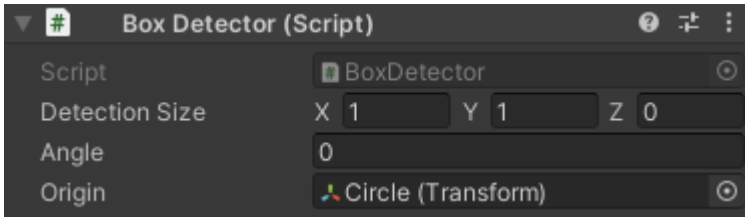


Figura 12-2 BoxDetector

El Circle Detector funciona de forma similar, però en forma de cercles, simplement caldrà donar-li el radi i l'origen (Figura 11-3).

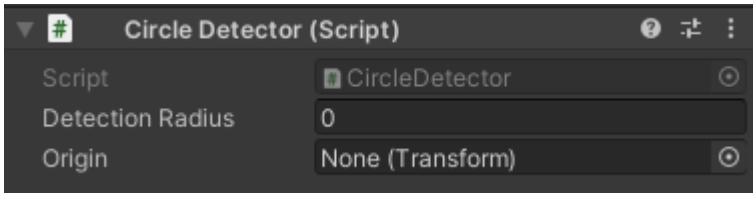


Figura 12-3 Circle Detector

Finalment per al RayDetector detectarà els objectes en una línia recta, els paràmetres que li donarem seran, l'origen, el destí i la llargada de la línia (Figura 11-4).

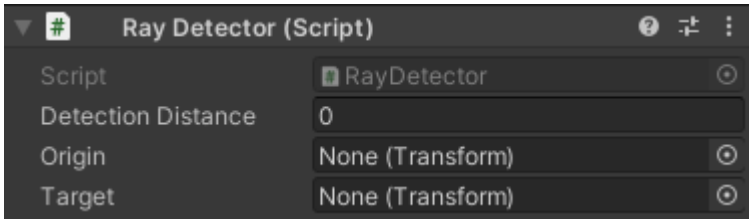


Figura 12-4 RayDetector

Els detectors normalment es posaran com a paràmetre a altres scripts que vulguin fer-los servir.

12.1.2 Exemple ús Interaccions

Interacter/Interactuable

Interacter i Interactuable son dos components que junts permeten a un objecte interactuar amb un altre prement una tecla. L'Interacter està pensat per posar al personatge principal i serveix per interactuar amb qualsevol Interactuable. Per fer-lo servir, només cal assignar-lo a l'objecte que volíem que interactuï i inicialitzar els paràmetres de Detector, que serà el que fa servir per detectar Interactuables, i una tecla que activarà l'Interactuable (Figura 11-5).

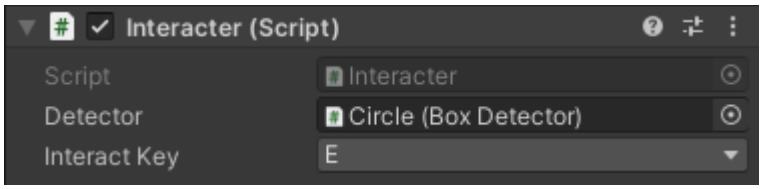


Figura 12-5 Interacter

L'Interactuable és encara més fàcil de fer servir. Simplement, li assignarem l'script Interactuable a qualsevol objecte que tingui un Collider2D. Després només cal assignar-li la funció o funcions que volem que activi (Figura 11-6).

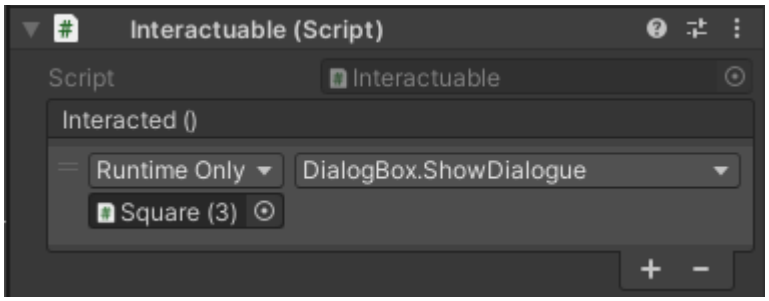


Figura 12-6 Interactuable

OnCollision2D / OnTriggerer2D

Aquests scripts s'encarreguen que les funcions d'entrada, sortida i estada dels colliders i triggers, puguin ser accedides des de l'inspector de Unity. Els dos es fan servir de la mateixa manera. Assignem l'script a un objecte que tingui un Collider2D, depenent de si és un trigger o no fem servir un script o l'altre. Després només caldrà assignar l'acció que farem en cas que hi hagi una interacció (Figura 11-7).

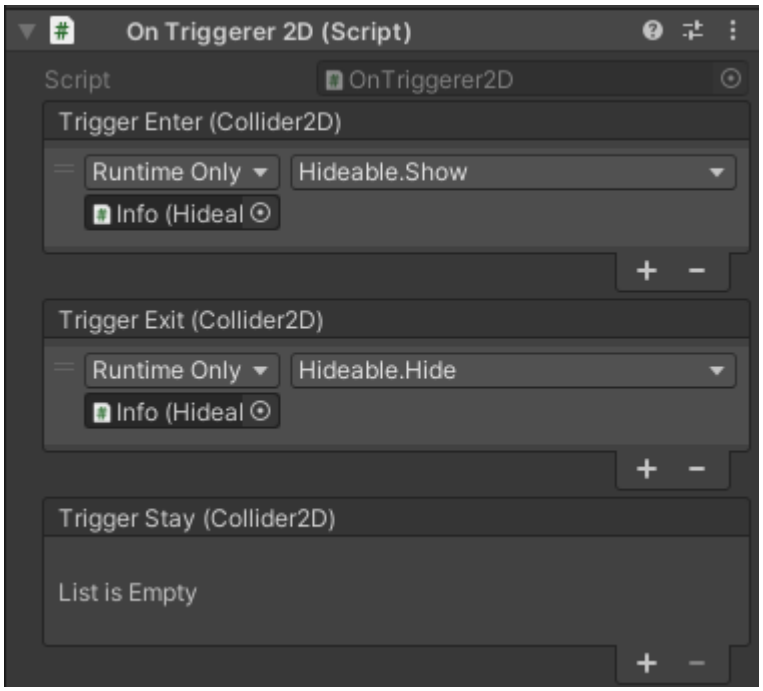


Figura 12-7 OnTrigger2D

ClickableObject

Aquest script serveix per a clicar elements de la interfície d'usuari. Encara que no sigui un botó, si té l'script ClickableObject podrem assignar-li accions a fer al ser clicat. Hi ha quatre possibles actes que podem fer servir per activar altres funcions. Aquestes són: clic dret, clic esquerra, clic mig, i clic que dona com a paràmetre un integer amb el tipus de clic (Figura 11-8).

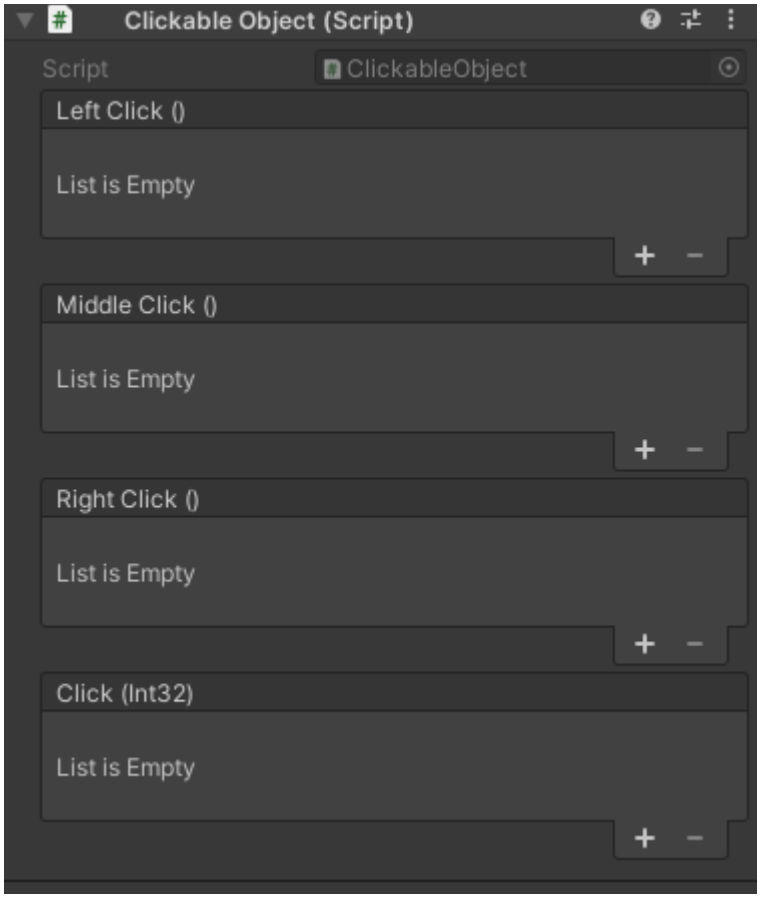


Figura 12-8 Clickable Object

12.1.3 Exemple d'ús Temporitzador

Per fer-lo servir, simplement creem un Timer amb la funció que volem cridar i el temps que volem que passi abans de cridar-la. Aquesta funció i temps poden ser canviats amb els mètodes setMethod i setDuration.

Si volguéssim cridar una funció amb paràmetres o que fes alguna cosa més complicada, podríem fer servir la expressió "delegate" i entre "{}" posar-hi la part de codi que vulguem que s'executi. També podem fer servir el mètode loop per fer que hi hagi un bucle infinit, o el mètode loopN perquè el bucle només duri un nombre determinat d'iteracions.

Finalment per a fer que comenci el temporitzador, faríem servir el mètode start i quan vulguem que pari farem servir el mètode stop (Figura 11-9)

```
Timer t = new Timer(MyFunction, 1);

t.setDuration(2.3f);
t.setMethod(delegate
    {
        MyFunction2(1);
        if(true) MyFunction();
    });

t.loop();
t.loopN(3);

t.start();
t.stop();
```

Figura 12-9 Exemple funcionament timer

12.1.4 Manual Tags

El sistema funciona de la següent forma: primer de tot es creen les Tags individuals. Per fer-ho cal anar al desplegable que es genera en fer clic amb el botó dret del ratolí dins de qualsevol carpeta d'assets de Unity. Aquí es pot trobar l'opció de crear una Tag o una TagList. Primer farem les Tags (Figura 11-10).

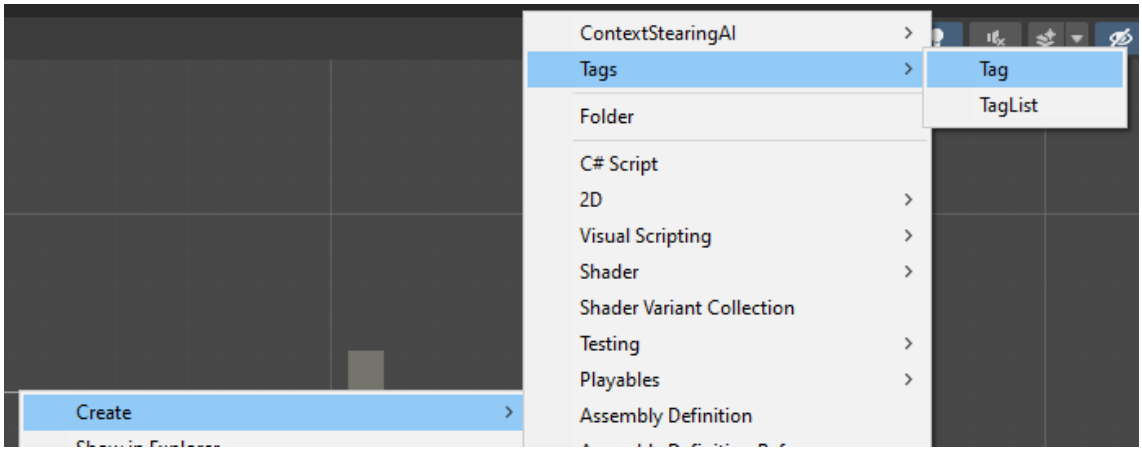


Figura 12-10 Desplegable Tag

Un cop es genera l'script de la tag (Figura 11-11) ens apareixerà el següent:

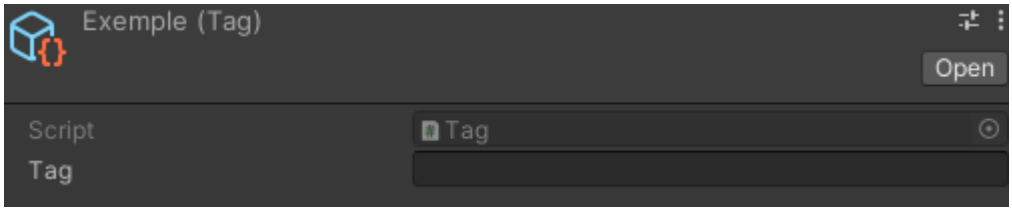


Figura 12-11 Exemple (Tag)

Aquí simplement hi posarem el nom que el sistema de tags farà servir per identificar aquesta tag. Un cop creades les tags que es creguin necessàries, caldrà fer un conjunt de tags. De la mateixa manera que s'han generat les tags, a continuació caldrà generar una TagList i un cop generada, se li assignaran les tags que li corresponguin (Figura 11-12).

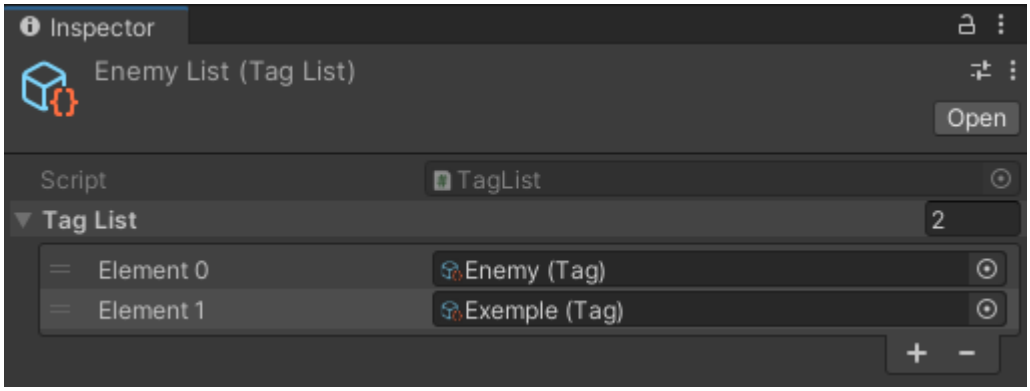


Figura 12-12 Creació TagList

Un cop creada la TagList, se li podrà afegir a un objecte posant-li l'script de Tagged i se li assignarà la TagList que s'ha creat prèviament.

12.1.5 Exemple d'ús del moviment de personatge

PlayerController2DTopDown

Primer veurem el moviment de personatge en un joc amb vista des de dalt. El personatge podrà anar de dreta a esquerra, amunt i avall, a més a més també podrà utilitzar un esprint si l'usuari ho vol. Per a fer que un objecte faci servir el moviment de personatge simplement caldrà afegir-li els scripts de Collider2D que vulguem, el de RigidBody2D amb la gravetat a zero i el de PlayerController2DTopDown. Per a personalitzar el moviment, tenim les opcions següents (Figura 11-13).

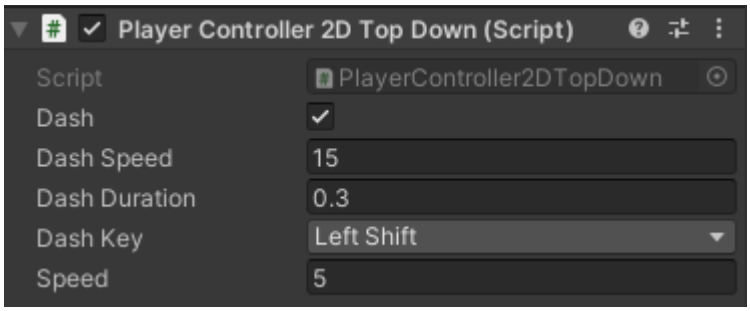


Figura 12-13 PlayerController2DTopDown.

Les quatre primeres opcions són per editar l'esprint que pot fer el personatge i l'última és per canviar la velocitat normal d'aquest.

PlayerController2DHorizontal

L'altre script de moviment de personatge és el de moviment horitzontal, aquest té moltes més opcions de personalització, ja que implementa: el salt, enganxar-se a les parets, entre d'altres. Les variables que podem fer servir per personalitzar el moviment són (Vegeu Figura 11-14).

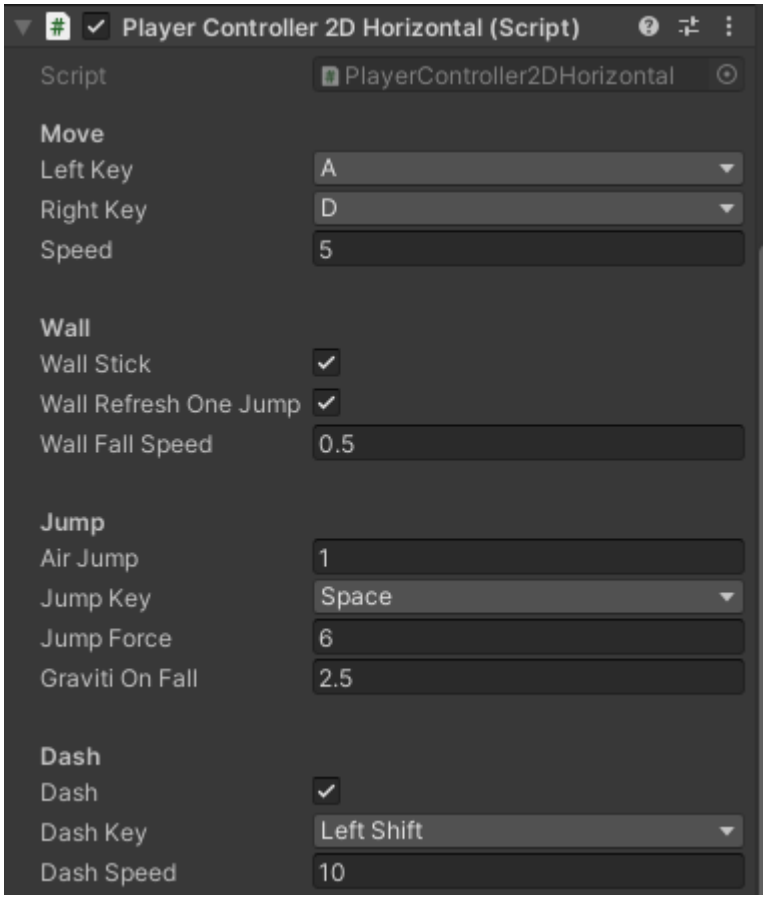


Figura 12-14 PlayerController2DHorizontal

Les del grup "Move" serveixen per editar el moviment base. Les de la categoria "Wall" canvien el comportament del personatge quan està a una paret. A l'apartat "Jump" tenim tot el necessari per a canviar les mecàniques de salt. Finalment, a la zona de "Dash" canviem els paràmetres relacionats amb l'esprint.

12.1.6 Exemple d'ús dels menús

El més important a l'hora de crear els menús és fer servir el MenuController, aquest té tot el necessari perquè funcionin els menús. A partir d'aquí, cada cop que creem un botó haurem d'assignar-li la funció que necessitem del controller. A més a més, si volguéssim fer que el botó amagués o mostrés altres interfícies o menús, podem fer servir l'script ChangeUI que ens permet amagar una sèrie d'interfícies i mostrar-ne unes altres.

A l'hora de tenir implementats els menús el més ràpid possible, hi ha un prefab amb tots els menús bàsics funcionals, de forma que simplement arrossegant aquest prefab a l'escena a la qual estiguem, ja tindrem els menús. L'única cosa que farà falta és activar els menús de perdre o guanyar quan s'hagin complert les condicions necessàries per mostrar-los (Figura 11-15).

Figura 12-15 Prefab MenuController

Cada un dels menús està en un prefab diferent per a facilitar l'edició, de forma que només canviant el prefab, s'aplicaran els canvis fets a tots els nivells del joc que tinguin el menú en qüestió.

En el cas en el qual vulguem crear els nostres menús des de zero, podem accedir a les funcions de la següent manera. Un cop creat el botó al qual li vulguem afegir alguna funcionalitat, li assignarem a l'apartat OnClick del botó, el MenuControler i podrem seleccionar quina de les funcions disponibles voldrem (Figures 11-16 i 11-17).

Figura 12-16 Event On Click MenuController

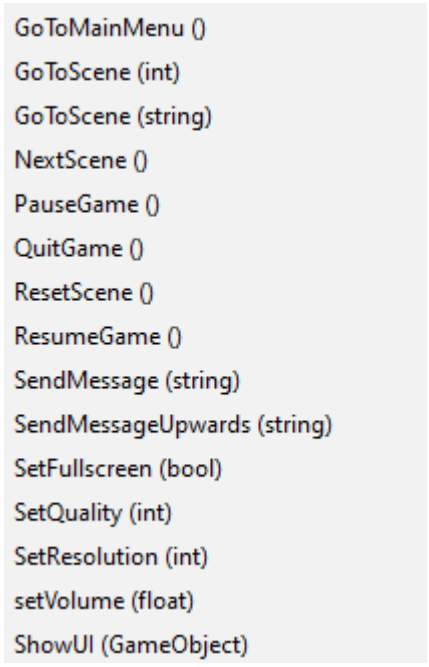


Figura 12-17 Funcions MenuController

ChangeUI

L'script ChangeUI s'encarrega d'amagar un conjunt d'interfícies i ensenyar-ne unes altres. Es pot activar fent servir botons de forma que permetin canviar entre diferents interfícies o menús. Per fer-lo servir simplement fa falta donar com a paràmetre les interfícies que vulguem amagar o ensenyar i quan vulguem activar aquest canvi cridarem la funció Activate.

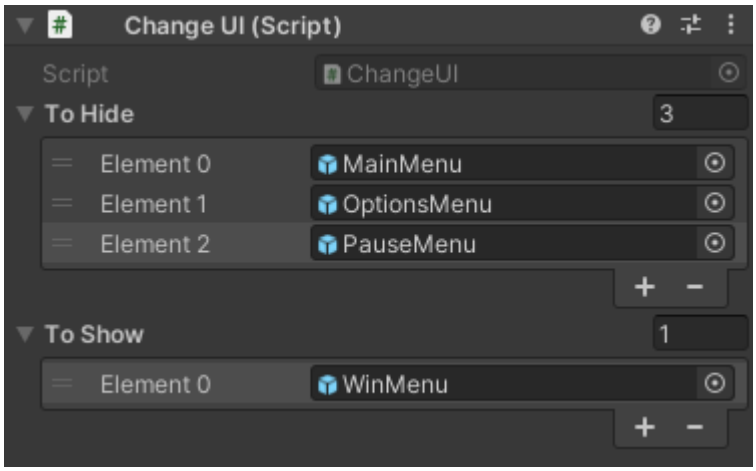


Figura 12-18 Change UI

12.1.7 Exemple d'ús de l'Inventari genèric

Per a veure què hauria de fer l'usuari per crear el sistema d'inventari a partir de l'inventari genèric, explicarem pas a pas tots els passos necessaris per a fer-lo funcionar. Primer de tot, crearem un nou inventari, crearem un objecte Empty a l'escena d'Unity i afegir-li un Canvas com a fill (Figura 11-19).

Figura 12-19 Creació jerarquia inventari

Ara hem d'afegir l'InventoryDisplay, tenim dues opcions disponibles, la més fàcil i ràpida és posar-li l'script de RectangleInventoryDisplay. Aquest script crea un inventari a partir del nombre de ranures d'amplada i alçada, la grandària de la ranura, l'espai entre ranures i un prefab ja creat d'un slot (Figura 11-20).

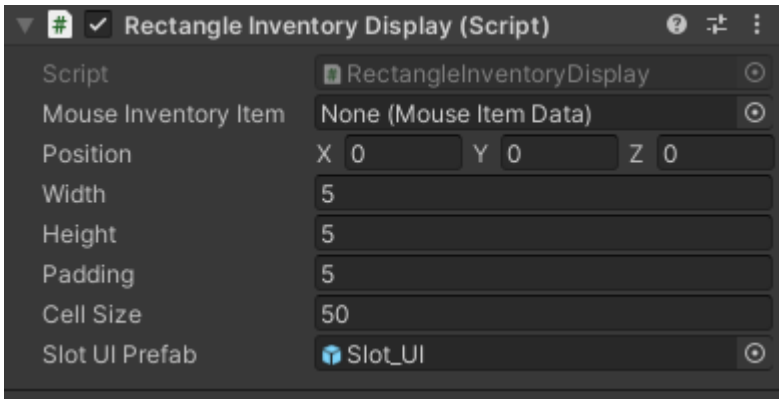


Figura 12-20 Funció Rectangle Inventory Display

La segona opció és fer servir l'script de GenericInventoryDisplay. En aquesta, en comptes d'aquests paràmetres que ja ens creen un inventari, haurem de posar tants prefabs d'un slot com ranures vulguem que tingui l'inventari, amb l'avantatge que els podem posar en la posició que vulguem (Figura 11-21).

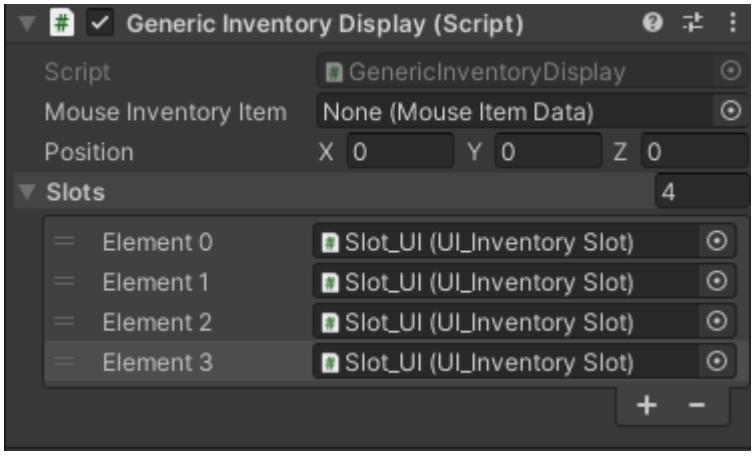


Figura 12-21 Generic Inventory Display

Un cop tenim l'inventari fet crearem l'inventari controller. Aquest ens permet veure els inventaris que vulguem. Simplement, crearem un Empty amb l'script d'InventoryController. Aquest ens demana dos GameObjects com a paràmetre: un serà per l'inventari del player i l'altre per a mostrar cofres o altres inventaris (Figura 11-22).

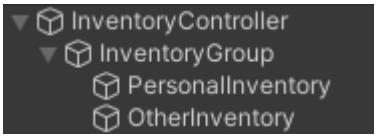


Figura 12-22 Jerarquia InventoryController

Aquests dos GameObjects seran un canvas amb l'script InventoryShower, que demana un GameObject com a paràmetre, que serà el canvas que hem creat amb l'script de InventoriDisplay (Figura 11-22).

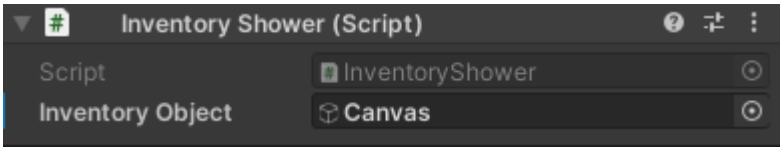


Figura 12-23 Inventory Shower

Finalment per a poder moure ítems entre els inventaris i per un mateix inventari afegirem el MouseSlot_UI del qual hi ha un prefab fet. En el mateix InventoryController hi podem posar altres inventaris que es poden mostrar constantment com pot ser una hotbar (Figura 11-24).

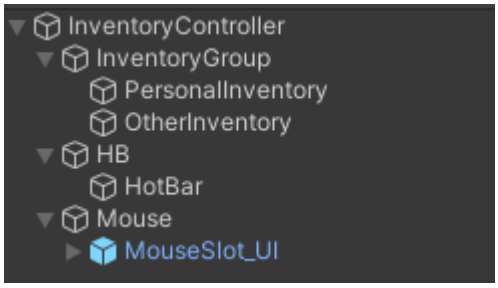


Figura 12-24 Altres inventaris a la jerarquia de InventoryController

Obrir inventaris segons la proximitat del player

Per a veure l'inventari així, com els inventaris dels cofres que estiguin al voltant del Player, necessitarem posar al player s'script de ChestOpener. Aquí li direm quin detector fer servir per a detectar els cofres de la zona, el taglist que identifica els cofres i la tecla que activarà la cerca de cofres i la funció de mostrar o amagar de l'InventoryController (Figura 11-25).

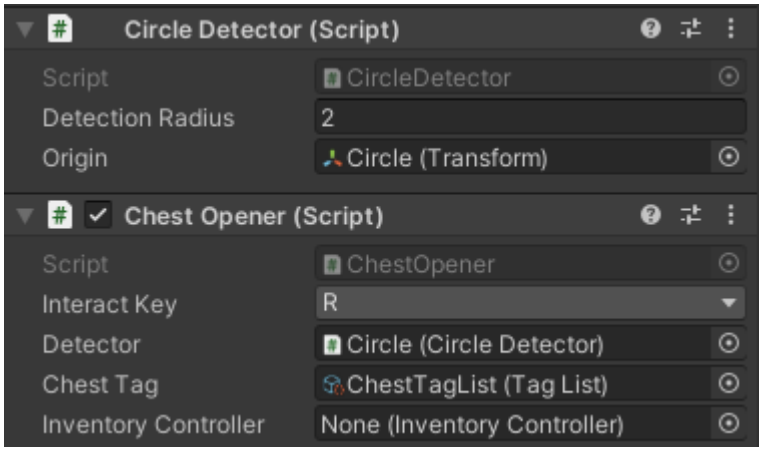


Figura 12-25 Chest Opener

Finalment, crearem els cofres. Per fer-ho creem un objecte amb un collider2d l'script Taggable amb la TagList que identifica el cofre i l'script Chest on li posarem com a paràmetre el GameObject que té l'script InventoryDisplay, cal recordar que, a no ser que es vulgui que dos cofres tinguin el mateix inventari, s'ha de crear un objecte amb InventoryDisplay per cada cofre (Figura 11-26).

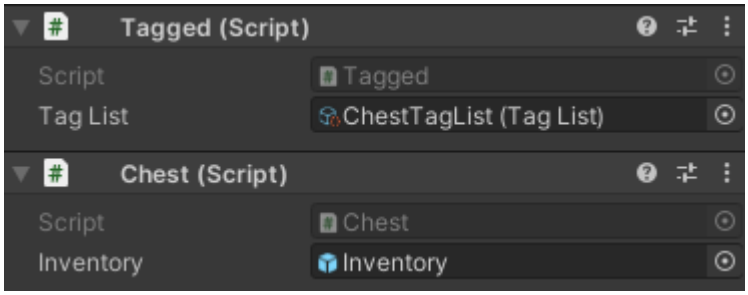


Figura 12-26 Chest

Crear ítems

Els ítems que es poden guardar a l'inventari s'han de crear. Per fer-ho s'ha creat un ScriptableObject que s'encarrega de crear-lo a partir de les variables que li donem. Només caldrà anar al desplegable que es genera en fer clic amb el botó dret del ratolí dins de qualsevol carpeta d'assets de Unity i anar al desplegable d'Object i triar Ítem (Figura 11-27).

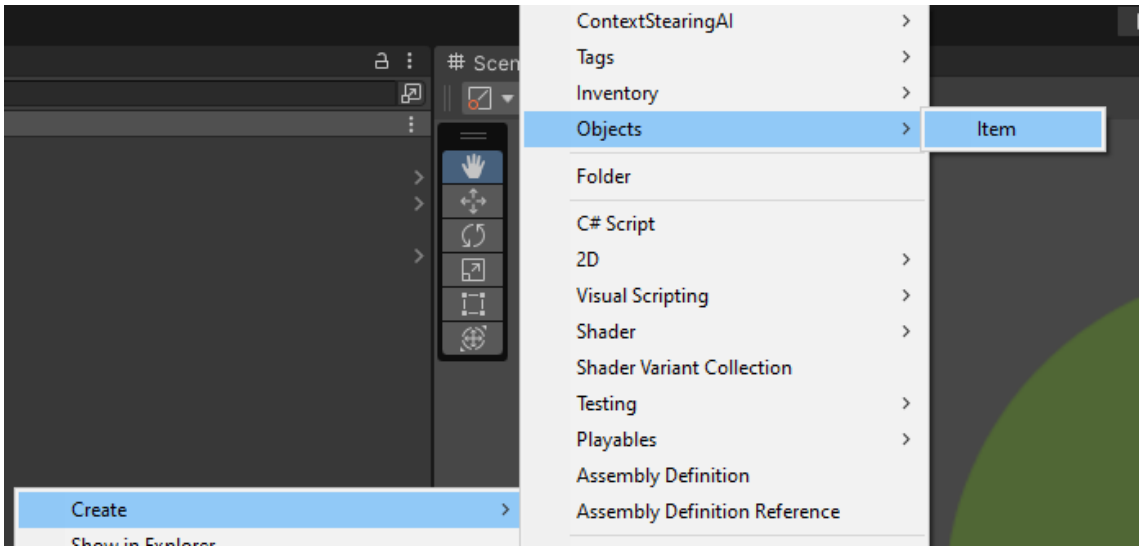


Figura 12-27 Desplegable Ítem

Aquí posarem el nom i descripció de l'Ítem. També li podem relacionar l'ítem amb un GameObject com a paràmetre per si ho necessitéssim per algun motiu. Aquest pas és opcional, però servirà a l'hora de aconseguir informació sobre l'ítem que hi ha a l'inventar (Figura 11-28).

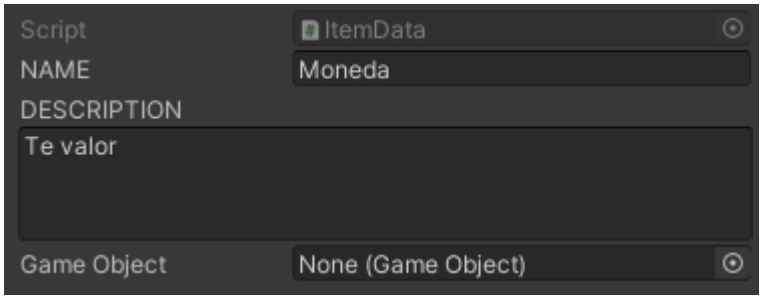


Figura 12-28 ÍtemData

Després creem la informació necessària perquè l'ítem pugui anar a un slot. Per fer-ho tornem a anar al desplegable, però aquesta vegada a l'opció Inventory i seleccionarem l'opció d'ÍtemData. Aquí afegim com a paràmetre la quantitat màxima que hi pot haver d'aquest ítem a una ranura i la icona que el representarà a l'inventari, en cas d'haver creat un ítem anteriorment també li podem posar (Figura 11-29).

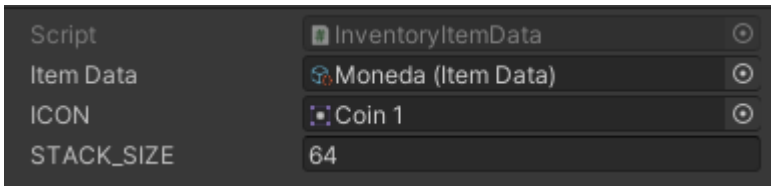


Figura 12-29 InventoryÍtemData

Recollir ítems del mapa

L'última funcionalitat que pot tenir l'inventari és posar-hi ítems que hi hagi per terra. Per crear l'ítem que estarà per terra crearem un Empty amb un SptriteRenderer un Colider2D que posarem com a trigger i l'script Collectable (Figura 11-30).

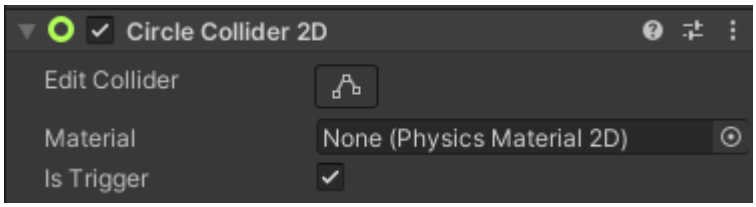


Figura 12-30 CircleController2D

En aquest script podem posar l'InventoryÍtemData que hem creat abans i la quantitat d'aquest ítem que hi ha. També podem dir-li si volem que, en cas que el jugador no tingui espai suficient per recollir tots els ítems, desaparegui l'objecte o no (Figura 11-31).

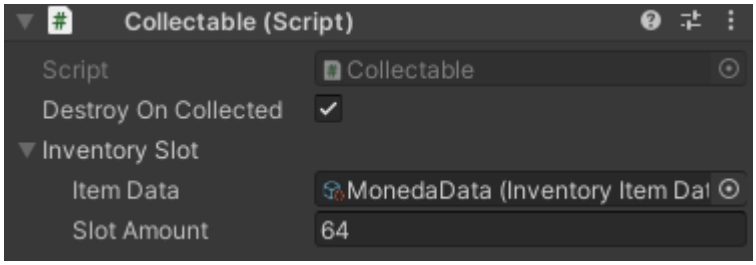


Figura 12-31 Collectable

Finalment, li posarem a l'encarregat de recollir l'objecte l'script Collector on li direm quin InventoryDisplay rebrà els ítems (Figura 11-32).

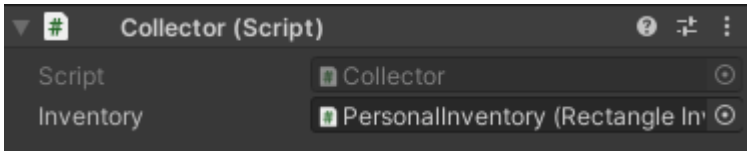


Figura 12-32 Collector

12.1.8 Exemple d'ús del sistema d'atac

Ara anem a veure pas a pas el que hem de fer per a crear el nostre sistema d'atac i com podrem personalitzar-lo.

Crear tipus de dany

Com s'ha mencionat anteriorment, hi ha diferents tipus de dany. A la majoria de jocs de rol podem trobar tipus de dany com pot ser el dany elemental que seria aigua, foc, etcètera, també hi ha dany per tall, dany perforant o dany per cop.

A l'haver-hi tants tipus de dany i havent-hi la possibilitat que un dany sigui de dos tipus a l'hora, com pot ser una fletxa de foc, hem fet un sistema que permet totes les combinacions de dany que vulguem. Això s'ha aconseguit amb dues formes d'entrada de dades. La primera seria emplenant una llista de tipus de dany. D'aquesta forma només caldria apuntar en aquesta llista el tipus de dany que farem. L'altra forma seria al revés, apuntarem els tipus de dany que no fem. Això pot anar molt bé a l'hora de crear defenses, ja que podem dir que protegeixen de tota mena de mal excepte el foc, per posar un exemple. Això ho farem de la següent manera: només caldrà anar al desplegable que es genera en fer clic amb botó dret del ratolí dins de qualsevol carpeta d'assets de Unity i anar al desplegable d'Attack i triar AttackTypeCombinations. En aquest element, primer de tot seleccionarem quina forma d'entrada de dades voldrem amb el booleà isGeneral. En cas de ser general podrem dir quines són les excepcions a NOTtypeOfAttack. Si no és general, posarem el tipus a TypeOfAttack (Figura 11-33).

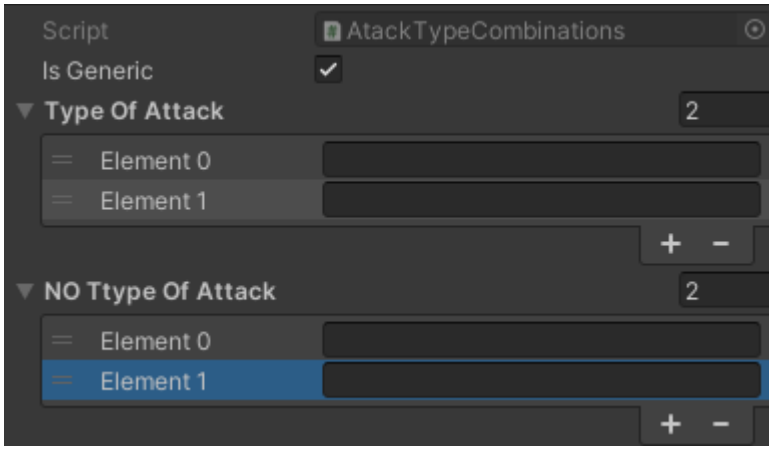


Figura 12-33 AttackTypeCombinations

Crear atacs

Un cop ja tenim el tipus d'atac que farem, podem crear l'atac. Per això anirem a la pestanya d'Attack on hem trobat l'AttackTypeCombinations, però ara triarem l'opció Attack.

A l'Attack tenim diverses variables de personalització. Primer de tot hi posarem el tipus d'atac que hem creat anteriorment. Després tenim les opcions de la quantitat de mal que fa. Aquestes són Damage, per al dany directe, i DamagePercent, per a fer mal com a percentatge de la vida del que rep el mal. El següent nivell de personalització és per si el dany és instantani o es va repetint de tant en tant. Si marquem la casella nomenada isInstant, no fa falta que toquem cap més variable, però en cas de no voler que sigui instantani, tenim dues opcions més. La primera seria durant quant temps volem que duri aquest dany no instantani, i la segona serà la freqüència a la qual rebrem dany durant aquest temps (Figura 11-34).

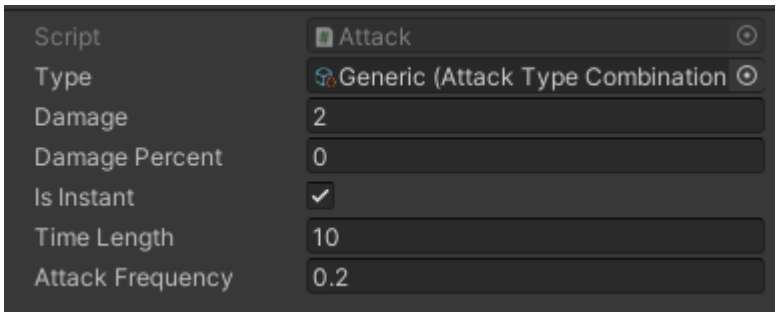


Figura 12-34 Attack

Crear defensa

Ara veurem com ens podem defensar d'aquests atacs. Per a defensar-nos podem crear defenses. Aquestes es creen de forma molt similar als atacs. Primer de tot anirem a la pestanya d'Attack i triarem l'opció Defense. En aquest objecte hi trobem tres variables, la primera hi posarem l'AttackTypeCombinations que ens servirà per saber contra quins atacs ens protegim. Les dues següents variables són el percentatge de mal que podem parar. La primera indica el mal directe que parem i la segona el percentatge de mal que parem (Figura 11-35).

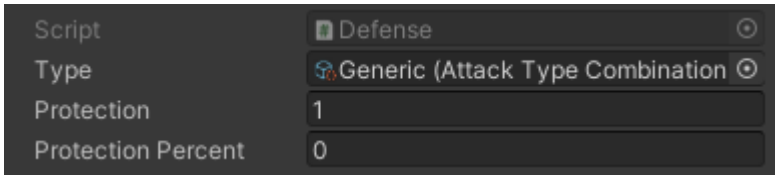


Figura 12-35 Defense

Crear atacant

Un cop hem creat tots els elements anteriors ja podem començar a utilitzar-los a la nostra escena. Aquí el que farem serà crear un atacant, tenim dues formes de crear-los.

Temim l'atacant amb detecció d'àrea, que atacarà quan un objecte que pugui ser atacat quan entri al trigger que té assignat. Per a crear aquest tipus d'atacant crearem un Empty amb un Collider2D i marcarem l'opció de trigger. Després, li posem l'script d'AttackOnTrigger, i com a variables li posarem el tipus d'atac que volem que faci. La segona variable que té aquesta classe és pel cas en què el dany que fem no sigui instantani i vulguem que en sortir de la zona que fa mal, el mal pari. També tenim un UnityEvent al que li podem assignar funcions que vulguem que s'activin quan s'ataca a algú. Un exemple d'ús seria en el cas que hi hagués una zona on el terra fos àcid i volguéssim que, quan algú s'hi fes mal, apareguessin esquitxades a l'àcid (Figura 11-36).

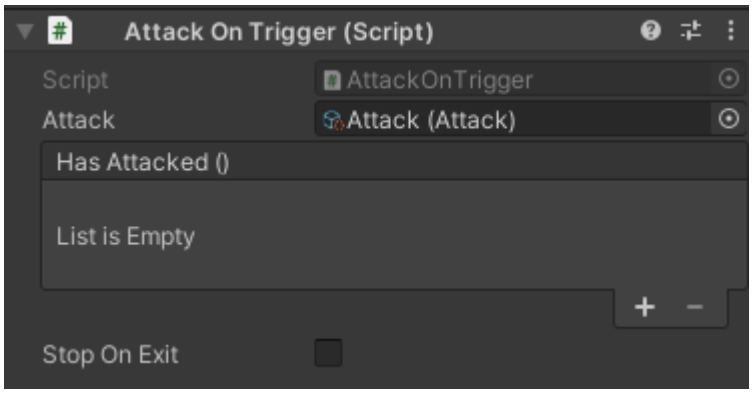


Figura 12-36 Attack On Trigger

L'altra opció que tenim per crear un atacant és crear un AttackOnTrigger. Aquest té les mateixes qualitats que l'anterior, amb la diferència que no necessita un Collider2D, però necessita un Detector per a saber si hi ha algú a l'àrea d'atac i a més a més l'hauríem d'activar des d'un altre lloc (Figura 11-37).

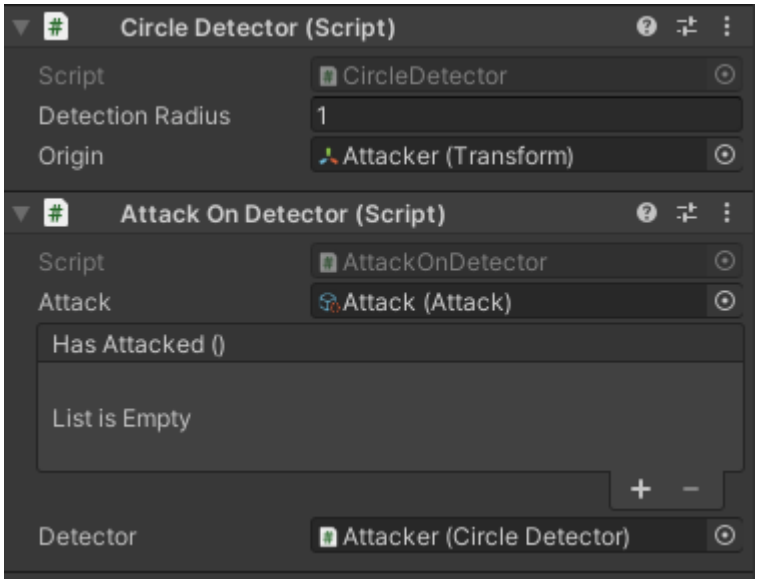


Figura 12-37 CircleDetector i Attack On Detector

Crear objecte que rep l'atac

Ja només ens falta crear l'objecte que rep els atacs. Per fer-ho simplement hem d'agafar un objecte com podria ser un Empty i assignar-li l'script Attackable. Aquest només té tres variables, la vida màxima, la vida actual i les defenses. Igual que l'atacant, aquest també té un UnityEvent que s'activa en cas de rebre mal, això ens pot servir per activar partícules de sang o un text amb la vida que ha perdut (Figura 11-38).

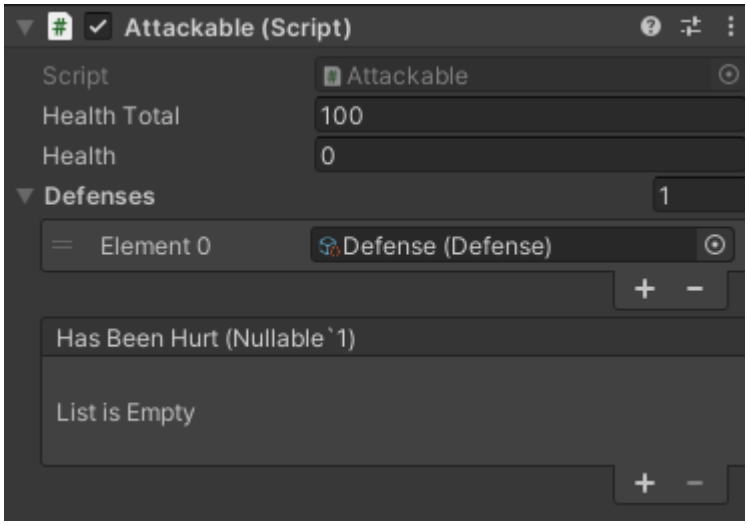


Figura 12-38 Attackable

12.1.9 Exemple d'ús del sistema de diàleg

Per a veure què hauria de fer l'usuari per crear el sistema de diàleg, explicarem tots els passos necessaris per a crear el diàleg, així com la interfície on és mostrarà i els objectes que serviran per activar aquest diàleg.

Primer de tot crearem el diàleg. Per fer-ho caldrà anar al desplegable que es genera en fer clic amb el botó dret del ratolí dins de qualsevol carpeta d'assets d'Unity i anar al desplegable Dialogue, on escollirem l'opció Dialogue (Figura 11-39).

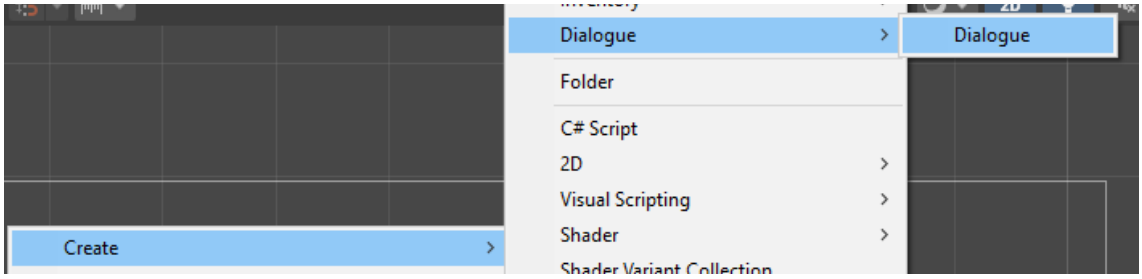


Figura 12-39 Desplegable Dialogue

A continuació emplenarem el diàleg. Dins del diàleg trobem una llista anomenada Content. Si volem afegir un nou paràgraf simplement farem clic al signe "+" de la llista (Figura 11-40).

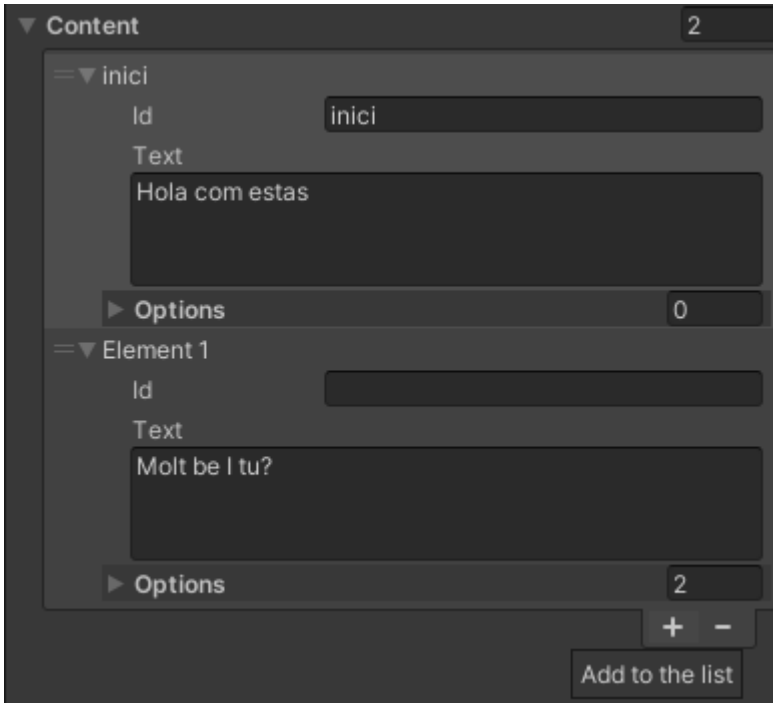


Figura 12-40 Creació Dialogue

Cada un dels elements de la llista els anomanem paràgrafs. Dins d'aquests paràgrafs hi ha tota la informació necessària per a que el DialogueManager creï el diàleg.

A cada paràgraf veurem tres camps, el primer és l'identificador, que només l'hem de fer servir en cas de que vulguem que s'hi pugui accedir des d'un altre paràgraf. Després tenim el text que és mostrarà com a diàleg. I finalment tenim una llista d'opcions. Aquestes serviran per a definir quins botons posar al paràgraf.

Cada botó té un text a mostrar, un booleà per a dir si va al següent paràgraf o a un en concret i el tercer és per dir a quin paràgraf en concret vol anar, aquí s'ha de posar l'identificador del paràgraf (Figura 11-41).

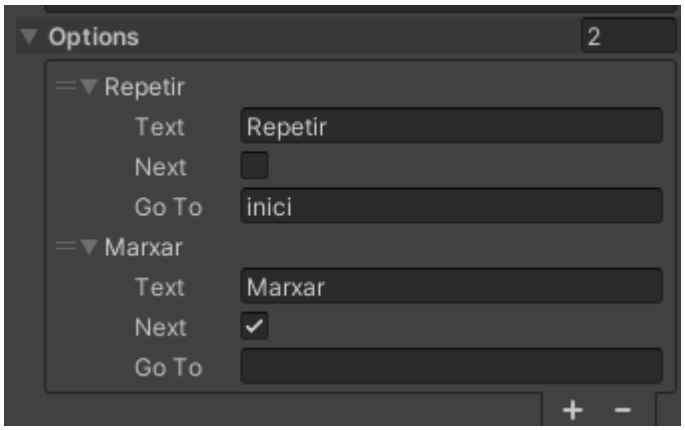


Figura 12-41 Creació opcions paràgraf

Un cop creat un diàleg crearem a l'escena el DialogueManager. Aquí només caldrà crear un objecte de tipus Text i un de tipus Canvas. El text mostrarà el contingut del diàleg i el Canvas servirá de pare per als botons que crearem per a triar entre les opcions. Per temes estètics, és recomanable utilitzar altres Canvas per a organitzar la posició d'aquests dos elements a la pantalla. A més a més, també caldria posar en el Canvas dels botons algun script de tipus Layout. Aquests ja venen amb Unity i permeten mostrar els fills d'un Canvas o similar de forma que quedin tots organitzats en vertical, horitzontal o en una graella (Figura 11-42).

Figura 12-42 Jerarquia Canvas DialogueManager

Si volguéssim fer cap canvi al text o a la distribució d'elements es faria directament per aquí.

Ara farà falta que creem uns prefabs amb els botons que farem servir. En fan falta dos, un per a quan no haguem posat opcions que servirà per anar al següent paràgraf i l'altra que serà on posarem el text de les opcions. Tots els canvis estètics i de tipus de text que vulguem fer als botons els farem per aquí.

Finalment, ho ajuntarem tot com a paràmetres del DialogueManager. El Text i el Button Canvas són els elements dels quals hem parlat abans. Els Button prefab són els prefabs dels botons i el Global Cavas és el Cavas que conté tot el que es mostrà quan obrim el diàleg. D'aquesta forma el Manager pot mostrar i amagar tota la interfície d'usuari de diàleg fàcilment.

També té la possibilitat d'assignar una tecla per passar pàgina en cas que no hi hagi opcions (Figura 11-43).

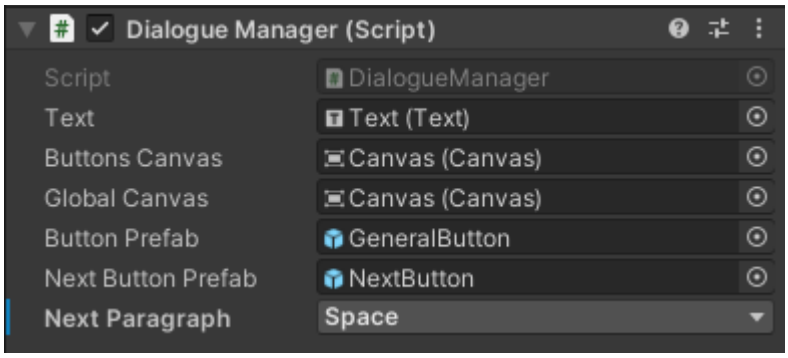


Figura 12-43 Dialogue Manager

Finalment crearem els objectes que ens mostraran els diàlegs. Aquests objectes tindran l'script DialogueBox, al qual li posarem per paràmetre el diàleg que hem creat al principi (Figura 11-44).

Figura 12-44 Dialog Box

Després només caldrà fer servir qualsevol forma d'interacció que vulguem per a executar la funció ShowDialogue del Dialogue box. Això es pot fer amb un Interactuble o un Triggerer2D o qualsevol altra forma que vulguem (Figura 11-45).

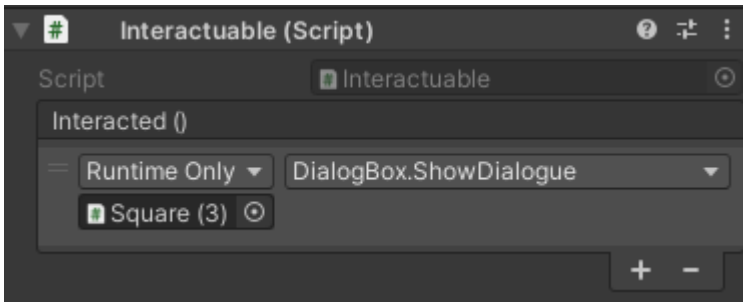


Figura 12-45 Interactuable

12.1.10 Exemple d'ús IA

Per a fer servir aquesta Context Stearing AI s'han de fer els següents passos: Cal anar al desplegable que es genera en fer clic amb el botó dret del ratolí dins de qualsevol carpeta d'assets d'Unity. Aquí es pot trobar tot el necessari per a la creació de la IA (Figura 11-46).

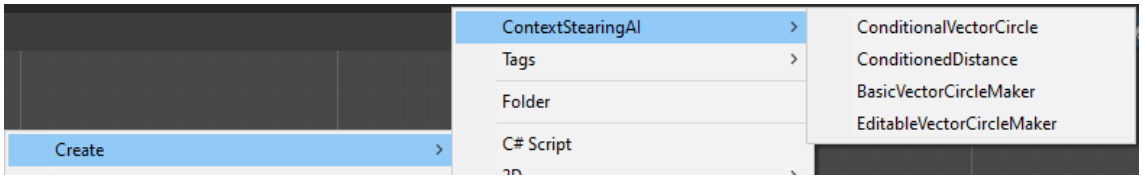


Figura 12-46 Desplegable ContextStearingAI

Es pot començar per qualsevol dels VectorCircleMakers. Aquests tenen la funció de crear cercles de vectors que s'utilitzaran per indicar a la IA en quines direccions reaccionar.

L'EditableVectorCircleMaker és el més directe de fer servir, ja que simplement se li dona a cada direcció el valor que es cregui convenient (Figura 11-47).

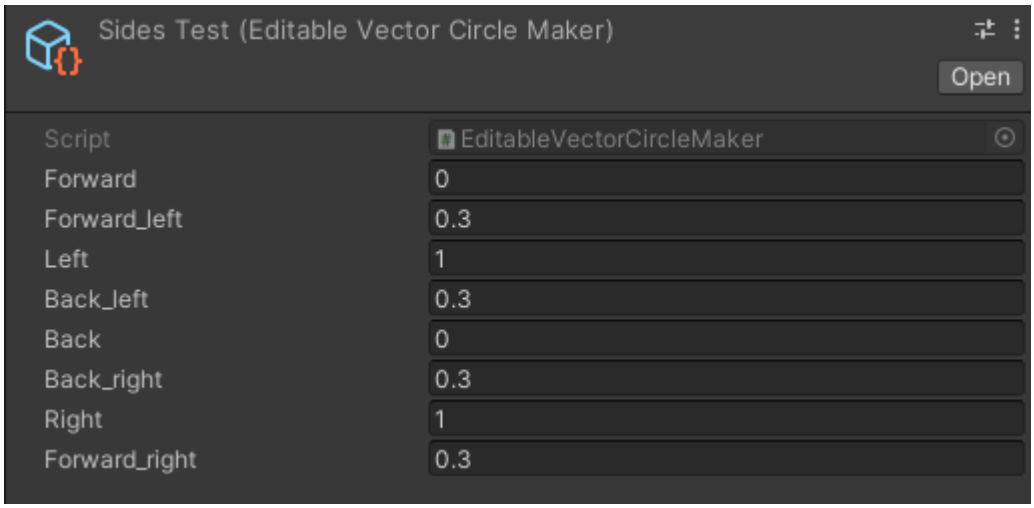


Figura 12-47 EditableVectorCircleMaker

El BasicVectorCircleMaker funciona diferent. S'han de triar vectors ja fets i combinar-los per fer el vector final que es vulgui. Es pot tirar el pes dels vectors i si s'afegeixen al vector final o es resten (Figura 11-48).

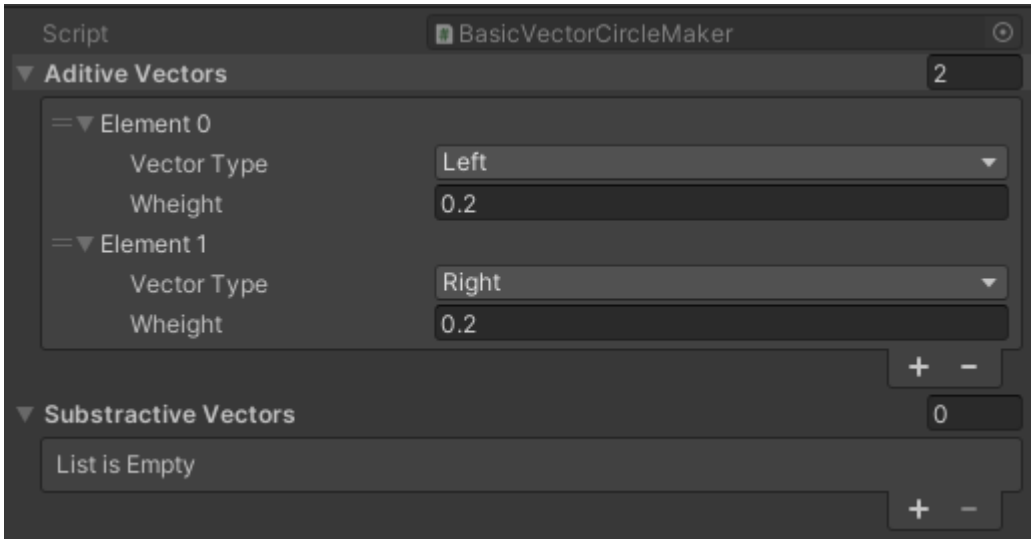


Figura 12-48 BasicVectorCircleMaker

Un cop tenim fets els vectors crearem les condicions anant al menú de la Figura 11-46 i generant una ConditionedDistance. Aquí triarem les condicions de l'activació de la reacció, així com la força de la reacció depenent de la distància. Primer hi assignarem una TagList amb les Tags que faran activar la reacció, després la distancia mínima i màxima d'activació i, finalment, com varia la resposta segons la distancia (Figura 11-49). Hi ha 5 opcions de reacció segons la distància:

- Constant – la reacció sempre té la força màxima.
- Linear Positiva – com més a prop, més forta la reacció de forma lineal.
- Lienar Negativa – com més lluny, més forta la reacció de forma lineal.
- Exponencial – com més a prop, més forta la reacció de forma exponencial.
- Exponencial inversa - com més lluny més forta la reacció de forma exponencial.

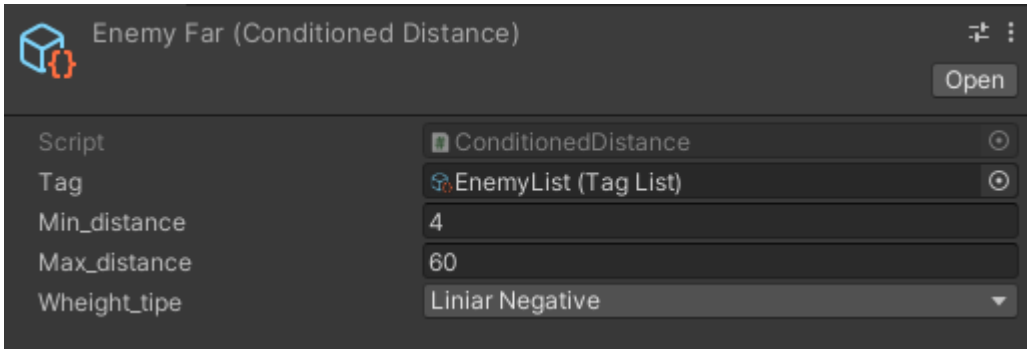


Figura 12-49 Enemy Far

Un cop fetes les condicions i els cercles de vectors, crearem el ConditionalVectorCircle que és el conjunt de la condició, la direcció de reacció i el tipus de reacció (Figura 11-50).

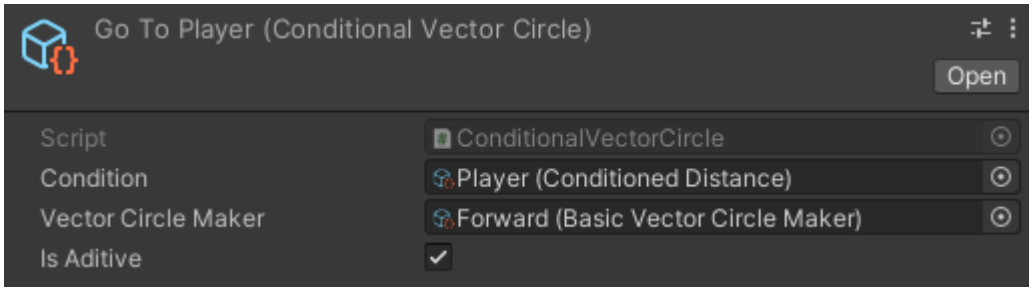


Figura 12-50 ConditionalVectorCircle

Finalment, afegirem la IA a un objecte. Per fer-ho afegirem un objecte de tipus empty a l'objecte que vulguem que tingui la IA. A aquest objecte li afegirem un colider2D de tipus trigger i l'script de la ContextStearingAI (Figura 11-51).

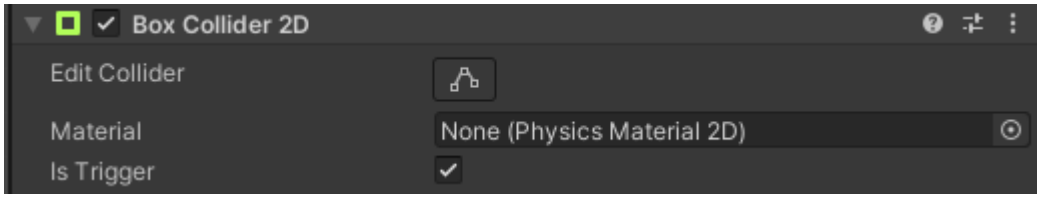


Figura 12-51 Box colider2D

A la AI li afegim tots els ConditionalVectorCircle que vulguem que tingui en compte, i un llindar perquè es quedi quiet quan la direcció a la qual hagi d'anar tingui molt poca força. Aquest llindar serveix per evitar que el personatge tremoli en alguns casos en els quals s'hauria de quedar quiet (Figura 11-52).

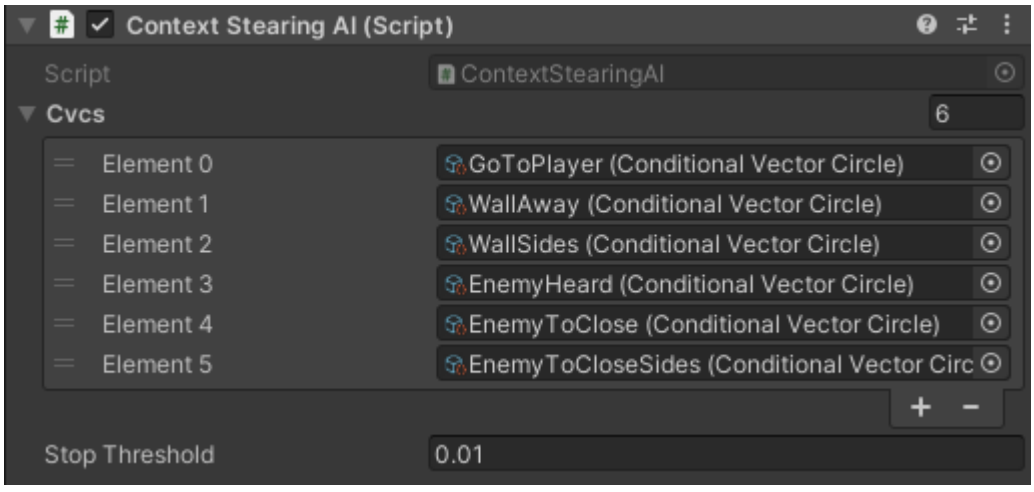


Figura 12-52 Context Stearing AI

Ara només ens falta fer que el personatge tingui en compte la IA a l'hora de moure's. Per fer una demostració de com es faria, s'ha fet un script que té com a variable la IA i

una velocitat i simplement mou el personatge cap a la direcció que en retorna la IA a la velocitat que li diem (Figura 11-52).

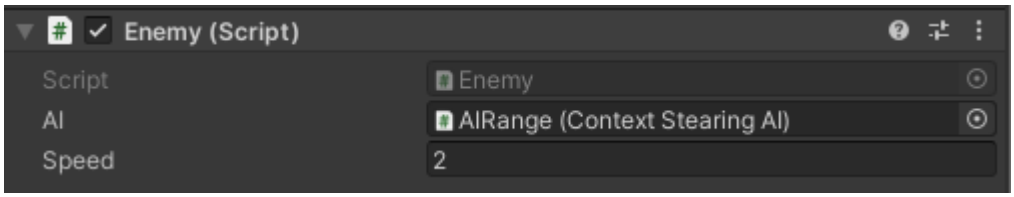


Figura 12-53 Enemy

A la jerarquia que ens quedaria: el pare que és l'objecte que fa servir la IA i el fill que és l'objecte que té el trigger que detecta que hi ha pel voltant i l'script de la IA.

12.2 Instal·lació

Cal descarregar el ZIP amb les llibreries. A continuació, cal descomprimir el fiter ZIP i arrossegar la carpeta a la carpeta d'assets d'Unity del projecte pel qual es vulgui fer servir la llibreria.