Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Video game development with Unity

Autor

Jiaqi Jin

La Laguna, 26 de Junio de 2020

D. **Jesús Miguel Torres Jorge**, con N.I.F. 43.826.207-Y profesor Contratado Doctor de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor.

CERTIFICA(N)

Que la presente memoria titulada:

“Video game development with Unity”

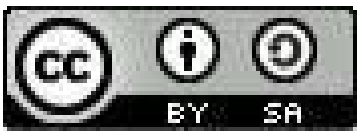
ha sido realizada bajo su dirección por D. **Jaqi Jin**, con N.I.E. X-8796447-M.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 26 de junio de 2020

Agradecimientos

Jesús Miguel Torres Jorge

Licencia



Resumen

El objetivo de este trabajo consiste en diseñar un prototipo básico de un videojuego RPG con el motor Unity. Integra todas aquellas herramientas necesarias proporcionadas por el motor para el desarrollo del juego, como herramientas de edición de animación, sistemas de iluminación, inteligencia artificial, etc. El prototipo incluye algunas mecánicas importantes relacionadas con la jugabilidad del juego.

Los juegos creados por Unity se pueden lanzar en todas las plataformas móviles, y todo el proceso de compilación y lanzamiento es completamente automático y puede optimizar automáticamente el rendimiento del juego al mejor posible.

Palabras clave: Videojuego, Unity, Motor del juego, Prototipo.

Abstract

The objective of this work is to design a basic prototype of a RPG video game with the Unity engine, this will integrate all those necessary tools provided by the engine for game development, such as animation editing tools, lighting systems, artificial intelligence, etc. Prototype will include some more important mechanics related to the gameplay of the game.

Games created by Unity can be launched on all mobile platforms, and the entire build and launch process is fully automatic and can automatically optimize game performance to the best possible state.

Keywords: prototype, Game engine, Unity, Games.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1 Unity3D

Unity3D es una herramienta integral de desarrollo de juegos multiplataforma desarrollada por Unity Technologies que le permite crear fácilmente contenido interactivo como videojuegos 3D, visualización arquitectónica, animación 3D en tiempo real, etc. Es un motor de juego profesional totalmente integrado.

1.1.1 Características del motor

- Interfaz de programación visual para completar varios trabajos de desarrollo, edición eficiente de scripts, desarrollo conveniente.
- Importación instantánea automática. Unity admite la mayoría de los modelos 3D, los huesos y las animaciones se importan directamente, y los materiales de textura se convierten automáticamente al formato U3D.
- Desarrollo multiplataforma y despliegue de obras con un solo clic;
- La capa inferior es compatible con OpenGL y DirectX 11, un motor de física simple y práctico, sistema de partículas de alta calidad, fácil de usar, efecto realista;
- La unidad tiene un excelente rendimiento, excelente eficiencia de desarrollo y ventajas extremadamente rentables;
- Apoye el desarrollo desde aplicaciones independientes hasta juegos en línea para jugadores múltiples a gran escala.

1.2 Áreas de aplicación

Juegos móviles, juegos en línea, juegos finales, juegos de realidad virtual, dispositivos portátiles, cajas de TV, etc.

1.3 Ventajas

- Simple y rápido: Con la interfaz de usuario simple de Unity, puede hacer cualquier trabajo. Esto le ahorra mucho tiempo y puede editarse de manera integral.
- potencia gráfica: Unity tiene una canalización de representación de gráficos altamente optimizada para DirectX y OpenGL.
- Importación de recursos: Unity admite todos los formatos de archivo principales y puede funcionar con la mayoría de las aplicaciones relacionadas.
- Unity le permite presentar su trabajo en múltiples plataformas.
- Unity hace que el software de plataforma de juegos más popular de la industria sea más fácil de desarrollar.
- Shader: El sistema de sombreado de Unity combina facilidad de uso, flexibilidad y alto rendimiento.

Capítulo 2
Mecánicas

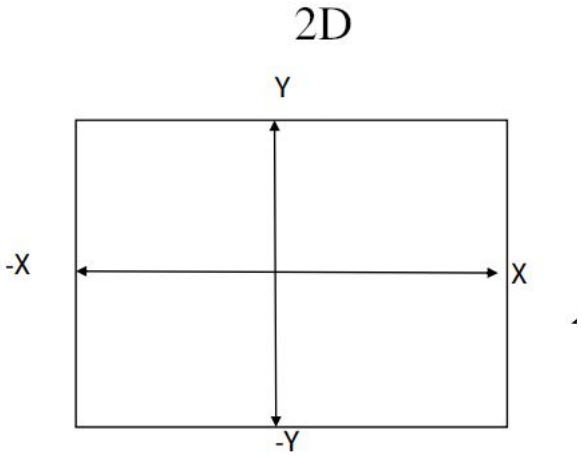
En el capítulo anterior se ha introducido la plataforma para el desarrollo del videojuego de este proyecto. Pero ¿cuál es el concepto del videojuego en sí?.
Los videojuegos al final son programas que crean un mundo digital de manera los elementos interactivos brindan a los jugadores de una forma experimentar, explorar e incluso vivir en un mundo virtual.

El desarrollo de toda las industrias de los videojuegos ha venido emparejado con el desarrollo del hardware, tanto en PC como en consolas con cada vez más prestaciones. Esto ha permitido el paso de los juegos 2D o 3D o, dentro de estas categorías, que se mejores aspectos como la calidad gráfica o la jugabilidad.

En los siguientes secciones van introduciendo los elementos necesarios tantos para la jugabilidad del juego como el desarrollo en sí mismo.

2.1 Control del jugador

En el motor Unity, existen múltiples formas para lograr el propósito del movimiento de los objetos, pero lo esencial es la modificación de las posiciones del objeto en cada frame.
En primer lugar, el personaje se mueve horizontalmente y verticalmente en el espacio “x” (horizontal) y “z” (vertical) sin tener cuenta la posición “y” del plano de coordenada. Las señales que recibe el personaje respecto a estos coordenadas son 0,1, y -1. Si hay un cero no hay señal de entrada y 1(-1) significa que hubo un señal de entrada.



El personaje se mueve respecto a estos señales en el plano de coordenadas, pero aquí surge un problema, cuando se trata de un espacio bidimensional, cómo podría calcular el punto en el espacio cuando se hay dos señales de entrada x e y respectivamente. Hay fórmulas como cálculo de la hipotenusa entre dos puntos en el sistema de coordenada, pero el resultado se sale del rango entre 0 y 1.Hay cálculos matemáticos que convierte un sistema de coordena cuadrada en forma circular que se denomina **Mapping a Square to a Circle,** gracias a este formula todo los puntos que calculamos en el sistema está dentro del rango entre 0 y 1 con los cálculos.

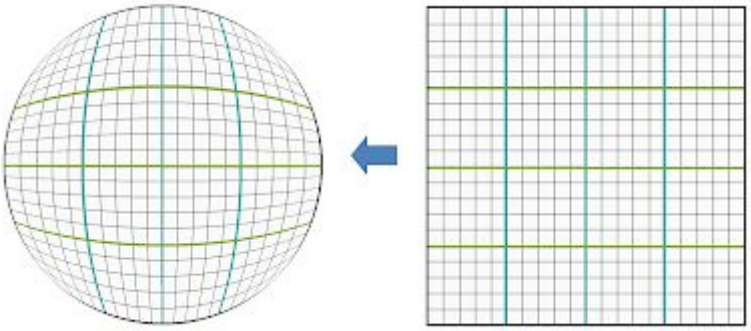
```
//Gradually changes a value towards a desired goal over time.
Dup = Mathf.SmoothDamp(Dup, targetDup, ref velocityDup, 0.1f);
Dright = Mathf.SmoothDamp(Dright, targetDright, ref velocityDright, 0.1f);

Vector2 tempVect = SquareToCircle(new Vector2(Dright, Dup));
float Dright2 = tempVect.x;
float Dup2 = tempVect.y;

Dmag = Mathf.Sqrt((Dup2 * Dup2) + (Dright2 * Dright2)); // movimiento del personajes
Dvec = Dright * transform.right + Dup * transform.forward; // giro del personakes
```

La fórmula es la siguiente :

$$x = \frac{1}{2}\sqrt{2 + 2\sqrt{2}\,u + u^2 - v^2} - \frac{1}{2}\sqrt{2 - 2\sqrt{2}\,u + u^2 - v^2}$$

$$y = \frac{1}{2}\sqrt{2 + 2\sqrt{2}\,v - u^2 + v^2} - \frac{1}{2}\sqrt{2 - 2\sqrt{2}\,v - u^2 + v^2}$$

donde convierte un espacio cuadrado en un espacio circular.



```
protected Vector2 SquareToCircle(Vector2 inputArea)
{
    Vector2 output = Vector2.zero;

    output.x = inputArea.x * Mathf.Sqrt(1 - (inputArea.y * inputArea.y) / 2.0f);
    output.y = inputArea.y * Mathf.Sqrt(1 - (inputArea.x * inputArea.x) / 2.0f);
    return output;
}
```

Como ha mencionado anteriormente, Unity te permite transportar el juego para múltiples plataformas como la consola, playstation etc. Todas aquellas señales de entrada son las mismas para el personaje dentro del juego, solo se modifica en el hardware de entrada (teclado, mandos de consola etc). Una forma para recopilar estos señales basta con crear una clase abstracta donde recoge todas las informaciones de entrada externa que recibe(correr , saltar) resulta muy útil en este caso.La finalidad de una clase abstracta es servir como clase base para otras clases(clase teclado, clase joystick etc). En Unity para crear una clase abstracto simplemente definimos la clase utilizando la palabra clave **abstract**, de esta forma todas las entrada se heredan de esa clase, así lograría generar una múltiple tipos de entradas externas.

2.1.1 Control por el teclado

Los señales de entrada se guarda en la clase “**MyButton**”donde recoge aquellos entradas por el teclado, han de definir tantas clases “**MyButton**” necesario para guarda toda las informaciones. Cada clase “**MyButton**” contiene varios atributos que pueden almacenar diferentes tiempos del juego (Ispressing,OnPress ...) cuando el jugador haya pulsado una tecla, luego estos atributos tiene un uso posteriores en cuanto a los movimientos del jugador, control de animaciones etc, ya que muchos de los acciones que se producen dentro de la escena depende de los señales de las entradas por el teclado.

```
public MyButton buttonA = new MyButton();
public MyButton buttonB = new MyButton();
public MyButton buttonC = new MyButton();
public MyButton buttonD = new MyButton();
```

2.2 Animaciones

Unity tiene un sistema de animación excelente y sofisticado llamado Mecanim. Mecanim proporciona, un flujo de trabajo y configuraciones de animaciones fácil para todos los elementos de Unity incluyendo objetos, personajes, y propiedades.

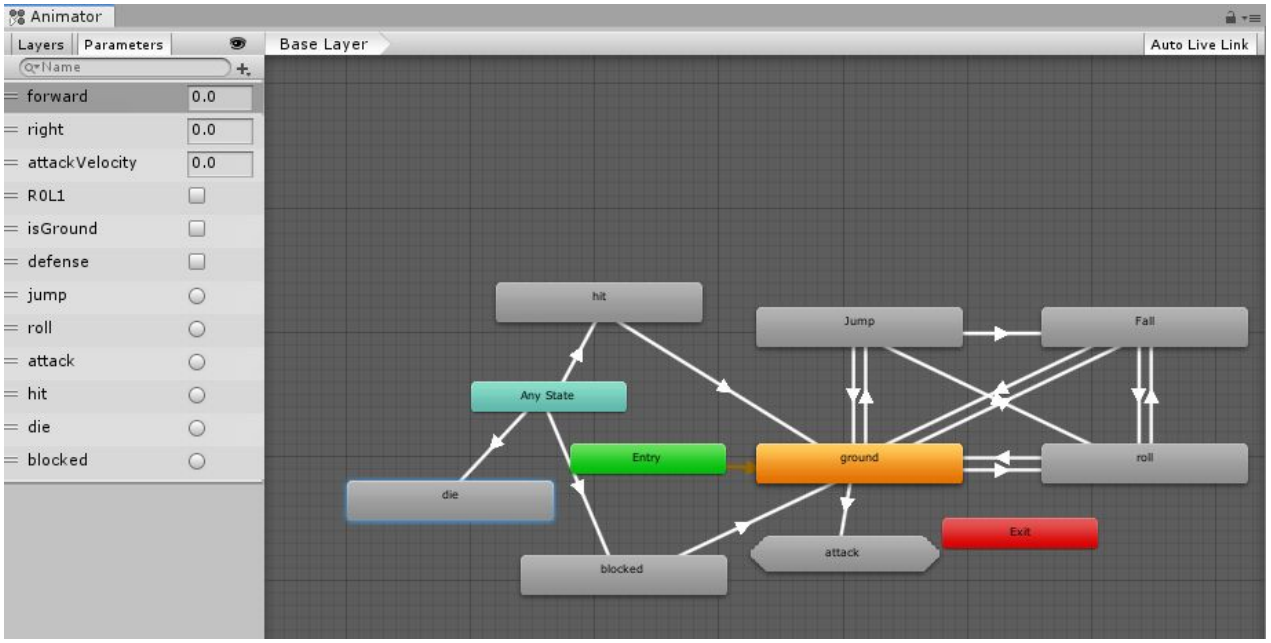
Flujo de trabajo de la animación es un sistema de animación se basa en el concepto de clips de animación, que contienen información sobre cómo ciertos objetos deberían cambiar su posición, rotación u otras propiedades con el tiempo. Cada clip puede considerarse como una sola grabación lineal. Los clips de animación se organizan en un sistema estructurado similar a un diagrama de flujo llamado Animator Controller. El Animator Controller actúa como una "Máquina de estado" que realiza un seguimiento de qué clip debería estar reproduciéndose actualmente y cuándo las animaciones deberían cambiar o combinarse.

Un simple controlador de animador podría solo contener uno o dos clips, por ejemplo para controlar un encendido de girar y rebotar, o para animar una puerta que se abre y se cierra. Un controlador avanzado más avanzado puede contener docenas de animaciones humanoides para todas las acciones de los personajes principales, y puede mezclar entre múltiples clips al mismo tiempo para proporcionar un movimiento fluido a medida que el jugador se mueve alrededor de la escena.

Cada una de estas piezas, los clips de animación, el controlador de animador y el avatar, se unen en un GameObject a través del componente Animator. Este componente tiene una referencia a un Controlador Animator y, si es necesario, al Avatar para este modelo. El controlador Animator, a su vez, contiene las referencias a los clips de animación que utiliza.

Los clips de animación son uno de los elementos fundamentales para el sistema de animaciones de Unity. Unity puede importar animaciones desde fuentes externas, y ofrece la habilidad de crear clips de animación desde cero dentro del editor usando la ventana de Animación.

Ejemplo de transición del animación :



Gracias a este sistema de flujo de animaciones, podemos crear diferentes estados de animaciones y controlar ellos mediantes los parámetros que previamentes se ha creado.
El personaje se incorpora múltiples estados de animaciones , dependiendo la entrada por el teclado, el flujo de animaciones van intercambiando. Además podemos hacer uso de animaciones para cambiar o producir diferentes comportamiento en las propiedades del personaje(señales de entrada , cambios en cuanto a las velocidades del personaje etc), o limpiar algunos señales de los parámetros del animator ya que la señal “Trigger” se superpone .

Ejemplo de códigos cuando queremos cambiar algunos parámetros en el momento de animación :


```
public void OnJumpEnter()
{
    pi.inputEnable = false;
    lockMoving = true;
    thrustVect = new Vector3(0, jumpVelocity, 0);
    trackDir = true;
}
```

2.2 Corrección de la Animación

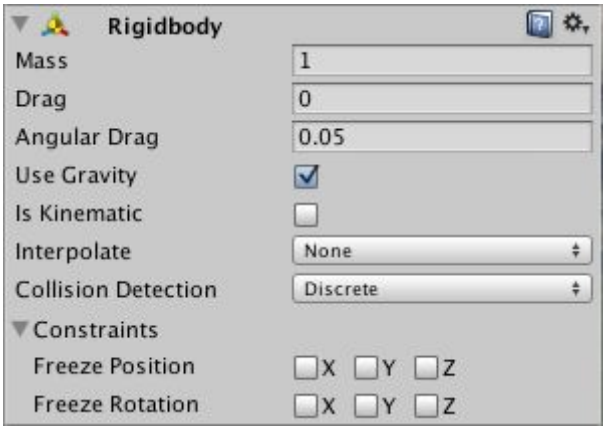
Muchas veces las animaciones no es deseados debería acudir a las artistas para que lo solucione o el cuerpo del personajes al reproducirse una secuencia de animación no obtiene un efecto deseados(el modelo importado FBX,COLLADA, etc al tipo de animación

Humanoide). Una forma de resolver este problema es la modificación de la estructura ósea existente del Avatar(sus posiciones) llamando a la función "**Animator.SetBoneLocalRotation**"(cambia la rotación local de un hueso humano) cuando se reproduce cierto tipos de animaciones.

```
Transform leftLoweArm = anim.GetBoneTransform(HumanBodyBones.LeftLowerArm);
leftLoweArm.localEulerAngles += vect;
anim.SetBoneLocalRotation(HumanBodyBones.LeftLowerArm, Quaternion.Euler(leftLoweArm.localEulerAngles));
```

2.3 RigidBody y Collider

Unity3D contiene un motor física "PhysX" incorporado con un alto grado de simulación física. Pueden agregar propiedades físicas reales al objeto para calcular la colisión del objeto. Los Rigidbodies le permite a sus GameObjects(objetos de la escena) actuar bajo el control de la física. El Rigidbody puede recibir fuerza y torque para hacer que sus objetos se muevan en una manera realista.



Los componentes **Collider** definen la forma de un objeto para los propósitos de colisiones físicas. Los colliders más simples (y menos intensivos al procesador) son los llamados primitive (primitivos) tipos de collider. En 3D, estos son Box Collider, Sphere Collider y Capsule Collider. En 2D, se utilizar el Box Collider 2D y Circle Collider 2D.

Un collider configurado como Trigger (utilizando la propiedad Is Trigger) no se comporta como un objeto sólido y simplemente le permitirá a otros colliders pasar a través de él. Cuando "IsTrigger" = false, el colisionador se activa de acuerdo con el motor física la colisión ocurre. Cuando "IsTrigger" = true, la colisión es ignorado por el motor de física y no tiene efecto de colisión. La detección requiere que el cuerpo rígido adicional se designe como un colisionador de cuerpo rígido. Cuando un collider entra su espacio, un trigger va a llamar la función OnTriggerEnter en los scripts del trigger del objeto.

2.3.1 Sensor

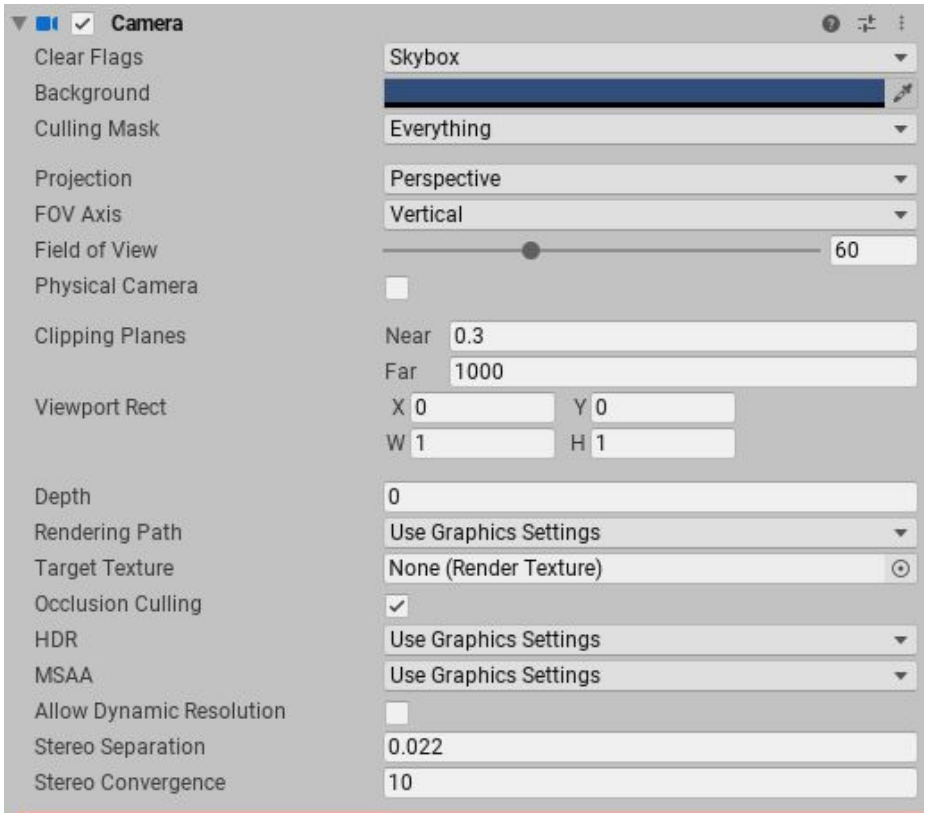
Con la introducido el uso de Collider, se puede crear diferentes tipos de sensores a los objetos de la escena gracias a este sistema física, como agregando el componente de colisión al personaje cuando entra contacto con el plano de la escena y poder mover sobre él sin atravesar el plano. También hace el uso para detectar si el personaje está en el suelo o en el aire mediante el la llamada a la función "**Physics.OverlapCapsule**", comprobando el estado de la cápsula(collider) dada con el mundo de la física y devuelve todos los colisionadores superpuestos. Otros usos de los collider que puede dar, como sí generan algunas colisiones con algunos tipos de armas en la escena o si previamente ha

entrado contacto con algunos de los NPC del juego etc, cada unos de ellos producen un comportamiento diferentes sobre el personaje.

```
point1 = transform.position + transform.up * (radius - offset);
point2 = transform.position + transform.up * (capCol.height - offset) - transform.up * radius;

Collider[] collision = Physics.OverlapCapsule(point1, point2, radius , LayerMask.GetMask("Ground"));
if (collision.Length != 0)
{
    //print("collision");
    SendMessageUpwards("IsGround");
}
else
{
    SendMessageUpwards("IsNotGround");
}
```

2.4 Cámara

La cámara representan como el usuario va a ver el juego. Un juego puede ser en primera persona, en tercera, puede tener una cámara fija o realizar un travelling a través del escenario cómo va a ser en este caso, ir persiguiendo al personaje.Al personalizar y manipular las cámaras, puedes hacer que la presentación de tu juego sea realmente única. Unity admite diferentes rutas de representación. Debes elegir cuál usar según el contenido de tu juego y la plataforma / hardware de destino. Las diferentes rutas de representación tienen diferentes características y características de rendimiento que afectan principalmente a las luces y las sombras. La ruta de representación utilizada por su proyecto se elige en la configuración del reproductor. Además, puede anularlo para cada cámara.



Las cámaras son esenciales para mostrar su juego al jugador. Se pueden personalizar, crear scripts o progenitores para lograr casi cualquier tipo de efecto imaginable. Para un juego de rompecabezas, puede mantener la cámara estática para una vista completa del rompecabezas. Para un juego de carreras, es probable que la cámara siga al vehículo de tu jugador.

Para crear una visualización más cercano al estilo RPG, usando un poco de trucos para crear un efecto del movimiento más ágil. La cámara se mueve verticalmente respecto a la escena y el personaje se rota verticalmente respecto a la escena restringiendo algunos ángulos, de esta forma se lograría un mejor efecto en cuanto a las rotaciones de las vistas del personaje. Para conseguir el seguimiento de la cámara al personaje de una forma más suavizado, se cambian gradualmente el vector de la posición de la cámara hacia al personaje deseada con el tiempo.

```
Vector3 tempModelEuler = model.transform.eulerAngles;

player.transform.Rotate(Vector3.up, pi.Jright * horizontalSpeed * Time.fixedDeltaTime);

tempEulex -= pi.Jup * verticalSpeed * Time.fixedDeltaTime; // se resta ya que se mueve d
tempEulex = Mathf.Clamp(tempEulex, -40, 30); // maximo angulo de rotacion
camaraHandler.transform.localEulerAngles = new Vector3(tempEulex, 0, 0);
// asiganr el euler angulo de ese momento del update
model.transform.eulerAngles = tempModelEuler;
```

```
//asigno las posicion del camaraPos al posiicon de la camara main
camara.transform.position = Vector3.SmoothDamp(camara.transform.position, transform.position, ref camaraDampVelocity, camaraDappValue);
//camara.transform.eulerAngles = transform.eulerAngles;
camara.transform.LookAt(camaraHandler.transform);
```

2.4.1 Sistema de lock

Con la utilización de la cámara, se puede fijar la vista del personaje respecto a algunos tipos de objetos de la escena que deseen, consiguiendo que no pierde el objetivo al objetivo. Con el uso de motor físico del Unity previamente explicado, utilizando la función "**Physics.OverlapBox**"(encuentra toda las colisiones dentro del cuadrado), fija la vista del personaje al objetivo si el jugador manda una señal de "lock".

```
Vector3 modelOrigin1 = model.transform.position;
//print(modelOrigin1);
Vector3 modelOrigin2 = modelOrigin1 + new Vector3(0, 1, 0);
Vector3 CenterLapbox = modelOrigin2 + model.transform.forward * 5.0f;
Collider[] cols = Physics.OverlapBox(CenterLapbox, new Vector3(0.5f, 0.5f, 5.0f), model.transform.rotation, LayerMask.GetMask("Enemy"));
```

Un uso que puede dar a este sistema sería es en cuanto al combate del personaje respecto a algunos tipos de monstruos, de esta forma el personaje se puede fijar a la posición del monstruos sin perder la vista, de esa forma lograría un para los jugadores un mejor sensación de diversión.

Una vez fijado el objetivo, aparece un icono en la parte central del objetivo donde la vista del jugador está centrado en el icono. Se puede abandonar la vista al monstruo

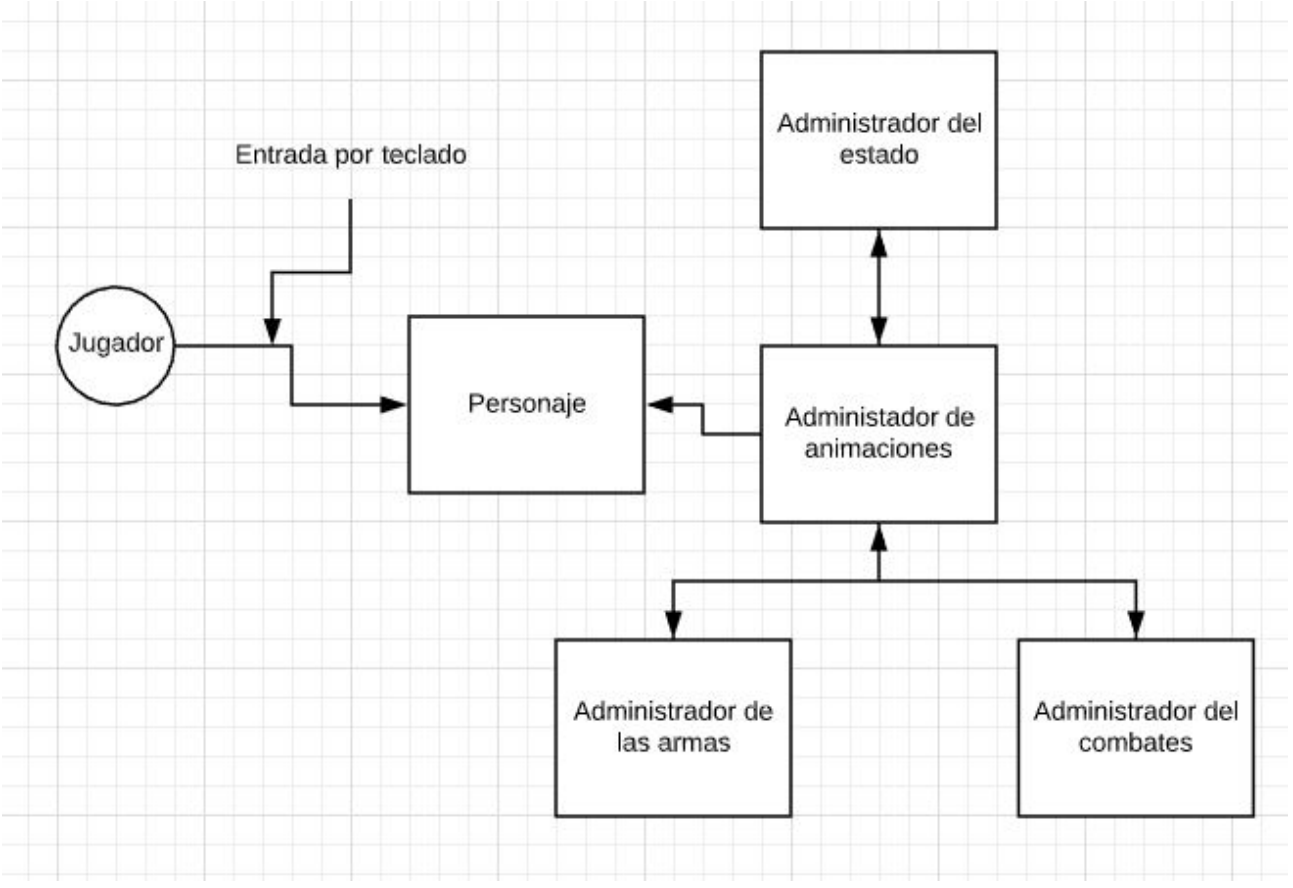
mandando la señal del lock otra vez o distanciando una ciertas distancias respecto al objetivo.

2.5 Administradores

Para crear una estructura de control más fluido en el juego, se puede crear diferentes estructuras de datos de control para administrar los comportamientos del personaje, sería una forma muy útil en cuanto a la limpieza o lógica del juego en sí. Cada uno de los administradores del personaje ejerce un labor propio, para cada uno de ellos está interconectado en sí para compartir toda las informaciones o cambios que se producen en el personaje en la escena del juego.

El personaje del juego tiene diferentes atributo en el juego(nombre,vida,monedas,fuerza etc), todos estos atributos está controlado por los diferentes administradores del personaje. Los componente principales se refleja en el siguiente esquema, donde el jugador manda señal de entrada, y éste dependiendo de las señales de entrada por el teclado realizara determinadas acciones (animaciones , cambios en los atributos, movimientos etc). Luego esos tipos de señales(ataque, defensa etc) serán almacenado en los diferentes administradores que lleva encima.

En los siguientes secciones se detallan los diferentes administradores reflejados en el gráfico. Cada unos están interconectado, y comparte algunos informaciones. En los siguientes secciones se explicará con detalles sobre los diferentes administradores del personaje.



2.5.1 Administrador de animación

El administrador de animación es el componente principal donde conecta a los diferentes administradores del personaje.

Almacena diferentes componentes del jugador:

- Componentes de animación (Sección **2.2 Animaciones**, controla todas las animaciones que se produce).
- Sensores del jugador (Componente físico del jugador).
- Modelo del jugador(“eliteKnight”,objeto 3D).

Todo los componentes del jugador definido previamente tiene un uso posterior para los diferente administradores. En Administrador de animación**,** dependiendo el estado de las transiciones de los animaciones, algunos no requieren informaciones de los atributos del personaje (si saltar o atacar) pero otra si lo necesita (cuando la vida del jugador es cero cambiar al estado muerto etc) porque controla la parte de las animación del jugador y éste mandar señales a los códigos donde controla la parte del animación.

2.5.2 Administrador del jugador

El Administrador del jugador almacena diferentes estados del personaje (roll, saltar, atacar, defensa etc), para lograr almacenar esos señales cogen el componente de animación que está almacenado en el administrador del animación y recopilan toda las señales que se produce en él. También almacena algunos atributos del personaje como la vida ,el maná del personaje y la stamina del personaje.

```
isGround = am.ac.CheckState("ground");
//Debug.Log(isGround);
isJump = am.ac.CheckState("jump");
isFall = am.ac.CheckState("fall");
isRoll = am.ac.CheckState("roll");
isAttack = am.ac.CheckStateTag("attackR") || am.ac.CheckStateTag("attackL");
isHit = am.ac.CheckState("hit");
isDie = am.ac.CheckState("die");
isBlocked = am.ac.CheckState("blocked");
//isDefense = am.ac.CheckState("defense1h", "defense");
isAllowDefense = isGround || isBlocked;
isDefense = isAllowDefense && am.ac.CheckState("defense1h", "defense");
isUnAttackble = isRoll;
```

2.5.3 Administrador de combate

El Administrador de combate almacena informaciones de los colisiones que se produce al personaje, en este caso cuando entra contacto con algunos armas de la escena.

2.5.4 Administrador de arma

El Administrador de arma almacena el arma que lleva el personaje y el colisionador que lleva encima del arma. En la sección **2.2 Corrección de Animación** se ha introducido la estructura ósea del modelo 3D del juego, para encontrar un objeto que cuelga encima de la mano del personaje (arma en este caso) se puede hacer manualmente arrastrando el objeto en sí al componente que almacena el arma (no sería una forma útil) o usando un métodos recursivos que encuentra todo los hijos de la estructura ósea del personaje hasta encontrar la parte donde se localiza el arma que lleva encima.

De esta forma se consiguen se ahorran el tiempo de ir encontrando el arma y resuelve el problema de arrastrar manualmente cuando el jugador intenta cambiar su arma en el juego. Es una forma más eficiente en cuanto al diseño del juego.
El codigo seria asi:

```
public static Transform DeepFind(this Transform parent, string targetName) {

    Transform tempTrans = null;

    foreach(Transform child in parent) {
        if(child.name == targetName) {
            return child;
        }
        else {
            tempTrans = DeepFind(child, targetName);
            if(tempTrans != null) {
                return tempTrans;
            }
        }
    }
    return null;
}
```

2.5.4 Lógicas

En las secciones anteriores se ha explicado los diferentes administradores del personaje, donde el administrador de animación almacena diferentes componentes del jugador(modelos 3D,componentes de animaciones etc), luego esos componentes almacenado será compartido con otros administradores. El administrador de arma busca los armas que lleva el personaje en su estructura ósea y controla el componente de colisión que lleva encima de él (no queremos que el colisionador está habilitado durante todo el tiempo sino solo cuando recibe señal de ataque,podríamos crear funciones de eventos en los animaciones del personaje donde lo controla). Luego el administrador de combates comprueba si el jugador se ha producido algunas colisiones con algunas armas del enemigo en la escena. El administrador del personaje almacenan las señales de los animaciones que se produce, estos nuevos señales recopilado por el administrador del personaje será utilizado otra vez en el administrador del animación, por ejemplo cuando recipe señal de colisión con algunos colisionador de arma manda un señal de golpe y el animador del personaje se va al estado de golpe reproduciendo la animación deseada.

Capítulo 3

Sistema Inventario :

En un juego RPG para que sea más interactivo con el jugador ha de tener que algunos objetos donde el jugador puede interaccionar con él. En este capítulo introducimos el concepto del sistema de inventario.

3.1 Introducción

Uno de los primeros sistemas de inventario real implementado ampliamente por los desarrolladores de RPG, los inventarios basados en la "Regla de 99" se asocian más comúnmente con JRPG clásicos como Final Fantasy VI, Chrono Trigger y las primeras

entradas en la serie Pokémon. El paradigma de diseño detrás de este sistema es extremadamente simple: los jugadores pueden esconder una cantidad de artículos en su inventario, pero solo un número fijo de cada uno.

3.1.1 Conceptos de artículos y mochilas

Los artículos y las mochilas son un concepto amplio. Los artículos generalmente incluyen accesorios, materiales, artículos de búsqueda, monturas y mascotas. Las mochilas se pueden dividir en mochilas de apoyo, mochilas de material, mochilas para artículos de tareas, barras de montaje y barras para mascotas.

A veces, por conveniencia, los accesorios, los materiales y los elementos de la misión se pueden combinar en la misma mochila, de modo que los accesorios y los materiales se almacenarán en la misma estructura de datos.

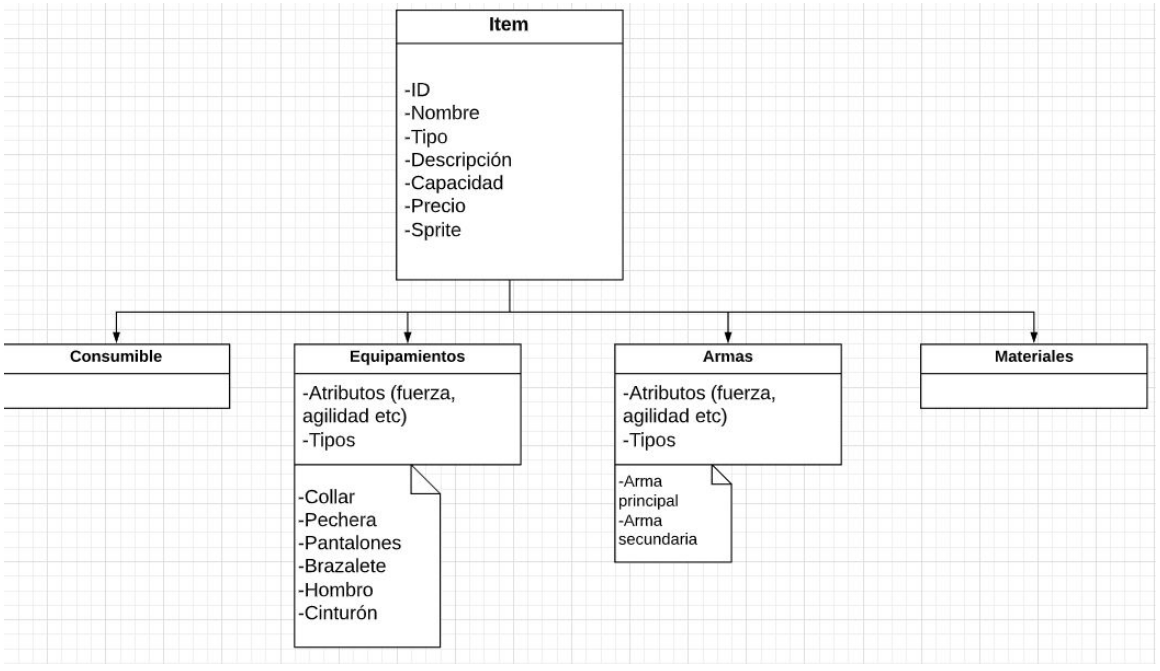
El sistema de mochila también incluirá otras mochilas de propósito especial, como un almacén (almacenamiento de accesorios almacenados a través del NPC del almacén), una barra de ventas (almacenamiento de los accesorios vendidos por el jugador al NPC para una recompra fácil) y una barra de pérdida (almacenamiento del puesto del jugador Bienes) etc.

3.2 Artículos (Items)

En el capítulo anterior se ha explicado en concepto de los artículos en el sistema de inventario. Los artículos se divide en muchos tipos pero en el sistema del juego que se va a implementar se introducir 4 tipos de objetos diferentes : Consumibles,Equipamientos, Armas y Materiales.

Para cada uno de los tipos se crea una clase **Ítem** donde almacena los diferentes propiedades de los artículos del juego (ID, nombre, Calidad del objeto etc), todo los objetos se hereda de esa clase **Ítem**, de esa forma solo se añade algunos propiedades a las clases heredadas, por ejemplo la clase consumible contiene atributos adicionales com el HP(genera la vida) y el MP(genera el maná).

Esquema principal de los Artículos del juego :



3.3 Administrador del Inventario

El administrador del inventario ejerce el labor de cargar ficheros json para el sistema de artículos que se ha implementado. En los juegos existen muchos tipos de objetos, para este caso, es buena idea guardar todos las informaciones de los artículos en un script json(es un formato de texto sencillo para el intercambio de datos) ya que es una forma más ordenada de visualizar los diferentes objetos que van a crear. Luego se carga el script json que has generado y lo guarda en una lista de artículos (Ítems) previamente definido.

Una estructura del fichero json sería así, donde el id indica el ID del objeto, el nombre del artículo, su precio, la cantidad etc.

```
"id": 1,
"name": "hp_flask",
"type": "Consumible",
"quality": "Common",
"description": "aumenta el hp",
"capacity": 10,
"buyPrice": 10,
"sellPrice": 5,
"hp": 10,
"mp": 0,
"sprite": "Sprites/Items/hp"
```

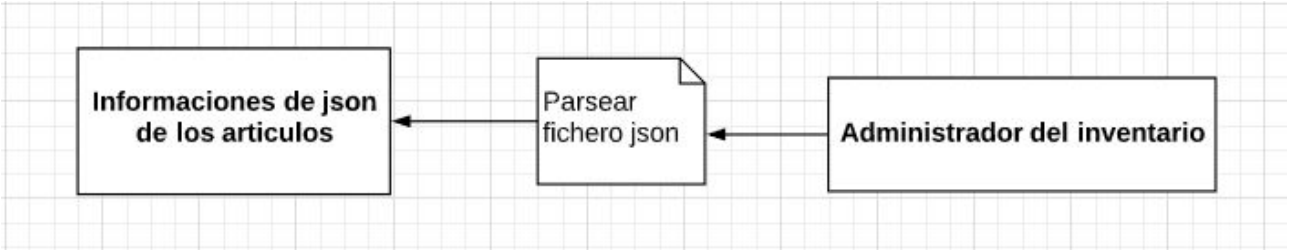
Para codificar / decodificar los datos JSON en una estructura de datos de tiempo de ejecución, existen múltiples librería como el **JSONObject** del asset store de **Unity** donde se puede parsear y crear un objetos json, y convertir en otros estructuras de datos.

```
itemList = new List<Item>();
//obtener el objeto texassert del unity
TextAsset itemText = Resources.Load<TextAsset>("Items");
string itemJson = itemText.text;
//print(itemJson);
JSONObject j = new JSONObject(itemJson);
```

Una vez cargado el fichero json ,hay que procesar todo el fichero json y lo guarda para cada unos de los atributos en distintos variables (int y string), luego estos son almacenado en los diferentes tipos de clases de artículos que se crea (artículos de consumible, artículos de arma etc).

```
Item newItem = null;
switch (type)
{ //int id, string name, ItemType type, Quality quality, string des, int capacity, int buyPrice, int sellPrice, string spr
    case ItemType.Consumible:
        int hp = (int)(item["hp"].n);
        int mp = (int)(item["mp"].n);
        newItem = new Consumible(id, name, type,quality, description, capacity, buyPrice, sellPrice, sprite, hp, mp);
```

Todos los Ítems previamentes definidos se guarda luego un una lista de Ítem.



3.4 Inventario

Inventarios o mochilas son estructura de datos donde el jugador almacena diferentes tipos de objetos o Ítems (artículos) en el sistema de datos.

En el juego se van a introducir 4 tipos de inventarios:

- Almacén del jugador (Donde puede almacenar objetos)
- Inventario de objetos (Inventario principal del juego)
- Panel del jugador (Donde almacena diferentes tipos de armas y armaduras)
- Panel de venda (vender y comprar objetos).

3.4.1 Estructura del Inventario

La estructura principal del inventario componen múltiples huecos o slots

Cada unos de los huecos del inventario puede guardar diferentes tipos de objetos previamente explicado. Para crear diferentes tipos paneles de inventarios y tener una estructura de datos que se puede manejar de forma eficiente, un primer acercamiento es crear una clase principal del panel donde contiene los atributos principales del panel, luego cada unos de los diferentes paneles que vas a crear se hereda de la clase padre, porque la estructura principal del inventario es lo mismo. De esa forma se ahorra las dublicas de códigos y consiguen más eficiencias en cuanto al desarrollo de los códigos.

Una vez creado la estructura principal del inventario (componen de diferentes slot que más adelante se va a explicar) han de tener funciones que comprueba la capacidad de los objetos almacenado, ya que un hueco no puede almacenar infinitos objetos

```
private Slot FindEmptySlot()
{
    foreach (Slot slot in slotList)
    {
        if(slot.transform.childCount == 0)
        {
            return slot;
        }
    }
    return null;
}
```

y detectar si hay huecos libre para seguir almacenando objetos, una vez el inventario está lleno no puede seguir añadiendo objetos en los slots o comprobar si los slots pertenece al mismo tipo, si es así se puede almacenar el ese mismo hueco comprobando el almacenamiento del objeto(En la sección **3.4.2** explica detalladamente el slot).

```
1 reference
private Slot FindSameIdSlot(Item item)
{
    foreach(Slot slot in slotList)
    {
        if(slot.transform.childCount >= 1 && slot.GetItemId() == item.ID && slot.IsFilled() == false)
        {
            return slot;
        }
    }
    return null;
}
```

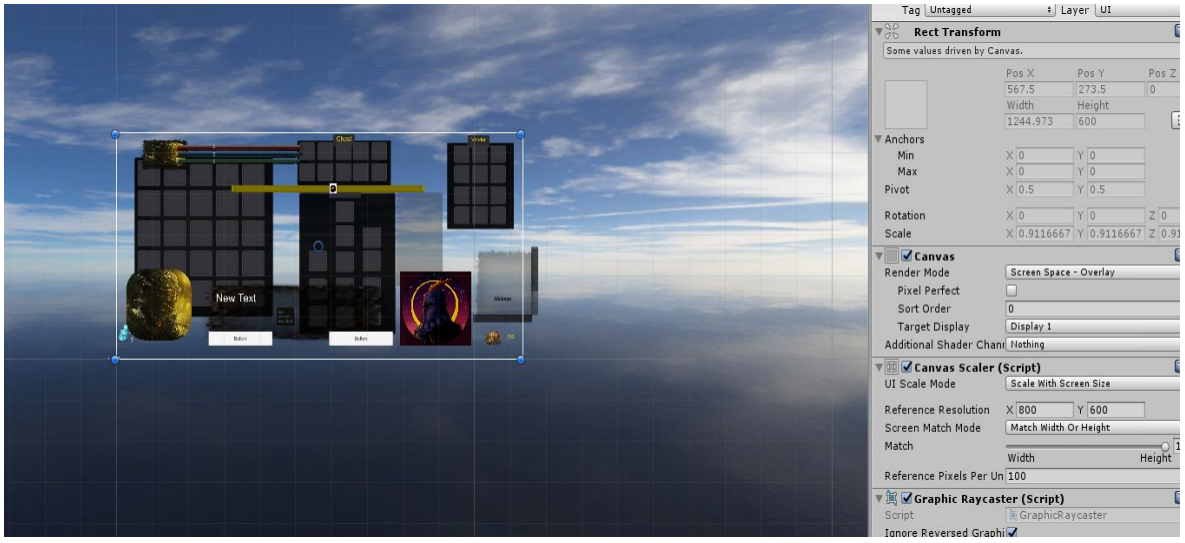
Las clases heredadas del panel principal ya no tiene que comprobar estos otra vez.
También ha de tener algunos funciones para guardar diferentes objetos en los slots del panel pasando el ID del artículo.

```
public bool StoreItem(int id)
{
    Item item = InventoryManager.Instance.GetItemById(id);
    return StoreItem(item);
}
```

```
public bool StoreItem(Item item)
{
    if(item == null)
    {
        Debug.LogWarning("No existe el ITEM id !!!!");
        return false;
    }
    if(item.Capacity == 1)
    {
        Slot slot = FindEmptySlot();
        if(slot == null)
        {
            Debug.LogWarning("No hay mas huecos !!");
            return false;
        }
        else
        {
            slot.StoreItem(item); // alamcena el objeto de el slot del inventario
        }
    }
```

```
else
{
    Slot slot = FindSameIdSlot(item);
    if(slot != null)
    {
        slot.StoreItem(item);
    }
    else
    {
        Slot emptySlot = FindEmptySlot();
        if(emptySlot != null)
        {
            emptySlot.StoreItem(item);
        }
        else
        {
            Debug.LogWarning("No hay mas huecos !!");
            return false;
        }
    }
}
return true;
```

Los paneles de los inventarios al final son interfaces de usuarios. Para crear una interfaz en Unity necesita el componente "**Canvas**". El **Canvas** es el área donde todos los elementos UI deben estar. El Canvas es un Gameobject con un componente Canvas en él, y todos los elementos UI deben ser hijos de dicho Canvas, para crear un nuevo elemento UI, tal como una Image (imagen) utilizando el menú GameObject > UI > Image, automáticamente crea un Canvas si ya no hay uno en la escena ya. El elemento UI es creado como un hijo de este Canvas. El área Canvas es mostrado como un rectángulo en la vista de la escena.



Una vez creado la interfaz del panel del inventario, como se puede manejar el control del panel del inventario. Unity nos proporciona otra componente que se llama grupo Canvas.

El grupo Canvas se puede utilizar para controlar ciertos aspectos de un grupo completo de elementos de la interfaz de usuario desde un lugar sin necesidad de manejarlos

individualmente. Las propiedades del Canvas Group afectan al GameObject en el que se encuentra, así como a todos los elementos secundarios.

Para mostrar el panel del inventario solo se modificar el valor de **alpha** (La opacidad de los elementos de la interfaz de usuario en este grupo) y el **blocksRaycasts** que si es verdad (el jugador puede interaccionar con el elemento) y si queremos cerrar el panel lo deshabilita la opción (poner falso).

3.4.2 Slot

En el capítulo anterior se ha explicado los **huecos** o los **slots,** que son estructura donde los artículos u objetos del sistema de inventario.

Como unity nos permite crear prefabs (instanciación de objetos reutilizables desde los recursos del proyecto), creando así el **slot** prefabs que son reutilizados en los paneles del inventario, cada uno de los slots componen de un Ítem o artículo (en la sección **3.4.2 Interfaz del Ítem** explica la estructura del Ítem), se puede almacenar los Ítems en cada unos de los slots cuando es llamada.

```
public void StoreItem(Item item)
{
    if(transform.childCount == 0)
    {
        GameObject itemGameObject = Instantiate(itemPrefabs) as GameObject;
        itemGameObject.transform.SetParent(this.transform);
        itemGameObject.transform.localScale = Vector3.one;
        itemGameObject.transform.localPosition = Vector3.zero;
        itemGameObject.GetComponent<ItemUI>().SetItem(item);
    }
```

Para que el usuario puede interaccionar con los Ítem de cada unos de los slots mediante el ratón (coger el objeto, dejar el objeto), Unity nos proporciona controles de los movimientos del ratón o clicks declarando "**IPointerEnterHandler,IPointerExitHandler, IPointerDownHandler**" en la cabecera de la clase ya que son eventos del sistema de Unity.

Aquí se muestra un ejemplo de código de cómo equipar un objeto en el panel del jugador (equipando una armadura o arma), donde el jugador hace un click con el ratón al objeto Item, comprueba que no ha cogido ningún objeto en ese momento y recoge la interfaz del Ítem detectando si es de tipo arma o armadura, luego se lo pasa esa información al Ítem

(sección 3.2 define que es Ítem) creando un nuevo objeto y llama a la función del panel del jugador para equipar este artículo(Item).

```
if (eventData.button == PointerEventData.InputButton.Right)
{
    if(InventoryManager.Instance.IsPickItem == false && transform.childCount > 0)
    {
        ItemUI currenItemUI = transform.GetChild(0).GetComponent<ItemUI>();
        if(currenItemUI.Item is Equipment || currenItemUI.Item is Weapon)
        {
            currenItemUI.SubAmount(1);
            Item currenItem = currenItemUI.Item;
            if (currenItemUI.Amount <= 0)
            {
                DestroyImmediate(currenItemUI.gameObject);
            }
            CharacterPanel.Instance.PutOn(currenItem); // pasar el objeto itemUI al panel

        }
    }
}
```

También hay que tener en cuenta de si el slot está vacío o no , la capacidad de almacenamiento de los Ítems(los artículos) en cada slot, ya que si la capacidad del **slot** del artículo no ha alcanzado el máximo, podemos seguir almacenando artículos en el mismo slot (lo hace comprobando el ID de cada artículo) y si la capacidad está del slot esta lleno buscamos otro huecos libre que encuentra en el panel de slot.Todos estos problemas se ha resolvido en las secciones anteriores.

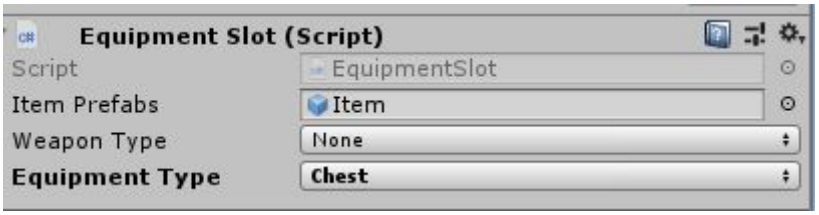
Una vez detallado los funcionamientos básicos del slot, para crear diferentes tipos de paneles, hay que instanciar diferentes tipos de slot creando prefabs para cada uno de ellos **:**

- Slot para equipamientos
- Slot para la venda de los artículos.

El slot de equipamiento tiene dos atributos que lo diferencia del slot normal,

```
public WeaponType weaponType; // tipos de weapons
public EquipmentType equipmentType; // tipos de equipment
```

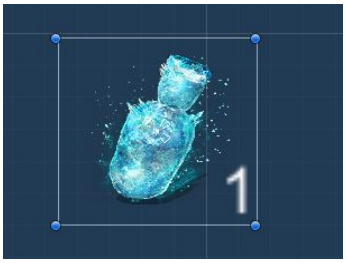
para cad uno de los slot en el panel de equipamientos en la sección del panel del inventario, cada hueco del slot que lleva el panel solo es posible almacenar un tipo de armas o armaduras determinadas. Los diferentes tipos de equipamientos está definido el la gráfica principal del sistema de inventario.



Para poder interaccionar con el artículo en cada uno de los slots, el funcionamiento es muy similar al slot normal ya que se hereda de la clase **Slot** y solo hay que modificar el código "**OnPointerDown(PointerEventData eventData)**" para el panel de equipamientos.

3.4.2 Interfaz del Ítem

La interfaz del Ítem (artículos) compone de los objetos como la la cantidad y el Imagen de artículo.



En esta sección debe crear un objeto prefab para el Interfaz del Ítem ya que se trata de un objeto reutilizables y los usos de éste se refleja continuamente en los **slots** de los artículos. Cuando un Ítem es cargado en el slot del panel del inventario, recoge aquellas informaciones del Ítem (artículo) que se ha generado, solo hay que cambiar el Sprite (son objetos gráficos 2D) del Ítem y la cantidad, de esta forma se puede generar todas aquellas items necesarias para la interfaz del usuario.

```
7 references
public void SetItem(Item item, int amount = 1)
{
    transform.localScale = animationScale;
    this.Item = item;
    this.Amount = amount;
    ItemImage.sprite = Resources.Load<Sprite>(item.Sprite);
    if (Item.Capacity > 1)
        AmountText.text = Amount.ToString();
    else
        AmountText.text = "";
}
3 references
```

La clase también permite sumar o restar diferentes cantidades o cambiar posición de los artículos en los diferentes slots cuando el usuario realiza algunos tipos de interacciones con los objetos.

3.5 Paneles de Inventario

- **Almacén del jugador** : Es un panel del sistema de inventario donde el jugador se puede almacenar todo los tipos de artículos. Está compuesto por múltiples slot.

- **Inventario de objetos :** Es muy similar al almacén del jugador pero este contiene más slot y es el panel principal de jugador.

Para crear un artículos en cada uno de los slots , todos los artículos contiene un ID que está almacenado en el administrador de inventario, para crear un objetos Item le pasa un ID del artículo que vas a generar, una vez el artículo es creado le pasamos a otros función que son capaces de crear una Interfaz de artículo para el slot correspondiente(en la **sección 3.4.2 Slot** explica el funcionamiento).

```
2 references
public bool StoreItem(int id)
{
    Item item = InventoryManager.Instance.GetItemById(id);
    return StoreItem(item);
}
```

- **Panel del jugador** : Este panel solo almacena ítems (artículos) que pertenece al tipo arma o de armadura. Incluye otras funcionalidades donde el usuario puede equipar y desequipar las armas o armaduras de los slots. Una vez el usuario interacciona con el **Slot** del inventario del jugador, recibe el señal y llama a la función **PutOn** de la clase de **panel del jugador** (en la sección **3.4.2 Slot** muestra el código de equipamiento) donde le pasa un artículos de tipo arma o de armadura y rellena el slot correspondiente.

```
Item outItem = null;
foreach(Slot slot in slotList)
{
    EquipmentSlot equipmentSlot = (EquipmentSlot)slot; // transforma en qup slot
    if (equipmentSlot.IsRightItem(item))
    {
        if(equipmentSlot.transform.childCount > 0) // ya tiene armaduras
        {
            ItemUI currentItemUI = equipmentSlot.transform.GetChild(0).GetComponent<ItemUI>();
            outItem = currentItemUI.Item;
            currentItemUI.SetItem(item,  1);
        }
        else
        {
            equipmentSlot.StoreItem(item); // asiganr el objeto a la casilla
        }
        break;
    }
```

Una vez los huecos de los slot de panel de jugador contiene algunos objetos almacenados, se actualiza automáticamente los atributos del personajes del juego y lo muestra en un panel auxiliar donde se puede visualizar los diferentes atributos del jugador(la agilidad,la fuerza etc).

```
foreach (EquipmentSlot slot in slotList)
{
    if(slot.transform.childCount >0) // si hay obejtos en la casilla
    {
        Item item = slot.transform.GetChild(0).GetComponent<ItemUI>().Item;
        if(item is Equipment)
        {
            Equipment e = (Equipment)item;
            strength += e.Strength;
            intellect += e.Intellect;
            agility += e.Agility;
            stamina += e.Stamina;

        }
        else if (item is Weapon)
        {
            power += ((Weapon)item).Damage;
        }
```

- Panel de venda: Panel auxiliar donde el jugador se puede realizar compras o ventas de los diferentes artículos en el panel de venda.
  En los mayoría de los juegos RPG proporciona un sistema donde puede vender o comprar objetos de la tienda del juego. En el juego se ha incorporado este sistema donde ofrece al usuario los diferentes tipos de artículos para la venda.

En primer lugar, se almacenan los diferentes tipos de artículos en los slots. Cuando el jugador interacciona el slot del panel de la venda de los artículos, manda un mensaje al objeto superior donde esta el script "**Vendor**",

```
if (eventData.button == PointerEventData.InputButton.Left && InventoryManager.Instance.IsPickItem == true)
{
    Item currentItem = transform.GetChild(0).GetComponent<ItemUI>().Item;
    transform.parent.parent.SendMessage("SellItem", currentItem);
}
else if (eventData.button == PointerEventData.InputButton.Right && InventoryManager.Instance.IsPickItem == false)
{
    if(transform.childCount > 0)
    {
        Item currentItem = transform.GetChild(0).GetComponent<ItemUI>().Item; // cojo el objeto que esta en el slot
        transform.parent.parent.SendMessage("BuyItem", currentItem); // envio mensaje hacia al padre
    }
}
```

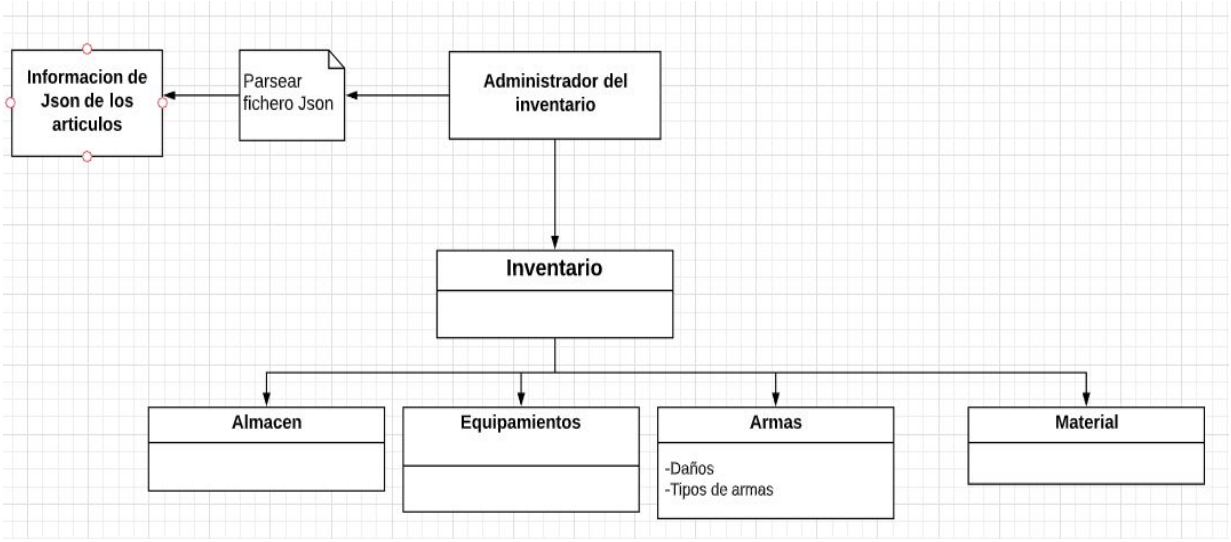
donde la clase **Vendor** ejerce el labor para la venta o compra de los artículos del panel. La clase proporciona múltiple funciones para realizar ciertas tareas juntos con el panel de la venda del juego,

```
public void BuyItem(Item item)
{
    bool isSuccess = player.ConsumeCoin(item.BuyPrice);
    if (isSuccess)
    {
        Knapsack.Instance.StoreItem(item);
    }
}
```

donde el jugador consume una cierta cantidad de monedas y se crea un artículos en el **Panel del jugador** a los artículos comprado**.**

En cuanto a la venda del objeto solo tenemos que destruir el articulo en el panel de jugador y devolver una cierta cantidad de monedas.

Estructura final del Inventario:



Capítulo 4
Sistema de misiones

En este capítulo se introduce el concepto sobre el sistema de los misiones. Una misión (tarea) generalmente se refiere al trabajo asignado o las responsabilidades dentro del juego. En muchos de los juegos que existen en el mercado, el propósito de las misiones es guiar al jugador para llevar a cabo a realizar ciertas actividades dentro del juego y darle al jugador un cierto medio de recompensa, éste es el labor de las misiones.

4.1 Introducción

Existen diversos formas, que generalmente se pueden dividir en los siguientes tipos:

- Tipo de diálogo: A --- B o A --- B --- A.
- Tipo de monstruo asesino: A --- monstruo asesino.
- Recoge tipo de elementos: Recoge elementos (mata monstruos o recoge).
- Explorando: Explorando el área.
- Usar tipo de accesorio: A ---- Usar accesorio.
- Guiado: Algunas operaciones debe completarse.

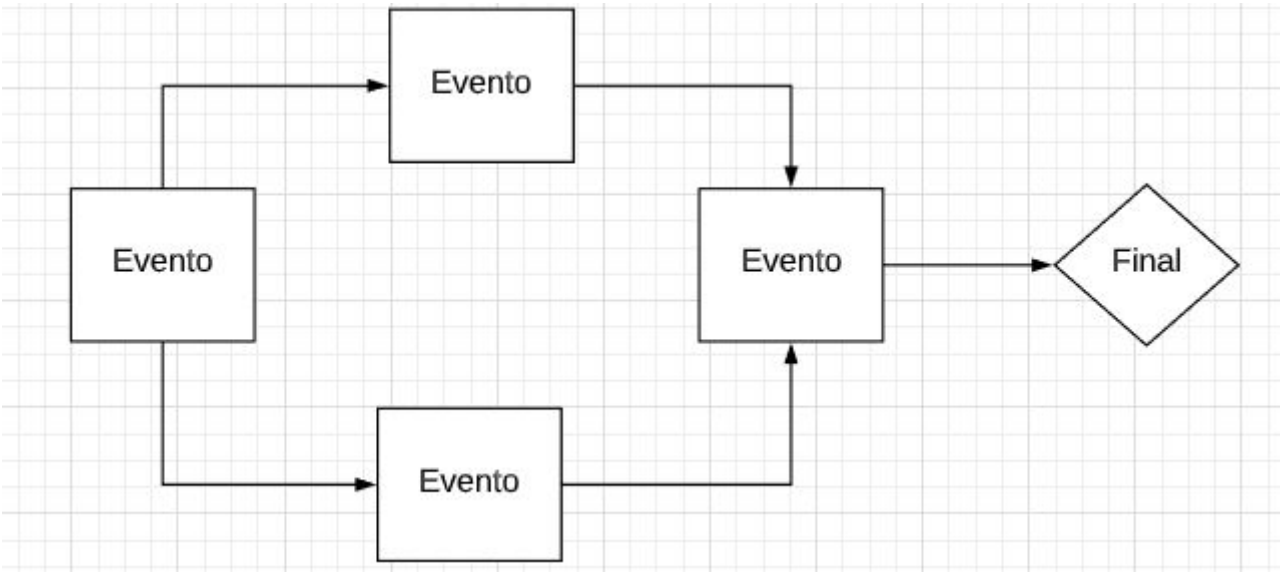
El diseño de la misión es intencional y direccional: el principio es dar al personaje habilidades y otros medios, establecer obstáculos frente al jugador y guiarlo para que use habilidades para superar los obstáculos.

En resumen, las misiones son aquellos sistemas que le dicen "**qué debe hacer ahora**" durante el juego. Después de completar estas tareas, generalmente obtendrá las recompensas correspondientes. Puede obtener una tarea empaquetando la ayuda, el comportamiento y las recompensas que el usuario necesita, luego, el diseñador puede insertar otros elementos en la tarea.

La idea de este diseño es demasiado simple, pero cuando miramos hacia atrás, encontramos que este diseño ha gobernado a toda la industria. Desde FPS a RPG, e incluso juegos móviles y juegos web, las tareas y los sistemas similares a las tareas están en todas partes.

4.2 Estructura de sistema de misión

El sistema de misión implica unos series de eventos donde el jugador tiene que participar en unos de esos eventos para completar el objetivo final.



La estructura principal de una secuencia de misiones que esconde detrás de esto es un grafo, un gráfico es una estructura de datos no lineal, se puede definir como una colección de nodos que también se denominan "**vértices**" y "**bordes**" que conectan dos o más vértices. Con esta estructura se crear un sistema de misión basado en los eventos.

4.2.1 Ruta de misión

La ruta de misión representa los **vértices** en nuestro grafo, donde hace un seguimiento de los eventos iniciales y de los eventos finales de cada misión. Es una línea donde une un evento juntos con otro evento.
Cuando se inicia el sistema de misión, debe de saber cuál es la siguiente misión tiene que completar el jugador.

```
public QuestEvent startEvent;
public QuestEvent endEvent;

1 reference
public QuestPath(QuestEvent from , QuestEvent end)
{
    startEvent = from;
    endEvent = end;
}
```

4.2.2 Eventos

En el sistema existen diferentes estados sobre los flujos de los eventos:

- Esperando.
- Misión actual.
- Completado.

Cada uno de los eventos contienen múltiples atributos como el nombre, la descripción, el id, el estado de evento, una lista de ruta de misiones donde el jugador tiene que completar, etc. El atributo order ejerce el labor sobre el orden de completar cada uno de los eventos. Como los eventos son los nodos del grafo, en el sistema de misión en cada evento tiene un orden, ya que el jugador no puede completar la misión sin haber completo algunos de los eventos previamente definido.

4.2.3 Misión

Una misión, como se mencionó en las secciones anteriores, es la actividades del juego donde el jugador tiene que realizar. En este caso, la misión al final es un sistema de grafo donde contiene un listado de eventos. De esta forma se podría añadir múltiples eventos en nuestra misión(grafo) con un nombre y el id del evento. Mediante esta estructura se puede unir múltiples rutas de eventos en el sistema de misión, donde se establece el evento principal y el evento destino, en el cual se añade los eventos en la ruta de la misión. Se puede crear tantas rutas necesarias para la ramificación de una misión dentro del juego, así el usuario tiene múltiple forma de completar una misión.

```
2 references
public void AddPath(string fromQustEvent , string toQuestEvent)
{
    QuestEvent from = FindQuestEvent(fromQustEvent);
    QuestEvent to = FindQuestEvent(toQuestEvent);
    if(from != null && to != null)
    {
        QuestPath p = new QuestPath(from, to);
        from.pathList.Add(p);
    }

}
```

Como cada un de los eventos tiene un orden de prioridad para poder ser completado, deben saber en qué punto o el fase se ha alcanzado el jugador en la rutas del grafo dentro del juego. Para comprobar éste, una forma es usar el algoritmo de **Búsqueda en anchura**, es un algoritmo de búsqueda donde recorrer o buscar elementos en un grafo, se comienza en la raíz y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol, en nuestro caso sería el evento a completar.

```
2 references
public void BFS(string id , int orderNumber = 1)
{
    QuestEvent thisEvent = FindQuestEvent(id);
    thisEvent.order = orderNumber;

    foreach (QuestPath e in thisEvent.pathList)
    {
        if (e.endEvent.order == -1)
            BFS(e.endEvent.GetID(), orderNumber + 1);
    }
}
```

4.3 Administrador de Misión

El administrador de misión ejerce el labor de crear múltiples eventos donde almacena para nuestra clase de misión.

```
QuestEvent a = quest.AddQuestEvent("test1", "desc 1",A);
QuestEvent b = quest.AddQuestEvent("test2", "desc 2",B);
QuestEvent c = quest.AddQuestEvent("test3", "desc 3",C);
QuestEvent d = quest.AddQuestEvent("test3", "desc 3",D);

quest.AddPath(a.GetID(), b.GetID());
quest.AddPath(b.GetID(), c.GetID());
quest.AddPath(b.GetID(), d.GetID());
```

Con esta estructura se obtiene un sistema de grafo.

Éste también ejerce el labor sobre la parte del UI del jugador (en la sección **4.5 Interfaz** explica con más detalles), incluyendo diferentes tipos de Gameobject (botones) para cada eventos y determina el estado de cada evento si lo ha completado.

4.4 Sistema de brújula

El sistema de brújula te permite guiar el jugador sobre la localización del objeto que tiene que buscar en el sistema de misiones.
Para crear una brújula de misiones de estilo cercano al RPG, muy parecida a lo que ves en los juegos de Bethesda como Skyrim y Fallout 4. Se cubre cómo mostrar con precisión la dirección a la que se enfrenta tu jugador en la brújula y de cómo muestra los puntos de referencia o misiones en la brújula representada por un íconos.

El íconos nos determina si la posición del jugador está a la dirección del objetivo, de esta forma permite guiar al jugador en la escena del juego. La barra amarilla representa la línea de la brújula, si la posición del íconos está situado en el centro indica que el objetivo está enfrente al jugador y en los extremos de la barra determina que el objetivo está 0 a 90 o -90 grado de la vista del jugador.

Para calcular el ángulo del objetivo se usa la función "**Vector3.SignedAngle**" del Unity donde te devuelve el ángulo con signo de la posición que desee calcular. En primer lugar se recoge los coroneles de la barra amarilla, y determina las distancias entre los dos puntos extremos de la barra. De esta forma solo hay que determina el punto de la vista del jugador sobre el objetivo restringiendo los ángulos entre 0 a +(-) 90 grados, luego el icono(flecha) se mueve respecto a esa vistas.

```
Vector3[] v = new Vector3[4]; // los cornes de la barra de la brujula
compasLine.GetLocalCorners(v);
float pointerScale = Vector3.Distance(v[1], v[2]);

Vector3 direction = target.transform.position - player.transform.position;
float angleToTarget = Vector3.SignedAngle(player.transform.forward, direction, player.transform.up);

angleToTarget = Mathf.Clamp(angleToTarget, -90, 90) / 180.0f * pointerScale;
rect.localPosition = new Vector3(angleToTarget, rect.localPosition.y, rect.localPosition.z);
```

4.5 Interfaz

La interfaz de la misión te permite visualizar el recorrido de los eventos de la misión, es decir en qué punto ha llegado el jugador dentro de cada fase de la misión.

Los sistemas de UI (interfaz de usuario) explicado en la sección **3.4.1 Estructura del Inventario,** para el caso de la misión, está compuesto por un scrollview que muestra su contenido dentro de un marco desplazable.

En este mismo marco contiene otro elemento del sistema de UI como el buton que te indica el estado de cada misión, los iconos de cada fase de una misión , el compás etc.

Los elementos del UI se cambian dependiendo los estados de cada uno de los eventos de la misión.Cuando se produce un cambio de eventos en la misión se cambia la textura o los iconos de la imagen.

```
3 references
public void UpdateButton(QuestEvent.EventStatus s)
{
    status = s;
    if(status == QuestEvent.EventStatus.DONE)
    {
        icon.texture = doneImage.texture;
        buttonComponent.interactable = false;

    }
    else if (status == QuestEvent.EventStatus.WAITING)
    {
        icon.texture = waitingImage.texture;
        buttonComponent.interactable = false;
    }
    else if (status == QuestEvent.EventStatus.CURRENT)
    {
        icon.texture = currentImage.texture;
        buttonComponent.interactable = true;
        //ClickHandler();
    }
}
```

Capítulo 5
Diálogos

1 - Introducción :

El NPC es la abreviatura de personaje no jugador. Es un tipo de personaje en el juego, que no es manipulado por el jugador en el juego. En muchos de los videojuegos, el NPC generalmente está controlado por la inteligencia artificial de la computadora o por algunos sistemas de eventos dentro del juego, y es un personaje con su propio modo de comportamiento. Por lo general, se puede dividir en NPC de trama, NPC de combate y NPC de servicio, y a veces hay NPC con múltiples funciones.

Para cada juego tiene varias funciones, que van desde la creación de personajes hasta habilidades profesionales. Estas funciones requieren un intermediario para guiar al jugador. En los juegos tradicionales, estas funciones están respaldadas por botones, y los jugadores pueden hacer clic rápidamente en los botones para lograr el propósito de la función que se utilizará. Sin embargo, con el rápido desarrollo de la industria del juego, un solo mapa del juego ya no puede satisfacer a los jugadores actuales. Por lo tanto, el mapa del juego ha comenzado a desarrollarse desde el primer mapa individual hasta docenas de mapas, e incluso cientos de mapas. Con el aumento de los mapas, algunas funciones se han agregado una tras otra. Los botones de una interfaz principal ya no pueden llevar todas las funciones del mapa. Entonces nació NPC.

5.2 Panel de diálogos :

Creo que todos los que gustan de los juegos de rol estarán impresionados por la trama en el juego RRG. Como el juego de rol, la trama está principalmente conectada por el diálogo entre los personajes. El jugador capta la historia integrando la información del diálogo de todos los personajes.

Un panel de diálogo está compuesto principalmente por un panel de visualización y el contenido que lleva dentro de él, en este caso son los textos y algunos botones de interacción.Cuando el jugador entre se acerca (Collision) al NPC de la escena se abrirá este panel de diálogos donde el jugador interacciona con el NPC.

5.3 ScriptableObject :

Un **ScriptableObject** es un contenedor de datos que puede usar para guardar grandes cantidades de datos, independientemente de las instancias de clase. Uno de los principales casos de uso de ScriptableObjects es reducir el uso de memoria de su proyecto evitando copias de valores.

Cada vez que crea una instancia de ese Prefab, obtendrá su propia copia de esos datos. En lugar de usar el método y almacenar datos duplicados, puede usar un **ScriptableObject** para almacenar los datos y luego acceder a ellos por referencia desde todos los Prefabs. Esto significa que hay una copia de los datos en la memoria.

Cuando usa el Editor, puede guardar datos en **ScriptableObjects** durante la edición y el tiempo de ejecución ya que los **ScriptableObjects** usan el espacio de nombres del Editor y las secuencias de comandos del Editor. Sin embargo, en una compilación implementada, no puede usar **ScriptableObjects** para guardar datos, pero puede usar los datos guardados de los ScriptableObject Assets que configuró durante el desarrollo.

Los datos que guarda de las Herramientas del editor en **ScriptableObjects** como un activo se escriben en el disco y, por lo tanto, son persistentes entre sesiones.

Es muy útil en este caso para guardar informaciones de los diálogos que se establece entre el jugador y el NPC, de esta forma ahorraría muchas memorias y muchos más fácil para implementar en cuanto a los sistemas de diálogos. Para crear una clase que pertenece al ScriptableObjects debería añadir la palabra clave “**ScriptableObjects** ”.

La clase **DialgoData** pertenece al **ScriptableObjects** donde se guardará informaciones de los datos de los imágenes y texto de diálogos.

```
[CreateAssetMenu()]
0 references
public class DialogData : ScriptableObject
{
    public List<DialogContent> contents;
}

[System.Serializable] // se puede visualizar en el editor
1 reference
public class DialogContent
{
    public string dialogText;
    public bool charaADisplay;
    public bool charaBDisplay;
}
```

5.4 Importar desde Excel

A veces el contenidos de los diálogos son muy extensos, una forma más simple de visualizar o crear el contenido de los diálogos es guardar la información en un Excel , luego todas esas informaciones será importado desde Unity.
Para importar los contenido de excel necesitamos algunos assert que nos facilita el uso ,en este cas incorporó Unity-QuickSheet-ByEXCEL que es una mejora de Unity-QuickSheet.

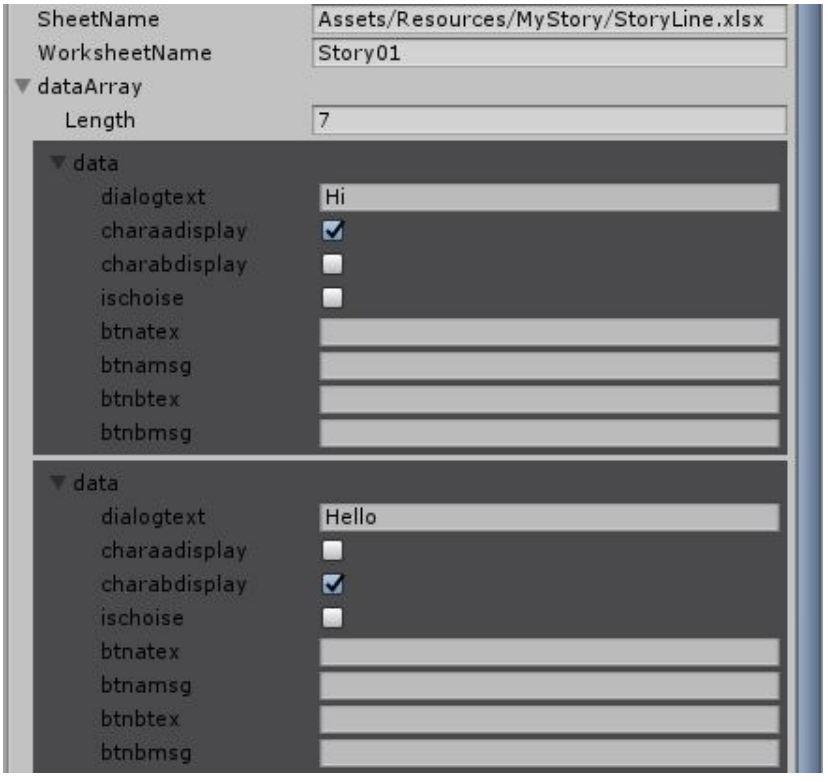
Se trata también de un objeto **ScriptableObjects** (en la sección 5.3 detalla que es un **ScriptableObjects**) donde contiene la siguiente estructura de información. En la parte del **Worksheet** debe añadir el fichero excel que vas a importar desde el ordenador, una vez cargado el fichero solo hay que hacer el click sobre el botón **Import**, en el path settings debe añadir la ruta de los archivos donde se va a guardar, una vez finalizado debe hacer click sobre el botón Generate y te genera los objetos necesarios. Más adelante se explica con más detalles.

Se crea un excel donde contiene la siguiente estructura :

A	B	C	D	E	F	G	
DialogText	CharaADisplay	CharaBDisplay	isChoise	btnATex	btnAMsg	btnBTex	btnBMs
Hola maestro	VERDADERO	FALSO	FALSO				
Hola valiente gue	FALSO	VERDADERO	FALSO				
Dejame pensar	VERDADERO	FALSO	VERDADERO	No lo estoy	GoHome	Estoy preparada	Yes
Una buena oport	VERDADERO	VERDADERO	FALSO				
Eres demasiado	VERDADERO	FALSO	FALSO				
Suerte	FALSO	VERDADERO	FALSO				
Gracias	VERDADERO	VERDADERO	FALSO				

La primera fila te indica los atributos que lo necesitan, por ejemplo el atributo **DialogText** guarda informaciones de los textos de los diálogos , el atributo **CharADisplay** y **CharBDisplay** son booleanos donde indicarán si deben aparecer una imagen del jugador o del NPC dentro del diálogo, el atributo **isChoise** es un booleano donde te indica si va a mostrar un botón de interacción para el jugador o no etc.
Una vez importado el fichero excel le indica la carpeta Runtime (El guión utilizado en el juego) y Editor(Plantillas usadas en el compilador) donde debería genera los ficheros C#.
En este caso ya no se necesita la clase **DialgoData** de la **sección 5.2** para guarda informaciones de los diálogos porque este sistema es más eficientes en cuanto a la limpieza y el desarrollo de la historia generando los **ScriptableObjects** necesarios.

Una vez generado, hacemos reimport al fichero excel y nos debería genera un **ScriptableObjects** (se puede generar tantos **ScriptableObjects**):

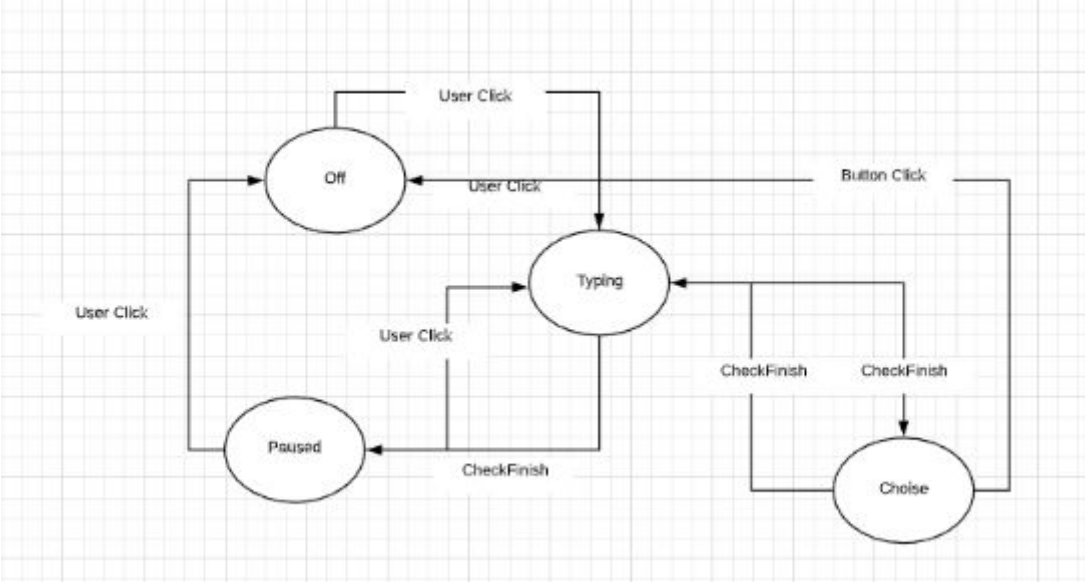


de esta forma se ahorraría de escribir informaciones en los códigos sino simplemente guardar informaciones en el fichero excel e importarlo.

5.5 Administrador del diálogo :

El administrador del diálogo ejercer la función de controlar todo los procesos de los diálogos que se establecen entre el jugador y el NPC de la escena.

Contiene una máquina de estados de los diálogos, dependiendo de cada unos de los estados ejercer una función u otro.



Typing es el estado inicial donde le mostrarla la conversación (diálogos) del NPC o del jugador, es cargado desde el ScriptableObjects (story_x).

```
//Load contetn
showUI();
LoadContent(data.dataList[currLine].Dialogtext.ToString(), data.dataList[currLine].Charaadisplay,
            data.dataList[currLine].Charabdisplay);
justEnter = false;
timerValue = 0;
```

Una vez finalizado el contenido del texto de la primera entrada se pasa al estado de **Pauses** donde el jugador interactúa en la escena para entrar al siguiente estado. Si en algún momento le apareciera una opción de elección.

entra al estado **Choice** donde el jugador debería elegir entre esas opciones. Si selecciona al **No lo estoy** se le cambiará al otro ScriptableObjects(Story02_) donde le mostrará una conversación diferente.

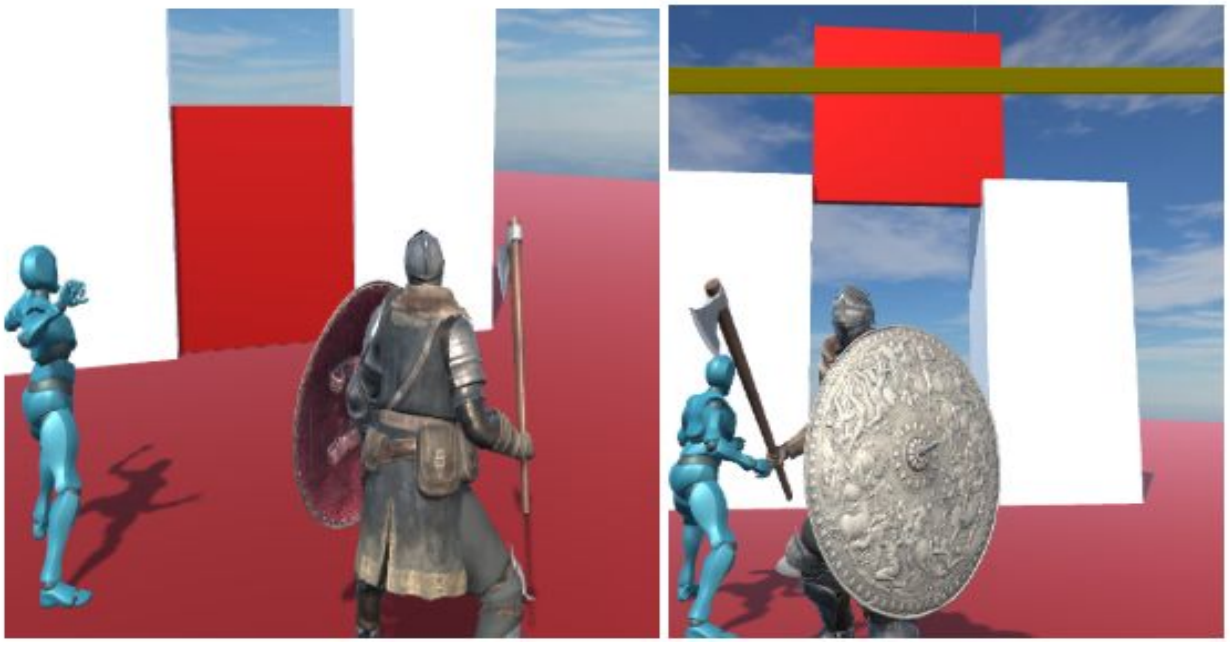
Story02_ es otra hoja de excel generado con la librería Unity-QuickSheet-ByEXCEL, contiene las siguientes informaciones.

DialogText	CharaADisplay	CharaBDisplay	isChoise
Debería entrenar	VERDADERO	FALSO	FALSO
Por supuesto	FALSO	VERDADERO	FALSO
Adios maestro, v	VERDADERO	FALSO	FALSO

```
    Story01 tempStory = Resources.Load<Story01>("Story02_");
    data = tempStory;

    Init();
    justEnter = false;
    LoadCharaTexture(assert.charATex, assert.charBTex);
    GoToState(STATE.TYPING);
    break;
```

y si selecciona **sí estoy** preparado le abrirá una puerta que está cerrado.

```
case "Yes":
    GameObject.FindGameObjectWithTag("AI Visual").SendMessage("OpenDoor");
    GoToState(STATE.OFF);
    break;
```



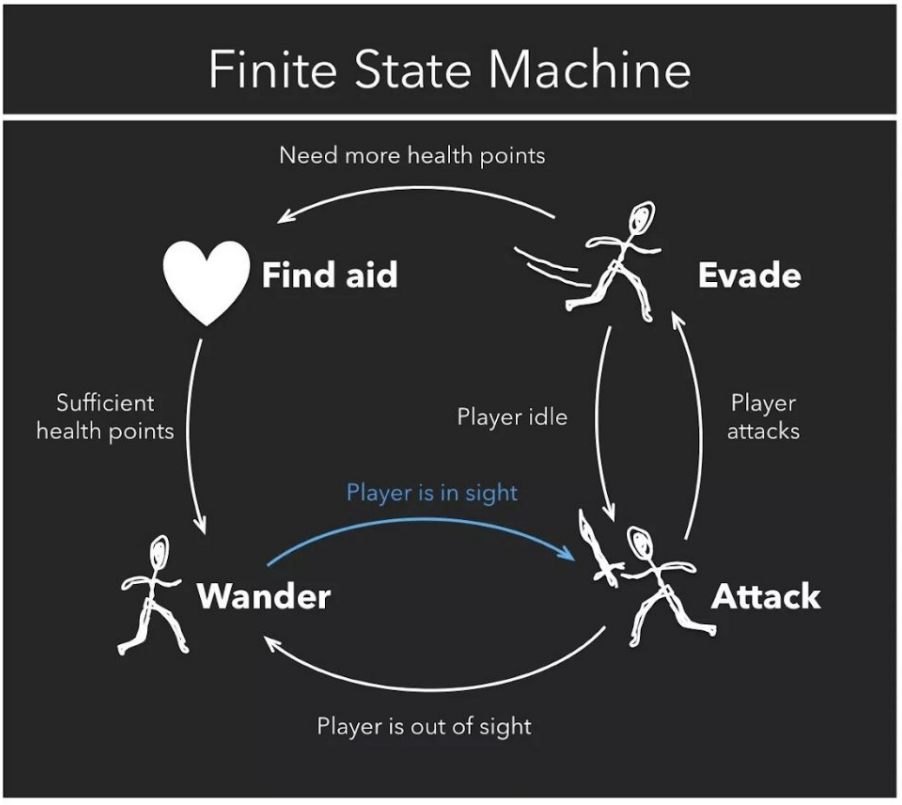
Capítulo 6
Inteligencia Artificial

Jugamos juegos principalmente para obtener una relajación, un disfrute y una emoción que no se puede obtener en la vida real. Esto requiere que el juego se pueda hacer para que coincida con el gusto del jugador, el tema del juego puede atraer al jugador y las reglas y los resultados del juego pueden hacer que el jugador esté satisfecho. En todo esto, la tecnología de inteligencia artificial juega un papel muy importante.

6.1 Introducción

Desde el nacimiento de la inteligencia artificial, se ha integrado estrechamente con los juegos. Porque la gente generalmente piensa que el proceso de los juegos humanos contiene inteligencia humana. Por lo tanto, cuando las personas crean un programa que puede completar cierto juego de humanos, creemos que este programa tiene algún tipo de "inteligencia" humana. Muchos juegos como el ajedrez y el interestelar se convertirán en un buen entorno para las pruebas de inteligencia artificial. El desarrollo del juego en sí ha traído mucha demanda de inteligencia artificial, como NPC (personaje no jugador) en el juego. La gente ha desarrollado muchos algoritmos para que un personaje en el juego se comporte más como un ser humano.

Un ejemplo de diseño del IA seria **Finite State Machine (**FSM**)**, es un modelo de IA con un cierto número de estados y transiciones de estado. El diseño de estado finito es la tecnología más común utilizada en la industria del juego, utilizada principalmente en juegos en tercera persona como acción, aventura, juegos de rol, etc.



El algoritmo FSM no es adecuado para todos los juegos. Imagina usar FSM en un juego de estrategia. Si el robot está preprogramado para responder de la misma manera cada

vez, entonces el jugador aprenderá rápidamente cómo vencer al robot, pero para nuestro caso sería lo ideal.

6.2 Simple IA

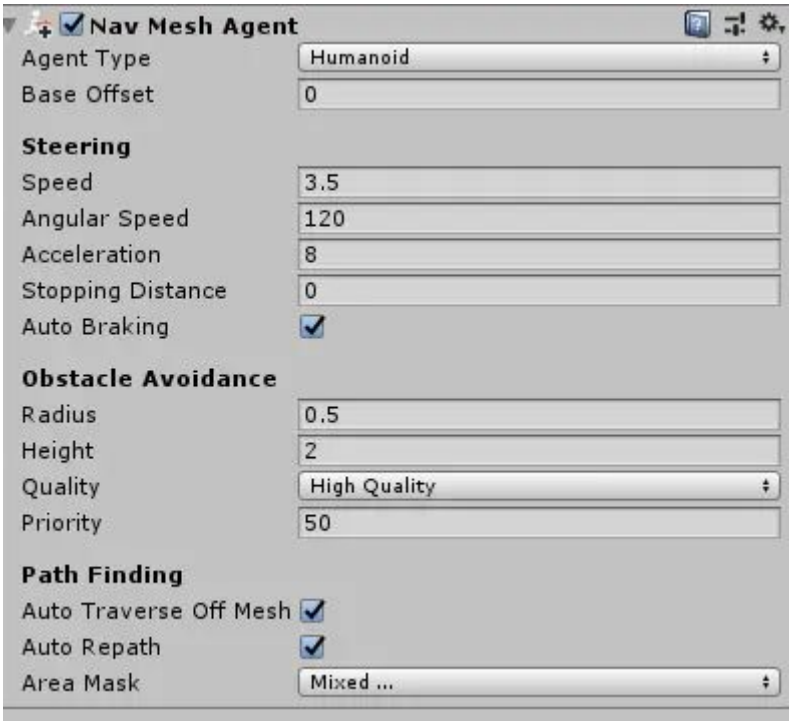
Para el comportamiento de los monstruos(IA) en el juego, se resume brevemente de la siguiente manera :

- El monstruo moverá dentro de un rango preestablecido.
- El jugador entra en el campo de visión del monstruo, y el monstruo se mueve con el jugador.
- Al entrar al rango de ataque, el monstruo ataca al jugador.
- El jugador abandona el campo de visión del monstruo, y el estado del monstruo vuelve al lugar del principio.

6.2.1 NavMesh

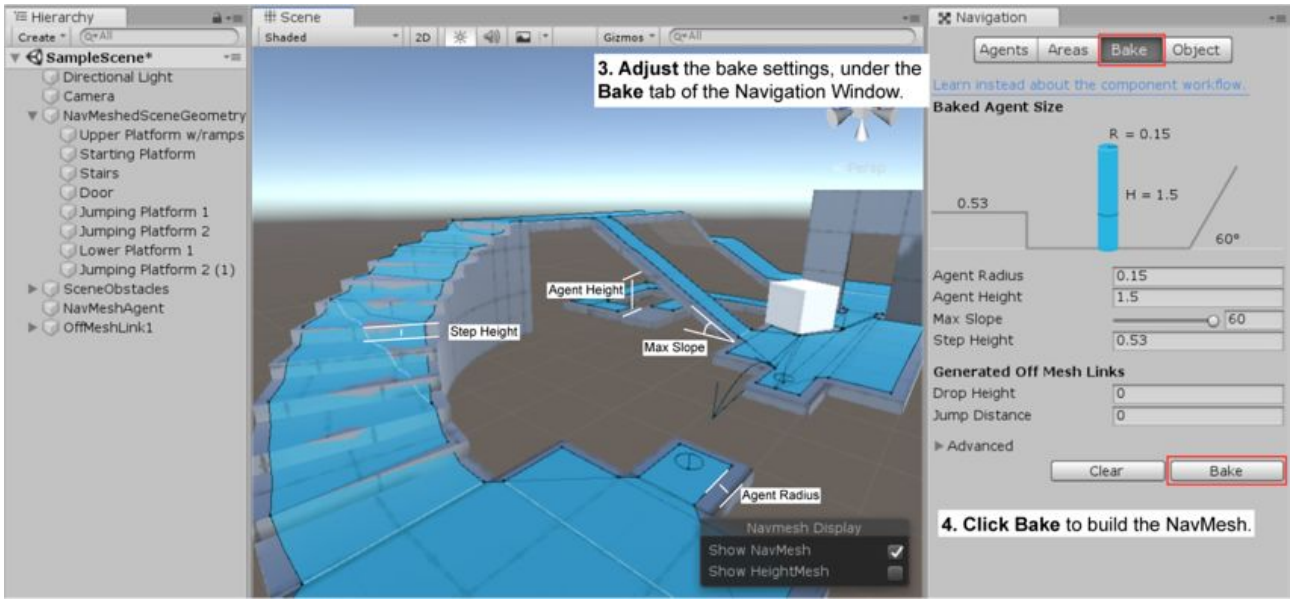
En muchos juegos, verás que cuando un personaje entra al área de vigilancia del monstruo, el monstruo empieza a perseguir al personaje y lo ataca. Cuando el personaje abandona el área, el monstruo se detendrá o continuará patrullando, esta es la función de la IA. El sistema de navegación en Unity nos permite proporcionar una ruta poderosa para encontrar a los personajes en nuestros juegos muy fácilmente. En esta sección se introduce los conceptos básicos de la navegación.

En primer lugar, usará este componente NavMeshAgent y lo agrega al monstruo correspondiente. Este componente es responsable de mover a nuestros personajes por una escena y encontrar caminos en un **Navmesh**.



El proceso de crear un **NavMesh** desde la geometría del nivel es llamado NavMesh Baking. El proceso colecciona los Render Meshes y Terrenos de todos los Gameobjects que pueden ser marcados como Navigation Static, y luego los procesa para crear un navigation mesh que aproxima las superficies que se pueden caminar del nivel.

El NavMesh resultante será mostrado en la escena como una superposición azul en el nivel subyacente de la geometría cuando la ventana de Navigation esté abierta y visible.

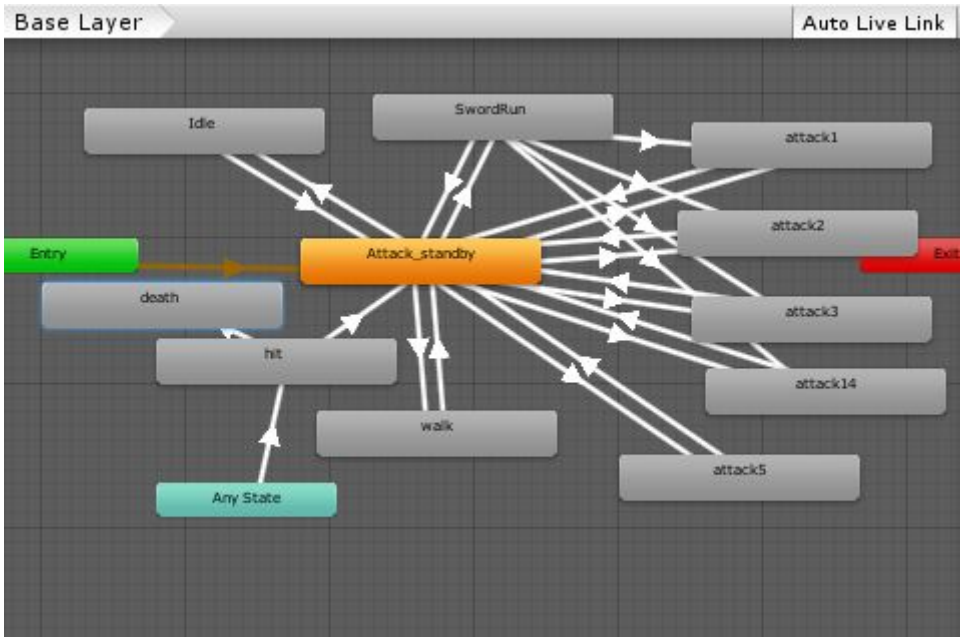


El NavMesh representa el área donde se puede mover el centro del agente. Conceptualmente, no importa si considera al agente como un punto en un NavMesh encogido o un círculo en un NavMesh de tamaño completo, ya que los dos son equivalentes. Sin embargo, la interpretación del punto permite una mejor eficiencia en tiempo de ejecución y también permite al diseñador ver de inmediato si un agente puede pasar por las brechas sin preocuparse por su radio.

6.2.2 Propiedades del IA

La IA (Monstruos) mencionado anteriormente contiene diferentes estado (estado de patrulla, de ataque etc), y el componente Animator es utilizado para asignar una animación al IA en la escena.

Animator del IA :



El estado inicial del IA es el estado de patrulla donde moverá dentro de unos rangos o puntos establecido. Cuando el jugador entra al rango de vista del IA, empieza a perseguir al jugador cambiando la animación al “SwordRun”. Si se acercara a un cierto rango de distancias al jugador ataca al jugador.

El IA contiene otros componentes como **Capsule Collider** para detectar colisiones como cuando se colisiona con una arma, donde recibe daños del arma que se ha entrado al contacto con la cápsula.La arma que lleva encima también tiene un **Capsule Collider** que se habilita cuando realiza un ataque.

Capítulo 7
Conclusiones y líneas futuras

La creación de un videojuego requiere conocimientos de diversos ámbitos como programación, diseñador de arte, banda sonoras, etc.Pero lo principal es la pasión y la perseverancia en este área ya que el juego no se crea ni un día para otro, hay que familiarizarse con la división del trabajo y el proceso en el desarrollo de juegos, dominar el uso de diversas herramientas de desarrollo. Ser capaz de expresar lo que quieres en diagramas de flujo y prototipos de forma intuitiva, familiarizado con el proceso de desarrollo y prueba del juego.

Otro puntos importante es el concepto de diseño del juego, la meteorología son unos teorías que nos guía la creación del comportamiento del juego, pero el concepto de diseño del juego es la fuerza principal que nos impulsa a crear el juego y nuestro objetivo principal.

Los conocimientos básicos que he dominado sobre el uso del motor Unity me ha permitido indagar más sobre los diferentes aspectos relacionado con el desarrollo de los videojuegos, el proceso desarrollo, de producción etc. Se puede incluido en el juego las ideas que quieres expresar, ya que el juego no solo proporciona diversiones sino algunos pensamientos del autor, es una nueva forma de expresarse la ideología.

Me gustaría seguir desarrollando el juego, ampliando los contenido en sí, como generar mapas procedurales para el sistema de mazmorra, incluir algunos aspectos relacionado con un sistema sofisticado de cadenas de misiones. Muchas veces los juego son atraedor sobre todo es por la historia que se ofrece, es una buena idea de crear contenidos respecto a este tema, añadiendo algunos ámbitos relacionado con algunos rompecabezas para que sea más divertidos. Al final me gustaría trabajar más en el tema visual del juego, añadiendo más contenidos artístico para un mejor visual.

Capítulo 8
Summary and Conclusions

The creation of a video game requires knowledge about various areas such as programming, art designer, soundtrack, etc. The main thing to involve this is passion and perseverance in this area since the game is not created overnight, you have to get familiar with the division of labor and process in game development, the use of various development tools. Being able to express ideas in flowcharts and prototypes, familiarize with the game development and testing process.

Another important point is the concept of game design, meteorology are theories that guide us in creating game behavior, but the concept of game design is the main ideology that drives us to create the game and our main objective.

The basic knowledge that I have got about the use of the Unity engine has allowed me to investigate more about the different aspects related to the development of video games, the process, the production, etc. You can include some ideas that you want to express in the game, since the game not only provides diversions, this will include some author thoughts , it is a new way of expressing ideology.

Capítulo 9

Presupuesto

Para la realización de este proyecto ha sido necesario alrededor de 350 horas de trabajos. Siendo el trabajo para un puesto de formación superior universitario, el promedio del coste de trabajo es alrededor de 20€/hora de suelto para profesionales de investigación y asumiendo que los costos de los materiales son resueltas por la universidad de la laguna sin añadir costes adicionales. La realización del proyecto resulta en total un 7500€. Estos sueltos se puede ver desglosado en la sección 7.1.

7.1 Coste del proyecto

Número de hora de trabajo	**350**
Coste por hora	**10€**
Total	**7.500€**

Capítulo 10

Anexos

Repositorio del proyecto : https://github.com/samjxxxx/UnityTFG

Bibliografía

- https://docs.unity3d.com/ScriptReference/
- https://learn.unity.com/tutorial/create-your-first-unity-project?language=en&courseId=5c59cf22edbc2a001f59aa5d
- https://learn.unity.com/tutorial/beginner-walkthroughs?language=en&courseId=5c59cf22edbc2a001f59aa5d
- https://www.youtube.com/watch?v=q-FPR1I2B74
- https://gamedevacademy.org/unity-3d-animation-tutorial/
- https://forum.unity.com/threads/how-to-develop-a-quest-system-process-ways.92902/
- https://answers.unity.com/questions/703193/a-simple-quest-system.html
- https://docs.unity3d.com/ScriptReference/CharacterController.Move.html
- https://docs.unity3d.com/es/530/ScriptReference/CharacterController.Move.html
- https://www.youtube.com/channel/UCq9_1E5HE4c_xmhzD3r7VMw
- https://medium.com/@yonem9/create-an-unity-inventory-part-1-basic-data-model-3b54451e25ec
- https://learn.unity.com/tutorial/adventure-game-phase-2-inventory-system
- https://answers.unity.com/questions/1414977/enemy-ai-movement-1.html
- https://www.youtube.com/watch?v=2WnAOV7nHW0
- https://learn.unity.com/tutorial/3d-physics
- https://www.youtube.com/watch?v=ym_DuyruVME
- https://docs.unity3d.com/ScriptReference/Collision-gameObject.html
- https://blogs.unity3d.com/es/2019/05/14/introducing-the-animation-rigging-preview-package-for-unity-2019-1/
- https://www.youtube.com/watch?v=o8RD0UGemp8
- https://docs.unity3d.com/es/2018.4/Manual/UICanvas.html
- https://www.youtube.com/watch?v=EDh2DGgSN1Y
- https://www.youtube.com/watch?v=ejp1pooM2rE
- https://www.reddit.com/r/Unity3D/comments/1q807z/what_would_be_the_easiest_way_to_make_a_quest/
- https://www.youtube.com/watch?v=38PUho9J6ps
- https://www.youtube.com/watch?v=_nRzoTzeyxU
- https://www.youtube.com/watch?v=f-oSXg6_AMQ
- https://www.reddit.com/r/Unity3D/comments/824dwf/dialogue_systems_frustration_need_advice/
- https://github.com/Brackeys/Dialogue-System
- https://www.youtube.com/watch?v=welWr2OFMw4
- https://jayanam.com/unity-3d-dialogue-system-tutorial/
- https://www.youtube.com/watch?v=xppompv1DBg
- https://answers.unity.com/questions/938221/basic-enemy-ai-in-c.html
- https://answers.unity.com/questions/274809/how-to-make-enemy-chase-player-basic-ai.html
- https://docs.unity3d.com/es/2019.4/Manual/nav-BuildingNavMesh.html
- https://github.com/L1247/Unity-QuickSheet-ByEXCEL
- https://answers.unity.com/questions/386129/is-this-a-good-way-to-script-an-inventory-system.html
- https://forum.unity.com/threads/tutorial-creating-an-inventory-with-scriptable-objects.751910
- https://www.mixamo.com/#/?page=1&type=Motion%2CMotionPack