industriales
etsii

Escuela Técnica Superior de Ingeniería Industrial

# UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería Industria

# Desarrollo de librerías de soporte para el desarrollo de videojuegos en Unity para niños y adolescentes

**TRABAJO FIN DE GRADO**

GRADO EN INGENIERIA TELEMATICA

**Autor:** **Giménez Martínez, Pedro**
Director: Egea López, Esteban

Universidad Politécnica de Cartagena

# Índice

# 1.Introducción

El objetivo a cumplir dentro del proyecto es desarrollar una librería que sea capaz de aportar a los jóvenes programadores las funcionalidades básicas dentro de un videojuego, para que éstos sean capaces de comenzar a programar habiendo obtenido la base previa de manera que desarrollarán sus proyectos gracias a que se les aportará multitud de herramientas comunes en los videojuegos tales como:

- El control de personaje
- El control de la cámara
- Un sistema básico de vida para la destrucción de objetos
- Un sistema de recolección
- Múltiples Sistemas para el manejo de objetos
- Establecimiento de la interfaz
- Distintos efectos que podrán añadirse a distintos objetos
- Modos de juego básicos para fomentar la creación
- Un sistema de puntuación
- Árboles de comportamiento para la creación de *IAs* (inteligencia artificial).

Todos estos sistemas mencionados darán libertad a los jóvenes programadores o niños en los colegios interesados en la tecnología a realizar sus proyectos en base a la modificación de los sistemas mencionados.

Como herramienta ha sido utilizada para la programación la aplicación de Unity ésta nos permite usar su propio motor de físicas, el cual aporta los componentes básicos en un videojuego. Dentro de Unity haremos uso de distintos componentes tales como Microsoft Visual Studio para el desarrollo de scripts personalizados e interconectados, y Behavior Bricks para la creación de inteligencias artificiales dentro de los videojuegos.

Con las herramientas dadas los jóvenes programadores podrán adquirir las habilidades básicas para poder iniciarse en el mundo de los videojuegos y poder plasmar sus propias ideas y gustos en los videojuegos que ellos creen, aportando una base con las cuales podrán adquirir nuevos conocimientos y que puedan aplicarlos de manera intuitiva en sus propios proyectos para que en un futuro se puedan desarrollar como programadores, con las librerías hechas en el proyecto el programador será capaz de realizar multitud de tareas dentro del ámbito de los videojuegos algunas de estas tareas serán tales como desarrollar inteligencias artificiales basadas en un árbol de comportamiento, crear personajes controlables con una cámara que les siga y multitud de herramientas tales como un inventario funcional, puertas, modos de juego… De manera que el programador tendrá la capacidad de crear todo lo que quieran en base a la combinación de los distintos elementos dados.

Esta librería permitirá que los más jóvenes, con la ayuda ofrecida podrán centrarse en los elementos gráficos, tanto para los objetos como para el diseño de niveles, haciendo uso de distintos programas de diseño gráfico (tales como el Paint), mientras que aquellos que se inicien en el lenguaje de programación podrán crear sus propios *scripts* y añadirlos a la librería para que estos funcionen en conjunto a los ya definidos dentro del proyecto.

# 2.Tecnologías Usadas

Como herramienta para el desarrollo del proyecto ha sido seleccionado como *Software* el programa Unity, gracias a que es un programa de libre acceso para la creación de videojuegos y con multitud de herramientas que serán de gran ayuda a lo largo del proyecto.


*Ilustración 1: Logotipo Unity*

El funcionamiento dentro del programa Unity, se basa en la creación de una escena que representara el nivel del videojuego y nuestro programador a través de la interfaz dada por Unity, será capaz de añadir los distintos *GameObject*, los cuales representaran los objetos creados por el mismo programador, este mismo programador les adherirá *Scripts*, código desarrollado por el programador que dará comportamiento al objeto que se le añada, o distintos componentes básicos ya implementados de base dentro de Unity, una vez creada la escena si se desea se podrá ejecutar el motor gráfico de Unity para iniciar la escena y realizar distintas pruebas conforme avanza el desarrollo del videojuego.

Unity nos ofrece gran cantidad de componentes para el desarrollo de aplicaciones de videojuegos tanto para 2D como para 3D, el proyecto ha sido pensado en el uso de aplicaciones 3D, algunos de los componentes principales que han sido usados a lo largo del proyecto para la creación de distintos objetos y elementos (dentro de Unity los distintos objetos y elementos son denominados *GameObject*) han sido:

- *Collider*
- *Rigidbody*
- *Script*

Una vez mencionados los componentes más importantes usados a lo largo del proyecto, cabe destacar que, una vez creado el objeto, Unity permitirá almacenar ese objeto en un fichero para poder guardarlo y hacer uso de el de distintas maneras dentro de la aplicación, este objeto guardado en un fichero será denominado *Prefab*.

**Collider:**

Este componente será aquel que permitirá que los objetos sean capaces de detectar colisiones entre otros objetos, su funcionamiento se basara en el uso de un área invisible alrededor del objeto que envolverá al mismo para detectar las colisiones con su contacto, dentro de Unity existen distintas versiones de este componente, que se diferenciaran en la forma de área que define, las cuales serán:

- **Box Collider**: el área definida será con forma de cubo.
- **Sphere Collider**: el área definida será con forma de esfera.
- **Capsule Collider**: el área definida será con forma de capsula.
- **Terrain Collider**: el área definida será con la forma de la superficie del objeto.
- **Mesh Collider**: el área definida será con forma de malla.

En el proyecto ha sido de gran importancia añadir distintos *Collider* a multitud de objetos para la detección de colisiones que será usada en gran medida a lo largo de la biblioteca.

**Rigidbody:**

Este componente dará a los objetos del proyecto la capacidad de actuar bajo la física, dado que podrán recibir fuerzas para que estos se puedan mover de una manera más realista, estas fuerzas pueden ser tales como la fuerza de la gravedad o las fuerzas agregadas a través del uso de distintos *scripts*.

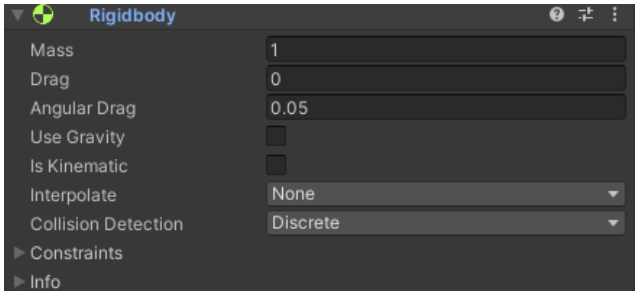


*Ilustración 2: Componente Rigidbody*

Algunas de las variables del objeto que podemos definir serán: la masa (*Mass*), la resistencia al aire tanto normal como angular (*Drag*), si usaremos la gravedad (*Use Gravity*) o si el objeto no tiene la física activada (*Is Kinematic*). Dentro del proyecto algunos de los usos aplicados a este componente son para emular el lanzamiento de objetos o hacer uso de la misma fuerza de gravedad.

**Script:**

Los *scripts* serán aquellos componentes que han sido creados de manera manual para poder definir aquellos comportamientos que uno desee, estos componentes pueden ser añadidos como los componentes previamente mencionados.

Para su creación será usado el lenguaje C#, y se harán uso de distintos procesos derivados de una clase denominada *MonoBehaviour*, con la cual se implementarán multitud de funciones ordenadas y gestionados por multitud de eventos los cuales están definidos por la imagen de la ilustración 3.

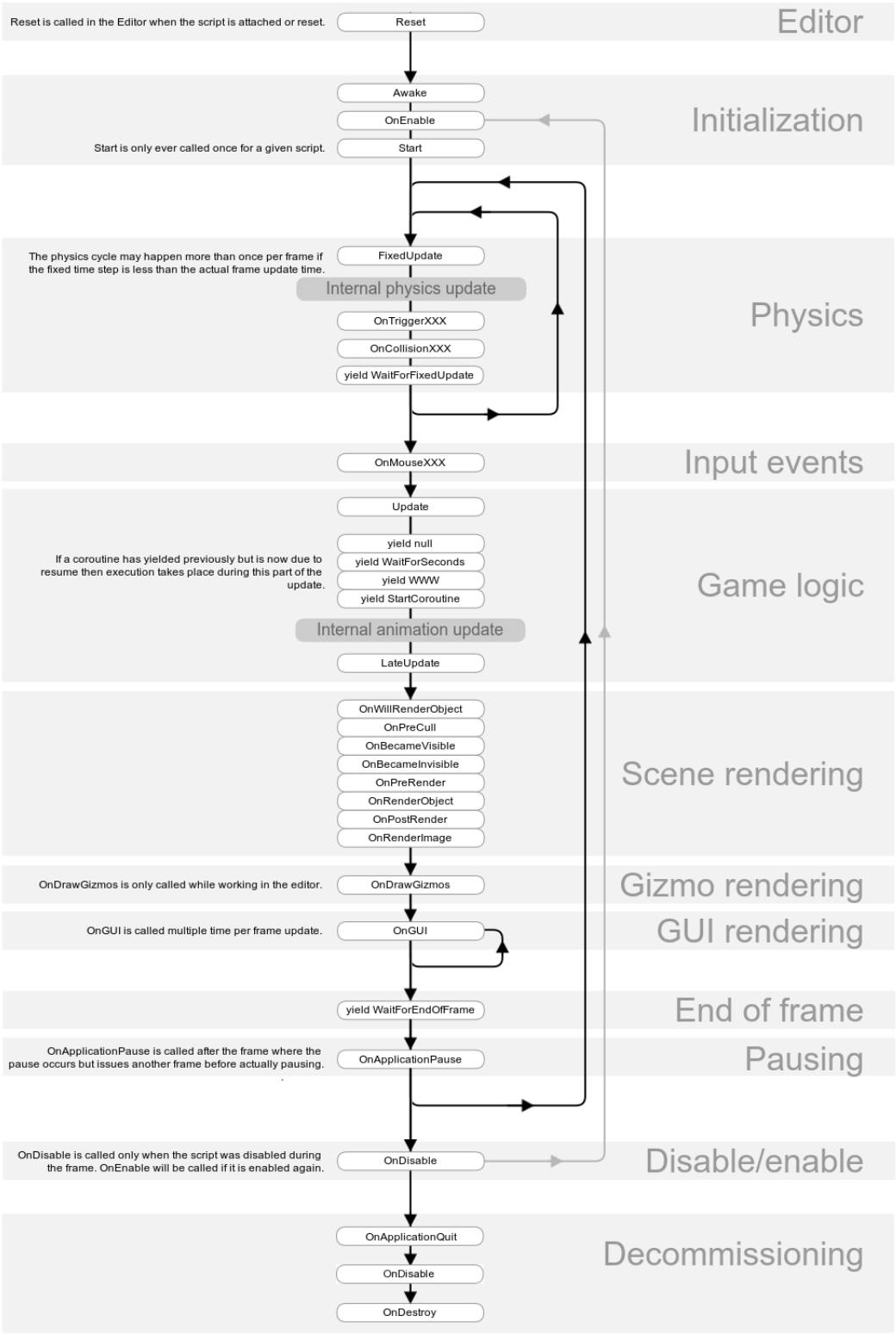


*Ilustración 3: Eventos de los scripts*

A la hora de crear un *script* haremos uso de las funciones de la ilustración, llamándolas dentro del *script* y modificándolas para su uso personalizado, de manera que podemos crear un comportamiento personalizado haciendo uso de los distintos procesos por los que pasa un script, algunos de estos procesos a destacar son:

- **Awake ()**: Sera llamado una sola vez al activar el objeto.
- **Start ()**: Sera llamado antes de la primera llamada de *Update*
- **Update ()**: Sera llamado una vez por fotograma.
- **OnCollision ()**: Sera llamado cuando se detecte una colisión en el objeto.
- **OnDestroy ()**: Sera llamado cuando el objeto se destruya.

# 2.1. Microsoft Visual Studio

Microsoft Visual Studio es la herramienta usada durante el proyecto para poder programar los distintos scripts que han sido utilizados en el proyecto, este programa es un entorno de programación que nos permite manejar distintos lenguajes de programación, en el proyecto ha sido elegido el lenguaje de C#.


*Ilustración 4: Logotipo Microsoft Visual Studio*

# 2.2. Behavior Bricks

Behavior Bricks es una herramienta de Unity, que se puede obtener a través de la tienda de *asset* de Unity de manera gratuita, esta herramienta nos permitirá crear a través de una interfaz nueva distintos arboles de comportamiento (*Behaviour Tree*),un árbol de comportamiento es una tecnología comúnmente usada en los videojuegos para dar una *IA* ( Inteligencia artificial) a un *NPC* ( personaje no jugable) por el cual tendrá la capacidad de tener un comportamiento más complejo, gracias al uso de tareas más básicas las cuales se irán ejecutando con respecto a las situaciones expuestas en el árbol de comportamiento, los cuales podemos asociar a un objeto usando un *script* que estará incluido en los ficheros de la herramienta.


*Ilustración 5: Logotipo Behavior Bricks*

A la hora de crear un árbol de comportamiento, el editor consistirá en un sistema de bloques que juntando distintos bloques podremos crear el comportamiento que realizara la *IA*, estos bloques están organizados dependiendo de la función que realicen en el árbol de comportamiento:

- **Acciones**: Cuando el árbol llegue a este tipo bloque realizara la acción descrita por el mismo bloque con los parámetros añadidos, dentro del editor encontramos los siguientes grupos de acciones que se podrán usar:
    - **Animación**: Si queremos que el objeto ejecute una animación.
    - **Audio**: Podremos ejecutar un audio en esta situación.
    - **Básico**: Definir alguna variable cuando lleguemos a la acción o que el objeto se quede en espera.
    - **GameObject**: Acciones relacionadas con distintos objetos para poder interactuar con ellos.
    - **Navegación**: Acciones que darán movimiento al objeto.
    - **Physics**: Acciones que darán velocidad o fuerza al objeto.
    - **Vector3**: Acción que devolverá una posición aleatoria.
- **Behaviour**: Dentro de este bloque podemos encontrar los distintos arboles de comportamiento que hay dentro del proyecto, dado que se podrán añadir como bloques para extender el comportamiento de la *IA*.
- **Composiciones**: Estos bloques tendrán la función de definir el orden de ejecución de los distintos bloques del árbol, definiendo cual se ejecutará primero, si se repetirá un fragmento del árbol...
- **Condiciones**: Las condiciones serán bloques que se podrán adherir a otros bloques para definir cuando se podrán ejecutar los mismos bloques en las situaciones definidas por el bloque de condiciones, un ejemplo de estos bloques son los de percepción que se basaran en medir la distancia con un objeto definido por el programador.

# 3. Scripts

La librería desarrollada consta de multitud de *scripts* organizados en distintas carpetas distribuidos con respecto a la funcionalidad que ofrecen al programador, cada carpeta presentando un ámbito desarrollado en la aplicación:

- Controladores
- Sistemas añadidos
- Modos de juego base
- Scripts vida
- Scripts puntuación
- Efectos

Los *scripts* de la librería pueden funcionar interconectados entre sí, y pueden combinarse entre sí para buscar un efecto deseado por el programador, por lo que se podrán situar en un mismo objeto para aplicar distintas funcionalidades del mismo, permitiendo al programador usar su propia imaginación para la creación de distintos objetos y permitiendo la comunicación entre ellos usando los scripts planteados.

# 3.1. Controladores

Los *Scripts* definidos en el ámbito controladores confeccionan principalmente todas aquellas funciones básicas en todo tipo de videojuegos, más concretamente el movimiento básico del mismo, el control de la cámara y la capacidad de pausar el juego a voluntad del jugador.

- **PlayerController**: Este *Script* definira el movimiento que realizara el jugador en el mismo videojuego, para su correcto funcionamiento sera necesario que se incluya al mismo objeto una camara y un *Character Controller*, su funcionamiento se basa principalmente en el uso de *Update* de Unity donde se llamara constantemente a una funcion denominada *move*(), dentro de esta función se realizara una comprobación de las teclas verticales y horizontales (WASD) y se realizara el movimiento con respecto a la camara, tambien se comprobara si el objeto esta en el suelo y si se pulsa la tecla "*Jump*" ( Espacio) para comprobar si es capaz de realizar un salto, y como añadido se a creado un metodo denominado *autoJump()* en el caso de que el programador quiera realizar un salto en otra situación.


*Ilustración 6: Parámetros PlayerController.*

- **CameraControl:** El *script CameraControl* se basará principalmente en el control de la cámara, más concretamente la función de esta cámara es mantener fijo un objetivo (*target*), opcionalmente podremos hacer que siga al objetivo y permita un control vertical con el *mouse*, junto al objetivo se puede personalizar una serie de parámetros que permitirá personalizar la cámara, ya sea estableciendo distancia con el objetivo, creando una sensibilidad de movimiento y el valor de interpolación para dar más suavidad a la cámara (*LerpValue*).

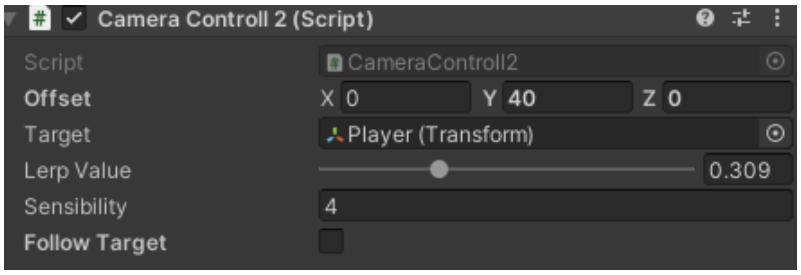


*Ilustración 7: Parámetros CameraControl.*

- **PauseMenu:** Este *script* se encargara de todas las opciones básicas dentro de un menú de pausa, específicamente la opción de pausar y restaurar el videojuego haciendo uso del *timescale* ( deteniendo también la cámara con una variable *bool*) de base esta especificado que el control de esta pausa será con el botón "P" del teclado, otras opciones añadidas a este menú son: reiniciar la escena que el programador especifique y salir del videojuego, todos estos elementos es recomendables añadirlos dentro de un *canvas* personalizado con distintos botones para su uso dentro del juego.

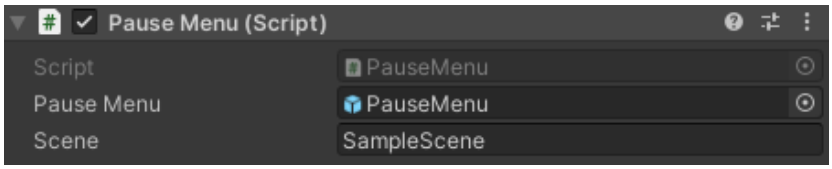


*Ilustración 8: Parámetros PauseMenu.*

# 3.2. Sistemas Añadidos

Los *Scripts* sistemas añadidos son todos aquellos que definen acciones más secundarias con una mayor personalización, permitiendo al programador poder hacer uso de ellos con respecto a las necesidades de su proyecto de esta manera podrá hacer más único su proyecto.

- **InventorySystem**: La función principal de este *script* se basa en añadir objetos donde quiera el programador, para ello el programador podrá incluir en una lista sus propios *prefabs* e indicar donde quiere que estos aparezcan para que el propio jugador con la ruleta del ratón sea capaz de usar los *prefabs* que el programador le haya añadido, podemos destacar dentro del *script* distintos métodos para desactivar y activar distintos *prefabs* y que solo este activo uno.

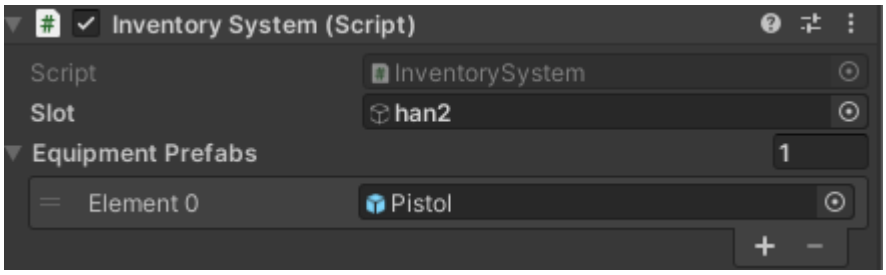


*Ilustración 9: Parámetros InventorySystem*

- **GrabSystem**: Permite el programador dar al jugador una mecánica por la cual este podrá agarrar objetos ("e") y soltarlos ("r"), el funcionamiento por código de esta mecánica se basa en añadir un punto de agarre donde se cogerán los objetos, los cuales tendrán que usar el tag "*Grab*" si quieren ser agarrados, como condición para su funcionamiento el objeto con el tag debe tener un *rigidbody* y este *script* estar en el mismo punto de agarre.

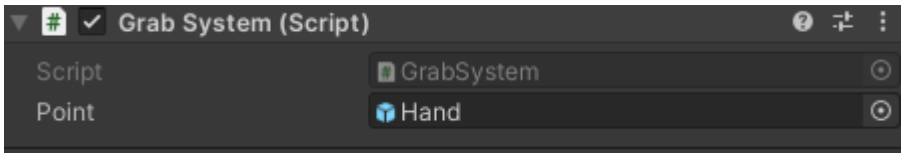


*Ilustración 10: Parámetros GrabSystem*

- **Shoots:** Dentro de sistemas añadidos podemos encontrar el fichero *Shoots* que compendia todos los *scripts* relacionados con un sistema de disparo, dentro de ello podemos encontrar los siguientes *scripts*:
    - **SimpleShootSystem:** el funcionamiento de este *script* se basa en cuando se pulse la tecla de disparo (denominada k) este revisara si podrá disparar (contiene un elemento *ammoSystem* y este le permite), y el funcionamiento del disparo se basa en instanciar un *prefab* y que este sea impulsado por una fuerza desde un punto denominado *FirePoint*, por el cual el objeto *prefab* seleccionado será lanzado desde el punto denominado *FirePoint* permitiendo personalizar el objeto al gusto del programador.

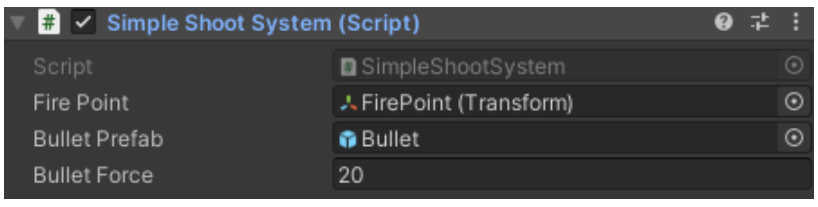


*Ilustración 11: Parámetros SimpleShootSystem*

- **HeldShoot**: Este *Script* es una variación del *SimpleShootSystem*, debido a que hereda el método *shoot* y las mismas variables, añadiendo una variable que determinara el tiempo entre *shoots*.

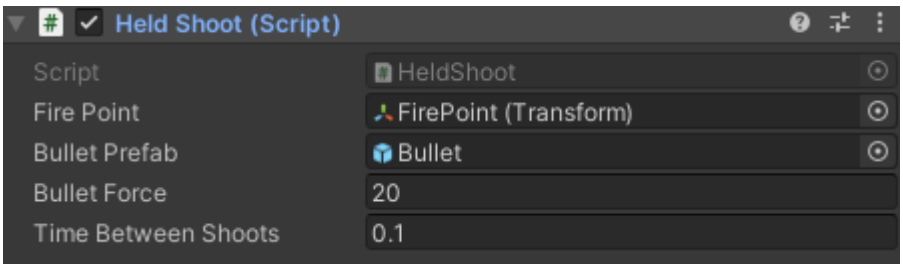


*Ilustración 12: Parámetros HeldShoot*

- **AmmoSystem**: Este *Script* tiene que estar anexionado al mismo objeto que tenga cualquiera de los dos *scripts* anteriores debido a que determinara las veces que se podra llamar al metodo *Shoots* y que se pueda ver visualmente por la interfaz si se puede disparar, tambien si el programador lo desea podra limitar esta capacidad según guste y añadir un *id* para poder usar de identificador en el siguiente *script*.

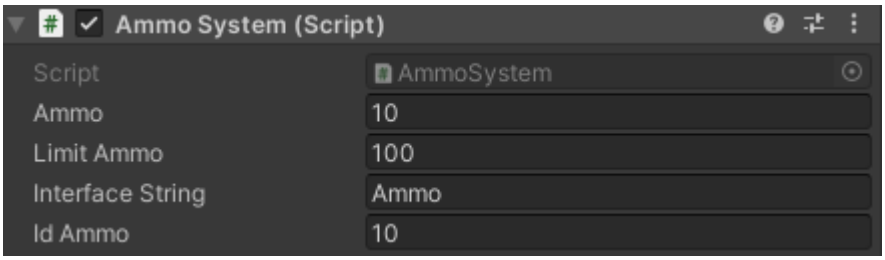


*Ilustración 13: Parámetros AmmoSystem*

- **RechargeAmmo**: Este *script* podra reponer la capacidad de disparar a *ammoSystem* permitiendo aumentar la variable *ammo* y con un identificador se podra especificar que *ammoSystem* se quiere repone, su funcionamiento se basa en añadir este script a algun objeto que se quiera usar y con el contacto de un objeto especifico se añadira *ammo*( al chocar se podra especificar si este objeto querra ser eliminado).

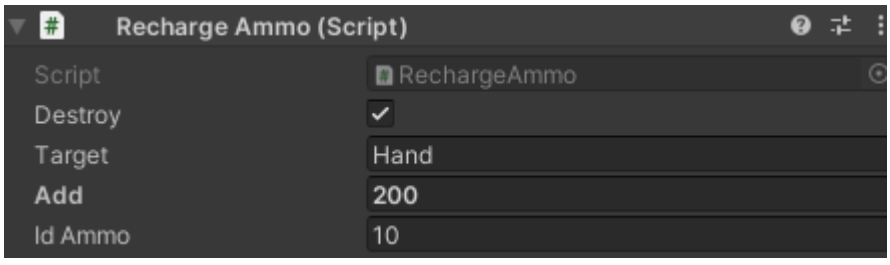


*Ilustración 14: Parámetros RechargeAmmo*

- **KeySystem**: En esta carpeta podemos encontrar todos los scripts relacionados con un sistema de llaves creado para sus distintos usos:
    - **KeyInventorySystem**: este sencillo *script* es donde se almacenará una lista de *string* que determinará las *Keys* que el jugador tenga y el método para añadirlas.

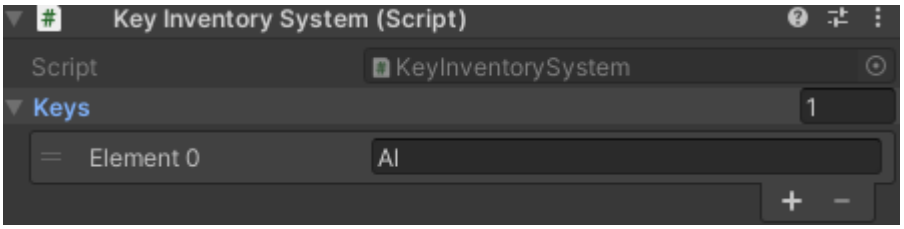


*Ilustración 15: Parámetros KeyInventorySystem*

    - **Door**: Este *script* es un añadido para poder crear puertas y que se puedan abrir, si en la lista anterior esta añadida la *key* de la misma puerta que se quiere abrir, la manera de apertura esta programada a traves del choque con un objeto.

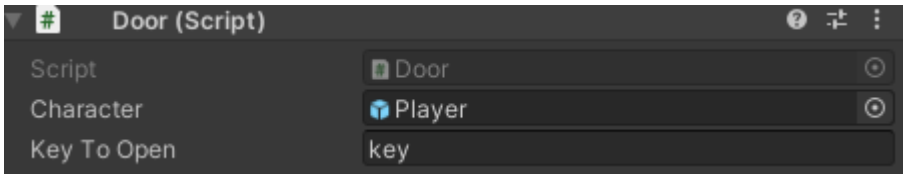


*Ilustración 16: Parámetros Door*

- **CollectItems**: Esta carpeta contendrá todos los elementos con los que podremos reunir los objetos y *keys* en sus respectivos inventarios y establecer un punto de recogida.
    - **Collector**: Este *script* es la base seguida por los siguientes *scripts* que estarán anexionados al objeto a recoger, puesto que heredarán sus métodos y permitirá dar la opción a que el objeto se destruya.
    - **CollectorGameObject**: En esta situación se añadirá al *inventorySystem* un objeto *prefab* establecido por el programador.


*Ilustración 17: Parámetros CollectorGameObject*

- **CollectorKey**: En este *script* el programador escribira un *string* que sera la *key* que el jugador podra recoger.

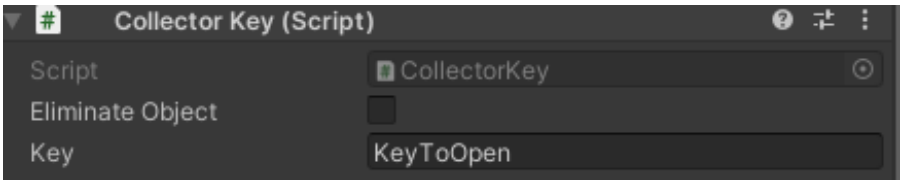


*Ilustración 18: Parámetros CollectorKey*

- **CollectItems**: Este *script* deberá combinarse con el objeto que haga referencia al propio jugador, dado que permitirá que nuestro jugador al entrar en contacto con algunos de los *scripts* anteriores, ejecute el metodo de recogida que estarán en sus propios *scripts* de manera que el jugador será capaz de recoger objetos y añadirlos al inventario gracias a su uso.

# 3.3. Modos de juego Base

Los *Scripts* de modos de juego base definirán una serie de reglas por las que se regirán el juego que creara el desarrollador, dándole al programador la capacidad de definir qué tipo de videojuego quera desarrollar modificando las distintas reglas impuestas.

- **Score**: Con este *Script* podremos establecer unas reglas básicas de juego basadas en la puntuación, la cual se podrá modificar a través de un método denominado *AddPoints ()* accedido a través de los *scripts* de puntuación, de manera simultánea se le da la opción al programador de poder darle una marca final a su videojuego estableciendo un límite de puntos para que este acabe y la siguiente escena para cargar su juego.

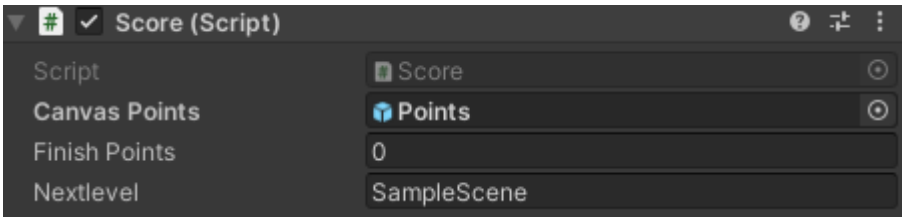


*Ilustración 19: Parámetros Score*

- **SearchKey**: En esta situación el juego planteado se basara en buscar *keys* y añadirlas en *keyInventorySystem*, el programador podra dar como parametro de entrada una lista de *string* que se definiran como las *keys* que debera añadir el jugador para finalizar el juego y el proximo nivel que querra acceder al reunir todas las *keys*, como añadido podremos seleccionar un *canvas* para que se pueda mostrar por la interfaz las *keys* restantes que el jugador necesita.

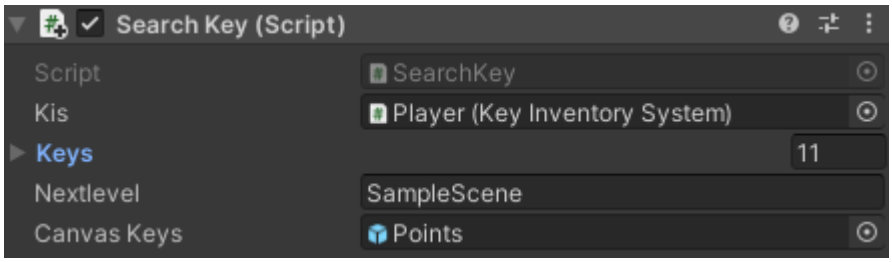


*Ilustración 20: Parámetros SearchKey*

# 3.4. Scripts Vida

Los *Scripts* vida establecen un sistema de salud que se podrá modificar y añadir distintos efectos al objeto que se adhiera, permitiéndole al programador organizar sus *IAs* y objetos de tal manera que podrá programar su eliminación bajo las condiciones que el mismo programador quiera, algunos ejemplos de uso seria la creación de trampas, enemigos, etc.

- **LifePlayer**: El *Script LifePlayer* generara la base de la vida en cuanto al jugador respecta, en este *script* manejaremos la vida como un valor que se podrá ver visualmente con la interfaz, que se ira modificando al entrar en contacto con distintos objetos ya sea manteniendo contacto constante ( si el objeto consta de *trigger* para mantener contacto) o con un choque de objetos, en esta modificación para evitar problemas tendrá un retraso programado por el programador entre distintas modificaciones y solo si estos objetos mantienen un *script Damage* para modificar la vida, si esta llega a 0 se entenderá como una eliminación y se recargara la escena que el programador desee.

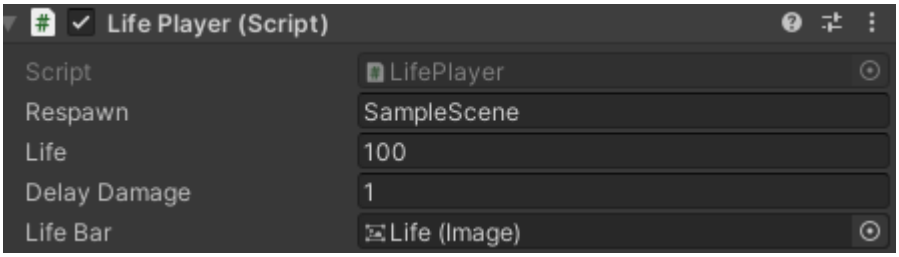


*Ilustración 21: Parámetros LifePlayer*

- **HealthBar**: Este *script* se encargara de darle una barra de vida a entes que no sean el personaje que se controle, esta anexionado a un *prefab* que indicara visualmente la vida a través de una barra por encima del ente, como parámetros estableceremos la vida máxima que este llevara y la vida actual que tendrá de manera que esta vida se podrá modificar, dentro del *update* de Unity ha sido definido dos métodos, uno para modificar los parámetros con la vida actual y otro para apuntar constantemente a la cámara y sea más fácil visualizar la barra, cuando la vida llegue a 0 este objeto será destruido.


*Ilustración 22: Parámetros HealthBar*

- **Damage**: Este *script* se encargará de modificar la vida por contacto con el objeto, como parámetros de entrada de este *script* podemos definir: *damage* (la cantidad de daño que realizara), *damageTime* (cuanto tiempo tardara en hacer ese daño) y *currenDamageTime* (parámetro usado en los distintos *DamageEffect*), como añadido se ha aplicado una serie de variables *booleanas* para añadir los distintos *DamageEffect* y usar estos mismos parámetros para su uso.


*Ilustración 23: Parámetros Damage*

- **Bullet**: Este *script* definirá el comportamiento que tendrá los objetos denominados bala, creado principalmente para juegos basados en disparos o arcade, la principal función a destacar dentro de este *script* es que al entrar en contacto con un objeto y tener un *script* de vida este bajara el valor de la vida del objeto que entre en contacto, este valor que bajara puede ser diferente si es el jugador o si es un objeto, añadido a su comportamiento se ha complementado el *script* con un reciclaje de 5 segundos para evitar acumulaciones de este objeto y que no se sature el juego por exceso.

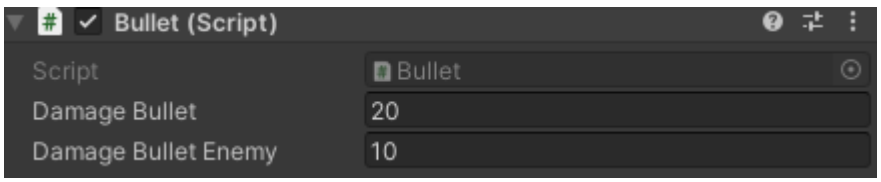


*Ilustración 24: Parámetros Bullet*

- **DamageEffect**: Dentro de los *scripts* de vida encontraremos la carpeta *DamageEffect* que tendrá todos aquellos efectos que se querrán añadir al daño, todos los parámetros de entrada serán los mismos usados en el *script damage*, los *scripts* que encontramos son:
    - **DamageEffect**: Este *script* tendrá la base que heredaran los demás *scripts* puesto que a partir del cual obtendrá el método *execute* donde se generara el efecto.
    - **DelayDamageEffect**: En esta situación cuando el personaje choque con el objeto se iniciará una subrutina que pasado un tiempo establecido (*damagetime*) recibirá el daño.
    - **ContinuousDamageEffect**: Este *script* permitirá que el daño afecte con el tiempo, haciendo que se haga una cantidad de daño repetidas veces en el tiempo.
    - **BlowUpDamageEffect**: Este *script* realizará el daño de forma normal, solo que en esta situación entraremos en el *script player* y el *player* hará un salto automático.

# 3.5. Scripts Puntuación

Los *Scripts* puntuación son aquellos que permitirán modificar de distintas maneras la puntuación definida por el *Script Score* de Modos de juego base, de esta manera el programador podrá establecer distintos objetivos para dar puntuaciones a su juego.

- **AdjustScore**: Este *script* definirá la base de todos los demás *scripts* de puntuación dado que heredarán su método para acceder a *score* y modificarlos con la puntuación que el programador desee dar, cada uno lo hará de una manera diferente.

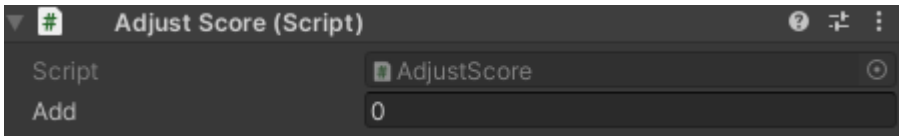


*Ilustración 25: Parámetros AdjustScore*

- **EliminateScore**: En este *script* la situación en la que se añadirá la puntuación será cuando el objeto sea destruido, ya bien por acto del jugador o por situación generada por el programador.

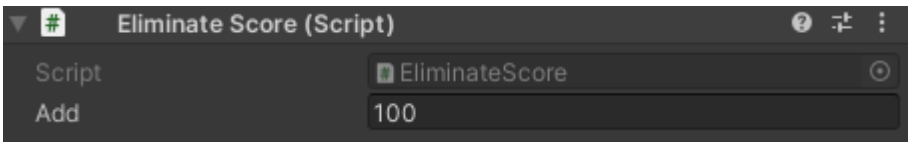


*Ilustración 26: Parámetros EliminateScore*

- **ShockScore**: En este caso la puntuación se añadirá tras entrar un objeto (haciendo uso de su nombre) en contacto con el del *script* y el programador podrá definir si este objeto se destruirá al entrar en contacto.

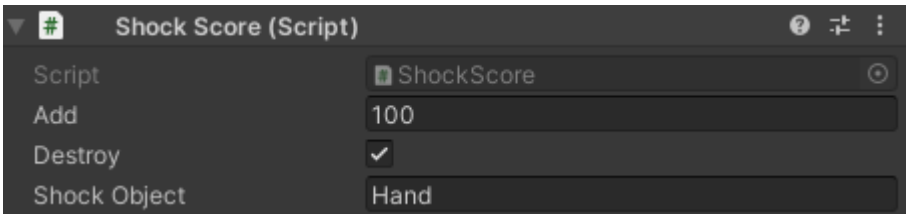


*Ilustración 27: Parámetros ShockScore*

# 3.6. Efectos

Los *Scripts* Efectos son aquellos que en situaciones más específicas desencadenara un efecto establecido por el programador ya sea en un mismo objeto al que se adhiera o en un objeto vacío para definir una situación.

- **DeathDrop**: Este *Script* permite que cuando el objeto al que se adhiera se elimine este en su misma posición generara un objeto prefabricado.

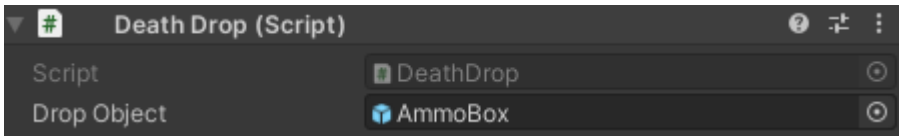


*Ilustración 28: Parámetros DeathDrop*

- **Spawn**: Este *script* permitirá que al objeto añadido cada intervalo de tiempo ira generando objetos prefabricados a su alrededor de manera aleatoria a una distancia máxima, tanto la distancia máxima como el tiempo de fabricación será definido por el programador.

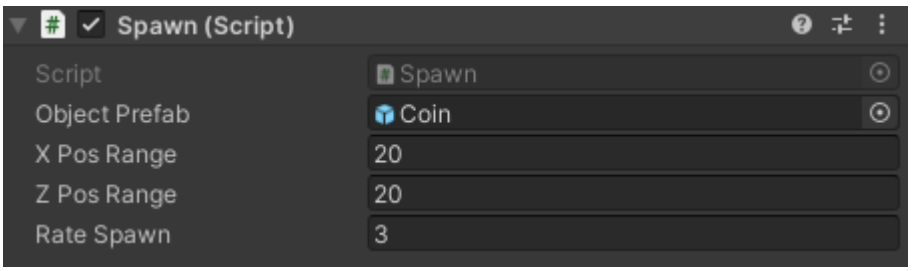


*Ilustración 29: Parámetros Spawn*

# 4. Arboles de comportamiento

A lo largo del proyecto han sido añadidos múltiples arboles de comportamiento haciendo uso de la herramienta Behaviour Bricks, estos árboles de comportamiento aportaran una gran variedad y ayuda en distintas situaciones que el programador este pensando realizar, permitiendo que este pueda crear distintos *NPC* para las situaciones que este desee.

Para hacer uso de estos árboles de comportamiento habrá que asignarle al objeto un *script* denominado *BehaviourExecutorComponent* y asignarle el propio árbol que se desee con los parámetros deseados, si el programador lo desea será capaz de ajustar la tasa de ejecución de tareas o detener el árbol cuando desee.

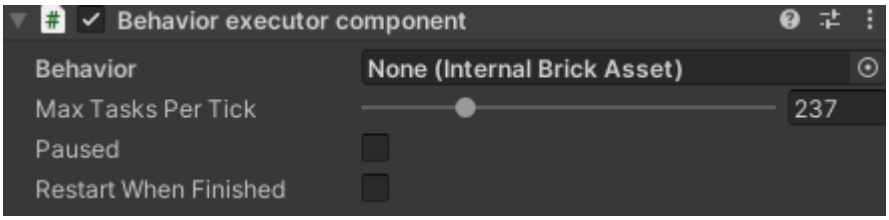


*Ilustración 30: Parámetros BehaviorExecutorComponent*

Dentro de Behaviour Bricks estarán contenidos distintos arboles de comportamiento básicos como ejemplos básicos, algunos de estos árboles son:

- **DoneWander**: Este árbol se usará para que el objeto al que se le aplique este comportamiento se mueva de manera aleatoria en un área definida por el *NavMesh* y seleccionada como parámetro de entrada al árbol, donde primero se calculara la posición a moverse y luego se moverá a esa misma posición.

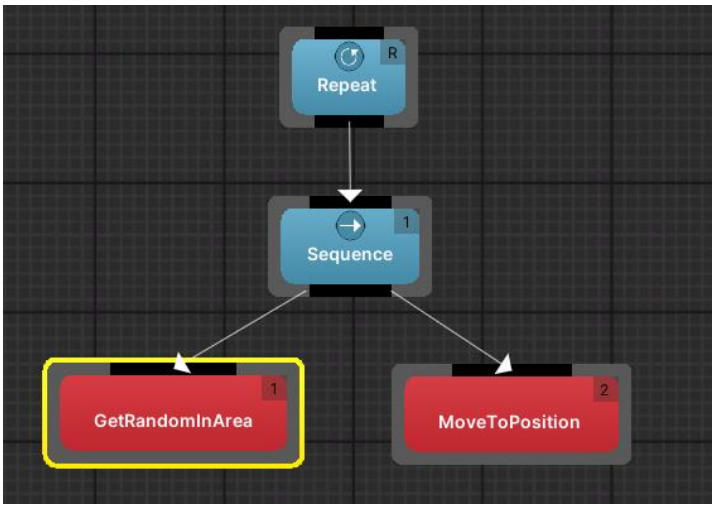


*Ilustración 31: Árbol DoneWander*

- **DoneEnemyBehavior**: Este árbol presentara el comportamiento de un enemigo que se moverá de manera aleatoria usando, el árbol previamente mencionado y cuando detecte al objeto asignado para seguir este se moverá a su posición.

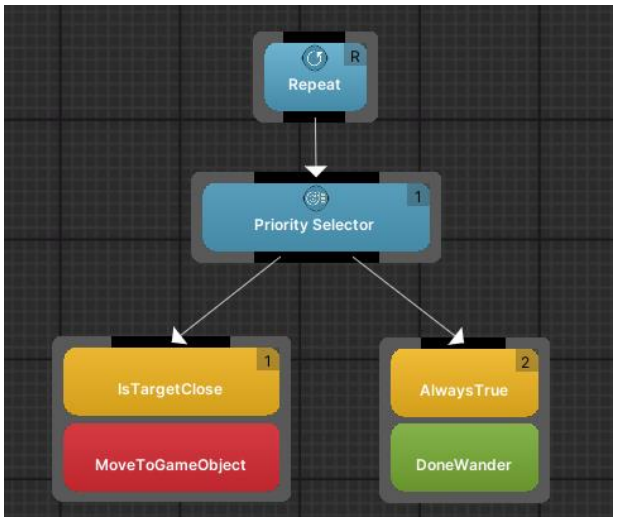


*Ilustración 32: Árbol DoneEnemyBehavior*

En este árbol los parámetros de entrada usados son el objeto a detectar y el área asignada para moverse de manera aleatoria.

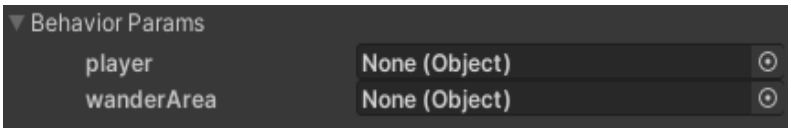


*Ilustración 33: Parámetros DoneEnemyBehavior*

- **DoneClickAndGo**: Este comportamiento se basará en una secuencia de acciones, que harán que el objeto se mueva hacia el ratón una vez se pulse el *click* derecho y cuando acabe reiniciara el comportamiento de este árbol, permitiendo que el objeto sea movido gracias al ratón, habrá que definir la cámara usada.

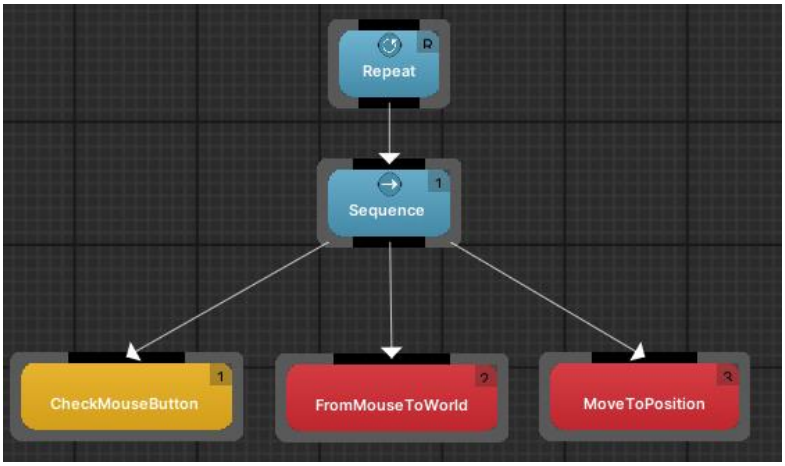


*Ilustración 34: Árbol DoneClickAndGo*

- **DoneAbortableClickAndGo**: Este árbol presentará el mismo comportamiento anterior, solo que con la diferencia de que si volvemos a pulsar *click* derecho la nueva posición definida por el ratón será sobrescrita, redirigiendo al objeto a la nueva posición.

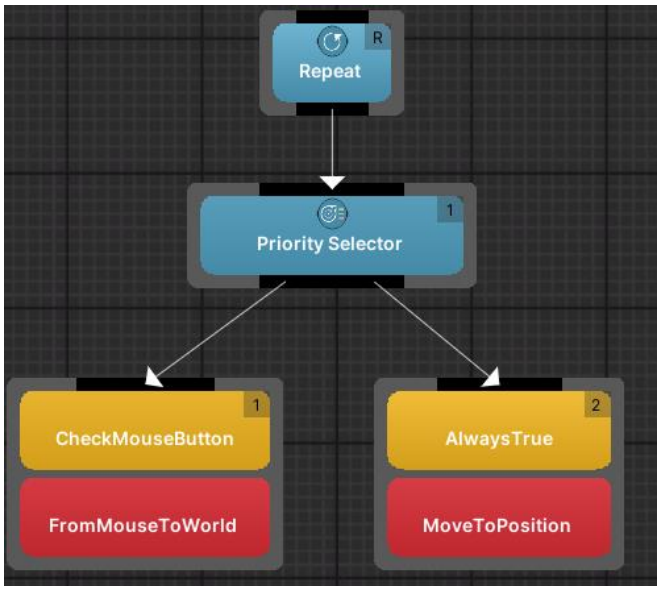


*Ilustración 35: Árbol DoneAbortableClickAndGo*

- **DoneEnemyBehaviour**: Este árbol presentará una *IA* más avanzada a las anteriormente mencionadas, se dará un comportamiento basado en los siguientes pasos:
    1. Si no detecta al objetivo se moverá de manera aleatoria haciendo uso del árbol *DoneWander*.
    2. En cuanto detecte al objetivo se moverá a su posición
    3. Cuando este lo suficientemente cerca se parará el objeto y procederá a disparar un objeto definido como bala.
    4. Si detecta que no hay luz este se objetó detendrá toda acción y se quedará quieto.

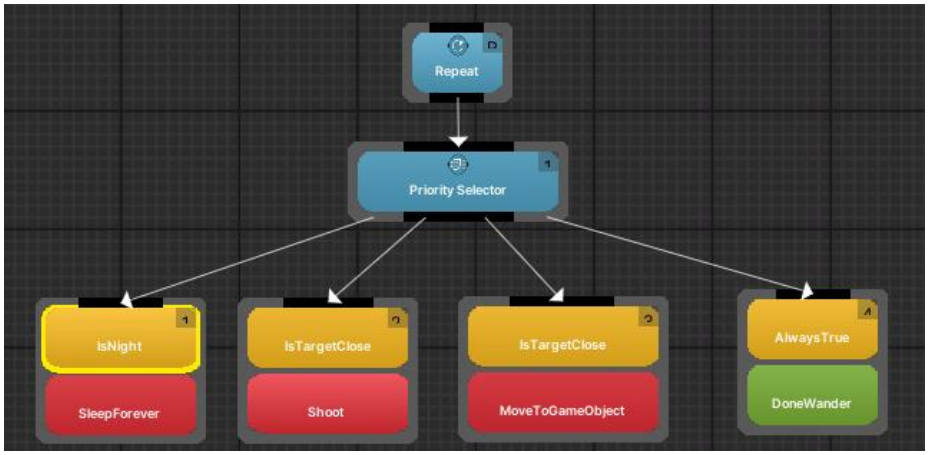


*Ilustración 36: Árbol DoneEnemyBehaviour*

Como parámetros de entrada se deberá añadir al jugador que será objetivo del objeto, un punto de referencia donde se realizará el disparo, el objeto que represente la bala y el área de movimiento (*wanderArea*).

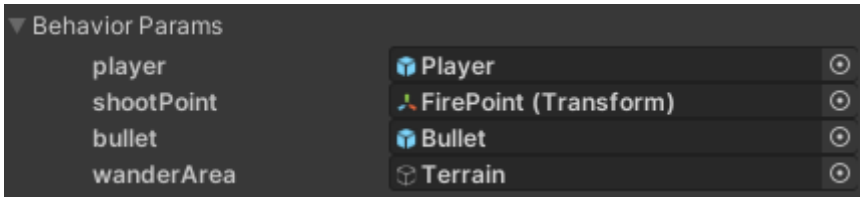


*Ilustración 37: Parámetros DoneEnemyBehaviour*

Aparte de los árboles propios dados por Behaviour Bricks han sido definidos unos árboles nuevos los cuales son:

- **LookAndShoot**: Este árbol presenta un sencillo comportamiento en que el ente asociado a este árbol realizara el siguiente proceso:
    1. Mirara al objetivo asociado.
    2. Esperará un lapso de tiempo establecido.
    3. Disparara el objeto asociado.

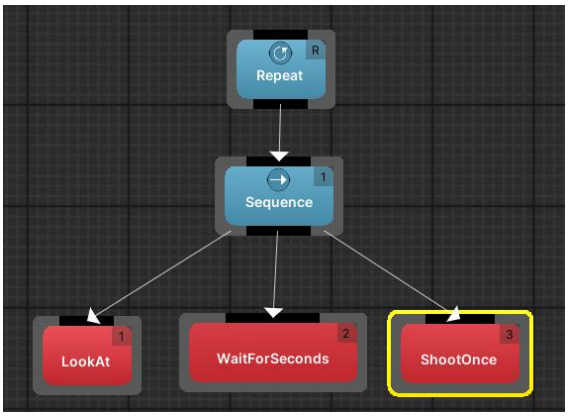


*Ilustración 38: Árbol LookAndShoot*

Todo este proceso se realizará en bucle, los parámetros que nuestro programador deberá establecer serán: un *gameobject* que representara al objetivo, el lapso de tiempo de espera, un punto de referencia donde instanciar y un *gameobject* que representa el objeto que instanciara como disparo.

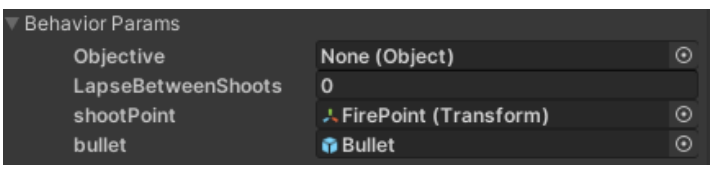


*Ilustración 39: Parámetros LookAndShoot*

- **PistolEnemy**: Este comportamiento presenta una complejidad mayor al anterior dado que en este habrá una prioridad dependiendo de la distancia, dándose la siguiente situación en bucle dependiendo de la distancia del objetivo:
    1. Si el objetivo está a una distancia por la cual el NPC no es capaz de detectarlo este tomara el comportamiento de patrullar una zona establecida con movimientos aleatorios alrededor de ella, haciendo uso del árbol *DoneWander*.
    2. Si comienza a detectar al objetivo en su área este procederá a seguir al objetivo con la intención de acercarse a él.
    3. En cuanto se encuentre a una distancia establecida más cercana se procederá a realizar el árbol *LookAndShoot* con el mismo objetivo.

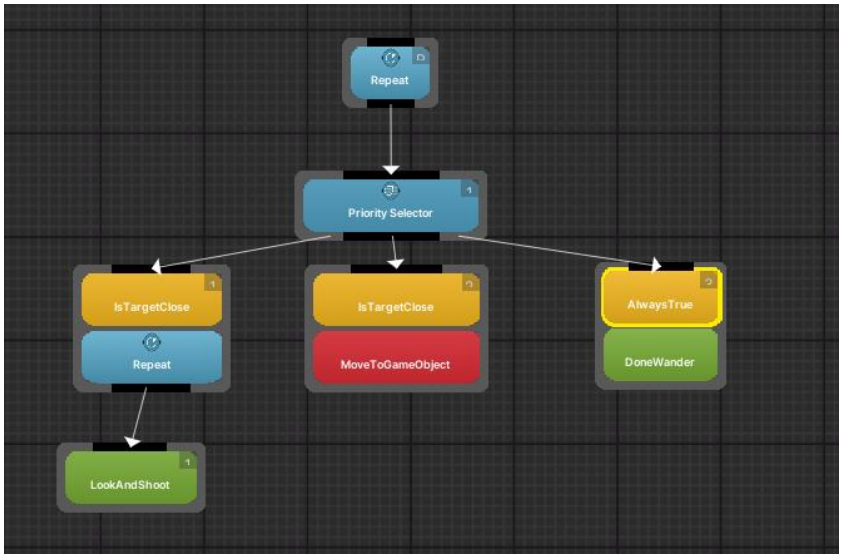


*Ilustración 40: Árbol PistolEnemy*

Este árbol tendrá como subrama al árbol anterior *LookAndShoot*, repitiendo en bucle los pasos mencionados, los parámetros que tendrán que ser introducidos por el programador son: *wanderArea* (el área de vigilancia), el objetivo que se seguirá, un *shootPoint* para instanciar como en *lookAndShoot*, un *gameObject* representando el objeto a instanciar *Bullet*, el *delay* usado en *LookAndShoot* y dos distancias como referencia para la comparación con el objetivo.

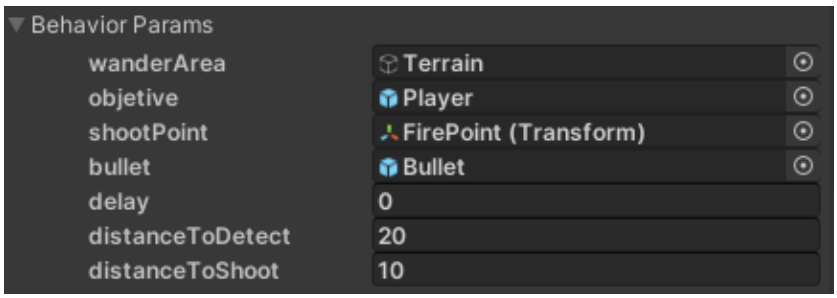


*Ilustración 41: Parámetros PistolEnemy*

- **GuardBehaviour**: En esta situación simularemos el comportamiento de un guardia protegiendo un ente que seleccionará el programador, este seguirá un comportamiento parecido a *PistolEnemy* con la diferencia de estar alrededor de un ente, el bucle que seguirá será el siguiente:
    1. Patrullara de manera aleatoria la zona alrededor del ente
    2. Si este se aleja del ente que tiene que proteger al patrullar, se dirigirá de nuevo alrededor suyo.
    3. Si detecta un enemigo se acercará a el
    4. En cuanto esté en el rango del enemigo este procederá a realizar la acción de *LookAndShoot* con el objetivo detectado.

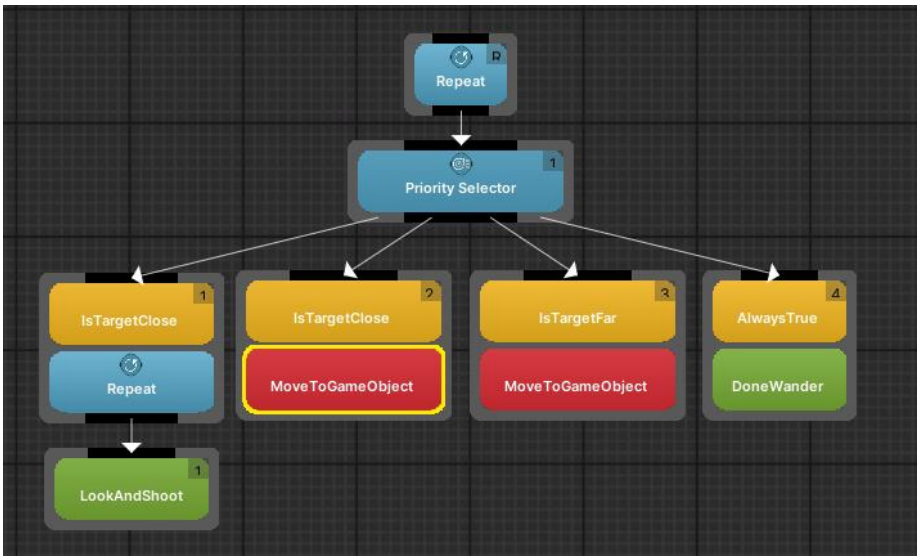


*Ilustración 42: Árbol GuardBehaviour*

Como parámetros de entrada en este árbol podemos definir: *wanderArea* (el área de vigilancia), el objetivo que seguirá y el que detectará como enemigo, tres distancias para definir la acción que seguirá con respecto a los objetivos, un *shootPoint* usado en *LookAndShoot* y un objeto para instanciar.

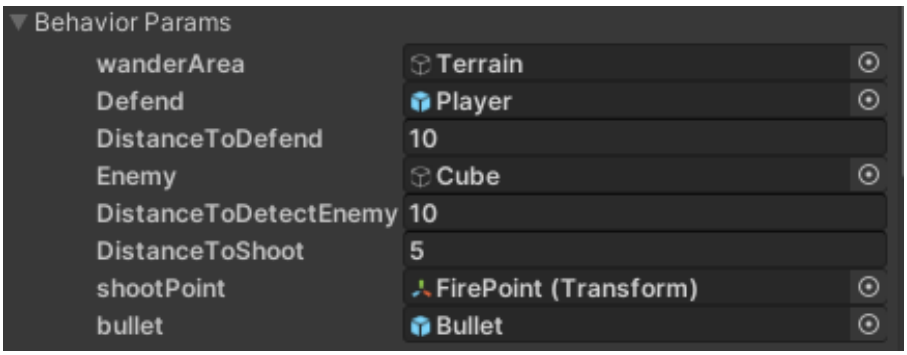


*Ilustración 43: Parámetros GuardBehaviour*

- **AfraidNPC**: Este árbol simulará el comportamiento de un *NPC* que intentará huir del objetivo asignado por el programador, tomando como ruta la inversa donde se situará el objetivo para huir de él, el comportamiento en bucle será el siguiente:
    1. El objetivo se quedará en espera en situación de que no detecte al objetivo
    2. En cuanto detecta al objetivo este procederá a calcular la trayectoria a seguir para huir y la seguirá mientras el objetivo este en su área de detección.

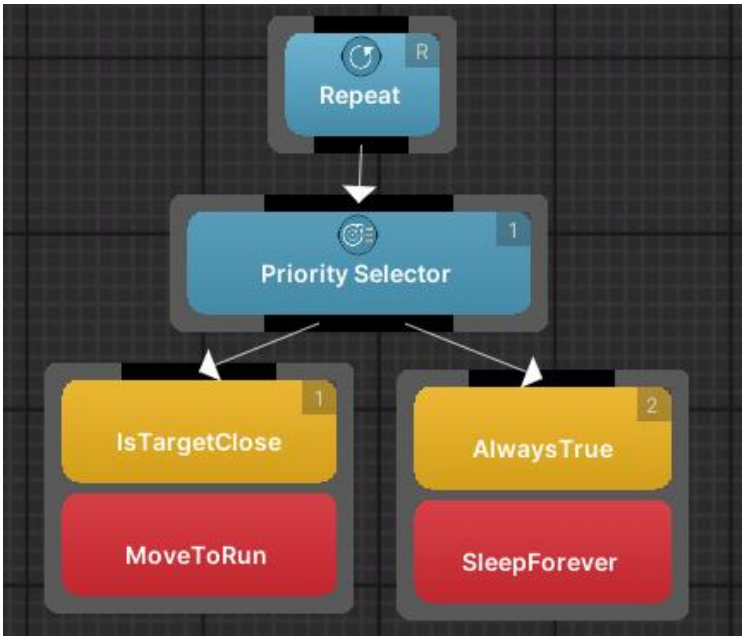


*Ilustración 44: Árbol AfraidNPC*

Este árbol definirá como parámetros los siguientes: *wanderArea* (el área que seguirá para moverse), el objetivo asignado del que huira y la distancia a la que será capaz de detectarlo.

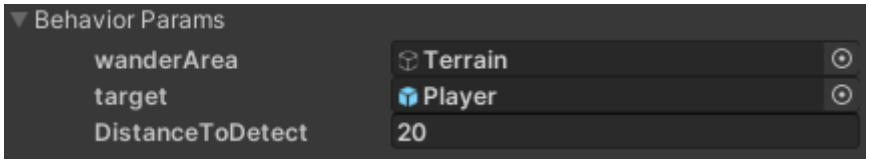


*Ilustración 45: Parámetros AfraidNPC*

# 5. Objetos Prefabs

Dentro del proyecto habrá objetos *prefabs* de ejemplo que mostrarán algunos de los usos en combinación de los *scripts* descritos, esto tiene como objetivo que al programador le sea más fácil ver un uso de los *scripts* y poder crear sus propios objetos en base a los dados.


*Ilustración 46: Icono Character*

Uno de los *prefabs* más completos es *Character*, que contendrá lo siguiente:

- **Player**: Este elemento del *prefab* será aquel que manejará el usuario, dentro de este elemento podemos encontrar todos aquellos *scripts* relacionados a cualquier elemento con el jugador:
    - *PlayerController*
    - *LifePlayer*
    - *PauseMenu*
    - *KeyInventorySystem*
    - *InventorySystem*
    - *CollectItems*

  Los *scripts* mencionados se pueden personalizar con respecto al gusto del programador y su idea de juego, este elemento *player* contiene dos partes una visual que se podrá personalizar a gusto del programador y la otra parte denominada *hand* por si el programador quiere usar un punto de referencia con *inventorySystem*.

- **MainCamera**: Una cámara simple que contendrá el *script* anterior de *CameraControl* y que tendrá como objetivo el cuerpo del carácter que manejará el usuario.
- **Canvas**: Este elemento representara toda la interfaz que controlara el usuario, de base está programada para contener un menú de pausa (pulsando "p"), la vida del *player*, la munición en caso de usar cualquier tipo de *shootSystem* y la puntuación en caso de usar *Score* o *SearchKey*.

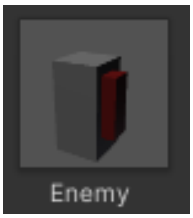


*Ilustración 47: Icono Enemy*

Este *prefab* denominado *Enemy* es aquel que nuestro programador podrá usar para establecerle un árbol de comportamiento y usar como *NPC*, de manera que en sus propios juegos podrá crear todo tipo de personajes según sus gustos, dentro de este *prefab* ha sido incluido un elemento que representa la parte gráfica, una *healthbar* en caso de que se quiera usar como enemigo o eliminar de alguna manera y un *firePoint* que podrá usar el programador en caso de necesitar un punto de referencia.


*Ilustración 48: Icono AmmoBox y Coin*

Estos dos *prefabs* seguirán un comportamiento muy parecido, dado que cada uno contendrá un *script* que interactuará con el *canvas* del *Character*:

- ***Coin***: Contendrá el *script ShockScore* para poder añadir puntuación al score del *character*, de esta manera imitará el comportamiento de una moneda como en los juegos clásicos para que nuestro programador pueda usarla en caso de su necesidad.
- **AmmoBox**: Contendrá el *script RechargeAmmo* para interactuar con la parte *ammo* del *canvas* del *character* y que nuestro programador pueda llevar control de ese elemento del usuario.

Ambos objetos podrán personalizar los valores de sus *scripts* y modificar visualmente para ajustarse mejor a la idea del videojuego que procese el programador.


*Ilustración 49: Icono HealthBar*

*HealthBar* es un *prefab* cuyo uso es darle al objeto al que se adhiera una vida que pueda interactuar con el sistema de daño creado en el proyecto, de manera que el programador podrá establecer que elementos querrá que puedan ser eliminados tanto por el jugador como por elementos externos, para su correcto funcionamiento es recomendable que sea un elemento hijo de la parte del objeto que reciba el impacto.

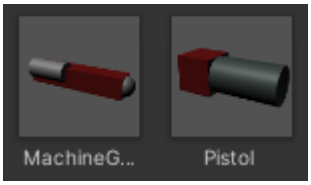


*Ilustración 50: Iconos MachineGun y Pistol*

Estos dos *prefabs* tienen como uso dar un ejemplo al programador de objetos que usen el *shootSystem* (y sus variantes) y *ammoSystem*, estos dos objetos podrán ser equipados por el jugador con el *inventorySystem* o añadidos a través del sistema *Collector* creado en el proyecto, dentro de estos *prefabs* podemos encontrar dos partes una visual (para poder personalizarla a gusto del programador) y un *firePoint* (para poder usar como punto de referencia de *shootSystem*), como *scripts* dentro de estos objetos podemos encontrar tanto el respectivo *shootSystem* como el *ammoSystem*, ambos *scripts* con parámetros base que pueden ser personalizados a gusto del programador.


*Ilustración 51: Icono Bullet*

Este *prefab* será aquel que podrá ser usado como objeto para los *shootSystem* o para instanciar y dañar al *player*, podemos encontrar el *script bullet* que definirá su comportamiento, tiempo de vida y parámetros personalizables para que el programador lo establezca a su gusto.

# 6. Demos Expuestos

Anexionado al trabajo el programador como ayuda adicional tendrá a su disposición ejemplos de minijuegos realizados con el material dado, estos minijuegos representaran situaciones tradicionales que podrían ocurrir en distintos videojuegos y podrán servirle de ejemplo de uso de los distintos *scripts* y elementos dados en el trabajo, observando cómo podrían trabajar de manera conjunta, cada uno de estos minijuegos están pensados en utilizar los distintos *GameModes* realizados durante el trabajo y con situaciones de uso de los mismos.

**Minijuego Supervivencia:**

Este minijuego está basado en los clásicos videojuegos de supervivencia en que el objetivo prioritario de estos videojuegos es conseguir la mayor cantidad posible de puntos, por ello el *gamemode* usado en esta situación será el *Score*.

Este minijuego contara con los siguientes elementos:

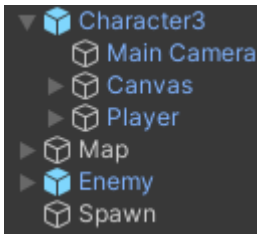


*Ilustración 52: Elementos minijuego Supervivencia*

- El propio *prefab* de *Character* que tendrá nuestro jugador y que habrá sido modificado para la escena pensada en el minijuego, la cámara estará fijada en un punto a una altura determinada del propio *player* de manera que simulara una vista cenital, el *canvas* no ha sufrido apenas modificaciones y el *player* tendrá todos sus *scripts* activos dado que se hará uso de gran parte de sus elementos.
- Un elemento denominado *Map* que determinara el escenario del propio juego (incluyendo los propios muros) y ayudara a realizar un mapeado con *Navigation* para que las *IAs* sean capaces de moverse.
- El *prefab Enemy* con el objetivo del jugador y el árbol de comportamiento de *PistolEnemy* de manera que seguirá al jugador y procederá a dispararle al alcanzar una distancia, las modificaciones empleadas han sido la de añadir dos elementos el *EliminateScore* y el *DeathDrop* con *AmmoBox* como elemento a aparecer, estos dos *scripts* han sido usados para que cuando sea eliminado *enemy* el player obtenga puntos y pueda recargar.
- Un elemento vacío que tendrá el *script* de *Spawn* con el mismo *enemy* de manera que ira generando *enemy* de manera constante

El bucle jugable de este minijuego se basará en el control del *player* y que se vayan generando en la zona distintas copias de *enemy* por lo que le seguirán y dispararan, si el *player* los elimina obtendrá puntos y podrá mantenerse al recargar *ammo*.

**Minijuego Búsqueda:**

Este minijuego plantea el realizar una búsqueda de esferas para completar un nivel básico, todas las esferas representan objetos con *keys* numeradas y relacionadas al *gamemode SearchKey* con el objetivo de buscar las *keys* de manera que cuando se complete finalizara el nivel, como añadido al nivel se tendrán distintas plataformas donde se repartirán las esferas para representar mayor dinamismo.

En este minijuego podremos ver la siguiente composición de elementos:


*Ilustración 53: Elementos Minijuego Búsqueda*

- El *prefab* del *Character*, en esta situación las modificaciones empleadas habrán sido la de desactivar en el *canvas* la parte respectiva a la vida y el *ammo*, y en la parte del *player* desactivar los *scripts* en la vida y *inventorySystem* dado que no se hará uso de estas partes en el minijuego.
- La propia disposición del mapa donde se contendrán todas las plataformas de las que nuestro *player* podrá saltar o caer, el suelo donde se producirá el minijuego y las paredes que delimiten el área donde el jugador estará.
- Y el conjunto de objetos *Keys* que determinarán todas las *Keys* que nuestro player tendrá que recoger, todos los objetos dentro de esta agrupación serán esferas con el *script CollectorKey*, haciendo referencia a las llaves que se buscan en el *Gamemode*.

La jugabilidad en este minijuego nuestro *player* tendrá el objetivo de recolectar todas las esferas amarillas que serán asociadas a distintas *keys* para que el minijuego se termine, todo esto mientras el *player* se moverá por distintas plataformas para poder recolectarlas.

# 7. Conclusión

Este proyecto ha sido pensado para ayudar a los futuros programadores a iniciarse en el mundo de la programación para poder exponer sus ideas en un medio en expansión. Ofreciéndole diversas ayudas para que estos puedan plasmar sus ideas al medio de los videojuegos.

Las líneas futuras a destacar con este proyecto, mediante el uso de Unity y el lenguaje C#, puede llegar a expandirse de multitud de maneras, dado que en el proyecto actualmente han sido programados conceptos básicos que se pueden encontrar en multitud de videojuegos, podemos llegar a aplicar conceptos básicos nuevos con los que crear nuevos estilos de juegos y poder abarcar más ámbitos dentro del mundo de los videojuegos, al llegar a añadir nuevos conceptos con los básicos ya implementados podemos realizar multitud de combinaciones nuevas de *scripts* con los anteriores, por lo que el crecimiento de este proyecto seria de manera exponencial y podremos abarcar gustos más específicos para los jóvenes programadores, otra de las ideas que podríamos llegar a experimentar en un futuro avance del proyecto seria adecuar las mecánicas y los distintos sistemas implementados en un sistema multijugador, permitiendo que el programador cree videojuegos online, haciendo uso de una versión nueva de la librería adecuada a videojuegos multijugador.