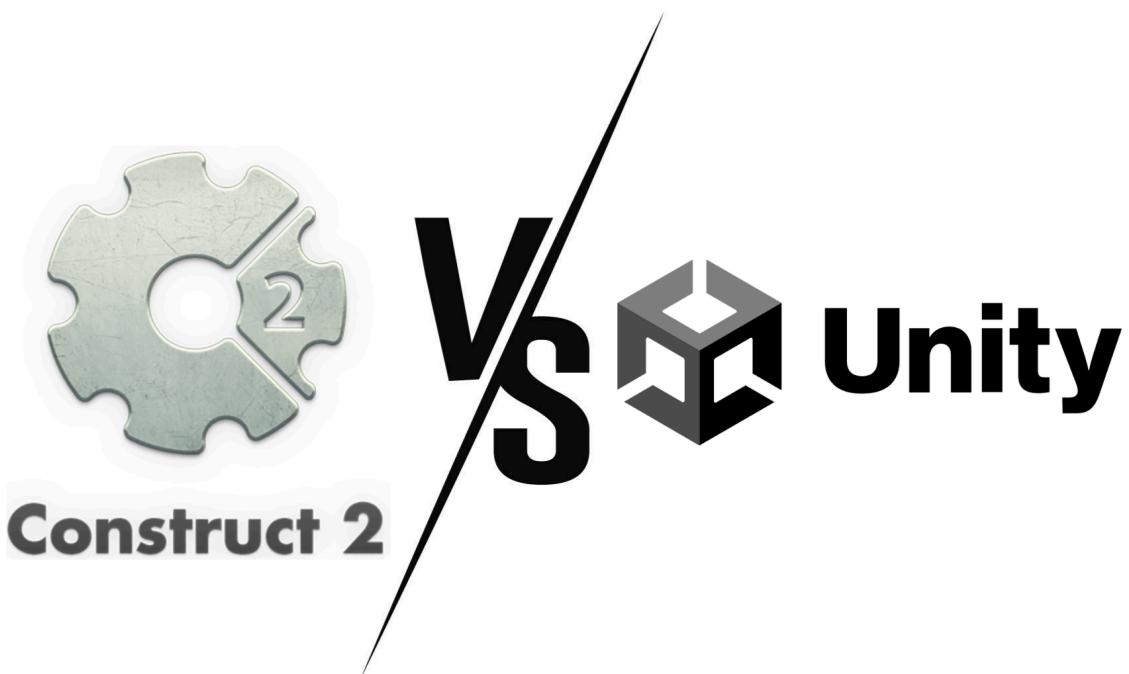


Comparativa Unity vs Construct 2



Centro escolar : I.E.S Ramón del Valle Inclán

Alumno : Abraham Carrasco Molina

Tutora : Eva Palomo Cárcamo

Curso : 2ºDam

Fecha : 21/06/2024



Índice

Introducción.....	4
Importancia del proyecto.....	4
Objetivos del Proyecto.....	4
Justificación de la elección de Construct 2 y Unity.....	5
Planteamiento del Proyecto.....	6
Construct 2.....	7
Sistemas de eventos y comportamientos.....	7
Selección de instancia de objeto.....	7
JavaScript.....	7
Plataformas compatibles.....	8
¿Por qué para este proyecto se eligió la versión de Construct 2?.....	8
Juego en Construct.....	9
Entorno de desarrollo.....	9
Menús laterales (con el ejemplo del juego).....	12
Juego (Parte visual y funcional).....	15
Escena 0.....	15
Escena 1.....	16
Escena 2.....	17
Escena 3.....	18
Escena 4.....	19
Hoja de Eventos Menu.....	20
Hoja de Eventos 1.....	29
System.....	30
ControlarPlayer.....	32
IA CPU.....	33
IA CPU2.....	34
Disparar.....	36
Fin.....	38
Cámara.....	39
Audio.....	40
Unity.....	41
Usabilidad.....	41
Motor Gráfico.....	41
Sombreadores y shaders.....	41
Scripting.....	42
Control de versiones.....	42



Mecanim.....	43
Juego en Unity.....	44
Glosario.....	59



Introducción

En este proyecto de Grado Superior, se presenta una comparativa entre dos plataformas de desarrollo de videojuegos: Construct 2 y Unity.

La idea surge de la motivación e interés del autor en el sector de los videojuegos, y su deseo de crear y compartir sus propias creaciones en un campo en constante evolución.

Importancia del proyecto

El desarrollo de videojuegos se ha convertido en una de las industrias en crecimiento con un impacto significativo tanto económicamente como en la cultura moderna. La elección de la plataforma de desarrollo a la hora de crear un videojuego es una decisión importante, que depende de los requisitos a cumplir, ya que puede influir en el éxito y la eficiencia del proceso de creación de un juego. Debido a esto, es fundamental realizar este tipo de comparaciones exhaustivas entre diferentes herramientas para permitir tomar mejores decisiones y opiniones sobre para qué proyectos puede ser más rentable en término general utilizar una plataforma u otra.

Objetivos del Proyecto

Los objetivos principales de este proyecto son los siguientes :

1. Analizar y documentar las características principales de Construct 2 y Unity.
2. Desarrollar un juego semejante utilizando cada una de estas plataformas para comprender sus diferencias y similitudes.
3. Realizar una comparativa detallada entre Construct 2 y Unity en términos de facilidad de uso, rendimiento, flexibilidad etc.
4. Proporcionar recomendaciones y conclusiones basadas en los resultados obtenidos para ayudar a nuevos desarrolladores a seleccionar la plataforma más adecuada según sus necesidades y requisitos específicos.



Justificación de la elección de Construct 2 y Unity

La elección de Construct 2 y Unity como plataformas para este proyecto se basa en varias consideraciones. Construct 2 es conocido por su facilidad de uso y su capacidad para crear juegos 2D de manera sencilla y rápida, con lo que se vuelve una opción atractiva e importante para desarrolladores principiantes o con proyectos con poco tiempo para poder ser realizados. El alumno que realiza este proyecto visualizó un curso de Construct 2 y ahí fue donde lo descubrió y decidió investigar más al respecto y desarrollar un videojuego con esa tecnología. Por otro lado, Unity es una de las herramientas más populares y versátiles en la industria del desarrollo de videojuegos, ofreciendo potentes capacidades para crear juegos en 2D y 3D, así como soporte multiplataforma y una gran comunidad de usuarios y recursos disponibles. A continuación en los siguientes apartados se especificará más al respecto sobre las distintas plataformas y el desarrollo del videojuego en cada una de ellas.



Planteamiento Personal del Proyecto

Primero se realizó una investigación sobre las dos plataformas (Construct 2 y Unity). La plataforma de Construct 2 parecía bastante más sencilla , por lo que se decidió realizar esa parte del proyecto en Navidad y junto con los meses de prácticas la otra parte correspondiente a Unity.

Flujo de trabajo personal

Primero pensé sobre qué estilo podía realizar el videojuego, intenté ver variedad y obtener inspiración. Descubrí una imagen de un personaje parecido a un mago en píxeles y decidí crear mis propios diseños de magos para mi juego, esto lo realice en la página [Piskel](#). Esta página es sencilla y permite hacer buenos diseños de píxeles en 2D de una forma simple. Una vez completado el diseño de los magos, había que pensar que posibles enemigos podrían tener, cuyo pensamiento me llevó a hacer un zombie como enemigo (en la misma página de Piskel). Una vez tenía definidos los personajes principales, había que pensar en el mapa y decoración de este mismo. Para ello utilice varias páginas [Itch.io](#) y [Kenney](#) . En ellas se pueden encontrar assets de todo tipo, ya sean visuales en 2D y 3D, de audio, gratis o de pago etc. Son dos páginas que recomiendo bastante, son fiables y con un gran contenido.

Una vez tenía todo elegido y creado, pensé en un tipo de jugabilidad que me gustará, que fuera en 2D, ya que el diseño en 3D requiere de mayor tiempo y conocimientos (he probado algún programa para ello y son bastante más complicados.) Y una vez se tiene ya en mente todo lo anterior y gran parte de ello desarrollado, se puede comenzar a realizar el videojuego, comenzando por la movilidad del personaje principal, luego sus posibles acciones como disparar etc. Posteriormente se realiza un poco del mapa para poder probar bien la movilidad del personaje. Una vez todo funcione correctamente, se puede comenzar a añadir enemigos, con sus respectivas acciones (movilidad, etc). Se comprueban ciertos posibles errores entre el contacto del personaje principal y los enemigos, el daño que se inflige, cuando se elimina uno de los objetos ... Una vez se comienza con todo este proceso es normal que a cada momento vayan surgiendo nuevas ideas para mejorar el proyecto, ya sean visuales, efectos, nuevas mecánicas... El resto del proceso e ideas van surgiendo automáticamente, y en caso de tener un poco de bloqueo mental, siempre se puede buscar inspiración de otros juegos o bien en las páginas mencionadas anteriormente. No olvidar el desarrollo de la UI (Interfaz de usuario), en ella se suelen colocar las vidas de personajes, menús y algunas opciones más o detalles que se quieran mostrar, como podría ser una puntuación. Decidí ir realizando la UI con respecto a cómo avanzaba el desarrollo de los diferentes proyectos (Construct y Unity) ya que me iban surgiendo nuevas ideas cada día y siempre tenía que cambiar o añadir novedades.



Construct 2

Construct es un motor de videojuegos basado en HTML5 desarrollado por [Scirra Ltd](#). Principalmente está dirigido a no programadores, es decir, personas sin experiencia o con poca experiencia, permitiendo la creación rápida de juegos a través de programación visual.

Características principales

Sistemas de eventos y comportamientos

El método principal para programar juegos y aplicaciones en Construct es a través de “hojas de eventos”. Cada hoja de eventos tiene una lista de eventos que contienen declaraciones condicionales o desencadenantes. Posteriormente cuando se detalle el videojuego desarrollado se mostrarán ejemplos sobre ello. Una vez cumplidos dichos condicionales se pueden realizar acciones o funciones.

La lógica de eventos como OR y AND así como los subeventos, permiten programar sistemas sofisticados sin aprender un lenguaje de programación que comparablemente sería más difícil. Permite la creación de grupos que se pueden utilizar para habilitar y deshabilitar múltiples eventos a la vez y para organizarlos mejor.

Selección de instancia de objeto

Construct evita seleccionar [instancias](#) específicas de objetos al agregar [eventos](#), en favor de filtrar todas las instancias de un tipo de objeto en la pantalla. Al agregar eventos, el editor permite al usuario especificar condiciones o comprobaciones que debe cumplir cada instancia de objeto en la pantalla antes de que agregue o ejecute el evento. Dichos eventos se pueden encadenar mediante subeventos, lo que permite crear comportamientos más complicados.

JavaScript

La versión más moderna de Construct (Construct 3), admite JavaScript como lenguaje de secuencias de comandos opcional, para satisfacer las necesidades de los usuarios avanzados y la popularidad de las soluciones existentes.





Plataformas compatibles

Las principales plataformas de exportación de Construct 2 están basadas en HTML5. Afirma ser compatible con Google Chrome, Firefox, Internet Explorer 9+, Safari 6+, y Opera 15+ en navegadores de escritorio, y soporte para Safari en IOS 6+, Chrome y Firefox para Android, Windows Phone 8+, BlackBerry 10+ y [Tizen](#).

Además puede exportar a varios mercados y plataformas en línea, incluidos Facebook, Chrome Web Store, Firefox Marketplace, Amazon Appstore, Construct Arcade (su propia plataforma para alojar juegos), [Kongregate](#) y algunas más.

También posee la capacidad de exportar a varias plataformas que brindan un comportamiento de aplicación nativa y sin conexión: Windows, MacOS y Linux de 32 y 64 bits, compatibles con la exportación de [NW.js](#).

Además maneja el soporte móvil nativo para IOS y Android mediante [Cordova](#).

En el año 2015 se lanzó un complemento para hacer que los juegos basados en Construct sean compatibles con Nintendo Web Framework.

El 13 de abril de 2016, Scirra anunció que la compatibilidad con [UWP](#) de Construct 2 permitirá publicar juegos en Xbox One.

¿Por qué para este proyecto se eligió la versión de Construct 2?

La razón principal por la elección, es debido a que la última versión solo permite realizar 50 eventos de forma gratuita, en cambio, la versión 2 permite hasta 100 eventos (sin contar subeventos). Esto permitía para la realización del proyecto un videojuego mejor desarrollado y con más posibilidades para realizarlo y añadir diferentes tipos de eventos y funciones.

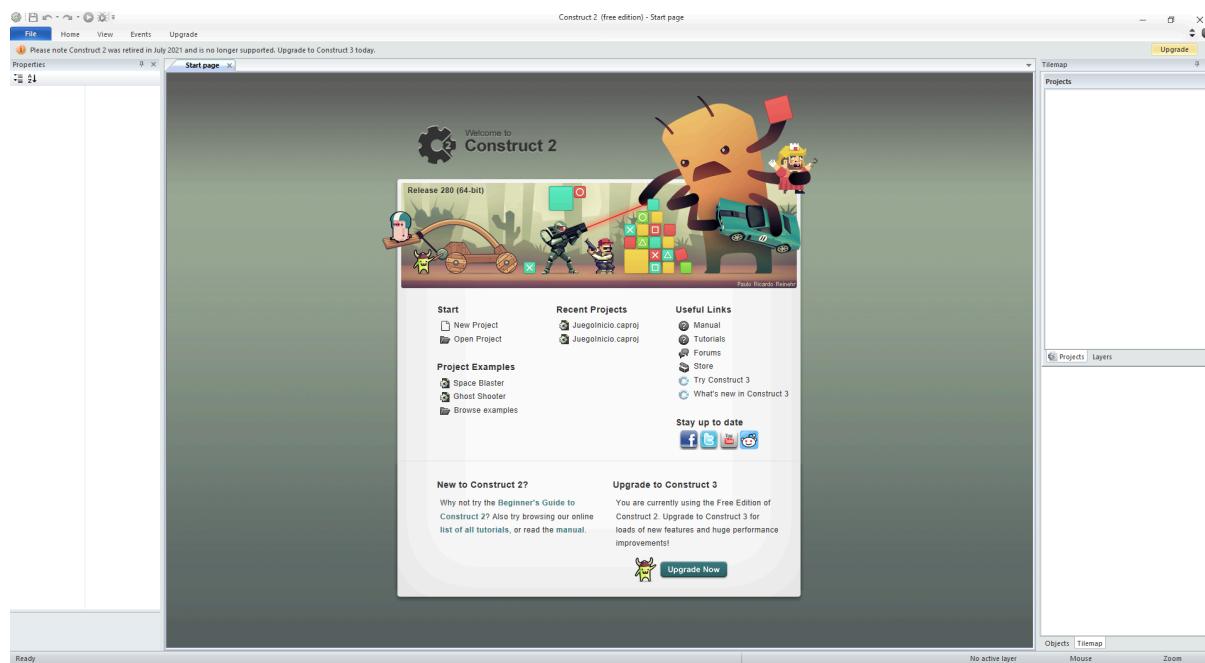
Por lo tanto era la opción más viable para realizar una buena comparación en cuanto a desarrollo del videojuego con respecto a otra plataforma.



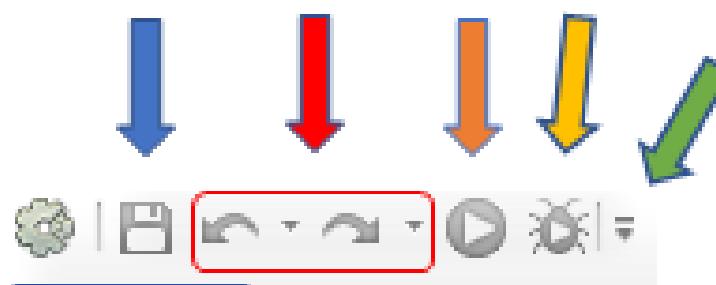
Juego en Construct

Primero se va a mostrar el entorno de desarrollo de Construct 2, posteriormente se mostrarán las escenas del videojuego y más tarde se desarrollará la parte funcional de este mismo. Esa será la estructura.

Entorno de desarrollo



La imagen anterior muestra el inicio del entorno, con una ventana central donde se muestran los proyectos anteriores, además de algunos proyectos de ejemplo y links interesantes. Los menús laterales se explicarán más detalladamente cuando se explique más información sobre el juego en concreto que se ha realizado para este proyecto, a continuación se van a mostrar las opciones que ofrecen los menús superiores.



La flecha **azul** indica el ícono de guardar el proyecto.

La flecha **roja** indica los botones típicos de deshacer yrehacer.

La flecha **naranja** representa el botón de play para el layout actualmente seleccionado

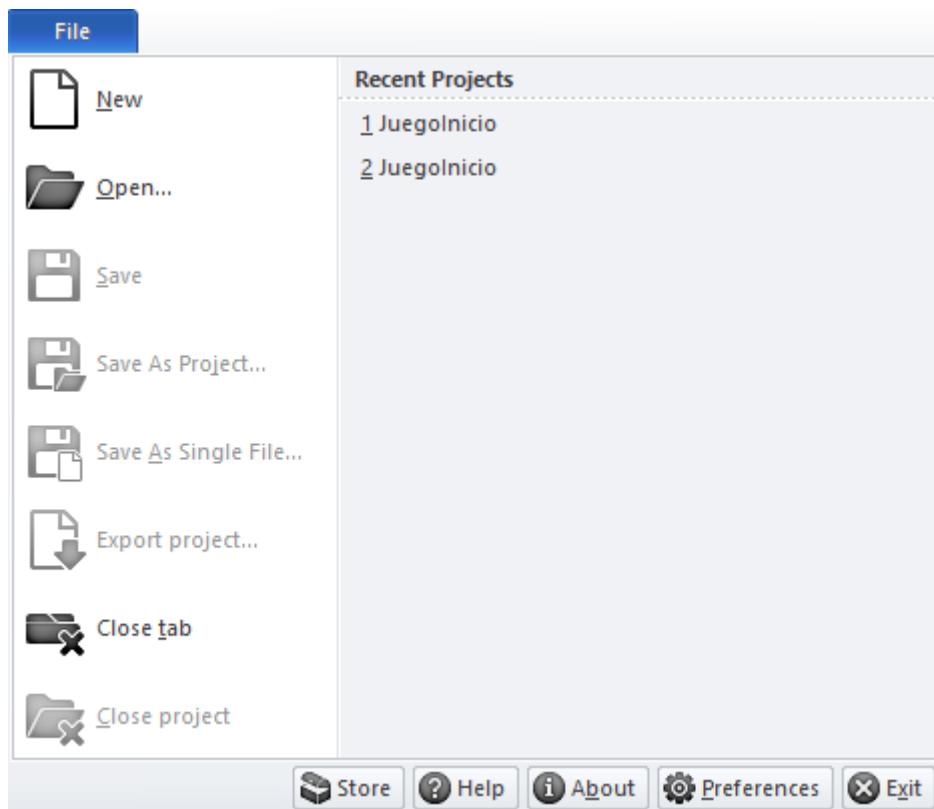
La flecha **amarilla** igual que el anterior pero en este caso en modo debug.



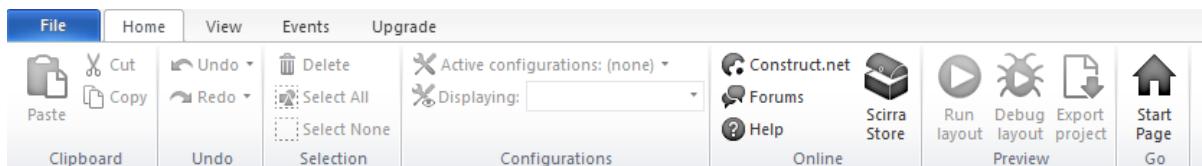
La flecha verde es un acceso rápido por si alguien quiere modificar este menú.



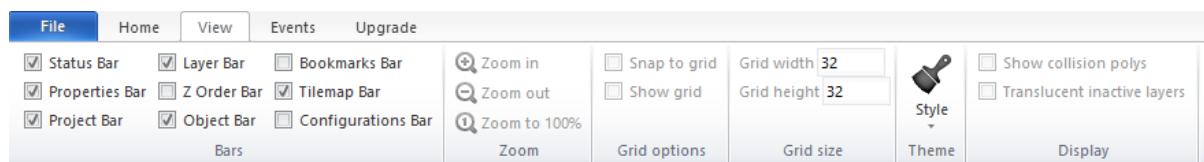
Este menú con las opciones de **File**, **Home**, **View**, **Events**, **Upgrade**, se sitúa justo debajo del explicado anteriormente.



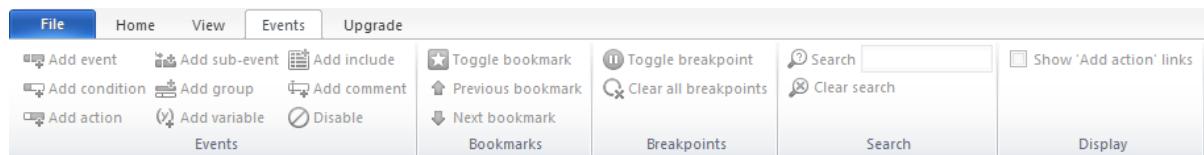
La opción de **File**, nos permite crear un nuevo archivo/proyecto, abrir uno existente, o bien los recientes que aparecen en la parte derecha de la imagen. También permite guardar el archivo, bien como proyecto o único archivo, exportar el proyecto y cerrarlo , además de algunos botones de ayuda en la parte inferior derecha.



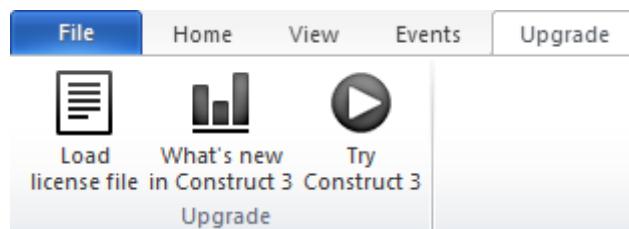
La opción de **Home**, permite las opciones típicas de este tipo de menús en cualquier programa, añadir como diferencia el botón de Scirra Store donde se pueden comprar assets para utilizar en Construct. Redirige al siguiente link : [Scirra Store](#).



La opción de **View**, muestra checkbox para ocultar o mostrar diferentes barras/menús, además permite hacer o quitar zoom, mostrar grid, es decir mostrar una cuadrícula para que sea más fácil colocar los assets correctamente y un par de opciones más.



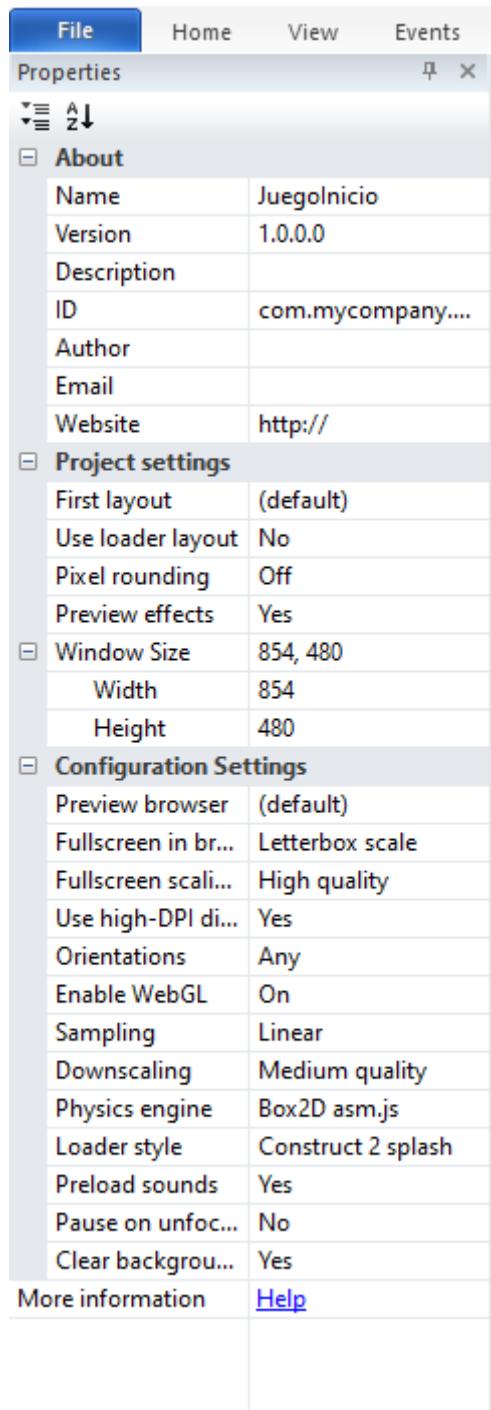
La opción de **Events**, permite agregar todo lo relacionado a funcionalidad desde el mismo menú. También se puede realizar esto desde hojas de eventos, posteriormente con la explicación detallada del juego se mostrará.



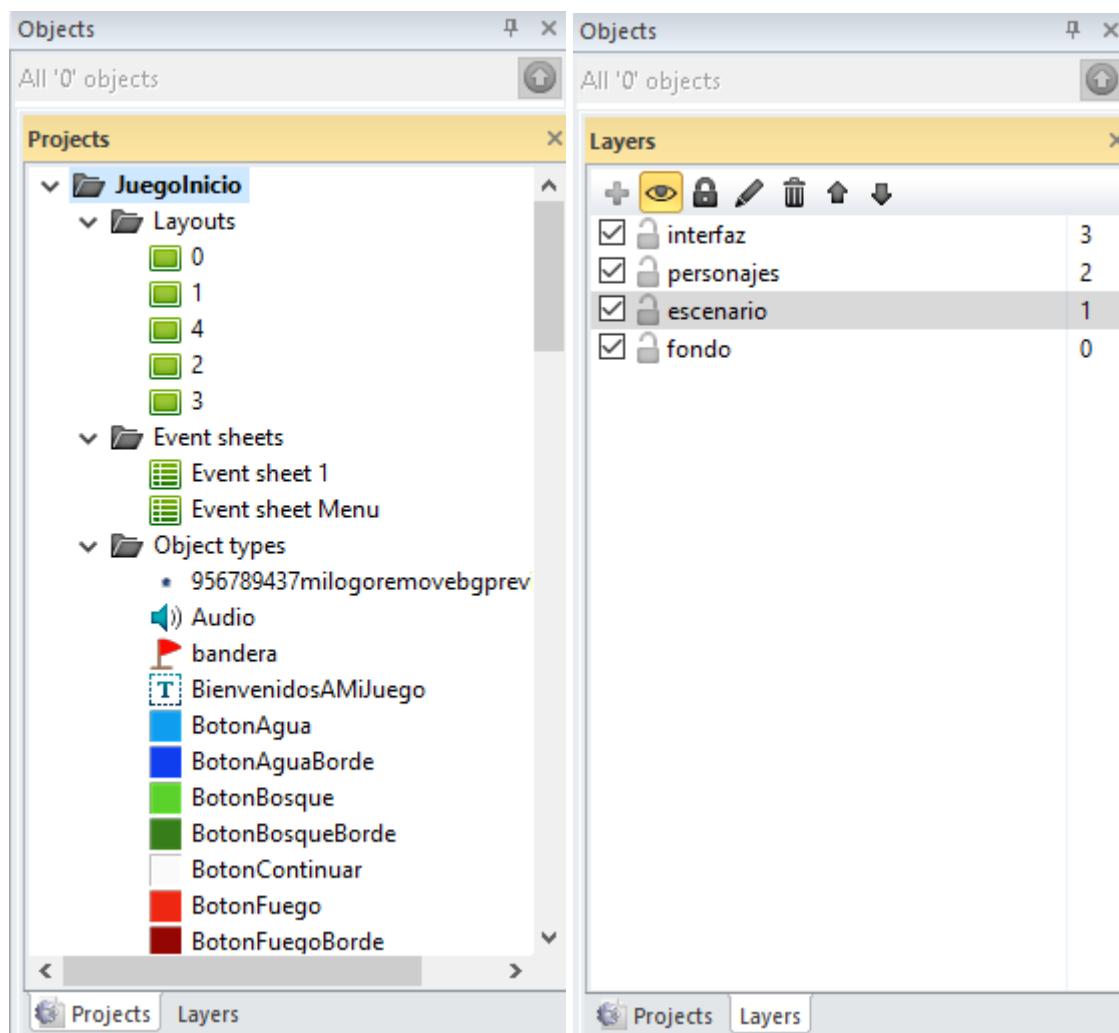
La opción de **Upgrade**, se encuentra para intentar que el usuario que aún utilice la versión de Construct 2, actualice a la versión Construct 3.



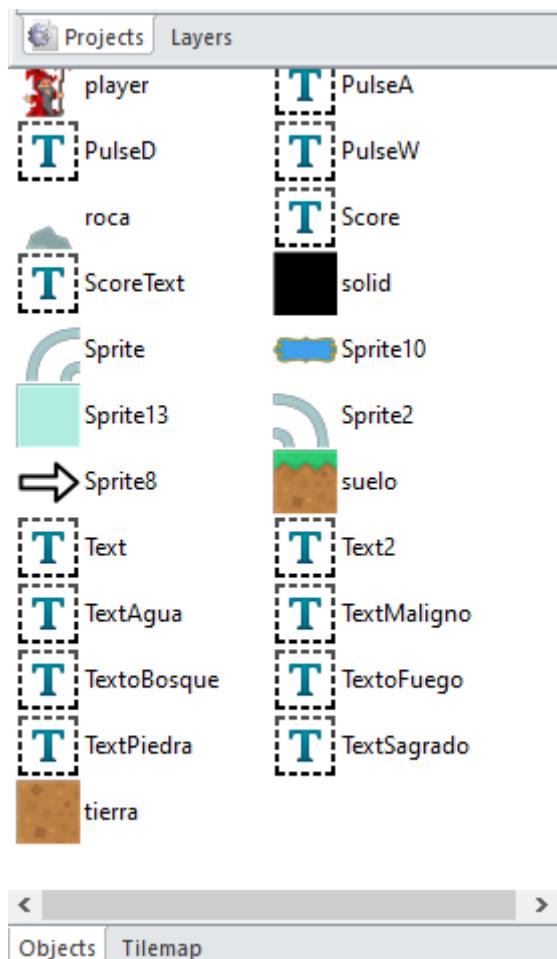
Menús laterales (con el ejemplo del juego)



Justo debajo del anterior menú explicado, se encuentra el apartado de propiedades, al no tener ningún objeto seleccionado aparecen las propiedades del proyecto.



En la parte superior derecha se encuentra el menú del proyecto con su organización y los layers existentes. Desde aquí se pueden seleccionar objetos, audios o bien visualizar cierto layer en concreto de una forma fácil y rápida.



Debajo de este menú se sitúa otro con los Objects y Tilemap si es que los hay.



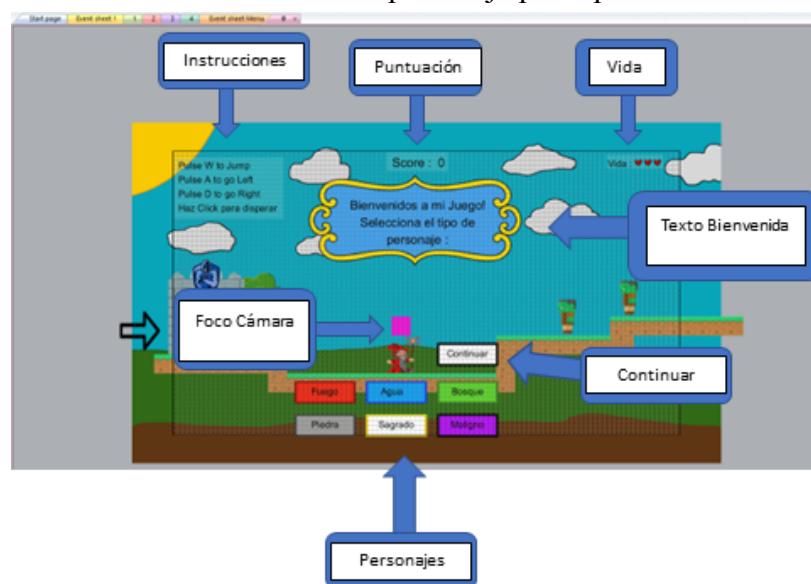
Juego (Parte visual y funcional)

Primero se van a mostrar las diferentes escenas y posteriormente las hojas de eventos con sus respectivas explicaciones.

Escena 0

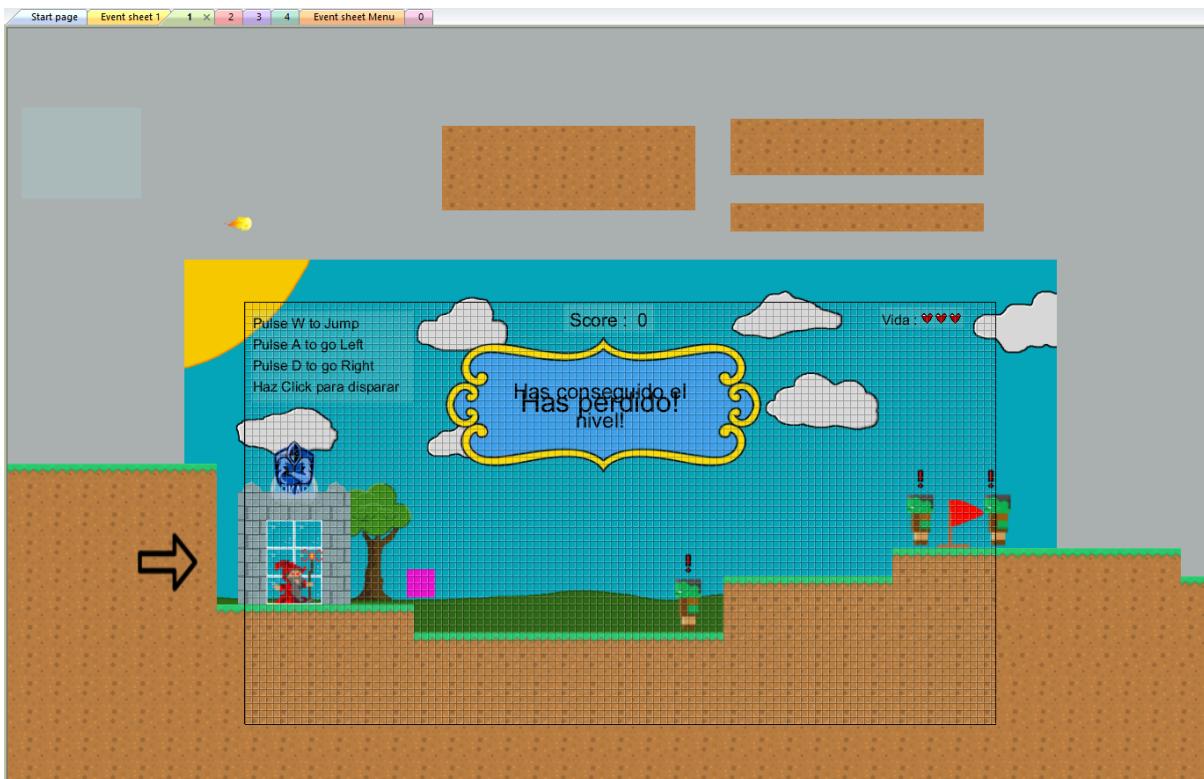


Está es la escena inicial del videojuego, en ella se puede visualizar el fondo, las instrucciones en la parte superior izquierda para jugar, un texto en la zona central de bienvenida, en la parte superior derecha se puede ver la puntuación del usuario. También en la parte superior derecha se puede ver las vidas del jugador. En la zona baja se visualizan 6 botones, estos cambiarán el personaje principal y una vez seleccionado se pulsará el botón de continuar para avanzar a la siguiente escena. La funcionalidad será explicada posteriormente. El foco de la cámara es el cuadrado violeta de encima del personaje principal.





Escena 1

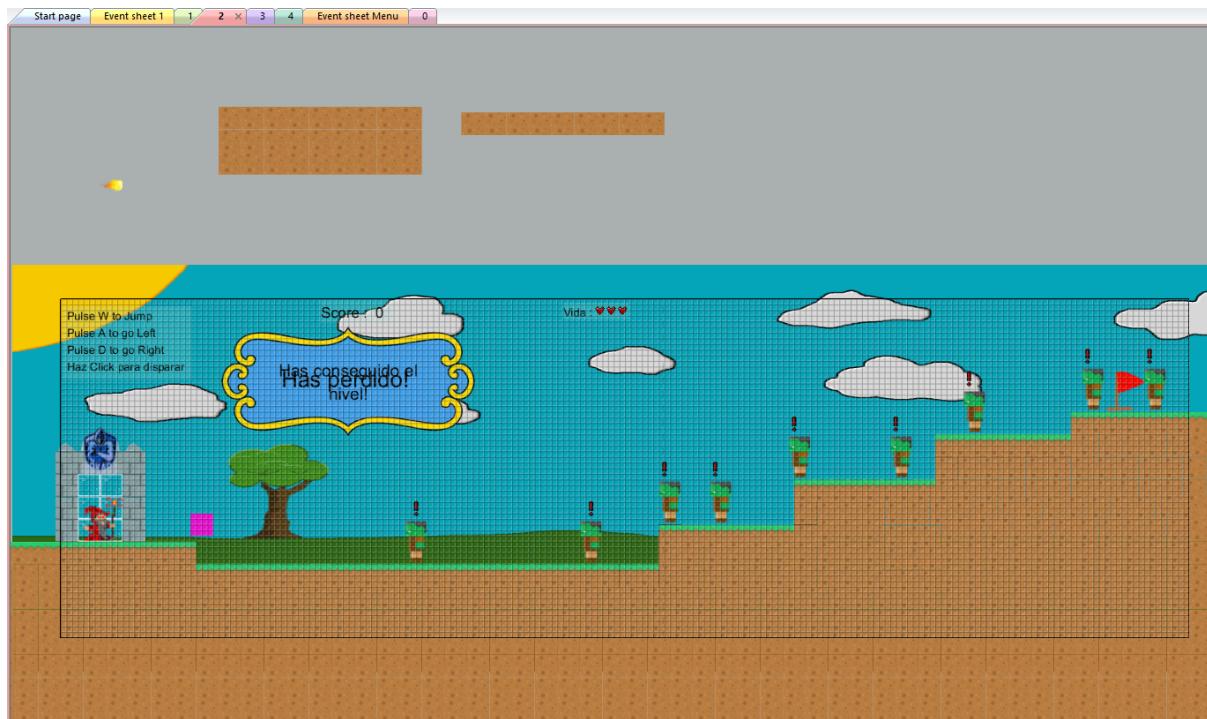


Corresponde al primer nivel, en ella se pueden visualizar opciones diferentes con respecto a la anterior escena, como dos textos, dependiendo de si el nivel es superado o no, los signos de exclamación encima de los enemigos (luego se explica cuando aparecen en la funcionalidad) y el objeto bandera que también se detalla posteriormente.





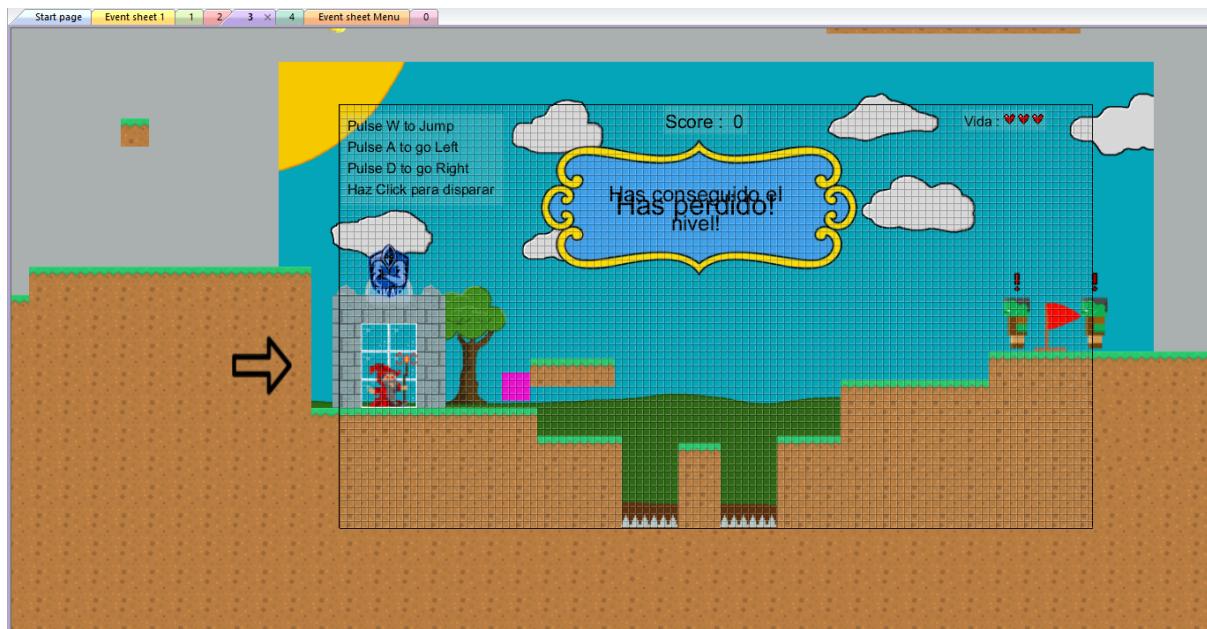
Escena 2



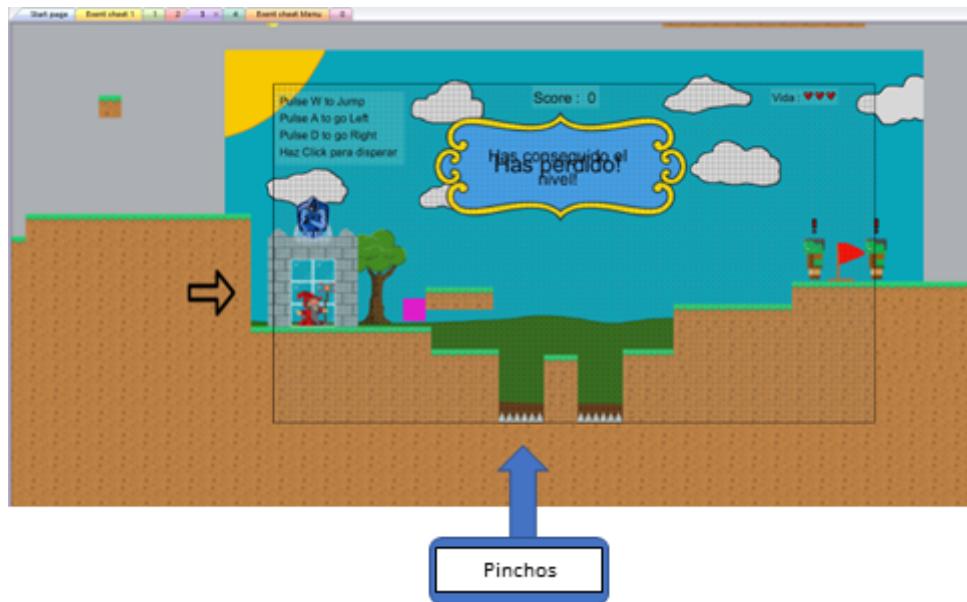
Mismos objetos, fondo etc que la escena anterior, solo cambia un poco el mapa y el terreno del suelo con una forma distinta y el añadido de más enemigos.



Escena 3

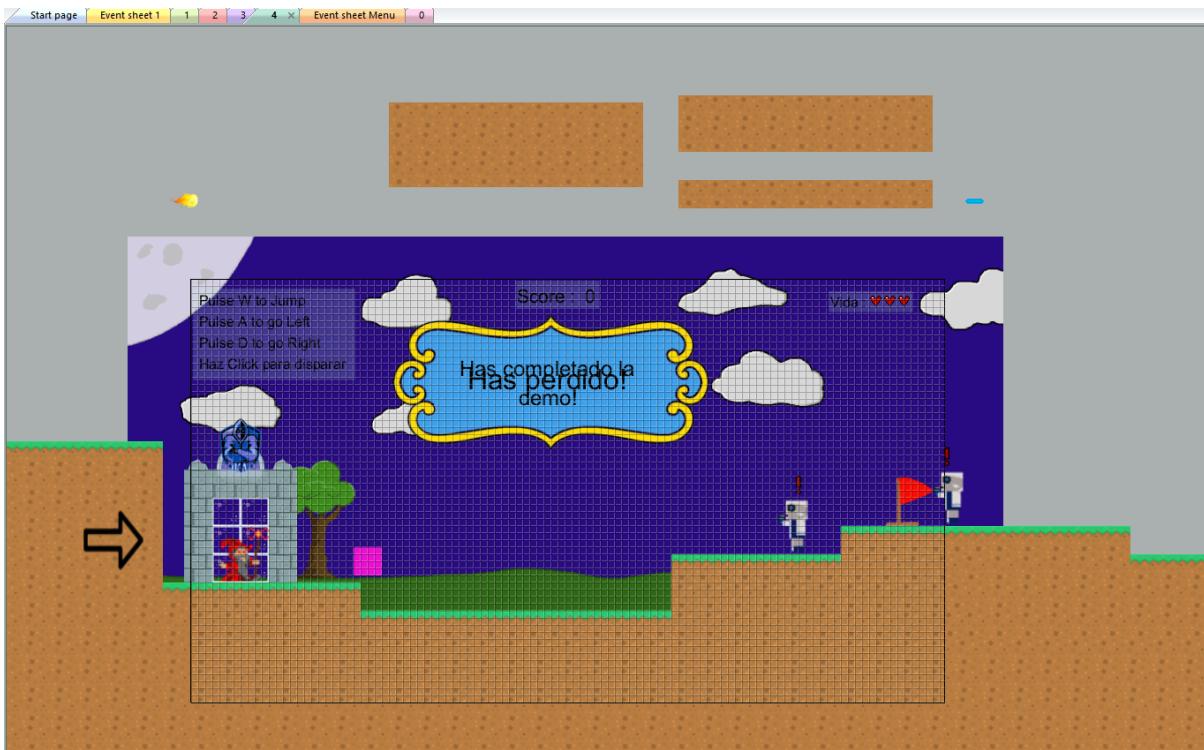


De la misma forma que la escena anterior cambia un poco el recorrido del nivel, y este si tiene un objeto nuevo que son los pinchos que se visualizan en la zona inferior. Su función será explicada en las hojas de eventos.





Escena 4



Esta es la escena final, en ella cambia el fondo, los enemigos y el texto de victoria, que en este caso será “Has completado la demo!”



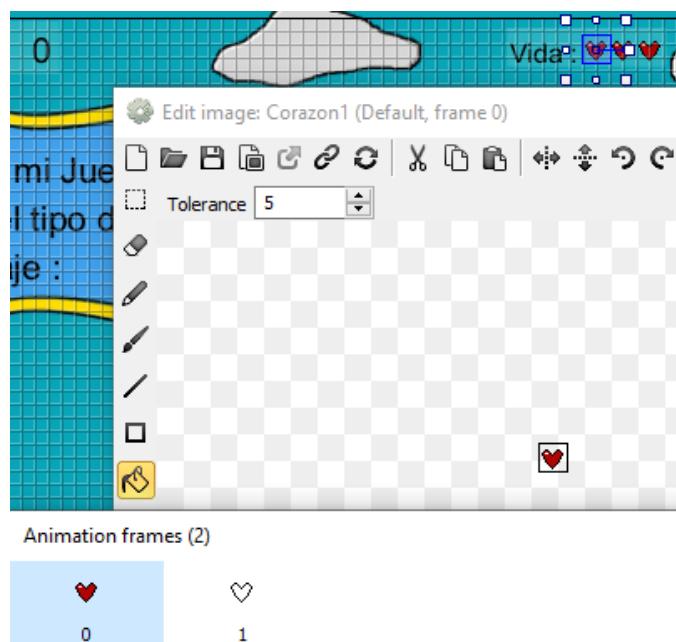


Hoja de Eventos Menu

En esta hoja de eventos que sirve para la escena 0, se agrupan todos los eventos en un grupo llamado menú, y también se crea la variable global personaje.

Según el botón del personaje que se haga clic, esta variable tendrá x valor, a continuación se muestran todas las casuísticas.

Para comenzar se paran las animaciones de los objetos corazón (hacen referencia a la vida del jugador), esto se debe a que dichos objetos tienen dos frames, uno para cuando ese punto de vida existe, y otro para cuando te lo quita un enemigo. En la imagen inferior se muestran los dos frames con uno de los objeto corazón como ejemplo. Al parar la animación, el frame por defecto es el 0.



Ahora se van a mostrar las funcionalidades de los botones según el personaje que se seleccione, se explicaran debajo de la imagen correspondiente dichas funcionalidades para que se entienda de una forma sencilla y clara:

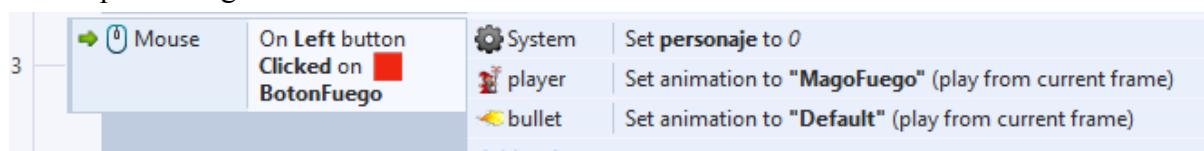




Recordemos los botones para seleccionar el personaje de la [Escena 0](#), en el caso de hacer clic en el botón cuyo nombre aparece como “Fuego”



Se cumplira lo siguiente:



La variable global personaje obtendrá el valor 0, la animación del player tendrá el valor “MagoFuego”, y la animación correspondiente a la bala(bullet) obtendrá el valor “Default”, estos valores corresponde a la siguiente visualización:



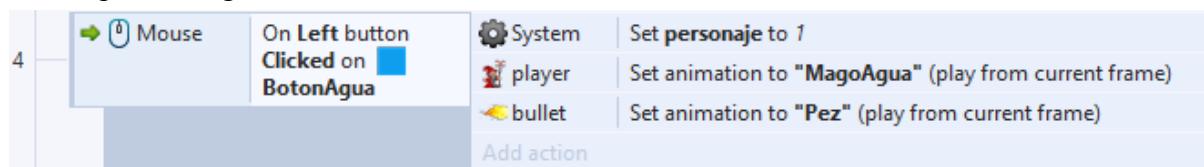
Con el objeto player seleccionado se pueden ver sus diferentes animaciones y con el objeto bullet igual



En el caso de hacer clic en el botón cuyo nombre aparece como “Agua”



Se cumplira lo siguiente:



La variable global personaje obtendrá el valor 1, la animación del player tendrá el valor “MagoAgua”, y la animación correspondiente a la bala(bullet) obtendrá el valor “Pez”, estos valores corresponde a la siguiente visualización:

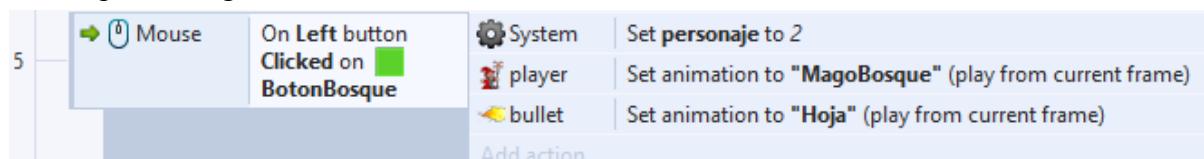




En el caso de hacer clic en el botón cuyo nombre aparece como “Bosque”



Se cumplira lo siguiente:



La variable global personaje obtendrá el valor 2, la animación del player tendrá el valor “MagoBosque”, y la animación correspondiente a la bala(bullet) obtendrá el valor “Hoja”, estos valores corresponde a la siguiente visualización:

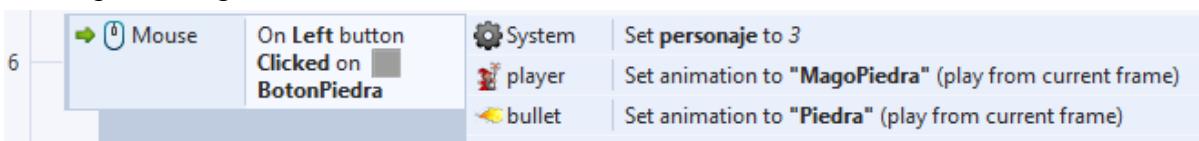




En el caso de hacer clic en el botón cuyo nombre aparece como “Piedra”



Se cumplira lo siguiente:

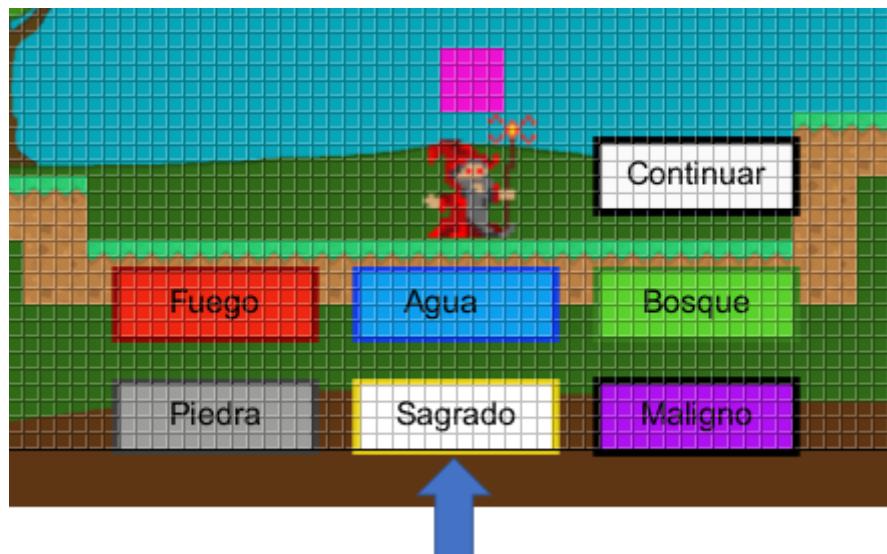


La variable global personaje obtendrá el valor 3, la animación del player tendrá el valor “MagoPiedra”, y la animación correspondiente a la bala(bullet) obtendrá el valor “Piedra”, estos valores corresponde a la siguiente visualización:

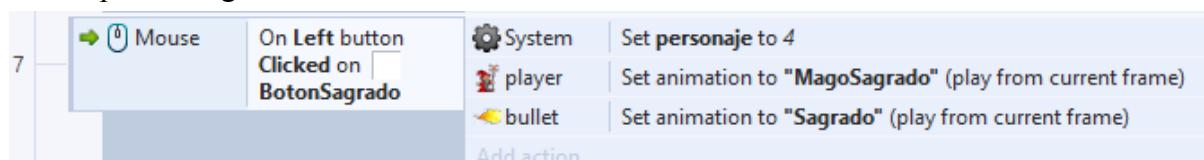




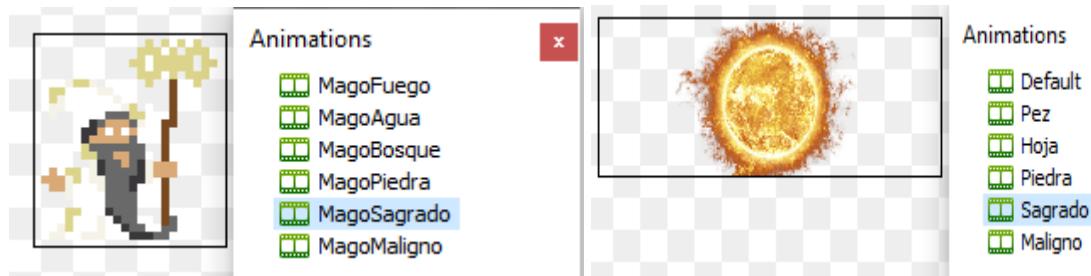
En el caso de hacer clic en el botón cuyo nombre aparece como “Sagrado”



Se cumplira lo siguiente:



La variable global personaje obtendrá el valor 4, la animación del player tendrá el valor “MagoSagrado”, y la animación correspondiente a la bala(bullet) obtendrá el valor “Sagrado”, estos valores corresponde a la siguiente visualización:

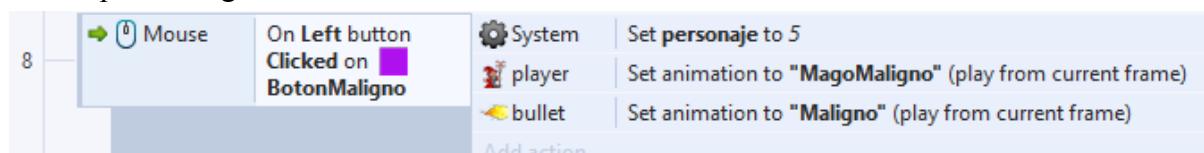




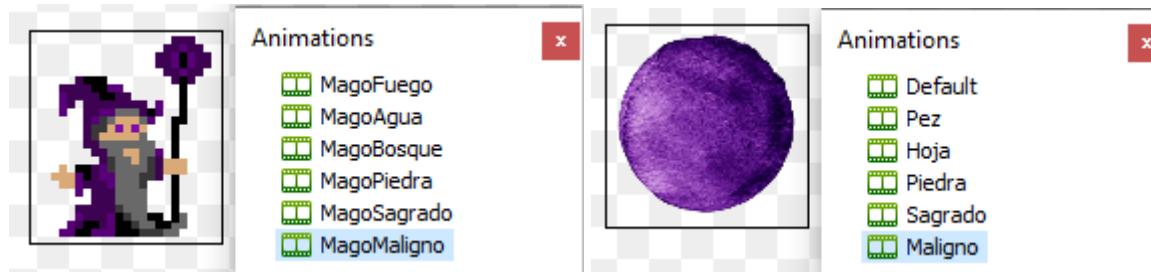
En el caso de hacer clic en el botón cuyo nombre aparece como “Maligno”



Se cumplira lo siguiente:



La variable global personaje obtendrá el valor 5, la animación del player tendrá el valor “MagoMaligno”, y la animación correspondiente a la bala(bullet) obtendrá el valor “Maligno”, estos valores corresponde a la siguiente visualización:





En el caso de hacer clic en el botón cuyo nombre aparece como “Continuar”



Se cumplira lo siguiente:



Es decir, se pasará a la [Escena 1](#).



Hoja de Eventos 1

Start page Event sheet 1 x 1 2 3 4 Event sheet Menu 0

- Global number Balas = 1
- Global number música = 1
- Global number VIDAUSUARIO = 3
- Global number PersonajeGlobal = 0
- Global number nivel = 1
- Global number velocidadcamara = 0.13
- Global number distanciacamara = 48

- 1 System
- 11 ControlarPlayer
- 21 IA CPU
- 28 IA CPU2
- 42 Disparar
- 69 Fin
- 78 Camara
- 82 audio

Add event

En esta hoja de eventos, estos se dividen en diferentes grupos, pero antes se va a mostrar las variables globales en detalle:

Global number Balas = 1
Global number música = 1
Global number VIDAUSUARIO = 3
Global number PersonajeGlobal = 0
Global number nivel = 1
Global number velocidadcamara = 0.13
Global number distanciacamara = 48

- La variable bala como máximo podrá ser 5, es decir, solo pueden haber 4 balas al mismo tiempo en la pantalla.
- La variable música, es para que cuando se acabe la canción, automáticamente se repita en bucle, esto se hace modificando la variable música.
- La variable VIDAUSUARIO, hace referencia como su nombre indica a la vida del jugador, se representará con los corazones vistos anteriormente.
- La variable PersonajeGlobal, obtiene el valor de la variable personaje del menú, de esta forma se mantiene dicha elección durante todo la demo.
- La variable nivel, indica el nivel que se juega actualmente, y se modificará cada vez que un nivel acabe.
- Las variable velocidadcamara y distanciacamara, representan lo que indica su nombre, con unos valores adecuados para que el movimiento de la cámara no produzca mareos o incomodidad visual.



Ahora se procede a ir desglosando los grupos de eventos y detallando los:

1	System
11	ControlarPlayer
21	IA CPU
28	IA CPU2
42	Disparar
69	Fin
78	Camara
82	audio

System

1	System		
2	System On start of layout	System Set PersonajeGlobal to personaje	
		Corazon1 Stop animation	
		Corazon2 Stop animation	
		Corazon3 Stop animation	
		Add action	
3	System PersonajeGlobal = 0	player Set animation to "MagoFuego" (play from current frame)	
		Add action	
4	System PersonajeGlobal = 1	player Set animation to "MagoAgua" (play from current frame)	
		Add action	
5	System PersonajeGlobal = 2	player Set animation to "MagoBosque" (play from current frame)	
		Add action	
6	System PersonajeGlobal = 3	player Set animation to "MagoPiedra" (play from current frame)	
		Add action	
7	System PersonajeGlobal = 4	player Set animation to "MagoSagrado" (play from current frame)	
		Add action	
8	System PersonajeGlobal = 5	player Set animation to "MagoMaligno" (play from current frame)	
		Add action	
9	Keyboard On R pressed	System Restart layout	
		Add action	
10	System Is on mobile device	BotonJu... Set Visible	
		BotonLeft Set Visible	
		BotonRig... Set Visible	
		BotonSh... Set Visible	
		Add action	

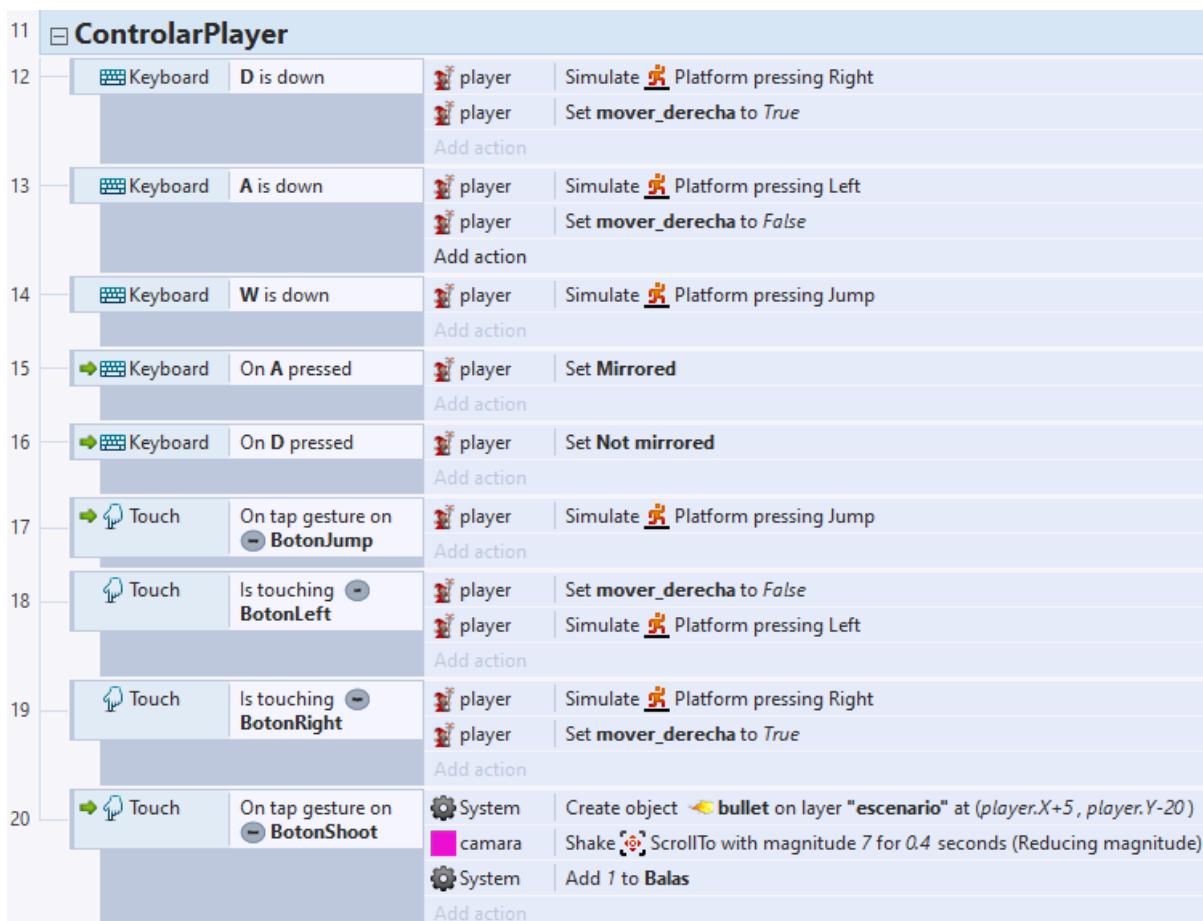


A continuación se va a ir explicando siguiendo los índices que se visualizan en la foto en la parte izquierda:

- **2:** Cuando inicia el layout, se obtiene el valor de la variable global personaje en PersonajeGlobal y así se obtiene el personaje escogido anteriormente, también se paran las animaciones de los corazones (ya que son frames) para que se visualicen rellenos.
- **3:** Si la variable PersonajeGlobal = 0, se selecciona el personaje MagoFuego como se vio anteriormente en la explicación de la hoja de eventos menu.
- **4:** Si la variable PersonajeGlobal = 1, se selecciona el personaje MagoAgua
- **5:** Si la variable PersonajeGlobal = 2, se selecciona el personaje MagoBosque
- **6:** Si la variable PersonajeGlobal = 3, se selecciona el personaje MagoPiedra
- **7:** Si la variable PersonajeGlobal = 4, se selecciona el personaje MagoSagrado
- **8:** Si la variable PersonajeGlobal = 5, se selecciona el personaje MagoMaligno
- **9:** Si se pulsa la tecla “R” del teclado, el nivel se reiniciará.
- **10:** Si el jugador está jugando a la versión de móvil, le aparecerán los botones correspondientes para poder moverse y disparar.



Controlar Player



La explicación será igual que en el apartado anterior, mediante los índices de las izquierda, y de esta forma es realizada en todos los apartados.

- **12:** Si la tecla “D” está siendo pulsada el personaje se moverá hacia la derecha y la variable de instancia mover_derecha tendrá el valor = True.
 - **13:** Si la tecla “A” está siendo pulsada el personaje se moverá hacia la izquierda y la variable de instancia mover_derecha tendrá el valor = False.
 - **14:** Si la tecla “W” está siendo pulsada el personaje realiza un salto.
 - **15:** Si la tecla “A” está siendo presionada, el personaje se pondrá en modo espejo, es decir, se colocará orientado hacia la izquierda.
 - **16:** Si la tecla “D” está siendo presionada, el personaje no estará en modo espejo, y por lo tanto, estará en su orientación normal, que es hacia la derecha.
 - **17:** Esta opción y las siguientes son para en referencia a los botones desde móvil, en este caso, si se aprieta el botón BotonJump, el personaje realizará un salto.
 - **18:** Si se pulsa el botón BotonLeft, el personaje se moverá hacia la izquierda, y la variable de instancia mover_derecha tendrá el valor = False.
 - **19:** Si se pulsa el botón BotonRight, el personaje se moverá hacia la derecha, y la variable de instancia mover_derecha tendrá el valor = True.



- **20:** Si se pulsa el botón BotonShoot, se dispara una bala con origen aproximado en el centro de la parte superior del bastón que posee cada mago, y saldrá en la orientación en la que esté desplazándose el personaje. Además se añadirá 1 a la variable global Balas, y se produce un efecto shake en la cámara (pequeño efecto de movimiento que proporciona un plus visual).

IA CPU

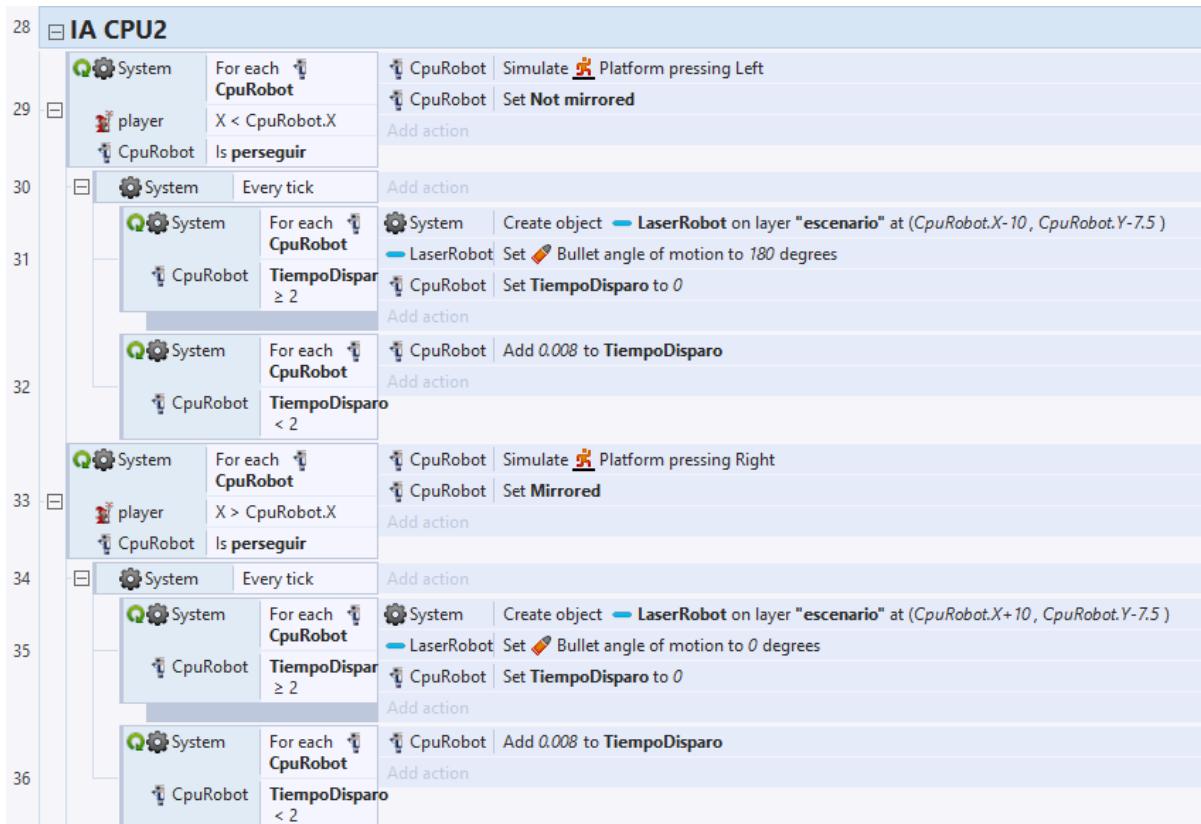
21 □ IA CPU	
22	System For each cpu player X < cpu.X Is perseguir
23	System For each cpu player X > cpu.X Is perseguir
24	System For each cpu System distance(player.X,player.Y,cpu.X,cpu.Y) < 300
25	cpu Simulate Platform pressing Left
26	cpu Set Not mirrored
27	cpu Simulate Platform pressing Right
	cpu Set Mirrored
	cpu Set perseguir to True
	Add action
25	cpu Simulate Platform pressing Jump
26	cpu Simulate Platform pressing Jump
	Add action
27	cpu Play Alerta not looping at volume -15 dB (tag "")
	Exclamac... Set Visible
	System Wait 0.4 seconds
	Exclamac... Set Invisible
	Add action

- **22:** Por cada instancia de cpu (zombie), si dicha instancia está a la derecha del personaje (coordenadas $x < \text{cpu.x}$) y la variable perseguir del cpu en concreto es True, este comenzará a moverse hacia la izquierda sin modo espejo.
- **23:** Por cada instancia de cpu, si dicha instancia está a la izquierda del personaje (coordenadas $x > \text{cpu.x}$) y la variable perseguir del cpu en concreto es True, este comenzará a moverse hacia la derecha en modo espejo, para que su orientación sea en dirección derecha.



- **24:** Por cada instancia de cpu, si la distancia con el jugador es igual a = $(\text{player.X}, \text{Player.Y}, \text{cpu.X}, \text{cpu.Y}) < 300$, la variable perseguir de esa instancia tendrá el valor True, y comenzará a desplazarse hacia el jugador.
- **25:** Si un cpu tiene una plataforma a la izquierda (una pared) realizará un salto.
- **26:** Si un cpu tiene una plataforma a la derecha (una pared) realizará un salto.
- **27:** Si un cpu comienza a moverse, se activará un sonido (Alerta) y se hará visible la exclamación que se encuentra encima de su cabeza durante 0.4 segundos, y luego volverá a ser invisible. Esto se realizará cuando la distancia del jugador sea una determinada con respecto a la exclamación
 $(290 \leq \text{distance}(\text{player.X}, \text{player.Y}, \text{Exclamacion.X}, \text{Exclamacion.Y}) \leq 320)$

IA CPU2

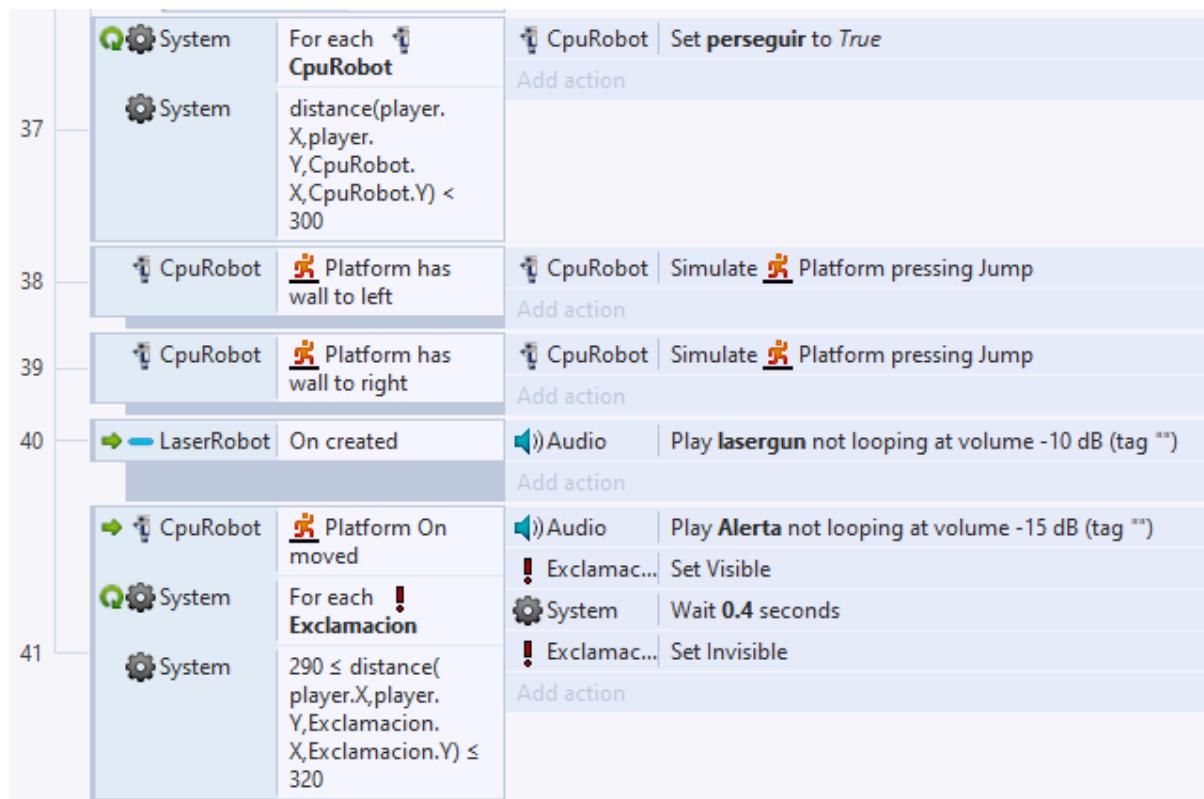


- **29:** Por cada instancia de CpuRobot, si Player.X < CpuRobot.X y la variable de la instancia CpuRobot perseguir es True, éste se moverá hacia la izquierda persiguiendo al jugador y sin estar en modo espejo.
- **30:** Cada tick (cien nanosegundos) se comprobarán los subapartados dentro de este.
- **31:** Por cada CpuRobot se crea un objeto laserRobot aproximadamente a la altura de la pistola de esta instancia de cpu, se crea en un ángulo de 180º, para que vaya recto hacia delante (en la orientación que esté mirando), esto se producirá cada “2”



segundos (aproximadamente debo comprobarlo), una vez que se dispare la variable TiempoDisparo se reiniará a 0.

- **32:** Por cada instancia de CpuRobot, cada tick , añadirá 0.008 al TiempoDisparo hasta que TiempoDisparo sea 2.
- **33,34,35,36:** igual que los apartados anteriores pero con el personaje hacia la derecha.



- **37:** Por cada instancia de CpuRobot se coloca su variable perseguir a True, si la distancia(Player.X,Player.Y,CpuRobot.X,CpuRobot.Y) < 300
- **38,39:** si la instancia de CpuRobot tiene una pared a algún lado saltará.
- **40:** Cada vez que esta cpu dispara , se reproducirá el sonido lasergun sin looping
- **41:** Esta parte es exactamente igual que con la otra cpu (zombie) para el hecho de que se active/aparezca el signo de exclamación en la parte superior del personaje de CpuRobot.



Disparar



- **43:** Es como el onTouch del apartado “20” pero con el clic del ratón.
- **44,45,46,47,48,49:** Dependiendo del valor de la variable global PersonajeGlobal, la bala (bullet), tendrá una animación determinada.
- **50:** Si el usuario no se está moviendo hacia la derecha, la bala saldrá en dirección a la izquierda.
- **51:** Igual que el apartado anterior, pero en inverso , todo hacia la derecha.
- **52:** Por cada instancia de bala que se encuentra fuera de la pantalla, ésta se destruirá y se restará 1 a la variable Balas. (Sino causaría problemas de rendimiento).
- **53:** Si la bala colisiona con una instancia Cpu, se destruirá, y se restará 1 a la variable Balas y se sumará 1 a la variable global Score. Se actualizará el texto “Score” con la puntuación actualizada y se restará 1 la variable cpulife de la instancia Cpu con la que haya colisionado la bala.



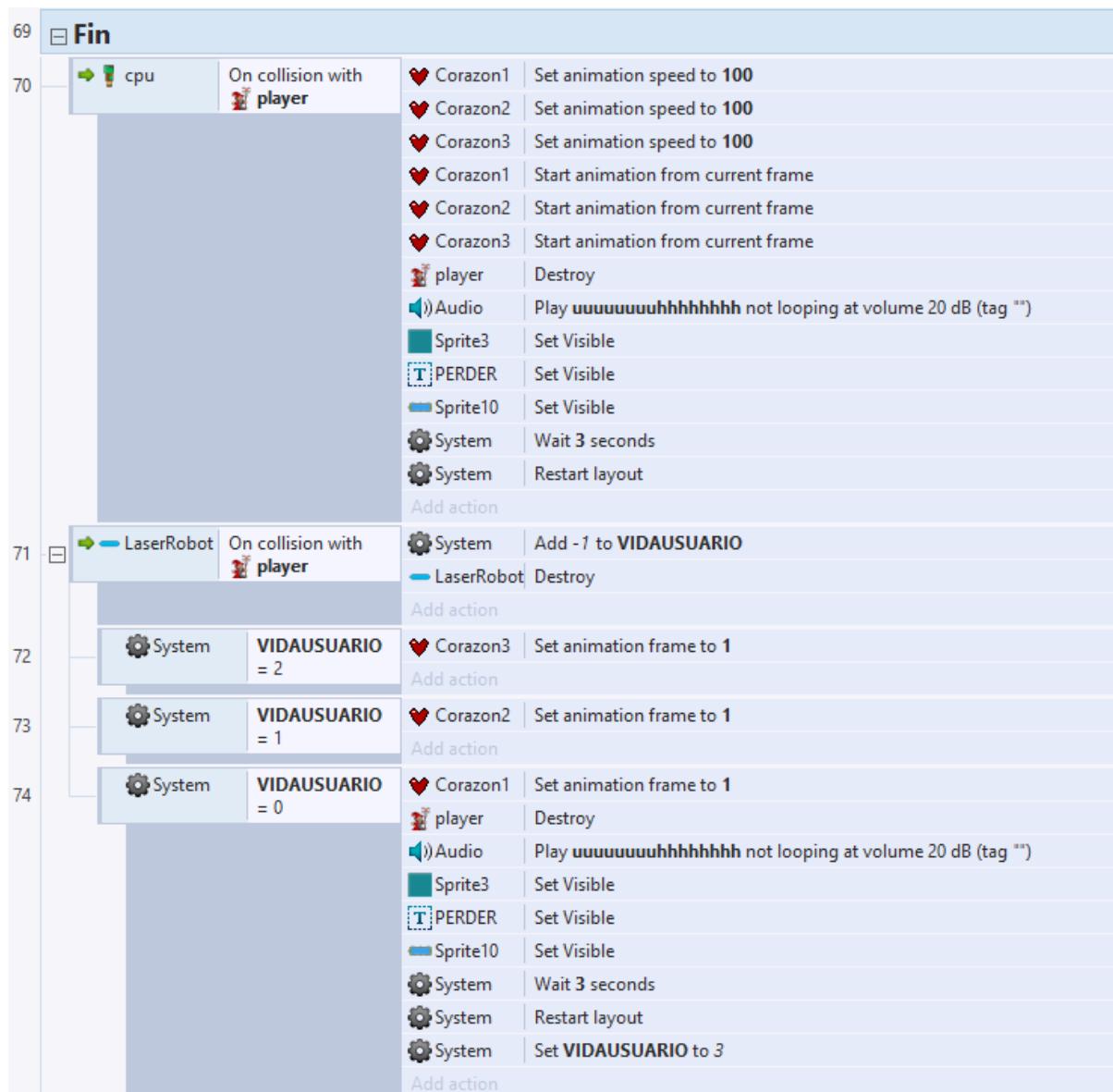
54	Pinchos	On collision with cpu	cpu Add -1 to cpulife
			Add action
55	cpu	cpulife = 5	cpu Set animation to "Default" (play from current frame)
			Add action
56	cpu	cpulife = 4	cpu Set animation to "Sangre2" (play from current frame)
			Add action
57	cpu	cpulife = 3	cpu Set animation to "Sangre3" (play from current frame)
			Add action
58	cpu	cpulife = 2	cpu Set animation to "Sangre4" (play from current frame)
			Add action
59	cpu	cpulife = 1	cpu Set animation to "Sangre5" (play from current frame)
			Add action
60	System	For each cpu	cpu Destroy
		cpu cpulife = 0	Add action
61	bullet	On collision with CpuRobot	bullet Destroy
	System	For each CpuRobot	System Add -1 to Balas
			player Add 1 to Score
			Score Set text to player.Score
			CpuRobot Add -1 to robotlife
			Add action
62	Pinchos	On collision with CpuRobot	CpuRobot Add -1 to robotlife
			Add action
63	CpuRobot	robotlife = 5	cpu Set animation to "Default" (play from current frame)
			Add action
64	cpu	cpulife = 4	cpu Set animation to "Sangre2" (play from current frame)
			Add action
65	cpu	cpulife = 3	cpu Set animation to "Sangre3" (play from current frame)
			Add action
66	cpu	cpulife = 2	cpu Set animation to "Sangre4" (play from current frame)
			Add action
67	cpu	cpulife = 1	cpu Set animation to "Sangre5" (play from current frame)
			Add action
68	System	For each CpuRobot	CpuRobot Destroy
		CpuRobot robotlife = 0	Add action

- **54:** Si un cpu colisiona con los pinchos, la variable cpulife de esa instancia se le restará 1.
- **55,56,57,58,59:** Dependiendo del valor actual de la variable cpulife, la cpu tendrá una animación determinada
- **60:** Si cpulife = 0 , esa instancia cpu se destruirá.

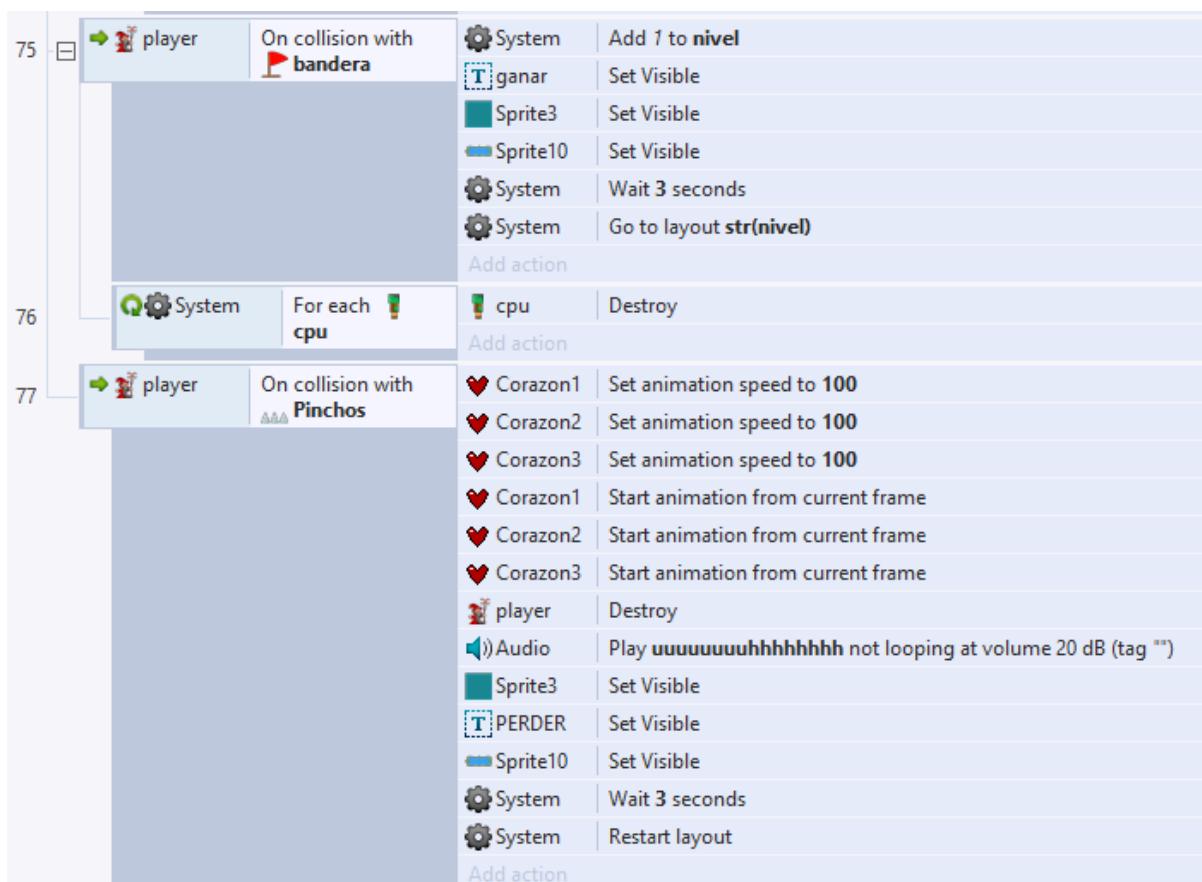


- **61,68:** Sería igual que los apartados 53,60 pero con CpuRobot en lugar de Cpu.

Fin

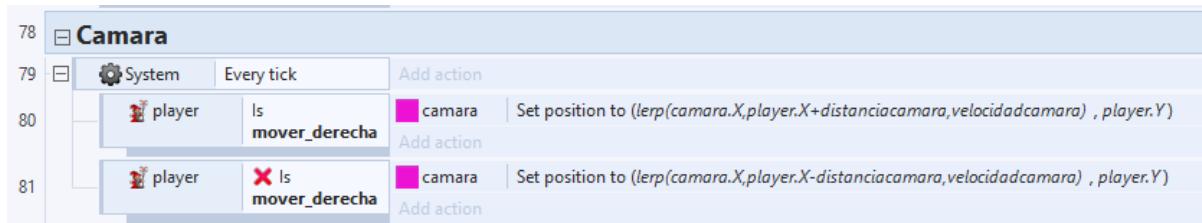


- **70:** Si un cpu colisiona con el jugador, los corazones pasarán a estar vacíos, la instancia de jugador será destruida. Sonará un sonido determinado en ese momento. Y se mostrará un texto que de que has perdido, una vez pasen 3 segundos, el layout se reiniciará.
- **71:** Si un LaserRobot colisiona con el jugador, la variable VIDAUSUARIO se le resta 1 , y la bala láser se destruye.
- **72:** si la variable VIDAUSUARIO = 2, el Corazon3 será hueco.
- **73:** si la variable VIDAUSUARIO = 1, el Corazon2 será hueco.
- **74:** Si la VIDAUSUARIO es 0, ocurrirá lo mismo que en el apartado 70.



- **75:** Si el jugador colisiona con el objeto Bandera, avanzará al nivel siguiente y se mostrará un texto por haber completado el nivel.
- **76:** Al ganar el jugador las instancias cpu se destruirán.
- **77:** Si el usuario colisiona con los pinchos, los corazones se mostrarán huecos, la instancia del jugador será destruida, sonará el sonido de cuando recibe un impacto , y se mostrará el texto de PERDER y posteriormente se reiniciará el layout a los 3 segundos.

Cámara



- **79,80,81:** Cada tick, se comprueba el movimiento del jugador, y según sea este hacia la derecha o izquierda, la cámara se colocará en un lugar determinado.



Audio

82 **audio**
 83 Audio Tag "musica" is playing Audio Play norival (online-audio-converter.com) looping at volume -20 dB (tag "music")
 System musica = 1 System Set musica to 0
 Add action

- 83: Si la música no se está ejecutando, comenzará a reproducirse en loop . (Una vez inicie el juego ya estará en bucle la canción reproduciéndose)

Enlaces a las versiones del Juego

[Juego PC](#)

[Juego Móvil](#)

Para probar el juego en Móvil, se debe abrir desde él y se recomienda que esté en la orientación horizontal.





Unity

Es un motor de videojuegos multiplataforma creado por [Unity Technologies](#), que se lanzó en el año 2005. Está disponible como plataforma de desarrollo para Windows, Mac y Linux. Esta herramienta es para desarrolladores con algo más de experiencia.

Características principales

Usabilidad

Unity puede usarse junto con [Blender](#), [3ds Max](#), [Maya](#), [Softimage](#), [ZBrush](#), [Cinema 4D](#), [Adobe Photoshop](#) y [Adobe Fireworks](#) entre otras aplicaciones. Los cambios que se realicen a los objetos creados con estos productos se actualizan automáticamente en todas las instancias de ese objeto durante todo el proyecto sin necesidad de volver a importar manualmente.

Motor Gráfico

Utiliza [OpenGL](#)(en Windows, Mac y Linux), [Direct3D](#)(solo en Windows), [OpenGL ES](#)(en Android y iOS) e interfaces propietarias (Wii).

Tiene soporte para [mapeado de relieve](#), [mapeado de reflejos](#), [mapeado por paralelo](#), oclusión ambiental en espacio de pantalla, sombras dinámicas utilizando [mapas de sombras](#), [render a textura](#) y [efectos de post-procesamiento](#) de pantalla completa.

Sombreadores y shaders

Se usa para la creación de [sombreadores \(Shaders\)](#) el lenguaje ShaderLab, que es un lenguaje declarativo que se utiliza en archivos fuente sombreador, y utiliza una sintaxis de llaves anidadas para describir un objeto sombreador.

Información a destacar en ShaderLab:

- Definir la estructura general del objeto Shader.
- Usar bloques de código para agregar programas de sombreado escritos en [HLSL](#).
- Usar comandos para establecer el estado de renderizado de la GPU antes de que ejecute un programa de sombreado o para realizar una operación que implique otro paso.
- Exponer propiedades de su código de sombreado para que pueda editarlas en el material/inspector y guardar como parte de un material de assets.
- Especificación de los requisitos del paquete para [SubShaders](#) y [Passes](#). Esto permite a Unity ejecutar ciertos SubShaders y Passes solo cuando se instalan paquetes particulares en el proyecto.



- Definir el comportamiento alternativo para cuando Unity no pueda ejecutar ninguno de los SubShaders con un objeto Shader en el hardware actual.

Se pueden escribir shaders de tres formas distintas:

- Surface shaders, es un enfoque de generación de código que hace que sea mucho más fácil escribir sombreadores iluminados que usar programas de sombreado de píxeles.
- Vertex y Fragment shaders, la diferencia entre ellos es el proceso desarrollado en el proceso de renderizado. Los sombreadores Vertex (de vértices) podrían definirse como un programa de sombreado que modifican la geometría de la escena y realizan la proyección 3D. Los sombreadores Fragment (de fragmentos) están relacionados con la ventana de renderizado y definen el color de cada pixel.
- Shaders de función fija, necesitan ser escritos para hardware viejo que no soporta sombreadores programables.

Un shader puede incluir múltiples variantes y una especificación declarativa de reserva, lo que permite a Unity detectar la mejor variante para la tarjeta gráfica y si no son compatibles, recurrir a un shader alternativo que pueda sacrificar características para una mayor compatibilidad.

Scripting

Viene a través de [Mono](#), la implementación de código abierto de .NET Framework. Los programadores pueden utilizar Unity Script (lenguaje inspirado en la sintaxis de ECMAScript), C# o Boo (inspirados en Python).

A partir de la versión 3.0 añade una versión personalizada de [MonoDevelop](#) para la depuración de scripts.

Control de versiones

Unity incluye Unity Asset Server, una solución de control de versiones para todos los assets de juego y scripts, utilizando PostgreSQL como backend, un sistema de audio construido con la [biblioteca FMOD](#), reproducción de vídeo con códec [Theora](#), un motor de terreno y vegetación, con árboles con soporte de [billboarding](#).





Mecanim

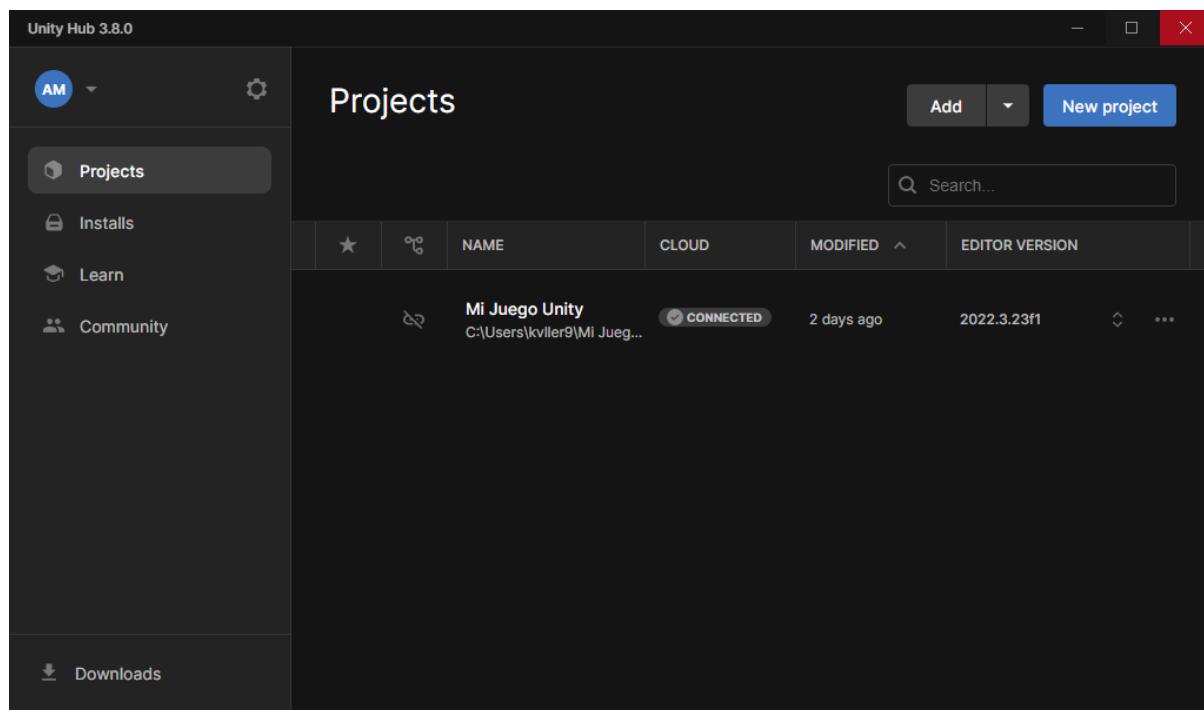
Es la tecnología de animación de Unity. Está diseñada para llevar el movimiento fluido y natural de los personajes con una interfaz eficiente. Incluye herramientas para la creación de máquinas de estados, árboles de mezcla, manipulación de los conocimientos nativos y retargeting automático de animaciones, desde el editor de Unity.

Además, hay una serie de animaciones redestinables que están disponibles en el Unity Asset Store de Unity. Muchos de estos archivos de animación y captura de movimiento son proporcionados sin costo por Unity Technologies. Mecanim también ofrece tanto de pago como gratuitos.



Juego en Unity

Entorno de desarrollo



Unity Hub

Es una aplicación que sirve para gestionar todas las instalaciones y proyectos de Unity.

Funcionalidades principales

1- Gestión de versiones de Unity: Permite instalar y mantener múltiples versiones de Unity los cuales es muy útil si se trabajan en diferentes proyectos que tengan versiones diferentes.

2- Creación y Gestión de Proyectos: Ofrece una interfaz centralizada para crear y acceder a proyectos, además permite configurar y seleccionar plantillas para diferentes tipos de proyectos.

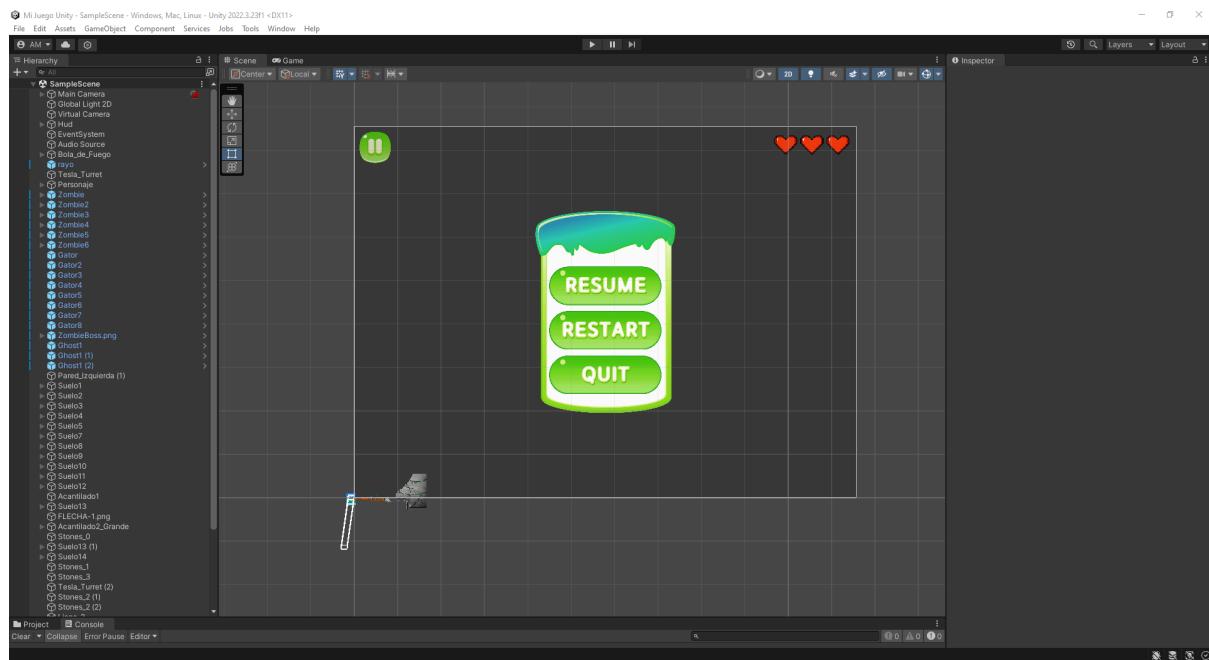
3- Acceso a recursos y documentación: Proporciona acceso a la tienda de Assets de Unity, y enlaces directos para diferentes recursos de aprendizaje.

4- Servicios de Unity: Permite la administración de las licencias de Unity e integración con servicios en la nube de Unity como Unity Collaborate.

5- Proyectos recientes: Muestra una lista de los proyectos más recientes en los que se entró.



Unity



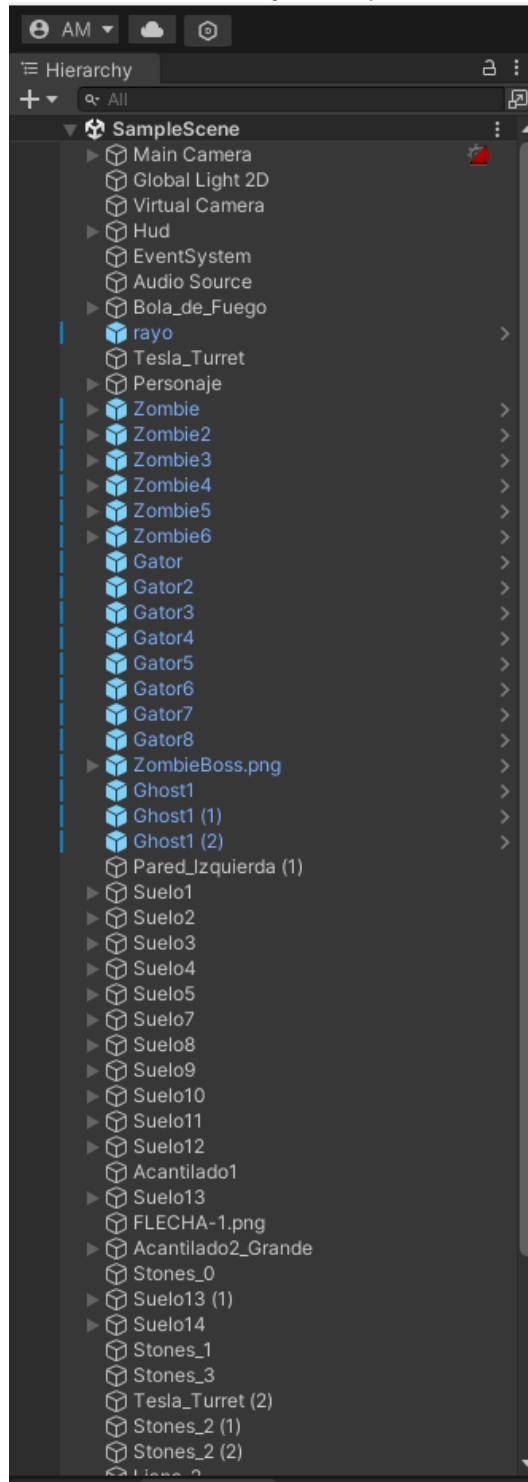
Así es como se visualiza un proyecto en desarrollo cuando se abre, en este caso está seleccionado el objeto HUD, por ese motivo en la ventana central se visualiza el menú de pausa.



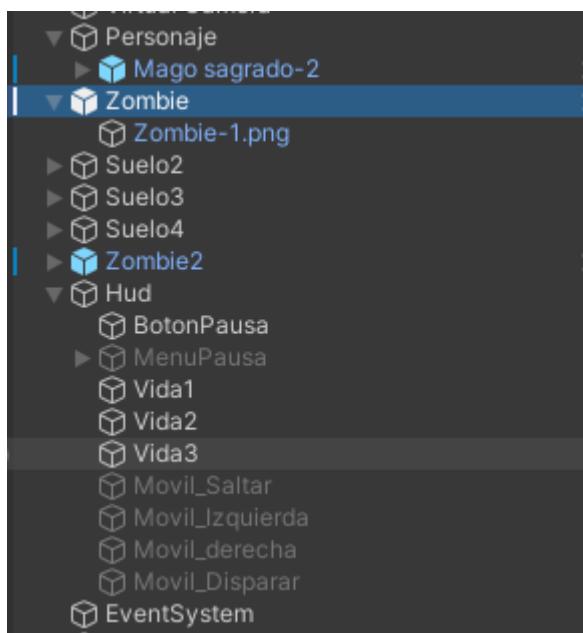


Mi Juego Unity - SampleScene - Windows, Mac, Linux - Ur

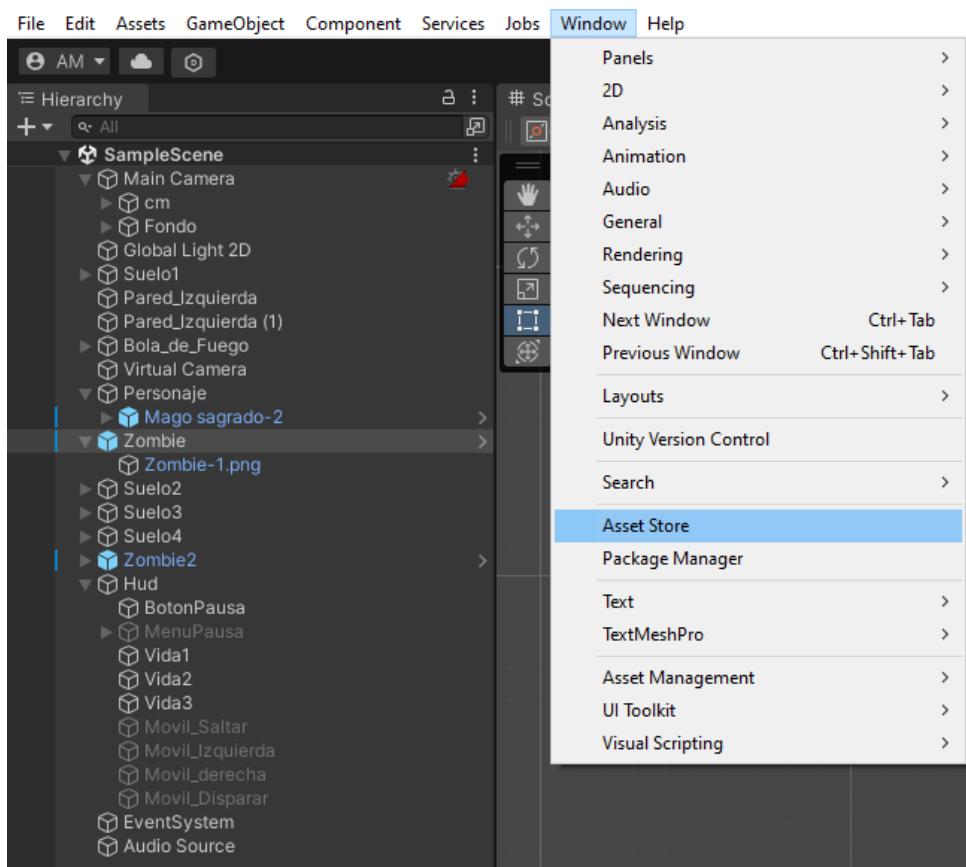
File Edit Assets GameObject Component Services



En la ventana izquierda, se visualizan todos los objetos que pertenecen al videojuego.



Algunos objetos pueden estar formados por más en su interior, ya sean sprites, botones, imágenes etc.



De los menús superiores destacar la opción de ir a la Asset Store de Unity, y los menús de GameObject y Component.



GameObject

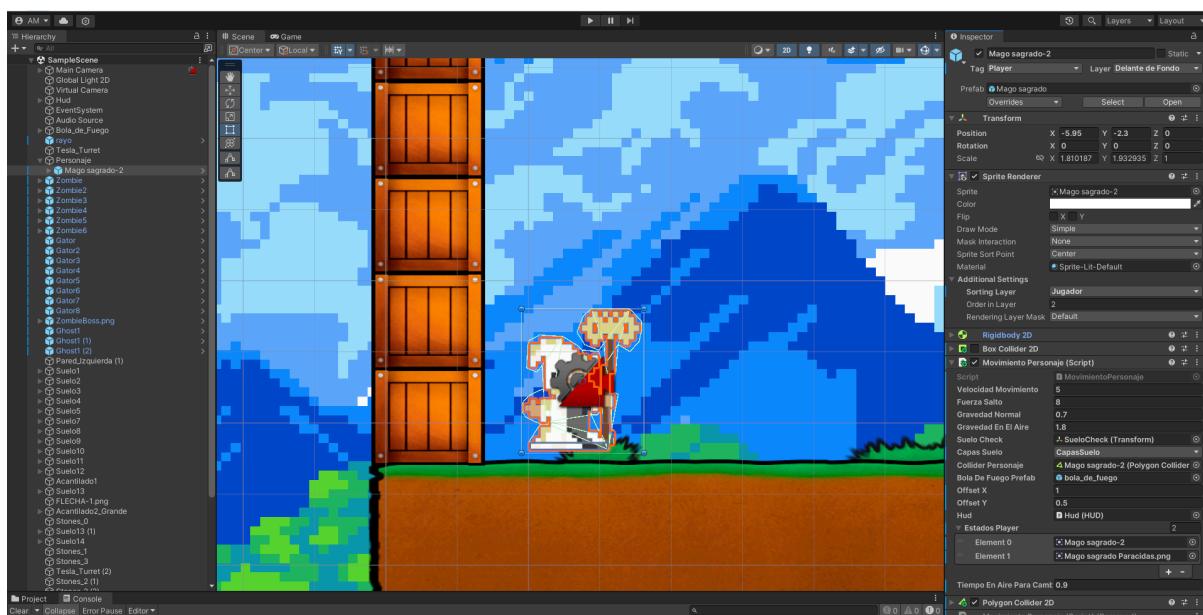
Es el apartado donde podemos crear objetos 2D, 3D, cámaras, audios etc. Cualquier tipo de objeto permitido en Unity.

Component

En este apartado se pueden añadir físicas, videos, audios, ui, efectos etc.

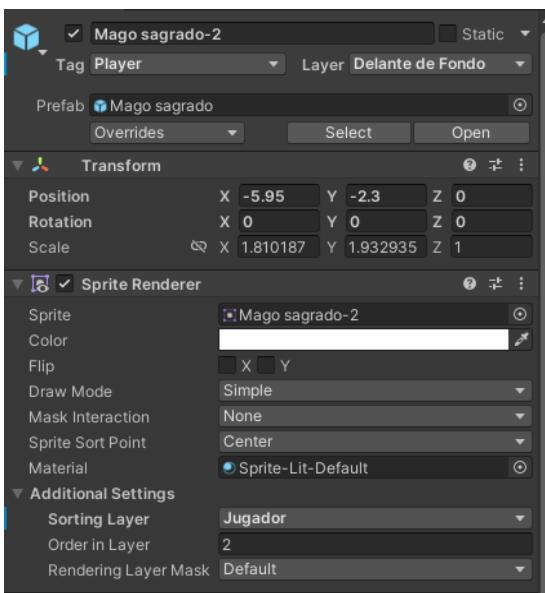
A continuación iré mostrando los diferentes objetos del videojuego, comenzando por el personaje principal.

Personaje Principal

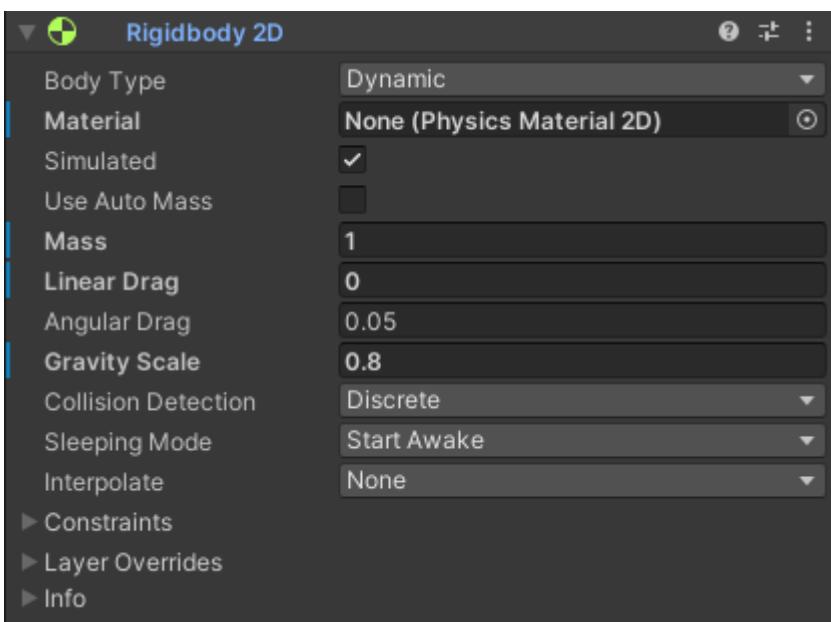


Al seleccionar un objeto en concreto, en la ventana derecha se podrá modificar todo de este mismo.. En este caso está seleccionado el personaje principal, se puede cambiar el tamaño del sprite,su tag y layer, orientación, la imagen etc.

Posee el **Tag Player** para hacer referencia a él en otros scripts que lo requieran.



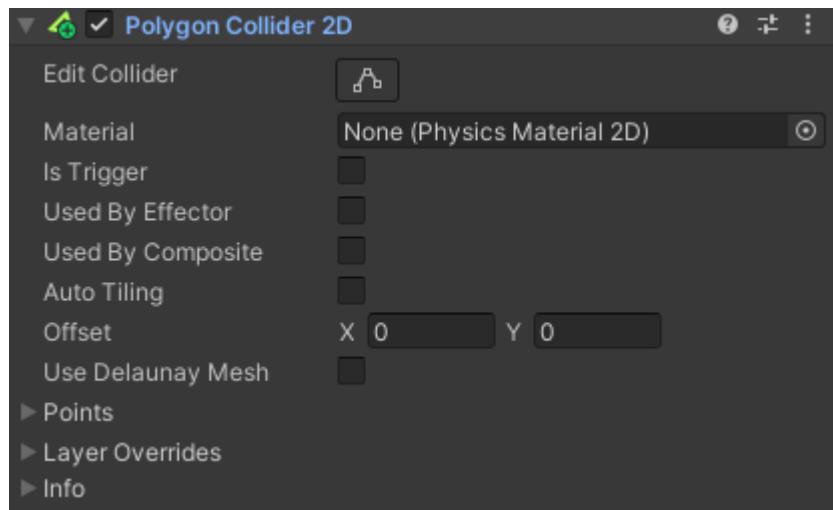
RigidBody



En este apartado se manejan las físicas del objeto, si tiene movimiento, cuánto es su masa, si tiene algún material en concreto, cuánto le afecta la gravedad etc.

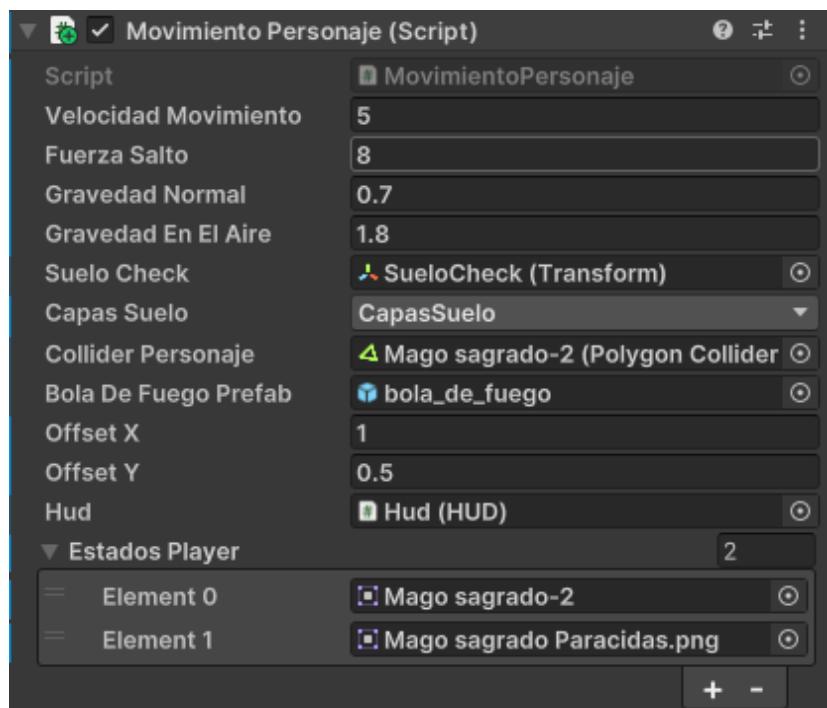


Polygon Collider 2D



Aquí se edita la hitbox (área que ocupa y que podría colisionar con otros objetos), está puede ser Polygon que permite hacer la hitbox personalizada al completo, o bien Box Collider, Circle Collider etc , que tienen una forma predefinida.

Script



En los scripts se pueden visualizar todas las variables asignadas y referencias a otros objetos y otros scripts. En este caso el script se llama **MovimientoPersonaje**.



```

public class MovimientoPersonaje : MonoBehaviour
{
    public float velocidadMovimiento = 5f; // Velocidad de movimiento del personaje
    public float fuerzaSalto = 20f; // Fuerza del salto del personaje
    public float gravedadNormal = 0.8f; // La gravedad normal de tu juego
    public float gravedadEnElAire = 1.6f; // Una gravedad más alta mientras está en el
aire
    public Transform sueloCheck;
    public LayerMask capasSuelo; // Capas que representan el suelo
    public Collider2D colliderPersonaje; // Collider del personaje
    public GameObject bolaDeFuegoPrefab; // Prefab de la bola de fuego
    public float offsetX = 1f; // Desplazamiento horizontal
    public float offsetY = 0.5f; // Desplazamiento vertical
    private int vidas=3;
    public HUD hud;
    private Rigidbody2D rb;
    private SpriteRenderer spriteRenderer;
    public Sprite[] estadosPlayer; // Array que contendrá las imágenes de los diferentes
estados del player
    public float tiempoEnAireParaCambio = 0.9f; // Tiempo en segundos para cambiar el
sprite en el aire
    private float tiempoEnAireActual = 0.0f;
    private bool espacioPulsado = false;
    Color nuevoColor = new Color(1f, 0.725f, 0.725f);

    void Start()
    {
        rb = GetComponent<Rigidbody2D>();
        spriteRenderer = GetComponent<SpriteRenderer>();
    }

    void Update()
    {
    }
}

```



```

transform.rotation = Quaternion.Euler(transform.rotation.eulerAngles.x,
transform.rotation.eulerAngles.y, 0f);

// Obtener la entrada horizontal (A y D, o las flechas izquierda y derecha)
float movimientoHorizontal = Input.GetAxis("Horizontal");

// Calcular la velocidad del movimiento
Vector2 velocidad = new Vector2(movimientoHorizontal * velocidadMovimiento,
rb.velocity.y);

// Aplicar la velocidad al Rigidbody2D del personaje
rb.velocity = velocidad;

// Voltear el sprite horizontalmente si se mueve hacia la izquierda
if (movimientoHorizontal < 0)
{
    spriteRenderer.flipX = true; // Voltear el sprite hacia la izquierda
}
else if (movimientoHorizontal > 0)
{
    spriteRenderer.flipX = false; // Mantener el sprite en su orientación original
}

// Verificar si el personaje está en el suelo y se presiona la tecla de salto
if (Input.GetKeyDown(KeyCode.Space) && EstaEnElSueloPies())
{
    // Aplicar una fuerza hacia arriba para simular el salto
    rb.AddForce(Vector2.up * fuerzaSalto, ForceMode2D.Impulse);

    rb.gravityScale = gravedadEnElAire;
}

// Verificar si se mantiene pulsado el espacio
if (Input.GetKey(KeyCode.Space))

```



```

{
    espacioPulsado = true;
}
else
{
    espacioPulsado = false;
}

// Verificar si el personaje está en el aire y ha pasado el tiempo necesario para el
cambio de sprite
if (!EstaEnElSuelo() && espacioPulsado)
{
    tiempoEnAireActual += Time.deltaTime;

    // Cambiar el sprite si el tiempo en el aire supera el umbral establecido
    if (tiempoEnAireActual >= tiempoEnAireParaCambio)
    {
        spriteRenderer.sprite = estadosPlayer[1];
    }
}
else
{
    // Reiniciar el temporizador si el personaje no está en el aire
    tiempoEnAireActual = 0.0f;
    spriteRenderer.sprite = estadosPlayer[0];
}

// Verificar si se presiona la barra espaciadora
if (Input.GetKeyDown(KeyCode.W))
{
    // Levantar al personaje y ponerlo recto
    rb.velocity = new Vector2(rb.velocity.x, 0f); // Detener cualquier movimiento
    vertical
}

```



```

    }

    // Verificar si se hace clic
    if (Input.GetMouseButtonDown(0))
    {
        LanzarBolaDeFuego();
    }

    if (vidas == 0)
    {
        hud.Pausa();
    }

}

void OnCollisionEnter2D(Collision2D collision)
{
    if (collision.gameObject.CompareTag("Zombie") ||
    collision.gameObject.CompareTag("Gator") ||
    collision.gameObject.CompareTag("Pinchos") ||
    collision.gameObject.CompareTag("RayoTesla") ||
    || collision.gameObject.CompareTag("ZombieBoss") ||
    collision.gameObject.CompareTag("Ghost"))
    {
        PerderVida();
    }

    if (collision.gameObject.CompareTag("Bandera"))
    {
        CompletarDemo();
    }
}

bool EstaEnElSueloPies()
{
}

```



```
// Verificar si hay colisión con el suelo en la posición del objeto de verificación del suelo
//capassuelo
Collider2D[] colliders = Physics2D.OverlapCircleAll(sueloCheck.position, 0.2f, capasSuelo);

rb.gravityScale = gravedadNormal;

// El personaje está en el suelo si al menos uno de los colliders está presente
return colliders.Length > 0;
}

bool EstaEnElSuelo()
{
    // Verificar si el collider del personaje está tocando las capas del suelo
    return Physics2D.IsTouchingLayers(colliderPersonaje, capasSuelo);
}

void LanzarBolaDeFuego()
{
    // Obtener la posición del centro del personaje
    Vector3 posicionPersonaje = transform.position;

    // Calcular la dirección de lanzamiento
    Vector2 direccionLanzamiento = spriteRenderer.flipX ? Vector2.left :
    Vector2.right;

    // Calcular la posición desplazada para instanciar la bola de fuego
    Vector3 posicionDesplazada = new Vector3(posicionPersonaje.x + (offsetX *
(spriteRenderer.flipX ? -1 : 1)), posicionPersonaje.y + offsetY, posicionPersonaje.z);

    // Instanciar la bola de fuego en la posición desplazada del centro del personaje
    GameObject bolaDeFuego = Instantiate(bolaDeFuegoPrefab, posicionDesplazada,
Quaternion.identity);
}
```



```

// Aplicar una fuerza a la bola de fuego en la dirección calculada
    bolaDeFuego.GetComponent<Rigidbody2D>().AddForce(direccionLanzamiento *
500f);
}

public void PerderVida() {
    StartCoroutine(GetDamage());
    vidas -= 1;
    hud.DesactivarVida(vidas);
}

IEnumerator GetDamage()
{
    spriteRenderer.color = nuevoColor;
    yield return new WaitForSeconds(0.1f);
    spriteRenderer.color = Color.white;
}

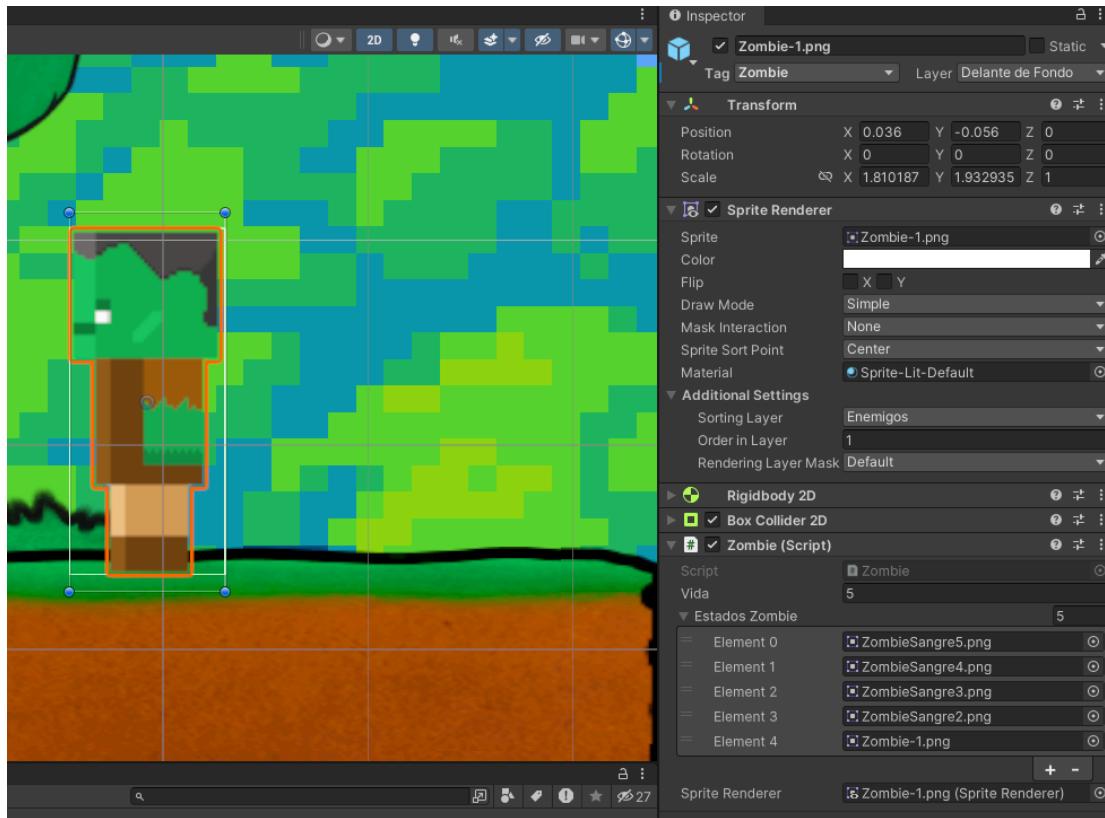
void CompletarDemo()
{
    // Detener el movimiento del personaje
    velocidadMovimiento = 0;
    fuerzaSalto = 0;
    rb.velocity = Vector2.zero; // Detener cualquier movimiento en curso

    // Mostrar el mensaje de finalización de demo
    hud.MostrarMensajeFinalizacion();
}
}

```

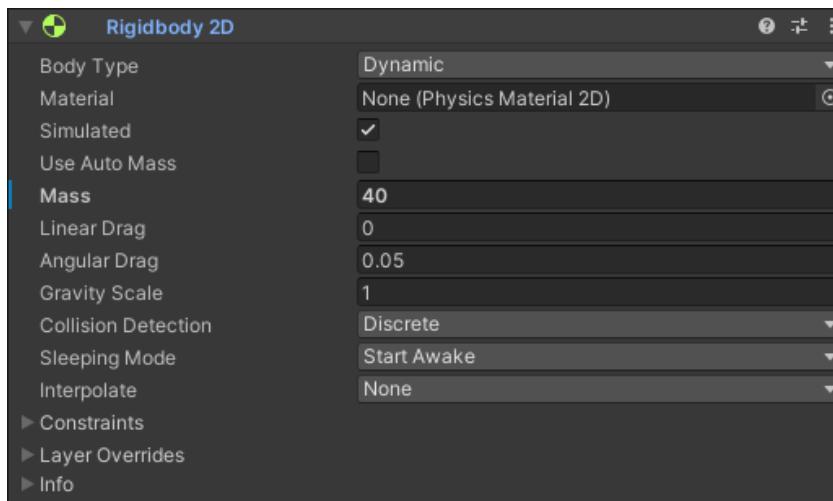


Zombie



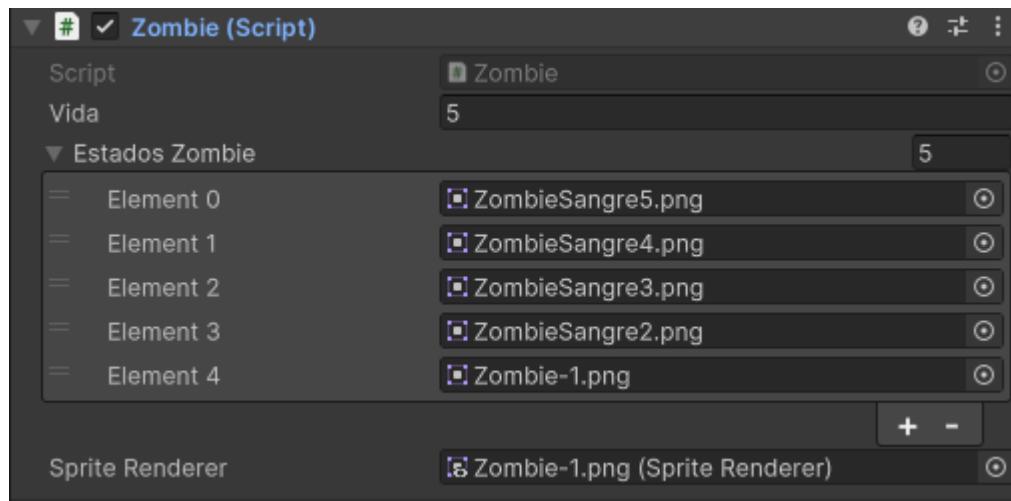
Los zombies tienen el tag de Zombie, una imagen asignada que cambiará con respecto a cuando reciba balas, un rigidbody para darle consistencia, su propio script “Zombie” que posteriormente mostraré y un Box Collider 2D para detectar colisiones con el objeto zombie , este último como se muestra en la imagen es cuadrado alrededor del zombie.

RigidBody





Script



```
public class Zombie : MonoBehaviour
{
    public int vida = 5; // Vida inicial del zombie
    public Sprite[] estadosZombie; // Array que contendrá las imágenes de los diferentes estados del zombie
    public SpriteRenderer spriteRenderer; // Referencia al componente SpriteRenderer del zombie

    private Transform player; // Referencia al transform del jugador principal
    private float velocidadMovimiento; // Velocidad de movimiento del zombie
    private bool esVisible = false; // Indica si el zombie es visible en la cámara
    Color nuevoColor = new Color(1f, 0.725f, 0.725f);

    void Start()
    {
        // Buscar el GameObject del jugador principal por su tag y obtener su transform
        player = GameObject.FindGameObjectWithTag("Player").transform;

        // La velocidad inicial del zombie es el doble de la velocidad normal
        velocidadMovimiento = 2f;
    }

    void Update()
    {
```





```

transform.rotation = Quaternion.Euler(transform.rotation.eulerAngles.x,
transform.rotation.eulerAngles.y, 0f);

// Si el jugador existe, mover el zombie hacia la posición del jugador
if (player != null && esVisible)
{
    // Calcular la dirección hacia el jugador
    Vector3 direccion = (player.position - transform.position).normalized;

    // Calcular la nueva posición hacia la cual moverse
    Vector3 nuevaPosicion = transform.position + direccion * velocidadMovimiento *
Time.deltaTime;

    // Mover el zombie hacia la nueva posición
    transform.position = nuevaPosicion;

    // Visualización en modo espejo
    if (direccion.x < 0) // Si se mueve hacia la derecha
    {
        spriteRenderer.flipX = false; // No voltear la imagen
    }
    else if (direccion.x > 0) // Si se mueve hacia la izquierda
    {
        spriteRenderer.flipX = true; // Voltear la imagen horizontalmente
    }
}

void OnBecameVisible()
{
    // Marcar que el zombie es visible en la cámara
    esVisible = true;
}

```



```

void OnCollisionEnter2D(Collision2D collision)
{
    // Verificar si la colisión es con una bala
    if(collision.gameObject.CompareTag("Bala"))
    {
        // Destruir la bala
        Destroy(collision.gameObject);

        // Reducir la vida del zombie
        vida--;

        // Cambiar la imagen del zombie según la vida restante
        if(vida > 0 && vida <= estadosZombie.Length)
        {
            StartCoroutine(GetDamage());
            spriteRenderer.sprite = estadosZombie[vida - 1]; // Restamos 1 porque los arrays
comienzan en 0
        }
    }

    // Actualizar la velocidad del zombie
    if(vida > 0)
    {
        velocidadMovimiento /= 1.1f;
    }
    else
    {
        // Destruir el objeto del zombie
        Destroy(gameObject);
    }
}

IEnumerator GetDamage()

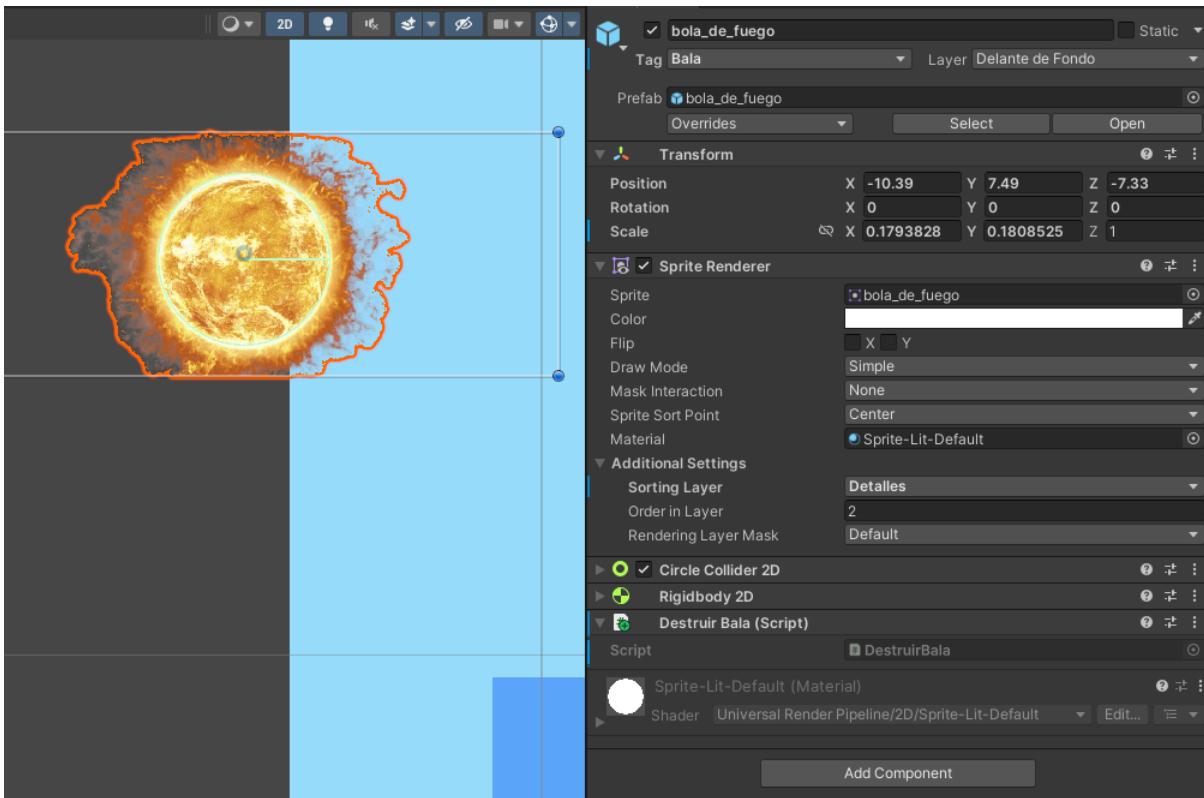
```



```
{  
    spriteRenderer.color = nuevoColor;  
    yield return new WaitForSeconds(0.1f);  
    spriteRenderer.color = Color.white;  
}  
}
```

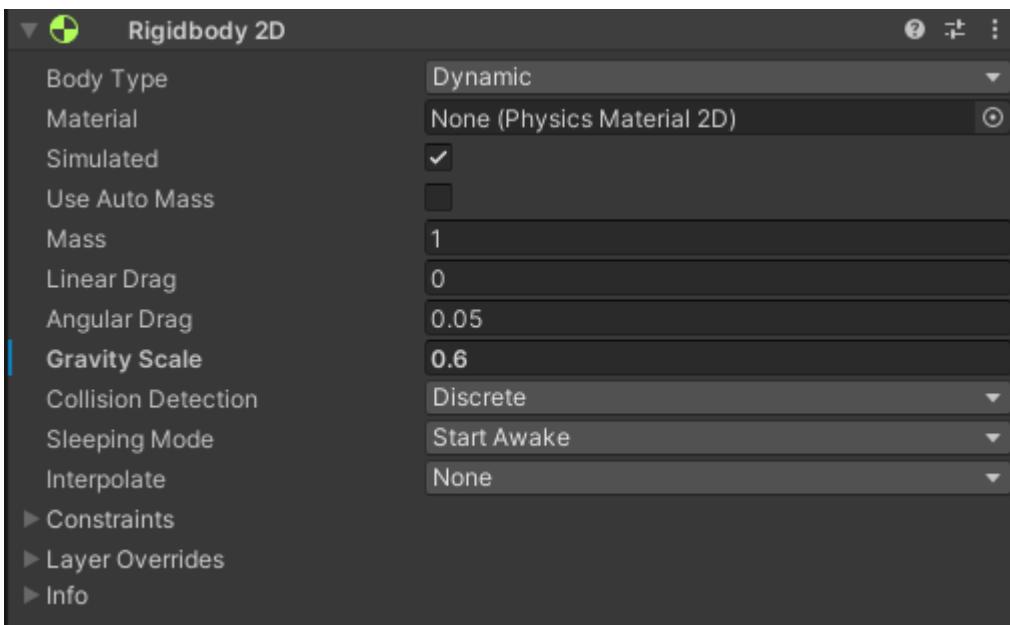


Bola de fuego (Bala)



Es la bala que dispara el personaje principal, posee un Circle Collider 2D, su Rigidbody y su propio script asociado llamado “Destruir Bala” .

RigidBody





Script

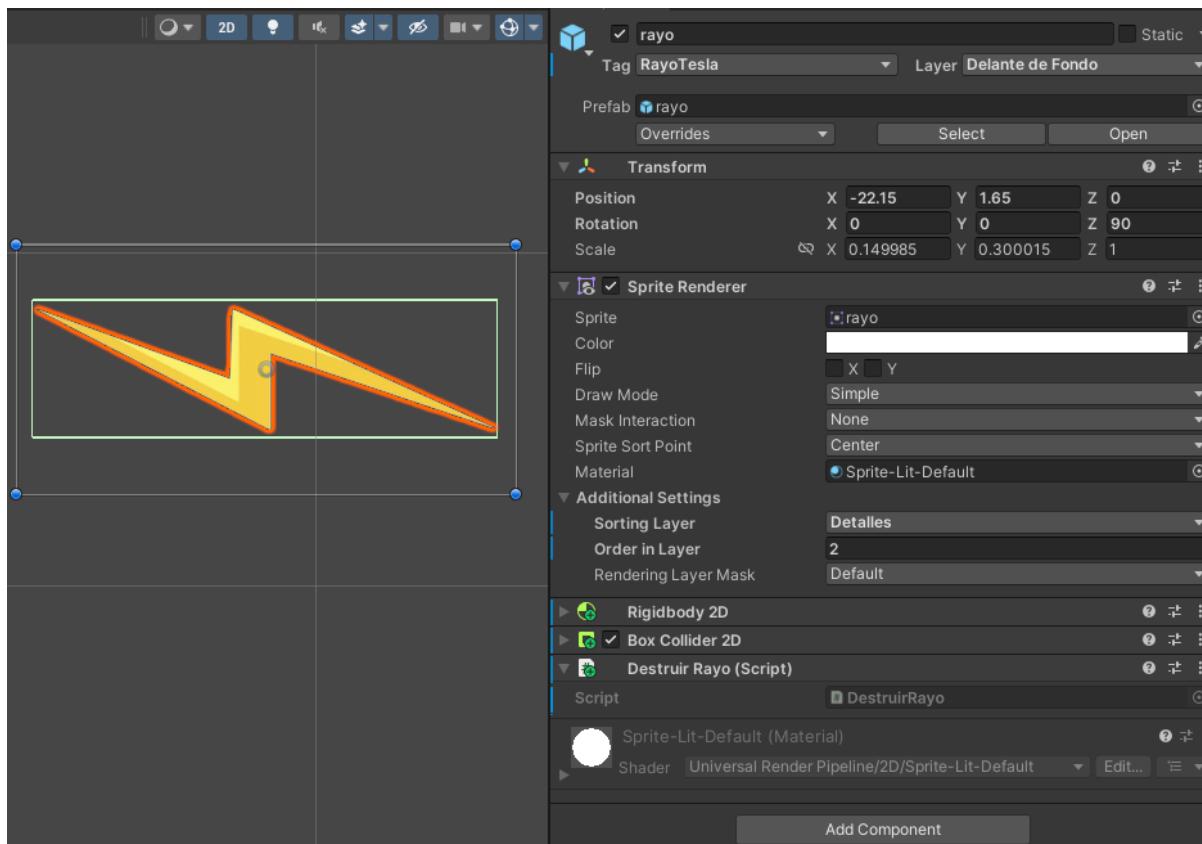
```
public class DestruirBala : MonoBehaviour
{
    void OnCollisionEnter2D(Collision2D collision)
    {
        // Verificar si la bola de fuego colisiona con otro objeto que no sea el personaje
        if (!collision.gameObject.CompareTag("Player"))
        {
            // Destruir la bola de fuego
            Destroy(gameObject);
        }
    }

    // Se llama cuando el objeto ya no es visible por ninguna cámara
    private void OnBecameInvisible()
    {
        // Destruir la bala
        Destroy(gameObject);
    }
}
```



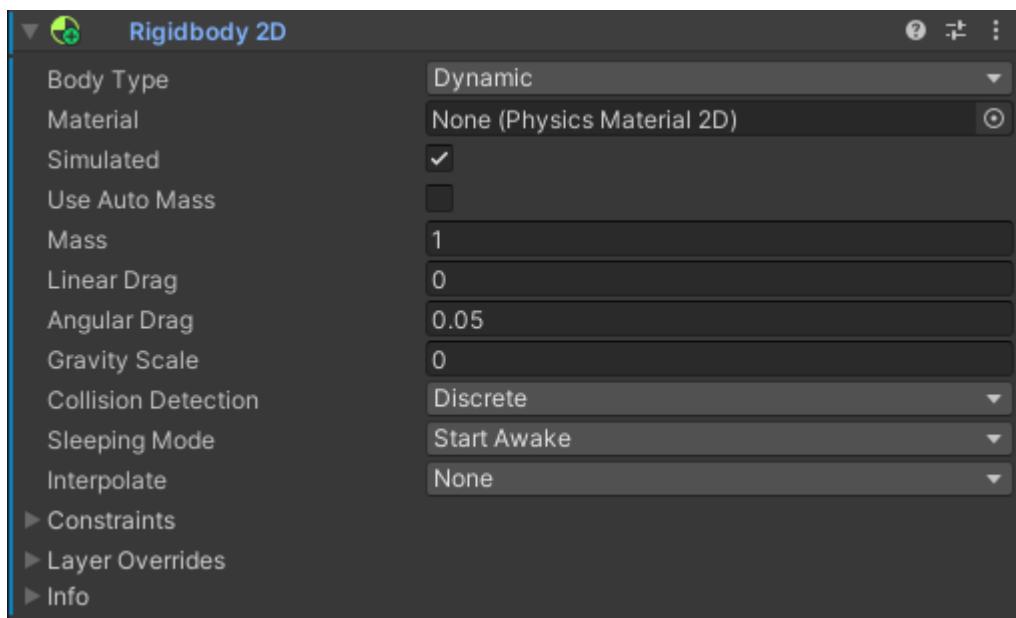


Rayo



Es la bala que dispara la torreta, posee un Box Collider 2D, su RigidBody y su propio script asociado llamado “Destruir Rayo” .

Rigidbody





Script

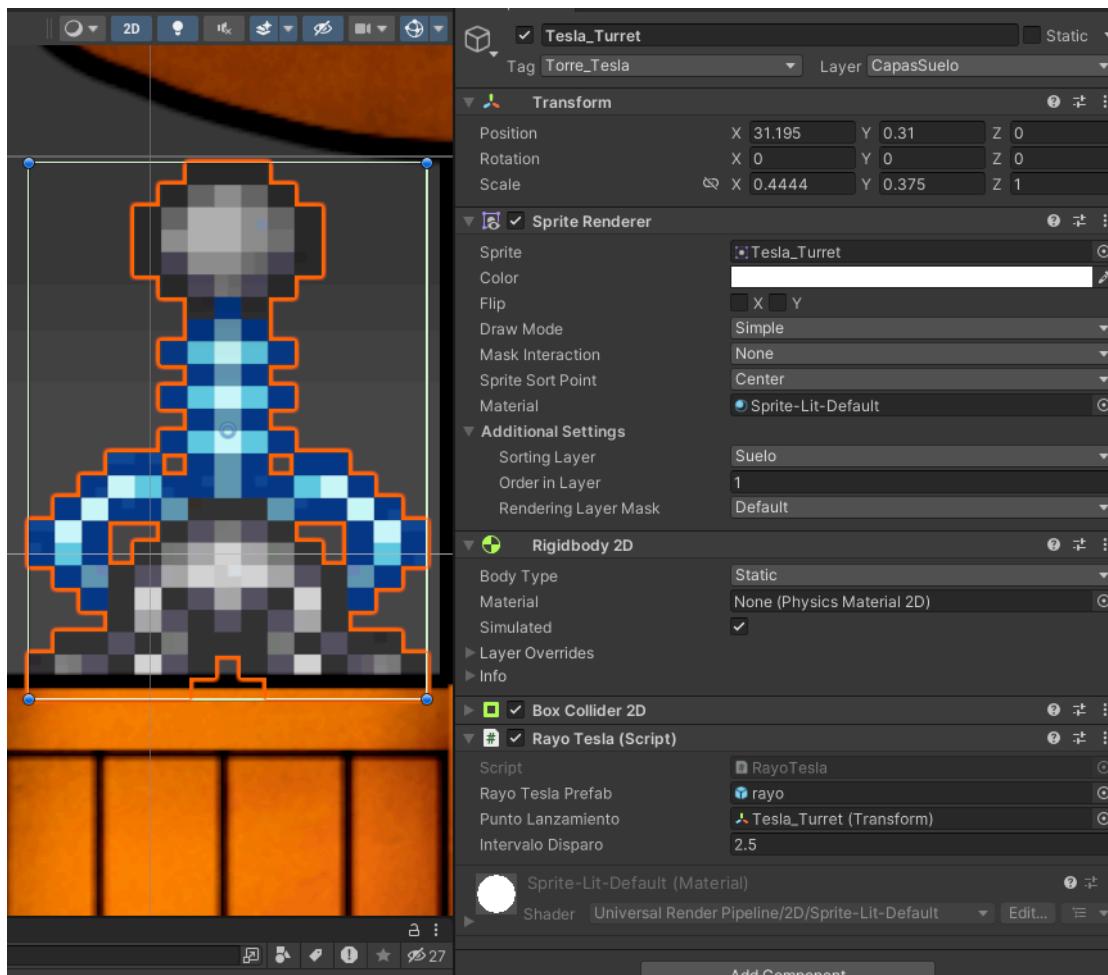
```
public class DestruirRayo : MonoBehaviour
{
    void OnCollisionEnter2D(Collision2D collision)
    {
        // Verificar si el rayo colisiona con otro objeto que no sea el personaje
        if (collision.gameObject.CompareTag("Player") ||
            collision.gameObject.CompareTag("Caja"))
        {
            // Destruir el rayo
            Destroy(gameObject);
        }
    }

    // Se llama cuando el objeto ya no es visible por ninguna cámara
    private void OnBecameInvisible()
    {
        // Destruir el rayo
        Destroy(gameObject);
    }
}
```



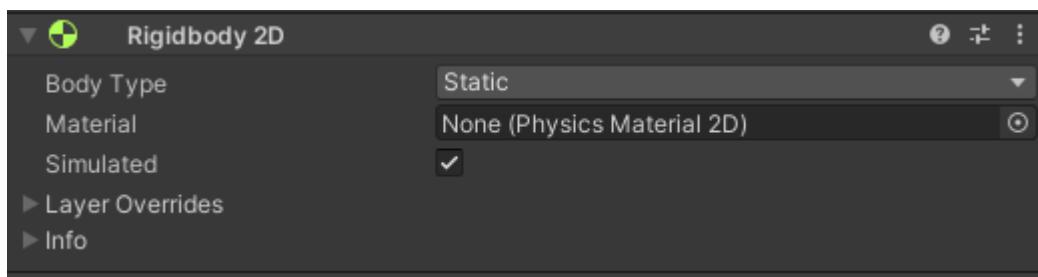


Tesla Turret



Es la torreta que dispara rayos, posee un Box Collider 2D, su Rigidbody y su propio script asociado llamado “Rayo Tesla” . No se puede destruir y está introducida en el videojuego para dificultar el avance del jugador.

RigidBody





Script

```
public class RayoTesla : MonoBehaviour
{
    public GameObject rayoTeslaPrefab; // Prefab del rayo Tesla
    public Transform puntoLanzamiento; // Punto de origen del rayo Tesla
    public float intervaloDisparo = 2.5f; // Intervalo entre disparos en segundos

    private float tiempoUltimoDisparo; // Tiempo del último disparo

    void Update()
    {
        // Verificar si ha pasado el tiempo suficiente para disparar nuevamente
        if (Time.time - tiempoUltimoDisparo >= intervaloDisparo)
        {
            LanzarRayo();
            tiempoUltimoDisparo = Time.time;
        }
    }

    void LanzarRayo()
    {
        if (rayoTeslaPrefab != null && puntoLanzamiento != null)
        {
            // Calcular la posición de lanzamiento del rayo Tesla (a la izquierda del punto de
            // lanzamiento)
            Vector3 posicionLanzamiento = puntoLanzamiento.position - new Vector3(1, 0, 0); // 
            Ajusta el valor de 1 según sea necesario

            // Obtener la rotación para que el rayo Tesla apunte hacia la derecha (rotación de 90
            // grados en el eje Z)
            Quaternion rotacion = Quaternion.Euler(0, 0, 90);

            // Instanciar el rayo Tesla en la posición de lanzamiento con la rotación adecuada
            GameObject rayoTesla = Instantiate(rayoTeslaPrefab, posicionLanzamiento, rotacion);
        }
    }
}
```





```
// Obtener la dirección de lanzamiento (hacia la izquierda)
Vector2 direccionLanzamiento = Vector2.left;

// Aplicar una fuerza al rayo Tesla en la dirección calculada
rayoTesla.GetComponent<Rigidbody2D>().AddForce(direccionLanzamiento * 500f);

}

else

{

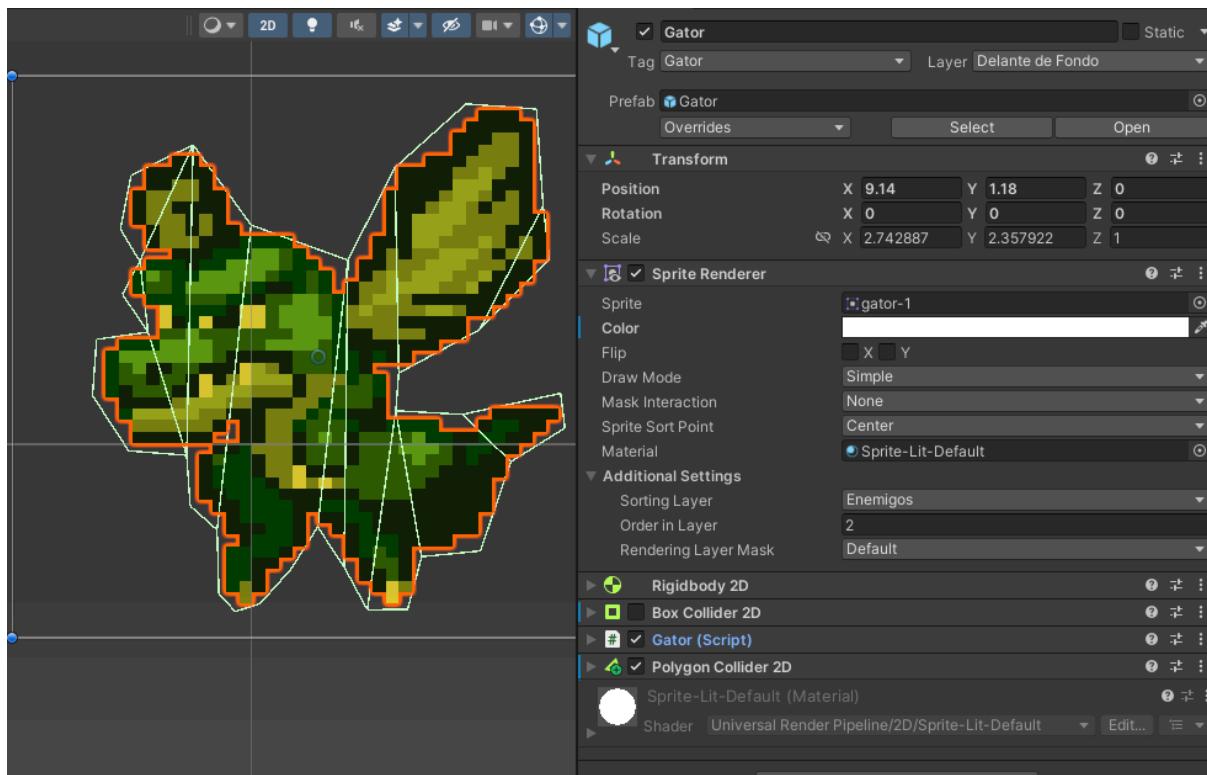
    Debug.LogError("Falta asignar el prefab del rayo Tesla o el punto de lanzamiento.");
}

}

}
```

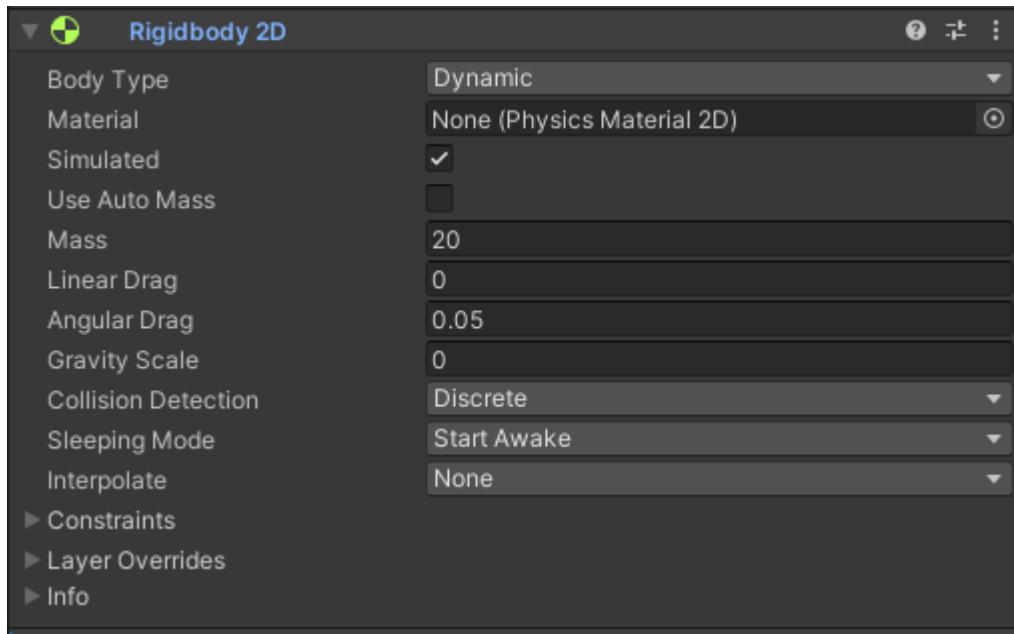


Gator



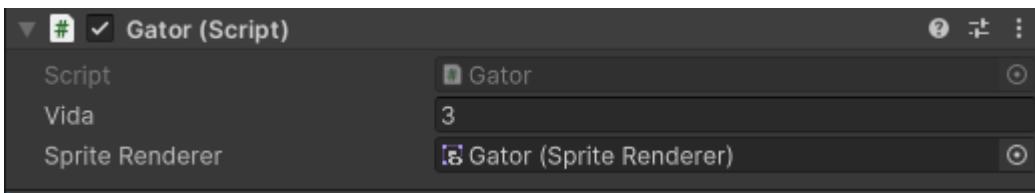
Los gator tienen el tag de Gator, un rigidbody para darle consistencia, su propio script “Gator” y un Polygon Collider 2D para detectar colisiones con el objeto gator.

RigidBody





Script



```
public class Gator : MonoBehaviour
{
    public int vida = 3; // Vida inicial del gator
    public SpriteRenderer spriteRenderer; // Referencia al componente SpriteRenderer del
gator
    private Transform player; // Referencia al transform del jugador principal
    private float velocidadMovimiento; // Velocidad de movimiento del gator
    private bool esVisible = false; // Indica si el gator es visible en la cámara
    private Rigidbody2D rb; // Referencia al Rigidbody2D del gator
    Color nuevoColor = new Color(1f, 0.725f, 0.725f);

    void Start()
    {
        // Buscar el GameObject del jugador principal por su tag y obtener su transform
        player = GameObject.FindGameObjectWithTag("Player").transform;

        velocidadMovimiento = 3f;
        // Obtener el Rigidbody2D del gator
        rb = GetComponent<Rigidbody2D>();

        // Desactivar la gravedad del Rigidbody2D para que el gator no caiga
        rb.gravityScale = 0f;
    }

    void Update()
    {

        transform.rotation = Quaternion.Euler(transform.rotation.eulerAngles.x,
        transform.rotation.eulerAngles.y, 0f);
    }
}
```





```

if (player != null && esVisible)
{
    // Calcular la dirección hacia el jugador
    Vector3 direccion = (player.position - transform.position).normalized;

    // Calcular la nueva posición hacia la cual moverse
    Vector3 nuevaPosicion = transform.position + direccion * velocidadMovimiento *
Time.deltaTime;

    // Mover el zombie hacia la nueva posición
    transform.position = nuevaPosicion;

    // Visualización en modo espejo
    if (direccion.x < 0) // Si se mueve hacia la derecha
    {
        spriteRenderer.flipX = false; // No voltear la imagen
    }
    else if (direccion.x > 0) // Si se mueve hacia la izquierda
    {
        spriteRenderer.flipX = true; // Voltear la imagen horizontalmente
    }
}

void OnBecameVisible()
{
    esVisible = true;
}

void OnCollisionEnter2D(Collision2D collision)
{
    // Verificar si la colisión es con una bala
    if (collision.gameObject.CompareTag("Bala"))
}

```



```
{
    // Destruir la bala
    Destroy(collision.gameObject);

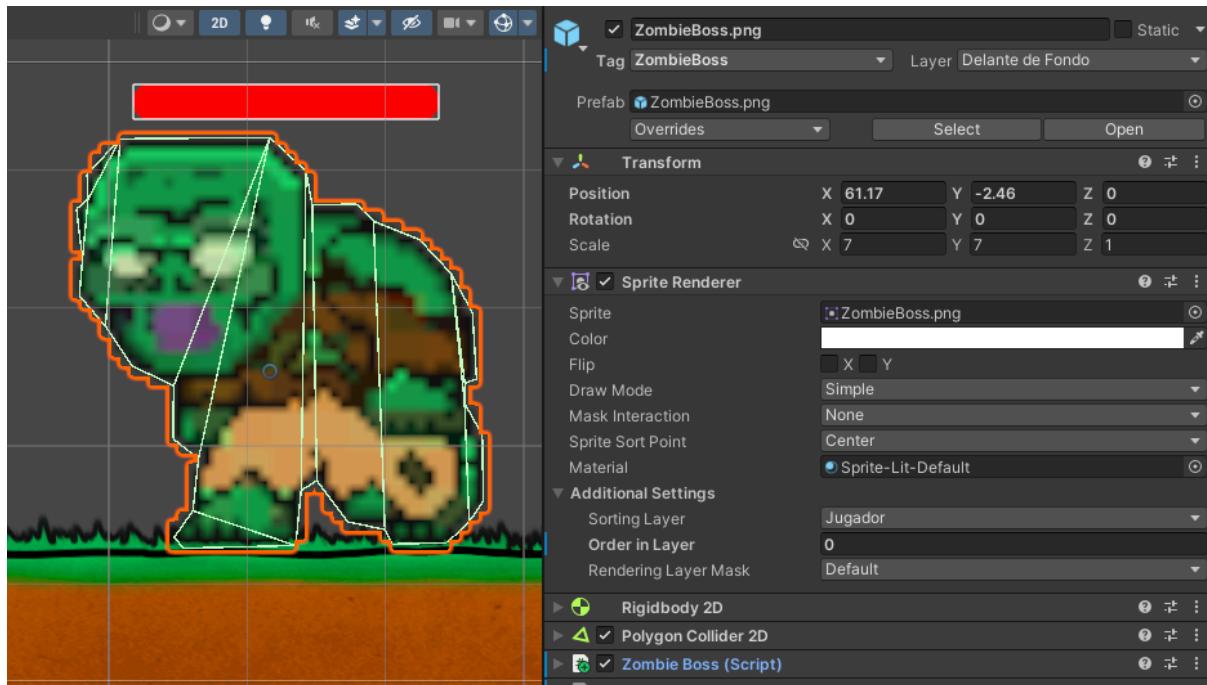
    // Reducir la vida del gator
    vida--;

    if (vida > 0)
    {
        StartCoroutine(GetDamage());
        velocidadMovimiento /= 1.05f;
    }
    else
    {
        // Destruir el objeto
        Destroy(gameObject);
    }
}

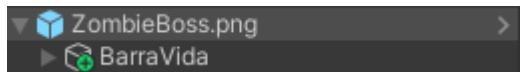
IEnumerator GetDamage()
{
    spriteRenderer.color = nuevoColor;
    yield return new WaitForSeconds(0.1f);
    spriteRenderer.color = Color.white;
}
```



ZombieBoss

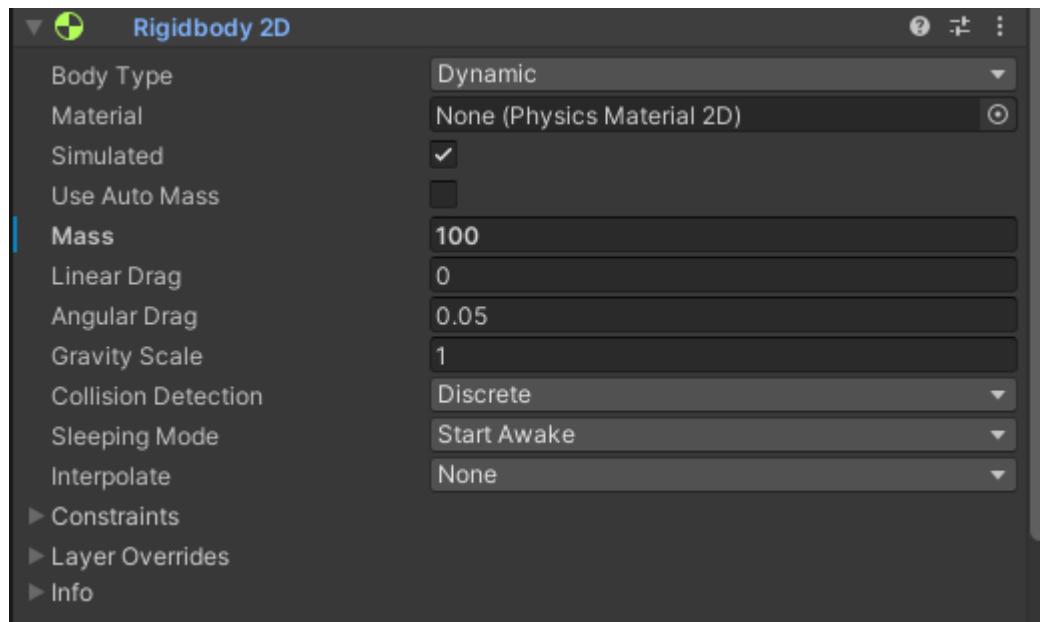


El ZombieBoss tienen el tag de ZombieBoss , una imagen asignada que cambiará con respecto a cuando reciba balas, un rigidbody para darle consistencia, su propio script “ZombieBoss” y un Polygon Collider 2D para detectar colisiones con el objeto ZombieBoss, este objeto tiene una particularidad y es que dentro de su estructura posee un objeto BarraVida que se documentará después del ZombieBoss.

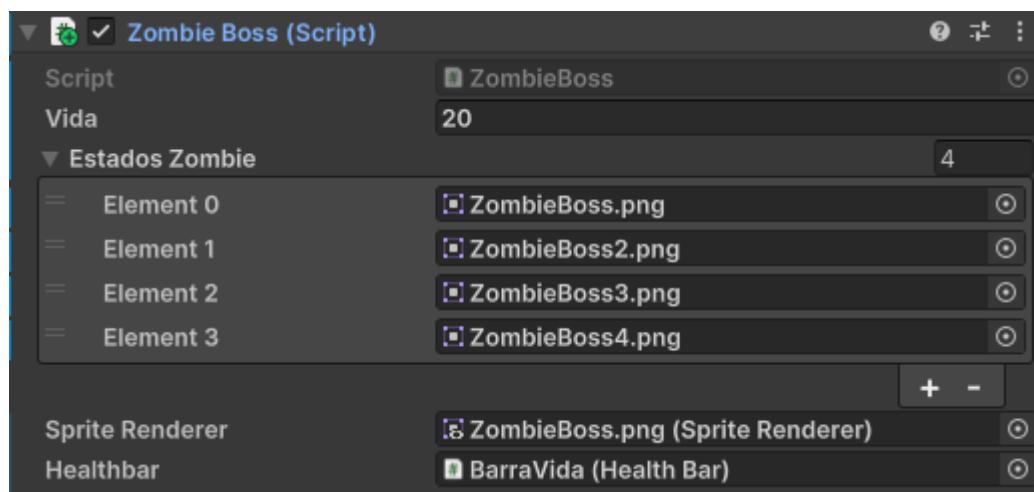




RigidBody



Script



```
public class ZombieBoss : MonoBehaviour
{
    public int vida = 20; // Vida inicial del zombie
    public Sprite[] estadosZombie; // Array que contendrá las imágenes de los diferentes estados del zombie
    public SpriteRenderer spriteRenderer; // Referencia al componente SpriteRenderer del zombie
    private Transform player; // Referencia al transform del jugador principal
    private float velocidadMovimiento; // Velocidad de movimiento del zombie
```



```

private bool esVisible = false; // Indica si el zombie es visible en la cámara
[SerializeField] private HealthBar healthbar;
Color nuevoColor = new Color(1f, 0.725f, 0.725f);

void Start()
{
    // Buscar el GameObject del jugador principal por su tag y obtener su transform
    player = GameObject.FindGameObjectWithTag("Player").transform;

    // La velocidad inicial del zombie es el doble de la velocidad normal
    velocidadMovimiento = 3f;

    healthbar.UpdateHealthbar(20,vida);
}

void Update()
{
    transform.rotation = Quaternion.Euler(transform.rotation.eulerAngles.x,
    transform.rotation.eulerAngles.y, 0f);

    // Si el jugador existe, mover el zombie hacia la posición del jugador
    if (player != null && esVisible)
    {
        // Calcular la dirección hacia el jugador
        Vector3 direccion = (player.position - transform.position).normalized;

        // Calcular la nueva posición hacia la cual moverse
        Vector3 nuevaPosicion = transform.position + direccion * velocidadMovimiento *
        Time.deltaTime;

        // Mover el zombie hacia la nueva posición
        transform.position = nuevaPosicion;
}

```



```
// Visualización en modo espejo
if (direccion.x < 0) // Si se mueve hacia la derecha
{
    spriteRenderer.flipX = false; // No voltear la imagen
}
else if (direccion.x > 0) // Si se mueve hacia la izquierda
{
    spriteRenderer.flipX = true; // Voltear la imagen horizontalmente
}
}

void OnBecameVisible()
{
    // Marcar que el zombie es visible en la cámara
    esVisible = true;
}

void OnCollisionEnter2D(Collision2D collision)
{
    // Verificar si la colisión es con una bala
    if (collision.gameObject.CompareTag("Bala"))
    {
        // Destruir la bala
        Destroy(collision.gameObject);

        // Reducir la vida del zombie
        vida--;
        healthbar.UpdateHealthbar(20,vida);
        // Cambiar la imagen del zombie según la vida restante
        if (vida >= 15 && vida < 20)
        {
            StartCoroutine(GetDamage());
        }
    }
}
```



```

        spriteRenderer.sprite = estadosZombie[0]; // Restamos 1 porque los arrays
comienzan en 0
    }
else if(vida >= 10 && vida < 15)
{
    StartCoroutine(GetDamage());
    spriteRenderer.sprite = estadosZombie[1]; // Restamos 1 porque los arrays
comienzan en 0
}
else if(vida >= 5 && vida < 10)
{
    StartCoroutine(GetDamage());
    spriteRenderer.sprite = estadosZombie[2]; // Restamos 1 porque los arrays
comienzan en 0
}else{
    StartCoroutine(GetDamage());
    spriteRenderer.sprite = estadosZombie[3]; // Restamos 1 porque los arrays
comienzan en 0
}

// Actualizar la velocidad del zombie
if(vida > 0)
{
}

else
{
    // Destruir el objeto del zombie
    Destroy(gameObject);
}

}

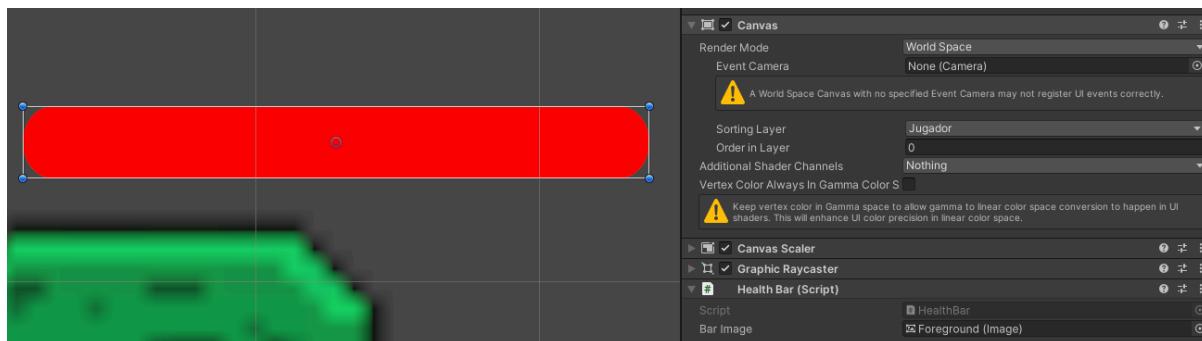
IEnumerator GetDamage()

```



```
{
    spriteRenderer.color = nuevoColor;
    yield return new WaitForSeconds(0.1f);
    spriteRenderer.color = Color.white;
}
}
```

BarraVida del ZombieBoss



La Barra de vida se basa en un canvas para representarla y un script llamado “HealthBar” para realizar la actualización de la vida cada vez que el objeto ZombieBoss recibe daño.

Script

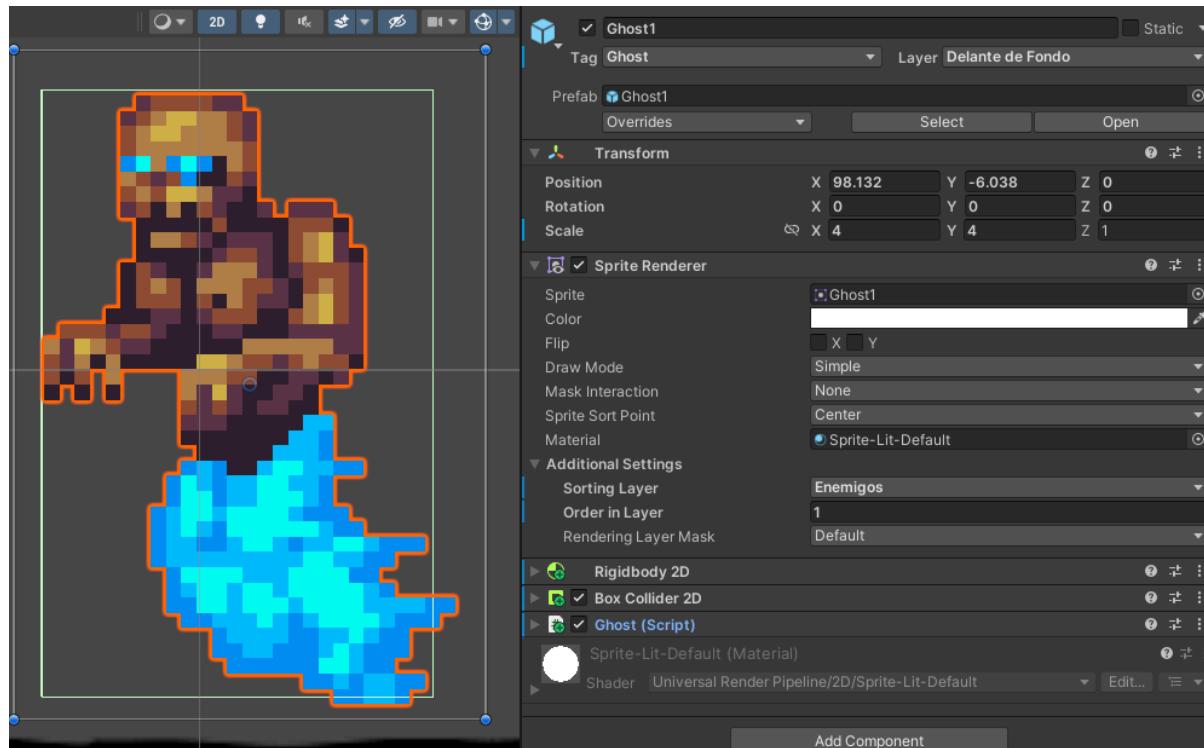
```
public class HealthBar : MonoBehaviour
{
    [SerializeField] private Image barImage;

    public void UpdateHealthbar(float maxHealth, float health)
    {
        barImage.fillAmount = health / maxHealth;
    }
}
```



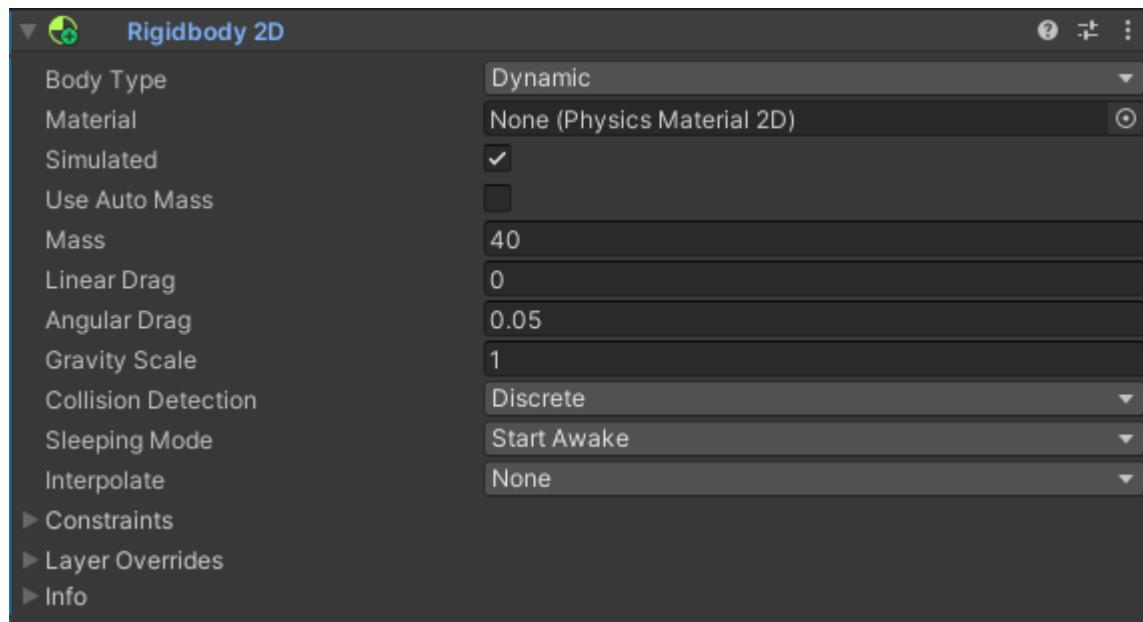


Ghost



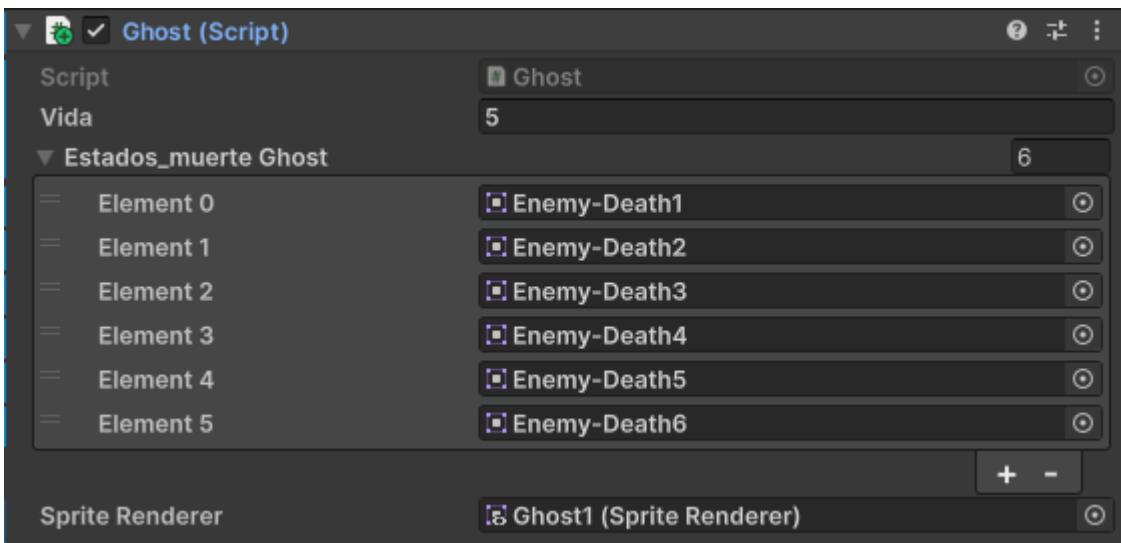
Los Ghost tienen el tag de Ghost, un rigidbody para darle consistencia, su propio script “Ghost” que posteriormente mostraré y un Box Collider 2D para detectar colisiones con el objeto Ghost , este último como se muestra en la imagen es cuadrado alrededor del objeto.

RigidBody





Script



```
public class Ghost : MonoBehaviour
{
    public int vida = 5; // Vida inicial del ghost
    public Sprite[] estados_muerteGhost; // Array que contendrá las imágenes de los diferentes estados del ghost
    public SpriteRenderer spriteRenderer; // Referencia al componente SpriteRenderer del ghost
    private Transform player; // Referencia al transform del jugador principal
    private float velocidadMovimiento; // Velocidad de movimiento del ghost
    private bool esVisible = false; // Indica si el ghost es visible en la cámara
    Color nuevoColor = new Color(1f, 0.725f, 0.725f);

    void Start()
    {
        // Buscar el GameObject del jugador principal por su tag y obtener su transform
        player = GameObject.FindGameObjectWithTag("Player").transform;

        // La velocidad inicial del ghost es el doble de la velocidad normal
        velocidadMovimiento = 2f;
    }

    void Update()
```



```
{
    transform.rotation = Quaternion.Euler(transform.rotation.eulerAngles.x,
    transform.rotation.eulerAngles.y, 0f);

    // Si el jugador existe, mover el ghost hacia la posición del jugador
    if (player != null && esVisible)
    {
        // Calcular la dirección hacia el jugador
        Vector3 direccion = (player.position - transform.position).normalized;

        // Calcular la nueva posición hacia la cual moverse
        Vector3 nuevaPosicion = transform.position + direccion * velocidadMovimiento *
        Time.deltaTime;

        // Mover el ghost hacia la nueva posición
        transform.position = nuevaPosicion;

        // Visualización en modo espejo
        if (direccion.x < 0) // Si se mueve hacia la derecha
        {
            spriteRenderer.flipX = false; // No voltear la imagen
        }
        else if (direccion.x > 0) // Si se mueve hacia la izquierda
        {
            spriteRenderer.flipX = true; // Voltear la imagen horizontalmente
        }
    }

    void OnBecameVisible()
    {
        // Marcar que el ghost es visible en la cámara
        esVisible = true;
    }
}
```



```
}
```

```
void OnCollisionEnter2D(Collision2D collision)
{
    // Verificar si la colisión es con una bala
    if (collision.gameObject.CompareTag("Bala"))
    {
        // Destruir la bala
        Destroy(collision.gameObject);

        // Reducir la vida del ghost
        vida--;

        // Actualizar la velocidad del ghost
        if (vida > 0)
        {

            StartCoroutine(GetDamage());

            velocidadMovimiento /= 1.1f;
        }
        else
        {
            StartCoroutine(Explosion(0));
            // Destruir el objeto del ghost
            //Destroy(gameObject);
        }
    }
}

IEnumerator Explosion(int index)
{
    // Desactivar el collider del Ghost antes de destruirlo
    GetComponent<Collider2D>().enabled = false;
```



```
// Pausar la física del Ghost durante la explosión
Rigidbody2D ghostRigidbody = GetComponent<Rigidbody2D>();
if (ghostRigidbody != null)
{
    ghostRigidbody.simulated = false;
}

// Verificar si el índice está dentro de los límites del array
if (index >= 0 && index < estados_muerteGhost.Length)
{
    // Cambiar el sprite del Ghost al índice actual
    spriteRenderer.sprite = estados_muerteGhost[index];

    // Esperar un tiempo antes de mostrar el siguiente sprite
    yield return new WaitForSeconds(0.1f); // Ajusta el tiempo según lo deseas

    // Llamar a la función de explosión nuevamente con el siguiente índice
    StartCoroutine(Explosion(index + 1));
}

else
{
    // Esperar un tiempo adicional después de la explosión completa
    yield return new WaitForSeconds(0.1f); // Ajusta el tiempo según lo deseas

    // Destruir el objeto del Ghost después de que termine la animación
    Destroy(gameObject);
}

}

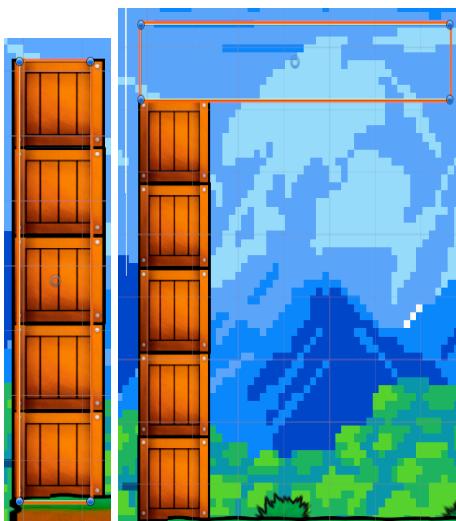
IEnumerator GetDamage()
{
    spriteRenderer.color = nuevoColor;
    yield return new WaitForSeconds(0.1f);
}
```



```
    spriteRenderer.color = Color.white;  
}  
}
```

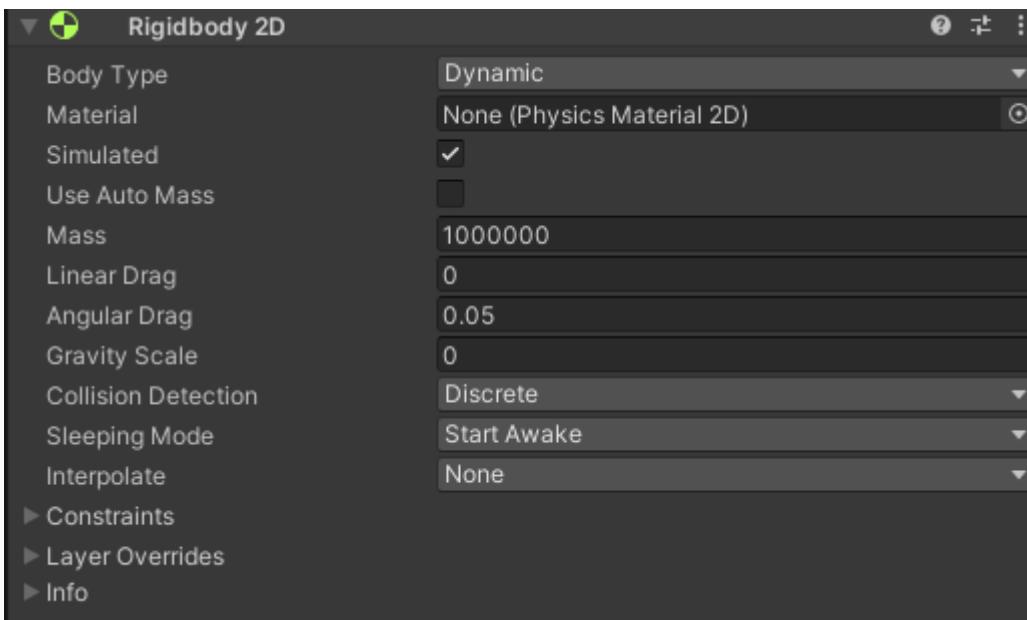


Paredes de limitación



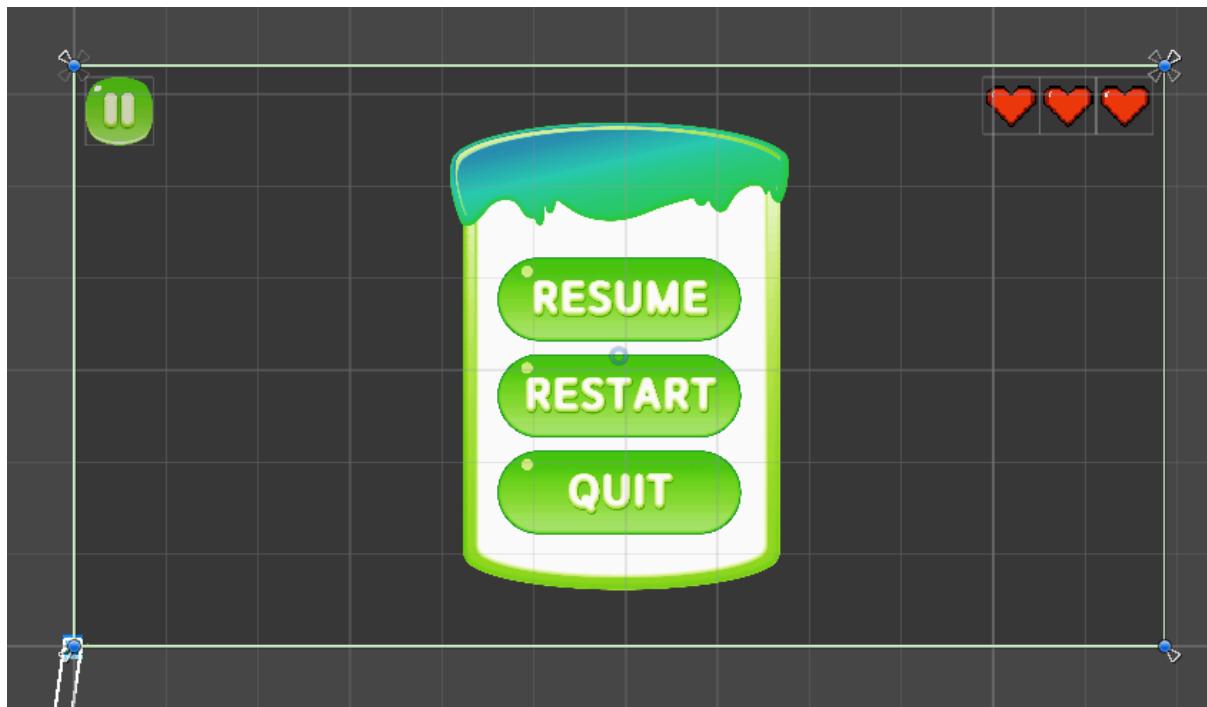
En estas imágenes se visualizan las paredes de limitación para que el usuario no pueda salirse del mapa. Dichas paredes poseen un RigidBody

Rigidbody



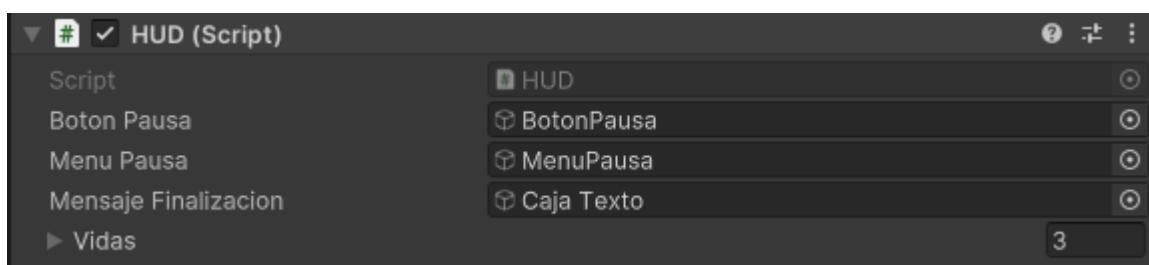


HUD



Aquí se muestra el HUD, con el menú, la vida y el botón de pausa.

Script



```
public class HUD : MonoBehaviour
{
    // Start is called before the first frame update
    [SerializeField] private GameObject botonPausa;
    [SerializeField] private GameObject menuPausa;
    [SerializeField] private GameObject mensajeFinalizacion;
    private bool juegoPausado = false;
    public GameObject[] vidas;

    public void DesactivarVida(int indice) {
```



```

        vidas[indice].SetActive(false);
    }

public void ActivarVida(int indice) {
    vidas[indice].SetActive(true);
}

private void Update(){
    if (Input.GetKeyDown(KeyCode.Escape))
    {
        if (juegoPausado)
        {
            Reanudar();
        }
        else
        {
            Pausa();
        }
    }
}

public void Pausa(){
    juegoPausado = true;
    Time.timeScale = 0f;
    botonPausa.SetActive(false);
    menuPausa.SetActive(true);
}

public void Reanudar(){
    juegoPausado = false;
    Time.timeScale = 1f;
    botonPausa.SetActive(true);
    menuPausa.SetActive(false);
}

```



```
public void Reiniciar(){
    Time.timeScale = 1f;
    SceneManager.LoadScene(SceneManager.GetActiveScene().name);
}

public void Cerrar(){
    Debug.Log("Cerrando juego");
    Application.Quit();
}

public void MostrarMensajeFinalizacion()
{
    juegoPausado = true;
    Time.timeScale = 0f;
    menuPausa.SetActive(true);
    mensajeFinalizacion.SetActive(true); // Mostrar el mensaje de finalización
    botonPausa.SetActive(false);
}
```



Conclusión

El desarrollo del proyecto ha sido divertido para mí, sobre todo el apartado de Unity ya que permite una variedad increíble para desarrollar videojuegos. Mi recomendación personal es la siguiente:

Si una persona con poca experiencia o nada en programación quiere intentar realizar un videojuego su opción más viable es Construct, permite facilidad para el desarrollo y además creo que es una herramienta que puede ayudar al aprendizaje de programación, aunque en este caso Construct 2 no permite editar código, el hecho de poder realizar las acciones por bloques, favorecen visualmente un aprendizaje rápido de los conceptos básicos de programación orientada a videojuegos.

En cambio si es una persona con experiencia sin duda alguna recomiendo Unity, es cierto que puede llegar a ser complicado dependiendo del nivel que posea el usuario o el conocimiento que tenga sobre videojuegos , pero permite tantas posibilidades para el desarrollo del mismo, destacar por ejemplo las físicas, en construct no se pueden manejar apenas, mientras que en Unity puedes manejar todas las físicas de los objetos.

Unity me ha sorprendido para bien, a pesar de ser consciente del potencial que tiene, y en el futuro estoy seguro de que realizaré más proyectos en esta plataforma.
 Construct creo que es bastante buena plataforma también para un nivel inicial o básico, puede ser una gran herramienta. Es cierto que también permite hacer grandes juegos, pero para ese tipo de proyectos yo elegiría siempre Unity.

Ha sido un proyecto bonito de realizar y que me ha motivado a investigar más en el futuro con estas plataformas y otras parecidas como Unreal Engine 5.



Glosario

Scirra Ltd: Startup londinense que lanza el creador de juego HTML5 (Construct)

Instancias: En el caso de Construct una instancia se podría considerar una copia, es decir, en un juego donde haya enemigos iguales que aparezcan varias veces, cada enemigo será una instancia.

Eventos: Consiste en una condición o conjunto de condiciones que se deben cumplir y una acción o conjunto de acciones que se ejecutan si dichas condiciones se cumplen.

Tizen: Es un sistema operativo móvil basado en GNU/Linux desarrollado por Samsung Electronics.

Kongregate: Es un portal de juegos web y editor de videojuegos estadounidense.

NW.js: Es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript, asíncrono, con E/S de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.

Cordova: Es un popular entorno de desarrollo de aplicaciones móviles, originalmente creado por Nitobi.

UWP (*Plataforma universal de Windows*): Es una API o una plataforma común de aplicaciones presentes en todos los dispositivos que cuentan con Windows 10, 11 y sus variantes.

Unity Technologies: Es una compañía americana de desarrollo de software para la creación de videojuegos. Conocida por crear Unity.

Blender: Es un programa informático multiplataforma, dedicado especialmente al modelado, iluminación, renderizado, la animación y creación de gráficos tridimensionales.

3ds Max: Autodesk 3ds Max es un programa de creación de gráficos y animación 3D desarrollado por Autodesk.

Maya: Autodesk Maya es un programa informático dedicado al desarrollo de gráficos 3D por ordenador, efectos especiales, animación y dibujo.



Softimage: Empleado para la creación de animaciones por ordenador en nuevas películas, en comerciales y videojuegos.

Zbrush: Es un software de modelado 3d, escultura y pintura digital que constituye un nuevo paradigma dentro del ámbito de la creación de imágenes de síntesis gracias al original planteamiento de su proceso creativo.

Cinema 4D: Es un software de creación de gráficos y animación 3D.

Adobe Photoshop: Es un editor de fotografías desarrollado por Adobe Systems Incorporated. Usado principalmente para el retoque de fotografías y gráficos.

Adobe Fireworks: Es un editor de gráficos vectoriales y mapas de bits.

OpenGL: Es una especificación estándar que define una API multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D.

Direct3D: Es parte de DirectX siendo propiedad de Microsoft. Consiste en una interfaz de programación de aplicaciones para gráficos 3D.

OpenGL ES: Es una variante simplificada de la API gráfica OpenGL diseñada para dispositivos integrados tales como teléfonos móviles, PDAs y consolas de videojuegos.

Mapeado de relieve (Relief Mapping): Es una técnica utilizada en gráficos por ordenador para simular la apariencia tridimensional de una superficie sin aumentar la complejidad geométrica real del modelo.

Mapeado de reflejos (Reflection Mapping): Es una técnica que simula la reflexión de objetos y entornos de superficies reflectantes. Es comúnmente utilizada para crear superficies metálicas, espejos y agua reflectante.

Mapeado por paralelo (Parallax Mapping): Es una técnica utilizada para simular la percepción de profundidad en texturas bidimensionales.

Mapas de sombras (Shadow Mapping): Es una técnica utilizada para simular la interacción de la luz y las sombras en un entorno tridimensional. Este mapa se utiliza para calcular y proyectar sombras sobre los objetos en la escena durante el proceso de renderizado.



Render a textura (Render to texture): Es una técnica que consiste en renderizar la salida gráfica de una escena o de un objeto en una textura en lugar de mostrarla directamente en la pantalla. Esto es útil para crear efectos de espejos, cámaras de seguridad, cámaras en pantalla, entre otros tantos efectos visuales.

Efectos de Post-procesamiento (Post-processing Effects): Son técnicas utilizadas para aplicar efectos visuales a la imagen renderizada después de que ha sido generada por la cámara en tiempo real. Dichos efectos se aplican a la imagen final antes de ser mostrada en pantalla y pueden incluir corrección de color, desenfoque, efectos de profundidad de campo, efectos de distorsión, efectos de iluminación, entre otros. Esto permite mejorar significativamente la calidad visual de un juego o una aplicación en tiempo real.

Sombreadores (Shaders): son programas informáticos utilizados en gráficos por computadora para definir la apariencia visual de objetos y superficies en una escena tridimensional. Especifican cómo se calcula el color, la textura, la iluminación y otros aspectos visuales de cada píxel en una pantalla. Pueden ser utilizados para crear una amplia variedad de efectos visuales, desde texturas simples hasta efectos avanzados de iluminación, sombras, distorsiones y materiales altamente realistas.

HLSL (High-Level Shading Language): Es un lenguaje de programación de alto nivel utilizado para escribir shaders en el contexto gráfico por computadora y desarrollo de juegos. Los shaders escritos en HLSL describen cómo se calcula el color, la iluminación, las sombras y otros aspectos visuales de los objetos en una escena tridimensional.

Subshaders : Son bloques de código dentro de un mismo shader/sombreador, que contienen diferentes versiones del mismo con diferentes niveles de detalle o complejidad. Por ejemplo, un subshader puede contener una versión básica del sombreador para hardware menos potente y otra versión más detallada para hardware de gama alta.

Passes : Son secciones dentro de un subshader que especifican cómo se procesa un objeto durante el proceso de renderizado. Cada pass define una etapa de procesamiento, como la aplicación de iluminación, sombras, efectos de post-procesamiento, entre otros. Un subshader puede contener múltiples passes para aplicar diferentes efectos visuales o técnicas de renderizado a un objeto.

Mono : Es un entorno de ejecución de software libre y de código abierto que permite ejecutar aplicaciones desarrolladas en el lenguaje de programación C#.



MonoDevelop : Es un entorno de desarrollo integrado (IDE) de código abierto diseñado específicamente para el desarrollo de aplicaciones utilizando el lenguaje C# y otros lenguajes compatibles con el marco de trabajo .NET. MonoDevelop ofrece características como edición de código, depuración, compilación, gestión de proyectos y control de versiones.

Theora : Es un formato de compresión de vídeo de código abierto, diseñado para proporcionar una alternativa de código abierto al estándar de compresión de vídeo propietario H.264. Utiliza técnicas de compresión con pérdida para reducir el tamaño de los archivos de vídeo mientras se mantiene una calidad de imagen razonable.

Retargeting : Es el proceso de ajustar una aplicación o un proyecto para que funcione en una plataforma de destino diferente. Esto puede implicar modificar el código fuente, recompilar el software con configuraciones específicas para la plataforma de destino, y realizar pruebas adicionales para asegurar la compatibilidad y el rendimiento óptimo en la nueva plataforma.

Biblioteca FMOD : Es una biblioteca de software desarrollada por Firelight Technologies que proporciona una interfaz para la reproducción y gestión de audio en tiempo real en aplicaciones de videojuegos y multimedia.

Billboarding : Es una técnica utilizada en gráficos por computadora para simular la rotación de objetos planos (generalmente imágenes o texturas) para que siempre estén orientados hacia la cámara del observador. Esto crea la ilusión de que el objeto plano siempre está mirando directamente hacia el espectador. Es particularmente utilizado para representar objetos como árboles, hierba, letreros y partículas en juegos y aplicaciones interactivas, lo que ayuda a mejorar la sensación de profundidad y realismo a la escena.