VIDEOJUEGOS HECHOS CON SOFTWARE LIBRE

Oskar Huang

Tutorizado por:

Jesús Aguilera Sánchez

29 - 10 - 2024

1. INTRODUCCIÓN

En los últimos años, los videojuegos han sido un objeto de contínuo desarrollo, hasta el punto de crear videojuegos hiperrealistas. Por eso, en estos últimos años, el uso de software libre para la creación de videojuegos se ha convertido en una opción más practicable para los desarrolladores independientes y sin buena situación económica. Ya que dichos motores y herramientas de desarrollo, presentan una mayor accesibilidad y flexibilidad.

La principal razón que me motivó a escoger esta presente investigación es que desde pequeño siempre tuve una gran pasión por los videojuegos, he jugado y probado muchos tipos de videojuegos, desde juegos *Shooters* hasta juegos *Family Friendly* y espero que en este trabajo de investigación me permita obtener conocimientos necesarios para desarrollar un juego completo.

La hipótesis que deseo comprobar es **si el uso de herramientas de software libre facilita la creación de videojuegos para los desarrolladores independientes, sin costos y con una buena calidad**. Sin embargo, antes de decidir, es necesario haber investigado los motores gráficos más conocidos en el mercado y elegir un motor gráfico para comenzar con el desarrollo del videojuego.

En este trabajo de investigación, los objetivos más generales son principalmente la adquisición de competencias personales como la autonomía, el autoaprendizaje, la gestión del tiempo, el pensamiento crítico, la resolución de problemas, la comunicación escrita y oral, así como la responsabilidad y el compromiso. Sin embargo, el objetivo principal planteado en el presente trabajo es llevar a cabo el desarrollo de un videojuego en 2D a partir de las herramientas de software libre. Dentro de este videojuego, tengo planteado unos objetivos específicos, como lo son terminar de desarrollar el videojuego y meter mecanismos que mejoren la jugabilidad del mismo videojuego.

La metodología aplicada en este trabajo es la metodología del autoaprendizaje y la experimentación práctica. En este proceso, se irá adquiriendo conocimientos y habilidades de computación de forma autodidacta, aprovechando los recursos audiovisuales y documentos de internet. A la hora de desarrollar el videojuego, seguramente no se encuentren recursos actuales, por ello, el proceso de desarrollo a la hora de programar código será de prueba y error.

A continuación, se comienza con el marco teórico, este consta de investigar una muestra de todos los motores gráficos existentes en el mercado, después de haberlos explicado por encima, se elige un motor gráfico y se comienza con el desarrollo del videojuego.

Finalmente, se desarrolla el marco práctico, este consta del desarrollo del videojuego en sí, el cual tendrá dos partes: la primera destinada a los funcionamientos básicos del videojuego, y la segunda orientada a los funcionamientos avanzados del videojuego. Pero, antes de comenzar con el desarrollo del videojuego, se explica cómo funciona el motor de videojuego escogido.

2. MARCO TEÓRICO

2.1. El motor gráfico

Un **motor gráfico** es una biblioteca ***software*** especializado en crear videojuegos o aplicaciones interactivas. Este proporciona herramientas para renderizar escenas en 2D o 3D, simular físicas, gestionar animaciones, integrar música y sonido, permitir a los desarrolladores añadir lógica dentro del videojuego o de la aplicación, etc…

Un **motor gráfico** simplifica la creación de videojuegos y aplicaciones interactivas, debido a que ofrece a los desarrolladores formas de incluir partes en sus juegos o aplicaciones sin tener que empezar desde cero.

2.2. Motores gráficos más conocidos

En la actualidad existen varios tipos de **motores gráficos**, cada uno con sus propias características y ventajas. ¿Y cuáles son los más conocidos?

En este apartado se explicarán las características sobre **motores gráficos** gratuitos, esto se refiere a que no hay que hacer ningún tipo de pago para poder utilizar el **motor gráfico**. Y después de haberlos revisado, se escogerá un motor gráfico para poder comenzar con el desarrollo del videojuego.

 **Unity**

Unity fue desarrollado por la compañía de Unity Technologies en 2005 como un motor de videojuegos para Mac, pero después cambiaron su enfoque hacia un motor gráfico más versátil. Este, es un motor de videojuego multiplataforma. Es decir, permite a los desarrolladores crear juegos o aplicaciones en diferentes plataformas, como PC, consolas y dispositivos móviles. En Unity se requiere crearse una cuenta, poseer una licencia válida y aceptar una serie de términos y condiciones que van a ir cambiando a lo largo del tiempo.

Unity ofrece una interfaz fácil de usar con herramientas para el desarrollo de videojuegos 2D y 3D, con soporte para ***Scripting*** en **C#** y en **UnityScript**. Además,

ofrece una gran cantidad de documentación, recursos, clases online gratis y una gran comunidad de la cual te ayudan con cualquier duda que tengas. Aunque Unity sea fácil de utilizar, los desarrolladores deben aceptar la política de licencias en *Unity*. En versiones anteriores, si no se pagaba una membresía, aparecía su marca de agua al inicio del videojuego o aplicación.

Hay una amplia gama de videojuegos móviles, de PC y de consolas hechos en Unity, estos pueden ser en 2D, 3D, realidad virtual y realidad aumentada. También se puede utilizar para crear animaciones en 2D y en 3D, como lo es “Baymax Dreams”, producida por Disney en conjunto a Unity.

Godot

Godot fue desarrollado en el año 2001 por un estudio de juegos argentinos, el motor fue lanzado como un motor de código abierto en el año 2014. Desde entonces, la propia herramienta pertenece a la comunidad de Godot y se puede modificar el motor a tu propio antojo. Asimismo, en Godot no hace falta ningún tipo de licenciamiento. Es decir, no hay ningún tipo de contrato establecido entre la empresa y la persona.

Godot, siendo un motor de software libre, es bueno para desarrollar videojuegos en 2D y 3D. Además, Godot, tiene cuatro lenguajes oficiales, como lo es **GDScript**, **C#**, **C + +** y **C**. GDScript, es la mejor opción al tratarse de Godot, ya que es un lenguaje creado exclusivamente para Godot, este está basado en Python. Es decir, si ya has aprendido sobre Python, no te costará nada aprender sobre el código de programación de Godot. Asimismo, es fácil de aprender para los principiantes y es cómodo para los desarrolladores más experimentados.

Las ventajas que tiene Godot, es que puedes personalizar las herramientas, es un motor que permite la exportación a múltiples plataformas, es ideal para desarrolladores **Indies** que no tienen mucho presupuesto, Godot tiene una mucha documentación en foros y es un motor gráfico especializado en juegos 2D, por lo que cuenta con herramientas para hacer juegos 2D de manera sencilla y versátil.

Unreal Engine

Es un motor gráfico desarrollado por la compañía de EPIC GAMES para desarrollar videojuegos, películas animadas y experiencias interactivas. Este software soporta el lenguaje de programación **C + +**. Unreal Engine es una plataforma de código abierto y gratuita. Sin embargo, cuando el juego esté generando ciertas ganancias, Unreal Engine exige un 5% sobre las ganancias brutas si superas un millón en ingresos.

Unreal Engine, es un motor gráfico avanzado, este es conocido por su hiperrealismo visual y su avanzado sistema de renderizado en tiempo real. Este motor es una herramienta muy productiva que permite a los desarrolladores crear la lógica del videojuego con el uso de **Blueprints** sin tener que escribir. Además, de ser un motor multiplataforma, este, es utilizado por grandes producciones y videojuegos AAA.

Unreal Engine, a pesar de ser un motor potente, su aprendizaje es bastante complejo. Requiere más tiempo de aprendizaje, especialmente en el uso de **C + +** o en la gestión de su sistema de físicas avanzadas. Este motor, no es recomendable para cualquier ordenador, incluso los videojuegos más pequeños, pueden pesar mucho, por eso es recomendable tener un ordenador de gama media/alta al utilizar Unreal Engine.

Scratch

Scratch fue desarrollado y publicado en el año 2003 por MIT Media Lab. Este, es un programa informático diseñado para niños y programadores principiantes, tiene una interfaz amigable y fácil de entender, este consta de unos bloques de los cuales puedes ir construyendo poco a poco para formar un código sin la necesidad de aprender la lógica anteriormente.

Scratch es adecuado para videojuegos 2D muy simples. Además, Scratch permite a los usuarios compartir sus creaciones en la misma página web de Scratch, lo que facilita la colaboración y aprendizaje en comunidad.

Scratch, como tiene una gran comunidad, donde los usuarios comparten sus proyectos. Cuenta con muchos recursos educativos, como tutoriales y guías para principiantes, lo que facilita el aprendizaje autodidacta.

 **Game Maker**

Game Maker fue desarrollado por la compañía YoYo en el año 1999, al principio este era una herramienta para ayudar a estudiantes a crear animaciones, pero, con el paso del tiempo se ha estado mejorando y se convirtió en un motor de juego.

Game Maker tiene su propio lenguaje de programación denominado **Game Maker Language**, este se trata de un sistema de programación diseñado a partir de elementos de **JavaScript**, **C + +** y **C#**.

Game Maker tiene una funcionalidad de la cual no es necesario tener conocimientos sobre programación, debido a que se puede desarrollar el videojuego a partir de acciones, y el mismo Game Maker Studio genera los códigos.

Game Maker, es un motor gráfico especialmente adecuado para desarrollos de videojuegos 2D. Asimismo, Game Maker tiene una gran comunidad y una gran cantidad de recursos, como tutoriales, lo que facilita el desarrollo del videojuego.

Este motor gráfico, ofrece licencias gratuitas y de pago para múltiples plataformas. Es decir, la exportación del videojuego a consolas es gratis, en cambio, la exportación a dispositivos móviles y computadoras es de pago.

2.2.1. Decisión

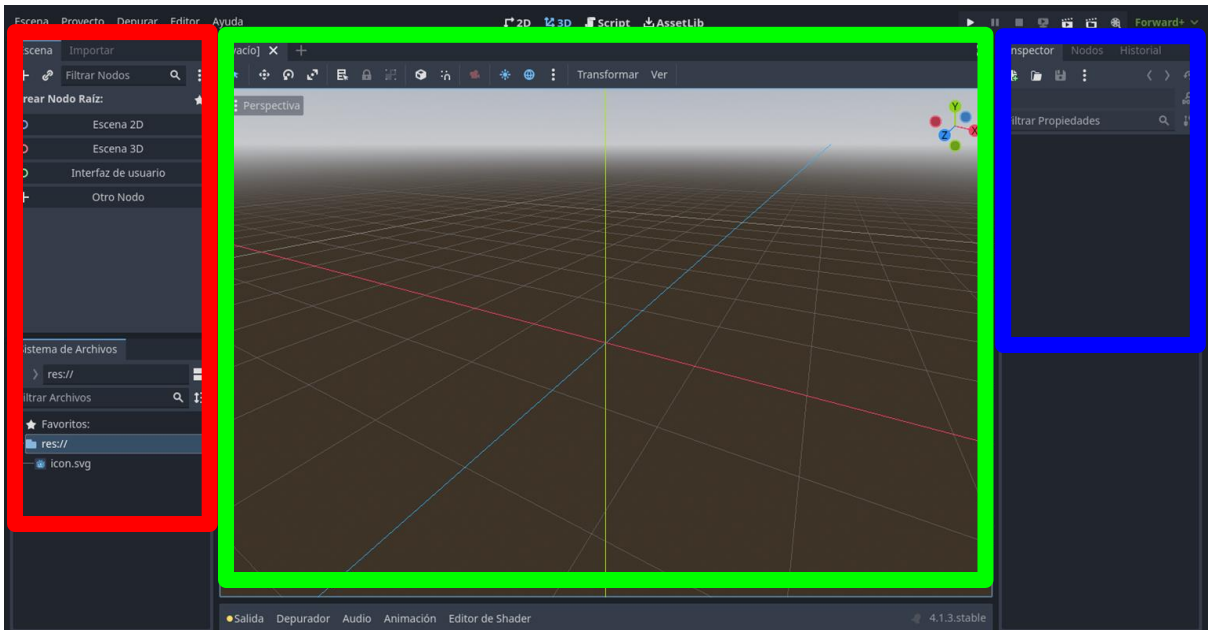
Una vez terminado de revisar una muestra de motores gráficos, decidiré que motor de juego elegir para comenzar con el desarrollo del videojuego. He optado por usar *Godot* porque, por un lado, su principal lenguaje de programación, ***GDScript***, es muy parecido a ***Python***, lenguaje que ya he aprendido un poco en la escuela y entiendo

que me será más fácil de aprender. Por otro lado, Godot es el único motor gráfico de **software libre** dentro de esta muestra. Además, el videojuego que deseo desarrollar es un videojuego plataformero en 2D, y *Godot* destaca mucho en la sencillez a la hora de crear videojuegos en 2D, esto es ventajoso para el desarrollo del videojuego mencionado.

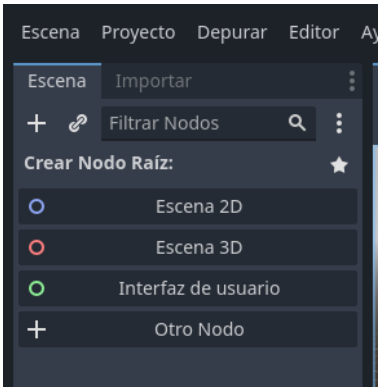
2.3. Funcionamiento de Godot

Acciones básicas en Godot

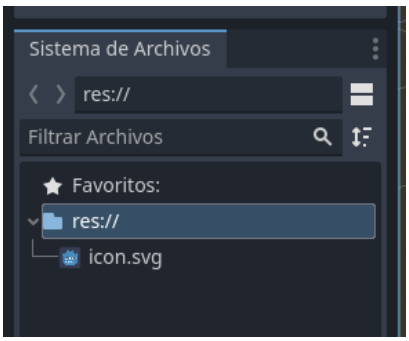
Como se puede ver en la imagen, *Godot* se divide en tres partes principales.



La parte de color rojo es donde se encuentran:



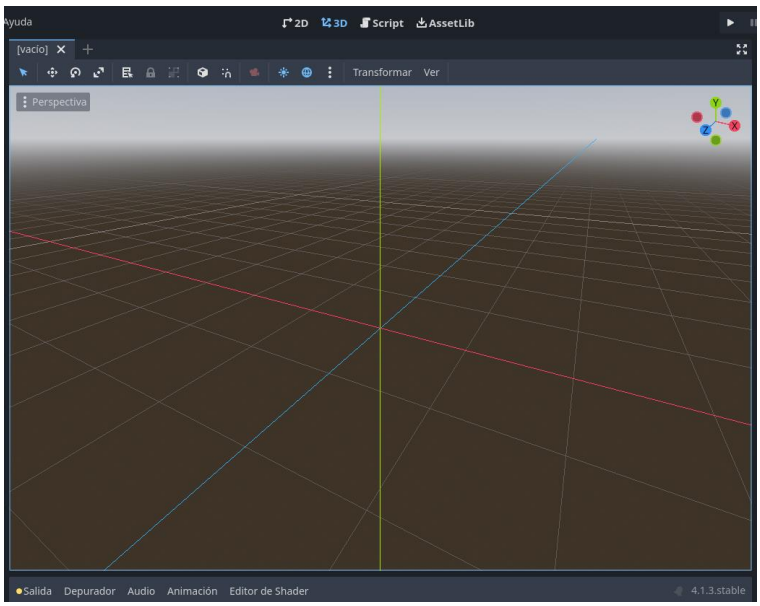
La de **creación de nodos** (en este apartado se pueden crear o cambiar de orden jerárquico los nodos).



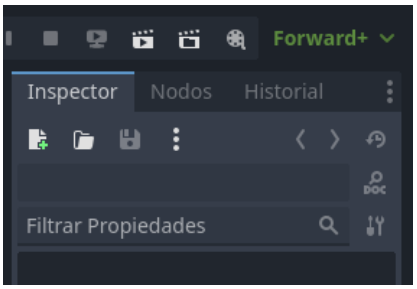
Un **explorador de archivos** (donde se guardan los archivos que se añaden desde el exterior a *Godot,* o archivos guardados y creados dentro de *Godot*).

La parte de color verde es donde se encuentra:

El **editor visual** (es la parte previa y visible del videojuego).



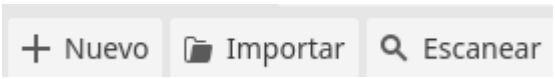
La parte de color azul es donde se encuentran:



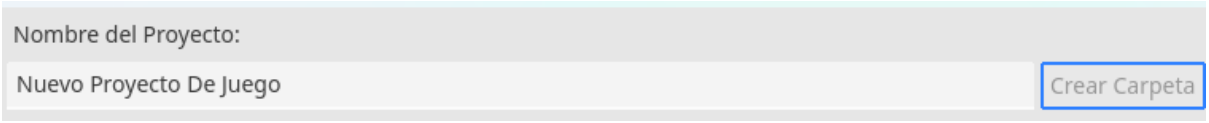
El **inspector** (en este apartado puedes ver las propiedades que tienen cada nodo y cambiar sus propiedades) y los **Nodos** (aquí se puede conectar nodos a otros, este apartado se verá más adelante).

Cómo crear un proyecto en Godot

Al entrar en Godot por primera vez, arriba a la izquierda de la pantalla verás tres apartados. Para crear un nuevo proyecto, se selecciona el apartado de nuevo.



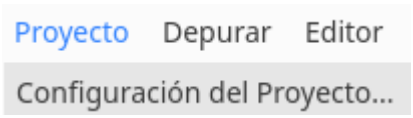
Después de haberlo seleccionado, te aparecerá la siguiente pantalla, lo único que hay que hacer es crear una nueva carpeta donde se guardan los archivos del proyecto.



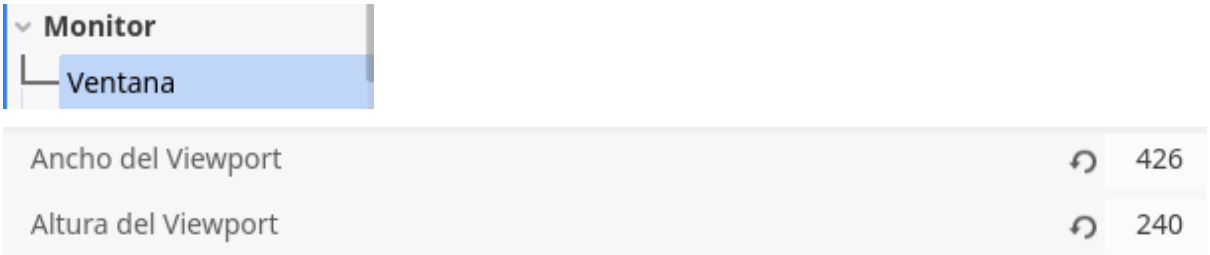
Cambio de configuraciones predeterminadas

Antes de comenzar a desarrollar el videojuego o la aplicación, es recomendable cambiar algunas configuraciones predeterminadas.

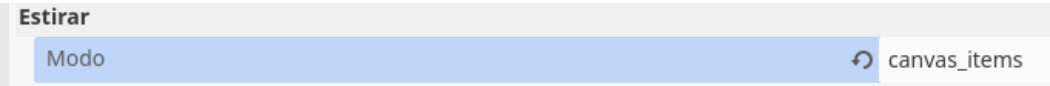
Para poder acceder al ajuste de Godot, se tiene que seleccionar el apartado de proyecto y la configuración del proyecto.

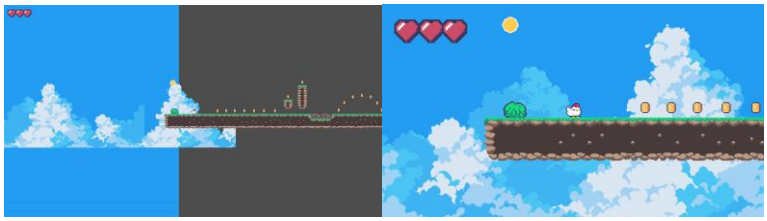


Lo primero será cambiar las dimensiones del viewport, este es el tamaño de la pantalla que tendrá el videojuego, en este caso el viewport será de 426X240. Para cambiarlo se tiene que seleccionar el apartado de ventana.



Asimismo, dentro del apartado de ventana, se cambia el modo de la ventana de disabled a canvas_items, esto hace que, al cambiar a pantalla completa, no se vea cortada.





Estructura de Godot

Godot se divide en una estructura muy clara y sencilla de entender: cada proyecto se organiza en un árbol de **escenas**, estas **escenas** pueden contener una serie de nodos que forman un árbol jerárquico de nodos.

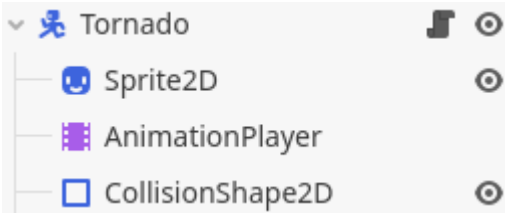
Este árbol jerárquico de nodos se divide en dos:

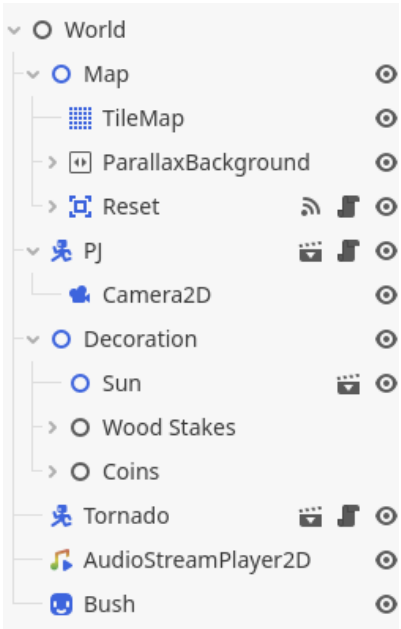
- **Nodo padre**, está en el nivel superior del árbol, contiene y gestiona a otros nodos por debajo de este.
- **Nodo hijo**, está en el nivel inferior del árbol y depende del nodo padre.

Los nodos son bloques prediseñados que se agrupan en las escenas para crear algo en específico o crear un personaje desde cero. Hay muchos tipos diferentes de nodos, como los que muestran una imagen, crear una animación, crear colisiones, y muchos más.

Ejemplo.

En este ejemplo, el nodo padre es un ***Character Body* 2D** que consiste en el cuerpo del personaje en el videojuego. Incluye un ***Sprite***, que muestra una imagen que define la apariencia visual del personaje. Al mismo tiempo, incluye una ***Collision Shape* 2D**, que es el responsable de la forma de colisión del personaje, un ***Animation Player*** para crear y controlar las animaciones, y por último una ***Camera* 2D** que sigue al personaje en la pantalla del juego.





Algo bastante interesante de *Godot*, es que se pueden amontonar varios nodos padres. Es decir, dentro de una escena general (un mapa), se puede añadir un nodo que contenga varios nodos de un enemigo (un nodo padre). Esta función de *Godot*, facilita la organización a la hora de desarrollar un videojuego o una aplicación.

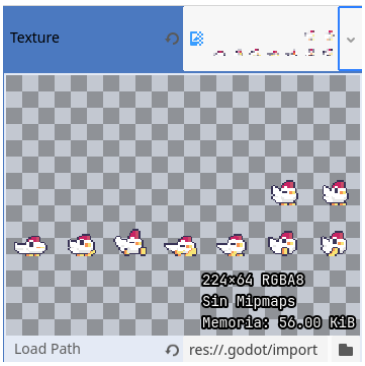
3. MARCO PRÁCTICO

3.1. Funcionamiento básico del videojuego

3.1.1. La creación del personaje

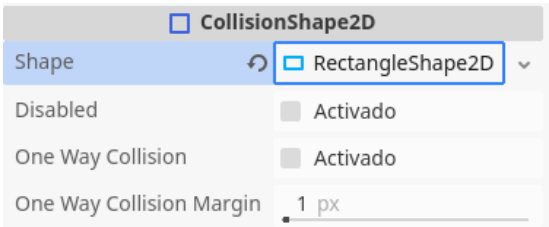
Para empezar, hay que crear una escena de la cual se denominará **PJ**, el nodo padre será el de ***CharacterBody2D***. A su vez, se debe añadir un nodo hijo ***Sprite*** que contendrá la imagen del personaje, y un nodo ***CollisionShape*** que definirá el área de colisión.

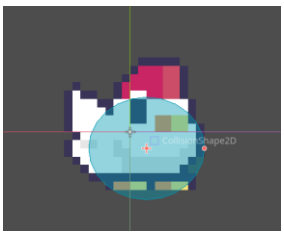
SpriteSheet utilizado para hacer el personaje con sus animaciones. Este se tiene que añadir en los archivos de ***Godot***.



Una vez terminado de agregar el ***SpriteSheet*** a los archivos de ***Godot***. Seguidamente, se tiene que añadir el ***SpriteSheet***, en el apartado de ***Texture***.

Después de incluir el ***SpriteSheet***, es necesario darle una forma a la colisión para que el personaje pueda interactuar con el entorno.



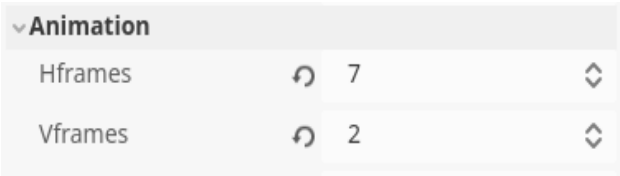


En el apartado de ***Shape*** se le debe dar forma a la colisión.

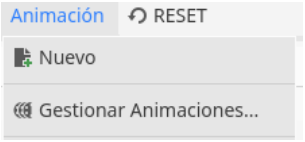
3.1.2. Movimientos básicos

Los personajes hacen uso de una animación, para darle una impresión de movimiento al momento de moverse. En lugar de simplemente verse como una imagen que no se

mueve, se debe agregar un nodo ***AnimationPlayer*** en la escena del personaje. De esta manera, ***AnimationPlayer***, hará posible el uso del ***SpriteSheet*** para gestionar los múltiples fotogramas, los cuales cuando se reproducen de forma fluida, pueden usarse para realizar animaciones. Así proporcionando un mayor dinamismo al personaje cuando este se mueva.

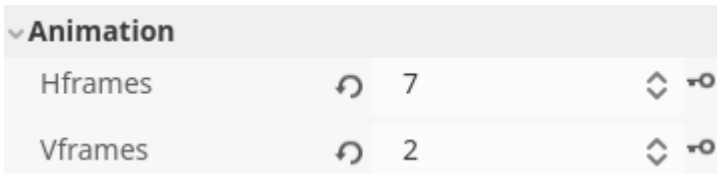


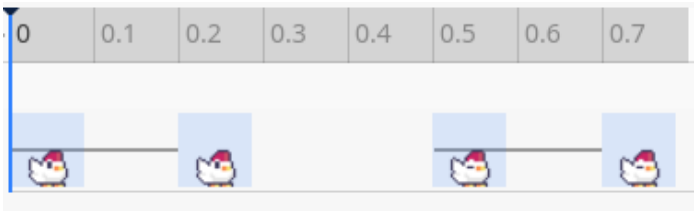
En el apartado del ***Sprite*** habrá una sección de animación, como este es un ***SpriteSheet*** se tiene que asignar 7 fotogramas en horizontal y 2 en vertical.



Para poder crear una animación, lo primero es dirigirse a ***AnimationPlayer*** y crear una nueva animación.

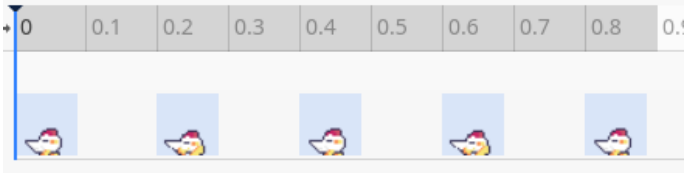
Una vez creada, se accede al nodo de ***Sprite*****,** y en la sección de animación, ***Frame***, se procederá a añadir los fotogramas de forma secuencial. Para poder añadir los frames de forma secuencial, se tiene que pulsar la llave al lado del número de frame que quieres añadir.





Animación ***Idle*** del personaje.

Una vez terminada la animación de reposo, se debe crear la animación de desplazamiento, la cual se denominará ***Walk***. Se repite el mismo proceso que con el de ***Walk***, pero con los ***Frames*** requeridos para esta animación.



Animación ***Walk*** del personaje.

En la parte inferior del nodo de la animación se encuentran tres apartados, y en el primer apartado se puede cambiar de continuo a discreto (Opcional), con el fin de proporcionar una animación más clásica.

Para permitir que el personaje avance de fotograma a fotograma. Es decir, que la animación camine de forma continua. Será necesario activar el modo ***Loop***.

Una vez terminadas las dos animaciones, se procede con la programación de las dos animaciones. Se comienza con la creación de un ***Script*** vacío en el nodo de ***Character Body 2D***.

```
1   extends CharacterBody2D
2
```

Al comienzo se definen tres variables, **velocidad**, **salto** y **dirección**, además de una constante, la **gravedad**.

```
5   var speed = 100
6   var direccion = 0.0
7   var jump = 250
8   var gravity = 7
```

A continuación, se define una función, de la cual se va ejecutando de manera repetitiva.

```
func _physics_process(delta):
```

Seguidamente, se establece que la **velocidad x** es igual al valor de la dirección multiplicado por el valor de la velocidad. Es decir, si la dirección es hacia la izquierda, el valor de la dirección será negativa y se moverá hacia la izquierda, y si la dirección es hacia la derecha, el valor de la dirección es positiva y se moverá hacia la derecha. En cambio, si el valor de la dirección es cero, el personaje quedará inmóvil.

```
15      velocity.x = direccion + speed
16
17      velocity.x = direccion * speed
```

direccion = Input.get_axis("left","right"). Esto hace que el programa detecte la dirección en la que se está moviendo el personaje.

```
18      direccion = Input.get_axis("left","right")
```

Consecutivamente, es necesario definir dos condiciones para gestionar las dos animaciones. Es decir, si el valor de la dirección es diferente de cero, la animación reproducida será la de ***Walk***. En cambio, si el valor de la dirección es igual a cero, se reproducirá ***Idle***.

```
19      if direccion != 0:
20          $AnimationPlayer.play("walk")
21      else:
22          $AnimationPlayer.play("idle")
```

Para que, al caminar de izquierda o derecha, la imagen gire hacia el lado de donde se está moviendo, se debe implementar la propiedad ***flip_h***. Para ello, se añade el ***Sprite*** utilizado, con el ***flip_h*** e igualarlo a dos condiciones. La primera es para revisar

si el valor de la dirección es negativo y la segunda determina si el personaje está en movimiento o no.

```
24    $Sprite2D.flip_h = direccion < 0 if direccion != 0 else $Sprite2D.flip_h
```

Para añadir, se debe implementar la capacidad de salto del personaje y la afectación de la gravedad hacia el personaje. Para ello, es necesario verificar si el personaje está en el suelo y si se ha presionado la tecla de salto, entonces se aplica una velocidad vertical positiva. Si el personaje no está en el suelo o no se ha presionado la tecla, el personaje no saltará. Luego, se verifica si el personaje está en el suelo o no, si no lo está se le aplica una fuerza de gravedad, así haciendo caer al personaje.

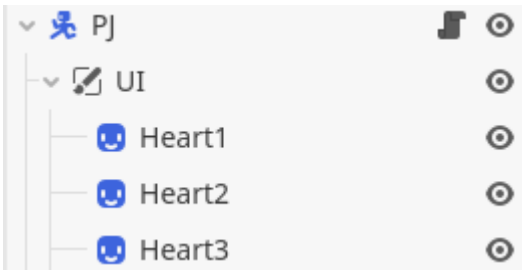
```
28    if is_on_floor() and (Input.is_action_just_pressed("ui_accept") or Input.is_action_just_pressed("up")):
29        velocity.y -= jump
30
31    if !is_on_floor():
32        velocity.y += gravity
```

Finalmente, para que el personaje pueda moverse en el entorno, se debe añadir la propiedad de ***move_and_slide***.

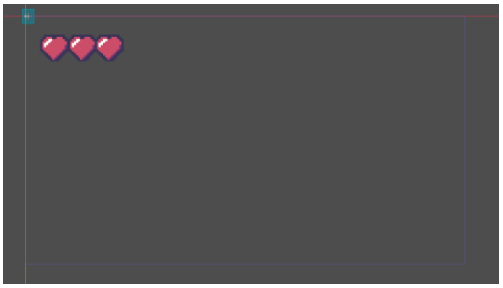
```
35    move_and_slide()
```

3.1.3. Barra de vida

En este caso, el personaje tendrá tres corazones.



Se comienza añadiendo un ***CanvasLayer*** en la escena del personaje. Seguidamente, dentro del ***CanvasLayer*** se añaden 3 ***Sprites*** de un corazón.



Para que los 3 ***Sprites*** de la barra de vida se vean correctamente en la pantalla al iniciar el juego, se tendrá que colocar los 3 sprites dentro del recuadro. Como muestra la imagen.

Para que el jugador tenga tres corazones, se tiene que añadir una nueva variable, la de ***Lifes***, y se iguala a tres. Que es el máximo de corazones que tendrá.

```
10    var lifes = 3
```

En la función de ***physics process(delta)*** se añade un argumento llamado ***handleHearts()***, al principio dará error debido a que todavía no se encuentra la definición en ***Godot***.

```
12  func _physics_process(delta):
13      handleHearts()
```

Seguidamente, se añaden dos nuevas funciones en el ***Script*** del personaje. Una función hará que la vida disminuya en uno cada vez que el personaje reciba daño, hasta tener cero corazones y se reiniciará la partida. La otra función hará que los corazones se vayan haciendo invisibles a medida que el personaje vaya recibiendo daño.

En la función de ***LoseLife***, la vida disminuye en uno, y si la vida es menor o igual a cero, se reinicia el nivel o la escena. Ahora cada vez que el personaje colisiona con un enemigo, perderá uno de vida.

```
35  func _LoseLife():
36      lifes -= 1
37      if lifes <= 0:
38          get_tree().reload_current_scene()
```

Dentro de la función de ***handleHearts***, si la vida es igual a tres, las tres imágenes están visibles, cuando la vida es igual 2, el tercer corazón ya no será visible, así sucesivamente hasta tener 0 corazones.

```
45  func handleHearts():
46      if lifes == 3:
47          $UI/Heart1.visible = true
48          $UI/Heart2.visible = true
49          $UI/Heart3.visible = true
50      if lifes == 2:
51          $UI/Heart1.visible = true
52          $UI/Heart2.visible = true
53          $UI/Heart3.visible = false
54      if lifes == 1:
55          $UI/Heart1.visible = true
56          $UI/Heart2.visible = false
57          $UI/Heart3.visible = false
58      if lifes <= 0:
59          $UI/Heart1.visible = false
60          $UI/Heart2.visible = false
61          $UI/Heart3.visible = false
```

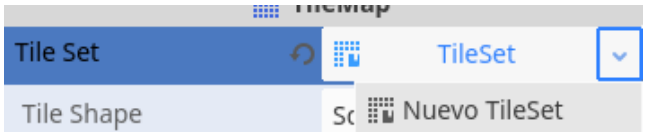
3.1.4. Base del mapa

Sprite utilizado para hacer el mapa.

El primer paso, es crear una nueva escena en el que el nodo padre es un ***Node***, del cual se denominará ***World***.

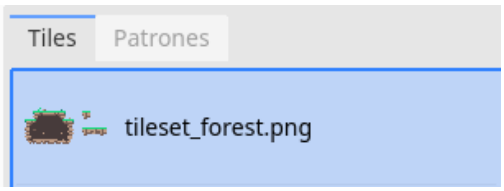
Seguidamente, añadir el personaje creado anteriormente a la nueva escena. Para lograrlo, es necesario arrastrar el archivo del personaje desde el gestor de archivos de ***Godot*** hacia la escena.

A continuación, se crea un nodo ***TileMap***, que es un componente que administra y permite la construcción del terreno o del entorno del juego utilizando mosaicos. Una vez creado el ***TileMap***, se debe seleccionar y en la sección de ***TileSet***, crear un nuevo ***TileSet***, este es un conjunto de mosaicos o ***Tiles*** que están en una sola imagen.

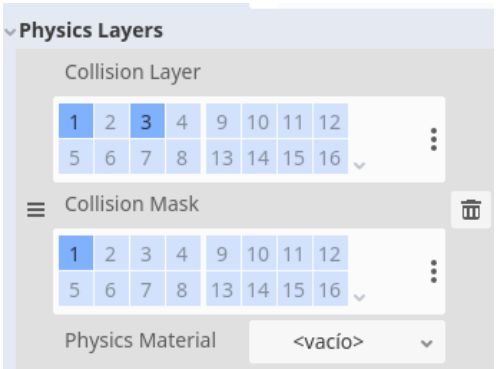


Después de haber creado el nuevo ***TileSet***, se debe seleccionar el apartado de ***TileSet*** abajo de la pantalla y añadir el ***Sprite***.

Apartado ***TileSet***.



Sprite añadido.



Para que los mosaicos sean interactuables, es decir, que el personaje pueda pisarlo o colisionar con ello, se necesita añadir un área de colisión. Para lograrlo, se debe seleccionar el nodo de ***TileMap*** y crear un nuevo ***Physic Layer***.

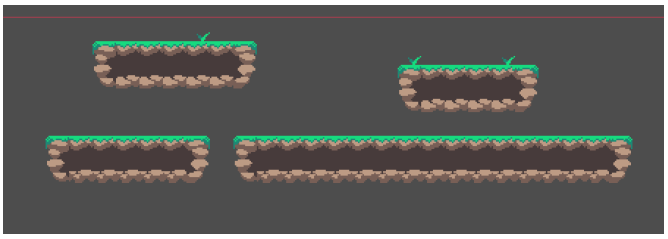
En la sección de ***TileSet***, seleccionar la herramienta para pintar y en la parte de físicas, establecer la capa de física cero, para luego definir las áreas de colisión en los ***Tiles***.

Para poder usar los mosaicos para pintar, primero se tiene que seleccionar ***TileMap*** y luego la herramienta de pintar dentro de la barra de herramientas del ***TileMap***.

Apartado ***TileMap***.

Herramienta de pintar en  el apartado de ***TileMap***.

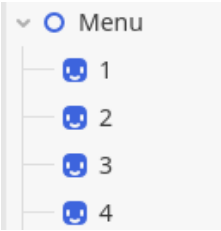
Selecciona el mosaico con el que quieras pintar.



Ejemplo de plataformas pintado con el ***TileMap***.

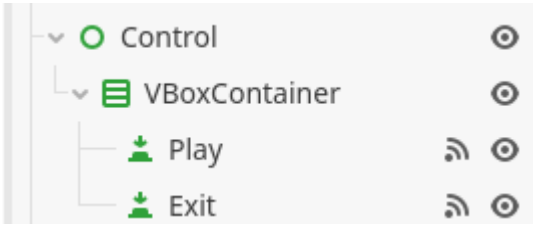
3.1.5. Pantalla de inicio

La pantalla de inicio de un videojuego debe de captar la atención de los jugadores, ya que el primer contacto que tienen al entrar en un videojuego, es la pantalla de inicio. Una pantalla de inicio atractiva, genera motivación a los jugadores para entrar al mundo y explorar.



Se comienza con una nueva escena, la cual se denomina ***Menu***, esta escena es un **Node 2D** y será el nodo padre que contendrá los nodos necesarios para hacer una pantalla de inicio. Seguidamente, para poder dar una sensación de profundidad, se tiene que añadir 4 capas de sprites, una sobre otra, así dando la sensación de profundidad.

Dentro del nodo principal, se inserta un nodo hijo ***Control***, que sirve de base para los siguientes elementos de la interfaz. Consecutivamente, se añade un nodo ***VBoxContainer***, este organiza de forma estructurada los elementos dentro de márgenes. En el interior del ***VBoxContainer***, se añaden dos botones, denominados ***Play*** y ***Exit***. El botón ***Play*** permite al jugador acceder dentro del videojuego, en cambio, el botón de ***Exit*** te expulsa de la aplicación. Para que estos botones funcionen, se requiere conectar la señal de la interacción del usuario con el botón.



Para poder configurar las señales, el primer paso es añadir un ***Script*** vacío al nodo principal.

```
1   extends Node2D
2
```

Una vez añadido el ***Script***, se selecciona el nodo de ***Play*** o ***Exit***, y en la parte superior derecha seleccionar el apartado de nodos. Dentro de nodos, se localiza la señal de ***Pressed***, y se conecta este con el nodo principal.

Señales conectadas al ***Script***.

```
4  func _on_play_pressed():
5      pass
6
7
8  func _on_exit_pressed():
9      pass
```

Una vez conectadas las dos señales al ***Script*** del nodo principal, se procede a escribir el código.

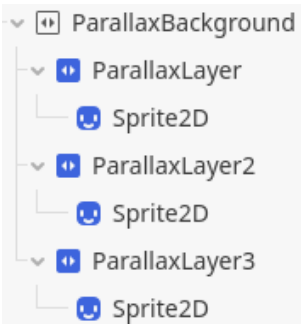
En la función de ***on_play_pressed***, se añade un ***Script*** para hacer un cambio de escena en cuanto el jugador interactúe con el botón, la escena deseada es la de ***World***. En la función de ***on_exit_pressed***, se añade un código del cual permite cerrar el juego.

```
4  func _on_play_pressed():
5      get_tree().change_scene_to_file("res://Recursos/Escenas/world.tscn")
6
7
8  func _on_exit_pressed():
9      get_tree().quit()
```

3.2. Funcionamiento avanzado del videojuego

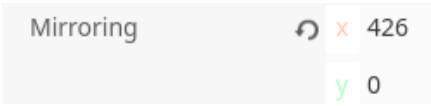
3.2.1. Fondo del mapa

Dentro de la escena de ***World***, se comienza creando un nodo hijo denominado ***ParallaxBackground***, este es una técnica en la que el fondo se mueve a diferentes velocidades para generar un efecto de profundidad. Seguidamente, se añaden varios ***ParallaxLayer***, tantos como capas de fondos tengas. Dentro de ella, se añaden los fondos, cuanto más fondos, más sensación de profundidad da el entorno del videojuego. Los ***Sprites*** más cerca de la pantalla, son los que más velocidad tienen, en cambio, cuanto más lejos de la pantalla, menos velocidad tienen.

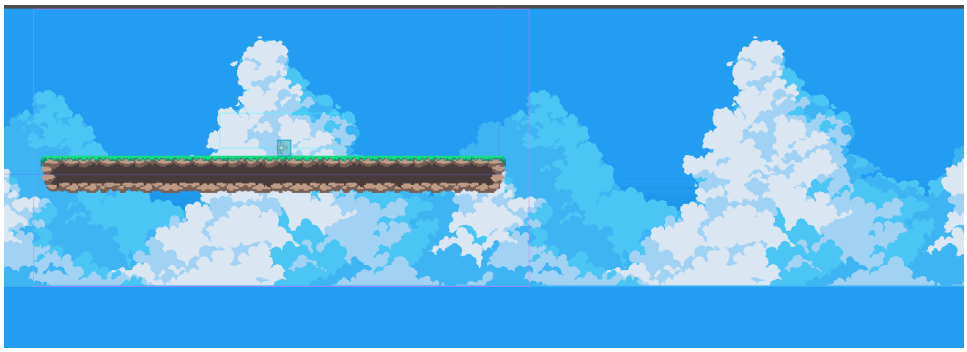


Para que los fondos se muevan a diferentes velocidades, en ***ParallaxLayer***, se selecciona el apartado de ***Motion***, y se ajusta la propiedad de ***Scale*** en eje X.

Finalmente, se tiene que seleccionar el ***ParallaxLayer*** correspondiente y buscar el apartado de ***Mirroring***. A continuación, se ajusta el eje X hasta que las dos imágenes no queden sobrepuestas una sobre la otra.

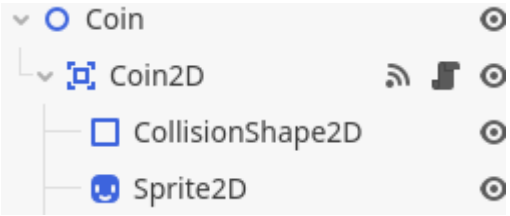


Gracias al apartado de ***Mirroring*** las dos imágenes no quedan sobrepuestas.



3.2.2. Objetos recogibles

 ***SpriteSheet*** de monedas utilizado.



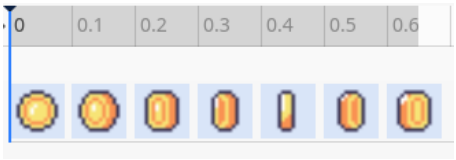
Se comienza creando una nueva escena denominada ***Coin***, del cual se añade una ***Area2D***, y dentro de ella se inserta una ***CollisionShape2D*** y un ***Sprite2D***.

Una vez terminado de asignar la textura del ***Sprite***, se tiene que asignar los fotogramas correspondientes en vertical y horizontal.

Seguidamente, se añade un ***AnimationPlayer2D***, en el cual se tendrá que crear una nueva animación para ***Coin***. Como esta moneda debe estar en constante animación,

hay que asignar la animación en modo ***Loop***. Para saber más, ir al punto 3.2.2. Movimientos básicos

Nodo ***AnimationPlayer***.

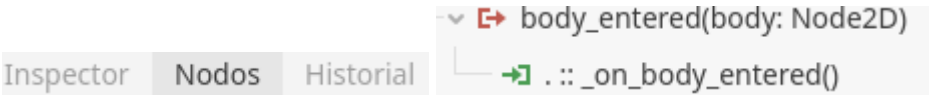


Animación de la moneda.

Lo siguiente es implementar una función para que el jugador pueda recoger las monedas. Para ello, se comienza creando un ***Script*** vacío en el nodo de ***Area2D***.

```
1   extends Area2D
2
```

Seguidamente, se conecta la señal de área 2D a sí mismo, para ello hay que ir a la esquina superior derecha, y en el apartado de nodos, seleccionar la señal de ***body_entered*** y conectarla con el ***Script*** vacío del área 2D.



Esta señal, generará automáticamente una función dentro del código, del cual se puede modificar para definir lo que sucederá cuando el jugador entre en contacto con el área, como poder recogerlo.

```
5  func _on_body_entered(body):
```

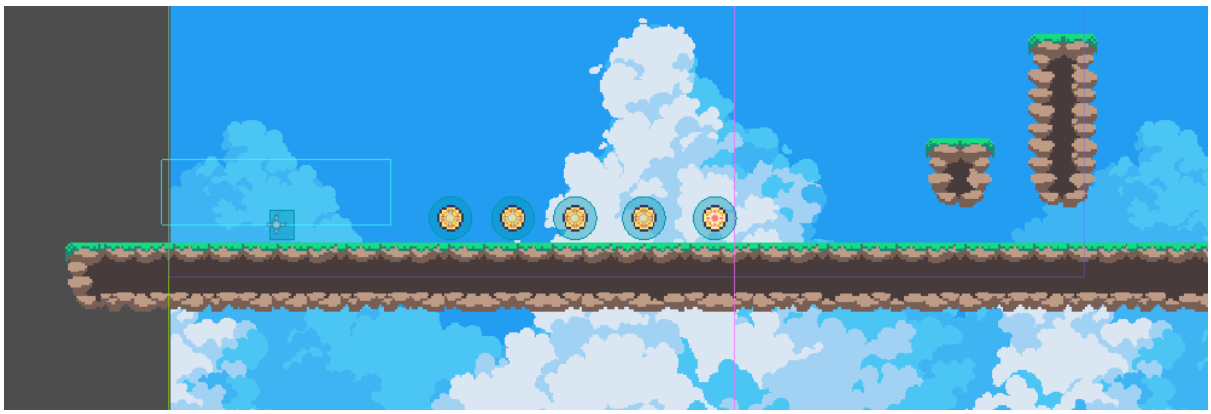
Después de haberse generado la función, se prueba escribiendo solamente el ***queue_free***, este hace que, si un cuerpo ha entrado en contacto con la moneda, esta desaparece.

```
5  func _on_body_entered(body):
6      queue_free()  # Elimina
```

Si la moneda desaparece con el contacto de la superficie del ***TileMap***, hay 2 opciones, colocar las monedas más arriba sin hacer contacto directo con la superficie o hacer que solamente el jugador pueda hacer desaparecer las monedas.

Para ello se tiene que añadir en la función un ***If***, seguidamente, se añade ***get_name*** para que el programa reconozca que el nombre del cuerpo que ha entrado en contacto con la moneda sea el del personaje, y que el personaje reconozca que el cuerpo con el que ha entrado en contacto sea el de la moneda. Además, se tiene que escribir ***queue_free*** para que desaparezca al entrar en contacto.

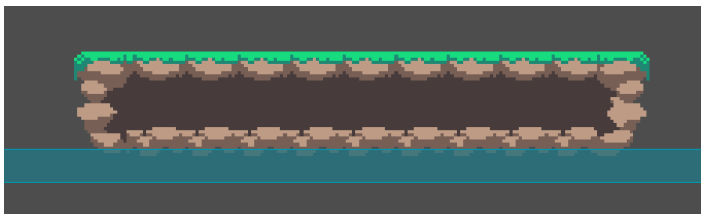
```
5  func _on_body_entered(body):
6      if body.get_name() == "PJ":
15         queue_free()
```



3.2.3. Respawn

Para hacer una zona de reseteo al caerse y que devuelva a la zona inicial al personaje, primero se tiene que añadir una ***Area2D***, con su zona de colisión.

La zona de colisión se ajusta tan grande como se requiera, pero debe de estar debajo del ***TileMap***, para que cuando el jugador al caer al vacío, la colisión del personaje interactuando con la colisión del vacío, se resetee el nivel.



Después, se crea un ***Script*** vacío en el ***Area2D***, seguidamente se selecciona el ***Area2D*** y seleccionar el apartado de nodos, para poder conectar ***on_body_entered***.

Por último, se necesita modificar la función, después de haberlo conectado, aparece la función de ***on_body_entered***.

```
1   extends Area2D
2
3
4   func _on_body_entered(body):
```

Para que el programa sepa que es el jugador el que ha caído al vacío, se añade un ***If***. Consecutivamente, hace que el área identifique el nombre que tiene el cuerpo que ha entrado en contacto y viceversa, y si los dos cuerpos se reconocen, se reinicia el nivel.

```
4   func _on_body_entered(body):
5       if body.get_name() == "PJ":
6           get_tree().reload_current_scene()
```

3.2.4. Estacas afiladas

Con el fin de que el juego se vea un poco más interesante, lo siguiente es añadir un objeto colisionable que dañe al jugador, en este caso son estacas de madera, que al colisionar con el personaje dañe al personaje y le baje un corazón.

En la escena de ***World***, se comienza creando un nuevo nodo hijo ***Area2D***, dentro del cual se inserta un ***Sprite*** y una ***CollisionShape***.

Una vez terminado de agregar los nodos, lo siguiente será añadir la siguiente función al código del personaje.

```
40 func _on_wood_stakes_body_entered(body):
```

Para ello, hay que seleccionar en la parte superior derecha, el apartado de nodos, y ***on_body_entered***. Seguidamente, se escribe un código para que el programa verifique que si el cuerpo que ha entrado en contacto con la estaca, sea el del personaje, y viceversa. Si este cuerpo es el del personaje, entonces este pierde una vida.

```
40 func _on_wood_stakes_body_entered(body):
41     if body.get_name() == "PJ":
42         body._LoseLife()
```

3.2.5. Enemigos

Para hacer un enemigo es necesario crear un nuevo escenario, el nodo padre será un ***CharacterBody2D***, el cual se denominará ***Enemy***. Dentro de la escena se debe añadir los nodos hijos básicos para que un personaje pueda realizar las acciones requeridas. En adición, se añade un ***Area2D***, dentro de esta se añade una ***CollisionShape***. La ***CollisionShape*** del personaje y la de la ***Area2D*** son diferentes,

ya que la colisión del personaje, se encarga de la interacción con el mapa para que no se caiga. En cambio, el de la ***Area2D***, se encarga de la detección del personaje.

Para que ***Enemy*** tenga movilidad, se tiene que añadir un nuevo ***Script*** vacío.

```
1    extends CharacterBody2D
```

Para comenzar, se declaran las variables de velocidad, gravedad y que ***moving left*** sea cierto.

```
4    var speed = 30
5    var gravity = 7
6    var moving_left = true
```

Dentro de la función de cuando se inicie el juego, se reproduce la animación de ***Enemy***.

```
8  func _ready():
9      $AnimationPlayer.play("Tornado")
```

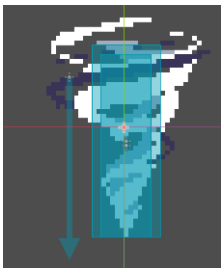
Lo siguiente es añadir una nueva función, la de ***move character***. Esta función, mueve al personaje.

Dentro de esta función, se aplica la gravedad al movimiento vertical. Además, se añade una condición que verifica si la variable ***moving left*** está activa. Esto significa que si está activa el valor de la velocidad estará en negativo y el enemigo se moverá hacia la izquierda. En cambio, si no está activa, se moverá hacia la derecha.

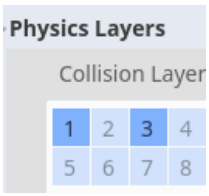
```
15 func move_character():
16     velocity.y += gravity
17     if (moving_left):
18         velocity.x = -speed
19         move_and_slide()
20     else:
21         velocity.x = speed
22         move_and_slide()
```

Una vez terminado con el código de la movilidad de ***Enemy***, es importante que, al colisionar con el personaje, el personaje pierda una vida. Para ello, se tiene que seleccionar el ***Area2D***, y conectar la señal de ***on_body_entered*** dentro del ***Script*** del enemigo. Dentro de esta señal, se añade un código para que el enemigo pueda identificar al personaje al colisionar y viceversa. Si estos se reconocen, entonces el personaje pierde una vida.

```
24 func _on_area_2d_body_entered(body):
25     if body.get_name() == "PJ":
26         body._LoseLife()
27         pass
```



Para que el enemigo detecte los bordes del mapa y cambie de sentido, se tiene que añadir un ***RayCast***, este es una flecha que detecta si está colisionando con cierto cuerpo, en este caso, el ***RayCast*** debe detectar el ***TileMap***.



En el apartado de ***RayCast***, primero se debe de seleccionar el número 3 de ***Collision Mask***. Para que pueda detectar el mapa, se tiene que seleccionar en el ***TileMap*** el número 3 en la ***Collision Layer***.

Una vez terminado, se añade en la función de proceso, que el ***Enemy*** se pueda mover y girar.

```
11 func _process(delta):
12     move_character()
13     turn()
```

Lo siguiente, es añadir una nueva función, ***Turn***, esta hace que el enemigo gire. Dentro de esta función, el código invierte la dirección de movimiento del personaje

cuando no detecta ninguna colisión, y voltea el ***Sprite*** al invertir la escala en horizontal.

```
29 func turn():
30     if not $Area2D/RayCast2D.is_colliding():
31         moving_left = !moving_left
32         scale.x = -scale.x
```

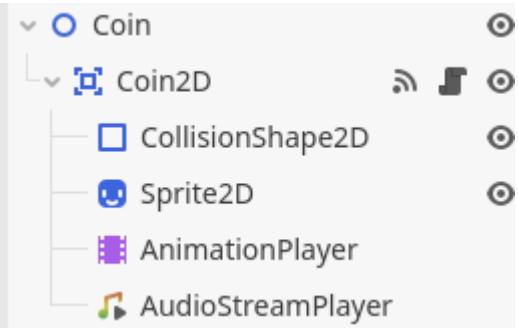
3.2.6. Música y Sonidos

La música y los sonidos ayudan a construir una mejor atmósfera. Es decir, la música ayuda a crear una inmersión al establecer un ambiente para ella. Y los sonidos, informan al receptor sobre las siguientes acciones.

Sonidos

Un ejemplo de sonido será el sonido al recoger una moneda dentro de un videojuego. Por ende, en este caso el sonido añadido dentro del juego será ese.

Primero, se comienza añadiendo un nodo denominado ***AudioStreamPlayer*** dentro de la escena de ***Coin***. El ***AudioStreamPlayer*** hace que un sonido que provenga de la izquierda se escuche por el lado izquierdo, esto da más realismo.



Lo siguiente será añadir el sonido de una moneda. Después de haber seleccionado el sonido, se tiene que seleccionar el ***Script*** del ***Area2D*** de la moneda.

A continuación, se añade una nueva variable dentro del ***Script*** del ***Area2D*** de la moneda e igualarlo al sonido.

Dentro de la función, primero se añade una condición, esta verifica si el cuerpo con el que ha entrado en contacto es el del personaje, entonces se reproduce el sonido y se cumplirá todo lo demás.

Después, se añade un argumento que hace que el ***Sprite*** de la moneda se vuelva invisible una vez recogida. Seguidamente, se añade una variable de ***Timer***, esta es importante añadirlo porque sin la variable, el sonido no se escucharía y directamente desaparecía. Para ello se escribe la variable de ***Timer***, en este caso el tiempo será de 0.21 segundos que es lo que dura el sonido en reproducirse

Luego, se define que el ***Timer*** se ejecute una sola vez, y que inicie el temporizador.

await permite pausar una operación antes de terminar de ejecutar el código.

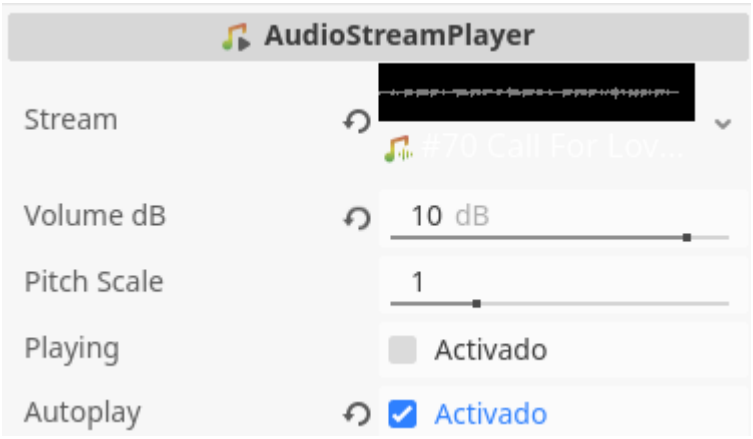
Finalmente, para eliminar la moneda dentro del mapa, se utiliza ***queue free***.

```
3   @onready var CoinSound = $AudioStreamPlayer
4
5   func _on_body_entered(body):
6       if body.get_name() == "PJ":
7           CoinSound.play()
8           $Sprite2D.visible = false
9           var timer = Timer.new()
10          add_child(timer)
11          timer.wait_time = 0.21
12          timer.one_shot = true
13          timer.start()
14          await timer.timeout  # Esperar el t
15          queue_free()  # Elimina la moneda d
16          pass
```

Música

La música es mucho más fácil de implementar que el sonido, lo primero que se hace es añadir el ***AudioStreamPlayer*** en la escena del cual se requiera una música, después de haberlo añadido, toca descargar e implementar la música que se desea.

Una vez añadida, solamente queda activar el modo ***Loop*** y ***Autoplay***. La canción añadida en este caso, no se puede activar el modo ***Loop***.



4. CONCLUSIONES

Al acabar este proyecto, puedo decir que he conseguido básicamente todos los objetivos que me había planteado, como los objetivos personales y la adición de mecánicas. El único objetivo planteado que no he podido lograr, es el de terminar de desarrollar el videojuego.

Pienso que escoger Godot, ha sido una buena opción, este es un motor de software libre, en el cual puedes configurarlo y personalizarlo como quieras. Es bastante práctico y cómodo a la hora de usarlo, hay mucha documentación y una gran comunidad en Godot.

A lo largo de esta investigación, puedo decir que la hipótesis que me había planteado al comienzo de esta investigación "**si el uso de herramientas de software libre facilita la creación de videojuegos para los desarrolladores independientes, sin costos y con una buena calidad**" es cierta. Una herramienta de software libre no solo facilita el desarrollo de un proyecto o varios proyectos que tengas, sino que, también puede ser de ayuda el poder cambiar el código fuente de la misma herramienta. Además, en este proyecto no se ha utilizado ningún elemento de pago y todos estos elementos son de buena calidad, en verdad, pude desarrollar un videojuego en 2D bastante decente.

Tuve algunos problemas a la hora de programar, ya que, los recursos de internet eran bastante antiguos o los códigos no eran compatibles con Godot 4.2.1. que es la versión que utilicé para desarrollar el videojuego.

En conclusión, esta investigación me ha sido útil para profundizar en el mundo de la informática, también, he podido lograr varios objetivos planteados al inicio de la investigación, como lo es el autoaprendizaje, la resolución de problemas, la responsabilidad y el desarrollo de mi propio videojuego. En todo este camino de investigación y desarrollo, he tenido muchas dificultades, pero, sobre todo, ha sido muy divertido haber podido desarrollar un videojuego.

5. BIBLIOGRAFÍA

Ángel, A. (2024, marzo 15). *¿Qué es un motor gráfico? Conoce los entresijos de los videojuegos*. PC Componentes. https://www.pccomponentes.com/motor-grafico-que-es?msockid=2e4a9c9b4a1f634b36e4883d4bb462b8

¿Qué es y cómo funciona un motor gráfico?. (s.f.). *master.d*. https://www.masterd.es/blog/que-es-como-funciona-motor-grafico

Manu, D. (2021, febrero 12). *Los 11 mejores motores gráficos para introducirse en el desarrollo de los videojuegos*. Vandal. https://vandal.elespanol.com/reportaje/los-11-mejores-motores-graficos-para-introducirse-en-el-desarrollo-de-videojuegos

David, E. (2019, junio 10). *Qué es Unity*. OpenWebinars. https://openwebinars.net/blog/que-es-unity/

Erick, R. (s.f.). *¿Qué es Unity (motor de videojuegos multiplataforma): Cómo funciona y para qué sirve?*. LovTechnology. https://lovtechnology.com/que-es-unity-motor-de-videojuegos-multiplataforma-como-funciona-y-para-que-sirve/

Creando y usando Scripts. (s.f.). *Unity Documentation*. https://docs.unity3d.com/es/530/Manual/CreatingAndUsingScripts.html

Documentación de Godot Engine. (s.f.). Godot Docs. https://docs.godotengine.org/es/4.x/index.html

Motor Godot: Qué es, Ventajas y Cómo funciona. (2022, junio 15). *mmmAcademy*. https://mmmacademy.es/godot-que-es-ventajas-funcionamiento/

¿Qué es Unreal Engine? Cómo funciona y por qué es tan famoso. (2023, marzo 2).. *SomosXbox*. https://www.somosxbox.com/que-es-unreal-engine-como-funciona-y-por-que-es-tan-famoso/

¿Qué es Scratch y para qué sirve este programa?. (2022, septiembre 15). *Crack The Code*. https://blog.crackthecode.la/que-es-y-para-que-sirve-scratch

Esteban, C. (2021, marzo 30). *Que es Game Maker Studio: Usos y versiones*. Tokio School. https://www.tokioschool.com/noticias/que-es-gamemaker-studio/
Qué es Game Maker Studio | Aprende Game Maker

Essssam. (s.f.). Rocky Roads Asset Pack. itch.io. https://essssam.itch.io/rocky-roads

Craftpix. (2022). Free Sky With Clouds Background Pixel Art Set. craftpix.net. https://craftpix.net/freebies/free-sky-with-clouds-background-pixel-art-set/?num=1&count=14&sq=sky&pos=0

ChillMindscapes. (s.f.). Free Chiptune Music Pack 4. itch.io. https://chillmindscapes.itch.io/free-chiptune-music-pack-4-chillmindscapes

Dask. (s.f.). Retro Sounds. itch.io. https://dagurasusk.itch.io/retrosounds

LuisCanary. (2013, octubre 23). *LuisCanary*. https://www.youtube.com/@LuisCanary

Renato Meyer. (2017, febrero 15). *Renato Meyer*. https://www.youtube.com/@RenatoMeyer/featured

Lukifah. (2006, julio 23). *Lukifah*. https://www.youtube.com/@Lukifah

Leedeo. (2016, julio 28). *Leedeo Studio*. https://www.youtube.com/@Leedeo

Mart Develop. (2014, octubre 14). *Mart Develop*. https://www.youtube.com/@MartDevelop

findemor. (2006, mayo 4). *findemor*. https://www.youtube.com/@findemor/videos

6. ANEXOS

ANEXO 1: Código completo sobre las físicas y el movimiento del personaje.

```gdscript
extends CharacterBody2D
class_name Character1


var speed = 100
var direccion = 0.0
var jump = 250
var gravity = 7

var lifes = 3

func _physics_process(delta):
	handleHearts()

	velocity.x = direccion + speed

	velocity.x = direccion * speed
	direccion = Input.get_axis("left","right")
	if direccion != 0:
		$AnimationPlayer.play("walk")
	else:
		$AnimationPlayer.play("idle")

	$Sprite2D.flip_h = direccion < 0 if direccion != 0 else $Sprite2D.flip_h

	if is_on_floor() and (Input.is_action_just_pressed("ui_accept") or Input.is_action_just_pressed("up")):
		velocity.y -= jump

	if !is_on_floor():
		velocity.y += gravity


	move_and_slide()
```

ANEXO 2: Funcionamiento de los corazones.