Final Fight Plus

Un proyecto final de CFGS DAM hecho por Cleo Sánchez Moreno.

Índice de Contenidos

Introducción

FinalFightPlus es una recreación del juego original, FinalFight, en formato *roguelike*, hecho en Godot. La idea surgió a partir de la falta de gráficos del juego original (ya que es un juego de terminal). Para ayudar a entender este documento, es necesario incluir un mini-glosario de términos con las palabras más usadas para que todos nos entendamos antes de seguir.

Glosario de términos frecuentes

• FinalFight

Éste es el clásico videojuego de prueba que Gustavo usa para enseñar los conceptos básicos de Java en Programación de primer año. Consiste en un juego de terminal, en el que el usuario introduce números para seleccionar las distintas opciones de interacción, y con ello el programa simula un combate entre el héroe (jugador) y los monstruos que vienen a detenerle. Conforme avanzan los bloques de la asignatura se van añadiendo más mecánicas al juego, como una función para guardar las mejores puntuaciones en disco, funciones para ordenar los enemigos según diferentes parámetros; y en Programación Android, de segundo año, incluso se desarrolla una versión móvil del mismo juego, con *sprites* y habilidades.

• *Roguelike*

Éste género de videojuegos es uno de los más longevos de la historia del medio. Su nombre viene del origen del género, *Rogue*: *Exploring the Dungeons of Doom* (1980), un videojuego cuyas características principales son sus mecánicas basadas en juegos de rol por turnos (Niveles de habilidad, Puntos de Vida, Atributos…), así como el hecho de que cada partida es única e irrepetible; es decir, que los mapas se generan aleatoriamente en cada partida, y que al morir se pierde todo el progreso y toca volver a empezar desde el principio, sin forma de guardar partida.

• Música dinámica

Este término acuñado en el mundo de los videojuegos hace referencia a una parte específica de la programación y el diseño de los mismos, en la que a diferencia de las series y películas, que tienen una banda sonora específica y estática, se aprovecha la interacción del jugador con el entorno para adaptar la música de ambientación a las acciones que se estén llevando a cabo (por ejemplo, cambiar a un tema agresivo cuando se entre en combate, y cambiar de nuevo a un tema más calmado cuando termine).

- *Mapgen*

Abreviatura inglesa para “generación de mapas”. Hace referencia al algoritmo de generación con el que se consiguen mapas aleatorios viables en cada partida. Se explica en detalle en su sección de código.

- *Pathfind*

Abreviatura inglesa para “búsqueda de caminos”. Hace referencia al algoritmo con el que se conectan dos entidades en el terreno de juego mediante una lista de puntos que representa el camino más corto, que también incluye la posibilidad de esquivar obstáculos y seguir dichos caminos en varias direcciones. Se explica en detalle en su sección de código.

Objetivos Técnicos

El objetivo principal es recrear la experiencia original de FinalFight en Godot, añadiendo gráficos, música, mapas generados aleatoriamente y mecánicas roguelike donde sea posible. Para esto es necesario considerar cuatro objetivos principales:

- Generación aleatoria de mapas.
  Este punto cumple uno de los requisitos principales de todo roguelike. Cada vez que se empieza una nueva partida debe ser distinta de la anterior.

- Búsqueda de caminos.
  Ésta es una mecánica surgida en tiempos más recientes, pero que nos ayudará a asegurar la calidad del producto final. En su sección se entrará más en detalles sobre esto.

• Música dinámica.

Además de traer FinalFight a la vida, uno de mis objetivos personales principales es experimentar con este peculiar diseño de los videojuegos.

• Mecánicas jugables.

Por supuesto, es necesario que el videojuego tenga las mecánicas básicas de FinalFight, así como algunas que se puedan añadir como resultado de desarrollar el videojuego en un formato visual.

Riesgos Técnicos

Éstas mecánicas principales pueden crear inconvenientes que será necesario resolver en el desarrollo del videojuego. Dos riegos notables son:

• En la creación de mapas, al generar mapas completamente aleatorios es posible, por ejemplo, que la salida de una cueva se genere en un claro inaccesible para el jugador; por lo que es necesario asegurar que el jugador siempre puede alcanzar la salida de la cueva, y que los enemigos siempre pueden alcanzar al jugador antes de empezar la partida.

• Al implementar música dinámica, es posible que al reproducir varias pistas de audio al mismo tiempo se den errores de solapamiento, o en las transiciones, que destruyan por completo la experiencia de juego. Siempre que se modifique la música es necesario extremar la precaución.

Estructura de código

Descripción General

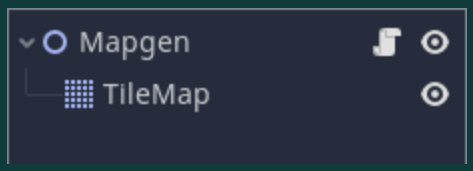
En los siguientes apartados se muestran los nodos usados y una breve explicación de cada uno.

Juego

Éste es el bucle principal de juego en su versión final. Implementa cinco escenas, que es como se llama a las clases en Godot, cada una con sus respectivos nodos, que también son clases, para confusión de nadie. Es interesante mencionar que los nodos se comunican entre sí mediante señales (representadas con el símbolo de conexión inalámbrica), que notifican a otros nodos cuando ocurren ciertos eventos y ejecutan bloques de código en sus scripts, como por ejemplo, un cambio en la música cuando el enemigo detecta al jugador.

Mapgen

Escena que maneja la generación de mapas. Incluye un script, mapgen.gd, y un nodo TileMap, que es esencialmente un diccionario de casillas con distintas propiedades que se usan en el script para generar los mapas aleatoriamente.



Jugador

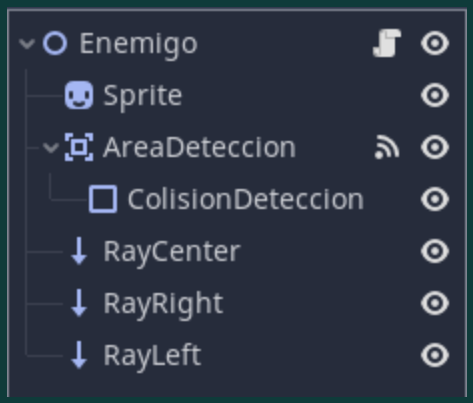
Escena que implementa los controles de movimiento e interacción del jugador con el programa. El nodo principal es un Area2D, una clase de Godot que permite monitorizar el contacto físico de unos elementos con otros mediante *Shapes*, o formas de colisión. Incluye un script, jugador.gd, un Sprite, que es la representación visual del objeto, la forma de la colisión, tres RayCast2D, que son nodos que detectan colisiones con otros objetos a distancia, un nodo Camera2D que lo sigue y muestra la pantalla de juego, y un nodo Label para mostrar los puntos actuales.

Salida

Escena que controla el elemento de salida, el final de la cueva. Incluye un script, salida.gd, un Sprite al igual que el Jugador, dos Áreas con sus respectivas colisiones, una de ellas para identificar cuándo el jugador pisa la casilla de salida y otra más amplia para reproducir una pista de audio estilo "Marco-Polo", que da pistas de que el jugador se está acercando o no a su objetivo. Es interesante mencionar que los nodos Area2D emiten señales al script, que monitorizan cuándo un cuerpo u otra área entran en contacto con ellos y ejecutan bloques de código en el script principal.

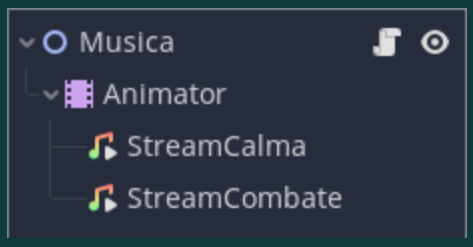
Enemigo

Escena que encapsula la IA de los enemigos. Incluye un script, enemigo.gd, un Sprite como las escenas anteriores, un área de detección que representa el estado de combate y emite una señal al script principal, y tres RayCast2D para detectar colisiones a distancia.



Música

Escena que controla la reproducción de música ambiente. Incluye un script, musica-background.gd, un nodo Animator que permite la automatización de *crossfades* o cambios de una pista a otra, y dos nodos AudioStreamPlayer, que reproducen las pistas de exploración y combate.

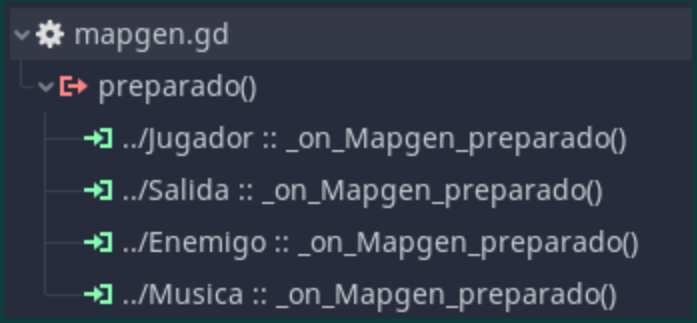


Descripción Exhaustiva

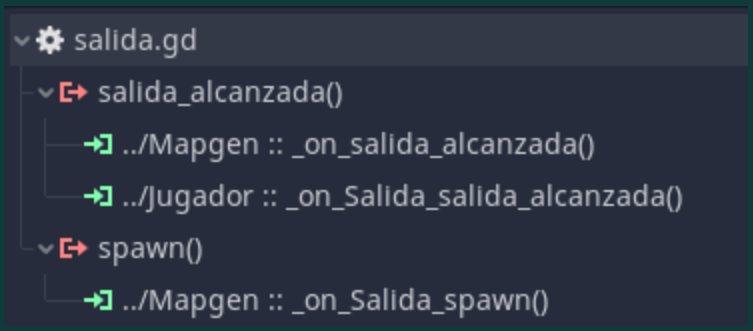
En este apartado entramos en detalle en cada una de las escenas, especialmente en los scripts, y explicamos ciertos bloques de código y su funcionamiento.

Juego

La escena de Juego es la más importante del proyecto. No solo porque es la principal, sino porque además conecta todas las demás escenas entre sí mediante señales, que son observadores de cada una de las escenas que activan bloques de código en scripts remotos cuando se conectan a ellos y se emiten. Echemos un vistazo.



En este ejemplo, el script mapgen.gd, de generación de mapas, tiene una señal personalizada preparado(), que se emite cuando el mapa ha sido generado y los puntos de navegación se han inicializado. Es el pistoletazo de salida para que el resto de scripts aparezcan y entren en acción.



Otra parte importante es el cambio de nivel. Cuando el jugador alcanza la salida, el nodo Area2D de Salida detecta el contacto y emite esta señal, que avisa a Mapgen para que genere un nuevo mapa, y a Jugador para que añada un punto al contador.

Global

Esta sección no es una escena en sí, sino un script precargado, global.gd, que se inicializa al ejecutar el programa y contiene valores constantes y funciones útiles para todas las demás escenas.

```
const MAPA_ANCHO := 48
const MAPA_ALTO := 36
const TAMANO_CASILLA := 16
const MEDIA_CASILLA := TAMANO_CASILLA / 2

const VOLUMEN_MAXIMO := -0.0
const VOLUMEN_MINIMO := -80.0
const VOLUMEN_CAMBIO := 5.0

var cellmap: Array
```

Algunos de esos valores constantes son el tamaño de los mapas, de las casillas individuales y los parámetros de volumen que deben usar los nodos de AudioStreamPlayer, es decir, las escenas con música. También se incluye una copia del mapa actual (la matriz que indica dónde están los suelos y los muros).

```gdscript
# Conversiones entre pixeles de la pantalla y posiciones en la matriz del mapa
func casilla_a_pixeles(casilla: Vector2) -> Vector2:
	return Vector2(casilla.x * TAMANO_CASILLA, casilla.y * TAMANO_CASILLA)


func casilla_a_pixeles_centro(casilla: Vector2) -> Vector2:
	casilla = casilla_a_pixeles(casilla)
	return Vector2(casilla.x + MEDIA_CASILLA, casilla.y + MEDIA_CASILLA)


func pixeles_a_casilla(coordenadas: Vector2) -> Vector2:
	return Vector2(
			int(coordenadas.x / TAMANO_CASILLA),
			int(coordenadas.y / TAMANO_CASILLA)
		)
```

Además, también incluye funciones útiles como la conversión de coordenadas de juego en casillas del mapa y viceversa (por ejemplo: (13, 5) -> (208, 80))…

```gdscript
# Devuelve un punto de spawn aleatorio en una casilla de suelo
func random_spawn() -> Vector2:
	var spawn_x: int
	var spawn_y: int
	var casilla_disponible = 1 # Pared

	while casilla_disponible:
		spawn_x = randi() % MAPA_ANCHO
		spawn_y = randi() % MAPA_ALTO
		casilla_disponible = cellmap[spawn_x][spawn_y]

	var spawn = Vector2(spawn_x, spawn_y)
	return spawn
```

…O la generación aleatoria de puntos de aparición en casillas de suelo, cosa que usan todos los elementos jugables.

Por último, pero no menos importante, también alberga el algoritmo de búsqueda de caminos, que requiere su propia sección. Pero antes de poder entrar en los detalles de la búsqueda de caminos, necesitamos generar un mapa para crear caminos en él.

Mapgen

La generación de mapas de FinalFightPlus consiste en una implementación propia del algoritmo *Cellular Automata*, un concepto propuesto por Stanislaw Ulam y John von Neumann en los años 40, y llevado al público general por John H. Conway en 1970 con su famoso "Juego de la Vida". El concepto principal del Juego de la Vida (en adelante JdlV) consiste en establecer una matriz de casillas con varios estados (por ejemplo, 1 y 0, vivo y muerto) con una configuración y distribución inicial y observar cómo evoluciona con un set de reglas predefinido. En el caso de nuestro juego, ésta es la implementación del algoritmo.

```
func _ready():
	randomize()
	_generar_matriz()
	_generar_mapa()

	for _x in range(SIMULACIONES):
		_simulacion()
	_cerrar_limites()

	_dibujar_mapa()
	global.cellmap = cellmap

	global.astar_ready()

	emit_signal("preparado")
	
	if !_spawn_preparado():
		return
	var path_jugador = global.astar_find_path(spawn_jugador, spawn_salida)
	var path_enemigo = global.astar_find_path(spawn_enemigo, spawn_jugador)
	if path_jugador == [] or path_enemigo == []:
		_ready()
```

En primer lugar, creamos una nueva semilla basada en la hora actual. Luego generamos la matriz, es decir, los arrays bidimensionales que constituyen el mapa. Esto es una función en sí en lugar de una variable porque Godot tiene una forma peculiar de crear matrices. Rellenamos la matriz con valores aleatorios basados en las reglas iniciales (imagen inferior, PROBABILIDAD_INICIAL), y recorremos la matriz pasando cada casilla por las reglas del JdlV hasta generar un mapa completamente distinto, tantas veces como sea necesario para crear cuevas tanto realistas como accesibles. Tras eso cambiamos todos los bordes del mapa con muros para encasillar el terreno de juego y pintamos el mapa en la pantalla.

Las demás funciones están relacionadas con el algoritmo de búsqueda de caminos, y las veremos en el siguiente apartado.

Éstas son las reglas del JdlV.

```
const SIMULACIONES = 3

export var PROBABILIDAD_INICIAL := 0.375
export var LIMITE_VIDA = 4
export var LIMITE_MUERTE = 3

var cellmap := []
onready var tilemap := $TileMap
signal preparado
```

La probabilidad inicial es la aleatoriedad con la que se generan las casillas, es decir, el ratio de casillas vivas con las que se rellena el mapa. Los límites de vida y muerte son los valores de simulación del JdlV, si una casilla muerta tiene LIMITE_VIDA casillas vivas a su alrededor, pasa a estar viva, y viceversa.

Ahora veamos cada paso del proceso en detalle.

```gdscript
# Crea una matriz 2D
func _generar_matriz():
	for i in global.MAPA_ANCHO:
		cellmap.append([])
		for j in global.MAPA_ALTO:
			cellmap[i].append(0)


# Rellena la matriz 2D con valores aleatorios según la regla inicial
func _generar_mapa():
	for x in range(global.MAPA_ANCHO):
		for y in range(global.MAPA_ALTO):
			if randf() < PROBABILIDAD_INICIAL:
				cellmap[x][y] = 1
			else:
				cellmap[x][y] = 0
```

Primero creamos un array bidimensional vacío, con los tamaños descritos en el script global. Por cada casilla de ancho, es decir, por cada columna, se genera un array con un cierto número de datos en él, es decir, las filas.

Acto seguido recorremos esa matriz y establecemos los valores de cada casilla aleatoriamente. Cuanto mayor sea la probabilidad inicial, más casillas vivas tendremos. Cabe mencionar que en nuestro TileMap las casillas con id=0 (muertas) son casillas de suelo atravesables, y las casillas con id=1 (vivas) son muros. Entonces, cuanto mayor sea la probabilidad, más muros se generarán.

```gdscript
# Hace una simulación del juego de la vida con el mapa actual.
# Sustituye el mapa resultante al terminar
func _simulacion():
	var cellmap_proc = cellmap.duplicate(true)

	for x in global.MAPA_ANCHO:
		for y in global.MAPA_ALTO:
			var casilla = cellmap_proc[x][y]
			var vecinos = _contar_vecinos_vivos(x, y)
			if (cellmap[x][y] == 1):
				# La casilla está viva (es un muro), cambia según las reglas de la muerte
				if (vecinos < LIMITE_MUERTE):
					casilla = 0
				else:
					casilla = 1
			else:
				# La casilla está muerta (es suelo), cambia según las reglas de la vida.
				if (vecinos > LIMITE_VIDA):
					casilla = 1
				else:
					casilla = 0
			cellmap_proc[x][y] = casilla

	cellmap = cellmap_proc
```

En el siguiente paso, recorremos el mapa generado aleatoriamente y sometemos cada casilla a las reglas definidas. Es importante mencionar que el mapa resultante se guarda en una copia y se sustituye al terminar, ya que si cambiamos cada casilla en mitad de una ejecución, los resultados no serían correctos. También importante, en la función _contar_vecinos_vivos(x: int, y: int), las casillas de fuera del mapa se cuentan como muros, ya que el jugador no debe ser capaz de salir de los límites.

Tras someter el mapa a varias pasadas del algoritmo, obtenemos resultados prometedores.

Una vez tenemos los mapas generados, podemos colocar en ellos los distintos elementos jugables (Jugador, Enemigos, Salidas…), aunque no es recomendable hacerlo todavía. Uno de los problemas que tiene la aleatoriedad es que no siempre funciona como queremos. Existe una probabilidad de que en una generación, una pequeña área esté completamente rodeada por muros y sea inaccesible, y que la salida o el jugador aparezcan ahí y queden atrapados. En el desarrollo del juego, esto ha pasado innumerables veces, imposibilitando el ritmo de juego. Por eso, tenemos que asegurarnos de que cuando se colocan las

piezas en el tablero, todas ellas pueden acceder las unas a las otras. Para eso y más nos sirve la búsqueda de caminos.

A* Pathfind

El algoritmo de búsqueda A* (pronunciado AStar y escrito así en adelante) es una modificación del algoritmo de Dijkstra, también conocido como *Búsqueda de costo uniforme*. Su objetivo es encontrar el camino más corto hacia un destino, o hacia el destino más cercano de una lista, pasando por una serie de puntos predefinidos.

```
# Algorito AStar
onready var astar = AStar.new()

var inicio_camino = Vector2() setget _set_path_start_position
var fin_camino = Vector2() setget _set_path_end_position
var camino = []


func astar_ready():
  astar.clear()
  var casillas_atravesables = _astar_agregar_casillas_atravesables()
  _astar_conectar_casillas(casillas_atravesables)
```

Como vemos en la imagen (función astar_ready()), el funcionamiento conceptual es simple: primero se elimina todo rastro de puntos anteriores, luego se recorren todas las casillas del mapa y se añaden como puntos aquellas en las que se pueden mover las entidades, y por último se conectan las casillas con sus vecinas para formar una malla de movimiento. Con esa malla, el algoritmo puede encontrar el camino más corto de un origen a un destino atravesando los puntos agregados. Veamos cómo funciona.

```gdscript
# Recorre todo el mapa y añade a la matriz astar nuevos puntos
# en las casillas atravesables
func _astar_agregar_casillas_atravesables():
  var lista_puntos = []
  for y in range(MAPA_ALTO):
    for x in range(MAPA_ANCHO):
      if cellmap[x][y]: # Si la casilla es un muro, no se puede atravesar
        continue
      var punto = Vector2(x, y)
      lista_puntos.append(punto)

      # La clase AStar hace referencia a sus puntos con índices, por lo tanto,
      # necesitamos una forma de traducir las coordenadas a un índice y viceversa.
      # Así cada casilla tiene un id propio, del que se pueden deducir las coordenadas
      var id_punto = _astar_calcular_id_punto(punto)

      # Como AStar trabaja tanto con 2D como con 3D, hay que traducir los puntos
      # a Vector3. Esto solo ocurre aquí.
      astar.add_point(id_punto, Vector3(punto.x, punto.y, 0.0))
  return lista_puntos
```

En primer lugar, recorremos el mapa y añadimos al diccionario AStar todos los puntos que correspondan a casillas de suelo. Cabe mencionar que la clase AStar trabaja con identificadores para los puntos, y no con sus coordenadas, por lo que necesitamos una función que las "encripte y desencripte" fácilmente en los id.

```gdscript
func _astar_calcular_id_punto(punto):
  return punto.x + MAPA_ANCHO * punto.y
```

Precioso. Llegar a esto me tomó un día entero.

Una vez creados y guardados los puntos de navegación de cada casilla, es hora de conectarlos con sus vecinos.

```
func _astar_conectar_casillas(lista_puntos):
  for punto in lista_puntos:
    var id_punto = _astar_calcular_id_punto(punto)

    # Para cada casilla, conectamos con las de arriba, abajo, izquierda y derecha
    var puntos_relativos = PoolVector2Array([
      Vector2(punto.x + 1, punto.y),
      Vector2(punto.x - 1, punto.y),
      Vector2(punto.x, punto.y + 1),
      Vector2(punto.x, punto.y - 1)])

    for relativo in puntos_relativos:
      var id_relativo = _astar_calcular_id_punto(relativo)
      if casilla_fuera_de_limites(relativo):
        continue
      if not astar.has_point(id_relativo):
        continue
      astar.connect_points(id_punto, id_relativo, false)
```

Esto no tiene ningún misterio, se recorre de nuevo la matriz de puntos y si los puntos adyacentes son atravesables y están dentro de los límites del mapa, se conectan entre sí.
Pero finalmente, con la malla creada y conectada, llegamos a la parte interesante del algoritmo: la búsqueda de caminos *per se*.

```
func astar_find_path(inicio, fin):
  self.inicio_camino = inicio
  self.fin_camino = fin
  _calcular_camino()
  var camino_pixeles = []
  for punto in camino:
    var punto_pixeles = casilla_a_pixeles_centro(Vector2(punto.x, punto.y))
    camino_pixeles.append(punto_pixeles)
  return camino_pixeles
```

De nuevo, conceptualmente es simple. Se elige un punto de salida y un punto de destino, se guardan los puntos que se deben recorrer para llegar de uno al otro, y se devuelve el camino a seguir. Por suerte para todos, AStar2D es una clase integrada de Godot y el algoritmo de pathfind se resume en una sola función.

```gdscript
func _calcular_camino():
	var id_inicio = _astar_calcular_id_punto(inicio_camino)
	var id_fin = _astar_calcular_id_punto(fin_camino)
	camino = astar.get_point_path(id_inicio, id_fin)
```

Ahora podemos llamar a estas funciones desde el script de generación de mapas para comprobar que todos los elementos son accesibles.

¡EUREKA! Ésta parte del juego me costó doce días.

Música

La parte sustancial de este apartado no es el código del script musica-background.gd, sino la manera en la que se crearon y editaron esas canciones que se usan en el juego. Pero antes de eso, veamos el script.

```
const lista_calma = [
  "res://musica/calm-1-larnii.mp3",
  "res://musica/calm-2-larnii.mp3",
  "res://musica/calm-3-larnii.mp3",
  "res://musica/calm-4-larnii.mp3",
  "res://musica/calm-5-larnii.mp3",
  "res://musica/calm-6-larnii.mp3",
  "res://musica/calm-7-larnii.mp3",
  "res://musica/calm-8-CoQ.mp3"
]

const lista_combate = [
  "res://musica/combat-1-larnii.mp3",
  "res://musica/combat-2-larnii.mp3",
  "res://musica/combat-3-larnii.mp3",
  "res://musica/combat-4-larnii.mp3"
]
```

En primer lugar, empezamos cargando todas las referencias a los recursos que usaremos en este apartado. Esto parece simple e indoloro, pero en la sección de problemas encontrados se explica en detalle por qué quizá no fue la mejor idea.

```
func _ready():
  randomize()
  stream_calma.stream = _cancion_aleatoria(false)
  stream_combate.stream = _cancion_aleatoria(true)
  stream_calma.volume_db = global.VOLUMEN_MAXIMO
  stream_combate.volume_db = global.VOLUMEN_MINIMO
  stream_calma.play()
  stream_combate.play()
```

Al iniciar el juego e inicializar esta escena, lo primero que ocurre es que se cargan canciones de exploración y combate aleatorias y se establece el volumen de la exploración al máximo, y del combate al mínimo. Asumimos que al empezar una nueva partida el jugador no estará rodeado por enemigos inmediatamente, y lo asumimos con seguridad porque si se diera el caso, el jugador entraría en la zona de detección de dichos enemigos y activaría la música de combate inmediatamente, por lo que esto no nos supone un problema.

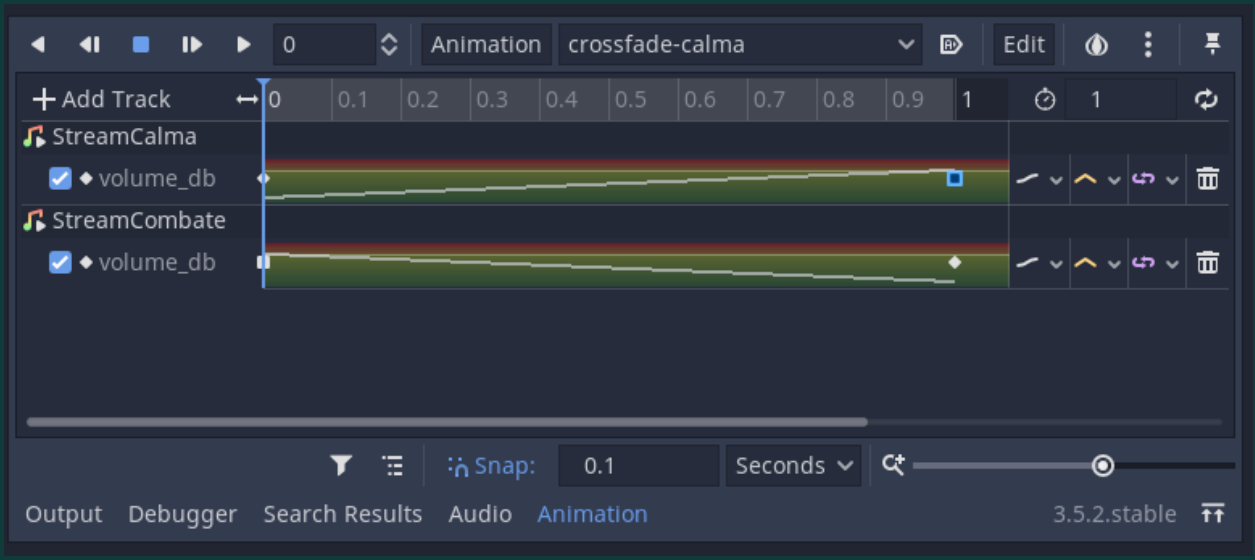
```
func _cancion_aleatoria(combate: bool):
    # Elegir cancion
    var path: String
    if !combate:
        var size = lista_calma.size()
        path = lista_calma[randi() % size]
    else:
        var size = lista_combate.size()
        path = lista_combate[randi() % size]

    # Cargar el recurso
    if File.new().file_exists(path):
        var res = load(path)
        res.set_loop(true)
        return res
```

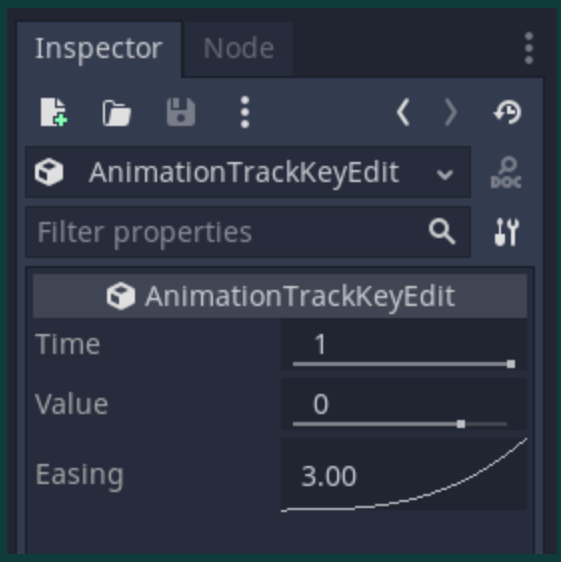
Es cierto que de un nivel a otro puede darse el caso de que una canción se repita; pero es un problema que tiene fácil solución, como veremos en la creación de las canciones.

```
53 func _on_musica_calma():
54     animator.play("crossfade-calma")
55
56
57 func _on_musica_combate():
58     animator.play("crossfade-combate")
```

Ésta sección hace uso de las señales explicadas al principio. Nótense los iconos en el gutter, indicando que estos bloques de código se activan con señales. Éstas son las mencionadas zonas de detección de los enemigos, que reproducen una animación específica. Hablemos un poco más de eso.

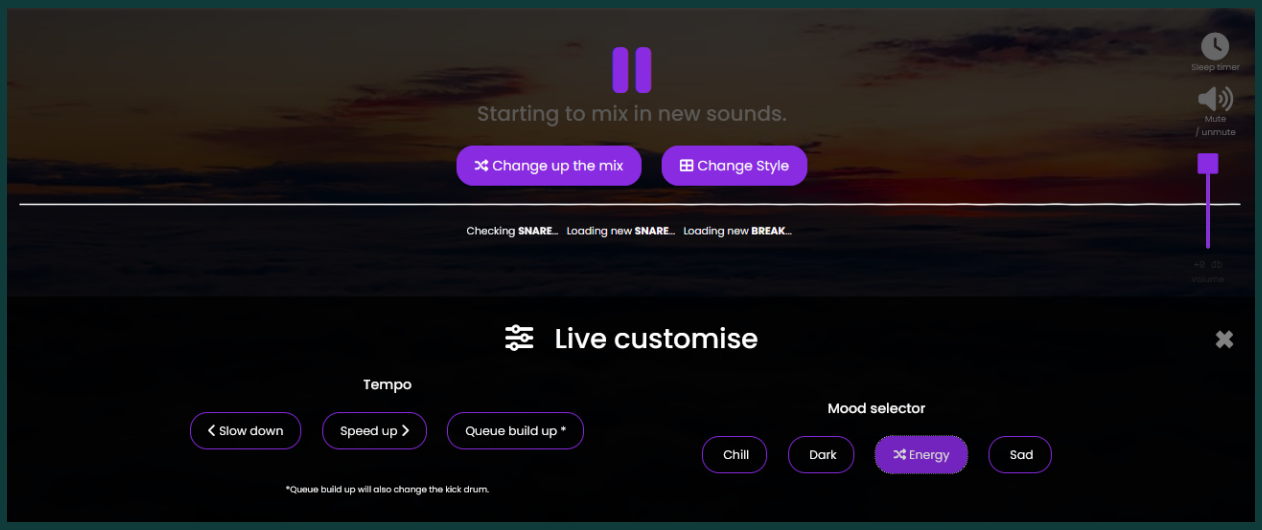


Este es el controlador de animaciones de Godot. Con esta herramienta se puede automatizar cualquier propiedad de nuestros nodos, desde la rotación de un sprite hasta el color de unas luces. Si una propiedad tiene un valor inicial y un valor final predefinidos, se puede animar. En este caso, creamos una secuencia que reduce el volumen de la música de combate y aumenta el de la exploración cuando termina una batalla. Toda animación se basa en *keyframes*, o fotogramas de referencia, que representan los valores que tiene la variable en distintos puntos de la animación.



Entre cada *keyframe* se dan las transiciones, monitorizadas por el valor *easing*. Este valor permite suavizar las transiciones, haciendo que no sean lineales, sino graduales. De esta forma, retardando la salida de la canción de combate y suavizando la entrada de la canción de calma conseguimos que la transición sea más suave, y que nunca haya un momento donde no se reproduzca nada, o peor aún, donde ambas pistas suenen al mismo volumen y se solapen entre sí.

Pero no podemos automatizar música si no tenemos música primero. He de decir que al principio del proyecto me ilusionaba la idea de componer las pistas a mano, pero una de las razones por las que no todos los videojuegos del mercado implementan estas funcionalidades es que es costoso. Componer una canción para una película que suba y baje con la acción que se ve en pantalla es relativamente sencillo, comparado con componer seis variaciones de una pista, intentando predecir cuándo el jugador dejará de escuchar una y entrará la otra, para cada pista de un juego de X horas de duración, con diferentes temáticas y ritmos; y teniendo en cuenta que todo eso se puede romper con tan solo pulsar el botón de pausa. Es una locura. Por eso, es mucho más atractivo dejar el aspecto de esa composición a una Inteligencia Artificial.



Larnii es una herramienta de generación de música infinita. Recoge *samples* o muestras de canciones de todo Internet, y las compone de formas únicas dependiendo del estado de ánimo seleccionado. Permite al usuario elegir entre una amplia variedad de géneros musicales, aumentar y reducir el tempo de la canción e incluso modificar ciertos elementos individuales, como la percusión y los pads. También permite descargar ciertas canciones pregeneradas, pero para este ejemplo lo que hice fue grabar a la IA mientras componía y seguir trabajando, y al final exportar el audio resultante y recortar las pistas para que tuvieran un inicio y un final. El 92.3% de la música del juego está compuesta por esta herramienta.

Jugador

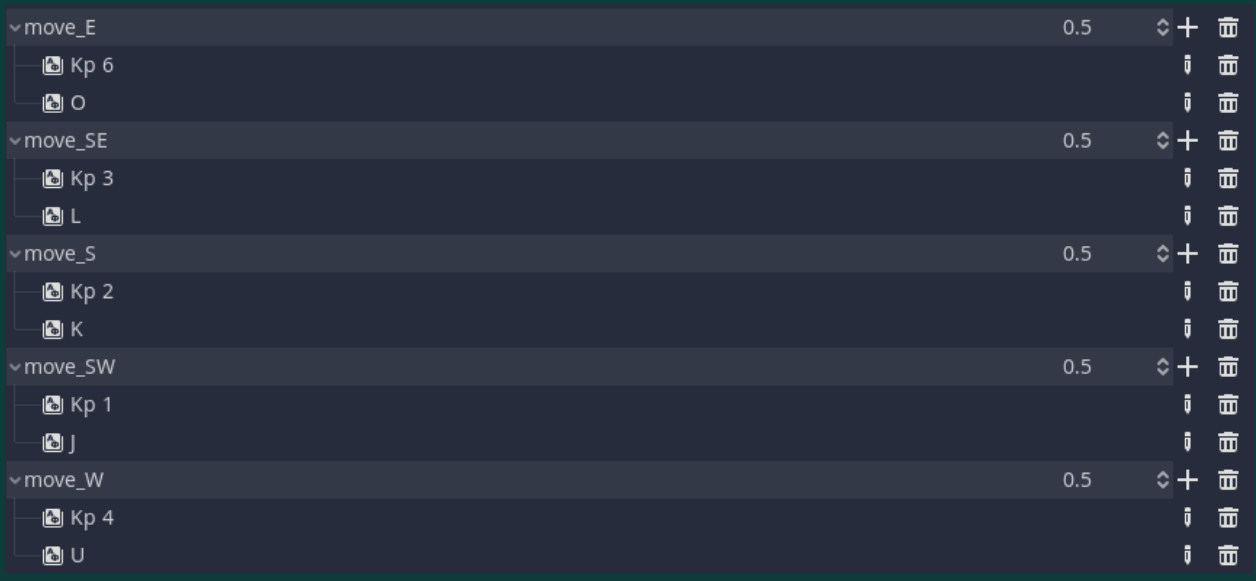
El funcionamiento de Jugador se divide en dos partes, moverse a casillas atravesables y detectar colisiones.

```
const MOVIMIENTOS = {
  "move_N":  Vector2.UP,
  "move_NE": Vector2.UP + Vector2.RIGHT,
  "move_E":  Vector2.RIGHT,
  "move_SE": Vector2.DOWN + Vector2.RIGHT,
  "move_S":  Vector2.DOWN,
  "move_SW": Vector2.DOWN + Vector2.LEFT,
  "move_W":  Vector2.LEFT,
  "move_NW": Vector2.UP + Vector2.LEFT,
}
```

Esta es la lista de movimientos disponibles. Nótese que el jugador puede moverse en diagonal, pero los caminos generados con AStar solo permiten las cuatro direcciones cardinales. Esto se debe a que en ocasiones el movimiento diagonal no es posible, como por ejemplo al atravesar un muro por un hueco creado por error en la generación de mapa. Sin embargo, el movimiento en diagonal en casillas atravesables sí es posible, y para eso están los RayCast2D, explicados en un momento.

```
func _unhandled_input(event):
  for direccion in MOVIMIENTOS.keys():
    if event.is_action_pressed(direccion):
      mover_a(direccion)
```

Cada vez que se pulsa una tecla se guarda el evento en un búfer, y el juego lo manda a recoger por las funciones _unhandled_input(event). El evento se propaga desde las partes inferiores del árbol de nodos, es decir, los nodos más específicos, hacia arriba hasta la raíz. Si un evento llega a la raíz sin gestionar, se ignora.



En nuestro caso, los eventos son los movimientos registrados, y en la configuración del proyecto cada uno está asignado a una serie de teclas específicas. Cuando se pulsa estas teclas, el evento se recoge y se gestiona en esta función.

```
func mover_a(direccion):
    var destino = global.casilla_a_pixeles(MOVIMIENTOS[direccion])
    var colisiones = 0

    for key in raycasts:
        var ray = raycasts[key]
        ray.cast_to = destino
        ray.force_raycast_update()
        if ray.is_colliding():
            colisiones += 1

    if colisiones < 3:
        position += destino
```

Al recibir un evento, se registra la casilla de destino a la que el jugador quiere acceder, se lanzan rayos de detección para comprobar si hay colisiones en el objetivo, y si no, se ejecuta el movimiento. Estos rayos están dispuestos en el centro del jugador, y ligeramente a la izquierda y a la derecha. Esto se debe a que dado que las

colisiones de los muros son cuadrados perfectos, en ciertas ocasiones al moverse en diagonal el rayo central colisiona con la esquina de un muro y no permite el movimiento. Con rayos adicionales a la izquierda y a la derecha nos podemos asegurar de que al tocar la esquina de un muro, el movimiento sigue siendo viable porque hay un espacio disponible en uno de los rayos extra.

Enemigo

La funcionalidad de los enemigos es uno de los aspectos que se tuvo que cortar por razones de tiempo. En una versión del proyecto tenían una inteligencia artificial rudimentaria, que les permitía perseguir al jugador y atacarle cuando estuviese cerca, pero estaba repleta de bugs y no llegó a la versión final. Éste es el resultado.

```
13
14 func _on_Mapgen_preparado():
15     var spawn = global.casilla_a_pixeles_centro(global.random_spawn())
16     position = spawn
17     emit_signal("spawn", spawn)
18
19
20 func _on_AreaDeteccion_area_entered(area):
21     if area is Jugador:
22         emit_signal("musica_combate")
23
24
25 func _on_AreaDeteccion_area_exited(area):
26     if area is Jugador:
27         emit_signal("musica_calma")
28
```

Todas estas funciones son señales. Cuando se prepara el mapa, se elige una posición de aparición y se genera un enemigo. Cuando el jugador entra en combate, se cambia a la música de combate, y viceversa cuando sale de la zona.

Salida

El script de salida tiene las mismas funciones que el de los enemigos, pero con una peculiaridad. En la escena de la salida hay un AudioStreamPlayer que reproduce una pista de unas maracas en bucle cuando el jugador se acerca, indicando que se está yendo en la

dirección correcta. Esta transición fue hecha a mano, sin uso de Animators, puesto que es lo suficientemente sencilla.

```
func _on_AreaAudio_area_entered(area):
  if area is Jugador:
    volumen_actual = global.VOLUMEN_MAXIMO


func _on_AreaAudio_area_exited(area):
  if area is Jugador:
    volumen_actual = global.VOLUMEN_MINIMO
```

Cuando el jugador se acerca, se aumenta el volumen, y cuando se aleja, se disminuye.

```
func _process(_delta):
  if audio_stream.volume_db == volumen_actual:
    return

  if audio_stream.volume_db < volumen_actual:
    audio_stream.volume_db += global.VOLUMEN_CAMBIO
  elif audio_stream.volume_db > volumen_actual:
    audio_stream.volume_db -= global.VOLUMEN_CAMBIO
```

En cada turno se comprueba el volumen actual y el volumen al que debería estar la pista, y si han cambiado, se aumenta o disminuye según sea necesario.

Más adelante caí en que esto es ineficiente, ya que cada turno hay que comprobar si la música ha cambiado, a diferencia de las animaciones, que se reproducen una sola vez cuando se indica y dejan de existir el resto del tiempo, pero el impacto en el rendimiento es inapreciable al jugar, así que lo dejé como estaba. Pero conste en acta que aprendí de mi error.

Secciones Extra

¿Por qué Godot?

La elección del motor de juego se basa en tres características.
La característica cero, es que simplemente me gusta.

Dicho eso, la primera es su facilidad de uso. En comparación con otros motores como Unity o GameMaker, Godot ofrece una experiencia de usuario mucho más amigable, con la inclusión de características como el botón de reset, un botón que aparece en cada propiedad que el usuario cambia en el inspector de código. A lo largo del desarrollo esto ha demostrado ser útil en la gran mayoría de aspectos, permitiendo experimentar y *“cachivachear”* sin miedo a romper nada que no se pueda deshacer. Además, el motor es gratis y de código libre. FOSS FTW.

La segunda característica es su ligereza. Mientras que Unity requiere una instalación completa del motor, una cuenta registrada con una clave de uso y varios minutos para siquiera ponerse en marcha, Godot consiste de: un archivo godot.exe. De 71MB. *Y ya está*. Además, los proyectos creados con este motor rara vez incluyen archivos de más de 100MB, lo que facilita la implementación con GitHub y la fácil subida de cambios al repositorio.

La tercera y la más importante es que la escala del juego permite el uso de motores de menor calibre para alcanzar los mismos resultados, lo que también resulta en juegos con menores requisitos técnicos para el ordenador.

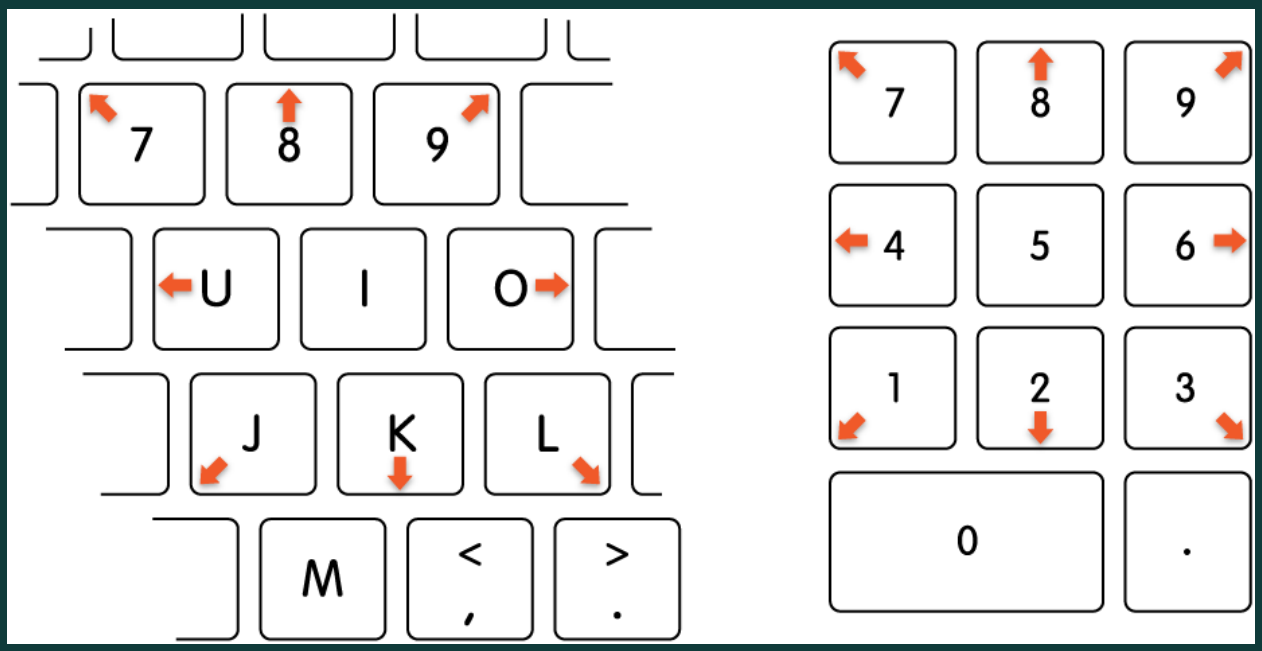
Herramientas de Arte y Audio

Las texturas fueron creadas con una herramienta online llamada Pixilart (véase Webgrafía), y editadas con GIMP. Las pistas de audio fueron generadas por Larnii (véase Webgrafía) y editadas con Audacity, salvo la pista CoQ, que fue compuesta usando FL Studio 20.

Requisitos y Plataforma de distribución

Los requisitos mínimos del programa son, una tarjeta de vídeo o un procesador con gráfica integrada con al menos 2MB de VRAM (es decir, que se puedan ver cosas en la pantalla), y opcionalmente una tarjeta de sonido para poder escuchar la música. También, en caso de que el teclado del equipo no tenga pad numérico, existe un esquema de control alternativo que permite el movimiento con las teclas convencionales.

Esquema de controles



Manual de Instalación de la Aplicación

No es necesario instalar nada. Tan solo descargar el proyecto desde la página de GitHub, ejecutar el archivo FinalFight++.exe y jugar. No se hace uso de librerías externas ni de bases de datos, aunque sí es necesario no borrar el archivo FinalFight++.pck, o de lo contrario el programa no ejecutará. Las carpetas de assets también son necesarias para que el juego pueda cargar las pistas de audio.

Problemas encontrados

· Mala administración del tiempo, no solo con la FCT y los motivos personales, sino en el proyecto en sí. Muchos de los commits en el repositorio que en teoría deberían centrarse en una sola característica modifican varias cosas a la vez. Es un fallo personal porque en cuanto veo que algo no está bien ajustado me desconcentra y lo necesito arreglar inmediatamente. Esto implica que muchas sesiones de trabajo se han descarriado a temas completamente diferentes y que las partes más grandes del proyecto han tomado mucho más tiempo del necesario, restando del resto.

· Selección de algoritmo de mapgen. Ésta fue una decisión difícil ya que hay mil y una formas de generar un mapa aleatoriamente. Por suerte tuve un poco de dirección con el videojuego del que tomé inspiración, Caves of Qud (del que por cierto nace el formato de este documento). En ese juego, los mapas se generan usando una mezcla de *Wave Function Collapse* y *Cellular Automata*, creando los primeros mapas con WFC a partir de imágenes predefinidas, y limando las asperezas con CA. El problema de esto es que muchas horas de investigación fueron destinadas al algoritmo de WFC, sin saber en mi inocencia que eso no era lo que el proyecto necesitaba, ya que las cuevas de este juego son completamente aleatorias y sin referencia. Crear una referencia e implementar dos algoritmos hubiera tomado como mínimo el doble de tiempo, así que aunque fue un problema, fue resuelto antes de que se convirtiera en un desastre.

· Algoritmo AStar. Los problemas con este algoritmo son incontables. Desde la curva de dificultad para siquiera entender cómo funciona, con sus puntos descritos por ids y sus mallas de conexión, hasta la sequía de recursos para Godot con respecto a otros motores, pasando por que es la única forma aparente de crear caminos en rejillas 2D en Godot, sin tener que poner cada punto a mano; y terminando en que la implementación tomó más prueba y error que cualquier otro aspecto del proyecto. Sin duda una experiencia educativa, pero que no querría repetir, y que tomó mucho más tiempo del esperado y bloqueó las demás partes del desarrollo (ya que si un enemigo aparece atrapado entre muros no se puede testear su IA). Muchos de estos problemas podrían haber sido resueltos son tan solo usar la versión más reciente de Godot, la versión 4.0, que implementa una clase llamada AStar2D**GRID** hecha específicamente para rejillas bidimensionales, con funciones

para hacer más eficiente la gestión y conexión direccional de los puntos. Pero la versión 4.0 hace grandes cambios en muchos otros aspectos del motor, por lo que no era una buena idea al principio del desarrollo empezar con el programa y aprenderlo de nuevo, ni tampoco era viable migrar la tecnología a la hora de programar los caminos.

• Godot y la exportación de proyectos. Éste también es un problema de la versión de Godot. Al exportar proyectos, Godot sólo empaqueta los recursos que se usan inmediatamente en cada escena, como los Sprites, cuyos recursos se asignan en el inspector de nodos. Sin embargo, la música se carga a través de código, por lo que esos recursos no se cargan hasta que no empieza la ejecución del programa. Las versiones de Godot desde 3.0 contienen este error y en los últimos momentos del desarrollo, limpiando todo el código posible, no hubo manera de averiguar cómo empaquetar el resto de assets, por lo que el juego ni siquiera tenía música. Por eso es necesario no borrar ni el archivo .pck, que contiene todos los assets precargados, ni las carpetas del proyecto, que contienen los cargados durante la ejecución. Lección aprendida: dejar la próxima vez varios días de gracia para resolver estos inconvenientes.

Webgrafía

Documentación en Videojuegos: Documento de diseño (GDD) - EL DOCUMENTALISTA AUDIOVISUAL
Introduction to the A* Algorithm
Algoritmo de búsqueda A*
Cellular automaton
Conway's Game of Life
Larnii
Pixilart