TRABAJO FINAL DE GRADO

Desarrollo de videojuego

Por Ángel Sánchez Mendoza

Índice

1. Introducción

Future vs Fantasy es un juego de acción y supervivencia inspirado principalmente en el estilo *Vampire Survivors*. El jugador controla a un personaje del futuro equipado con armas modernas que intenta viajar al pasado para cambiar el destino de la humanidad, pero por error acaba en otra dimensión y ahora debe enfrentarse a hordas de enemigos que habitan en un mundo de fantasía.

1.1 Antecedentes

En los últimos años, el género de videojuegos tipo *survivor roguelike*, como el popular *Vampire Survivors*, han ganado una notable popularidad por su jugabilidad sencilla pero adictiva, combinando mecánicas de supervivencia, progresión por oleadas y mejoras constantes del personaje. Estos juegos suelen caracterizarse por enfrentamientos masivos contra hordas de enemigos, crecimiento progresivo del jugador y un fuerte componente rejugable.

Inspirado en estas dinámicas, el presente Trabajo de Fin de Grado plantea el desarrollo de un videojuego original que combina la estética futurista con un mundo de fantasía, buscando ofrecer una experiencia algo diferente y visualmente atractiva para el jugador. Este proyecto se enmarca en un enfoque práctico de diseño y desarrollo de videojuegos, utilizando herramientas modernas que permiten gestionar tanto el apartado visual como la lógica de juego.

1.2 Desarrollo de la idea inicial

Future vs Fantasy es un videojuego de acción y supervivencia en el que el jugador encarna a un personaje proveniente del futuro, equipado con armas tecnológicamente avanzadas(o actuales para nosotros). Su misión era viajar al pasado para evitar las catástrofes que avecinan el futuro, pero un error en el proceso lo transporta a una dimensión alternativa dominada por criaturas mágicas y lugares extraños.

Este juego busca generar un contraste estético y mecánico entre tecnología y fantasía. El personaje principal deberá enfrentarse a oleadas de enemigos, recolectar cristales para mejorar y adaptarse a un entorno hostil y desconocido. Además, se implementa un sistema de progresión que permite mejorar habilidades básicas, armas y resistir mejor el avance imparable de los enemigos.

1.3 Objetivos

El objetivo general del proyecto es desarrollar un videojuego completo y funcional que sea divertido de jugar en cualquier momento, que tengas 10 minutos libres y quieras despejar tu mente un rato jugándolo.

Objetivos específicos:

- Diseñar un personaje jugable con mecánicas de combate y progresión.
- Implementar sistemas de generación de enemigos, experiencia y niveles.
- Integrar un HUD que muestre vida, puntuación y tiempo de juego.
- Desarrollar varios mapas con características y enemigos distintos.
- Ofrecer una experiencia rejugable mediante los diferentes entornos.

1.4 Tecnologías utilizadas

- **Motor gráfico:**
  Se ha utilizado **Godot Engine** como motor de desarrollo principal, por su carácter libre, multiplataforma y su eficiencia en el desarrollo de videojuegos en 2D. Además, su sistema de nodos facilita una estructura jerárquica clara y modular, ideal para proyectos de pequeña y mediana escala.

- **Conexión a base de datos:**
  El sistema de puntuaciones se gestiona a través de una API sencilla conectada a una base de datos **MongoDB**, en la que se utiliza una única colección llamada players. Esta colección almacena los nombres de los jugadores junto a sus puntuaciones máximas, permitiendo así un ranking persistente y externo al propio juego.

  Enlace a la API: https://github.com/Ang3l1llo/API_PSP

- **Gráficos:**
  El apartado visual se basa en **pixel-art**, lo cual encaja con la estética retro y minimalista del género. Los gráficos han sido seleccionados a partir de una mezcla de assets libres de diferentes autores, procurando mantener una coherencia visual que refuerce la identidad artística del juego.

- **Inteligencia Artificial:**
  La **IA** implementada para los enemigos es de carácter básico. Una vez generados (“spawneados”), los enemigos se dirigen directamente hacia el jugador, ejecutando ataques de manera aleatoria entre las opciones que tienen disponibles. A pesar de su simplicidad, este comportamiento es suficiente para generar presión constante sobre el jugador y fomentar el dinamismo en el combate.

- **Control de versiones:**
  Git y **github** para controlar los cambios y salvar el proyecto.

- **Diseño de sonido:**
  Principalmente ***riffusion*** para la música, una herramienta online de generación de música mediante IA. Para los efectos un pack de efectos de sonido cortesía de mi profesor y tutor del proyecto Javier Ortega.

2. Descripción

El resultado obtenido final del proyecto es un videojuego funcional y completo con las siguientes características:

2.1 Género y Plataforma

- **Género**: Roguelike, Bullet Hell, Supervivencia.
- **Plataformas**: PC (posible adaptación a móvil en el futuro).

2.2 Historia y Ambientación

El protagonista es un viajero del tiempo cuyo objetivo es salvar a la humanidad del futuro que les espera, ya que está plagado de guerras por culpa de los que ostentan el poder, problemas sociales, pésimos gobernantes y líderes, mucha hambruna...Pero hay un problema que los científicos e ingenieros calculan mal y ocurre un error en la matriz del transportador, provocando que nuestro viajero llegue por error a otro universo en lugar del pasado, un mundo dominado por seres fantásticos. Con su arsenal de armas modernas, debe sobrevivir y derrotar a las oleadas de enemigos mientras busca una forma de regresar a su tiempo, o al menos, a su mundo.

2.3 Mecánicas del juego

- Movimiento en 360° (teclado).
- Disparo manual en la dirección del cursor del jugador.
- Hordas de enemigos aumentando progresivamente la dificultad con el tiempo.
- Mejoras y habilidades adquiridas con la experiencia que se consigue eliminando enemigos.

2.4 Controles

- **PC**: Teclas W-A-S-D o teclas de dirección para moverse, clic o barra espaciadora para disparar.
- **Móvil** (futuro desarrollo): Joystick virtual para moverse, botón virtual de disparo.

2.5 Personajes

- **Protagonista**: Un viajero del tiempo armado con tecnología moderna.
- **Enemigos**:
    - Caballeros medievales con espadas, arcos, escudos, lanzas..
    - Orcos con gran resistencia y ataque.
    - Magos que lanzan proyectiles mágicos.
    - Esqueletos terroríficos y slimes.
    - Bestias como hombres-lobo y hombres-oso.

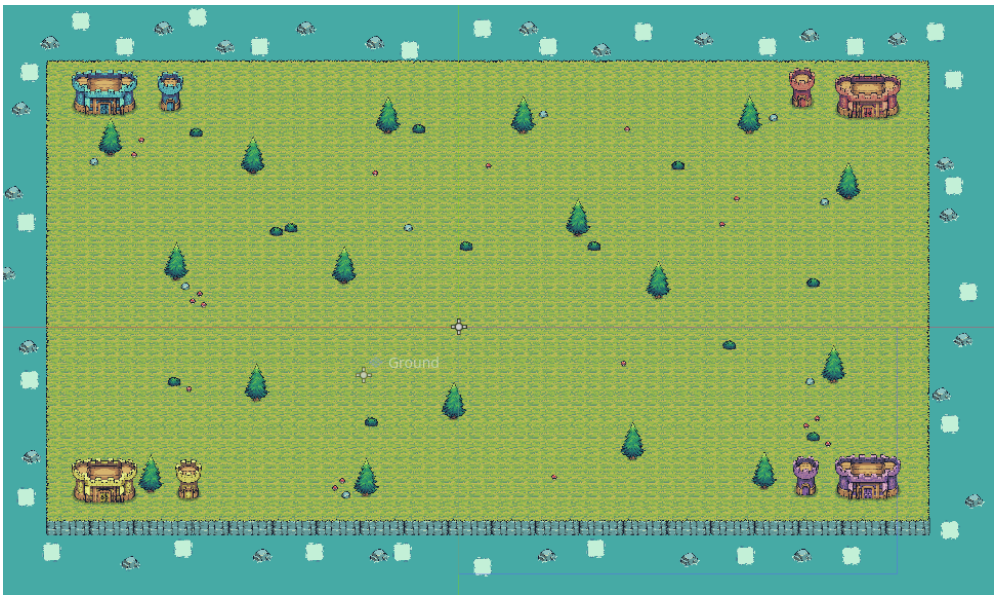
2.6 Armas y habilidades

- **Pistolas**: Cadencia buena, daño reducido al ser el arma más básica.
- **Subfusiles**: Cadencia muy alta, daño bajo que se compensa con la cantidad de balas por segundo.
- **Fusiles**: Cadencia buena, daño más elevado.
- **Escopeta**: Cadencia baja, daño elevado y disparo con dispersión de los proyectiles.
- Al subir de nivel se podrá elegir o bien mejorar el arma, o bien mejorar habilidades(más vida o velocidad de movimiento)

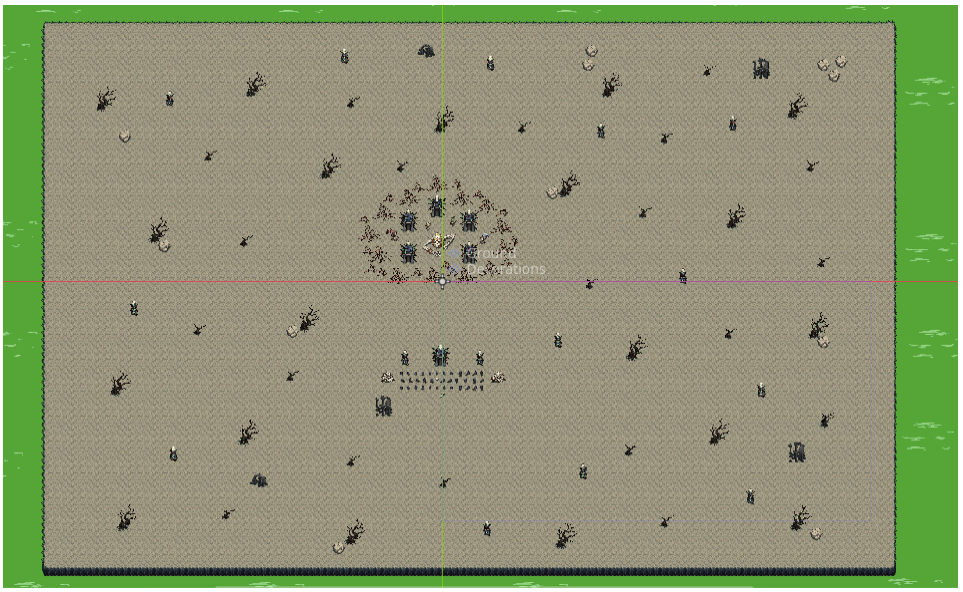
2.7 Niveles y Mapas

Hay 3 mapas distintos, cada uno con su propio diseño y características.

- **MeadowLands:** Las praderas, es el mapa más sencillo que simularía las tierras de unos reinos, los enemigos son humanos como arqueros y caballeros.



- **MisteryWoods**: Los bosques misteriosos, este mapa es algo más complejo y tétrico, con referencias más oscuras. Los enemigos son esqueletos.



- **FinalZone:** La zona final, este es el nivel más difícil, los enemigos tienen más vida, daño, velocidad y hay mayor cantidad. Son orcos, bestias y magos para darle un plus de dificultad con un enemigo que lanza proyectiles más fuertes.

2.8 Sistema de progresión

- Eliminar enemigos hace que suelten un cristal, que dependiendo del color dará más o menos experiencia al recolectarlo.
- Subida de nivel que permite mejorar habilidades básicas como la vida o velocidad de movimiento o bien cambiar tu arma actual por una mejor.

2.9 Objetivo del juego

- Sobrevivir eliminando enemigos para poder subir nivel, mejorar tu personaje y así poder aguantar los 10 minutos que dura cada nivel.

2.10 Música

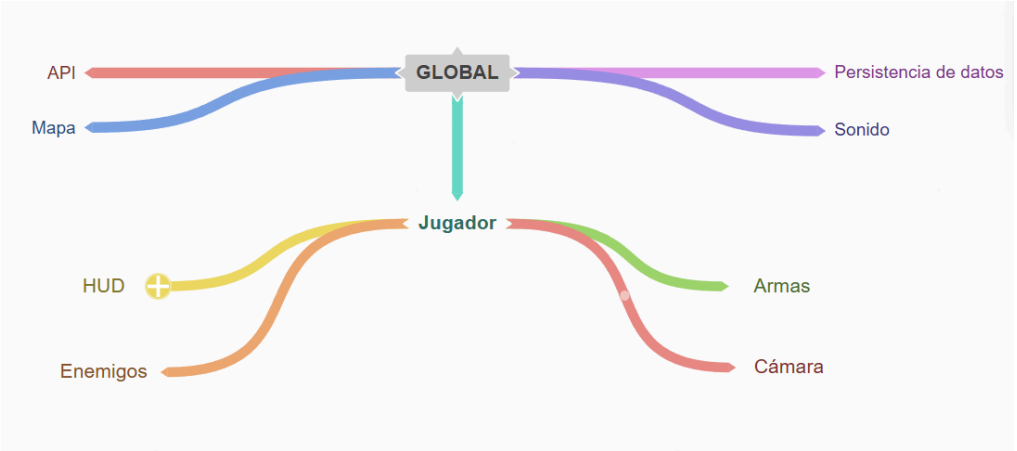
Cada nivel tiene 3 canciones diferentes, la primera es siempre la más “suave” o tranquila, la segunda es un poco más intensa y la tercera es la más potente para realzar esos momentos de tensión en los últimos minutos de cada nivel en los que es más difícil sobrevivir. En cuanto a los estilos se mezcla una electrónica tipo 8 bits, rock and roll y heavy metal.

3. Instalación

El juego se distribuye mediante un **ejecutable independiente**, lo que facilita su instalación y uso por parte del usuario final. No se requieren conocimientos técnicos ni instalaciones adicionales. Basta con ejecutar el archivo proporcionado para comenzar a jugar.

4. Diseño funcional

El videojuego está desarrollado en Godot, un motor que sigue el paradigma de orientación a objetos mediante un sistema de nodos organizados en escenas. Aunque GDScript (el lenguaje nativo de Godot) no utiliza clases tradicionales, la estructura de nodos permite un diseño modular análogo a la programación orientada a objetos. Dada esta particularidad, para representar el diseño del proyecto se ha elaborado un esquema de nodos, en lugar de un diagrama UML, ya que refleja de manera más fiel la arquitectura real implementada en el motor:



5. Desarrollo

5.1 Secuencia de desarrollo

- ☐ **Diseño inicial:** Conceptualización del juego, historia base y mecánicas inspiradas en *Vampire Survivors* con toques propios.

- ☐ **Prototipado:** Implementación de un primer personaje jugable, sistema de movimiento, disparo y generación básica de enemigos.

- ☐ **Sistema de progresión:** Creación de recogida de experiencia, subida de nivel y mejoras.

- ☐ **Mapas y enemigos:** Desarrollo de distintos mapas y enemigos con comportamientos variados.

- ☐ **HUD y feedback visual:** Inclusión de barra de vida, temporizador, efectos.

- ☐ **Creación de diferentes tipos de menús:** para la interfaz con el usuario.

- ☐ **Conexión con la base de datos:** Desarrollo de una API y su integración para guardar puntuaciones de los jugadores.

- ☐ **Persistencia de datos:** Desarrollo de un sistema de guardado local para la partida.

- ☐ **Gestión global del estado del juego:** Implementación de un script cuyo patrón de diseño es un singleton que centraliza múltiples aspectos del juego: nombre del jugador, puntuación, conexión con la API externa, control del progreso entre niveles, y funcionalidades compartidas como sonidos globales.

- ☐ **Pulido y pruebas finales:** Ajustes en la dificultad, visuales y rendimiento.

5.2 Dificultades encontradas

Durante el desarrollo surgieron diversas dificultades, entre ellas:

- La **gestión de colisiones y navegación** de enemigos en mapas con obstáculos, que requirió ciertos ajustes.
- El sistema de **generación de enemigos** fuera de cámara, evitando zonas no jugables como agua o árboles, esto fue con diferencia lo más complejo.
- **Animación** de muchos enemigos con diferentes patrones de ataque, lo que conlleva mucho tiempo.
- Mantener la **coherencia visual** entre distintos assets gráficos de autores variados.
- Conseguir que la **cámara** no diera saltos o generase el típico efecto de “lag” o el “tearing”.

5.3 Decisiones afrontadas

1. Lo más complejo fue el desarrollo del sistema de generación de enemigos o **SpawnManager**.

```
func spawn_enemies():
    for i in range(enemies_per_spawn):
        var _spawned = false
        for attempt in range(100): #Pongo muchos intentos para aumentar la probabilidad de que salga bien
            var pos = get_random_spawn_position()

            if is_valid_spawn(pos):

                var enemy = pick_enemy(current_map).instantiate()
                enemy.global_position = pos
                get_parent().add_child(enemy)
                current_enemies.append(enemy)

                if enemy.has_signal("enemy_died"):
                    enemy.connect("enemy_died", Callable(self, "_on_enemy_died").bind(enemy))

                _spawned = true
                break
```

Inicialmente, los enemigos aparecían en zonas no deseadas, como fuera del mapa o sobre el agua. Para solucionarlo, se definió una *zona segura* utilizando un nodo ***Rect2***, que actúa como una región rectangular dentro de la cual se permite la aparición de enemigos.

```
@export var spawn_area_rect: Rect2
```

Sin embargo, dentro de esa zona también había árboles, lo cual seguía provocando apariciones no deseadas. Para resolverlo, se implementó un sistema de **validación mediante colisiones**: se simula una forma circular con un radio de 50 píxeles centrada en la posible posición de aparición. Si esa forma colisiona con objetos que utilizan exclusivamente la **máscara de colisión 5** (reservada para los árboles), se considera que la posición no es válida y se busca una nueva.

```gdscript
#Para comprobar si ese spot es valido
func is_valid_spawn(pos: Vector2) -> bool:
	var test_shape = CircleShape2D.new()
	test_shape.radius = 50

	var query = PhysicsShapeQueryParameters2D.new()
	query.shape = test_shape
	query.transform = Transform2D(0, pos)
	query.collision_mask = 1 << 4

	var space_state = get_world_2d().direct_space_state
	var result = space_state.intersect_shape(query)

	if result.size() > 0:
		return false

	return true
```

Este procedimiento se repite hasta encontrar una posición válida o agotar un número de intentos. Este enfoque permitió mantener la generación de enemigos dentro de un área jugable, evitando a la vez la superposición con obstáculos naturales.

Para la **posición aleatoria** del enemigo, en esa zona segura pero fuera de la cámara del jugador, se implementa el siguiente método:

```gdscript
func get_random_spawn_position() -> Vector2:
	var rect = spawn_area_rect
	var random_distance = randf_range(spawn_distance_min, spawn_distance_max)
	var angle = randf() * TAU
	var offset = Vector2.RIGHT.rotated(angle) * random_distance

	var spawn_pos = player.global_position + offset

	for _i in range(50):
		if spawn_area_rect.has_point(spawn_pos) and spawn_pos.distance_to(player.global_position) >= spawn_distance_min and spawn_pos.distar
			return spawn_pos

	spawn_pos = Vector2(
		randf_range(rect.position.x, rect.position.x + rect.size.x),
		randf_range(rect.position.y, rect.position.y + rect.size.y)
	)
	return spawn_pos
```

Si entra dentro del rango de la distancia máxima y mínima (para que no salga ni en la cámara del jugador ni tampoco demasiado lejos), retorna una posición.

Para la **elección aleatoria** del **enemigo**, se tiene en cuenta el mapa, el peso que tiene cada enemigo, se genera un número aleatorio entre 0 y el peso total. Luego, se recorre la tabla acumulando los pesos hasta encontrar el primer enemigo cuyo peso acumulado supera el valor aleatorio generado. Este es el enemigo seleccionado.

```
#Para elegir de forma aleatoria el enemigo
func pick_enemy(map_name: String) -> PackedScene:
    var enemy_weights = enemies_by_map.get(map_name, {})
    if enemy_weights.is_empty():
        return null

    var total_weight = 0.0
    for weight in enemy_weights.values():
        total_weight += weight

    var rand = randf() * total_weight
    var cumulative = 0.0

    for enemy_scene in enemy_weights:
        cumulative += enemy_weights[enemy_scene]
        if rand <= cumulative:
            return enemy_scene

    # Fallback (no debería llegar aquí nunca)
    return enemy_weights.keys()[0]
```

Este enfoque garantiza que algunos enemigos aparezcan más frecuentemente que otros, sin necesidad de repetir elementos ni generar valores duplicados. Por ejemplo, un enemigo con peso 0.6 tendrá una probabilidad de aparición mayor que otro con peso 0.2, pero sin estar completamente restringido a valores fijos.

El siguiente reto fue el tratamiento de enemigos que quedaban atascados al chocar con decoraciones del entorno. Para evitar que el jugador pudiera simplemente huir y evadir a todos los enemigos, se introdujo una lógica por la cual, si un enemigo se aleja demasiado del jugador (por encima de cierta distancia), este desaparece y vuelve a generarse en otra localización. Esto garantiza una presión constante sobre el jugador y mantiene el ritmo del juego.

```
#Para eliminar un enemigo si se encuentra demsiado lejos
func check_enemies_distance():
    for enemy in current_enemies.duplicate():
        if not is_instance_valid(enemy):
            current_enemies.erase(enemy)
            continue

        if player.global_position.distance_to(enemy.global_position) > despawn_distance:
            current_enemies.erase(enemy)
            enemy.queue_free()
```

El sistema también gestiona dinámicamente la **dificultad**: la frecuencia, cantidad y variedad de enemigos se ajusta automáticamente en función del tiempo transcurrido de partida, lo cual se calcula usando los valores exportados del HUD (**elapsed_time y game_duration**). Además, cada mapa tiene asociada una tabla de enemigos con proporciones distintas, lo que permite adaptar el desafío a la zona y mejorar la variedad.

```gdscript
func _process(delta):
    if not player:
        return

    check_enemies_distance()
    update_spawn_parameters()

    spawn_timer += delta
    if spawn_timer >= spawn_interval:
        spawn_timer = 0.0
        if current_enemies.size() < max_enemies:
            spawn_enemies()
```

```gdscript
#Va actualizando la cantidad y tiempos del spawn
func update_spawn_parameters():

    var elapsed_time = hud.get_elapsed_time()
    var progress = clamp(elapsed_time / hud.game_duration, 0.0, 1.0)

    # Primer valor como empieza, segundo valor como termina
    spawn_interval = lerp(spawn_interval_start, spawn_interval_endgame, progress)
    enemies_per_spawn = int(lerp(enemies_per_spawn_start, enemies_per_spawn_endgame, progress))
    max_enemies = int(lerp(max_enemies_start, max_enemies_endgame, progress))
```

```gdscript
var enemies_by_map = {
    "MeadowLands": {
        preload("res://Scenes/Enemies/Archer.tscn"): 0.5,
        preload("res://Scenes/Enemies/ArmoredAxeman.tscn"): 0.6,
        preload("res://Scenes/Enemies/knight.tscn"): 0.3,
        preload("res://Scenes/Enemies/knightTemplar.tscn"): 0.3,
        preload("res://Scenes/Enemies/SwordMan.tscn"): 0.2,
        preload("res://Scenes/Enemies/Lancer.tscn"): 0.05
    },
    "MisteryWoods": {
        preload("res://Scenes/Enemies/Skeleton.tscn"): 0.6,
        preload("res://Scenes/Enemies/SkeletonArcher.tscn"): 0.5,
        preload("res://Scenes/Enemies/ArmoredSkeleton.tscn"): 0.2,
        preload("res://Scenes/Enemies/Slime.tscn"): 0.3,
        preload("res://Scenes/Enemies/GreatSwordSkeleton.tscn"): 0.05
    },
    "FinalZone": {
        preload("res://Scenes/Enemies/orc.tscn"): 0.6,
        preload("res://Scenes/Enemies/ArmoredOrc.tscn"): 0.4,
        preload("res://Scenes/Enemies/EliteOrc.tscn"): 0.3,
        preload("res://Scenes/Enemies/WereWolf.tscn"): 0.2,
        preload("res://Scenes/Enemies/OrcRider.tscn"): 0.1,
        preload("res://Scenes/Enemies/WereBear.tscn"): 0.05,
        preload("res://Scenes/Enemies/Wizard.tscn"): 0.05
    }
```

2. El problema de tanta animación lo solucioné usando condicionales y desactivación de colisiones: Como cada enemigo tiene varios ataques, con rangos y formas diferentes, lo que hago es que si realiza el *ataque1,* solo afecta la *colisión1,* desactivando las demás, si realizase el 2 pues la colisión 2.. Esto también se podría haber solucionado usando un nodo llamado *animationplayer* pero requería mucho tiempo aprendizaje en el diseño y opté por solucionarlo mediante código de la forma propuesta.

```gdscript
func attack_if_possible():
    if not can_attack:
        return

    is_attacking = true
    can_attack = false

    # Ataque aleatorio
    var attack_type = randi() % 3

    if attack_type == 0:
        attack_shape1.disabled = false
        attack_shape2.disabled = true
        attack_shape3.disabled = true
        await play_and_wait("ATTACK1")

    if attack_type == 1:
        attack_shape1.disabled = true
        attack_shape2.disabled = false
        attack_shape3.disabled = true
        await play_and_wait("ATTACK2")

    else:
        attack_shape1.disabled = true
        attack_shape2.disabled = true
        attack_shape3.disabled = false
        await play_and_wait("ATTACK3")

    attack_shape1.disabled = true
    attack_shape2.disabled = true
    attack_shape3.disabled = true
```

3. Con el fin de centralizar y facilitar la gestión de variables y funciones compartidas entre escenas, se implementó un **script singleton** denominado **Global,** que actúa como un nodo autoload. Este nodo se mantiene persistente a lo largo del juego y permite acceder a información esencial como el nombre del jugador, la puntuación, el progreso en los niveles o las llamadas a la API externa.

```gdscript
# Variables para la puntuación y la API
var player_name: String = ""
var player_id: String = ""
var score: int = 0
var score_at_level_start: int = 0
var top5_players: Array = []


#Varibales para guardar partida
var progress = {
    "MeadowLands": false,
    "MisteryWoods": false,
    "FinalZone": false
}
```

Para implementar persistencia entre sesiones, se desarrolló una lógica de guardado utilizando archivos en formato JSON dentro del directorio **user://**

```
func save_progress(level_name: String):
    var save_path = "user://savegame.json"

    # Cargar datos actuales guardados para no perderlos
    var data = {
        "progress": {
            "MeadowLands": false,
            "MisteryWoods": false,
            "FinalZone": false
        },
        "player_name": Global.player_name,
        "score": Global.score
    }

    if FileAccess.file_exists(save_path):
        var file = FileAccess.open(save_path, FileAccess.READ)
        var content = file.get_as_text()
        var parsed = JSON.parse_string(content)
        if typeof(parsed) == TYPE_DICTIONARY:
            data = parsed
        file.close()

    # Actualizar progreso si nivel válido
    if level_name != "":
        var clean_name = level_name.get_file().get_basename()
        data["progress"][clean_name] = true

    # Actualizar nombre y score actuales (por si cambiaron)
    data["player_name"] = Global.player_name
    data["score"] = Global.score

    # Guardar datos actualizados
    var fileNewData = FileAccess.open(save_path, FileAccess.WRITE)
    fileNewData.store_string(JSON.stringify(data))
    fileNewData.close()
```

La siguiente función sería la encargada de traerse ese json y así poder cargar los datos.

```
# Funciñon para cargar partida
func load_progress():
    var save_path = "user://savegame.json"
    if FileAccess.file_exists(save_path):
        var file = FileAccess.open(save_path, FileAccess.READ)
        var content = file.get_as_text()
        var parsed = JSON.parse_string(content)
        if typeof(parsed) == TYPE_DICTIONARY:
            if parsed.has("progress"):
                progress = parsed["progress"]
            if parsed.has("player_name"):
                Global.player_name = parsed["player_name"]
            if parsed.has("score"):
                Global.score = parsed["score"]
        file.close()
```

El juego está conectado a una API externa desarrollada en *ASP.NET Core*, que permite registrar al jugador y actualizar su puntuación online. La función ***create_player*** permite registrar un nuevo jugador al comienzo de la partida:

```
# Llamada a la API para crear jugador
func create_player(nombre: String, callback_node: Node):
    player_name = nombre
    var url = "https://api-psp-1nuc.onrender.com/api/Player"
    var headers = ["Content-Type: application/json"]
    var body = JSON.stringify({ "nombre": nombre, "puntuacion": 0 })

    var request = HTTPRequest.new()
    add_child(request)
    request.request_completed.connect(callback_node._on_create_player_completed)
    var err = request.request(url, headers, HTTPClient.METHOD_POST, body)
    if err != OK:
        push_error("Error al crear jugador: %s" % err)
```

La puntuación se actualiza con la función **add_points:**

```
# Llamada a la API para sumar puntos
func add_points(points: int):
    if player_id == "":
        print("ID del jugador no definido aún")
        return
    score += points

    #Ese /%s es una forma de formatear el string en GDScript
    var url = "https://api-psp-1nuc.onrender.com/api/Player/%s/addpoints" % player_id
    var headers = ["Content-Type: application/json"]
    var body = str(points)

    var request = HTTPRequest.new()
    add_child(request)
    request.request(url, headers, HTTPClient.METHOD_PUT, body)
```

6. Pruebas

Las pruebas realizadas al tratarse de un videojuego fueron principalmente de tipo **manual y visual**, empleando:

- **Impresiones por consola (print())** para comprobar estados internos (vida, nivel, daño, posición de enemigos, etc.).

- **Verificación visual** de los comportamientos en pantalla: movimiento, colisiones, ataques, efectos de daño y animaciones.

- Se realizaron múltiples partidas de prueba para ajustar tiempos, daño, experiencia y frecuencia de aparición de enemigos.

A pesar de no haber empleado herramientas de testing automatizado, las pruebas realizadas fueron suficientes para validar el correcto funcionamiento del sistema en condiciones normales de uso.

7. Distribución

7.1 Tecnologías de distribución

El proyecto se distribuye como un **ejecutable exportado desde Godot Engine**, compatible con sistemas Windows, lo que permite ejecutarlo sin necesidad de instalar el motor ni librerías adicionales.

Para la distribución, se utilizó la herramienta de **exportación integrada en Godot**, la cual empaqueta todos los recursos del juego y genera un único archivo ***.exe.***

7.2 Proceso de distribución

El proceso consistió en:

1. Configurar las opciones de exportación dentro de Godot, como el nombre o icono.

2. Generar el ejecutable final con todos los recursos integrados.

8. Instalación

El proceso de instalación es extremadamente sencillo:

1. Descargar el juego desde github, alojado en la carpeta ***build_windows*** en el enlace siguiente: https://github.com/Ang3l1llo/TFG

2. Ejecutar el archivo y listo.

9. Conclusiones

Mi resultado final ha sido, en muchos aspectos, mejor de lo esperado, aunque también ha habido elementos que no han salido como me gustaría.
Al principio no tenía muy clara la dirección exacta que quería seguir: buscaba algo original, pero sin alejarme demasiado de las bases que funcionan, inspirándome en *Vampire Survivors*, pero intentando diferenciarme al reducir el protagonismo de los múltiples proyectiles automáticos.

Mi objetivo principal era centrarme en la parte de programación, buscando que el juego se viera y sintiera **fluido y pulido al jugarse**, y creo que lo he conseguido en gran parte.
Uno de los aspectos que menos satisfecho me deja es el **movimiento de los enemigos**, ya que me habría gustado que fuera más natural y dinámico, con una IA algo más inteligente.

Por otro lado, estoy muy contento con varios logros:

- Todas las **animaciones** de los enemigos funcionan correctamente, con diferentes patrones de ataques que suceden aleatoriamente.

- Las **armas** y sus **proyectiles** están implementados y diferenciados según su tipo.

- La **interfaz** de usuario y los menús están bien organizados, usando múltiples nodos de control.

- La **duración del juego es de 30 minutos**, algo que no esperaba conseguir inicialmente. Además, gracias al diseño modular, el juego es fácilmente ampliable y escalable con más niveles o enemigos simplemente añadiendo nuevos *assets*.

En resumen, estoy satisfecho con el resultado. Aunque me habría gustado mejorar la **IA y la movilidad enemiga**, y también que fuese **apto para móviles,** creo que con algo más de tiempo podría haberlo logrado. Aun así, el proyecto refleja bien la idea inicial y tiene una buena base sobre la que seguir construyendo y mejorando.

10. Bibliografía

- **Juegos en godot:** Iniciando con la programación de videojuegos - *Diego Darío Lopez Mera*

- Apoyo en diversos tutoriales y foros.