CONSEJERÍA DE EDUCACIÓN

Comunidad de Madrid

IES ENRIQUE TIERNO GALVAN

Parla

CFGS DESARROLLO DE APLICACIONES MULTIPLATAFORMA

Curso 2024/2025

Proyecto DAM

TITULO: *Campesino Místico*

Alumno: *Noma Kelvin Obamedo*

Tutor: *Jesús García Romero*

Junio de 2025

Índice

1. Descripción general del proyecto: "Campesino Místico"

1.1. Introducción y Concepto Central

Este proyecto presenta el diseño y desarrollo de **"Campesino Místico"**, un videojuego de acción y aventura, ambientado en un oscuro mundo de fantasía medieval.

El juego busca ofrecer una experiencia inmersiva, llevando al jugador a una época dura y peligrosa, marcada por la brutalidad, la superstición y la injusticia. La historia gira en torno a un campesino que vive una vida sencilla, hasta que un día regresa a su hogar y encuentra que su familia ha sido asesinada por un grupo de bandidos.

Este suceso cambia su vida para siempre. Lleno de dolor y con un fuerte deseo de justicia o quizás de venganza, el protagonista se ve obligado a dejar atrás su pasado y embarcarse en un viaje lleno de peligros, recorriendo tierras hostiles y enfrentándose a enemigos y desafíos constantes.

A lo largo de esta aventura, el jugador acompaña al campesino en su transformación, descubriendo nuevas habilidades, enfrentando decisiones difíciles y explorando un mundo tan bello como cruel.

1.2. Ambientación y Tono

El mundo de **"Campesino Místico"** tiene una atmósfera realista, inspirada en la dureza de la Edad Media. Es un lugar opresivo, donde la vida es difícil y peligrosa, y donde cada rincón puede esconder una amenaza. Esta realidad está mezclada con elementos de misterio, leyendas olvidadas.

Los escenarios del juego incluyen aldeas, campos de cultivo abandonados, bosques. Todos estos lugares están diseñados para transmitir una sensación constante de tensión y peligro, reforzando el ambiente del juego.

La historia toca temas serios y profundos, como la pérdida, la búsqueda de uno mismo en tiempos difíciles y la delgada línea que separa la justicia de la venganza.

1.3. Pilares Fundamentales del Juego

El diseño de **"Campesino Místico"** se basa en tres pilares principales que trabajan juntos para ofrecer una experiencia completa:

1. **Combate en Tiempo Real**

   El sistema de combate está pensado para ser **intenso y desafiante**, poniendo énfasis en la sensación de vulnerabilidad del protagonista, especialmente al inicio del juego.

   El jugador comenzará sin experiencia, pero a medida que avance, podrá acceder a armas más elaboradas. Será clave aprender los movimientos de los enemigos, esquivar o bloquear ataques, posicionarse bien y gestionar la resistencia para sobrevivir.

   El progreso en combate irá desbloqueando nuevas habilidades y técnicas, reflejando cómo el campesino va transformándose en un guerrero, aunque no lo haya buscado.

2. **Narrativa con Decisiones del Jugador**

   Aunque la historia principal tiene un punto de partida claro, el juego permitirá al jugador tomar decisiones importantes durante su avance.

   Estas elecciones afectarán las relaciones con otros personajes, el desarrollo de subtramas, el acceso a ciertas zonas o recursos, y en algunos casos, incluso el final del juego.

El objetivo es que el jugador sienta que sus acciones tienen consecuencias reales, y que su versión del campesino sea única según las decisiones que tome.

1.4. Propósito del Documento

El objetivo principal de esta memoria técnica es explicar en detalle todo el proceso de desarrollo del videojuego *Campesino Místico*. Desde el análisis inicial y el diseño (Conceptualización), hasta la implementación.

Este documento está pensado para ser una **guía completa** del proyecto, mostrando sus partes técnicas, las decisiones tomadas durante el desarrollo, y cómo se implementaron las distintas funciones del juego.

2. Análisis

2.1. Diagrama de Flujo del Juego ("Concepto de juego")

En el **anexo 10.1.1** se presenta el concepto de *El Campesino Místico*, con el recorrido del juego y sus posibles finales.

Este diagrama ilustra la estructura general de navegación y los principales estados por los que puede pasar la aplicación desde la perspectiva del usuario. Representa el flujo de la experiencia de juego.

Componentes Clave:

- **Menú Principal**: Punto de entrada y nodo central desde el cual se accede a las demás secciones principales (Juego, Créditos).
- **Juego**: Representa el estado activo de jugabilidad, donde el usuario interactúa con las mecánicas centrales (combate, exploración). Es el núcleo del bucle de juego.
- **Ganar / Perder**: Son los posibles resultados inmediatos de la sesión de juego activa. Indican éxito o fracaso en el objetivo actual.
- **Siguiente nivel / Reinicio de juego**: Representan las transiciones lógicas tras ganar o perder. Siguiente nivel implica progresión, mientras que Reinicio de juego permite al jugador reintentar tras un fracaso, ambos retornando al estado Juego.
- **Juego terminado**: Estado alcanzado tras completar el último nivel o la secuencia final del juego.
- **Karma check**: Un nodo crucial que representa un sistema de evaluación de las acciones o decisiones del jugador acumuladas durante la partida. Este sistema determina la narrativa final.

- **Final bueno / Final malo:** Representan los distintos desenlaces narrativos posibles, condicionados por el resultado del Karma check. Esto introduce consecuencias a las acciones del jugador.

- **Crédito**: Estado donde se muestra la información de los desarrolladores, accesible tanto desde el Menú Principal como después de visualizar cualquiera de los finales.

Propósito e Importancia: Este diagrama es fundamental para entender la arquitectura general de juego y la experiencia del usuario. Define las rutas de navegación posibles, el ciclo de vida de una partida (inicio, juego, resultado, continuación/fin) y cómo se integran elementos narrativos clave como los finales múltiples basados en las acciones del jugador. Sirve como guía para la implementación de la gestión de escenas y transiciones.

2.2. Diagrama de Máquina de Estados Finitos (FSM)

En el **anexo 10.1.3** se muestra el diagrama FSM (Máquina de Estados Finitos) del personaje principal, donde se representan los distintos estados de movimiento como caminar, saltar, caer o quedarse quieto, y las transiciones entre ellos.

En este juego utilizamos una Máquina de Estados Finitos (FSM**)** para organizar mejor el comportamiento de los personajes y del propio juego. Gracias a este sistema, el código es más claro, ordenado y fácil de mantener, lo que facilita futuras mejoras y hace que el desarrollo sea mucho más eficiente.

Estados:

Cada estado representa una acción o situación concreta del personaje. En este caso:

- Idle: Quieto, esperando.
- Walk: Caminando.
- Jump: Saltando hacia arriba.
- Fall: Cayendo después de un salto.

Estado Inicial:

Es el primer estado cuando el personaje comienza. Normalmente es Idle.

Transiciones:

Las flechas entre los estados indican **cómo puede cambiar el personaje** de un estado a otro:

- De Idle a Walk cuando empieza a moverse, y de vuelta cuando se detiene.
- De Idle o Walk a Jump si se pulsa saltar.
- De Jump a Fall cuando empieza a descender.
- De Fall a Idle o Walk al tocar el suelo.

El FSM permite controlar el comportamiento y las animaciones del personaje de forma clara, sin tener que usar muchos if/else complicados.

3. Conceptualización

3.1. Introducción

"Campesino Místico" es un juego de acción y plataformas en 2D con un fuerte componente de secreto, ambientado en un mundo de fantasía medieval.

El jugador controla a un humilde campesino cuya vida cambia para siempre al encontrar a su familia asesinada. Consumido por el dolor y la sed de venganza, no tiene otra opción que adentrarse en un mundo hostil y peligroso.

Sin haber recibido entrenamiento alguno, el personaje debe aprender a sobrevivir usando el entorno y las sombras a su favor. Moviéndose, esquivando trampas y enemigos, irá perfeccionando sus habilidades.

El juego combina una historia personal de venganza y lucha contra la injusticia con una jugabilidad que mezcla plataformas precisas, todo dentro de escenarios oscuros y llenos de peligros.

3.2. Storyboard

En el **anexo 10.1.2** se muestra el storyboard de *El Campesino Místico*, donde se cuenta la historia del juego. Este es un storyboard de 5 cuadros que describe el inicio de una historia de campesino.

1. Un campesino regresa a su hogar en un entorno medieval.
2. Descubre que su familia ha sido asesinada, llenándose de dolor y deseo de justicia.
3. Se embarca en un viaje de venganza, sin experiencia, hacia un mundo hostil.
4. El viaje implica explorar un mundo abierto lleno de enemigos.
5. Se enfrentará a combates desafiantes motivado por la venganza. Sus decisiones afectarán el curso de la historia en una narrativa interactiva.

En resumen, narra cómo la tragedia transforma a un simple campesino en un guerrero decidido a cambiar su mundo, donde cada elección que tome tendrá consecuencias profundas.

3.3. Objetivo de juego

El propósito fundamental del jugador en "Campesino Místico" es guiar al protagonista campesino a través de un viaje peligroso, impulsado por la tragedia personal y la búsqueda de retribución en un mundo hostil. Los objetivos se pueden separar en dos niveles principales:

1. **Objetivo a Nivel de Juego**:
    - **Navegar y sobrevivir:** El jugador debe recorrer entornos 2D desafiantes como bosques utilizando habilidades de plataformas como saltar, correr, luchar etc.
    - **Evitar o neutralizar amenazas:** Deberá superar trampas mortales y gestionar la presencia de enemigos como animales, bandidos o criaturas.
    - **Alcanzar la salida o punto clave:** El objetivo inmediato en cada nivel será encontrar y llegar al final de la sección, superando obstáculos y enemigos para continuar la historia y descubrir nuevas zonas del mundo.
2. **Objetivo General / Narrativo:**
    - **Descubrir la Verdad:** Investigar los responsables del asesinato de la familia del protagonista y los oscuros secretos detrás de este crimen.
    - **Buscar Venganza / Justicia:** Enfrentar a los culpables, luchando contra aquellos que los respaldan y contra el sistema que perpetúa la injusticia en el reino.

- **Forjar/Crear un Destino:** Tomar decisiones clave que influirán en la narrativa, las relaciones con otros personajes y en la moralidad del protagonista. ¿Se dejará consumir por la venganza o buscará redención y algo más? Estas elecciones determinarán el curso del juego.
- **Sobrevivir y Evolucionar:** El protagonista pasará de ser un campesino indefenso a un superviviente, adquiriendo habilidades esenciales (agilidad, uso de herramientas y armas) para enfrentarse a un mundo peligroso y cumplir con su misión.

El jugador debe dominar las mecánicas de plataformas para superar los desafíos de cada nivel, mientras sigue una historia de venganza personal que lo obliga a tomar decisiones difíciles y a enfrentarse a enemigos cada vez más fuertes, lo que influye en el desarrollo y el final de su trágica historia.

3.4. Controles

El juego utiliza controles sencillos e intuitivos para las acciones fundamentales. Por defecto, estos son:

- **Saltar:** [Flecha Arriba]
- **Moverse a la Izquierda:** [Flecha Izquierda]
- **Moverse a la Derecha:** [Flecha Derecha]
- **Atacar**: [Tecla Espacio]

Estos controles te permitirán mover a tu personaje por el mundo, explorar los entornos, esquivar trampas y enfrentarte o evitar enemigos en tu búsqueda.

Estos son solo los controles *predeterminados*. Puedes personalizar completamente estas asignaciones de teclas a tu gusto desde el menú de **Opciones -> Controles**, como se describe en la sección de **UI**.

3.5. Mecánicas Principales

Para avanzar en el juego, deberás dominar estas mecánicas principales:

- **Movimiento:** Controla a tu personaje corriendo hacia la izquierda o derecha con las teclas de dirección. También puedes saltar usando la tecla asignada, lo cual te permitirá superar obstáculos y llegar a plataformas elevadas. El control preciso de estos movimientos es esencial.
- **Combate:** El combate se realiza exclusivamente con la tecla **Espacio**. Al pulsarla, se ejecuta una animación de ataque, que es tu única forma de combatir.
- **Plataformas:** El mundo del juego está lleno de plataformas que tendrás que dominar. Algunas son sólidas y fijas, otras flotan en el aire. Tendrás que medir bien tus saltos para moverte entre ellas, evitar caídas y explorar nuevas zonas.
- **Recolectables:** A lo largo de tu aventura encontrarás objetos especiales que representan tu vida. Recogerlos será vital para sobrevivir, ya que te permitirán recuperar salud o resistir más daño.

Dominar el movimiento, la navegación entre plataformas y la recolección de objetos será fundamental para superar todos los desafíos que encontrarás en este mundo peligroso.

4. Arte

Define cómo se ve y se siente todo el mundo, desde los escenarios hasta los personajes y enemigos, creando una atmósfera coherente.

4.1. Escenario

El juego *Campesino Místico* cuenta con **dos niveles principales**: BaseLevel y Final_level. Ambos utilizan el sistema **TileMap** de Godot Engine, lo que permite construir entornos modulares, optimizados y fáciles de modificar. El diseño sigue un enfoque por capas, donde cada capa tiene un propósito visual o funcional distinto.

Estructura General del Nivel

Cada nivel está compuesto por:

- Múltiples **TileMapLayers** para organizar el terreno, plataformas y decoraciones.
- **Nodos individuales** para elementos únicos como tiendas, cofres, enemigos o rocas.
- Objetos interactivos como Collectable_spawn, EntitySpawn, Diamante, y zonas de muerte (DeadZone).

Estilo Artístico y TileSets

- Estilo **píxel art** con enfoque medieval.
- Paleta de colores suaves, terrosos y verdes, que refuerzan el ambiente natural del bosque.
- Tamaño estándar de los tiles: **16x16 píxeles**.
- Cada capa usa su propio **TileSet** para separar funciones y facilitar la edición.

Además de los TileMaps, ambos niveles integran **nodos únicos** para enriquecer la escena.

Escenarios del Juego

- **BaseLevel:**
  - Primer nivel del juego.
  - Introduce al jugador en un entorno natural con plataformas simples y rutas claras.
  - Incluye generadores, rocas, cofres y estructuras básicas.
- **Final_level:**
  - Nivel más avanzado.
  - Presenta desafíos más complejos y variedad de plataformas.
  - Incluye árboles, escaleras, estructuras en varios niveles y ambientación más densa.
  - Contiene sistema de enemigos, zonas de jefe.

Utilizo el GameState para gestiona el flujo de transición entre niveles, encargándose de cargar nuevas escenas y limpiar las anteriores. Esta funcionalidad facilita el cambio fluido entre distintas etapas del juego, como pasar de los distintos niveles.

Gracias a esta estructura modular, añadir nuevos niveles en el futuro es sencillo, ya que solo se necesita agregar la nueva escena y llamarla desde el GameState, sin necesidad de modificar la lógica base del juego.

4.2. Personajes

El diseño e implementación de los personajes en *"Campesino Místico"* sigue un enfoque modular y reutilizable, utilizando una combinación de escenas base y componentes especializados para definir su comportamiento, apariencia y características.

4.2.1. Enemigo

Todos los enemigos del juego se construyen a partir de una escena base común que actúa como plantilla. Este enfoque modular garantiza uniformidad estructural, facilita la reutilización de componentes y permite una rápida creación y mantenimiento de distintos tipos de enemigos.

Escena Base del Enemigo (BaseEnemy): Esta escena funciona como clase abstracta o plantilla principal para todos los enemigos. Define una estructura estándar y componentes fundamentales que pueden ampliarse según las necesidades específicas de cada enemigo.

La construcción de los enemigos se basa en una escena base modular que organiza todos los elementos esenciales para su comportamiento y apariencia. La estructura típica es la siguiente:

- **Nodo Raíz (BaseEnemy)**: Un CharacterBody2D que actúa como contenedor principal de todos los componentes del enemigo.
- **AnimatedSprite2D**: Se encarga de representar gráficamente al enemigo y reproducir sus animaciones básicas (como estar quieto, caminar, recibir daño o morir).
- **CollisionShape2D**: Define el área de colisión física del enemigo, permitiendo la interacción adecuada con el entorno, ataques del jugador y otros objetos.
- **FlipHandler**: Un nodo o script personalizado que gestiona la inversión horizontal del AnimatedSprite2D, haciendo que el enemigo "mire" en la dirección correcta sin necesidad de crear animaciones separadas para izquierda y derecha.
- **GravityHandler**: Otro componente personalizado que se encarga de aplicar fuerza gravitatoria al enemigo. Esto sugiere un diseño flexible donde incluso las fuerzas físicas básicas son tratadas como módulos reutilizables, ideal para enemigos que no siempre heredan de CharacterBody2D.

Reutilización y Herencia

Cada enemigo hereda directamente de esta escena base o replica su estructura. Además, reutiliza scripts y nodos modulares, como FlipHandler, GravityHandler, entre otros. Esto permite:

- Crear nuevos enemigos de forma rápida.
- Mantener comportamientos comunes centralizados.
- Minimizar la duplicación de código.
- Escalar el proyecto de forma eficiente y ordenada.

Las escenas de los enemigos representan diferentes clases de enemigos, cada uno con comportamientos específicos como patrullar, atacar o seguir al jugador. Cada enemigo se implementa utilizando la estructura base BaseEnemy, combinada con componentes adicionales especializados que definen sus habilidades y reacciones. Este enfoque permite crear una amplia variedad de enemigos reutilizando y adaptando la misma base de forma eficiente y escalable.

Estructura

- **Componentes Específicos de Patrol**
  - **AIHandler**: Controla el movimiento, la lógica de patrullaje, el seguimiento del jugador u otras decisiones automáticas.
  - **HealthHandler**: Administra la salud del enemigo, incluyendo el registro de daño recibido.
  - **DeathHandler**: Gestiona la animación de muerte, la eliminación de la escena y la activación de recompensas.
  - **MovementHandler**: Se encarga del desplazamiento, aplicando velocidad y aceleración según lo indicado por la IA.
  - **RayCast2D**: Detectan obstáculos y márgenes del terreno para evitar caídas o colisiones.
  - **Hurtbox (Area2D)**: Define el área donde el enemigo puede recibir daño.
  - **HitboxHandler**: Si el enemigo puede atacar, este componente define su zona ofensiva.
  - **Drop_Handler**: Instancia objetos al morir, como monedas, salud u otros ítems

Todos los enemigos utilizan sprites en píxel art dentro del nodo AnimatedSprite2D, manteniendo una coherencia visual con el resto del juego, incluidos niveles, personajes y elementos interactivos.

4.2.2. Jefes

En cada escenario, el jugador debe enfrentarse y derrotar a un jefe obligatorio para poder avanzar. Si el jefe no es derrotado, el jugador no podrá progresar al siguiente nivel. Esta mecánica se implementa mediante un sistema de colisión que bloquea el camino hasta que el jefe ha sido vencido.

El bloqueo del camino está vinculado al estado del jefe y se representa visualmente mediante un efecto especial (por ejemplo, un muro con explosión animada). Una vez que el jefe es derrotado, se activa una animación o efecto (como explosiveEffect) que desactiva la colisión y permite al jugador continuar.

Este enfoque asegura que el jugador deba superar desafíos clave antes de avanzar, fortaleciendo el ritmo y la estructura del juego.

4.2.3. Protagonista (Jugador)

La implementación del personaje jugador (Player) se basa en un avanzado diseño orientado a componentes, complementado por una Máquina de Estados Finitos (FSM) que gestiona sus comportamientos y animaciones. Este enfoque garantiza una alta modularidad, facilita el mantenimiento del código y permite expandir fácilmente las capacidades del jugador a medida que evoluciona el juego.

Nodo Raíz

- Tipo: CharacterBody2D.
- Función: Gestiona el movimiento, las colisiones y la gravedad utilizando el sistema físico de Godot 2D.

Componentes Visuales y Físicos

- **AnimatedSprite2D**: Controla la representación gráfica del jugador, reproduciendo animaciones como caminar, saltar, caer, atacar y morir, basadas en el estado actual manejado por la FSM.

- **CollisionShape2D**: Define la forma física del jugador para detectar colisiones con el entorno (suelo, paredes, plataformas).

Contenedor de Componentes

Agrupa scripts especializados que encapsulan funcionalidades específicas, siguiendo un diseño altamente modular:

- **InputHandler:** Captura y procesa las entradas del jugador (teclado, mando) y comunica intenciones como "moverse" o "saltar" a otros componentes o a la FSM.
- **MovementHandler:** Gestiona el movimiento horizontal (caminar/correr) aplicando velocidad y aceleración basadas en la entrada recibida y el estado actual.
- **JumpHandler:** Controla la lógica de salto, aplicando la fuerza vertical cuando el jugador ejecuta un salto permitido.
- **FlipHandler:** Voltea horizontalmente el jugador la dirección del movimiento.
- **GravityHandler:** Aplica gravedad de manera personalizada al jugador.
- **HitBoxHandler:** Gestiona las áreas de daño del jugador (si realiza ataques activos).
- **HealthHandler:** Administra la salud del jugador, aplicando daño o curación según las interacciones.
- **DeathHandler:** Se activa cuando la salud llega a cero, controlando la animación de muerte y gestionando eventos como el Game Over.
- **CollectionHandler:** Detecta la recolección de objetos (por ejemplo, ítems de curación) y aplica sus efectos.
- **StompBoxHandler:** Maneja el área de pisotón, permitiendo derrotar enemigos al caer sobre ellos, e iniciando los estados de rebote y pisotón.

Máquina de Estados Finitos (FSM)

La FSM organiza y gestiona el comportamiento y las transiciones del jugador entre diferentes estados:

PlayerState (nodo principal): Coordina los estados activos e implementa las transiciones basadas en entradas o condiciones.

Estados hijos:

- PlayerIdleState: El jugador está quieto en el suelo.

- PlayerWalkState: El jugador camina o corre.
- PlayerJumpState: El jugador asciende tras saltar.
- PlayerFallState: El jugador cae tras un salto o caída libre.
- PlayerStompState: Estado breve tras pisar exitosamente a un enemigo.
- PlayerBounceState: Estado que gestiona el rebote posterior al pisotón.

Este enfoque modular permite ampliar o modificar el comportamiento de forma sencilla, manteniendo el código limpio, escalable y coherente con el estilo visual píxel art del resto del juego.

4.2.4. NPC

Los NPCs (personajes no jugables) en este proyecto están diseñados con un enfoque modular, permitiendo tanto comportamientos autónomos como interacción con el jugador. Cada NPC se compone de múltiples scripts y nodos que controlan su movimiento, estado, animaciones y sistema de diálogo. A continuación, se describe detalladamente cómo funciona.

Cada NPC se instancia a partir de una escena (NPC) que hereda de CharacterBody2D. En su nodo raíz contiene:

- la representación visual y animaciones
- Un recurso (NpcResource) que define su nombre, sprite, retrato y datos de audio para el diálogo.
- Un nodo de comportamiento (NpcBehaviour) que puede ser del tipo NpcBehaviourWander o NpcBehaviourPatrol
- Un nodo DialogInterraction que gestiona el sistema de interacción y diálogo

Al iniciar la escena, el NPC configura su apariencia según el recurso asignado y emite una señal que activa su comportamiento.

Comportamiento de los Npcs

El sistema implementa dos tipos principales de comportamiento para los NPCs:

- **NpcBehaviourWander**: El NPC se desplaza aleatoriamente hacia la izquierda o derecha dentro de un rango predefinido. Alterna entre los estados *Idle* (quieto durante un tiempo aleatorio) y *Walk* (movimiento breve en una dirección). Si

sobrepasa el límite de su rango, invierte automáticamente la dirección. Este ciclo se repite indefinidamente mientras el jugador no interactúe con él.

- **NpcBehaviourPatrol**: El NPC recorre una ruta fija compuesta por nodos PatrolLocation. Al alcanzar cada punto, se detiene durante un tiempo de espera personalizado antes de continuar hacia el siguiente. La patrulla es cíclica y se ejecuta de forma continua mientras el NPC permanezca activo.

Interacción y Diálogo

La interacción con el jugador está gestionada por el nodo DialogInterraction, que utiliza un Area2D para detectar cuándo el jugador se aproxima. Al entrar en el área, se activa una animación visual (por ejemplo, un icono flotante). Si el jugador pulsa la tecla de acción (ui_accept), se inicia el sistema de diálogo.

Durante la conversación:

- El NPC detiene automáticamente su comportamiento autónomo.
- Se muestra una ventana de diálogo que incluye el nombre del personaje, su retrato y el texto correspondiente.
- El jugador puede avanzar el diálogo mediante teclas, y si se configura, también se pueden mostrar elecciones interactivas a través del nodo DialogChoice.

Este sistema modular permite crear fácilmente NPCs con comportamientos variados y diálogos personalizados sin duplicar código. Además, su integración con el sistema de diálogo contribuye a enriquecer la narrativa y la inmersión del jugador en el mundo del juego.

4.3. Coleccionables

En *Campesino Místico*, los objetos coleccionables son muy importantes. Algunos le dan vida al jugador. Otros solo son objetos sin efecto de salud y otros se usan para cosas importantes como comercio o progreso. El sistema implementado permite tanto la colocación manual en el nivel como la generación dinámica mediante spawners.

Generador de Vida

Esta escena actúa como un generador dinámico de objetos coleccionables durante el juego. Permite instanciar coleccionables de forma manual o automática de tiempo definidos.

Estructura

- **Nodo Raíz**: Es el contenedor principal de la escena y gestiona toda la lógica del spawner a través del script collectable_spawn.gd.
- **SpawnTimer (Timer)**: Nodo Timer que controla la frecuencia de generación de los coleccionables.

Funcionalidad

- Cada vez que el temporizador se activa, elige un punto al azar.
- Crea un objeto según la escena definida (Coin Scene).
- El objeto aparece y espera a ser recogido.

Escena Base de Coleccionable

Funciona como una escena base o abstracta para todos los tipos de coleccionables del juego. Aunque visualmente vacía, su script asociado, collectable.gd, contiene la lógica común para gestionar el comportamiento de los coleccionables.

Gracias a este diseño, es fácil reutilizar y extender este comportamiento para diversos tipos de coleccionables, asegurando coherencia y simplificando la creación de nuevos objetos interactivos en el juego.

4.3.1. **Coleccionable Específico: Moneda Base (base_coin)**

Representa un tipo concreto de coleccionable, un corazon que otorga vida. Esta es la escena que instancia el Collectable_spawn.

Estructura

- **Nodo Raíz (RigidBody2D):** Utiliza un RigidBody2D, lo que significa que la moneda interactúa con el motor de físicas del juego. Esto le otorga propiedades físicas como masa, gravedad (controlada por el parámetro Gravity Scale) y rebote

(con un valor de Bounce = 0.4), lo que hace que la moneda caiga, rebote y se comporte de manera natural cuando se genera en el juego.

- **AnimatedSprite2D:** Muestra la imagen animada de la moneda.
- **CollisionShape2D (hijo de RigidBody2D y Area2D):** Define la forma física de la moneda para detectar colisiones con el entorno, como el suelo, paredes y plataformas. La forma puede ser un círculo o una caja dependiendo del diseño de la moneda.
- **Area2D:** Componente clave para la detección de la recolección de la moneda. Este nodo se utiliza para detectar cuándo el jugador entra en contacto con la moneda. También usa una colision2d

4.3.2. Ítems

El juego cuenta con distintos ítems que el jugador puede recoger. Algunos tienen efectos especiales, como la restauración de vida, mientras que otros no afectan la salud, pero aun así pueden ser recolectados y usados más adelante en el juego.

Tipos de Objetos y Efectos

Cada objeto puede tener un efecto diferente. Algunos restauran salud, otros simplemente se recogen y se guardan en el inventario.

- **Apple**: Restaura 2 puntos de vida
- **Poción**: Restaura vida completa
- **Rock**: No tiene efecto asignado (en el futuro sí)
- **Gem**: No tiene efecto asignado (en el futuro sí)

También se puede reproducir un sonido cuando el jugador lo recoge.

4.3.3. Diamante

El diamante es un objeto coleccionable que el jugador puede recoger, pero **no restaura vida** ni tiene efectos inmediatos. Su función es **acumularse en el inventario** y, en el futuro, se usará como moneda o recurso para **comprar objetos** o **desbloquear mejoras** dentro del juego

4.3.4. Cofre

El juego también incluye una escena llamada Treasure, la cual representa un objeto especial. Es un ítem coleccionable que puede contener recompensas, como diferentes tipos de ítems que el jugador obtiene al interactuar con él.

Estructura de la Escena

- **Area2D:** Detecta cuándo el jugador entra en contacto con el cofre.
- **AudioStreamPlayer2D:** Reproduce un sonido, por ejemplo, al abrir el cofre o recoger su contenido.
- **Label:** Puede mostrar información adicional, como el nombre del ítem o la cantidad de recompensa.
- **Audio:** Controla las animaciones del cofre, como abrirse o desaparecer después de ser recogido.

4.3.5. DestructableBox

Es una caja que se puede romper si recibe golpes de objetos que tienen una lógica llamada HitBoxHandler. Tiene una cantidad de vida (max_health). Cada vez que la golpean, pierde vida. Cuando su vida baja a cero, la caja se destruye.

Al destruirse:

- Reproduce una aniamcion llamado destroy.
- Aparecen varios diamantes (según num_diamantes) que saltan con un efecto de animación.
- Luego de un momento, la caja desaparece de la escena.

4.4. Colisiones

Para gestionar correctamente las interacciones físicas entre los distintos elementos del juego, se ha utilizado el sistema de **Physics Layers** y **Collision Masks** de Godot. Cada tipo de objeto está asignado a una capa específica, lo que permite definir con precisión qué objetos deben detectarse entre sí y cuáles no.

Layer	Etiqueta	Uso
Layer 1	floor	Suelos, plataformas, paredes y otros elementos del entorno.
Layer 2	player	Jugador principal.
Layer 3	enemy	Todos los enemigos del juego.
Layer 4	collectable	Objetos colectables como monedas, vida, ítems, etc.

Máscaras de Colisión

Cada objeto en el juego tiene su **Collision Mask** configurada para detectar únicamente lo que necesita. A continuación, se describen los casos principales:

- **Jugador**: El jugador está asignado a la **Layer 2**, permitiéndole ser detectado por otros elementos como enemigos o recolectables. Su máscara de colisión está configurada para detectar:
  - **Layer 1 (floor)**: para caminar sobre el terreno.
  - **Layer 3 (enemigo)**: para detectar y reaccionar ante enemigos.
  - **Layer 4 (colectable)**: para recoger objetos como monedas, vida u otros ítems.
- **Enemigos**: Todos los enemigos están asignados a la **Layer 3**, lo que permite identificarlos fácilmente como entidades hostiles. Su máscara de colisión les permite detectar:
  - **Layer 1 (floor)**: para moverse correctamente sobre plataformas y detectar colisiones con el entorno.
  - **Layer 2 (player)**: si deben reaccionar ante el jugador, ya sea para atacarlo o colisionar con él.

- **Colectables**: Los objetos colectables están asignados a la **Layer 4**, reservada para ítems interactivos como monedas, vidas, mejoras, etc. Solo detectan:
    - **Layer 2 (player)**: para activarse cuando el jugador entra en contacto con ellos.
- **Entorno**: Los elementos del entorno, como suelos y paredes, están en la **Layer 1**. No tienen máscara de colisión, ya que no necesitan detectar otros objetos; únicamente sirven como superficie física para que otros elementos interactúen con ellos.

4.5. Trampas

Actualmente, el juego cuenta con una única trampa: la **DeadZone**. Esta zona está colocada fuera del área jugable, generalmente bajo el terreno o en los extremos del escenario, y su función principal es detectar si el jugador ha caído fuera de los límites del nivel.

Cuando el jugador entra en esta zona, el nivel se reinicia automáticamente. Este mecanismo garantiza que el juego no quede bloqueado ni en un estado inválido, ofreciendo una recuperación inmediata y controlada en caso de caída.

Aunque por ahora solo existe esta trampa, el sistema está preparado para permitir la incorporación de nuevas trampas en el futuro, ampliando así las posibilidades de diseño y desafío del juego.

5. Estrategia de Distribución y publicación

5.1. Estado Actual

Actualmente, el juego no ha sido distribuido en ninguna plataforma pública. Sin embargo, se dispone de un ejecutable funcional generado por el motor Godot, lo que permite ejecutar y probar el juego localmente en entornos de escritorio compatibles (principalmente Windows y Linux).

Esta versión sirve como base para pruebas internas, iteraciones técnicas y preparación para una futura publicación.

5.2. Publicación Inicial

Se prevé una primera distribución gratuita a través de la plataforma **Itch.io**, debido a su facilidad de uso tanto para desarrolladores como para jugadores. Se contemplan las siguientes opciones:

- **Versión ejecutable** para Windows y Linux, disponible como descarga directa.
- **Versión HTML5** para juego en navegador, si se habilita la exportación correspondiente en Godot.

Además, se utilizarán los mecanismos de interacción que ofrece Itch.io para recibir comentarios de los jugadores, lo cual permitirá mejorar las futuras versiones del juego.

5.3. Expansión a Steam

Si el juego alcanza los objetivos de calidad y visibilidad, se considera su publicación en **Steam**. Esto implicará:

- Registro en **Steamworks** como desarrollador (pago único de 100 USD).
- Preparación de materiales promocionales (capturas, tráiler, descripción).

5.4. Extensión a Dispositivos Móviles

A mediano o largo plazo, se contempla una posible expansión a dispositivos móviles con sistemas **Android** y **iOS**. Esto requerirá:

- Adaptación del juego a interfaces táctiles.
- Exportación desde Godot a plataformas móviles (Android/iOS).
- Cumplimiento de los requisitos de publicación en **Google Play Store** y **Apple App Store**.

6. Interfaz de Usuario (UI)

En esta sección se describe cómo es la interfaz gráfica del juego y cómo se espera que el jugador interactúe con ella.

6.1. Descripción de la interfaz

La interfaz de Campesino Místico fue pensada para ser fácil de usar y agradable visualmente. Se construyó con los Nodos de Control de Godot Engine, lo que permite organizar bien todo lo que el jugador ve en pantalla.

El diseño tiene un estilo medieval, que va muy bien con la historia del juego. Se usaron imágenes que parecen pergaminos antiguos y madera para los botones y paneles. Además, la tipografía elegida tiene un aire clásico, parecida a la que se ve en libros antiguos.

Navegar por los menús es muy sencillo. Todo está ordenado para que el jugador entienda al instante qué hacer. Hay botones para empezar el juego, ver las opciones o salir, como en muchos otros juegos. También se cuidó que todo funcione bien en distintas resoluciones, y que los botones reaccionen cuando se pasa el cursor encima, dando una señal clara de que se pueden pulsar.

En resumen, es una interfaz clara, estética y fácil de usar, que ayuda a que el jugador se conecte rápidamente con el mundo del juego.

Menú Principal:

Cuando se inicia el juego *Campesino Místico*, aparece una pantalla con una imagen de fondo que representa un pueblo medieval, lo que ayuda a que el jugador entre en el ambiente del juego desde el primer momento.

En el centro de la pantalla se muestra el título del juego, **"Campesino Místico"**, con una letra grande y decorativa que llama la atención. Justo debajo del título, se encuentran las opciones principales del menú: por ejemplo, *Jugar*, opciones y *Salir*. Estas opciones están colocadas una debajo de la otra, usando un **VBoxContainer**, lo que hace que todo se vea ordenado y fácil de entender.

El jugador puede usar el ratón o el teclado para moverse por el menú y seleccionar lo que desea hacer, de forma clara y sin complicaciones.

- **Start Game:** Botón para iniciar una nueva partida o continuar la existente.
- **Options:** Botón para acceder al menú de configuración.

- **Exit:** Botón para cerrar la aplicación del juego.

Usabilidad: El diseño es limpio y directo. Los botones son grandes, claramente etiquetados y fáciles de seleccionar con el ratón. La jerarquía visual guía al usuario hacia las acciones más comunes (empezar a jugar).

Menú de Opciones:

Al hacer clic en **"Options"** desde el menú principal, el jugador accede a una nueva pantalla donde puede ajustar varias configuraciones del juego.
En el centro de la pantalla hay un panel con fondo de **textura de pergamino**, que mantiene el estilo medieval del juego. En la parte superior de este panel aparece el título **"Options"** con una fuente clara y decorativa.
Las distintas opciones están organizadas en **pestañas** usando un **TabContainer.**
Esto permite dividir la configuración en tres secciones bien diferenciadas:

- **Sounds**: Para ajustar el volumen de la música y los efectos.
- **Screen**: Para cambiar la resolución o poner el juego en pantalla completa.
- **Controls**: Para modificar las teclas o botones del juego.

En la parte inferior del panel hay un botón **"Exit"**, que permite volver fácilmente al menú principal sin guardar cambios complicados.

Pestañas de Configuración:

- **Screen (Pantalla):** Esta pestaña contiene controles para modificar la Resolución de pantalla (permitiendo elegir entre varias opciones predefinidas o comunes) y el Modo de Pantalla (Pantalla Completa, Ventana, Ventana sin bordes)
- **Sonidos:** Aquí el usuario puede ajustar los niveles de volumen generales, de la música y de los efectos de sonido (SFX) mediante deslizadores.
- **Controles:** Esta pestaña muestra una lista de las acciones del juego y la tecla o botón asignado actualmente ("Action_key"). Permite al usuario personalizar los controles, asignando nuevas teclas a cada acción. Se utiliza un ScrollContainer para asegurar que todas las acciones sean visibles incluso si son muchas. La disposición usa HBoxContainer para alinear la descripción de la acción y el botón.

Usabilidad: El uso de pestañas organiza eficazmente la gran cantidad de opciones, evitando una pantalla sobrecargada. La personalización de controles es una característica de accesibilidad y preferencia importante. El botón "Exit" proporciona una salida clara.

Persistencia de Configuración:

Un punto importante en la pantalla de opciones es que todos los cambios que hace el usuario se guardan automáticamente. Esto incluye cosas como la resolución, el modo de pantalla (pantalla completa o ventana), el volumen del sonido y los controles del juego.

Para lograr esto, se ha usado un **script AutoLoad** (también llamado **Singleton**) en Godot. Este script está siempre activo mientras el juego se ejecuta, y se encarga de lo siguiente:

- Cargar las configuraciones guardadas cuando se inicia el juego.
- Aplicar esos valores a los sistemas correspondientes: gráficos, sonido y controles.
- Escuchar las señales que envían los botones, deslizadores y selectores de la interfaz cuando el jugador cambia alguna opción.
- Guardar automáticamente los nuevos valores en un archivo de configuración.

Este archivo se guarda en el nivel del proyecto, no a nivel global del sistema, utilizando la ruta **user://settingsData.save**. Esto significa que la configuración se guarda solo para este juego, sin afectar a otros programas ni compartir datos entre distintos juegos.

Gracias a este sistema, el jugador no necesita hacer clic en ningún botón de "Guardar". Cada vez que cambia algo, se guarda al instante. Así, si cierra el juego y lo vuelve a abrir más tarde, todo estará tal como lo dejó, lo que mejora mucho la experiencia.

6.2. Manual de usuario

Esta guía rápida describe cómo utilizar los menús del juego "Campesino Místico".

Menú Principal:

- Al iniciar el juego, te encontrarás en el Menú Principal.
- Haz clic en Start Game para comenzar a jugar.
- Haz clic en Options para ajustar la configuración del juego.
- Haz clic en Exit para cerrar el juego.

Menú de Opciones:

Tras hacer clic en Options, accederás al menú de configuración.

- **Navegación por Pestañas**: Haz clic en las pestañas en la parte superior del panel ("Sounds", "Screen", "Controls") para cambiar entre las diferentes categorías de ajustes.
- **Ajustar Configuración de Pantalla (Pestaña Screen):**
  - Busca las opciones de "Resolución" y "Modo de Pantalla".
  - Utiliza los menús desplegables o botones para seleccionar la resolución y el modo de pantalla deseados (Pantalla Completa / Ventana).
- **Ajustar Sonido (Pestaña Sounds):**
  - Localiza los deslizadores o campos para "Volumen General", "Música" y "Efectos de Sonido".
  - Arrastra los deslizadores o introduce valores para ajustar el volumen a tu gusto.
- **Cambiar Controles (Pestaña Controls):**
  - Verás una lista de acciones y sus teclas asignadas.
  - Para cambiar una tecla, haz clic en el botón que muestra la tecla actual junto a la acción deseada.
  - El juego esperará a que presiones una nueva tecla. Presiona la tecla que deseas asignar a esa acción.
- **Guardado de Cambios:** No necesitas buscar un botón de "Guardar". **Todos los cambios que realices en el menú de opciones se guardan automáticamente** en el sistema en cuanto los modificas, gracias al sistema AutoLoad implementado.
- **Salir de Opciones:** Cuando hayas terminado de ajustar la configuración, haz clic en el botón **Exit** en la parte inferior del panel de opciones para regresar al Menú Principal. Tus cambios ya estarán guardados y aplicados.

7. Implementación del Videojuego

La implementación de *Campesino Místico* se llevó a cabo utilizando Godot Engine **4**, una plataforma de desarrollo de código abierto especializada en videojuegos 2D. El proyecto fue estructurado aplicando principios de modularidad y reutilización de componentes, aprovechando la arquitectura basada en nodos característica del motor.

Durante el proceso se aplicaron metodologías de desarrollo ágil, control de versiones con Git y prácticas de testing continuo, lo que permitió garantizar un rendimiento sólido y una experiencia de usuario consistente.

7.1. Motor de desarrollo

El videojuego fue desarrollado con Godot Engine 4.4, un motor de código abierto orientado al desarrollo de videojuegos 2D y 3D. Su sistema basado en nodos proporciona una estructura jerárquica clara y modular que facilita la creación de interfaces gráficas, lógica del juego y efectos visuales de manera eficiente. La arquitectura de Godot permite encapsular funcionalidades específicas en escenas reutilizables, lo que resulta ideal para un enfoque de desarrollo iterativo y escalable.

7.2. Lenguaje de programación

Toda la lógica del juego fue implementada en **GDScript**, el lenguaje de scripting nativo de Godot. Con una sintaxis similar a Python, GDScript ofrece una curva de aprendizaje rápida y una integración directa con el motor. Su enfoque en la simplicidad y legibilidad del código permitió una rápida iteración durante las etapas de desarrollo y prototipado, a la vez que facilitó la depuración y el mantenimiento del proyecto.

7.3. Metodología de trabajo

Durante el proceso de desarrollo se aplicaron metodologías de trabajo **ágil**, con ciclos de desarrollo cortos y objetivos concretos por cada iteración. Se utilizó **Git** como sistema de control de versiones, permitiendo un flujo de trabajo ordenado, con ramas diferenciadas por tareas y checkpoints funcionales.

Se adoptaron prácticas de **testing continuo**, asegurando la estabilidad del juego después de cada implementación importante. Las pruebas se realizaron de forma manual en cada nivel, validando la integración de sistemas como la salud, el combate, el respawn y la progresión entre escenas.

7.4. Cámara del jugador

La cámara principal del juego está controlada por un nodo PlayerCamera, una clase personalizada que extiende Camera2D. Este componente sigue al jugador con un movimiento suavizado, evitando cambios bruscos durante la navegación por el escenario.Características principales del sistema de cámara:

- **Seguimiento suave**: Mediante interpolación (lerp) hacia la posición del jugador, configurable con una propiedad follow_speed
- **Offset personalizable:** Se puede ajustar el desplazamiento de la cámara respecto al jugador (camera_offset), ideal para juegos con orientación lateral o plataformas verticales.
- **Activación dinámica**: La cámara se conecta automáticamente al sistema mediante SignalBus, activándose cuando el jugador está listo (on_player_ready).
- **Sacudida de cámara:** Durante eventos especiales, como explosiones o la muerte de un jefe se puede activar un efecto de sacudida de cámara (camera shake) mediante la señal *on_camera_shake*, generando una vibración visual breve que añade intensidad e inmersión.

Este sistema modular permite reutilizar la misma lógica de cámara en cualquier escena con mínima configuración, manteniendo coherencia visual en todo el juego.

8. Aspectos destacables de desarrollo

8.1. Innovaciones

8.1.1. Sistema modular basado en handlers

Uno de los pilares fundamentales del proyecto ha sido el diseño modular mediante handlers reutilizables. Se crearon scripts individuales para gestionar funcionalidades específicas del personaje y enemigos, como el movimiento, la gravedad, la muerte, la recolección de objetos, entre otros.

Cada handler está compuesto por un script .gd y su escena correspondiente. tscn, permitiendo que los comportamientos se asignen de forma flexible y manteniendo una separación clara de responsabilidades. Esta estructura ha mejorado significativamente la escalabilidad y el mantenimiento del código.

8.1.2. Checkpoint y sistema de respawn

Se implementó un sistema de checkpoint mediante una escena Area2D con un Marker2D como punto de respawn. Cuando el jugador entra en la zona de colisión, se

actualiza automáticamente su posición de respawn, permitiendo que, tras morir, reaparezca en el último punto alcanzado en lugar de reiniciar el nivel desde el principio.

8.1.3. Portal de transición

Para gestionar la transición entre niveles o escenas, se utilizó un nodo Area2D que emite una señal personalizada a través de un sistema centralizado (SignalBus). Esto facilita la comunicación entre nodos sin acoplamiento directo y mejora la claridad del flujo del juego.

8.1.4. Sistema de recursos personalizados

El uso de recursos personalizados (Resource) ha sido clave para definir datos reutilizables de forma centralizada. Por ejemplo, cada NPC utiliza un NpcResource que contiene su nombre, retrato, frames de animación y audio. Esto permite configurar múltiples NPCs con distintos diálogos y apariencias sin necesidad de duplicar lógica o escenas.

8.1.5. Uso de iconos personalizados en escenas

Para mejorar la organización visual dentro del editor de Godot, se utilizó la anotación @icon en scripts clave, lo que permite asignar un icono personalizado a las escenas y nodos en el árbol de nodos.

El uso de iconos es una herramienta útil para mantener claridad visual y acelerar el trabajo con escenas grandes o con múltiples instancias similares.

Se ha hecho un uso intensivo de iconos personalizados, etiquetas y estructura de carpetas para mejorar la experiencia de desarrollo en el editor de Godot. Esto ha facilitado la organización y la rápida localización de scripts o escenas específicas, algo especialmente útil en proyectos en expansión.

8.1.6. Cofres del tesoro interactivos

Se desarrolló un sistema de cofres reutilizable mediante el nodo TreasureChest, que permite otorgar objetos al jugador de forma visual y controlada. Cada cofre contiene un recurso externo (ItemData) que define el objeto y su textura, así como la cantidad correspondiente. Este diseño modular permite configurar rápidamente diferentes tipos de cofres sin duplicar lógica.

Cuando el jugador entra en el área de interacción del cofre, se muestra un aviso en pantalla y se activa la posibilidad de abrirlo. Al pulsar la tecla de acción, el cofre se abre mediante una animación y entrega su contenido al inventario del jugador.

Este sistema se integra perfectamente con el inventario y refuerza la experiencia de exploración y recompensa dentro del juego.

8.1.7. Sistema de guardado de configuración

Se desarrolló un sistema de guardado y carga de configuraciones mediante archivos cifrados. Utiliza un archivo local (SettingsData.save) y el nodo SettingSignalBus para enviar y recibir datos entre sistemas. La información se guarda como un diccionario en formato JSON y se cifra con contraseña usando FileAccess.open_encrypted_with_pass, garantizando mayor seguridad.

Al iniciar el juego, los datos se cargan automáticamente si existen. Cualquier cambio en la configuración se guarda de inmediato mediante señales, asegurando que las preferencias del jugador (como controles, volumen, etc.) se mantengan entre sesiones.

8.1.8. Uso de Autoloads para sistemas globales

El proyecto hace uso de autoloads para centralizar funcionalidades clave que deben estar accesibles desde cualquier parte del juego. Estos scripts se cargan automáticamente al iniciar la aplicación y se mantienen activos en todo momento.

Gracias a este enfoque, se consigue una arquitectura limpia, organizada y fácil de mantener, donde los datos y eventos globales están siempre disponibles sin necesidad de pasar referencias manualmente.

8.1.9. Sistema de estados del jugador (FSM)

Para gestionar los distintos comportamientos del jugador de forma organizada y escalable, se implementó una máquina de estados finitos (FSM). Cada estado del jugador como caminar, saltar, atacar o morir está representado por su propio script que hereda de una clase base (BasePlayerState). Estos estados se almacenan dentro del directorio playerstates.

Los cambios de estado son gestionados por el nodo FiniteState, que coordina transiciones fluidas según la lógica de juego. Esto evita condicionales excesivos dentro del nodo del jugador y facilita el mantenimiento del código.

8.1.10. Sistema de Efectos Visuales con Partículas (CPUParticles2D)

Se utilizaron CPUParticles2D para añadir efectos visuales que enriquecen la experiencia del juego. Los usos principales fueron:

- **Lluvia ambiental**: Utilizado para crear atmósfera en escenarios exteriores. Simula gotas de lluvia constantes con movimiento vertical y dispersión suave.
- **Daño recibido (BloodEffect)**: Al recibir daño, se activa un efecto visual de sangre mediante partículas rojas con rebote, dirección aleatoria y autodestrucción tras unos segundos.
- **Explosión Visual**: Muestra una explosión rápida con muchas partículas que salen en todas las direcciones. Se usa cuando algo explota, como un enemigo, una trampa o un objeto que se destruye. El efecto aparece y desaparece solo.
- **Fuego**: Hace que se vea fuego. Es un efecto que no se apaga y se usa para cosas como antorchas o fogatas. Las partículas suben y cambian de color, como una llama de verdad

Cada efecto se encapsula en una escena individual que puede instanciarse dinámicamente. Esto permite reutilizar los efectos y mantener el proyecto organizado. Los nodos de partículas están configurados con temporizadores internos para autodestruirse tras su ejecución, evitando consumo de recursos innecesario

8.1.11. Sistemas de aparición (Spawners)

El juego cuenta con dos sistemas de spawneo automatizado que gestionan dinámicamente la creación de enemigos y objetos coleccionables en tiempo de ejecución:

- **EntitySpawn:** Este sistema gestiona la aparición de enemigos y del jugador. Al comenzar el nivel, se instancia el jugador en un punto de generación específico. Los enemigos, por su parte, son generados de forma controlada a partir de una lista de puntos de aparición (Marker2D), utilizando un temporizador que regula la frecuencia.

- **CollectableSpawn:** Este sistema se encarga de generar objetos coleccionables (como monedas) en puntos aleatorios definidos en la escena. Utiliza también un temporizador con intervalos variables para mantener una aparición dinámica e impredecible. Los objetos instanciados se colocan dentro de un contenedor global de entidades y su posición se selecciona aleatoriamente entre los puntos disponibles.

8.1.12. Sistema de Créditos con Efectos Visuales Dinámicos

Para enriquecer la experiencia final del juego, se diseñó un sistema de créditos interactivo, utilizando un **RichTextLabel** con efectos visuales animados aplicados al texto para mostrar texto con **BBCode**. Este sistema se encuentra implementado en la escena **credit.tscn** y hace uso de múltiples scripts modulares para aplicar transformaciones visuales al contenido mostrado.

El contenido del crédito se almacena en un archivo externo **credits_text.tx**t ubicado en el mismo directorio. Se carga en tiempo de ejecución a través de un script personalizado **TextEffectsLoader** que se encarga también de aplicar los efectos visuales correspondientes.

- **Sistema de Efectos Visuales:** El sistema de efectos se compone de módulos individuales, cada uno diseñado para alterar la presentación del texto mediante animaciones o transformaciones visuales. Estos efectos son:
  - **FadeTimedEffect**: Hace que las letras aparezcan o desaparezcan poco a poco.
  - **FireEffect**: Hace que el texto brille como si tuviera fuego o luz parpadeante.
  - **GlitchEffect**: Crea un efecto de error visual, como si el texto estuviera dañado o fallando.
  - **ShakeEffect**: Hace que las letras tiemblen o se muevan rápido hacia los lados.
  - **WaveEffect**: Mueve las letras en forma de ola, una tras otra.
  - **TypewriterEffect**: Muestra el texto como si se estuviera escribiendo en una máquina de escribir, letra por letra.

Los efectos se aplican automáticamente a partir de las etiquetas definidas en el archivo de texto, permitiendo una edición flexible y centralizada sin necesidad de modificar la escena manualmente.

8.1.13. Sistema de Gestión de Sonido con AudioManager y Buses

Para centralizar y optimizar el control del sonido en *Campesino Místico*, se ha implementado un nodo AudioManager, el cual actúa como sistema global para gestionar el audio de música, efectos de sonido (SFX) y volumen maestro. Este componente aprovecha la arquitectura de buses de audio de Godot Engine y se integra con el menú de opciones para ofrecer una experiencia auditiva coherente y configurable.

- **AudioManager**: Nodo de tipo Node cargado como Autoload (singleton), lo que permite su acceso desde cualquier parte del juego.
- **AudioStreamPlayer**: Usado internamente para reproducir música y sonidos globales, como ambiente o efectos del menú.
- **Bus Layout personalizado**: Configurado desde el panel de audio de Godot, con al menos tres buses:
    - **Máster**: Bus principal que controla todo el audio del juego.
    - **Music**: Bus independiente para la música de fondo.
    - **SFX**: Bus exclusivo para efectos de sonido (combate, salto, daño, etc.).

8.2. Pruebas y Validación

Durante el desarrollo del videojuego *Campesino Místico*, se implementaron distintos métodos de validación para garantizar la funcionalidad y estabilidad del sistema:

- **Pruebas manuales funcionales**:
    - Se verificó manualmente cada escena (niveles, menús, jefes, etc.) tras cada iteración importante del desarrollo.
    - Se comprobó el correcto funcionamiento de los sistemas de combate, recolección de objetos, interacción con NPCs, y mecánicas de salto y movimiento.
- **Pruebas de integración**:
    - Se realizaron pruebas de transición entre escenas usando los portales de transición para validar que el GameState gestionara correctamente la carga y descarga de recursos.
    - Se probó la persistencia de datos de configuración entre sesiones de juego.
- **Pruebas unitarias**:

    - Aunque Godot no cuenta con un sistema de pruebas unitarias formal como otros entornos, se implementaron scripts de prueba para verificar el funcionamiento aislado de handlers críticos como HealthHandler, MovementHandler y CheckpointManager.
- **Control de errores**:
    - Se utilizó el monitor de errores y el debugger interno de Godot para capturar excepciones en tiempo de ejecución, especialmente durante interacciones con el sistema de inventario y el sistema de partículas.

8.3. Futuras Líneas de Desarrollo

A continuación, se detallan algunas propuestas de mejora para enriquecer la experiencia de juego, con especial énfasis en las áreas multijugador y el sistema de karma, así como otras funcionalidades complementarias.

8.3.1. Mejora del Sistema Multijugador

- **Sincronización de pantallas:** Para el modo multijugador se propone implementar un sistema de sincronización en tiempo real entre las pantallas de ambos jugadores. Esto asegurará que las acciones, movimientos, enemigos y eventos se vean de forma idéntica en los dos dispositivos, sin desajustes. Se sincronizarán constantemente variables como posición, animaciones, ataques, vida y estados del entorno.
- **Conexión en línea:** Además, se integrará una conexión en línea que permita que dos jugadores se conecten desde diferentes lugares por internet.

8.3.2. Evolución del Sistema de Karma

Profundizar en la narrativa moral y sus consecuencias en el mundo del juego.

- **Registro moral detallado**: Implementación de un sistema que clasifique las acciones del jugador (buenas, neutrales o malas), asignando valores acumulables.
- **Reputación regional**: Introducción de un karma localizado por zonas, modificando la actitud de los NPCs en función de la reputación.

- **Finales más variados:** Se propone añadir nuevos tipos de finales además del final bueno o malo. Estos finales estarán basados en combinaciones específicas de decisiones que el jugador haya tomado a lo largo del juego. Esto permitirá que la historia tenga desenlaces más personalizados y no solo dos opciones fijas.

8.3.3. Otras Funcionalidades en Consideración

- **Inventario avanzado**: Clasificación por tipo, peso y rareza, con un sistema de fabricación basado en materiales recolectados.
- **Enemigos y jefes mejorados**: IA más compleja que reaccione al karma del jugador; jefes con múltiples fases según decisiones previas.
- **Sistema de guardado de estado del jugador:** Se plantea implementar una funcionalidad que permita guardar automáticamente el estado completo del jugador al salir del juego. Esto incluye recursos recogidos, salud, ubicación, nivel actual, karma acumulado y cualquier otro progreso relevante. De este modo, al volver a iniciar el juego, el jugador podrá continuar exactamente desde donde lo dejó, sin perder avances ni tener que repetir secciones ya superadas.
- **Sistema de traducción**: Se intentó implementar un sistema de traducción para soportar múltiples idiomas, pero no fue posible completarlo por falta de tiempo. Está previsto integrarlo en futuras versiones del proyecto.

8.4. Línea de Tiempo del Desarrollo

La siguiente orden resume las fases clave del desarrollo del proyecto:

Fase	Periodo	Actividades realizadas
Diseño conceptual	Marzo	Definición de historia, ambientación, mecánicas y storyboard inicial

Estructura técnica inicial	Abrir	Configuración del proyecto en Godot, creación de FSM, y sistema modular de handlers
Sistema de UI y configuración	Abril	Menús, opciones, persistencia de datos, y autoloads globales
Desarrollo de niveles	Abril	Construcción de escenarios y enemigos, implementación de jefes y trampas
Validación y pruebas	Abril, mayo, junio	Pruebas funcionales, corrección de errores, ajustes de rendimiento
Documentación	Marzo, abril, mayo, junio	Redacción de memoria técnica, creación de anexos, publicación en GitHub

9. Conclusión

Durante el desarrollo del proyecto, el tiempo limitado dificultó llevar a cabo el juego como lo había imaginado. Uno de los mayores desafíos fue implementar la máquina de estados finita (FSM) junto con los controladores (handlers), ya que ambos requerían una planificación y estructura mucho más cuidadas de lo que inicialmente pensé.

Además, surgieron errores en la gestión de la vida del jugador debido a una desincronización en las señales, lo que llevó a reconstruir el sistema. Se incorporó una barra de vida que permite representar de forma más precisa y funcional el estado del jugador.

La implementación de la transición entre niveles mediante el GameState también resultó compleja, ya que requería mantener de forma coherente el estado del juego entre

escenas. Esto hizo evidente la necesidad de planificar con mayor precisión la lógica global del juego.

10. Bibliografía

A continuación, se detallan las referencias a las fuentes de documentación, herramientas y plataformas clave utilizadas durante el desarrollo y la documentación de este proyecto.

10.1. Referencias Bibliográficas y Herramientas

- **Boords(2025)**. Herramienta en línea utilizada para la creación de storyboards. Recuperado de: https://boords.com
- **Canva(2025)**. Plataforma utilizada para la elaboración de diagramas y materiales gráficos. Recuperado de: https://www.canva.com
- **Creative Commons**. (2025). *CC0 1.0 Universal (CC0 1.0) Public Domain Dedication*. Licencia de dominio público utilizada para los recursos del juego. Recuperado de: https://creativecommons.org/publicdomain/zero/1.0/
- **Godot Engine Contributors**. (2025). *Godot Engine Documentation*. Fuente principal de consulta técnica durante el desarrollo. Recuperado de: https://docs.godotengine.org
- **Itch.io**. (2025). Plataforma de distribución de videojuegos y recursos digitales, utilizada para la obtención de assets visuales y de audio. Recuperado de: https://itch.io
- **FreeConvert – PNG to SVG Converter**. Herramienta utilizada para convertir imágenes PNG a SVG https://www.freeconvert.com/png-to-svg
- **Linda Briggen** (2025). hizo el storyboard usando la animación de mi protagonista como referencia para captar bien sus movimientos y expresiones.

10.2. Nota sobre Recursos y Licencias

La documentación oficial de **Godot Engine** fue la fuente técnica principal consultada durante el desarrollo. Las herramientas **Canva** y **Boords** se emplearon para la

creación de diagramas analíticos y storyboards, respectivamente, como parte del proceso de diseño y documentación del proyecto.

Los assets visuales (fondos, personajes, trampas, etc.) y recursos de audio (música, efectos sonoros) fueron obtenidos a través de **Itch.io**, bajo licencias **Creative Commons CC0 1.0 Universal**, lo que permite su uso libre sin necesidad de atribución. Sin embargo, se recomienda ofrecer crédito a los autores o realizar donaciones como muestra de reconocimiento.

10.3. Comunidades y Foros

Durante el desarrollo, se consultaron activamente foros y comunidades en línea, que brindaron soporte técnico, intercambio de ideas y resolución de problemas:

- Boords. (2025). *Boords – Storyboarding made simple*. https://boords.com
- Canva. (2025). *Inicio - Canva*. https://www.canva.com
- Creative Commons. (2025). *CC0 1.0 Universal (CC0 1.0) Public Domain Dedication*. https://creativecommons.org/publicdomain/zero/1.0/
- Godot Engine contributors. (2025.). *Godot Engine documentation*. https://docs.godotengine.org
- Itch.io. (2025). *Itch.io – Indie game hosting*. https://itch.io
- FreeConvert. (s.f.). *PNG to SVG Converter*. https://www.freeconvert.com/png-to-svg
- Reddit. (2025). *Godot Engine subreddit*. https://www.reddit.com/r/godot
- Discord. (2025). *Godot Engine Community Server*. https://discord.gg/godotengine
- Michael Games. (2025). *Michael Games Discord server – Welcome and Rules*. • Discord
- Caffeinated Crows Coffee Cup. (2025). *Streams – Discord community server*. Discord
- Gwizz.(2025). *Streams – Discord community server*. Discord

10.4. Tutorial

Durante el desarrollo del juego, se utilizó como referencia un recurso externo con el fin de implementar correctamente la lógica de interacción con NPCs, la gestión de ítems y los **Handlers** de eventos y **FSM**

10.4.1. NPC y ítems

La implementación de NPCs e ítems se basó en las indicaciones del siguiente recurso:

- **Autor**: [**Michael Games**]
- **Enlace**: [Michael Games - YouTube]

Este material ofreció ejemplos prácticos que facilitaron:

- La creación de NPCs con mecánicas de interacción básicas.
- La gestión de ítems, tanto en su entrega como en su recolección.

10.4.2. Handlers y sistema FSM

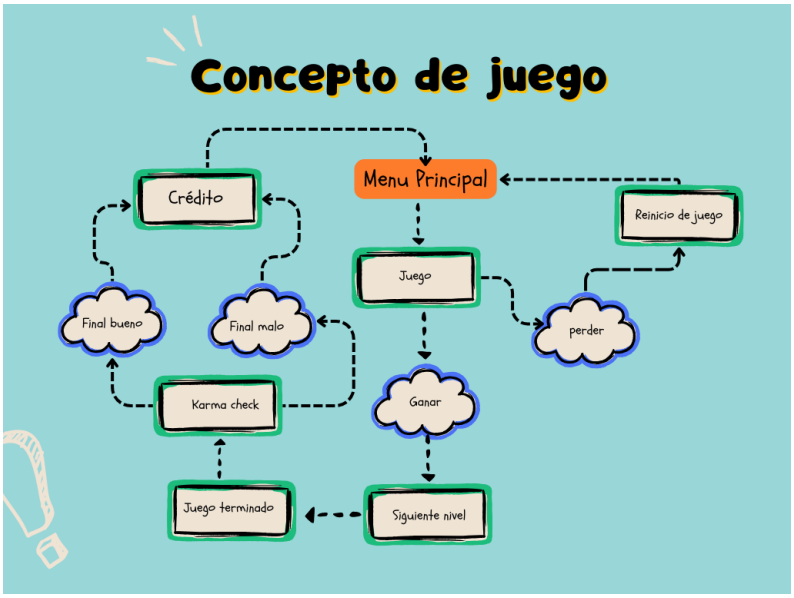
La implementación de **Handlers** y **FSM** se basó en las indicaciones del siguiente recurso:

- **Autor**: [**CoffeeCrow**]
- **Enlace**: [CoffeeCrow - YouTube]

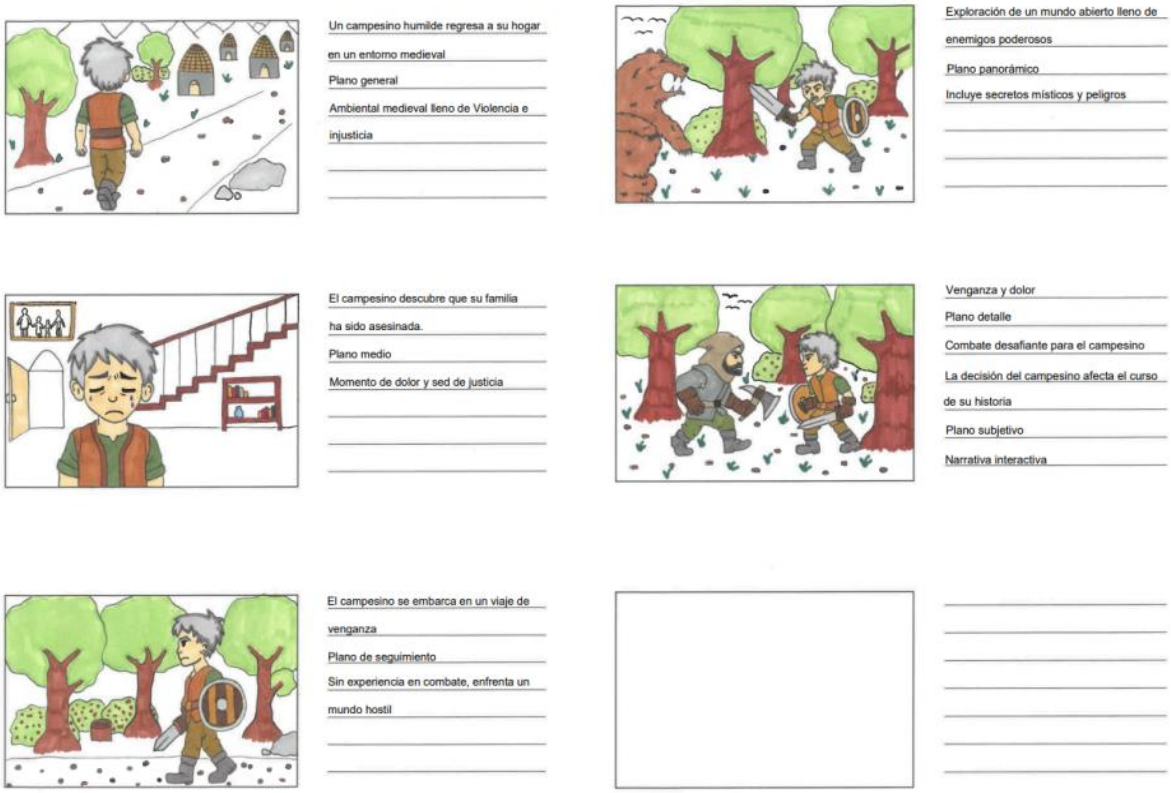
11. Anexo

11.1. Imágenes

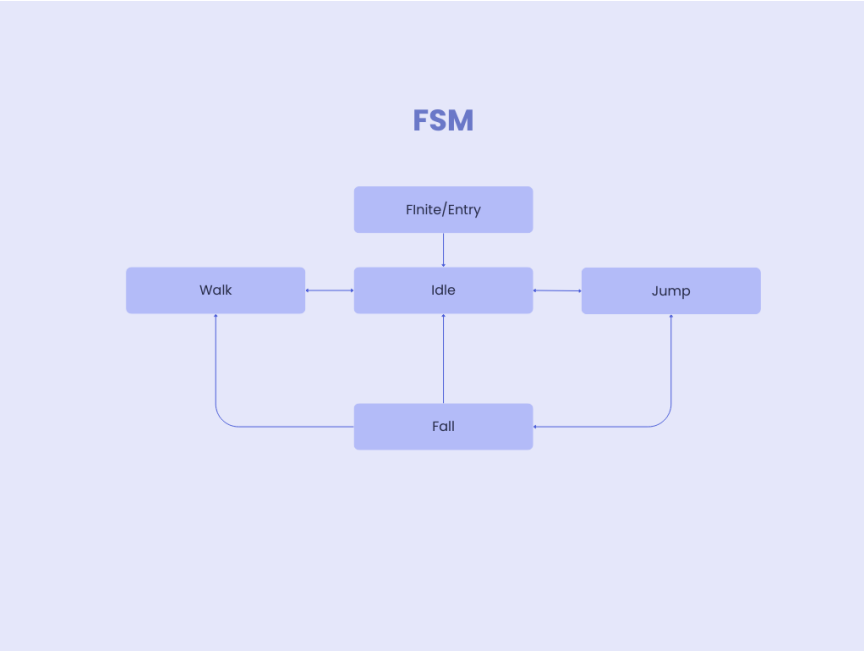
11.1.1. Concepto de juego



11.1.2. Storyboard



11.1.3. FSM (Finite State Machine)



11.2. Git-Hub/ Repositorio

El proyecto completo se encuentra disponible en el siguiente repositorio de GitHub: https://github.com/Kelvinbex2/Proyecto_final