**Proyecto videojuego 2D Godot Engine**

**PRESENTADO POR:**

**Sebastian Andres Perez Arevalo 192459**

**Kevin Libardo Rosas Quintero 192550**

**Darwin Sneider Lindarte Celis 192510**

**PARA**

**JESUS EDUARDO GUERRERO RODRIGUEZ**

**UNIVERSIDAD FRANCISCO DE PAULA SANTANDER SECCIONAL**

**13/06/2025**

**OCAÑA**

**ROLES:**

**KEVIN = Analista e Implementador**

**SNEIDER = diseñador y beta-tester**

**SEBASTIAN = Planificador y documentador**

**Informe de Desarrollo del Proyecto – Juego 2D en Godot**

**1. Introducción**

Este documento tiene como objetivo detallar el proceso de desarrollo de un videojuego 2D realizado con el motor **Godot Engine**, en el marco de un proyecto grupal, siguiendo los principios del Ciclo de Vida del Desarrollo de Software **(SDLC)**. El producto final es un videojuego en 2D que cuenta con un protagonista el cual es un guerrero y enfrenta a enemigos variados en un entorno, el protagonista tendrá la capacidad de atacar, recibir daño, saltar y morir.

Para garantizar una estructura ordenada y eficiente, se siguieron las fases del **SDLC**, asegurando una correcta planificación, ejecución y mantenimiento del proyecto:

**-Planificación.**

Durante esta fase se definió la idea principal del videojuego: un juego 2D de acción con combates entre un guerrero y distintos enemigos como slimes y golems. Escogimos Godot como herramienta de desarrollo debido a la cantidad de información guía para su uso y su capacidad para facilitar un enfoque en juegos 2D, además de su sistema de nodos el cual conseguimos asimilar debido a su similitud con el sistema de herencias, clases y métodos que encontramos al momento de utilizar Java por lo tanto logramos conectar los conocimientos ya adquiridos sobre ese lenguaje de programación y traerlos al ambiente de **Godot Engine**.

**-Análisis de Requisitos.**

Se identifico la aplicación o motor del juego que usaríamos como herramienta, además, los elementos esenciales que incluye el mismo:

* **Investigamos previamente sobre el motor para juegos 2D más apto.**
* **Estudiamos y comprendimos cómo funciona Godot.**

**Funcionales:**

* El jugador puede moverse en 2D (izquierda, derecha, saltar).
* Recolección de objetos (como las monedas o llaves).
* Enemigos o elementos interactivos, agregamos variedad de enemigos y objetos que permiten al jugador interactuar con su entorno.
* Gestión de vidas/energía (barra de vida).

**No funcionales:**

* El juego debe ser fluido (mínimo 30 FPS).
* Compatible con dispositivos de escritorio.
* Código modular y reutilizable.

**-Diseño.**

En esta etapa se diseñó la arquitectura general del juego, incluyendo:

* Estructuras que se encontraran dentro del juego como plataformas y fondo del juego, además de los nodos en Godot que utilizamos para armar la base general del juego.
* Sistema de colisiones y físicas básicas.
* Creamos la interfaz de inicio y el game over al momento de que el jugador muera

Se tomaron decisiones sobre Sprite, animaciones.

**-Implementación (Codificación).**

en esta fase utilizamos.

**Lenguaje y Motor**

* **Motor:** Godot Engine.
* **Lenguaje:** GDScript, orientado a nodos y escenas.

-Utilizamos el sistema basado en nodos de manera que cada entidad del juego al realizarlas como escenas independientes se le pueda dar una reutilización y un mantenimiento más fácil.

**-Cada elemento tiene su propio script .gd, asegurando separación de responsabilidades.**

Se agrego durante el proceso del videojuego diferentes estructuras, entidades y objetos con los que el personaje puede interactuar, además, del personaje en si mismo:

**Personaje Principal.**

* Script: **Character\_Body\_2d.gd**
* Funcionalidades implementadas:
  + Movimiento 2D básico.
  + Hicimos que la cámara siguiera el personaje de manera que este sea un centro y se consiga percibir todo lo que lo rodea
  + Sistema de disparo mediante el apropiado uso del nodo bullet.tscn.
  + Sistema de colisión para detectar enemigos y objetos.
  + Barra de vida visual**.**

**Estados definidos:** IDLE, RUNNING, JUMP, ATTACKING, SHOOTING, DYING.

Las propiedades de estas interacciones del personaje principal son:

**Propiedades principales:**

* Movimiento normal y carrera.
* Saltos con gravedad, de manera que cuando el personaje salte también caiga ni muy rápido ni muy lento.
* Ataques cuerpo a cuerpo y disparos.
* Barra de salud (con take\_damage() y healthbar).
* Recolección de monedas y llaves.
* Cambio de escena al morir.

**IDLE ──► RUNNING ──► JUMP**

**│ ▲ │**

**▼ │ ▼**

**ATTACKING ◄ SHOOTING**

**│**

**▼**

**DYING**

**jugador Funcionalidades**.

[Jugador]

|

|--- [Moverse (izq/der)]

|--- [Saltar]

|--- [Atacar]

|--- [Disparar]

|--- [Recolectar moneda/llave]

|--- [Recibir daño]

|--- [Cambiar de escena (GameOver)]

**Sistema de Disparo**

* Implementado mediante el nodo bullet.tscn y su correspondiente script.
* Las balas se generan y se mueven en una dirección hasta colisionar con un enemigo u obstáculo.

**Bala (bullet.tscn, instanciada desde fire())**

Las balas se crean dinámicamente y se mueven en la dirección del jugador.

**Objetos Interactivos**

* coin.tscn: Objeto recolectable por el jugador.
* door.tscn: permite la interacción con una puerta que hay en el juego.

**Moneda (coin.gd)**

**Tipo:** Area2D

**Función:** Inicia una animación y se elimina al entrar en contacto con el jugador.

**Enemigos.**

Agregamos diferentes tipos de enemigos algunos son pasivos y no atacan al jugador sin embargo tienen colisión e interactúan con el personaje, algunos como los golems pueden atacar al jugador y ser atacados por el mismo, a estos por ejemplo se les añadió su propio script para permitir el daño al jugador .

* **is\_alive = false**: Marca al enemigo como muerto para evitar nuevas interacciones.
* **call\_deferred("spawn\_coins", 3)**: Llama a la función spawn\_coins después de que se liberen los recursos del frame actual. Esto previene errores si se intentan instanciar objetos justo antes de eliminar el nodo lo que a su vez permite que los enemigos suelten monedas cuando mueran.
* **call\_deferred("queue\_free")**: Elimina el nodo enemigo del árbol de escena, pero de forma segura y postergada.
* **coin\_scene.instantiate()**: Crea una nueva instancia de una moneda.
* **randf\_range(-20, 20)**: Genera una posición aleatoria cerca del enemigo para dispersar las monedas.
* **get\_tree().current\_scene.add\_child(coin)**: Agrega la moneda al mundo actual del juego.
* permite que el enemigo reciba daño si un área (por ejemplo, una **bala o ataque cuerpo a cuerpo**) entra en su hurtbox.
* Verifica si el área o su nodo padre tiene un método get\_damage(), que devuelve el daño infligido.
* Si lo tiene, aplica ese daño usando take\_damage().

+-------------------------------+

| Enemy |

+-------------------------------+

| - is\_alive: bool |

| - damage\_amount: int |

| - coin\_scene: PackedScene |

+-------------------------------+

| +\_die(): void |

| +spawn\_coins(amount: int): void |

| +\_on\_hurtbox\_body\_entered(): void |

| +\_on\_hurtbox\_area\_entered(): void |

+------------------------------+

**Lógica de Juego.**

* Scripts para cada mecánica individual (por ejemplo coin.gd, Door.gd), lo que mantiene el código modular es decir está dividido en secciones para que sea más fácil hacer un mantenimiento y de esa manera probar y prevenir errores de manera más sencilla.
* Sistema sencillo de colisiones lo que permite de manera más fácil la interacción del personaje principal con su entorno.

**Métodos principales utilizados dentro del programa**.

| **Método** | **Descripción** |
| --- | --- |
| \_physics\_process(delta) | Lógica de movimiento, ataques y disparo. |
| start\_attack() | Activa ataque cuerpo a cuerpo. |
| fire() | Instancia y dispara una bala. |
| take\_damage(amount) | Resta salud y gestiona muerte. |
| \_die() | Cambia de escena a Game Over. |
| set\_count\_coin(n) | Actualiza contador de monedas. |
| set\_count\_key(n) | Actualiza contador de llaves. |

**-Fase de pruebas.**

Durante esta fase se realizaron pruebas continuas para verificar:

* **Funcionamiento de las mecánicas.**
* **Comportamiento correcto de enemigos verificación de que sus colisiones e interacciones funcionen de manera óptima.**
* **corrección de errores críticos considerados bugs que dañan la experiencia del jugador.**
* **Verificamos que la cámara siguiera de manera correcta y flexible hacia el entorno para que se observe el juego sin excluir partes importantes.**

**Tipos de prueba realizadas:**

* **Pruebas de jugabilidad manual:** movimiento, ataque, disparo.
* **Validación de colisiones:** monedas, enemigos, plataformas entre otros.
* **Estados del jugador:** verificación visual y de comportamiento.
* **Transición de escenas:** al morir.