|  |  |  |
| --- | --- | --- |
| Asignatura: | Día y hora: | Grupo: |
| Docente:  Rafael E. Rodríguez Guerrero | Laboratorio 3 | **Aula:** |
| Estudiantes: | **Carnet** | **Nota** |
| 1. Jorge Rene Guardado Deleon | **GD100422** |  |
| 1. Steve Enedicto Madrid Chavez | **MC103322** |  |
| 1. Brayan Alberto Flores Rosales | **FR101022** |  |
| 1. Jossue Daniel Peña Hernandez | **PH100221** |  |
|  |  |  |
|  |  |  |

**PARTE I** **CANVAS**

Realizar un Canvas que contenga 4 GameObjects vacíos los cuales actuaran como los diferentes partes de un Menú el cual se activara y desactivara de acuerdo al botón presionado, debe contener :

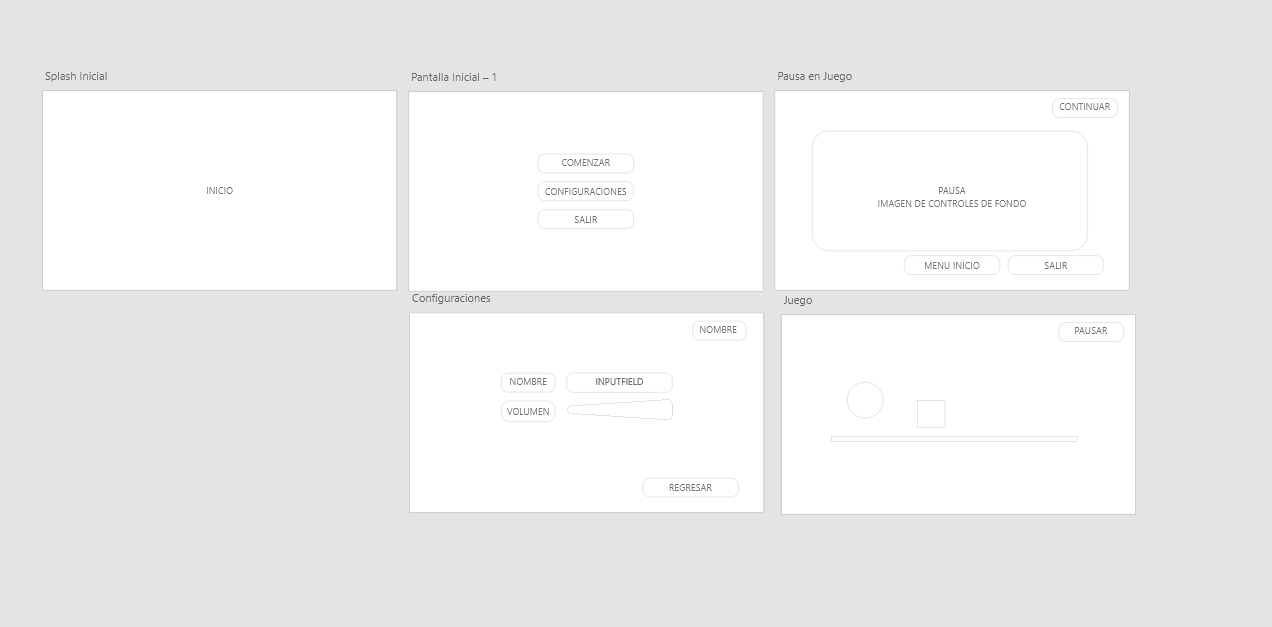
GameObject 1 Pantalla de Inicio que al Presionar el Botón comenzar me Llevara a un Menú Inicial   
  
GameObject 2 Pantalla de Menú Inicio Con tres botones Ordenados utilizando el Script Vertical Layout Group   
\* Primer botón debe ir a la Scene del Juego al iniciar debe Mostrar una Splash de los controles

\* Segundo botón debe ir Una pantalla de Configuraciones

GameObject3 Pantalla Configuraciones que debe Tener Un Input para ingresar un String que sea el Nombre de un Usuario debe Guardarse e imprimirlo en una Esquina

GameObject 4 debe tener Imágenes que nos Indiquen los controles a utilizar Puedes Crearla y luego Ingresarla. Además, debe tener un botón para poder Colocarla como botón de pausa

Todas las salas deben Escalar con las resoluciones y estar ancladas al lugar que sea más conveniente



**Parte II GAME SCENE**

\*Crear un Canvas donde tendremos a la Vista el puntaje

\*Crear una Cubo el cual debe utilizar el Script de Movimiento, El cubo debe poder Moverse con el script agregarle un material y hacer que el Cubo de un Color ( Azul o Verde) Agregarle el Tag Jugador

\*Crear Un plano 100x100 debe pertenecer al Layer GroundCheck

En el plano vamos a distribuir 4 tipos de objetos

Esferas,( rojas ) Cantidad 5 Tag Enemigos

Cubos (Amarillos) Cantidad 3 Tag Aliados  
Agregar la script de movimiento Automatico al cubo

Cilindros ( Blancos) Cantidad 8 Tag PowerUp ( Trigger)

Capsulas (Gris) Cantidad 4 Tag Colectables

\*Generar un Ground Checker con un RayCast el Cubo y agregar el código necesario para que este salte solo cuando un rayo lanzado de su centro haga contacto con el Plano

\* Los Cilindros con Tag ( powerUp ), son Triggers, y cambian el color de la Esfera a (Dorado) durante 10 segundos, con este color pueden Destruir las esferas rojas y los aliados.

\* Al destruir las esferas o Cubos debe salir un debug por consola indicando el nombre del objeto + si es enemigo o Amigo

Si se destruye un Enemigo debe aumentar el puntaje que debe ser visto desde el UI y si es un Amigo debe disminuir el puntaje. Esta debe poderse llamar desde la variable en el inspector para probarse

Los cubos aliados solo pueden ser recolectados sin el bufo de los cilindros

Agregar un RayCast que salga del cubo y tenga un rango de 30f y cuando haga contacto con las Capsulas estas cambien a un color Negro y mande por consola la Distancia y la cantidad de Capsulas convertidas

**PARTE III TEORIA**

1. Que son los Coroutine y como se usan en Unity

En Unity, las coroutine son esencialmente métodos que devuelven una interfaz IEnumerator. Esta interfaz forma parte del espacio de nombres System.Collections y

proporciona los métodos básicos para iterar sobre una colección. Unity utiliza de manera ingeniosa esta interfaz para controlar el flujo de ejecución de las coroutine.

Una coroutine permite distribuir tareas a lo largo de varios fotogramas. En situaciones normales, cuando se llama a un método, se ejecuta hasta su finalización y luego devuelve el control al método llamante, junto con cualquier valor de retorno opcional. Esto significa que cualquier acción que ocurra dentro de un método debe suceder dentro de una sola actualización de fotograma. Sin embargo, en situaciones donde un método contenga una animación procedural o una secuencia de eventos a lo largo del tiempo, se puede utilizar una coroutine.

1. Investigar las diferentes Formas de guardar Variables entre Escenas en Unity
2. PlayerPrefs:
   * Descripción: PlayerPrefs es una forma sencilla de almacenar y recuperar datos en Unity. Puedes guardar valores como enteros, flotantes o cadenas de texto.
   * Uso: Utiliza PlayerPrefs.SetInt(), PlayerPrefs.SetFloat() o PlayerPrefs.SetString() para guardar datos y PlayerPrefs.GetInt(), PlayerPrefs.GetFloat() o PlayerPrefs.GetString() para recuperarlos.
   * Ventajas: Fácil de usar y adecuado para datos simples.
   * Limitaciones: No es seguro para datos sensibles y no es adecuado para grandes cantidades de información.
3. ScriptableObjects:
   * Descripción: Los ScriptableObjects son objetos que pueden almacenar datos como assets en tu proyecto. Son útiles para compartir datos entre scripts y escenas.
   * Uso: Crea un ScriptableObject personalizado con las variables que deseas guardar. Luego, referencia este objeto desde tus scripts.
   * Ventajas: Desacopla el código y evita la necesidad de una clase de administrador.
   * Limitaciones: No guarda datos en el disco duro.
4. Clases estáticas:
   * Descripción: Puedes crear una clase estática que almacene las variables que deseas conservar entre escenas.
   * Uso: Crea una clase estática con propiedades o campos para tus datos y accede a ellos desde cualquier script.
   * Ventajas: Acceso global a los datos y persistencia durante toda la ejecución del juego.
   * Limitaciones: No es adecuado para datos sensibles y puede dificultar el mantenimiento en proyectos grandes.
5. Singletons:
   * Descripción: Los singletons son clases que solo tienen una instancia en todo el juego. Puedes usarlos para almacenar datos entre escenas.
   * Uso: Crea una clase singleton con propiedades o campos para tus datos y accede a ellos desde cualquier script.
   * Ventajas: Acceso global a los datos y mayor control sobre la instancia única.
   * Limitaciones: Puede ser más complejo de implementar que las otras opciones.
6. Que son los Colliders

Los colliders son componentes que definen la forma de un objeto para propósitos de colisiones físicas. Aunque no son visibles, no necesitan coincidir exactamente con la forma del objeto visual (mesh). De hecho, una aproximación suele ser más eficiente y no afecta la apariencia del juego.

1. Diferentes tipos de Colliders

**Tipos de Colliders Primitivos:**

En 3D, los tipos primitivos de colliders incluyen:

Box Collider: Representa un cubo o paralelepípedo.

Sphere Collider: Define una esfera.

Capsule Collider: Similar a un cilindro con extremos semiesféricos.

**En 2D, puedes usar:**

Box Collider 2D: Para colisiones en 2D.

Circle Collider 2D: Define un círculo.

Colliders Compuestos:

Puedes agregar varios colliders a un solo objeto para crear formas más complejas.

Ajustando su posición y dimensiones, los colliders compuestos pueden aproximar la forma del objeto mientras mantienen una carga de procesamiento baja.

También puedes tener colliders adicionales en objetos hijos para mayor flexibilidad.

**Mesh Colliders:**

En 3D, los Mesh Colliders se ajustan exactamente a la forma del mesh del objeto.

En 2D, el Polygon Collider 2D no coincide perfectamente con la figura del sprite, pero puedes redefinirlo con cualquier nivel de detalle necesario.

1. Que es un Rigidbody

Es un componente que permite que los objetos se muevan de manera realista bajo la influencia de fuerzas y torque. En otras palabras, es una función que permite que un GameObject se vea afectado por la física simulada.

1. Cuál es la diferencia entre Ray y Raycast y cuales son Elementos o parámetros

Tanto el Ray como el Raycast están relacionados con la detección de colisiones y la simulación de rayos.

Las diferencias entre estos y sus elementos son:

1. **Ray**:
   * Un Ray es simplemente un rayo imaginario que se extiende desde un punto en una dirección específica.
   * No realiza ninguna colisión real ni interacción con objetos en la escena.
   * Se utiliza para determinar si un punto en una dirección dada está obstruido por algún objeto.
   * Los elementos clave de un Ray son:
     + Origen: El punto desde el cual se origina el rayo.
     + Dirección: La dirección en la que se extiende el rayo.
2. **Raycast:**
   * Un Raycast es una operación que lanza un rayo y verifica si colisiona con algún objeto en la escena.
   * Devuelve información detallada sobre lo que fue golpeado, como la distancia al objeto y la normal de la superficie.
   * Los elementos clave de un Raycast son:
     + Origen: El punto desde el cual se origina el rayo.
     + Dirección: La dirección en la que se extiende el rayo.
     + Distancia Máxima: La distancia máxima que el rayo debe verificar para colisiones.
     + LayerMask: Una máscara de capa que se utiliza para ignorar selectivamente colisiones con ciertos colliders.
     + QueryTriggerInteraction: Controla si los colliders de tipo “Trigger” generan una colisión o no.la
3. Realice un Ejemplo de cada los primeros 5 Atributos visto en Clases ( pueden usar los ejemplos pero no deben ser los mismo )

**SerializeField:**

public class AlgunasPersonas : MonoBehaviour

{

public string PrimerNombre = "John";

private int age = 40;

[SerializeField]

private bool hasHealthPotion = true;

private void Start()

{

if (hasHealthPotion)

{

Debug.Log("Nombre de la persona: " + PrimerNombre + ", Edad de la persona: " + age);

}

}

}

**HideInInspector:**

using UnityEngine;

public class HideInInspectorEjemplo : MonoBehaviour

{

[HideInInspector]

public int a = 5;

[HideInInspector, SerializeField]

private int b = 3;

[SerializeField]

private int c = 3;

}

**Range**

using UnityEngine;

public class Ejemplo : MonoBehaviour

{

[Range(1, 6)]

public int integerRange;

[Range(0.2f, 0.8f)]

public float floatRange;

}

**Space:**

using UnityEngine;

public class MiScript : MonoBehaviour

{

void Update()

{

transform.Rotate(Vector3.up \* Time.deltaTime \* 30f, Space.Self);

}

}

**Header**

using UnityEngine;

public class EjemploHeader : MonoBehaviour

{

[Header("Ajustes de vida")]

public int vida = 0;

public int vudamaxima = 100;

[Header("Ajustes de Escudo")]

public float Escudo = 0;

public float Escudomaximo = 0;

}