

DOCUMENTACIÓN UNITY

| | |
|--|----------|
| INTRODUCCIÓN..... | 2 |
| PREFABS..... | 3 |
| 1. Animals..... | 3 |
| 2. Lobby..... | 3 |
| 3. AudioTriggers..... | 3 |
| 4. Managers..... | 4 |
| 5. Objects..... | 4 |
| Arqueología Submarina:..... | 4 |
| Limpieza del Onashaga:..... | 4 |
| Photo:..... | 4 |
| 6. Particles..... | 5 |
| 7. Player..... | 5 |
| 8. SceneObjects..... | 5 |
| 9. Tools..... | 5 |
| Prefab sin carpeta: WALLS..... | 5 |
| SCRIPTS..... | 6 |
| 1. Player..... | 6 |
| Swimmer:..... | 6 |
| Descripción de los Métodos..... | 6 |
| Resumen de Componentes Principales..... | 7 |
| Watch:..... | 7 |
| Descripción de métodos principales..... | 7 |
| Awake..... | 7 |
| Start..... | 7 |
| Update..... | 7 |
| OnDisable..... | 7 |
| Sección: Pantallas (Screen)..... | 8 |
| ActiveScreen..... | 8 |
| ChangeScreen..... | 8 |
| UpdateScreen..... | 8 |
| ChangeScreenCorutine..... | 8 |
| Sección: Mapa (Map)..... | 8 |
| UpdateRadar..... | 8 |
| CalculateRadarPosition..... | 8 |
| UpdateRadarObject..... | 8 |
| FindInactiveRadarPoint..... | 8 |
| Sección: Tareas (Tasks)..... | 9 |
| RegisterTaskEvents / UnregisterTaskEvents..... | 9 |
| NavigateTaskButtons..... | 9 |
| NavigateTaskButtonsCorutine..... | 9 |

| | |
|--|----|
| ExecuteTaskButton..... | 9 |
| UpdateTaskButtonHighlight..... | 9 |
| 2. Tools..... | 10 |
| Tool:..... | 10 |
| CameraTool:..... | 11 |
| Explicación de los métodos..... | 11 |
| GpsTool..... | 13 |
| Descripción de los Métodos..... | 13 |
| Resumen de Componentes Principales..... | 14 |
| 3. Mission..... | 15 |
| Mission:..... | 15 |
| Descripción de los Métodos..... | 15 |
| Resumen de Componentes Principales..... | 15 |
| FishAI:..... | 15 |
| Descripción de los Métodos..... | 16 |
| Resumen de Componentes Principales..... | 16 |
| 3.1 Lobo Marino..... | 17 |
| Lobo Marino Behaviour..... | 17 |
| Lobos Activity Spots:..... | 18 |
| Lobo Marino Activity..... | 19 |
| 3.2 Limpieza..... | 20 |
| CleaningActivity..... | 20 |
| TrashItem..... | 21 |
| 3.3 Hidden..... | 22 |
| HiddenActivity..... | 22 |
| Componentes del Script:..... | 22 |
| Resumen de Componentes Principales:..... | 23 |
| Comportamiento:..... | 24 |
| ToritoBehaviour..... | 24 |
| Descripción de los Métodos:..... | 24 |
| Resumen de Componentes Principales:..... | 25 |
| PulpoBehaviour..... | 26 |
| Descripción de los Métodos..... | 26 |
| Resumen de Componentes Principales..... | 27 |
| ErmitañoBehaviour..... | 28 |
| Descripción de los Métodos..... | 28 |
| Resumen de Componentes Principales..... | 29 |
| 3.4 Fish..... | 30 |
| FishActivity..... | 30 |
| Componentes del Script:..... | 30 |
| Campos de Configuración:..... | 30 |
| Variables de Estado:..... | 30 |
| Métodos Clave:Start()..... | 30 |
| OnDestroy()..... | 30 |

| | |
|--|----|
| TryToAddAnimal(Transform[] animals)..... | 31 |
| CheckCompletion()..... | 31 |
| Resumen de Componentes Principales:..... | 31 |
| Comportamiento:..... | 31 |
| CormoranBehaviour..... | 32 |
| Descripción General:..... | 32 |
| Métodos del Script:..... | 32 |
| Resumen de Componentes Principales:..... | 33 |
| NototeniaBehaviour..... | 34 |
| Descripción General:..... | 34 |
| Métodos del Script:..... | 34 |
| Componentes Principales:..... | 35 |
| 3.5 Arqueología..... | 37 |
| ArqueologiaActivity..... | 37 |
| Componentes del Script..... | 37 |
| Campos de Configuración..... | 37 |
| Métodos Clave..... | 37 |
| Resumen de Componentes Principales..... | 38 |
| Campos Públicos..... | 38 |
| Métodos Importantes..... | 38 |
| Comportamiento..... | 38 |
| Item..... | 39 |
| 4. Lobby..... | 39 |
| FadeEffect..... | 39 |
| Componentes:..... | 39 |
| Métodos:..... | 39 |
| Comportamiento:..... | 40 |
| HoverCanvasController..... | 40 |
| Componentes:..... | 40 |
| Métodos:..... | 40 |
| Comportamiento:..... | 40 |
| LanguageSelector:..... | 40 |
| LevelLoader..... | 41 |
| LevelSelector..... | 42 |
| RayColorChanger..... | 43 |
| RayVisualizer..... | 43 |
| 5. Audio..... | 45 |
| AudioManager..... | 45 |
| FmodEvents..... | 46 |
| 5.1 Ambience..... | 47 |
| AmbienceChangeTrigger..... | 47 |
| 6. Physics..... | 48 |
| ManageBoxColliderOnGrab:..... | 48 |
| 7. Path..... | 49 |

| | |
|-------------------------------|----|
| PathfindingManager..... | 49 |
| 8. Editor..... | 52 |
| PathfindingManagerEditor..... | 52 |

INTRODUCCIÓN

Este documento tiene como objetivo servir como una guía detallada para comprender y trabajar en el proyecto de Unity "Galería Onashaga". A lo largo de sus capítulos, se abordarán aspectos clave de la administración del proyecto, incluyendo la estructura y función de los "prefabs", los "scripts", y las "scenes". El proyecto está desarrollado en Unity versión 2022.1.10f1 y utiliza las tecnologías de realidad virtual (VR) mediante los paquetes "XR Interaction Toolkit" v2.4.3 y "XR Plugin Management".

El documento se organiza en capítulos correspondientes a las principales carpetas del proyecto:

- **Prefabs:** Se detallará la construcción de cada prefab, explicando la razón de los componentes utilizados y su funcionalidad dentro del proyecto.
- **Scripts:** Este capítulo se centrará en el código del proyecto, proporcionando información sobre los scripts asociados a cada "GameObject", su herencia, composiciones, variables, y métodos.

PREFABS

La carpeta **PREFABS** está organizada en nueve subcarpetas, cada una con una función específica en el proyecto:

1. Animals

Contiene los prefabs de los animales presentes en las actividades.

Cada prefab incluye:

- Un GameObject padre con el script Behaviour del animal.
- Un GameObject hijo con el componente LOD Group, que organiza los diferentes LODs (Level of Detail).
 - El nivel LODHD puede contener un Animator si es necesario.

2. Lobby

Contiene los prefabs utilizados en la escena del lobby:

- **CanvasLobby**: Pantallas de texto descriptivas de las actividades. Incluye:
 - Rect Transform, Canvas, Canvas Scaler y GraphicRaycaster.
- **LevelLoader**: Maneja las transiciones entre el lobby y las actividades. Incluye el script Level Loader.
- **PlayerLobby**: Representa al jugador en el lobby con funcionalidades específicas.
Componentes principales:
 - PlayerInput, XR Origin, Input Action Manager.
 - Hijos como Camera Offset (que incluye Main Camera, Left Controller, Right Controller).
 - Main Camera contiene FMOD Studio Listener y un Post-process Layer.
 - Left Controller y Right Controller integran scripts como XR Controller, XR Ray Interactor, Ray Visualizer, Ray Color Changer, y un Line Renderer.
 - Modelos de las manos (Left Hand Model y Right Hand Model) como hijos correspondientes.

3. AudioTriggers

Incluye el prefab **ChangeAmbience**, que maneja cambios dinámicos en los sonidos ambientales.

Componentes:

- Box Collider, Cube (Mesh Filter) y el script Ambience Change Trigger.

4. Managers

Prefabs que gestionan diferentes sistemas de la aplicación:

- **AudioManager**: GameObject con el script Audio Manager.
- **Locomotion System**: Prefab estándar del XR Toolkit.
- **PathfinderManager**: Crea rutas para animales en las actividades. Incluye el script Pathfinder Manager.
- **XR Interactor Manager**: Prefab del XR Toolkit.

5. Objects

Prefabs de objetos interactivables, con distintas configuraciones según la actividad:

Arqueología Submarina:

- Prefabs como **Caja, Jarra, PlatoHondo, PlatoPlayo, y Taza**.
Componentes principales:
 - Script Item, colisionadores según la forma, un Mesh Filter y un Mesh Renderer.
 - Incluyen un GameObject Point para visualización en el mapa de la pulsera.

Limpieza del Onashaga:

- Prefabs como **Bottle, Tablon, Tablon de Madera, y Torus.027**.
Componentes principales:
 - Collider, Rigidbody, scripts XR Grab Interactable y Trash Item.
 - Hijos con colisionadores secundarios, Mesh Filter, y Mesh Renderer.
 - Modelos de manos adaptados para sostener los objetos.

Photo:

Prefab instanciado al tomar una foto con la herramienta Camera. Componentes:

- XR Grab Interactable, Manage Box Collider On Grab, Rigidbody, y un Box Collider.
- Hijos para el modelo (Mesh Filter, Mesh Renderer) y bordes de la foto (Sheet).

6. Particles

Incluye el prefab **Stars**, instanciado al completar una actividad. Componentes:

- Sistema de partículas (Particle System).

7. Player

Prefabs relacionados con el jugador:

- **Player**: GameObject con:
 - Scripts como XR Origin, Input Action Manager, y Swimmer.
 - Hijos (Main Camera, Left Controller, Right Controller) configurados con los scripts y modelos necesarios.
 - **Manos**: Animadas mediante el script Animate Hand On Input.
- **Watch**: Representa la pulsera del jugador. Incluye:
 - Canvas, Canvas Scaler y pantallas como Map, Radar, Task, Tools y Photos.
- Otros: Prefabs **RadarPlayer**, **RadarPoint**, y **FadeScreen**.

8. SceneObjects

Contiene prefabs de plantas y animales invertebrados.

- Cada prefab incluye un LOD Group con sus respectivos niveles de detalle y huesos (si aplica).

9. Tools

Prefabs de herramientas instanciadas desde la pulsera:

- **CameraTool**: Incluye scripts como Camera Tool, XR Grab Interactable, y un Rigidbody.
 - Hijos como Screen, GrabLeft, GrabRight y modelos de manos personalizados.
- **Gps**: Herramienta con scripts como XR Grab Interactable y Gps.
 - Hijos que incluyen el modelo, un Line Renderer, y elementos para el minijuego (Anillo, AnilloJuego, y Text).

Prefab sin carpeta: WALLS

GameObject que define paredes invisibles en los mapas de las actividades.

- Estas paredes solo se vuelven visibles al acercarse el jugador.

SCRIPTS

La carpeta **SCRIPTS** está organizada en ocho subcarpetas, cada una con una función específica en el proyecto:

1. Player

La carpeta Player contiene los scripts de: Swimmer, Watch y Custom_Gravity (no utilizado en el proyecto)

Swimmer:

Este script controla el movimiento del jugador en un entorno de realidad virtual (VR), simulando que "nada". Usa los controles de las manos para calcular la dirección y fuerza del movimiento, aplicando física mediante un Rigidbody. También reproduce un sonido cuando hay cambios significativos en la velocidad de las manos durante el nado.

Descripción de los Métodos

1. Awake()

- Inicializa las referencias y configura el Rigidbody para evitar que rote debido a la física.

2. FixedUpdate()

- Se ejecuta en cada frame de la simulación de física y llama al método Swimming() para manejar el movimiento del jugador.

3. Swimming()

Este método implementa la lógica principal del nado:

1. **Condiciones para nadar:**
 - Verifica si ha pasado suficiente tiempo desde el último impulso y si ambos botones de los controles están presionados.
2. **Cálculo de fuerza y dirección:**
 - Suma las velocidades de las manos, calcula la dirección y aplica fuerza al Rigidbody si supera un umbral mínimo (minForce).
3. **Efectos de sonido:**
 - Detecta cambios en la velocidad de las manos y reproduce un sonido si se cumple el umbral de cambio y el tiempo mínimo entre efectos (minTimeBetweenSFX).
4. **Resistencia (arrastre):**
 - Si hay velocidad, aplica una fuerza contraria para simular resistencia del agua.

Resumen de Componentes Principales

- **Variables de configuración:**
Controlan fuerza de nado, arrastre y tiempos mínimos entre acciones.
- **Referencias externas:**
 - InputActionReference: Entrada de los controladores VR.
 - trackingReference: Calcula el movimiento relativo.
 - AudioManager: Reproduce efectos de sonido.
- **Interacción con física:**
Usa un Rigidbody para aplicar fuerzas físicas que simulan el nado.

Watch:

El script administra una pulsera que muestra diferentes "pantallas" interactivas (mapa, tareas, herramientas y fotos). Cada pantalla tiene funcionalidades específicas, como rastrear objetos importantes en el mapa, navegar por tareas o administrar fotos tomadas en el entorno.

Descripción de métodos principales

Awake

- Configura referencias a entradas del jugador (PlayerInput) y asigna acciones de entrada para el panel de navegación.

Start

- Inicializa colores para diferentes tipos de etiquetas en el radar.
- Configura escuchadores para herramientas y entradas.
- Establece la pantalla inicial en "Tareas" y actualiza los botones de la interfaz.

Update

- Llama periódicamente a UpdateRadar para actualizar la visualización del radar si el mapa está activo.

OnDisable

- Elimina los escuchadores de eventos asignados en Start para evitar referencias persistentes al desactivar el objeto.

Sección: Pantallas (Screen)

ActiveScreen

- Alterna entre mostrar y ocultar la interfaz de la pulsera.

ChangeScreen

- Cambia la pantalla activa (por ejemplo, de "Mapa" a "Tareas") dependiendo de la entrada del jugador. Usa un sistema de enfriamiento para evitar cambios rápidos.

UpdateScreen

- Activa la interfaz de la pantalla seleccionada y desactiva las demás.
- Gestiona eventos específicos según la pantalla activa.

ChangeScreenCoroutine

- Corutina que aplica un retraso para procesar el cambio de pantalla.

Sección: Mapa (Map)

UpdateRadar

- Calcula y actualiza las posiciones de los objetos importantes dentro del rango del radar.
- Utiliza el LayerMask para detectar objetos relevantes en un área específica alrededor del jugador.

CalculateRadarPosition

- Normaliza la posición de los objetos detectados para ajustarlos al rango y escala del radar.

UpdateRadarObject

- Instancia o reutiliza puntos en el radar para representar objetos detectados, asignándoles colores basados en sus etiquetas.

FindInactiveRadarPoint

- Busca puntos de radar desactivados para reutilizarlos en lugar de instanciar nuevos.

Sección: Tareas (Tasks)

RegisterTaskEvents / UnregisterTaskEvents

- Registra o elimina escuchadores para eventos de navegación de botones en la pantalla de tareas.

NavigateTaskButtons

- Cambia la selección activa entre los botones de tareas según la entrada del jugador.

NavigateTaskButtonsCorutine

- Corutina que implementa un retraso al cambiar entre botones de tareas.

ExecuteTaskButton

- Activa el contenido relacionado con la tarea seleccionada, ocultando los demás.

UpdateTaskButtonHighlight

- Cambia el color del botón seleccionado para indicar su estado.

Sección: Herramientas (Tools)

- **AddToolListeners**

Registra los escuchadores de eventos selectEntered para cada herramienta, permitiendo la interacción del jugador con los objetos de herramientas.

- **RemoveToolListeners**

Elimina todos los escuchadores de eventos selectEntered de las herramientas, desactivando la interacción.

- **OnGrabBubble**

Método que gestiona la lógica cuando el jugador agarra una burbuja. Desactiva la herramienta activa (si la hay), activa todas las burbujas y, finalmente, activa la herramienta correspondiente, posicionándose en el lugar adecuado.

Sección: Fotos (Photos)

- **RegisterPhotosEvents**

Registra el evento de entrada para la navegación de botones en la pantalla de fotos, vinculando el método NavigatePhotosButtons.

- **UnregisterPhotosEvents**

Elimina el evento de entrada para la navegación de botones en la pantalla de fotos.

- **NavigatePhotosButtons**

Cambia la selección activa entre los botones de fotos dependiendo de la entrada del jugador y controla los tiempos de espera entre cada cambio.

- **NavigatePhotosButtonsCorutine**

Corutina que implementa un retraso en el cambio de fotos cuando se navega por las opciones, también gestiona el cambio de índice de foto y actualiza el botón seleccionado.

- **ExecutePhotosButton**
Activa la foto correspondiente al botón seleccionado, actualizando la interfaz con la foto actual. Verifica que el índice de foto esté en rango y que la foto no sea nula.
- **GetLast10PhotoSprites**
Retorna la lista de las últimas 10 fotos disponibles, en este caso representadas por sprites.
- **UpdatePhotosButtonHighlight**
Actualiza el color de los botones de fotos, resaltando el botón seleccionado con un color diferente (amarillo).
- **ResetPhotosButtonHighlight**
Restablece el color de todos los botones de fotos a blanco, desactivando el resaltado.

2. Tools

La carpeta Tools contiene los scripts de: Tool, CameraTool y GpsTool

Tool:

Este script hereda de "GrabbableObjectt" y maneja las herramientas de la aplicación. Las herramientas están definidas por un tipo, como cámara, tijera o GPS. El script define cómo se activan y destruyen las herramientas cuando el jugador interactúa con ellas. Cada herramienta tiene un "prefab" que representa el objeto en el mundo del juego, un "gameObject" que es la instancia de la herramienta, y una "bubble", que es un componente que se utiliza para representar la burbuja de la herramienta en el espacio VR. Además, se gestiona un tiempo de destrucción, tras el cual la herramienta desaparece si no se está sosteniendo.

Descripción de los Métodos

1. **Start()**
 - Inicializa las referencias necesarias al jugador, la cámara principal y el reloj. También ajusta el tiempo de destrucción inicial para las herramientas.
2. **Update()**
 - Este método se ejecuta una vez por frame, pero solo llama al Update() de la clase base. Aquí no hay lógica adicional, ya que el comportamiento principal se maneja en el FixedUpdate().
3. **FixedUpdate()**
 - Este método se ejecuta en intervalos fijos y llama al método IsGrab() para comprobar si la herramienta está siendo agarrada o no, gestionando su tiempo de destrucción.
4. **OnDestroy()**
 - Método que se ejecuta cuando el objeto es destruido, llamando al OnDestroy() de la clase base. En este caso, no realiza ninguna acción adicional.
5. **IsGrab()**

- Este método verifica si la herramienta está siendo sostenida por el jugador. Si no lo está, decrementa el contador de destrucción (`currentDestroyTime`). Si el tiempo de destrucción llega a cero, se llama al método `ToolDestroy()`.
6. **ToolDestroy()**
- Desactiva la herramienta (`gameObject`) y restablece su estado a la burbuja asociada. Si la herramienta es destruida, la burbuja vuelve a ser visible y se restablecen los valores del temporizador de destrucción.

Resumen de Componentes Principales

- **ToolType (enum):** Define los tipos posibles de herramientas.
- **ToolObject (clase):** Representa cada herramienta, incluyendo su tipo, su prefab, la instancia de juego (`gameObject`) y su burbuja.
- **Tool (clase):** Clase principal que maneja la lógica de las herramientas, incluyendo su activación, destrucción y el tiempo de espera para su desaparición.

CameraTool:

La clase `CameraTool` hereda de `Tool` y simula las funciones de una cámara en las actividades, permitiendo capturar fotos en el entorno virtual. La cámara puede ser controlada mediante entradas del usuario (como botones para tomar fotos y hacer zoom). Esta clase maneja la captura de imágenes, su visualización en el mundo virtual como objetos 3D, y su almacenamiento en el dispositivo, incluyendo la gestión de permisos de almacenamiento y la actualización de la galería de fotos. Además, implementa efectos visuales y sonidos relacionados con el proceso de toma de fotos.

Explicación de los métodos

1. **OnDrawGizmos()**
 - Este método se utiliza para mostrar una representación visual de la caja de colisión (`BoxCast`) en la escena de Unity. Sirve como una herramienta de depuración, permitiendo al desarrollador visualizar el área que se verifica al intentar capturar una foto.
2. **Start()**
 - Inicializa los componentes necesarios para la toma de fotos, incluyendo la configuración de los controles de entrada del jugador, la asignación de las acciones de entrada para tomar fotos y hacer zoom, y la creación de eventos de sonido (por ejemplo, el sonido de zoom).
3. **OnDestroy()**
 - Desuscribe las acciones de entrada y libera los recursos asociados, como los eventos de sonido. Es un método importante para evitar fugas de memoria al destruir la instancia de la clase.
4. **OnDisable()**
 - Este método se llama cuando la herramienta se desactiva. Desactiva las manos del jugador y detiene los sonidos asociados con el zoom.
5. **CapturePhoto()**
 - Este es el método principal encargado de capturar una foto. Primero espera hasta el final del frame, luego usa un `RenderTexture` para capturar la imagen de la cámara. Después de capturar la foto, la convierte en un objeto `Sprite` y

la guarda en la galería del dispositivo, con la opción de agregarla a un historial de fotos instanciadas en el juego.

6. **ConvertTextureToSprite(Texture2D texture)**
 - Convierte una textura 2D (la foto capturada) en un Sprite para que pueda ser utilizada en la interfaz gráfica del juego o en el mundo virtual como un objeto visual.
7. **OnPhoto(InputAction.CallbackContext context)**
 - Este método se llama cuando el jugador activa la acción de tomar una foto. Ejecuta el proceso de captura de fotos de forma asíncrona usando una corrutina.
8. **SavePhotoToPictures(Texture2D photo)**
 - Guarda la foto capturada en el almacenamiento del dispositivo. Si se está ejecutando en un dispositivo Android, se guarda en la carpeta de fotos del sistema. Si se está ejecutando en el editor de Unity, se guarda en una carpeta local del proyecto.
9. **RefreshGallery(string filePath)**
 - Actualiza la galería del sistema operativo para que la foto guardada sea reconocida y mostrada en la aplicación de galería del dispositivo.
10. **GetExternalStoragePath()**
 - Obtiene la ruta del almacenamiento externo en dispositivos Android. Dependiendo de la versión del sistema operativo, utiliza métodos distintos para obtener la ubicación donde se almacenarán las fotos.
11. **RequestStoragePermission()**
 - Solicita permisos para escribir en el almacenamiento externo del dispositivo, si no se han otorgado previamente.
12. **TakePhoto()**
 - Realiza un BoxCast desde la cámara para detectar objetos dentro del campo de visión de la cámara. Si se detectan objetos, invoca un evento que indica qué objetos fueron impactados. Este método se utiliza para verificar si la foto captura algo interesante (por ejemplo, un objeto dentro de la escena).
13. **RotateTexture(Texture2D texture)**
 - Rota la textura de la foto capturada para corregir su orientación, si es necesario. Este método asegura que la imagen capturada esté orientada correctamente en el mundo virtual.
14. **InstantiatePhotoPrefab(Texture2D photo)**
 - Instancia un prefab de foto en el mundo virtual, asignando la textura capturada como su material. Si el número de fotos instanciadas excede el límite, reutiliza la foto más antigua. También maneja efectos visuales y asegura que las fotos instanciadas tengan un estado inicial adecuado.

En resumen, la clase CameraTool gestiona la captura de fotos en un entorno de realidad virtual, permitiendo a los jugadores tomar fotos, verlas en el mundo virtual, y guardarlas en el almacenamiento del dispositivo. Además, incluye funcionalidades de zoom y efectos visuales y sonoros asociados con la toma de fotos.

GpsTool

El script GpsTool simula una herramienta de GPS que se activa para interactuar con objetos arqueológicos. Utiliza un raycast para detectar estos objetos en el entorno, y cuando el jugador apunta a un objeto arqueológico específico, inicia un minijuego que consiste en controlar un anillo expansible dentro de un área determinada. El objetivo del minijuego es mantener el anillo dentro de un círculo en expansión durante tres intentos consecutivos. Si el jugador lo logra, se completa un análisis del objeto arqueológico, se activan partículas de estrellas y se registra el análisis. Si el jugador falla, el minijuego se reinicia.

Descripción de los Métodos

1. **Start()**
 - Inicializa las configuraciones y las referencias necesarias para el funcionamiento del GPS. Esto incluye configurar el LineRenderer para mostrar el rayo de detección, inicializar las entradas del jugador y preparar el círculo de minijuego. También desactiva las manos del jugador al inicio.
2. **Update()**
 - Se ejecuta cada frame y maneja el estado de la herramienta. Si el jugador agarra la herramienta (isGrab), se habilitan los elementos visuales del minijuego (círculo, texto, etc.) y se dispara un rayo desde la herramienta para detectar objetos arqueológicos en el mundo del juego. Si se encuentra un objeto, se gestiona el minijuego. Si no se encuentra nada o se han cumplido los intentos, el minijuego se resetea.
3. **HandleRaycast()**
 - Dispara un rayo desde el pointRay para detectar objetos dentro del rango especificado. Si el rayo golpea un objeto con la etiqueta "ArchaeologicalItem" y este no ha sido analizado previamente, activa el minijuego. Si el objeto ya ha sido analizado, simplemente reinicia el minijuego sin realizar ninguna acción.
4. **StartMiniGame(Transform target)**
 - Inicia el minijuego cuando se apunta a un objeto arqueológico válido. Configura el objetivo actual y restablece las condiciones iniciales del minijuego, como el color y la escala del círculo del anillo.
5. **PlayMiniGame()**
 - Gestiona la lógica del minijuego. Se encarga de la expansión del anillo y verifica si está dentro de los límites del círculo objetivo. Si el anillo se expande demasiado, el jugador pierde y el minijuego se reinicia. Si el jugador está dentro del rango, se continúa el juego hasta completar los intentos.
6. **OnMiniGameInput(InputAction.CallbackContext context)**
 - Responde a la entrada del jugador (por ejemplo, presionar un botón). Si el jugador está en el minijuego y el anillo está dentro del rango adecuado, aumenta el contador de intentos exitosos. Si el jugador completa el número requerido de intentos exitosos, se completa el análisis del objeto. Si el jugador falla, el minijuego se reinicia.
7. **CompleteAnalysis()**

- Completa el análisis de un objeto arqueológico cuando el jugador ha tenido suficientes intentos exitosos. Muestra efectos visuales de partículas y registra el análisis del objeto en la actividad arqueológica.
- 8. **ResetMiniGame()**
 - Reinicia todos los elementos del minijuego. Esto incluye reiniciar el número de intentos, el círculo expansible, el color de la herramienta y la bandera del estado del minijuego.
- 9. **ChangeColor(Color color)**
 - Cambia el color del círculo visual que indica el estado del minijuego. Utiliza una corrutina para gestionar el cambio de color con un pequeño retraso si es necesario.
- 10. **ChangeColorCorutine(Color color)**
 - Una corrutina que gestiona de forma secuencial el cambio de color del círculo, permitiendo un breve retraso cuando el color es el inicial (azul). Esto mejora la experiencia visual durante la transición entre los diferentes estados del minijuego.

Resumen de Componentes Principales

- **Variables de configuración:**
 - Controlan aspectos como la distancia del raycast, los colores de los estados del minijuego, la velocidad de expansión del anillo, y los umbrales para determinar si el jugador ha fallado o tenido éxito en el minijuego.
- **Referencias externas:**
 - Se utilizan referencias a componentes como LineRenderer para visualizar el raycast, TextMeshPro para mostrar el progreso del jugador en el minijuego, y AudioManager para reproducir sonidos durante las interacciones.
- **Interacción con la física:**
 - Utiliza un Ray para detectar objetos dentro del mundo del juego. El minijuego se basa en la manipulación de la escala del objeto circleGame (el anillo expansible) para simular la interacción con el jugador.
- **Interacción con el jugador:**
 - Usa PlayerInput para manejar las entradas del jugador y activar el minijuego mediante la acción de presionar un botón.
 -

3. Mission

La carpeta Mission contiene los scripts de: FishAI y Mission y se divide en 5 subcarpetas cada una correspondiente a una actividad.

Mission:

Este script gestiona las actividades que deben completarse, y una vez completadas, carga el siguiente nivel o escena. Está diseñado para ser usado dentro de un flujo de juego donde, al completar una tarea o misión, se procede a la siguiente etapa.

Descripción de los Métodos

1. Start()

- Este método se llama al inicio del ciclo de vida del script. Busca un objeto de tipo Watch en la escena y lo asigna a la variable `_watch`. Esto puede ser útil si se requiere realizar un seguimiento del tiempo o una actividad relacionada más adelante en el juego.

2. CompleteActivity()

- Este método se invoca para indicar que la actividad o misión se ha completado. Cambia el valor de la variable `activityFinished` a `true` y luego carga el siguiente nivel, usando el método `LoadNextLevel` del objeto `levelLoader`, pasando el nombre de la escena "Lobby" como argumento.

Resumen de Componentes Principales

- **Variables de configuración:**
 - `activityFinished`: Indica si la actividad se ha completado o no.
 - `levelLoader`: Un objeto encargado de gestionar la carga de niveles en el juego.
 - `_watch`: Referencia a un objeto de tipo Watch, posiblemente utilizado para monitorear el tiempo o progreso de la actividad.
- **Interacción con otros componentes:**
 - El método `CompleteActivity()` interactúa con `levelLoader` para cargar un nuevo nivel ("Lobby") cuando se completa la misión.
 - La variable `_watch` es asignada durante el inicio para ser usada potencialmente en la lógica de la actividad.

Este script facilita la gestión de las transiciones entre niveles y el control del estado de la actividad, permitiendo que el juego avance automáticamente una vez que se haya completado una tarea específica.

FishAI:

Este script gestiona el comportamiento de un pez en la aplicación, controlando su movimiento y navegación a lo largo de un camino generado por nodos. Utiliza un sistema de búsqueda de caminos para determinar el siguiente objetivo del pez y calcula su trayectoria

para moverlo a lo largo del recorrido. El pez puede moverse en estado de "nadar" o "detenerse" en función de si ha llegado a su destino.

Descripción de los Métodos

1. **Start()**
 - Inicializa el nodo de inicio del pez, buscando el nodo más cercano a la posición actual del pez. También inicializa la lista de nodos path que representará el camino a seguir.
2. **Update()**
 - Se ejecuta en cada frame del juego. Si el pez está en el estado de movimiento "Swimming", se llama al método MoveAlongPath() para mover al pez a lo largo del camino.
3. **StartPath()**
 - Inicia el cálculo del camino hacia un objetivo. Si se ha especificado un target, el método obtiene el nodo de inicio y fin usando el gestor de caminos (PathfindingManager) y luego genera un recorrido entre estos nodos. Si no hay objetivo, busca un nodo aleatorio dentro de un rango específico para que el pez lo siga.
4. **FindRandomNodeAsTarget()**
 - Busca un nodo aleatorio dentro de un rango definido (minTargetDistance y maxTargetDistance) para usarlo como nuevo objetivo para el pez. Solo selecciona nodos que se encuentren a una distancia válida.
5. **MoveAlongPath()**
 - Este método gestiona el movimiento del pez a lo largo del camino:
 - Verifica si el pez ha llegado a un nodo (basado en un umbral de distancia).
 - Si está cerca de un nodo, lo elimina del camino y se mueve al siguiente.
 - Si el pez no está cerca de ningún nodo, lo mueve hacia el siguiente nodo en la lista path, calculando la dirección y ajustando la rotación del pez hacia esa dirección.
 - Si el pez ha terminado de recorrer el camino, cambia su estado de movimiento a "Stopped".

Resumen de Componentes Principales

- **Variables de configuración:**
 - speed: Controla la velocidad de movimiento del pez.
 - reachThreshold: Define la distancia mínima para considerar que el pez ha llegado a un nodo.
 - minTargetDistance y maxTargetDistance: Establecen el rango dentro del cual el pez puede buscar nuevos nodos.
 - currentStateMovement: Controla el estado actual del pez, ya sea nadando ("Swimming") o detenido ("Stopped").
- **Referencias externas:**
 - target: Representa el objetivo actual del pez.
 - startNode y endNode: Nodos que marcan el inicio y el fin del camino del pez.

- path: Lista de nodos que forman el camino que el pez sigue.
- PathfindingManager: Un gestor que calcula los nodos más cercanos y el camino entre ellos.
- **Interacción con física y movimiento:**
 - El movimiento se calcula usando la distancia entre el pez y los nodos del camino, y se aplica una dirección normalizada para moverlo hacia el siguiente nodo.
 - También se ajusta la rotación del pez para que siempre mire hacia la dirección del movimiento.

Este script simula un comportamiento de navegación en un entorno de juego, permitiendo que el pez siga un camino de nodos y se detenga cuando haya completado su recorrido.

3.1 Lobo Marino

La carpeta Lobo Marino contiene los scripts de: Lobo Marino Behaviour, Lobos Activity Spots y Lobos Marinos Activity

Lobo Marino Behaviour

Este script controla el comportamiento del lobo marino, incluyendo la navegación entre diferentes "spots" en el entorno, como rocas, algas y esponjas. El lobo marino sigue un camino aleatorio hacia estos puntos, realiza un giro en su movimiento y reproduce efectos de sonido relacionados con su nado.

Descripción de los Métodos:

1. **Start()**
 - Inicializa los componentes necesarios, como el animador y el evento de sonido. Obtiene los spots desde el objeto padre en la escena y comienza a mover al lobo marino hacia el primer spot.
2. **Update()**
 - Llama a la actualización de la clase base FishAI para controlar el movimiento del lobo marino.
3. **ChangeSpot()**
 - Selecciona un nuevo spot aleatorio en el que el lobo marino se moverá.
4. **GetNextSpot()**
 - Devuelve un nuevo spot aleatorio, asegurándose de que no sea el mismo que el anterior.
5. **MoveToSpot(Transform spot)**
 - Inicia el movimiento hacia un spot, comenzando el cálculo del camino y asegurándose de que el lobo marino llegue al punto antes de cambiar al siguiente.
6. **GetRandomPointInsideCollider(Collider collider)**
 - Genera un punto aleatorio dentro de los límites de un Collider, que representa un área en el mundo donde el lobo marino puede moverse.
7. **PerformFlipRoutine()**

- Realiza un giro (flip) después de un tiempo aleatorio, reduciendo gradualmente la velocidad y reproduciendo efectos de sonido durante el proceso.
- 8. **OnDrawGizmos()**
 - Dibuja un Gizmo en la posición del objetivo (target), útil para la visualización en el editor.
- 9. **SwimingSfxOn()**
 - Inicia el sonido del nado configurando sus atributos 3D y comenzando la reproducción si no está en reproducción.
- 10. **SwimingSfxOff()**
 - Detiene el sonido del nado si está en reproducción.

Resumen de Componentes Principales:

- **Variables de configuración:** Controlan la lista de spots, el estado del spot actual, la velocidad del lobo marino y si está realizando un giro.
- **Referencias externas:** Utiliza un Animator para los efectos visuales, AudioManager para efectos de sonido y FmodEvents para gestionar la reproducción de sonidos en 3D.
- **Interacción con física y animación:** El movimiento hacia los spots y la animación de giro se gestionan mediante el control de los estados y transiciones de animaciones en el componente Anima .

Lobos Activity Spots:

Este script controla el comportamiento de los spots en los que el lobo marino puede interactuar. Detecta cuando el lobo marino entra o sale de un área determinada y cambia su estado correspondiente.

Descripción de los Métodos:

1. **OnTriggerEnter(Collider other)**
 - Este método es llamado cuando otro objeto entra en el área del Collider del objeto que tiene este script. Si el objeto que entra tiene el tag "LoboMarino", busca el componente LoboMarinoBehaviour en ese objeto. Si lo encuentra, actualiza el estado del spot en el lobo marino al valor del estado del spot actual (SpotState).
2. **OnTriggerExit(Collider other)**
 - Este método es llamado cuando otro objeto sale del área del Collider del objeto que tiene este script. Si el objeto que sale tiene el tag "LoboMarino", busca el componente LoboMarinoBehaviour en ese objeto. Si lo encuentra, restablece el estado del spot del lobo marino a None, indicando que ya no está en el spot.

Resumen de Componentes Principales:

- **Variables de configuración:** La variable SpotState es un enumerado que representa el estado del spot (por ejemplo, si es de rocas, algas, esponjas, etc.).

- **Interacción con otros objetos:** Utiliza los métodos OnTriggerEnter y OnTriggerExit para detectar cuando el lobo marino entra o sale del área del spot. El SpotState del lobo marino se actualiza en consecuencia, lo que probablemente afecta su comportamiento y las interacciones con el entorno.
- **Referencias externas:** El script se basa en el componente LoboMarinoBehaviour para modificar el estado del lobo marino al interactuar con los spots.

Este script se adjunta a los objetos de la escena que actúan como los spots de interés, y ayuda a manejar las interacciones del lobo marino con esos spots.

Lobo Marino Activity

Este script hereda de Mission y es quién administra la actividad de “La Danza de los Lobos marinos”.

Descripción de los Métodos:

1. **Start()**
 - Se inicializan las variables de traducción para los títulos y descripciones de las tareas, usando un sistema de localización para los textos.
 - Se suscribe al evento OnPhotoTaken de CameraTool, que indica que una foto ha sido tomada, y se llama a la función TryToAddAnimal para procesar la foto.
 - Se crean las tareas usando el Watch, un sistema de seguimiento de tareas, que irá registrando el progreso de las misiones de los lobos marinos.
2. **OnDestroy()**
 - Se desuscribe del evento OnPhotoTaken de CameraTool para evitar posibles errores cuando el objeto que tiene este script se destruye.
3. **TryToAddAnimal(Transform[] animals)**
 - Este método se llama cuando se toma una foto (cuando OnPhotoTaken es disparado). Se verifica si los objetos en la foto tienen el tag "LoboMarino".
 - Dependiendo de las condiciones (si el lobo marino hizo un "flip", si hay al menos dos lobos marinos, o si un lobo marino está en el área de algas), se actualizan las tareas y el progreso de la misión en el sistema de seguimiento.
4. **CheckCompletion()**
 - Este método comprueba si todas las tareas de la misión están completas. Si lo están, llama a CompleteActivity(), lo que probablemente marca la misión como completada.

Resumen de Componentes Principales:

- **Variables de configuración:**
 - loboMarinoTag: El tag para identificar a los lobos marinos.
 - cameraTool: El objeto que maneja la cámara y las fotos.
 - Claves de localización: Las claves como loboMarinoFlipTitleKey, loboMarino2TitleKey, y loboMarinoAlgasTitleKey son utilizadas para obtener las traducciones de los títulos y descripciones de las tareas.
- **Sistema de Tareas:**

- Se crean tres tareas para el seguimiento del progreso del jugador: una para capturar un "flip" de un lobo marino, otra para capturar dos lobos marinos, y la última para capturar un lobo marino en el área de algas.
- **Interacción con el Sistema de Fotos:**
 - La función TryToAddAnimal procesa las fotos tomadas por el jugador y determina si cumplen con los requisitos para completar una de las tareas de la misión.
- **Verificación de Completitud:**
 - La misión se completa cuando todas las tareas han sido completadas.

3.2 Limpieza

La carpeta Limpieza contiene los scripts de: CleaningActivity y TrashItem

CleaningActivity

Este script está diseñado para gestionar la actividad de "Limpieza del Onashaga", en la que el jugador debe recolectar diferentes tipos de basura (botellas, ruedas, madera) para completar la misión.

Descripción de los Métodos:

1. **Start()**
 - Se inicializan las claves de localización para los títulos y descripciones de las tareas de la actividad de limpieza.
 - Se crean las tareas para recolectar botellas, ruedas y madera usando un sistema de seguimiento de tareas, donde se muestra el progreso de recolección.
2. **OnTriggerEnter(Collider other)**
 - Este método se activa cuando un objeto entra en el área de detección del jugador (como un collider).
 - Si el objeto tiene el componente TrashItem (que representa un ítem de basura), se suma al contador correspondiente según el tipo de basura (botellas, ruedas, madera).
 - Luego, se instancian partículas (probablemente un efecto visual) y se desactiva el objeto de basura.
 - Se imprime el progreso de la recolección en la consola para fines de depuración.
3. **TryToAddTrash(TrashItem trashItem)**
 - Este método recibe un objeto TrashItem y determina el tipo de basura. Dependiendo de su tipo (TrashType.Bottle, TrashType.Wheel, TrashType.Wood), se actualiza el contador correspondiente y el progreso de la tarea en el sistema de seguimiento.
 - Cada vez que se recolecta un ítem, se actualiza el progreso de la tarea correspondiente, mostrando la cantidad recolectada hasta el momento.
4. **CheckCompletion()**
 - Este método calcula si se ha completado la actividad de limpieza. Si la cantidad total de basura recolectada (botellas, ruedas, madera) es igual o

superior a la cantidad requerida para completar la misión, se llama a `CompleteActivity()`, marcando la misión como completada.

Resumen de Componentes Principales:

- **Variables de Configuración:**
 - `requiredBottlesToCompleteTask`, `requiredWheelsToCompleteTask`, `requiredWoodToCompleteTask`: Cantidades necesarias de botellas, ruedas y madera para completar la misión.
 - `bottlesCollected`, `netsCollected`, `wheelsCollected`, `woodsCollected`: Variables que llevan el seguimiento de la cantidad de basura recolectada por el jugador.
- **Sistema de Localización:**
 - Se usan claves de localización (`bottleTitleKey`, `wheelTitleKey`, etc.) para traducir los textos asociados con cada tipo de basura.
- **Partículas:**
 - `starParticles`: Efectos visuales que se instancian cada vez que el jugador recolecta un ítem de basura.
- **Seguimiento de Tareas:**
 - Cada tipo de basura tiene una tarea asociada en el sistema de seguimiento (`_watch`), que se actualiza cada vez que se recoge basura.
 - El progreso se muestra en formato de texto como "0/3" o "0/1" para indicar la cantidad de basura recolectada en comparación con la cantidad requerida.
- **Condición de Compleción:**
 - La actividad se marca como completa cuando el jugador ha recolectado la cantidad necesaria de cada tipo de basura.

Este script es ideal para una misión de recolección en un juego, donde el jugador debe completar tareas específicas y ver su progreso visualmente. También incluye una forma eficiente de manejar los tipos de basura y las condiciones para completar la misión.

TrashItem

Este script define un objeto de tipo **TrashItem** (objeto de basura), que hereda de `GrabbableObject`.

Descripción de los Componentes del Script:

1. **trashType (Enumeración de tipo TrashType)**
 - **Propósito:** Define el tipo de basura que representa este objeto.
 - **Descripción:** El tipo de basura se gestiona mediante la enumeración `TrashType`, lo que permite especificar diferentes tipos de basura, como botellas, ruedas o madera. Este campo es público, por lo que se puede asignar y modificar desde el editor o en tiempo de ejecución.
2. **id (Identificador único para el objeto de basura)**
 - **Propósito:** Este campo parece servir como identificador único para cada objeto de basura.
 - **Descripción:** El campo `id` es un entero que podría ser utilizado para distinguir cada instancia de `TrashItem` en el juego. Esto es útil si hay

múltiples objetos de basura del mismo tipo y es necesario llevar un seguimiento individual de ellos.

3. **Start()**

- **Propósito:** Inicialización del objeto.
- **Descripción:** Este método sobrescribe el método `Start()` de la clase base `GrabbableObject`. En este caso, parece que no se realiza ninguna acción adicional en `Start()`, ya que simplemente llama a `base.Start()` para ejecutar la lógica de inicialización de la clase base. Esto indica que este objeto probablemente hereda funcionalidad relacionada con la interacción o manipulación de objetos.

Resumen de Componentes Principales:

- **trashType:** Un campo público de tipo `TrashType`, que determina qué tipo de basura es este objeto (por ejemplo, botella, rueda, madera). Esta enumeración se utilizará en otro código para identificar y gestionar la basura recolectada.
- **id:** Un identificador único para cada objeto de basura, lo que permite hacer un seguimiento individual de cada instancia en el juego. Este campo podría ser útil si se requiere diferenciar entre varias instancias de la misma clase.
- **Start():** Este método se sobrescribe para llamar a la inicialización de la clase base `GrabbableObject`.

Resumen: Este script es una clase base para un objeto de basura que puede ser recolectado o manipulado por el jugador, y está diseñado para ser flexible, permitiendo diferentes tipos de basura y una identificación única de cada instancia.

3.3 Hidden

La carpeta `Hidden` contiene los scripts de: `ErmitañoBehaviour`, `PulpoBehaviour`, `ToritoBehaviour` y `HiddenActivity`.

HiddenActivity

Este script, que hereda de la clase `Mission`, es el que administra la actividad de “Escondidas”. En esta actividad, el objetivo es fotografiar una serie de animales únicos, y se controla el progreso mediante un sistema de tareas. El código gestiona las interacciones con la herramienta de cámara y los animales fotografiados.

Componentes del Script:

1. **Campos de Configuración:**
 - **requiredAnimalsToCompleteTask:** Define el número de animales únicos que deben ser fotografiados para completar la actividad (en este caso, 3).
 - **uniqueAnimalCount:** Lleva la cuenta de los animales únicos fotografiados.
2. **Tags de los Animales:**
 - **toritoTag, pulpoTag, ermitañoTag:** Se definen los tags para identificar los diferentes tipos de animales a fotografiar (un pez torito, un pulpo y un ermitaño).
3. **Herramienta de Cámara:**

- **cameraTool**: Referencia a la herramienta de cámara que se usa para capturar fotos de los animales.
- 4. **Claves de Traducción:**
 - **toritoTitleKey, toritoDescriptionKey, ermitañoTitleKey, ermitañoDescriptionKey, pulpoTitleKey, pulpoDescriptionKey**: Claves para obtener las traducciones de los títulos y descripciones de las tareas correspondientes a los animales a fotografiar.
 - Estas claves se traducen a cadenas de texto en diferentes idiomas utilizando un sistema de localización (probablemente con la ayuda de un sistema de traducción como I2 Localization).
- 5. **Lista de Animales Fotografiados:**
 - **_photographedAnimals**: Una lista que guarda las etiquetas de los animales que ya han sido fotografiados para evitar duplicados.
- 6. **Métodos Clave:**
 - **Start()**:
 - Se ejecuta al inicio de la actividad. Se traduce el título y la descripción de cada tarea usando las claves de localización.
 - Se suscribe al evento OnPhotoTaken de la herramienta de cámara para ejecutar la función TryToAddAnimal cada vez que se tome una foto.
 - Se crea una tarea para cada animal que debe ser fotografiado, utilizando el sistema de tareas (_watch).
 - **OnDestroy()**:
 - Desuscribe la función del evento OnPhotoTaken para evitar errores si el objeto es destruido.
 - **TryToAddAnimal(Transform[] animals)**:
 - Este método se ejecuta cuando se toma una foto y se le pasan los animales fotografiados.
 - Verifica si el animal ya ha sido fotografiado, y si no es así, lo añade a la lista de animales fotografiados y actualiza la tarea correspondiente (torito, ermitano o pulpo).
 - Si el animal corresponde a un tipo específico, se actualiza el progreso de la tarea de ese animal y marca la foto como exitosa.
 - **CheckCompletion()**:
 - Revisa si el número de animales únicos fotografiados ha alcanzado el objetivo necesario para completar la actividad.
 - Si se alcanza el número requerido de animales, se llama al método CompleteActivity() para finalizar la misión.

Resumen de Componentes Principales:

1. **Campos Públicos:**
 - **requiredAnimalsToCompleteTask**: Número total de animales únicos que se deben fotografiar.
 - **cameraTool**: Referencia a la herramienta de cámara que dispara las fotos.
 - **toritoTag, pulpoTag, ermitanoTag**: Los tags que identifican a los animales a fotografiar.

- **Claves de Localización:** Se usan para traducir los títulos y descripciones de las tareas relacionadas con los animales.
2. **Métodos Importantes:**
- **Start():** Inicializa las tareas y suscribe el evento de la cámara.
 - **OnDestroy():** Elimina la suscripción del evento para evitar errores cuando el objeto es destruido.
 - **TryToAddAnimal():** Verifica y actualiza las tareas basadas en los animales fotografiados.
 - **CheckCompletion():** Verifica si se han completado todas las tareas necesarias.

Comportamiento:

- El jugador tiene la tarea de fotografiar animales específicos (torito, pulpo, ermitaño).
- Cada vez que se toma una foto, se verifica si el animal es nuevo y se actualiza el progreso de la actividad.
- Si se alcanzan los requisitos de animales fotografiados, la actividad se completa.

ToritoBehaviour

Este script controla el comportamiento del “torito de los canales”, como su interacción con el jugador y su capacidad para moverse entre diferentes estados (escondarse, asomarse, nadar y huir) y sonidos relacionados con su actividad. A continuación, se describe el funcionamiento de los métodos y componentes clave.

Descripción de los Métodos:

Start() Inicializa los componentes esenciales para el comportamiento del pez:

- Obtiene la referencia al jugador.
- Obtiene todos los puntos de escondite de los hijos de un objeto contenedor en la escena.
- Crea la instancia del evento de sonido FMOD para el nadar.
- Inicializa una lista de escondites y selecciona uno aleatoriamente para que el pez comience en el estado de "escondido".

Update() Se ejecuta cada frame:

- Verifica si el jugador está cerca del pez o se está moviendo rápidamente dentro de los radios definidos.
- Si el pez está nadando y detecta una proximidad al jugador o un movimiento rápido, cancela la acción actual y empieza a huir hacia un nuevo escondite.

HideInSpot(Vector3 hidingSpot) Controla la acción de esconderse:

- Mueve al pez a un escondite y espera un tiempo aleatorio antes de cambiar al siguiente estado.
- Si no se le pasa un punto específico, selecciona un nuevo escondite aleatorio.

PeekOut() El pez saca la cabeza de su escondite:

- Cambia su estado a "asomarse" durante un tiempo determinado (definido por peekTime).
- Después de ese tiempo, el pez cambia al estado de nadar.

SwimAround() El pez nada por el entorno:

- Elige un nuevo escondite y comienza a nadar hacia él, reproduciendo el sonido de nadado (FMOD).
- El pez sigue nadando hasta que su estado de movimiento cambie.

FleeToHidingSpot() El pez huye hacia un nuevo escondite cuando el jugador lo asusta:

- Reproduce un sonido de escape y comienza a nadar hacia un nuevo escondite.
- Una vez llegue al escondite, vuelve al estado de "esconderse".

ChooseNewHidingSpot() Elige un nuevo escondite para el pez:

- Crea una lista de escondites disponibles, excluyendo el actual.
- Elige aleatoriamente un nuevo escondite de la lista.

UpdateFishSpeed() Actualiza la velocidad del pez según su estado:

- Si el pez está nadando, se establece la velocidad de nado.
- Si el pez está huyendo, se establece la velocidad de huida.
- Si está escondido o asomándose, se detiene.

IsPlayerMovingFast() Verifica si el jugador está moviéndose rápido dentro de un radio mayor:

- Usa un SphereCast para verificar si el jugador está dentro del radio y se mueve rápidamente.

IsPlayerClose() Verifica si el jugador está cerca:

- Usa un SphereCast para comprobar si el jugador está dentro de un radio más pequeño y cercano.

OnDrawGizmosSelected() Visualiza en el editor los radios de proximidad para ver los rangos de detección de jugador.

SwimingSfxOn() Activa el sonido del nado:

- Configura los atributos 3D del sonido, asociándolo con la posición del pez.
- Si el sonido no está en reproducción, lo inicia con un pitch aleatorio.

SwimingSfxOff() Detiene el sonido del nado si está en reproducción.

Resumen de Componentes Principales:

Variables de configuración:

- hidingSpots[]: Lista de puntos donde el pez puede esconderse.
- currentState: Estado actual del pez (escondido, asomándose, nadando, huyendo).
- scaredRadius y wideRadius: Radios de detección para el jugador cerca y en movimiento rápido.
- swimSpeed y scaredSpeed: Velocidades para nadar y huir.

Referencias externas:

- AudioManager: Se utiliza para crear y gestionar el evento de sonido para nadar.
- FmodEvents: Reproduce el sonido de escape y nadado.
- Transform _playerTransform: Referencia al jugador para comprobar su cercanía y movimiento.

Interacción con el entorno:

- El pez interactúa con su entorno eligiendo escondites, nadando hacia ellos y respondiendo al jugador con acciones como huir o asomarse.
- Utiliza raycast para detectar la proximidad y el movimiento del jugador.

Este script es una implementación para un comportamiento de inteligencia artificial (IA) que permite que el pez actúe de forma autónoma, reaccionando al entorno y al jugador con diferentes comportamientos y sonidos, dependiendo de su estado actual.

PulpoBehaviour

Este código controla el comportamiento del pulpo colorado, donde el pulpo se mueve alrededor de un punto central en una serie de puntos generados en un círculo. Los movimientos del pulpo están impulsados por un sistema de física utilizando un Rigidbody, y el pulpo rota suavemente hacia el siguiente punto.

Descripción de los Métodos

1. **Start()**
 - Inicializa el Rigidbody del pulpo y genera los puntos alrededor de un círculo.
 - Si hay puntos generados, selecciona aleatoriamente uno como punto de inicio y coloca al pulpo en esa posición.
2. **GeneratePoints()**
 - Genera un número específico de puntos (numberOfPoints) alrededor del pulpo, distribuidos en un círculo con un radio definido (radius).
 - Los puntos se calculan usando trigonometría, manteniendo la misma altura que la posición inicial del pulpo.
3. **FixedUpdate()**
 - Se ejecuta en cada frame de la simulación de física y realiza dos acciones:
 - Llama a CheckArrival() para verificar si el pulpo ha llegado al punto actual y, si es así, cambiar al siguiente punto.
 - Llama a RotateTowardsNextPoint() para rotar el pulpo suavemente hacia el siguiente punto.

4. **ApplyImpulse()**
 - Calcula la dirección hacia el siguiente punto y aplica un impulso utilizando el Rigidbody, simulando el movimiento del pulpo hacia ese punto.
 - La fuerza del impulso está determinada por la variable force.
5. **CheckArrival()**
 - Verifica si el pulpo ha llegado cerca del punto actual (distancia menor que el umbral arrivalThreshold).
 - Si es así, cambia al siguiente punto. Si isAscending es verdadero, el pulpo avanza al siguiente punto de manera ascendente; si es falso, retrocede al anterior de manera descendente.
6. **RotateTowardsNextPoint()**
 - Calcula la dirección hacia el siguiente punto y ajusta la rotación del pulpo para que siempre apunte hacia ese punto.
 - La rotación se realiza suavemente utilizando interpolación esférica (Quaternion.Slerp), asegurando un movimiento fluido.
7. **OnDrawGizmos()**
 - Se ejecuta en el editor de Unity para la depuración. Dibuja los puntos generados alrededor del pulpo y las líneas entre ellos para visualizar el patrón en el cual se mueve el pulpo.
 - Utiliza el sistema de Gizmos para dibujar esferas en los puntos generados y líneas entre los puntos.

Resumen de Componentes Principales

- **Variables de Configuración:**
 - numberOfPoints: Determina cuántos puntos generará el pulpo alrededor de un círculo.
 - radius: Define el radio de ese círculo.
 - force: Controla la fuerza del impulso que mueve al pulpo hacia el siguiente punto.
 - arrivalThreshold: Establece la distancia mínima a la cual el pulpo es considerado "llegado" a un punto.
 - isAscending: Define si el pulpo avanza ascendentemente o desciende entre los puntos.
- **Referencias a Componentes:**
 - Rigidbody: Utilizado para aplicar física y movimiento mediante fuerzas.
 - Vector3[] points: Arreglo que almacena los puntos generados alrededor del pulpo.
- **Interacción con Física:**
 - Usa un Rigidbody para mover al pulpo mediante impulsos físicos hacia los puntos generados.

Este comportamiento da como resultado un pulpo que se mueve en un patrón circular alrededor de su posición inicial, aplicando impulsos físicos y rotando suavemente hacia cada punto.

ErmitañoBehaviour

Este script controla el comportamiento de un ermitano que patrulla una zona, se esconde cuando detecta al jugador y regresa a patrullar una vez que el jugador se aleja. Utiliza un sistema de navegación (NavMesh) para moverse por el terreno y tiene un sistema de animación para simular su comportamiento, como esconderse en su caparazón. A continuación, se describen los métodos y su funcionamiento.

Descripción de los Métodos

1. **Start()**
 - Inicializa la posición inicial del ermitano y busca la referencia del jugador.
 - Obtiene el componente NavMeshAgent para mover al ermitano y establece la velocidad de movimiento.
 - Llama a SetNewRandomTarget() para determinar un punto de patrullaje aleatorio al inicio.
2. **Update()**
 - Se ejecuta cada frame y controla el comportamiento del ermitano en función de su estado actual (patrullar, esconderse o regresar a patrullar).
 - Dependiendo del estado (Patrol, Hide, ReturnToPatrol), llama a los métodos correspondientes para manejar la lógica.
3. **Patrol()**
 - Hace que el ermitano se mueva hacia un objetivo aleatorio.
 - Si alcanza el objetivo (stopDistance), genera un nuevo objetivo aleatorio utilizando SetNewRandomTarget().
4. **CheckPlayerDistance()**
 - Verifica la distancia entre el ermitano y el jugador.
 - Si el jugador está dentro del detectionRadius, cambia el estado a Hide y comienza un temporizador para el escondite (hideTimer).
5. **Hide()**
 - Cambia al estado de esconderse.
 - Desactiva el movimiento al detener el NavMeshAgent.
 - Reduce la escala del modelo para simular que el ermitano se esconde en su caparazón (a través de la animación).
 - Decrementa el temporizador de escondite. Si el temporizador llega a cero y el jugador aún está cerca, reinicia el temporizador. Si el jugador se aleja, el estado cambia a ReturnToPatrol.
6. **ReturnToPatrol()**
 - Cambia al estado de patrullaje después de que el ermitano ha salido de su escondite.
 - Si el modelo ha vuelto a su escala original, el estado se cambia a Patrol y el ermitano retoma la patrulla.
7. **MoveTowardsTarget()**
 - Mueve al ermitano hacia el punto objetivo utilizando el NavMeshAgent.
8. **SurfaceAlignment()**
 - Realiza un raycast desde la posición del modelo hacia abajo para detectar la superficie del terreno.

- Ajusta la rotación del modelo para alinearlo con la superficie, usando una interpolación suave (Slerp) para evitar rotaciones bruscas.
9. **SetNewRandomTarget()**
- Genera un nuevo punto de patrullaje aleatorio dentro de un área determinada por `areaSize` alrededor de la posición inicial del ermitano.
 - Si el punto aleatorio no está sobre el terreno, vuelve a intentar generar uno nuevo.
10. **OnDrawGizmos()**
- Dibuja en el editor los puntos de destino, el radio de detección del jugador y los raycasts utilizados para alinear la rotación del modelo con la superficie del terreno.
 - Ayuda a depurar visualmente el comportamiento del ermitano, mostrando las posiciones y los rayos de detección.

Resumen de Componentes Principales

- **Variables de configuración:** Controlan la velocidad de movimiento, la distancia de detección, el radio de patrullaje, el tiempo de escondite, y la curva de animación para la rotación.
- **Componente NavMeshAgent:** Utilizado para el movimiento del ermitano sobre el terreno, asegurando que se pueda mover de manera adecuada utilizando el sistema de navegación.
- **Animación:** Usa un Animator para gestionar las animaciones del ermitano, incluyendo el estado de escondite (al reducir su escala) y la salida del caparazón.
- **Detección del jugador:** A través de un Raycast, se verifica la proximidad del jugador. Si el jugador entra en el radio de detección, el ermitano se esconde y permanece en ese estado hasta que el jugador se aleja lo suficiente.

Este script simula un comportamiento de patrullaje y reacción a la presencia del jugador, haciendo que el ermitano se oculte cuando lo detecta y vuelva a su patrullaje cuando la amenaza desaparece.

3.4 Fish

La carpeta Fish contiene los scripts de: FishActivity, NototeniaBehaviour, CormoranBehaviour.

FishActivity

Este script hereda de la clase Mission y es responsable de gestionar la actividad "Pescando con Cormoranes". En esta actividad, el objetivo es fotografiar a los cormoranes cuando están realizando dos acciones específicas: buceando y saliendo a la superficie. El progreso se maneja a través de un sistema de tareas que se actualizan al tomar fotos en el momento adecuado. A continuación, se describen los componentes y métodos clave del código:

Componentes del Script:

Campos de Configuración:

- **cormoranTag**: Define la etiqueta usada para identificar al cormorán en el mundo del juego.
- **cameraTool**: Hace referencia a la herramienta de cámara que se usa para capturar las fotos de los cormoranes.
- **cormoranTitleKey, cormoranDescriptionKey, cormoran2TitleKey, cormoran2DescriptionKey**: Son claves de localización que permiten obtener los títulos y descripciones de las tareas para las fotos de los cormoranes, en diferentes idiomas.
- **_cTitleKey, _cDescriptionKey, _c2TitleKey, _c2DescriptionKey**: Estas variables almacenan los valores traducidos de las claves de localización.

Variables de Estado:

- **_cormoranTaskComplete** y **_cormoran2TaskComplete**: Indicadores booleanos que marcan si las tareas asociadas a los cormoranes han sido completadas (una para cuando el cormorán está buceando y otra para cuando está saliendo a la superficie).

Métodos Clave:Start()

Este método se ejecuta al inicio de la actividad:

- Se traduce el título y la descripción de las tareas utilizando las claves de localización proporcionadas.
- Se suscribe al evento OnPhotoTaken de la herramienta de cámara. Cada vez que se toma una foto, se ejecutará la función TryToAddAnimal para verificar si el animal fotografiado es un cormorán y si cumple con los requisitos de la tarea.
- Se crean dos tareas utilizando el sistema de seguimiento de tareas (_watch): una para la tarea del cormorán buceando y otra para la tarea del cormorán saliendo a la superficie.

OnDestroy()

Este método se ejecuta cuando el objeto es destruido:

- Se desuscribe la función `TryToAddAnimal` del evento `OnPhotoTaken` para evitar errores si el objeto es destruido antes de que se complete la actividad.

`TryToAddAnimal(Transform[] animals)`

Este método se ejecuta cuando se toma una foto y se pasa un array de animales que han sido fotografiados:

- Se recorre la lista de animales fotografiados y se verifica si alguno tiene la etiqueta `cormoranTag`.
- Si el animal está realizando la acción de "buceo", se marca la tarea asociada a esa acción como completada (actualiza el progreso de la tarea y marca la foto como exitosa).
- Si el animal está "salvando" desde el agua, se actualiza la segunda tarea relacionada con esa acción.
- Si ambos estados (buceo y salida a la superficie) han sido fotografiados, se actualiza el progreso de la actividad.

`CheckCompletion()`

Este método revisa si ambas tareas (tarea del cormorán buceando y tarea del cormorán saliendo a la superficie) han sido completadas:

- Si ambas tareas están completas, se llama a `CompleteActivity()` para finalizar la actividad.

Resumen de Componentes Principales:

- **Campos Públicos:**
 - **`cormoranTag`:** El tag que identifica al cormorán.
 - **`cameraTool`:** Herramienta usada para tomar fotos de los cormoranes.
 - **Claves de Localización:** Títulos y descripciones traducibles para las tareas relacionadas con los cormoranes.
- **Métodos Importantes:**
 - **`Start()`:** Inicializa las tareas y se suscribe al evento de la cámara.
 - **`OnDestroy()`:** Elimina la suscripción del evento de la cámara.
 - **`TryToAddAnimal()`:** Verifica y actualiza las tareas basadas en las fotos tomadas de los cormoranes.
 - **`CheckCompletion()`:** Verifica si ambas tareas necesarias se han completado.

Comportamiento:

- El jugador debe fotografiar dos situaciones específicas del cormorán: cuando está buceando y cuando está saliendo a la superficie.
- Cada vez que se toma una foto, el sistema verifica si el cormorán fotografiado está realizando la acción correspondiente. Si es así, se marca la tarea como completada.

- Si ambas tareas se completan, la actividad "Pescando con Cormoranes" se considera completada.

Este script gestiona las interacciones con los cormoranes y el sistema de tareas, garantizando que el jugador avance a través de la actividad de acuerdo con el progreso en la fotografía de los cormoranes en sus diferentes comportamientos.

CormoranBehaviour

Este script controla cómo se mueve, bucea y sube a la superficie un cormorán, incluye acciones como nadar en círculos, sumergirse hacia un pez y emerger nuevamente a la superficie. A continuación se explica el funcionamiento general del código y sus métodos:

Descripción General:

Este script maneja el comportamiento de un cormorán en un entorno de juego en 3D, simulando sus acciones de nadar, bucear y ascender. También incluye efectos de sonido mediante FMOD y controles sobre su movimiento circular, velocidad de inmersión y ascenso, y la interacción con peces en el agua. El cormorán nadará en círculos, se sumergirá hacia un pez y luego regresará a la superficie cuando haya terminado.

Métodos del Script:

- 1. Start()**
 - **Inicialización:** Establece la posición inicial del cormorán, el centro del movimiento circular y la velocidad de nado. También selecciona una dirección de giro aleatoria y asigna un evento de sonido de nado y buceo usando FMOD.
 - **Corutina de nado:** Inicia una corutina (SwimForRandomTime()) que hace que el cormorán nade durante un tiempo aleatorio antes de intentar sumergirse.
- 2. Update()**
 - **Estado del Cormorán:** En cada actualización del juego, este método verifica el estado actual del cormorán (Swimming, Diving, o Rising) y llama a los métodos correspondientes para controlar el movimiento y la acción según el estado actual.
 - **Rotación Suave:** Aplica una rotación suave hacia la rotación objetivo utilizando Quaternion.Slerp, permitiendo que el cormorán se gire de manera fluida.
- 3. SwimForRandomTime()**
 - **Nadar por un tiempo aleatorio:** Este método usa WaitForSeconds() para esperar un tiempo aleatorio (entre los valores definidos por minSwimTime y maxSwimTime) mientras el cormorán nada en círculos antes de intentar cazar un pez.
- 4. SelectRandomFish()**
 - **Seleccionar un pez aleatorio:** Busca en el entorno un pez que cumpla con ciertas condiciones (por ejemplo, que esté nadando y no esté siendo

perseguido por otro cormorán). Si se encuentra un pez válido, el cormorán cambia a estado Diving y se dirige hacia el pez.

5. **SwimInCircles()**

- **Movimiento circular:** Este método mueve al cormorán en un círculo alrededor de un punto central (centerPoint). El cormorán ajusta su posición y rotación en función de un ángulo que cambia constantemente, lo que crea el movimiento en círculos.

6. **DiveTowardsFish()**

- **Sumergirse hacia el pez:** Si el cormorán ha seleccionado un pez, se mueve hacia él a una velocidad determinada (diveSpeed). Si el cormorán se acerca lo suficiente al pez, cambia al estado Rising y realiza la acción de comer al pez (a través de la llamada a HandleBeingEaten() del pez).

7. **RiseToSurface()**

- **Ascender a la superficie:** Este método mueve al cormorán hacia la superficie (posición y = 0) y cambia su rotación para mirar hacia arriba. Cuando llega a la superficie, reinicia su posición y comienza un nuevo ciclo de nado.

8. **ResetToInitialPosition()**

- **Reiniciar la posición:** Restablece al cormorán a su posición original y reinicia todos los parámetros del movimiento (como el ángulo y la velocidad del nado). Luego comienza de nuevo el ciclo de nadar en círculos.

9. **SwimingSfxOn() y SwimingSfxOff()**

- **Reproducir sonido de nado:** Enciende y apaga el sonido de nado. Estas funciones verifican si el sonido está detenido o en reproducción, y ajustan su pitch (tono) aleatoriamente para darle un toque natural.

10. **DivingSfxOn() y DivingSfxOff()**

- **Reproducir sonido de buceo:** Similar a los métodos de nado, estas funciones gestionan el sonido del buceo, activando y desactivando el sonido según el estado del cormorán.

Resumen de Componentes Principales:

- **Variables de configuración:** Controlan varios aspectos del comportamiento del cormorán, como la velocidad de nado, el radio de los círculos, el tiempo de nado, las velocidades de buceo y ascenso, y las velocidades de rotación.
- **Estado del Cormorán:** Se maneja mediante el enum CormoranState, que tiene tres estados: Swimming (nadando), Diving (buceando) y Rising (ascendiendo).
- **Efectos de Sonido:** Utiliza el sistema de audio FMOD para reproducir efectos de sonido para los estados de nado y buceo.
- **Movimiento Circular:** El cormorán nada en círculos mediante un cálculo de posición basado en ángulos, creando una trayectoria circular suave.
- **Interacción con los Peces:** El cormorán puede seleccionar un pez y bucear hacia él, simulando el comportamiento de cazar. Cuando alcanza al pez, cambia de estado a Rising.

NototeniaBehaviour

Este script controla el comportamiento de un pez en un entorno de juego 3D, simulando su movimiento en el agua, la interacción con un cormorán, y la reproducción de efectos de sonido mediante FMOD. A continuación, se explican sus componentes principales y métodos.

Descripción General:

El script maneja cómo se mueve una nototenia, eligiendo un punto de inicio y un punto final aleatorios dentro de una lista de puntos de "spawn" en la escena. Cuando el pez es comido por un cormorán, su comportamiento se detiene y luego se reinicia en un intervalo aleatorio. Durante su movimiento, se reproduce un sonido de natación que varía ligeramente en pitch para hacerlo más natural. El script también maneja la reaparición del pez después de ser "eaten" (comido).

Métodos del Script:

1. Start()

Inicialización:

- Verifica que el modelo del pez (childModel) esté asignado, y si no, genera un error.
- Crea un evento de sonido usando FMOD para reproducir el sonido de natación (_swimming).
- Desactiva inicialmente el modelo del pez.
- Llama a CacheSpawnPoints() para obtener los puntos de "spawn" de la escena.
- Llama a una corutina (DelayedStart) que comienza el movimiento después de un retraso aleatorio.

2. Update()

Actualización:

- Verifica si el pez está detenido (estado FishState.Stopped).
- Si está detenido y no se está ejecutando una corutina de reaparición, inicia HandleRespawn() para que el pez se "regenera" después de un tiempo.

3. DelayedStart()

Inicio retardado:

- Espera un tiempo aleatorio entre 2.5 y 5 segundos antes de activar el modelo del pez y comenzar su movimiento.
- Llama a InitializePath() para configurar el movimiento del pez.

4. CacheSpawnPoints()

Almacena los puntos de "spawn":

- Busca un objeto en la escena llamado "SpawnPoints" y recoge todos los puntos de spawn dentro de él.
- Si no se encuentra, genera un error. Luego, guarda esos puntos en el array spawnSpots.

5. InitializePath()

Configuración del movimiento del pez:

- Elige aleatoriamente dos puntos de spawn (inicio y fin) y calcula posiciones aleatorias para esos puntos utilizando un pequeño rango de variación en los ejes Y y Z.
 - Asigna la velocidad de movimiento y cambia el estado del pez a FishState.Swimming (nadando).
 - Inicia el movimiento y el sonido de natación con SwimmingSfxOn().
6. **GetRandomIndex()**
Obtiene un índice aleatorio para seleccionar puntos de spawn:
- Selecciona un índice aleatorio, asegurándose de que no sea igual al índice de un punto previamente seleccionado (por ejemplo, no elegir el mismo punto de inicio y fin).
7. **RandomizePosition()**
Modifica aleatoriamente la posición de un punto:
- Variando los valores de Y y Z dentro de los rangos definidos por randomYOffset y randomZOffset.
8. **HandleRespawn()**
Maneja la reaparición del pez:
- Desactiva el modelo del pez, espera un intervalo aleatorio, y luego lo reactiva con una nueva ruta de movimiento.
9. **HandleBeingEaten()**
Maneja la interacción con el cormorán cuando el pez es comido:
- Detiene el sonido de natación, cambia el estado del pez a FishState.Stopped y comienza el proceso de reaparición a través de la corutina HandleRespawn().
10. **SwimmingSfxOn()**
Activa el sonido de natación:
- Configura los atributos 3D del sonido (con la posición del objeto en el mundo).
 - Si el sonido está detenido, lo inicia con un pitch aleatorio entre 0.8 y 1.2 para darle un toque más natural.
11. **SwimmingSfxOff()**
Desactiva el sonido de natación:
- Si el sonido está reproduciéndose, lo detiene con una transición suave (fade out).

Componentes Principales:

1. **Variables de configuración:**
 - childModel: El modelo visual del pez.
 - spawnSpots: Los puntos de aparición donde el pez puede aparecer.
 - startSpot y endSpot: Las posiciones de inicio y fin del movimiento del pez.
 - randomYOffset y randomZOffset: Rango de variación aleatoria para la posición del pez.
 - cormoran: Referencia al cormorán, que puede interactuar con el pez.
 - respawnCoroutine: La corutina que gestiona la reaparición del pez.
 - _swimming: Evento de sonido de natación.
2. **Estados del Pez:**
 - FishState.Swimming: Estado en el que el pez nada entre puntos.

- FishState.Stopped: Estado en el que el pez se detiene temporalmente (cuando es comido).
- 3. **Interacción con el Cormorán:**
 - Cuando el pez es comido, cambia su estado a detenido y se inicia la reaparición a través de HandleRespawn().
- 4. **Sonido:**
 - Utiliza FMOD para gestionar el sonido de natación. Se ajusta el tono (pitch) aleatoriamente para simular un comportamiento más dinámico.

3.5 Arqueología

La carpeta Arqueologia contiene los scripts de: ArqueologiaActivity e Item

ArqueologiaActivity

El script está diseñado para gestionar la actividad de "Arqueología Submarina" dentro de un juego, donde el jugador debe escanear tres tipos de objetos: cajas de madera, jarras y copas. El progreso de la actividad se lleva a cabo a través de un sistema de tareas que se actualiza cada vez que el jugador escanea un objeto de uno de estos tipos. A continuación, se describen los componentes y métodos clave del código:

Componentes del Script

Campos de Configuración

- **Contadores de objetos:**
 - woodBoxCounter, jugsCounter, cupCounter: Son las variables que almacenan la cantidad de objetos escaneados de cada tipo. Se incrementan cada vez que el jugador escanea un objeto de uno de estos tipos.
- **Número máximo de objetos:**
 - maxWoodBox, maxJugs, maxCups: Establecen el número máximo de cada tipo de objeto que el jugador debe escanear para completar la tarea de la actividad.
- **Listas de objetos escaneados:**
 - _scannedWoodBoxes, _scannedJugs, _scannedCups: Estas listas mantienen un registro de los objetos escaneados por el jugador, identificados por su ID único. Esto evita que el jugador escanee el mismo objeto más de una vez.
- **Claves de localización:**
 - woodBoxTitleKey, woodBoxDescriptionKey, jugsTitleKey, jugsDescriptionKey, cupsTitleKey, cupsDescriptionKey: Son las claves de localización que se utilizan para traducir los títulos y descripciones de las tareas relacionadas con los objetos escaneados a diferentes idiomas.
- **Variables de estado:**
 - _wTitleKey, _wDescriptionKey, _jTitleKey, _jDescriptionKey, _cTitleKey, _cDescriptionKey: Estas variables almacenan las traducciones obtenidas de las claves de localización, las cuales se utilizan en el juego para mostrar los textos en el idioma correspondiente.

Métodos Clave

- **Start()** Este método se ejecuta al inicio de la actividad y realiza las siguientes tareas:
 - Llama al método base.Start() para ejecutar cualquier lógica heredada de la clase Mission.
 - Traduce las claves de localización de los objetos (cajas de madera, jarras y copas) a su texto correspondiente en el idioma del jugador.
 - Utiliza el sistema de seguimiento de tareas (_watch) para crear tres tareas relacionadas con los objetos escaneados. Cada tarea muestra el progreso de

recolección en el formato 0/max, donde max es el número máximo de objetos que deben ser escaneados.

- **TryAddItem(ItemType itemType, int id)** Este método se ejecuta cada vez que el jugador escanea un objeto. Recibe dos parámetros: itemType (el tipo de objeto) y id (el ID único del objeto escaneado). Dependiendo del tipo de objeto, el método realiza lo siguiente:
 - Si el objeto no ha sido escaneado previamente (es decir, su ID no está en la lista de objetos escaneados), lo añade a la lista correspondiente (_scannedWoodBoxes, _scannedJugs, o _scannedCups).
 - Aumenta el contador correspondiente (ya sea woodBoxCounter, jugsCounter, o cupCounter).
 - Actualiza el progreso de la tarea usando _watch.UpdateTask(), donde se muestra el número de objetos escaneados hasta el momento, y marca la tarea como completada si el número máximo de objetos ha sido alcanzado.
 - Finalmente, llama a CheckCompletion() para verificar si todas las tareas se han completado.
- **CheckCompletion()** Este método verifica si todas las tareas relacionadas con los objetos (cajas de madera, jarras y copas) han sido completadas:
 - Si el número de objetos escaneados de cada tipo alcanza el número máximo requerido (maxWoodBox, maxJugs, maxCups), considera que todas las tareas están completas.
 - Si todas las tareas están completas, muestra un mensaje en consola ("¡Todas las tareas completadas!") y llama a CompleteActivity() para finalizar la actividad.

Resumen de Componentes Principales

Campos Públicos

- **Contadores de objetos:** woodBoxCounter, jugsCounter, cupCounter.
- **Claves de localización:** woodBoxTitleKey, woodBoxDescriptionKey, jugsTitleKey, jugsDescriptionKey, cupsTitleKey, cupsDescriptionKey.

Métodos Importantes

- **Start():** Inicializa las tareas y traduce los textos de las tareas.
- **TryAddItem():** Actualiza el progreso cuando el jugador escanea un objeto.
- **CheckCompletion():** Verifica si todas las tareas se han completado.

Comportamiento

El jugador debe escanear tres tipos de objetos: cajas de madera, jarras y copas. Cada vez que un objeto es escaneado, el progreso se actualiza automáticamente, y la tarea correspondiente se marca como completada cuando se alcanzan los objetivos de cantidad. Si todas las tareas se completan, la actividad se da por finalizada.

Este script gestiona la actividad "Arqueología Submarina", asegurándose de que el jugador avance en la actividad, manteniendo un registro del progreso de las tareas y proporcionando retroalimentación visual en función de los objetos escaneados.

Item

Este script representa los objetos que el jugador puede escanear en la actividad "Arqueología Submarina". Cada objeto tiene tres propiedades clave:

- **analyzed** (bool): Indica si el objeto ha sido analizado.
- **type** (ItemType): Define el tipo de objeto (por ejemplo, caja de madera, jarra, copa).
- **id** (int): Un identificador único para cada objeto.

Cuando el jugador escanea un objeto, se marca como "analizado", y se usa el type y id para gestionarlo en las tareas de recolección del juego.

4. Lobby

La carpeta Lobby contiene los scripts de: FadeEffect, HoverCanvasController, LanguageSelector, LevelLoader, LevelSelector, RayColorChange, RayVisualizer

FadeEffect

Este script gestiona un efecto de fundido en la pantalla, permitiendo hacer transiciones de visibilidad entre transparente y negro. Utiliza un material (normalmente un cuadrado que cubre toda la pantalla) para mostrar el efecto. Aquí están sus componentes y métodos clave:

Componentes:

- **fadeMaterial**: Material utilizado para el efecto de fundido. Debe estar asignado a un objeto (como un quad) que cubre la pantalla.
- **fadeDurationTime**: Duración del fundido, que se puede ajustar. Por defecto es de 3 segundos.
- **isFading**: Booleano que asegura que no se inicien múltiples fundidos al mismo tiempo.

Métodos:

- **Start()**: Al inicio, se inicia un fundido de transparente a negro llamando a StartCoroutine(StartFadeToClear()).
- **StartFadeToClear()**: Es una corrutina que, primero, asegura que el material esté completamente transparente (opacidad 1). Luego, espera 1 segundo antes de ejecutar el fundido hacia transparente usando FadeToClear().
- **FadeToBlack()**: Inicia el fundido de transparente a negro si no está ocurriendo otro fundido, llamando a la corrutina Fade() con los valores adecuados.
- **FadeToClear()**: Inicia el fundido de negro a transparente si no está ocurriendo otro fundido, llamando a la corrutina Fade() con los valores adecuados.
- **Fade()**: Realiza el fundido de un valor de opacidad (startAlpha) a otro (endAlpha) a lo largo del tiempo definido por fadeDurationTime. Durante el fundido, el valor de opacidad se interpola y se actualiza el material para reflejar el cambio visual.
- **SetMaterialAlpha()**: Actualiza la opacidad del material ajustando su componente alpha en el color, aplicando el valor calculado durante el fundido.

Comportamiento:

Este script permite realizar efectos visuales de fundido de manera sencilla, alternando entre negro y transparente, controlado por las funciones `FadeToBlack()` y `FadeToClear()`.

HoverCanvasController

Este script gestiona la visualización de un **canvas** que muestra información de las actividades en el lobby. Utiliza claves de localización para obtener los textos traducidos y mostrarlos en los elementos UI correspondientes.

Componentes:

- **canvas**: Objeto que representa el canvas en la interfaz de usuario que contiene los textos que se van a mostrar u ocultar.
- **descriptionUI, titleUI**: Referencias a los componentes `TextMeshProUGUI` que muestran la descripción y el título, respectivamente.
- **titleKey, descriptionKey**: Claves de localización que se usan para obtener las traducciones de los textos de título y descripción.

Métodos:

- **ShowCanvas()**:
 - Muestra el canvas y actualiza los textos de `titleUI` y `descriptionUI` usando las claves de localización (`titleKey` y `descriptionKey`).
 - Se obtiene la traducción de los textos mediante `LocalizationManager.GetTranslation()`.
 - Activa el canvas, lo que lo hace visible en la pantalla.
- **HideCanvas()**: Oculta el canvas al desactivarlo (`canvas.SetActive(false)`).

Comportamiento:

Este script permite mostrar y ocultar un canvas con información traducida, como el título y la descripción, que se cargan dinámicamente utilizando un sistema de localización (`I2 Localization`). Se activa o desactiva el canvas en función de la necesidad de mostrar la información al jugador.

LanguageSelector:

Este script gestiona la selección de un idioma en el juego, permitiendo al jugador cambiar entre diferentes idiomas disponibles. Además, resalta el idioma seleccionado cambiando el color de las imágenes asociadas a los botones de idioma.

Componentes:

- **languageCode**: Código del idioma (ej. "en" para inglés, "es" para español, etc.), que indica el idioma que se seleccionará.
- **imageParent**: Imagen asociada al idioma actual que se está mostrando en la interfaz. Se cambia su color para resaltar el idioma seleccionado.

- **otherLanguages:** Un arreglo de imágenes de otros idiomas disponibles, que se desactivan visualmente cuando el idioma está seleccionado.

Métodos:

- **Start():**
 - Comprueba si el idioma actual (almacenado en `LocalizationManager.CurrentLanguage`) es el mismo que el idioma asociado con este botón (`languageCode`).
 - Si es el mismo, el color de `imageParent` se cambia a amarillo, indicando que ese idioma está seleccionado. Si no, se establece un color transparente para desmarcarlo.
- **ChangeLanguage():**
 - Cambia el idioma global a `languageCode` (especificado para este botón).
 - Resalta el botón correspondiente (`imageParent.color = Color.yellow`).
 - Restaura el color de los otros idiomas a transparente para reflejar que no están seleccionados.
 - Actualiza el idioma de la aplicación usando `LocalizationManager.CurrentLanguage`.
 - Registra en consola el idioma seleccionado para confirmación.

Comportamiento:

El script permite cambiar el idioma global de la aplicación, modificando el valor de `LocalizationManager.CurrentLanguage`. Cuando el jugador selecciona un idioma, el script resalta la imagen del botón correspondiente y actualiza la interfaz para reflejar el nuevo idioma. Las imágenes de los otros botones se desmarcan visualmente. Los textos que utilicen `LocalizeComponent` se actualizan automáticamente al cambiar el idioma.

LevelLoader

Este script se encarga de gestionar la carga de niveles (escenas) en el juego, implementando una transición suave entre niveles usando efectos de fundido y música que se desvanece y reaparece.

Componentes:

- **fadeEffect:** Referencia al componente `FadeEffect` que se usa para aplicar los efectos de fundido entre escenas.

Métodos:

- **Start():**
 - Reproduce la música de fondo con un desvanecimiento (usando `AudioManager.Instance.FadeInMusic(0.5f)`).
 - Si el componente `fadeEffect` no está asignado manualmente en el editor, busca un objeto en la escena que tenga el componente `FadeEffect` (usando `FindObjectOfType<FadeEffect>()`).
- **LoadNextLevel(string levelName):**

- Llama a la corrutina Transition para realizar la transición a la siguiente escena, pasando el nombre de la escena como argumento.
- **Transition(string sceneName):**
 - Reduce el volumen de la música actual con un desvanecimiento (usando AudioManager.Instance.FadeOutMusic(0.5f)).
 - Aplica el efecto de fundido a negro con fadeEffect.FadeToBlack().
 - Espera el tiempo necesario para completar el fundido (utilizando fadeEffect.fadeDurationTime).
 - Luego, espera un segundo adicional antes de cargar la nueva escena con SceneManager.LoadScene(sceneName).

Comportamiento:

El script gestiona la transición entre niveles de manera fluida. Primero, desvanece la música de fondo, luego aplica un fundido a negro en la pantalla y finalmente carga la nueva escena. Los tiempos de espera permiten que la música y el efecto de fundido se completen antes de cambiar la escena.

LevelSelector

Este script gestiona la carga de una escena específica (nivel) cuando el jugador interactúa con un botón o algún otro evento.

Componentes:

- **sceneName:** El nombre de la escena que se desea cargar.
- **levelLoader:** Referencia al componente LevelLoader, que se encarga de la transición entre niveles (escenas).

Métodos:

- **LoadLevel():**
 - Llama al método LoadNextLevel() del componente levelLoader, pasando el nombre de la escena (sceneName) para cargar la siguiente escena.
 - Imprime en la consola un mensaje de depuración indicando que la escena ha cambiado.

Comportamiento:

Este script está diseñado para ser utilizado con un sistema de selección de niveles, por ejemplo, en un menú de niveles. Al activarse (por ejemplo, al hacer clic en un botón), llama al levelLoader para realizar la transición a la escena especificada en sceneName. La carga de la escena se realiza a través del LevelLoader, y se muestra un mensaje en la consola para indicar que la escena ha sido cambiada.

RayColorChanger

Este script está diseñado para cambiar el color de un **LineRenderer** cuando el rayo (o cursor) interactúa con un objeto, permitiendo efectos visuales de cambio de color al pasar el ratón o seleccionar el objeto.

Componentes:

- **lineRenderer**: Referencia al componente LineRenderer que se usará para dibujar el rayo o línea.
- **defaultColor**: Color predeterminado de la línea (por defecto blanco).
- **hoverColor**: Color que la línea toma cuando el rayo interactúa con un objeto (por defecto rojo).
- **selectColor**: Color que la línea toma cuando el objeto es seleccionado.

Métodos:

- **Start()**:
 - Se ejecuta al iniciar el script, asegurando que el LineRenderer tenga el color predeterminado (defaultColor) tanto al inicio de la línea (startColor) como al final (endColor).
- **OnHoverEntered(HoverEnterEventArgs args)**:
 - Se llama cuando el rayo (o cursor) entra en contacto con un objeto, cambiando el color del LineRenderer al color de interacción (hoverColor).
- **OnHoverExited(HoverExitEventArgs args)**:
 - Se llama cuando el rayo deja de interactuar con un objeto, restaurando el color del LineRenderer al color predeterminado (defaultColor).
- **OnSelected()**:
 - Se llama cuando el objeto es seleccionado, cambiando el color del LineRenderer al color de selección (selectColor).

Comportamiento:

Este script permite que un **LineRenderer** cambie de color en tres situaciones:

1. **Color por defecto** al inicio.
2. **Color de interacción** (hoverColor) cuando el rayo pasa sobre un objeto.
3. **Color de selección** (selectColor) cuando el objeto es seleccionado.

RayVisualizer

Este script se encarga de visualizar un **rayo** (raycast) utilizando VR, utilizando un **LineRenderer** para dibujar el rayo que se proyecta a través del espacio.

Componentes:

- **rayInteractor**: Hace referencia a un componente **XRRayInteractor**, que es responsable de gestionar el rayo de interacción, incluyendo su origen, dirección y las interacciones con objetos en el entorno 3D.

- **lineRenderer:** Es un componente **LineRenderer** utilizado para dibujar una línea (el rayo) desde el origen del rayo hasta el punto de impacto o hasta el final del alcance del rayo.

Métodos:

- **Update():**
 - Este método se ejecuta cada cuadro y es el encargado de actualizar la visualización del rayo.
 - Verifica si rayInteractor y lineRenderer no son null. Si alguno de ellos es null, no realiza ninguna acción.
 - Luego, obtiene el punto donde el rayo impacta, usando **rayInteractor.TryGetCurrent3DRaycastHit(out RaycastHit hit):**
 - Si el rayo impacta en un objeto (es decir, si hay un **hit**), el LineRenderer se configura para mostrar el rayo desde su origen hasta el punto de impacto (hit.point).
 - Si no hay impacto (el rayo no colisiona con ningún objeto), el LineRenderer dibuja una línea desde el origen del rayo en la dirección del rayo, hasta la distancia máxima de lanzamiento (maxRaycastDistance).

Comportamiento:

Este script permite visualizar el rayo en la escena en tiempo real:

1. **Cuando el rayo golpea un objeto**, el LineRenderer ajusta su punto final al punto de impacto.
2. **Cuando el rayo no golpea ningún objeto**, el LineRenderer dibuja una línea hasta la distancia máxima del rayo.

5. Audio

Esta carpeta contiene los scripts de: AudioManager y FmodEvents y se divide la subcarpeta: Ambience.

AudioManager

Este script se encarga de gestionar los eventos de sonido en el juego utilizando **FMOD**. Permite manejar la música de fondo, efectos de sonido y la ambientación, con capacidades de **fade-in** y **fade-out** para la música, además de gestionar las instancias de eventos de sonido y emisores de eventos.

Componentes Principales:

- **_eventInstances**: Lista que almacena las instancias de eventos de FMOD.
- **_eventEmitters**: Lista de emisores de eventos, representados por componentes StudioEventEmitter.
- **_musicEventInstance** y **_ambienceEventInstance**: Instancias de los eventos de música y ambientación.
- **Instance**: Instancia estática para asegurar que solo haya un AudioManager en la escena.

Métodos y Funciones:

1. **Awake()**:
 - Se asegura de que solo haya una instancia de AudioManager en la escena. Si encuentra más de una, muestra un error.
2. **Start()**:
 - Inicializa los eventos de música y ambientación utilizando las referencias de eventos definidas en el objeto FmodEvents.
3. **CreateEventInstance(EventReference eventReference)**:
 - Crea una nueva instancia de un evento de FMOD usando la referencia de evento proporcionada y lo agrega a la lista _eventInstances.
4. **InitializeMusic(EventReference musicEventReference)**:
 - Inicializa el evento de música, creando la instancia y comenzando su reproducción.
5. **InitializeAmbience(EventReference ambienceEventReference)**:
 - Inicializa el evento de ambientación, creando la instancia y estableciendo el parámetro correspondiente de la zona de ambientación (utilizando la enumeración AmbienceArea).
6. **PlayOneShot(EventReference sound, Vector3 worldPos)**:
 - Reproduce un efecto de sonido en una posición específica en el mundo, utilizando la función RuntimeManager.PlayOneShot.
7. **PlayOneShotRandomPitch(EventReference sound, Vector3 worldPos)**:
 - Reproduce un sonido en una posición específica, pero con un **pitch** aleatorio.
8. **SetAmbienceParameter(AmbienceArea area)**:
 - Establece el parámetro de ambiente (area) en el evento de ambientación, ajustando la atmósfera según la zona.

9. **InitializeEventEmitter(EventReference eventReference, GameObject emitterGameObject):**
 - Inicializa un emisor de eventos (StudioEventEmitter) en un objeto dado, y lo agrega a la lista `_eventEmitters`.
10. **CleanUp():**
 - Limpia todas las instancias de eventos y emisores de eventos, deteniendo y liberando sus recursos.
11. **OnDestroy():**
 - Llama a `CleanUp()` cuando el AudioManager se destruye.

Nuevas Funciones para Control de Música:

- **FadeOutMusic(float duration):**
 - Realiza un fade-out en la música, reduciendo gradualmente el volumen hasta llegar a 0 durante un tiempo específico (duration). Detiene el evento de música con la opción `ALLOWFADEOUT`, lo que permite que el volumen baje gradualmente.
- **FadeInMusic(float duration):**
 - Realiza un fade-in en la música, comenzando desde un volumen de 0 hasta llegar al valor máximo de 1 durante un tiempo específico (duration).
- **FadeMusicVolume(float startVolume, float endVolume, float duration):**
 - Controla el cambio gradual de volumen de la música a lo largo del tiempo. Utiliza la función `Mathf.Lerp` para hacer una transición suave entre el volumen inicial y final.

Comportamiento:

Este script maneja todos los aspectos relacionados con los sonidos del juego, incluyendo:

- Reproducción de música y efectos de sonido 3D.
- Control de la música de fondo (inicio, fade-in, fade-out).
- Gestión de sonidos ambientales (como el sonido de un área específica).
- Soporte para efectos de sonido con diferentes parámetros como el pitch.

FmodEvents

Este script se encarga de almacenar y proporcionar referencias a eventos de **FMOD** que corresponden a diferentes tipos de sonidos en el juego. La clase `FmodEvents` centraliza todos estos eventos para facilitar su acceso y uso a través del sistema de audio.

Componentes Principales:

- **Referencias a eventos de FMOD:** A través de las variables públicas `EventReference`, se gestionan los eventos de sonidos de ambiente, música, efectos de sonido de jugador, herramientas, reloj, peces, lobos y cormoranes.
- **Instance:** Instancia estática que asegura que solo haya una referencia a `FmodEvents` en la escena, similar a un patrón Singleton.

Propiedades:

1. **Ambience**: Referencia al evento de música ambiental.
2. **music**: Referencia al evento de música principal.
3. **swiming**: Efectos de sonido para nadar (probablemente para el jugador).
4. **TakingPhoto**, **ZoomIn**, **Radar**, **WinRadar**, **LoseRadar**, **Grab**: Efectos de sonido para interacciones con herramientas (por ejemplo, tomar fotos, radar).
5. **Change_Screen**, **Change_Task**: Efectos para el reloj (por ejemplo, cambios de pantalla o de tareas).
6. **fish_swiming**, **fish_escape**: Sonidos relacionados con los peces (nadar y escapar).
7. **Lobo_swiming**, **Lobo_flip**: Sonidos relacionados con un lobo (nadar y voltear).
8. **cormoran_diving**: Sonido para el cormorán sumergiéndose.

Comportamiento:

- **Patrón Singleton**: Utiliza la propiedad estática Instance para garantizar que solo exista una instancia de esta clase en la escena. Si se encuentra más de una, muestra un error en el log (Debug.LogError).
- **Awake()**: Se encarga de inicializar la instancia estática de FmodEvents y verifica que solo haya una de ellas en la escena. Si hay más de una, lanza un error.

Propósito:

Este script proporciona una manera estructurada y fácil de acceder a todos los eventos de FMOD definidos en el proyecto, sin necesidad de buscar cada referencia de sonido en la escena o en otros scripts. Cada tipo de sonido está bien organizado bajo una categoría que puede incluir música, efectos de sonido de herramientas, interacciones con el jugador, efectos de criaturas, etc.

5.1 Ambience

Esta carpeta contiene el script: AmbienceChangeTrigger

AmbienceChangeTrigger

Este script está diseñado para cambiar el área de ambiente sonoro en función de la zona donde se encuentra el jugador. Usa un **trigger** para detectar cuándo el jugador entra o sale de una zona específica, lo que a su vez ajusta los parámetros de sonido ambiental a través del AudioManager.

Componentes principales:

- **_area**: Variable que almacena el tipo de área de ambiente (AmbienceArea), que puede ser de tipo DeepArea o HighArea en este caso.
- **OnTriggerEnter**: Evento que se activa cuando el jugador entra en el área de un collider que tiene el tag "Player". Cambia el área de ambiente a DeepArea y actualiza el parámetro de ambiente a través de AudioManager.
- **OnTriggerExit**: Evento que se activa cuando el jugador sale del collider. Cambia el área de ambiente a HighArea y actualiza el parámetro de ambiente a través de AudioManager.

Explicación de las funciones:

1. **OnTriggerEnter:**

- Se llama cuando el jugador entra en el área del trigger (un collider con el tag "Player").
- Si el jugador entra en el área, el script cambia el tipo de ambiente a DeepArea.
- Llama a `AudioManager.Instance.SetAmbienceParameter` para actualizar el parámetro de ambiente en FMOD a la zona correspondiente (DeepArea).

2. **OnTriggerExit:**

- Se llama cuando el jugador sale del área del trigger.
- Cambia el área de ambiente a HighArea y actualiza el parámetro de ambiente de FMOD.

Propósito:

Este script se utiliza para gestionar dinámicamente el sonido ambiental en función de la ubicación del jugador en el juego. Si el jugador entra en una zona profunda (por ejemplo, un área submarina o una cueva), el sonido ambiental cambia a DeepArea. Cuando sale de esa área, el sonido ambiental vuelve a un área más superficial o normal como HighArea.

6. Physics

Esta carpeta contiene el script de: `ManageBoxColliderOnGrab`.

ManageBoxColliderOnGrab:

Este script está diseñado para manejar la activación y desactivación de los **colliders** de un objeto cuando el jugador agarra y suelta el objeto utilizando **XR Interaction Toolkit** de Unity.

Componentes principales:

- **XRGrabInteractable interactor:** Referencia al componente `XRGrabInteractable`, el cual permite que el objeto sea interactuado (agarrado y soltado) mediante las interacciones del usuario en VR.
- **`Collider[] colliders`:** Array de Colliders asociados al objeto. Estos serán desactivados y reactivados cuando el objeto sea agarrado o soltado.

Funcionamiento del script:

1. **Suscripción a eventos:**

- **`selectEntered.AddListener(OnSelectEntered)`:** Este evento se dispara cuando el objeto es agarrado por el jugador.
- **`selectExited.AddListener(OnSelectExited)`:** Este evento se dispara cuando el objeto es soltado por el jugador.

2. **Desuscripción de eventos:**

- Cuando el objeto se destruye o el script se desactiva, los eventos se eliminan para evitar que se llamen cuando ya no sea necesario.

3. **OnSelectEntered:**

- Este método se llama cuando el jugador agarra el objeto.
- Desactiva todos los colliders en el array colliders usando un bucle foreach. Esto puede ser útil para evitar que el objeto colisione con otros objetos cuando está siendo manipulado.

4. **OnSelectExited:**

- Este método se llama cuando el jugador suelta el objeto.
- Inicia una **corutina** (ReenableColliderWithDelay) para reactivar los colliders después de un pequeño retraso, lo cual es útil si se desea evitar que el objeto reaccione instantáneamente a las colisiones cuando se suelta.

5. **ReenableColliderWithDelay:**

- Esta corutina espera un tiempo determinado (en este caso, 0.5f segundos, ajustable) antes de reactivar todos los colliders del objeto.
- El retraso puede ser útil para evitar problemas de colisiones no deseadas cuando el jugador libera el objeto.

Propósito:

El propósito de este script es prevenir que el objeto interactúe o colisione con otros objetos cuando está siendo agarrado por el jugador. Cuando el jugador suelta el objeto, los colliders se reactivan después de un pequeño retraso para garantizar que las colisiones y físicas funcionen correctamente.

7. Path

Esta carpeta contiene los scripts de: Node y PathfindingManager

PathfindingManager

El script es responsable de generar y gestionar los nodos para la navegación de algunos animales. Este sistema de nodos se utiliza para crear recorridos que permiten a los animales moverse a través de un área tridimensional, evitando obstáculos.

Atributos:

- **obstacleMask (LayerMask):** Máscara de capa utilizada para identificar obstáculos en el área de navegación.
- **nodePrefab (GameObject):** Prefab utilizado para representar visualmente los nodos en el área de navegación.
- **areaSize (Vector3):** Dimensiones del área en la que se distribuirán los nodos. Se define en tres ejes: X, Y, Z.
- **distanceBetweenNodes (float):** Distancia entre los nodos generados. Controla la densidad de los nodos.
- **showAllGizmos (bool):** Controla si se deben mostrar los Gizmos de los nodos en el editor.
- **allNodes (List<Node>):** Lista que almacena todos los nodos generados en el área de navegación.

Métodos:

1. **CreateNodes():**
 - **Descripción:** Genera los nodos en un área tridimensional de acuerdo con las dimensiones de `areaSize` y separándolos a la distancia especificada en `distanceBetweenNodes`. Antes de crear los nodos, limpia la lista de nodos existentes.
 - **Funcionamiento:** Itera a través de cada dimensión del área y calcula la posición de cada nodo. Los nodos se almacenan en la lista `allNodes`.
2. **ConnectNodes():**
 - **Descripción:** Conecta los nodos que no estén cerca de obstáculos, creando relaciones entre ellos. También verifica si hay una línea clara entre los nodos para asegurar que los animales puedan moverse entre ellos sin obstáculos.
 - **Funcionamiento:** Recorre cada nodo y verifica su proximidad a obstáculos usando `Physics.CheckSphere`. Luego, conecta los nodos vecinos utilizando un algoritmo basado en las posiciones relativas de los nodos y comprobando si hay obstáculos entre ellos con `Physics.Linecast`.
3. **ClearNodes():**
 - **Descripción:** Limpia la lista `allNodes`, eliminando todos los nodos previamente generados.
4. **ToggleAllGizmos(bool state):**
 - **Descripción:** Enciende o apaga la visualización de los Gizmos que representan los nodos en el editor.
5. **GetAllNodes():**
 - **Descripción:** Devuelve la lista de todos los nodos generados.
6. **GetClosestNode(Vector3 position):**
 - **Descripción:** Encuentra el nodo más cercano a una posición específica.
 - **Funcionamiento:** Itera a través de todos los nodos y calcula la distancia a cada uno, retornando el nodo más cercano.
7. **FindPath(Node startNode, Node targetNode, Vector3 target):**
 - **Descripción:** Encuentra un camino entre dos nodos usando el algoritmo A*. El camino es generado evaluando los costos de movimiento entre nodos adyacentes y eligiendo el mejor camino basado en la heurística (distancia recta entre nodos).
 - **Funcionamiento:** Utiliza un conjunto de nodos abiertos (por explorar) y un conjunto de nodos cerrados (ya evaluados). Se calcula el coste de cada nodo usando un algoritmo basado en el A* y se reconstruye el camino una vez que se encuentra el objetivo.
8. **GetLowestFScoreNode(List<Node> openSet, Dictionary<Node, float> fScore):**
 - **Descripción:** Devuelve el nodo con el menor valor de F (la combinación del coste desde el inicio y la heurística).
9. **Heuristic(Node a, Node b):**
 - **Descripción:** Calcula la distancia en línea recta entre dos nodos (heurística utilizada por el algoritmo A*).
10. **ReconstructPath(Dictionary<Node, Node> cameFrom, Node currentNode, Vector3 target):**
 - **Descripción:** Reconstruye el camino desde el nodo de destino hasta el nodo inicial siguiendo los nodos padres almacenados en el diccionario `cameFrom`.

- **Funcionamiento:** Retrocede desde el nodo de destino hasta el nodo de inicio, reconstruyendo el camino completo y añadiendo un nodo en la posición objetivo.

11. OnDrawGizmos():

- **Descripción:** Dibuja Gizmos visuales de los nodos en el editor, usando esferas para representarlos. Si un nodo está cerca de un obstáculo, se muestra en color rojo, y si no, en azul.

Node

Este script gestiona el comportamiento de los nodos utilizados en PathfindingManager, donde cada nodo representa una posición en el espacio y mantiene información sobre sus vecinos.

Descripción de los Métodos

1. Constructor (Node(Vector3 pos))

Este es el constructor de la clase Node, que se utiliza para inicializar un nuevo nodo en el espacio 3D.

- **Parámetros:**
 - pos: Un objeto Vector3 que define la posición del nodo en el espacio.
- **Acciones:**
 - Inicializa el campo position con la posición proporcionada.
 - Establece neighborsId como una lista vacía para almacenar los identificadores de los nodos vecinos.

Resumen de Componentes Principales

- **position (Vector3)**
Representa la ubicación del nodo en el espacio 3D. Se utiliza para determinar la distancia y las conexiones entre nodos.
- **neighborsId (List<int>)**
Una lista de identificadores de los nodos vecinos. Estos son los nodos a los que este nodo está conectado, y se usan en algoritmos de navegación y búsqueda de caminos.
- **isNearObstacle (bool)**
Un valor booleano que indica si el nodo está cerca de un obstáculo. Se podría utilizar en algoritmos de planificación de rutas para evitar obstáculos.

8. Editor

Esta carpeta contiene el script: PathfindingManagerEditor

PathfindingManagerEditor

Este script personaliza la interfaz de usuario del editor de Unity para la clase PathfindingManager, proporcionando botones que permiten al usuario interactuar con la creación, conexión y limpieza de nodos.

Descripción de los Métodos

1. OnInspectorGUI()

Este método sobrescribe la función predeterminada OnInspectorGUI para personalizar la apariencia y funcionalidad de la interfaz del inspector en Unity.

- **Acciones:**

- Llama a DrawDefaultInspector() para mostrar todos los campos predeterminados de la clase en el inspector.
- Obtiene una referencia al PathfindingManager usando target.
- Dibuja tres botones en el inspector que permiten al usuario interactuar con el sistema de nodos:
 - **Crear Nodos:** Llama al método CreateNodes() para generar los nodos en el entorno.
 - **Conectar Nodos:** Llama al método ConnectNodes() para conectar los nodos entre sí.
 - **Limpiar Nodos:** Llama al método ClearNodes() para eliminar los nodos existentes.

Resumen de Componentes Principales

- **GUILayout.Button()**
Esta función de la interfaz de Unity dibuja un botón en el inspector y permite realizar acciones cuando el usuario hace clic sobre él.
- **DrawDefaultInspector()**
Función de Unity que dibuja todos los campos públicos de la clase en el inspector sin necesidad de hacerlo manualmente.
- **Referencias a PathfindingManager**
 - CreateNodes(): Crea los nodos en el entorno.
 - ConnectNodes(): Establece conexiones entre los nodos.
 - ClearNodes(): Elimina todos los nodos.