Universidad Nacional Autónoma de México

Facultad de Ingeniería

Proyecto Final #1: Videojuego FPS en Unity 2: Estructuras de Datos y Algoritmo

Camacho Garduño Miguel Angel

Semestre: 2021-2

Profesor: Marco Antonio Martínez Quintana

Estructuras de Datos y Algoritmos I

Fecha: 13 de agosto de 2021

Resumen

En este documento se va a hablar en la introducción sobre el uso de los videojuegos en el ámbito educativo, una breve historia sobre los videojuegos y el género de los First Person Shooter, además de hablar sobre qué es un nivel y la aplicación de éste y otros conceptos en Estructuras de Datos. En el desarrollo se abordará sobre en qué consiste el proyecto, su algoritmo, diagrama de flujo, pseudocódigo y el código fuente de algunos de los archivos utilizados. En los resultados veremos el juego funcionando, el canal de YouTube y el repositorio de GitHub accedido por terceros, y la comparativa entre febrero y agosto de 2021 de los recursos informáticos, costes asociados al proyecto y el diagrama de Gantt. Y al final están las conclusiones, referencias, el glosario y acrónimos.

Introducción

Un videojuego es un tipo de entretenimiento que, al usar cierto tipo de controles o mandos, permite simular experiencias en la pantalla de cualquier dispositivo electrónico compatible. A diferencia de otros tipos de entretenimiento, los videojuegos requieren que el usuario se involucre activamente con el contenido.

La historia de los videojuegos se remonta a la década de 1970, con los primeros videojuegos como Space Invaders o Asteroids, en la década de los 80 surgieron empresas japonesas como Namco, SEGA y Nintendo, esta última dueña y creadora de la franquicia de Mario Bros. En los noventa, surgieron franquicias como DOOM o Sonic.

Un videojuego de disparos en primera persona, como su nombre lo dice, es un tipo de videojuego en el que la interacción y la visión del entorno en el juego es en primera persona, como si el juego lo viviera en carne y hueso con sus propios ojos.

La historia de los videojuegos en primera persona se remonta a la década de 1970 cuando se creó el primer videojuego del género llamado Maze War, sin embargo, no tenían una historia desarrollada, sonido o colores. No fue hasta 1992 cuando id Software publicó Wolfenstein 3D, un juego en el un soldado estadounidense escapa de un castillo nazi hasta llegar al bunker de Hitler para matarlo, después en 1993 sacó DOOM, que fue uno de los primeros juegos con tener un componente multijugador y luego en 1996 Quake, que agregó más modos multijugador. En la actualidad, Activision lidera el género de los FPS con una entrega anual de la franquicia Call of Duty, seguido de EA con la franquicia Battlefield, y de Bethesda Softworks con franquicias propias como de id Software como DOOM, Wolfenstein, Rage, Fallout, etc.

El uso de los videojuegos en el ámbito educativo ayuda a los niños a desarrollar las siguientes habilidades: desarrollar la paciencia mediante la repetición, resolución de problemas, la coordinación ojo-mano, etc. Existen videojuegos con ediciones educativos como Minecraft que tienen plantillas para cubrir diversas materias como química, historia, geografía, etc. Otros como el Wii Fit te ayudan a tener una rutina para tener una vida saludable.

Un nivel, mapa, área, etapa, mundo, rack, tablero, pantalla o zona en un videojuego es el espacio total disponible para el jugador a la hora de completar un objetivo específico. Los niveles de los videojuegos suelen tener una dificultad que aumenta progresivamente para atraer la atención de jugadores de todos los tipos. Cada nivel muestra nuevo contenido y desafíos para mantener alto el interés de los jugadores. En los videojuegos existe un elemento llamado framerate, o fotogramas por segundo. Durante cada cuadro, hay una pila que se construye antes de que se pueda renderizar el juego. En el desarrollo de juegos lo llamamos bucle de dibujo. En cada cuadro, se necesita recopilar información antes de poder decirle a la computadora qué renderizar. Si un enemigo caminaba hacia la izquierda, se actualiza la animación y se mueve su posición. Si el jugador acaba de saltar, se quiere agregar velocidad vertical y comenzar a verificar si ha aterrizado en una plataforma en cada cuadro, agregando más acciones a la pila.

Desarrollo

Descripción general

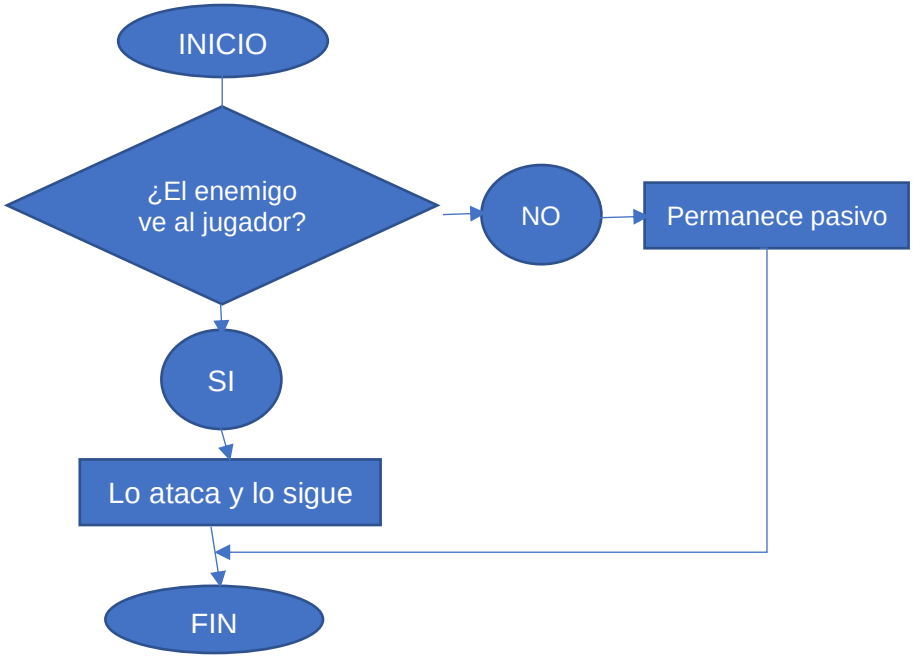
El videojuego consiste en modificar una plantilla de Unity agregando objetivos y más cuartos y enemigos, para que, al rentar el juego a un centro educativo, los niños puedan desarrollar estrategias para completar el juego en el menor tiempo posible y de seguir instrucciones/ordenes. Al mismo tiempo, los niños pueden empezar a identificar posibles estructuras de datos aplicadas en el juego.

Algoritmo (diagrama de flujo y pseudocódigo)

1. Mostrar el menú y las opciones
2. Si se selecciona la opción de invencibilidad
    a. El jugador no recibe daño
3. Si se selecciona la opción de contador de fotogramas
    a. Se muestran la cantidad de fotogramas por segundos
4. Si se selecciona la opción controles
    a. Se muestran los controles del juego tanto para teclado como para mando de Xbox
5. Si se selecciona la opción de captura de pantalla
    a. Se tomará una captura de lo que hay en la pantalla
6. Mostrar las barras de salud, jet pack y munición
    a. Si el jugador recibe daño
        i. Disminuir su salud
    b. Si se reduce el combustible del jet pack
        i. Se recarga el combustible
    c. Si se reduce la munición de las armas
        i. Se recarga la munición
7. Mostrar los objetivos
8. Definir qué objetivos son obligatorios y cuales opcionales
9. Si se completa un nivel, pasa al siguiente
    a. Si no, se reinicia el juego
10. Si el jugador tiene una salud igual a cero

a. Mostrar un mensaje de “Game Over”

Enemigo



INICIO

SI el enemigo ve al jugador ENTONCES

Lo ataca y lo sigue

FIN SI

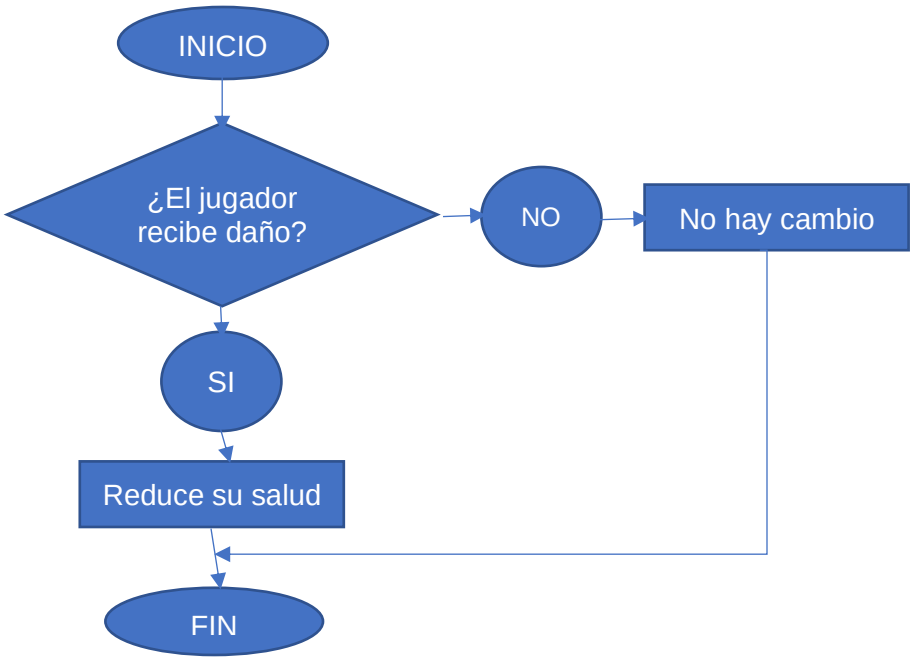
DE LO CONTRARIO

Permanece pasivo

FIN DE LO CONTRARIO

FIN

Salud



INICIO

SI el jugador recibe daño ENTONCES

Reduce su salud

FIN SI

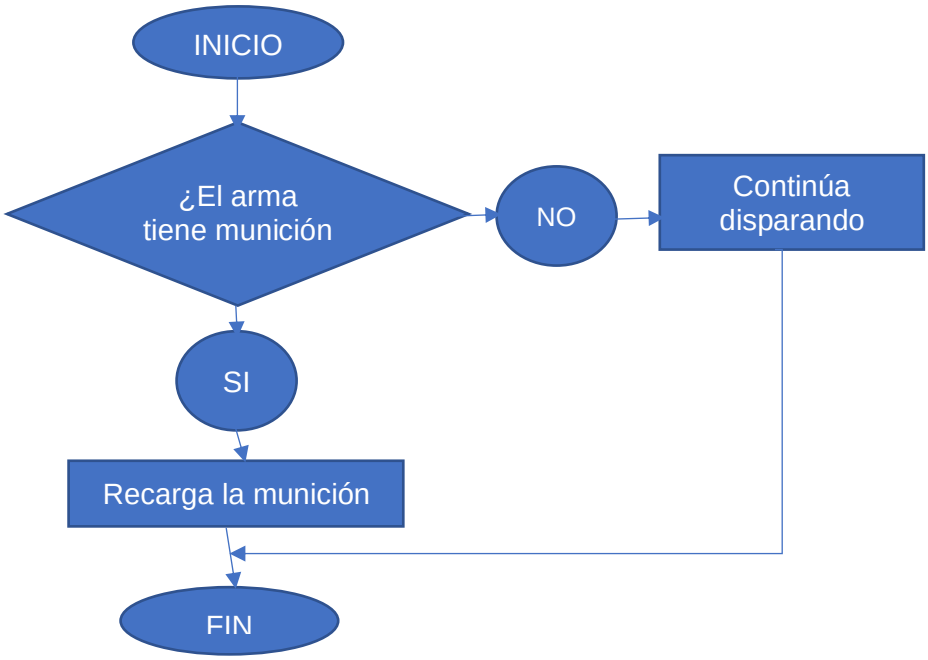
DE LO CONTRARIO

No hay cambio

FIN DE LO CONTRARIO

FIN

Munición arma



INICIO

SI el arma tiene munición baja ENTONCES

Recargar la munición

FIN SI

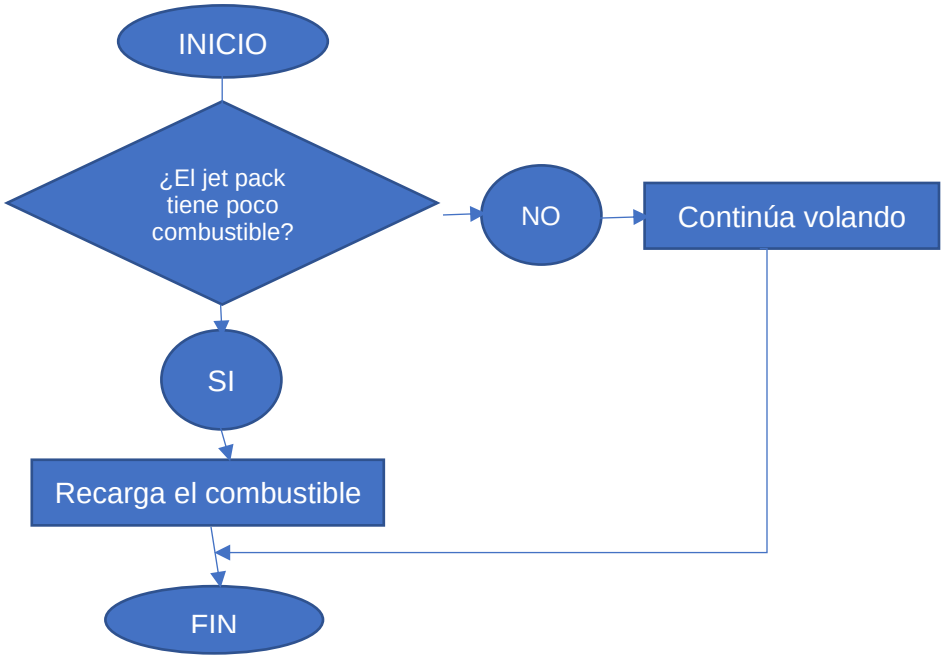
DE LO CONTRARIO

Continuar disparando

FIN DE LO CONTRARIO

FIN

Jet pack



INICIO

SI el jet pack tiene poco combustible ENTONCES

Recargar el combustible

FIN SI

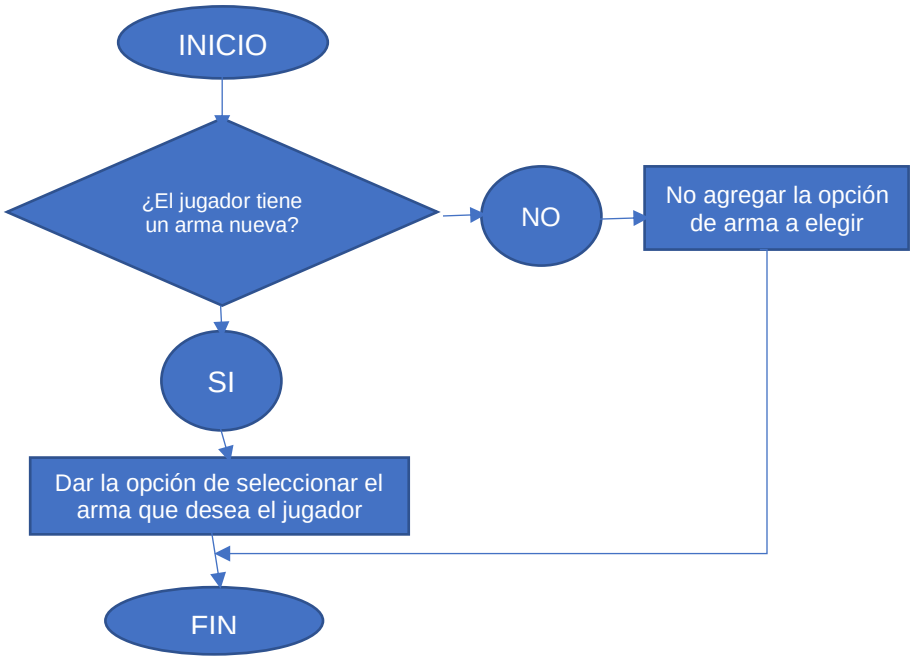
DE LO CONTRARIO

Continuar volando

FIN DE LO CONTRARIO

FIN

Agarrar arma



INICIO

SI el jugador obtiene una nueva arma ENTONCES

Dar la opción de seleccionar el arma que desea el jugador

FIN SI

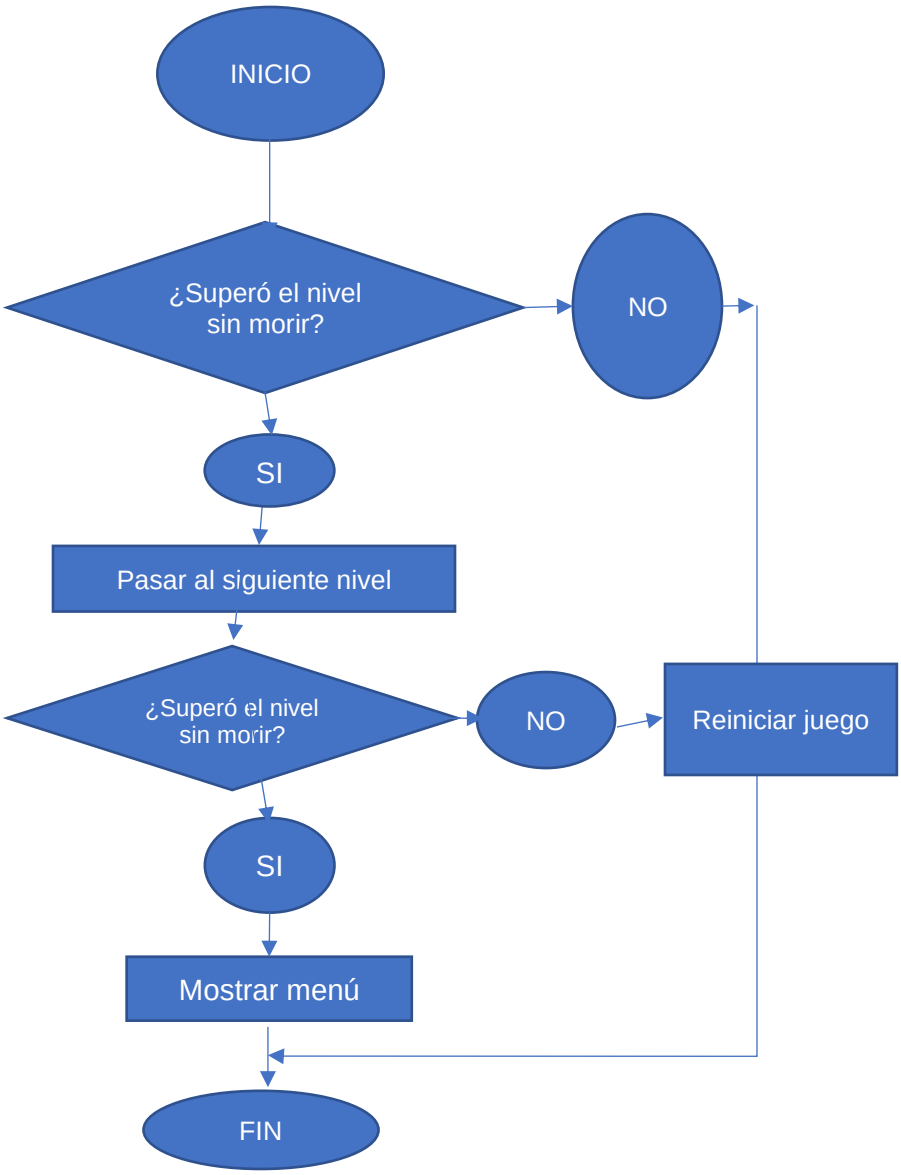
DE LO CONTRARIO

No agregar la nueva opción de arma a elegir

FIN DE LO CONTRARIO

FIN

Niveles del juego



INICIO

SI supera el nivel sin morir ENTONCES

Pasa al siguiente nivel

SI supera el nivel sin morir ENTONCES

Pasa al siguiente nivel

FIN SI

DE LO CONTRARIO

Se reinicia el juego

FIN DE LO CONTARIO

FIN SI

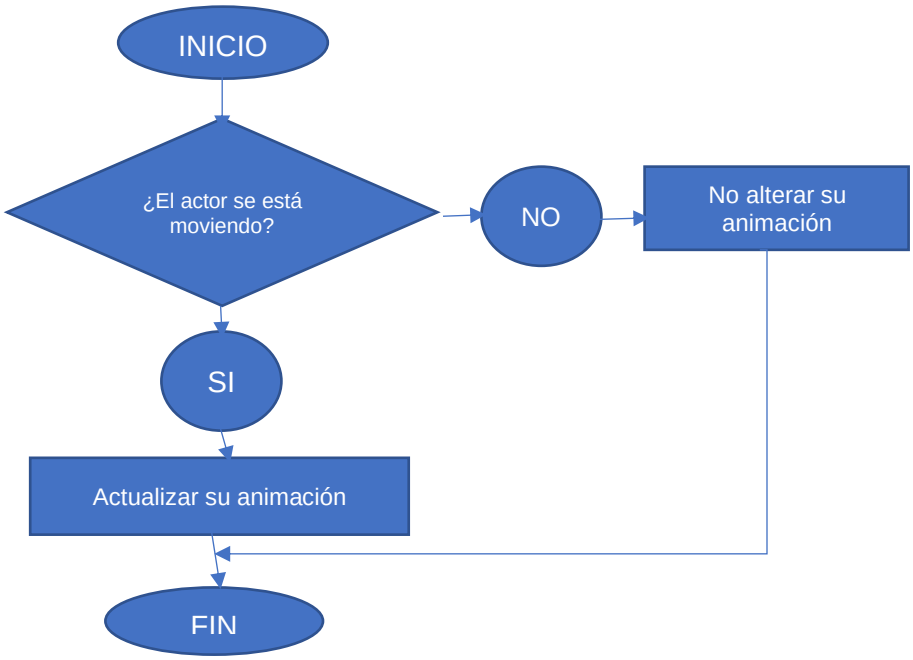
DE LO CONTRARIO

Se reinicia el juego

FIN DE LO CONTRARIO

FIN

Animaciones



INICIO

SI el actor se mueve ENTONCES

Actualizar su animación

FIN SI

DE LO CONTRARIO

No alterar su animación actual

FIN DE LO CONTRARIO

FIN

Código fuente

Actor.cs

```
using UnityEngine;

// This class contains general information describing an actor (player or enemies).
// It is mostly used for AI detection logic and determining if an actor is friend or foe
public class Actor : MonoBehaviour
{
    [Tooltip("Represents the affiliation (or team) of the actor. Actors of the same affiliation are friendly to eachother")]
    public int affiliation;
    [Tooltip("Represents point where other actors will aim when they attack this actor")]
    public Transform aimPoint;

    ActorsManager m_ActorsManager;

    private void Start()
    {
        m_ActorsManager = GameObject.FindObjectOfType<ActorsManager>();
        DebugUtility.HandleErrorIfNullFindObject<ActorsManager, Actor>(m_ActorsManager, this);

        // Register as an actor
        // Agregar un actor
        if (!m_ActorsManager.actors.Contains(this))
        {
            m_ActorsManager.actors.Add(this);
        }
    }

    private void OnDestroy()
    {
        // Unregister as an actor
        // Eliminar un actor
        if (m_ActorsManager)
        {
            m_ActorsManager.actors.Remove(this);
        }
    }
}
```

AudioManager.cs

```
using UnityEngine;
using UnityEngine.Audio;

public class AudioManager : MonoBehaviour
{
    public AudioMixer[] audioMixers;
    // Se va a evaluar la duración del audio
    public AudioMixerGroup[] FindMatchingGroups(string subPath)
    {
        for (int i = 0; i < audioMixers.Length; i++)
        {
            AudioMixerGroup[] results = audioMixers[i].FindMatchingGroups(subPath);
            if (results != null && results.Length != 0)
            {
                return results;
            }
        }

        return null;
    }
    // Se va a evaluar la duración del audio
    public void SetFloat(string name, float value)
    {
        for (int i = 0; i < audioMixers.Length; i++)
        {
            if (audioMixers[i] != null)
            {
                audioMixers[i].SetFloat(name, value);
            }
        }
    }
    // Se va a evaluar la duración del audio
    public void GetFloat(string name, out float value)
    {
        value = 0f;
        for (int i = 0; i < audioMixers.Length; i++)
        {
            if (audioMixers[i] != null)
            {
                audioMixers[i].GetFloat(name, out value);
                break;
            }
        }
    }
```

```
}
```

ChargedProjectileEffectsHandler.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ChargedProjectileEffectsHandler : MonoBehaviour
{
    [Tooltip("Object that will be affected by charging scale & color changes")]
    public GameObject chargingObject;
    [Tooltip("Scale of the charged object based on charge")]
    public MinMaxVector3 scale;
    [Tooltip("Color of the charged object based on charge")]
    public MinMaxColor color;

    MeshRenderer[] m_AffectedRenderers;
    ProjectileBase m_ProjectileBase;

    private void OnEnable()
    {
        m_ProjectileBase = GetComponent<ProjectileBase>();
        DebugUtility.HandleErrorIfNullGetComponent<ProjectileBase, ChargedProjectileEffect-
sHandler>(m_ProjectileBase, this, gameObject);

        m_ProjectileBase.onShoot += OnShoot;

        m_AffectedRenderers = chargingObject.GetComponentsInChildren<MeshRenderer>();
        // Cada vez que se lanza un corete, se altera los renders
        foreach (var ren in m_AffectedRenderers)
        {
            ren.sharedMaterial = Instantiate(ren.sharedMaterial);
        }
    }

    void OnShoot()
    {
        chargingObject.transform.localScale = scale.GetValueFromRatio(m_ProjectileBase.initial-
Charge);
        // Cada vez que se lanza un corete, va a cambiar de color el arma
        foreach (var ren in m_AffectedRenderers)
        {
```

```
            ren.sharedMaterial.SetColor("_Color", color.GetValueFromRatio(m_ProjectileBase.ini-
tialCharge));
        }
    }
}
```

ChargedWeaponEffectsHandler.cs

```
using UnityEngine;

[RequireComponent(typeof(AudioSource))]
public class ChargedWeaponEffectsHandler : MonoBehaviour
{
    [Header("Visual")]
    [Tooltip("Object that will be affected by charging scale & color changes")]
    public GameObject chargingObject;
    [Tooltip("The spinning frame")]
    public GameObject spinningFrame;
    [Tooltip("Scale of the charged object based on charge")]
    public MinMaxVector3 scale;

    [Header("Particles")]
    [Tooltip("Particles to create when charging")]
    public GameObject diskOrbitParticlePrefab;
    [Tooltip("Local position offset of the charge particles (relative to this transform)")]
    public Vector3 offset;
    [Tooltip("Parent transform for the particles (Optional)")]
    public Transform parentTransform;
    [Tooltip("Orbital velocity of the charge particles based on charge")]
    public MinMaxFloat orbitY;
    [Tooltip("Radius of the charge particles based on charge")]
    public MinMaxVector3 radius;
    [Tooltip("Idle spinning speed of the frame based on charge")]
    public MinMaxFloat spinningSpeed;
    //Lo ocupo para que haya sonido al usar un arma luego de cierto tiempo
    [Header("Sound")]
    [Tooltip("Audio clip for charge SFX")]
    public AudioClip chargeSound;
    [Tooltip("Sound played in loop after the change is full for this weapon")]
    public AudioClip loopChargeWeaponSFX;
    [Tooltip("Duration of the cross fade between the charge and the loop sound")]
    public float fadeLoopDuration = 0.5f;

    public GameObject particleInstance { get; set; }
```

```
    ParticleSystem m_DiskOrbitParticle;
    WeaponController m_WeaponController;
    ParticleSystem.VelocityOverLifetimeModule m_VelocityOverTimeModule;

    AudioSource m_AudioSource;
    AudioSource m_AudioSourceLoop;

    float m_LastChargeTriggerTimestamp;
    float m_ChargeRatio;
    float m_EndchargeTime;

    void Awake()
    {
        m_LastChargeTriggerTimestamp = 0.0f;

        // The charge effect needs it's own Audi-
oSources, since it will play on top of the other gun sounds
        m_AudioSource = gameObject.AddComponent<AudioSource>();
        m_AudioSource.clip = chargeSound;
        m_AudioSource.playOnAwake = false;
        m_AudioSource.outputAudioMixerGroup = AudioUtility.GetAudioGroup(AudioUtility.Audi-
oGroups.WeaponChargeBuildup);

        // create a second audio source, to play the sound with a delay
        m_AudioSourceLoop = gameObject.AddComponent<AudioSource>();
        m_AudioSourceLoop.clip = loopChargeWeaponSFX;
        m_AudioSourceLoop.playOnAwake = false;
        m_AudioSourceLoop.loop = true;
        m_AudioSourceLoop.outputAudioMixerGroup = AudioUtility.GetAudioGroup(AudioUtil-
ity.AudioGroups.WeaponChargeLoop);
    }

    void SpawnParticleSystem()
    {
        particleInstance = Instantiate(diskOrbitParticlePrefab, parentTrans-
form != null ? parentTransform : transform);
        particleInstance.transform.localPosition += offset;

        FindReferences();
    }

    public void FindReferences()
    {
        m_DiskOrbitParticle = particleInstance.GetComponent<ParticleSystem>();
```

```
        DebugUtility.HandleErrorIfNullGetComponent<ParticleSystem, ChargedWeaponEffect-
sHandler>(m_DiskOrbitParticle, this, particleInstance.gameObject);

        m_WeaponController = GetComponent<WeaponController>();
        DebugUtility.HandleErrorIfNullGetComponent<WeaponController, ChargedWeaponEffect-
sHandler>(m_WeaponController, this, gameObject);

        m_VelocityOverTimeModule = m_DiskOrbitParticle.velocityOverLifetime;
    }

    void Update()
    {
        if (particleInstance == null)
            SpawnParticleSystem();

        m_DiskOrbitParticle.gameObject.SetActive(m_WeaponController.isWeaponActive);
        m_ChargeRatio = m_WeaponController.currentCharge;

        chargingObject.transform.localScale = scale.GetValueFromRatio(m_ChargeRatio);
        if (spinningFrame != null)
        {
            spinningFrame.transform.localRotation *= Quaternion.Euler(0, spinningSpeed.Get-
ValueFromRatio(m_ChargeRatio) * Time.deltaTime, 0);
        }

        m_VelocityOverTimeModule.orbitalY = orbitY.GetValueFromRatio(m_ChargeRatio);
        m_DiskOrbitParticle.transform.localScale = radius.GetValueFrom-
Ratio(m_ChargeRatio * 1.1f);

        // update sound's volume and pitch
        if (m_ChargeRatio > 0)
        {
            if (!m_AudioSource.isPlaying && m_WeaponController.LastChargeTrigger-
Timestamp > m_LastChargeTriggerTimestamp)
            {
                m_LastChargeTriggerTimestamp = m_WeaponController.LastChargeTrigger-
Timestamp;
                m_EndchargeTime = Time.time + chargeSound.length;

                m_AudioSource.Play();
                m_AudioSourceLoop.Play();
            }

            float volumeRatio = Mathf.Clamp01((m_EndchargeTime - Time.time - fadeLoopDura-
tion) / fadeLoopDuration);
```

```
            m_AudioSource.volume = volumeRatio;
            m_AudioSourceLoop.volume = 1 - volumeRatio;
        }
        else
        {
            m_AudioSource.Stop();
            m_AudioSourceLoop.Stop();
        }
    }
}
```

Damageable.cs

```
using UnityEngine;

public class Damageable : MonoBehaviour
{
    [Tooltip("Multiplier to apply to the received damage")]
    public float damageMultiplier = 1f;
    [Range(0, 1)]
    [Tooltip("Multiplier to apply to self damage")]
    public float sensibilityToSelfdamage = 0.5f;

    public Health health { get; private set; }

    void Awake()
    {
        // find the health component either at the same level, or higher in the hierarchy
        health = GetComponent<Health>();
        if (!health)
        {
            health = GetComponentInParent<Health>();
        }
    }

    public void InflictDamage(float damage, bool isExplosionDamage, GameObject dam-
ageSource)
    {

        if(health)
        {
            var totalDamage = damage;
```

```
            // skip the crit multiplier if it's from an explosion
            if (!isExplosionDamage)
            {
                totalDamage *= damageMultiplier;
            }

            // potentially reduce damages if inflicted by self
            if (health.gameObject == damageSource)
            {
                totalDamage *= sensibilityToSelfdamage;
            }

            // apply the damages
            health.TakeDamage(totalDamage, damageSource);
        }
    }
}
```

DamageArea.cs

```
using System.Collections.Generic;
using UnityEngine;

public class DamageArea : MonoBehaviour
{
    [Tooltip("Area of damage when the projectile hits something")]
    public float areaOfEffectDistance = 5f;
    [Tooltip("Damage multiplier over distance for area of effect")]
    public AnimationCurve damageRatioOverDistance;

    [Header("Debug")]
    [Tooltip("Color of the area of effect radius")]
    public Color areaOfEffectColor = Color.red * 0.5f;

    public void InflictDamageInArea(float damage, Vector3 center, LayerMask layers, Que-
ryTriggerInteraction interaction, GameObject owner)
    {
        Dictionary<Health, Damageable> uniqueDamagedHealths = new Diction-
ary<Health, Damageable>();

        // Create a collection of unique health components that would be dam-
aged in the area of effect (in order to avoid damaging a same entity multiple times)
        Collider[] affectedColliders = Physics.OverlapSphere(center, areaOfEffectDistance, lay-
ers, interaction);
```

```
        foreach (var coll in affectedColliders)
        {
            Damageable damageable = coll.GetComponent<Damageable>();
            if (damageable)
            {
                Health health = damageable.GetComponentInParent<Health>();
                if (health && !uniqueDamagedHealths.ContainsKey(health))
                {
                    uniqueDamagedHealths.Add(health, damageable);
                }
            }
        }

        // Apply damages with distance falloff
        foreach (Damageable uniqueDamageable in uniqueDamagedHealths.Values)
        {
            float distance = Vector3.Distance(uniqueDamageable.transform.position, transform.po-
sition);
            uniqueDamageable.InflictDamage(damage * damageRatioOverDistance.Evaluate(dis-
tance / areaOfEffectDistance), true, owner);
        }
    }

    private void OnDrawGizmosSelected()
    {
        Gizmos.color = areaOfEffectColor;
        Gizmos.DrawSphere(transform.position, areaOfEffectDistance);
    }
}
```

DetectionModule.cs

```
using System.Linq;
using UnityEngine;
using UnityEngine.Events;

public class DetectionModule : MonoBehaviour
{
    [Tooltip("The point representing the source of target-detection raycasts for the enemy AI")]
    public Transform detectionSourcePoint;
    [Tooltip("The max distance at which the enemy can see targets")]
    public float detectionRange = 20f;
    [Tooltip("The max distance at which the enemy can attack its target")]
    public float attackRange = 10f;
    [Tooltip("Time before an enemy abandons a known target that it can't see anymore")]
```

```
    public float knownTargetTimeout = 4f;
    [Tooltip("Optional animator for OnShoot animations")]
    public Animator animator;

    public UnityAction onDetectedTarget;
    public UnityAction onLostTarget;

    public GameObject knownDetectedTarget { get; private set; }
    public bool isTargetInAttackRange { get; private set; }
    public bool isSeeingTarget { get; private set; }
    public bool hadKnownTarget { get; private set; }

    protected float m_TimeLastSeenTarget = Mathf.NegativeInfinity;

    ActorsManager m_ActorsManager;

    const string k_AnimAttackParameter = "Attack";
    const string k_AnimOnDamagedParameter = "OnDamaged";

    protected virtual void Start()
    {
        m_ActorsManager = FindObjectOfType<ActorsManager>();
        DebugUtility.HandleErrorIfNullFindObject<ActorsManager, DetectionModule>(m_Ac-
torsManager, this);
    }

    public virtual void HandleTargetDetection(Actor actor, Collider[] selfColliders)
    {
        // Handle known target detection timeout
        if (knownDetectedTarget && !isSeeingTarget && (Time.time - m_TimeLastSeenTar-
get) > knownTargetTimeout)
        {
            knownDetectedTarget = null;
        }

        // Find the closest visible hostile actor
        float sqrDetectionRange = detectionRange * detectionRange;
        isSeeingTarget = false;
        float closestSqrDistance = Mathf.Infinity;
        foreach (Actor otherActor in m_ActorsManager.actors)
        {
            if (otherActor.affiliation != actor.affiliation)
            {
                float sqrDistance = (otherActor.transform.position - detectionSourcePoint.posi-
tion).sqrMagnitude;
```

```
            if (sqrDistance < sqrDetectionRange && sqrDistance < closestSqrDistance)
            {
                // Check for obstructions
                RaycastHit[] hits = Physics.RaycastAll(detectionSourcePoint.position, (otherAc-
tor.aimPoint.position - detectionSourcePoint.position).normalized, detectionRange, -1, Que-
ryTriggerInteraction.Ignore);
                RaycastHit closestValidHit = new RaycastHit();
                closestValidHit.distance = Mathf.Infinity;
                bool foundValidHit = false;
                foreach (var hit in hits)
                {
                    if (!selfColliders.Contains(hit.collider) && hit.distance < closestValidHit.distance)
                    {
                        closestValidHit = hit;
                        foundValidHit = true;
                    }
                }

                if (foundValidHit)
                {
                    Actor hitActor = closestValidHit.collider.GetComponentInParent<Actor>();
                    if (hitActor == otherActor)
                    {
                        isSeeingTarget = true;
                        closestSqrDistance = sqrDistance;

                        m_TimeLastSeenTarget = Time.time;
                        knownDetectedTarget = otherActor.aimPoint.gameObject;
                    }
                }
            }
        }
    }

    isTargetInAttackRange = knownDetectedTarget != null && Vector3.Distance(trans-
form.position, knownDetectedTarget.transform.position) <= attackRange;

    // Detection events
    if (!hadKnownTarget &&
        knownDetectedTarget != null)
    {
        OnDetect();
    }

    if (hadKnownTarget &&
```

```
            knownDetectedTarget == null)
        {
            OnLostTarget();
        }

        // Remember if we already knew a target (for next frame)
        hadKnownTarget = knownDetectedTarget != null;
    }

    public virtual void OnLostTarget()
    {
        if (onLostTarget != null)
        {
            onLostTarget.Invoke();
        }
    }

    public virtual void OnDetect()
    {
        if (onDetectedTarget != null)
        {
            onDetectedTarget.Invoke();
        }
    }

    public virtual void OnDamaged(GameObject damageSource)
    {
        m_TimeLastSeenTarget = Time.time;
        knownDetectedTarget = damageSource;

        if (animator)
        {
            animator.SetTrigger(k_AnimOnDamagedParameter);
        }
    }

    public virtual void OnAttack()
    {
        if (animator)
        {
            animator.SetTrigger(k_AnimAttackParameter);
        }
    }
}
```

DisplayMessage.cs

```
using UnityEngine;

public class DisplayMessage : MonoBehaviour
{
    [Tooltip("The text that will be displayed")]
    [TextArea]
    public string message;
    [Tooltip("Prefab for the message")]
    public GameObject messagePrefab;
    [Tooltip("Delay before displaying the message")]
    public float delayBeforeShowing;

    float m_InitTime = float.NegativeInfinity;
    bool m_WasDisplayed;
    DisplayMessageManager m_DisplayMessageManager;

    void Start()
    {
        m_InitTime = Time.time;
        m_DisplayMessageManager = FindObjectOfType<DisplayMessageManager>();
        DebugUtility.HandleErrorIfNullFindObject<DisplayMessageManager, Dis-
playMessage>(m_DisplayMessageManager, this);
    }

    // Update is called once per frame
    void Update()
    {
        if (m_WasDisplayed)
            return;

        if (Time.time - m_InitTime > delayBeforeShowing)
        {
            var messageInstance = Instantiate(messagePrefab, m_DisplayMessageManager.Dis-
playMessageRect);
            var notification = messageInstance.GetComponent<NotificationToast>();
            if (notification)
            {
                notification.Initialize(message);
            }

            m_WasDisplayed = true;
        }
```

```
    }
}
```

EnemyController.cs

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.AI;
using UnityEngine.Events;

[RequireComponent(typeof(Health), typeof(Actor), typeof(NavMeshAgent))]
public class EnemyController : MonoBehaviour
{
    [System.Serializable]
    public struct RendererIndexData
    {
        public Renderer renderer;
        public int materialIndex;

        public RendererIndexData(Renderer renderer, int index)
        {
            this.renderer = renderer;
            this.materialIndex = index;
        }
    }

    [Header("Parameters")]
    [Tooltip("The Y height at which the enemy will be automati-
cally killed (if it falls off of the level)")]
    public float selfDestructYHeight = -20f;
    [Tooltip("The distance at which the enemy considers that it has reached its current path des-
tination point")]
    public float pathReachingRadius = 2f;
    [Tooltip("The speed at which the enemy rotates")]
    public float orientationSpeed = 10f;
    [Tooltip("Delay after death where the GameObject is destroyed (to allow for animation)")]
    public float deathDuration = 0f;


    [Header("Weapons Parameters")]
    [Tooltip("Allow weapon swapping for this enemy")]
    public bool swapToNextWeapon = false;
    [Tooltip("Time delay between a weapon swap and the next attack")]
    public float delayAfterWeaponSwap = 0f;
```

```
[Header("Eye color")]
[Tooltip("Material for the eye color")]
public Material eyeColorMaterial;
[Tooltip("The default color of the bot's eye")]
[ColorUsageAttribute(true, true)]
public Color defaultEyeColor;
[Tooltip("The attack color of the bot's eye")]
[ColorUsageAttribute(true, true)]
public Color attackEyeColor;

[Header("Flash on hit")]
[Tooltip("The material used for the body of the hoverbot")]
public Material bodyMaterial;
[Tooltip("The gradient representing the color of the flash on hit")]
[GradientUsageAttribute(true)]
public Gradient onHitBodyGradient;
[Tooltip("The duration of the flash on hit")]
public float flashOnHitDuration = 0.5f;

[Header("Sounds")]
[Tooltip("Sound played when recieving damages")]
public AudioClip damageTick;

[Header("VFX")]
[Tooltip("The VFX prefab spawned when the enemy dies")]
public GameObject deathVFX;
[Tooltip("The point at which the death VFX is spawned")]
public Transform deathVFXSpawnPoint;

[Header("Loot")]
[Tooltip("The object this enemy can drop when dying")]
public GameObject lootPrefab;
[Tooltip("The chance the object has to drop")]
[Range(0, 1)]
public float dropRate = 1f;

[Header("Debug Display")]
[Tooltip("Color of the sphere gizmo representing the path reaching range")]
public Color pathReachingRangeColor = Color.yellow;
[Tooltip("Color of the sphere gizmo representing the attack range")]
public Color attackRangeColor = Color.red;
[Tooltip("Color of the sphere gizmo representing the detection range")]
public Color detectionRangeColor = Color.blue;

public UnityAction onAttack;
```

```
    public UnityAction onDetectedTarget;
    public UnityAction onLostTarget;
    public UnityAction onDamaged;


    List<RendererIndexData> m_BodyRenderers = new List<RendererIndexData>();
    MaterialPropertyBlock m_BodyFlashMaterialPropertyBlock;
    float m_LastTimeDamaged = float.NegativeInfinity;

    RendererIndexData m_EyeRendererData;
    MaterialPropertyBlock m_EyeColorMaterialPropertyBlock;

    public PatrolPath patrolPath { get; set; }
    public GameObject knownDetectedTarget => m_DetectionModule.knownDetectedTarget;
    public bool isTargetInAttackRange => m_DetectionModule.isTargetInAttackRange;
    public bool isSeeingTarget => m_DetectionModule.isSeeingTarget;
    public bool hadKnownTarget => m_DetectionModule.hadKnownTarget;
    public NavMeshAgent m_NavMeshAgent { get; private set; }
    public DetectionModule m_DetectionModule { get; private set; }

    int m_PathDestinationNodeIndex;
    EnemyManager m_EnemyManager;
    ActorsManager m_ActorsManager;
    Health m_Health;
    Actor m_Actor;
    Collider[] m_SelfColliders;
    GameFlowManager m_GameFlowManager;
    bool m_WasDamagedThisFrame;
    float m_LastTimeWeaponSwapped = Mathf.NegativeInfinity;
    int m_CurrentWeaponIndex;
    WeaponController m_CurrentWeapon;
    WeaponController[] m_Weapons;
    NavigationModule m_NavigationModule;

    void Start()
    {
        m_EnemyManager = FindObjectOfType<EnemyManager>();
        DebugUtility.HandleErrorIfNullFindObject<EnemyManager, EnemyController>(m_Enemy-
Manager, this);

        m_ActorsManager = FindObjectOfType<ActorsManager>();
        DebugUtility.HandleErrorIfNullFindObject<ActorsManager, EnemyController>(m_Ac-
torsManager, this);

        m_EnemyManager.RegisterEnemy(this);
```

```
        m_Health = GetComponent<Health>();
        DebugUtility.HandleErrorIfNullGetComponent<Health, EnemyControl-
ler>(m_Health, this, gameObject);

        m_Actor = GetComponent<Actor>();
        DebugUtility.HandleErrorIfNullGetComponent<Actor, EnemyController>(m_Ac-
tor, this, gameObject);

        m_NavMeshAgent = GetComponent<NavMeshAgent>();
        m_SelfColliders = GetComponentsInChildren<Collider>();

        m_GameFlowManager = FindObjectOfType<GameFlowManager>();
        DebugUtility.HandleErrorIfNullFindObject<GameFlowManager, EnemyControl-
ler>(m_GameFlowManager, this);

        // Subscribe to damage & death actions
        m_Health.onDie += OnDie;
        m_Health.onDamaged += OnDamaged;

        // Find and initialize all weapons
        FindAndInitializeAllWeapons();
        var weapon = GetCurrentWeapon();
        weapon.ShowWeapon(true);

        var detectionModules = GetComponentsInChildren<DetectionModule>();
        DebugUtility.HandleErrorIfNoComponentFound<DetectionModule, EnemyController>(de-
tectionModules.Length, this, gameObject);
        DebugUtility.HandleWarningIfDuplicateObjects<DetectionModule, EnemyController>(de-
tectionModules.Length, this, gameObject);
        // Initialize detection module
        m_DetectionModule = detectionModules[0];
        m_DetectionModule.onDetectedTarget += OnDetectedTarget;
        m_DetectionModule.onLostTarget += OnLostTarget;
        onAttack += m_DetectionModule.OnAttack;

        var navigationModules = GetComponentsInChildren<NavigationModule>();
        DebugUtility.HandleWarningIfDuplicateObjects<DetectionModule, EnemyController>(de-
tectionModules.Length, this, gameObject);
        // Override navmesh agent data
        if (navigationModules.Length > 0)
        {
            m_NavigationModule = navigationModules[0];
            m_NavMeshAgent.speed = m_NavigationModule.moveSpeed;
            m_NavMeshAgent.angularSpeed = m_NavigationModule.angularSpeed;
```

```
            m_NavMeshAgent.acceleration = m_NavigationModule.acceleration;
        }

        foreach (var renderer in GetComponentsInChildren<Renderer>(true))
        {
            for (int i = 0; i < renderer.sharedMaterials.Length; i++)
            {
                if (renderer.sharedMaterials[i] == eyeColorMaterial)
                {
                    m_EyeRendererData = new RendererIndexData(renderer, i);
                }

                if (renderer.sharedMaterials[i] == bodyMaterial)
                {
                    m_BodyRenderers.Add(new RendererIndexData(renderer, i));
                }
            }
        }

        m_BodyFlashMaterialPropertyBlock = new MaterialPropertyBlock();

        // Check if we have an eye renderer for this enemy
        if (m_EyeRendererData.renderer != null)
        {
            m_EyeColorMaterialPropertyBlock = new MaterialPropertyBlock();
            m_EyeColorMaterialPropertyBlock.SetColor("_EmissionColor", defaultEyeColor);
            m_EyeRendererData.renderer.SetPropertyBlock(m_EyeColorMaterialProperty-
Block, m_EyeRendererData.materialIndex);
        }
    }

    void Update()
    {
        EnsureIsWithinLevelBounds();

        m_DetectionModule.HandleTargetDetection(m_Actor, m_SelfColliders);

        Color currentColor = onHitBodyGradient.Evalu-
ate((Time.time - m_LastTimeDamaged) / flashOnHitDuration);
        m_BodyFlashMaterialPropertyBlock.SetColor("_EmissionColor", currentColor);
        foreach (var data in m_BodyRenderers)
        {
            data.renderer.SetPropertyBlock(m_BodyFlashMaterialPropertyBlock, data.materialIn-
dex);
        }
```

```
        m_WasDamagedThisFrame = false;
    }

    void EnsureIsWithinLevelBounds()
    {
        // at every frame, this tests for conditions to kill the enemy
        if (transform.position.y < selfDestructYHeight)
        {
            Destroy(gameObject);
            return;
        }
    }

    void OnLostTarget()
    {
        onLostTarget.Invoke();

        // Set the eye attack color and property block if the eye renderer is set
        if (m_EyeRendererData.renderer != null)
        {
            m_EyeColorMaterialPropertyBlock.SetColor("_EmissionColor", defaultEyeColor);
            m_EyeRendererData.renderer.SetPropertyBlock(m_EyeColorMaterialProperty-
Block, m_EyeRendererData.materialIndex);
        }
    }

    void OnDetectedTarget()
    {
        onDetectedTarget.Invoke();

        // Set the eye default color and property block if the eye renderer is set
        if (m_EyeRendererData.renderer != null)
        {
            m_EyeColorMaterialPropertyBlock.SetColor("_EmissionColor", attackEyeColor);
            m_EyeRendererData.renderer.SetPropertyBlock(m_EyeColorMaterialProperty-
Block, m_EyeRendererData.materialIndex);
        }
    }

    public void OrientTowards(Vector3 lookPosition)
    {
        Vector3 lookDirection = Vector3.ProjectOnPlane(lookPosition - transform.position, Vec-
tor3.up).normalized;
        if (lookDirection.sqrMagnitude != 0f)
```

```
        {
            Quaternion targetRotation = Quaternion.LookRotation(lookDirection);
            transform.rotation = Quaternion.Slerp(transform.rotation, targetRotation, Time.del-
taTime * orientationSpeed);
        }
    }

    private bool IsPathValid()
    {
        return patrolPath && patrolPath.pathNodes.Count > 0;
    }

    public void ResetPathDestination()
    {
        m_PathDestinationNodeIndex = 0;
    }

    public void SetPathDestinationToClosestNode()
    {
        if (IsPathValid())
        {
            int closestPathNodeIndex = 0;
            for (int i = 0; i < patrolPath.pathNodes.Count; i++)
            {
                float distanceToPathNode = patrolPath.GetDistanceToNode(transform.position, i);
                if (distanceToPathNode < patrolPath.GetDistanceToNode(transform.posi-
tion, closestPathNodeIndex))
                {
                    closestPathNodeIndex = i;
                }
            }

            m_PathDestinationNodeIndex = closestPathNodeIndex;
        }
        else
        {
            m_PathDestinationNodeIndex = 0;
        }
    }

    public Vector3 GetDestinationOnPath()
    {
        if (IsPathValid())
        {
            return patrolPath.GetPositionOfPathNode(m_PathDestinationNodeIndex);
```

```
        }
        else
        {
            return transform.position;
        }
    }

    public void SetNavDestination(Vector3 destination)
    {
        if (m_NavMeshAgent)
        {
            m_NavMeshAgent.SetDestination(destination);
        }
    }

    public void UpdatePathDestination(bool inverseOrder = false)
    {
        if (IsPathValid())
        {
            // Check if reached the path destination
            if ((transform.position - GetDestinationOnPath()).magnitude <= pathReachingRadius)
            {
                // increment path destination index
                m_PathDestinationNodeIndex = inverseOrder ? (m_PathDestinationNodeIn-
dex - 1) : (m_PathDestinationNodeIndex + 1);
                if (m_PathDestinationNodeIndex < 0)
                {
                    m_PathDestinationNodeIndex += patrolPath.pathNodes.Count;
                }
                if (m_PathDestinationNodeIndex >= patrolPath.pathNodes.Count)
                {
                    m_PathDestinationNodeIndex -= patrolPath.pathNodes.Count;
                }
            }
        }
    }

    void OnDamaged(float damage, GameObject damageSource)
    {
        // test if the damage source is the player
        if (damageSource && damageSource.GetComponent<PlayerCharacterController>())
        {
            // pursue the player
            m_DetectionModule.OnDamaged(damageSource);
```

```
            if (onDamaged != null)
            {
                onDamaged.Invoke();
            }
            m_LastTimeDamaged = Time.time;

            // play the damage tick sound
            if (damageTick && !m_WasDamagedThisFrame)
                AudioUtility.CreateSFX(damageTick, transform.position, AudioUtility.Audi-
oGroups.DamageTick, 0f);

            m_WasDamagedThisFrame = true;
        }
    }

    void OnDie()
    {
        // spawn a particle system when dying
        var vfx = Instantiate(deathVFX, deathVFXSpawnPoint.position, Quaternion.identity);
        Destroy(vfx, 5f);

        // tells the game flow manager to handle the enemy destuction
        m_EnemyManager.UnregisterEnemy(this);

        // loot an object
        if (TryDropItem())
        {
            Instantiate(lootPrefab, transform.position, Quaternion.identity);
        }

        // this will call the OnDestroy function
        Destroy(gameObject, deathDuration);
    }

    private void OnDrawGizmosSelected()
    {
        // Path reaching range
        Gizmos.color = pathReachingRangeColor;
        Gizmos.DrawWireSphere(transform.position, pathReachingRadius);

        if (m_DetectionModule != null)
        {
            // Detection range
            Gizmos.color = detectionRangeColor;
            Gizmos.DrawWireSphere(transform.position, m_DetectionModule.detectionRange);
```

```
            // Attack range
            Gizmos.color = attackRangeColor;
            Gizmos.DrawWireSphere(transform.position, m_DetectionModule.attackRange);
        }
    }

    public void OrientWeaponsTowards(Vector3 lookPosition)
    {
        for (int i = 0; i < m_Weapons.Length; i++)
        {
            // orient weapon towards player
            Vector3 weaponForward = (lookPosition - m_Weapons[i].weaponRoot.transform.posi-
tion).normalized;
            m_Weapons[i].transform.forward = weaponForward;
        }
    }

    public bool TryAtack(Vector3 enemyPosition)
    {
        if (m_GameFlowManager.gameIsEnding)
            return false;

        OrientWeaponsTowards(enemyPosition);

        if ((m_LastTimeWeaponSwapped + delayAfterWeaponSwap) >= Time.time)
            return false;

        // Shoot the weapon
        bool didFire = GetCurrentWeapon().HandleShootInputs(false, true, false);

        if (didFire && onAttack != null)
        {
            onAttack.Invoke();

            if (swapToNextWeapon && m_Weapons.Length > 1)
            {
                int nextWeaponIndex = (m_CurrentWeaponIndex + 1) % m_Weapons.Length;
                SetCurrentWeapon(nextWeaponIndex);
            }
        }

        return didFire;
    }
```

```
    public bool TryDropItem()
    {
        if (dropRate == 0 || lootPrefab == null)
            return false;
        else if (dropRate == 1)
            return true;
        else
            return (Random.value <= dropRate);
    }

    void FindAndInitializeAllWeapons()
    {
        // Check if we already found and initialized the weapons
        if (m_Weapons == null)
        {
            m_Weapons = GetComponentsInChildren<WeaponController>();
            DebugUtility.HandleErrorIfNoComponentFound<WeaponController, EnemyControl-
ler>(m_Weapons.Length, this, gameObject);

            for (int i = 0; i < m_Weapons.Length; i++)
            {
                m_Weapons[i].owner = gameObject;
            }
        }
    }

    public WeaponController GetCurrentWeapon()
    {
        FindAndInitializeAllWeapons();
        // Check if no weapon is currently selected
        if (m_CurrentWeapon == null)
        {
            // Set the first weapon of the weapons list as the current weapon
            SetCurrentWeapon(0);
        }
        DebugUtility.HandleErrorIfNullGetComponent<WeaponController, EnemyControl-
ler>(m_CurrentWeapon, this, gameObject);

        return m_CurrentWeapon;
    }

    void SetCurrentWeapon(int index)
    {
        m_CurrentWeaponIndex = index;
        m_CurrentWeapon = m_Weapons[m_CurrentWeaponIndex];
```

```
        if (swapToNextWeapon)
        {
            m_LastTimeWeaponSwapped = Time.time;
        }
        else
        {
            m_LastTimeWeaponSwapped = Mathf.NegativeInfinity;
        }
    }
}
```

EnemyMaganer.cs

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

public class EnemyManager : MonoBehaviour
{
    PlayerCharacterController m_PlayerController;

    public List<EnemyController> enemies { get; private set; }
    public int numberOfEnemiesTotal { get; private set; }
    public int numberOfEnemiesRemaining => enemies.Count;

    public UnityAction<EnemyController, int> onRemoveEnemy;

    private void Awake()
    {
        m_PlayerController = FindObjectOfType<PlayerCharacterController>();
        DebugUtility.HandleErrorIfNullFindObject<PlayerCharacterController, EnemyMan-
ager>(m_PlayerController, this);

        enemies = new List<EnemyController>();
    }

    public void RegisterEnemy(EnemyController enemy)
    // Agregar un enemigo
    {
        enemies.Add(enemy);

        numberOfEnemiesTotal++;
    }

    public void UnregisterEnemy(EnemyController enemyKilled)
```

```
    // Eliminar un enemigo
    {
        int enemiesRemainingNotification = numberOfEnemiesRemaining - 1;

        if (onRemoveEnemy != null)
        {
            onRemoveEnemy.Invoke(enemyKilled, enemiesRemainingNotification);
        }

        // removes the en-
emy from the list, so that we can keep track of how many are left on the map
        enemies.Remove(enemyKilled);
    }
}
```

EnemyMobile.cs

```
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(EnemyController))]
public class EnemyMobile : MonoBehaviour
{
    public enum AIState
    {
        Patrol,
        Follow,
        Attack,
    }

    public Animator animator;
    [Tooltip("Fraction of the enemy's attack range at which it will stop moving towards tar-
get while attacking")]
    [Range(0f, 1f)]
    public float attackStopDistanceRatio = 0.5f;
    [Tooltip("The random hit damage effects")]
    public ParticleSystem[] randomHitSparks;
    public ParticleSystem[] onDetectVFX;
    public AudioClip onDetectSFX;

    [Header("Sound")]
    public AudioClip MovementSound;
    public MinMaxFloat PitchDistortionMovementSpeed;

    public AIState aiState { get; private set; }
```

```
    EnemyController m_EnemyController;
    AudioSource m_AudioSource;

    const string k_AnimMoveSpeedParameter = "MoveSpeed";
    const string k_AnimAttackParameter = "Attack";
    const string k_AnimAlertedParameter = "Alerted";
    const string k_AnimOnDamagedParameter = "OnDamaged";

    void Start()
    {
        m_EnemyController = GetComponent<EnemyController>();
        DebugUtility.HandleErrorIfNullGetComponent<EnemyController, EnemyMobile>(m_Ene-
myController, this, gameObject);

        m_EnemyController.onAttack += OnAttack;
        m_EnemyController.onDetectedTarget += OnDetectedTarget;
        m_EnemyController.onLostTarget += OnLostTarget;
        m_EnemyController.SetPathDestinationToClosestNode();
        m_EnemyController.onDamaged += OnDamaged;

        // Start patrolling
        aiState = AIState.Patrol;

        // adding a audio source to play the movement sound on it
        m_AudioSource = GetComponent<AudioSource>();
        DebugUtility.HandleErrorIfNullGetComponent<AudioSource, EnemyMobile>(m_Audi-
oSource, this, gameObject);
        m_AudioSource.clip = MovementSound;
        m_AudioSource.Play();
    }

    void Update()
    {
        UpdateAIStateTransitions();
        UpdateCurrentAIState();

        float moveSpeed = m_EnemyController.m_NavMeshAgent.velocity.magnitude;

        // Update animator speed parameter
        animator.SetFloat(k_AnimMoveSpeedParameter, moveSpeed);

        // changing the pitch of the movement sound depending on the movement speed
        m_AudioSource.pitch = Mathf.Lerp(PitchDistortionMovementSpeed.min, PitchDistor-
tionMovementSpeed.max, moveSpeed / m_EnemyController.m_NavMeshAgent.speed);
    }
```

```
    void UpdateAIStateTransitions()
    {
        // Handle transitions
        switch (aiState)
        {
            case AIState.Follow:
                // Transition to attack when there is a line of sight to the target
                if (m_EnemyController.isSeeingTarget && m_EnemyController.isTargetInAt-
tackRange)
                {
                    aiState = AIState.Attack;
                    m_EnemyController.SetNavDestination(transform.position);
                }
                break;
            case AIState.Attack:
                // Transition to follow when no longer a target in attack range
                if (!m_EnemyController.isTargetInAttackRange)
                {
                    aiState = AIState.Follow;
                }
                break;
        }
    }

    void UpdateCurrentAIState()
    {
        // Handle logic
        switch (aiState)
        {
            case AIState.Patrol:
                m_EnemyController.UpdatePathDestination();
                m_EnemyController.SetNavDestination(m_EnemyController.GetDestinationOn-
Path());
                break;
            case AIState.Follow:
                m_EnemyController.SetNavDestination(m_EnemyController.knownDetectedTar-
get.transform.position);
                m_EnemyController.OrientTowards(m_EnemyController.knownDetectedTar-
get.transform.position);
                m_EnemyController.OrientWeaponsTowards(m_EnemyController.knownDetect-
edTarget.transform.position);
                break;
            case AIState.Attack:
```

```
            if (Vector3.Distance(m_EnemyController.knownDetectedTarget.transform.posi-
tion, m_EnemyController.m_DetectionModule.detectionSourcePoint.position)
                >= (attackStopDistanceRatio * m_EnemyController.m_DetectionModule.at-
tackRange))
            {
                m_EnemyController.SetNavDestination(m_EnemyController.knownDetectedTar-
get.transform.position);
            }
            else
            {
                m_EnemyController.SetNavDestination(transform.position);
            }
            m_EnemyController.OrientTowards(m_EnemyController.knownDetectedTar-
get.transform.position);
            m_EnemyController.TryAtack(m_EnemyController.knownDetectedTarget.trans-
form.position);
            break;
        }
    }

    void OnAttack()
    {
        animator.SetTrigger(k_AnimAttackParameter);
    }

    void OnDetectedTarget()
    {
        if (aiState == AIState.Patrol)
        {
            aiState = AIState.Follow;
        }

        for (int i = 0; i < onDetectVFX.Length; i++)
        {
            onDetectVFX[i].Play();
        }

        if (onDetectSFX)
        {
            AudioUtility.CreateSFX(onDetectSFX, transform.position, AudioUtility.AudioGroups.En-
emyDetection, 1f);
        }

        animator.SetBool(k_AnimAlertedParameter, true);
    }
```

```
    void OnLostTarget()
    {
        if (aiState == AIState.Follow || aiState == AIState.Attack)
        {
            aiState = AIState.Patrol;
        }

        for (int i = 0; i < onDetectVFX.Length; i++)
        {
            onDetectVFX[i].Stop();
        }

        animator.SetBool(k_AnimAlertedParameter, false);
    }

    void OnDamaged()
    {
        if (randomHitSparks.Length > 0)
        {
            int n = Random.Range(0, randomHitSparks.Length - 1);
            randomHitSparks[n].Play();
        }

        animator.SetTrigger(k_AnimOnDamagedParameter);
    }
}
```

EnemyTurret.cs

```
using UnityEngine;

[RequireComponent(typeof(EnemyController))]
public class EnemyTurret : MonoBehaviour
{
    public enum AIState
    {
        Idle,
        Attack,
    }

    public Transform turretPivot;
    public Transform turretAimPoint;
    public Animator animator;
    public float aimRotationSharpness = 5f;
```

```
    public float lookAtRotationSharpness = 2.5f;
    public float detectionFireDelay = 1f;
    public float aimingTransitionBlendTime = 1f;

    [Tooltip("The random hit damage effects")]
    public ParticleSystem[] randomHitSparks;
    public ParticleSystem[] onDetectVFX;
    public AudioClip onDetectSFX;

    public AIState aiState { get; private set; }

    EnemyController m_EnemyController;
    Health m_Health;
    Quaternion m_RotationWeaponForwardToPivot;
    float m_TimeStartedDetection;
    float m_TimeLostDetection;
    Quaternion m_PreviousPivotAimingRotation;
    Quaternion m_PivotAimingRotation;

    const string k_AnimOnDamagedParameter = "OnDamaged";
    const string k_AnimIsActiveParameter = "IsActive";

    void Start()
    {
        m_Health = GetComponent<Health>();
        DebugUtility.HandleErrorIfNullGetComponent<Health, EnemyTur-
ret>(m_Health, this, gameObject);
        m_Health.onDamaged += OnDamaged;

        m_EnemyController = GetComponent<EnemyController>();
        DebugUtility.HandleErrorIfNullGetComponent<EnemyController, EnemyTurret>(m_Ene-
myController, this, gameObject);

        m_EnemyController.onDetectedTarget += OnDetectedTarget;
        m_EnemyController.onLostTarget += OnLostTarget;

        // Remember the rotation offset between the pivot's forward and the weapon's forward
        m_RotationWeaponForwardToPivot = Quaternion.Inverse(m_EnemyController.GetCur-
rentWeapon().weaponMuzzle.rotation) * turretPivot.rotation;

        // Start with idle
        aiState = AIState.Idle;

        m_TimeStartedDetection = Mathf.NegativeInfinity;
        m_PreviousPivotAimingRotation = turretPivot.rotation;
```

```
    }

    void Update()
    {
        UpdateCurrentAIState();
    }

    void LateUpdate()
    {
        UpdateTurretAiming();
    }

    void UpdateCurrentAIState()
    {
        // Handle logic
        switch (aiState)
        {
            case AIState.Attack:
                bool mustShoot = Time.time > m_TimeStartedDetection + detectionFireDelay;
                // Calculate the desired rotation of our turret (aim at target)
                Vector3 directionToTarget = (m_EnemyController.knownDetectedTarget.trans-
form.position - turretAimPoint.position).normalized;
                Quaternion offsettedTargetRotation = Quaternion.LookRotation(directionToTar-
get) * m_RotationWeaponForwardToPivot;
                m_PivotAimingRotation = Quaternion.Slerp(m_PreviousPivotAimingRotation, offset-
tedTargetRotation, (mustShoot ? aimRotationSharpness : lookAtRotationSharp-
ness) * Time.deltaTime);

                // shoot
                if (mustShoot)
                {
                    Vector3 correctedDirectionToTarget = (m_PivotAimingRotation * Quaternion.In-
verse(m_RotationWeaponForwardToPivot)) * Vector3.forward;

                    m_EnemyController.TryAtack(turretAimPoint.position + correctedDirectionToTar-
get);
                }

                break;
        }
    }

    void UpdateTurretAiming()
    {
        switch (aiState)
```

```
        {
            case AIState.Attack:
                turretPivot.rotation = m_PivotAimingRotation;
                break;
            default:
                // Use the turret rotation of the animation
                turretPivot.rotation = Quaternion.Slerp(m_PivotAimingRotation, turretPivot.rotation, (Time.time - m_TimeLostDetection) / aimingTransitionBlendTime);
                break;
        }

        m_PreviousPivotAimingRotation = turretPivot.rotation;
    }

    void OnDamaged(float dmg, GameObject source)
    {
        if (randomHitSparks.Length > 0)
        {
            int n = Random.Range(0, randomHitSparks.Length - 1);
            randomHitSparks[n].Play();
        }

        animator.SetTrigger(k_AnimOnDamagedParameter);
    }

    void OnDetectedTarget()
    {
        if(aiState == AIState.Idle)
        {
            aiState = AIState.Attack;
        }

        for (int i = 0; i < onDetectVFX.Length; i++)
        {
            onDetectVFX[i].Play();
        }

        if (onDetectSFX)
        {
            AudioUtility.CreateSFX(onDetectSFX, transform.position, AudioUtility.AudioGroups.EnemyDetection, 1f);
        }

        animator.SetBool(k_AnimIsActiveParameter, true);
        m_TimeStartedDetection = Time.time;
```

```
    }

    void OnLostTarget()
    {
        if (aiState == AIState.Attack)
        {
            aiState = AIState.Idle;
        }

        for (int i = 0; i < onDetectVFX.Length; i++)
        {
            onDetectVFX[i].Stop();
        }

        animator.SetBool(k_AnimIsActiveParameter, false);
        m_TimeLostDetection = Time.time;
    }
}
```

GameFlowManager.cs

```
using UnityEngine;
using UnityEngine.SceneManagement;

namespace Unity.FPS.Game
{
    public class GameFlowManager : MonoBehaviour
    {
        [Header("Parameters")] [Tooltip("Duration of the fade-to-black at the end of the game")]
        public float EndSceneLoadDelay = 3f;

        [Tooltip("The canvas group of the fade-to-black screen")]
        public CanvasGroup EndGameFadeCanvasGroup;

        [Header("Win")] [Tooltip("This string has to be the name of the scene you want to load when
winning")]
        public string WinSceneName = "WinScene";

        [Tooltip("Duration of delay before the fade-to-black, if winning")]
        public float DelayBeforeFadeToBlack = 4f;

        [Tooltip("Win game message")]
        public string WinGameMessage;
        [Tooltip("Duration of delay before the win message")]
        public float DelayBeforeWinMessage = 2f;
```

```
    [Tooltip("Sound played on win")] public AudioClip VictorySound;

    [Header("Lose")] [Tooltip("This string has to be the name of the scene you want to load when losing")]
    public string LoseSceneName = "LoseScene";


    public bool GameIsEnding { get; private set; }

    float m_TimeLoadEndGameScene;
    string m_SceneToLoad;

    void Awake()
    {
        EventManager.AddListener<AllObjectivesCompletedEvent>(OnAllObjectivesCompleted);
        EventManager.AddListener<PlayerDeathEvent>(OnPlayerDeath);
    }

    void Start()
    {
        AudioUtility.SetMasterVolume(1);
    }

    void Update()
    {
        if (GameIsEnding)
        {
            float timeRatio = 1 - (m_TimeLoadEndGameScene - Time.time) / EndSceneLoadDelay;
            EndGameFadeCanvasGroup.alpha = timeRatio;

            AudioUtility.SetMasterVolume(1 - timeRatio);

            // See if it's time to load the end scene (after the delay)
            if (Time.time >= m_TimeLoadEndGameScene)
            {
                SceneManager.LoadScene(m_SceneToLoad);
                GameIsEnding = false;
            }
        }
    }

    void OnAllObjectivesCompleted(AllObjectivesCompletedEvent evt) => EndGame(true);
```

```
    //Si el jugador muere, se reinicia el juego
    void OnPlayerDeath(PlayerDeathEvent evt) => EndGame(false);

    void EndGame(bool win)
    {
        // unlocks the cursor before leaving the scene, to be able to click buttons
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;

        // Remember that we need to load the appropriate end scene after a delay
        GameIsEnding = true;
        EndGameFadeCanvasGroup.gameObject.SetActive(true);
        if (win)
        {
            m_SceneToLoad = WinSceneName;
            m_TimeLoadEndGameScene = Time.time + EndSceneLoadDelay + DelayBeforeFadeToBlack;

            // play a sound on win
            var audioSource = gameObject.AddComponent<AudioSource>();
            audioSource.clip = VictorySound;
            audioSource.playOnAwake = false;
            audioSource.outputAudioMixerGroup = AudioUtility.GetAudioGroup(AudioUtility.AudioGroups.HUDVictory);
            audioSource.PlayScheduled(AudioSettings.dspTime + DelayBeforeWinMessage);

            // create a game message
            //var message = Instantiate(WinGameMessagePrefab).GetComponent<DisplayMessage>();
            //if (message)
            //{
            //    message.delayBeforeShowing = delayBeforeWinMessage;
            //    message.GetComponent<Transform>().SetAsLastSibling();
            //}

            DisplayMessageEvent displayMessage = Events.DisplayMessageEvent;
            displayMessage.Message = WinGameMessage;
            displayMessage.DelayBeforeDisplay = DelayBeforeWinMessage;
            EventManager.Broadcast(displayMessage);
        }
        else
        {
            m_SceneToLoad = LoseSceneName;
            m_TimeLoadEndGameScene = Time.time + EndSceneLoadDelay;
        }
```

```
        }

        void OnDestroy()
        {
            EventManager.RemoveListener<AllObjectivesCompletedEvent>(OnAllObjectivesCompleted);
            EventManager.RemoveListener<PlayerDeathEvent>(OnPlayerDeath);
        }
    }
}
```

Health.cs

```
using UnityEngine;
using UnityEngine.Events;

public class Health : MonoBehaviour
{
    [Tooltip("Maximum amount of health")]
    public float maxHealth = 10f;
    [Tooltip("Health ratio at which the critical health vignette starts appearing")]
    public float criticalHealthRatio = 0.3f;

    public UnityAction<float, GameObject> onDamaged;
    public UnityAction<float> onHealed;
    public UnityAction onDie;

    public float currentHealth { get; set; }
    public bool invincible { get; set; }
    public bool canPickup() => currentHealth < maxHealth;

    public float getRatio() => currentHealth / maxHealth;
    public bool isCritical() => getRatio() <= criticalHealthRatio;

    bool m_IsDead;

    private void Start()
    {
        currentHealth = maxHealth;
    }

    public void Heal(float healAmount)
    {
        float healthBefore = currentHealth;
        currentHealth += healAmount;
        currentHealth = Mathf.Clamp(currentHealth, 0f, maxHealth);
```

```
    // call OnHeal action
    float trueHealAmount = currentHealth - healthBefore;
    if (trueHealAmount > 0f && onHealed != null)
    {
        onHealed.Invoke(trueHealAmount);
    }
}

public void TakeDamage(float damage, GameObject damageSource)
{
    if (invincible)
        return;

    float healthBefore = currentHealth;
    currentHealth -= damage;
    currentHealth = Mathf.Clamp(currentHealth, 0f, maxHealth);

    // call OnDamage action
    float trueDamageAmount = healthBefore - currentHealth;
    if (trueDamageAmount > 0f && onDamaged != null)
    {
        onDamaged.Invoke(trueDamageAmount, damageSource);
    }

    HandleDeath();
}

public void Kill()
{
    currentHealth = 0f;

    // call OnDamage action
    if (onDamaged != null)
    {
        onDamaged.Invoke(maxHealth, null);
    }

    HandleDeath();
}

private void HandleDeath()
{
    if (m_IsDead)
        return;
```

```
        // call OnDie action
        if (currentHealth <= 0f)
        {
            if (onDie != null)
            {
                m_IsDead = true;
                onDie.Invoke();
            }
        }
    }
}
```

HealthPickup.cs

```
using UnityEngine;

public class HealthPickup : MonoBehaviour
{
    [Header("Parameters")]
    [Tooltip("Amount of health to heal on pickup")]
    public float healAmount;

    Pickup m_Pickup;

    void Start()
    {
        m_Pickup = GetComponent<Pickup>();
        DebugUtility.HandleErrorIfNullGetComponent<Pickup, Health-
Pickup>(m_Pickup, this, gameObject);

        // Subscribe to pickup action
        m_Pickup.onPick += OnPicked;
    }

    void OnPicked(PlayerCharacterController player)
    {
        Health playerHealth = player.GetComponent<Health>();
        if (playerHealth && playerHealth.canPickup())
        {
            playerHealth.Heal(healAmount);

            m_Pickup.PlayPickupFeedback();
```

```
            Destroy(gameObject);
        }
    }
}
```

Jetpack.cs

```
using UnityEngine;
using UnityEngine.Events;

[RequireComponent(typeof(AudioSource))]
public class Jetpack : MonoBehaviour
{
    [Header("References")]
    [Tooltip("Audio source for jetpack sfx")]
    public AudioSource audioSource;
    [Tooltip("Particles for jetpack vfx")]
    public ParticleSystem[] jetpackVfx;

    [Header("Parameters")]
    [Tooltip("Whether the jetpack is unlocked at the begining or not")]
    public bool isJetpackUnlockedAtStart = false;
    [Tooltip("The strength with which the jetpack pushes the player up")]
    public float jetpackAcceleration = 7f;
    [Range(0f, 1f)]
    [Tooltip("This will affect how much using the jetpack will cancel the gravity value, to start go-
ing up faster. 0 is not at all, 1 is instant")]
    public float jetpackDownwardVelocityCancelingFactor = 1f;

    [Header("Durations")]
    [Tooltip("Time it takes to consume all the jetpack fuel")]
    public float consumeDuration = 1.5f;
    [Tooltip("Time it takes to completely refill the jetpack while on the ground")]
    public float refillDurationGrounded = 2f;
    [Tooltip("Time it takes to completely refill the jetpack while in the air")]
    public float refillDurationInTheAir = 5f;
    [Tooltip("Delay after last use before starting to refill")]
    public float refillDelay = 1f;

    [Header("Audio")]
    [Tooltip("Sound played when using the jetpack")]
    public AudioClip jetpackSFX;

    bool m_CanUseJetpack;
    PlayerCharacterController m_PlayerCharacterController;
```

```
    PlayerInputHandler m_InputHandler;
    float m_LastTimeOfUse;

    // stored ratio for jetpack resource (1 is full, 0 is empty)
    public float currentFillRatio { get; private set; }
    public bool isJetpackUnlocked { get; private set; }

    public bool isPlayergrounded() => m_PlayerCharacterController.isGrounded;

    public UnityAction<bool> onUnlockJetpack;

    void Start()
    {
        isJetpackUnlocked = isJetpackUnlockedAtStart;

        m_PlayerCharacterController = GetComponent<PlayerCharacterController>();
        DebugUtility.HandleErrorIfNullGetComponent<PlayerCharacterController, Jet-
pack>(m_PlayerCharacterController, this, gameObject);

        m_InputHandler = GetComponent<PlayerInputHandler>();
        DebugUtility.HandleErrorIfNullGetComponent<PlayerInputHandler, Jetpack>(m_In-
putHandler, this, gameObject);

        currentFillRatio = 1f;

        audioSource.clip = jetpackSFX;
        audioSource.loop = true;
    }

    void Update()
    {
        // jetpack can only be used if not grounded and jump has been pressed again once in-air
        if(isPlayergrounded())
        {
            m_CanUseJetpack = false;
        }
        else if (!m_PlayerCharacterController.hasJumpedThisFrame && m_InputHandler.Get-
JumpInputDown())
        {
            m_CanUseJetpack = true;
        }

        // jetpack usage
        bool jetpackIsInUse = m_CanUseJetpack && isJetpackUnlocked  && currentFillRa-
tio > 0f && m_InputHandler.GetJumpInputHeld();
```

```
        if(jetpackIsInUse)
        {
            // store the last time of use for refill delay
            m_LastTimeOfUse = Time.time;

            float totalAcceleration = jetpackAcceleration;

            // cancel out gravity
            totalAcceleration += m_PlayerCharacterController.gravityDownForce;

            if (m_PlayerCharacterController.characterVelocity.y < 0f)
            {
                // handle making the jetpack compensate for character's downward velocity with bo-
nus acceleration
                totalAcceleration += ((-m_PlayerCharacterController.characterVelocity.y / Time.del-
taTime) * jetpackDownwardVelocityCancelingFactor);
            }

            // apply the acceleration to character's velocity
            m_PlayerCharacterController.characterVelocity += Vector3.up * totalAccelera-
tion * Time.deltaTime;

            // consume fuel
            currentFillRatio = currentFillRatio - (Time.deltaTime / consumeDuration);

            for (int i = 0; i < jetpackVfx.Length; i++)
            {
                var emissionModulesVFX = jetpackVfx[i].emission;
                emissionModulesVFX.enabled = true;
            }

            if (!audioSource.isPlaying)
                audioSource.Play();
        }
        else
        {
            // refill the meter over time
            if (isJetpackUnlocked && Time.time - m_LastTimeOfUse >= refillDelay)
            {
                float refillRate = 1 / (m_PlayerCharacterController.isGrounded ? refillDura-
tionGrounded : refillDurationInTheAir);
                currentFillRatio = currentFillRatio + Time.deltaTime * refillRate;
            }

            for (int i = 0; i < jetpackVfx.Length; i++)
```

```
            {
                var emissionModulesVFX = jetpackVfx[i].emission;
                emissionModulesVFX.enabled = false;
            }

            // keeps the ratio between 0 and 1
            currentFillRatio = Mathf.Clamp01(currentFillRatio);

            if (audioSource.isPlaying)
                audioSource.Stop();
        }
    }

    public bool TryUnlock()
    {
        if (isJetpackUnlocked)
            return false;

        onUnlockJetpack.Invoke(true);
        isJetpackUnlocked = true;
        m_LastTimeOfUse = Time.time;
        return true;
    }
}
```

MeshCombiner.cs

```
using System.Collections.Generic;
using UnityEngine;

public class MeshCombiner : MonoBehaviour
{
    public List<GameObject> combineParents = new List<GameObject>();

    [Header("Grid parameters")]
    public bool useGrid = false;
    public Vector3 gridCenter;
    public Vector3 gridExtents = new Vector3(10, 10, 10);
    public Vector3Int gridResolution = new Vector3Int(2, 2, 2);
    public Color gridPreviewColor = Color.green;

    private void Start()
    {
        Combine();
    }
```

```
    public void Combine()
    {
        List<MeshRenderer> validRenderers = new List<MeshRenderer>();
        foreach (GameObject combineParent in combineParents)
        {
            validRenderers.AddRange(combineParent.GetComponentsInChildren<MeshRen-
derer>());
        }

        if (useGrid)
        {
            for (int i = 0; i < GetGridCellCount(); i++)
            {
                if (GetGridCellBounds(i, out Bounds bounds))
                {
                    CombineAllInBounds(bounds, validRenderers);
                }
            }
        }
        else
        {
            MeshCombineUtility.Combine(validRenderers, MeshCombineUtility.RendererDispose-
Method.DestroyRendererAndFilter, "Level_Combined");
        }
    }

    void CombineAllInBounds(Bounds bounds, List<MeshRenderer> validRenderers)
    {
        List<MeshRenderer> renderersForThisCell = new List<MeshRenderer>();

        for (int i = validRenderers.Count - 1; i >= 0; i--)
        {
            MeshRenderer m = validRenderers[i];
            if (bounds.Intersects(m.bounds))
            {
                renderersForThisCell.Add(m);
                validRenderers.Remove(m);
            }
        }

        if (renderersForThisCell.Count > 0)
        {
            MeshCombineUtility.Combine(renderersForThisCell, MeshCombineUtility.RendererDis-
poseMethod.DestroyRendererAndFilter, "Level_Combined");
```

```
        }
    }

    int GetGridCellCount()
    {
        return gridResolution.x * gridResolution.y * gridResolution.z;
    }

    public bool GetGridCellBounds(int index, out Bounds bounds)
    {
        // lo ocupo para que haya nuevas formas sólidas al moverse
        bounds = default;
        if (index < 0 || index >= GetGridCellCount())
            return false;

        int xCoord = index / (gridResolution.y * gridResolution.z);
        int yCoord = (index / gridResolution.z) % gridResolution.y;
        int zCoord = index % gridResolution.z;

        Vector3 gridBottomCorner = gridCenter - (gridExtents * 0.5f);
        Vector3 cellSize = new Vector3(gridExtents.x / (float)gridResolution.x, gridEx-
tents.y / (float)gridResolution.y, gridExtents.z / (float)gridResolution.z);
        Vector3 cellCenter = gridBottomCorner + (new Vec-
tor3((xCoord * cellSize.x) + (cellSize.x * 0.5f), (yCoord * cellSize.y) + (cellSize.y * 0.5f), (zCoor
d * cellSize.z) + (cellSize.z * 0.5f)));

        bounds.center = cellCenter;
        bounds.size = cellSize;

        return true;
    }

    private void OnDrawGizmosSelected()
    {
        if (useGrid)
        {
            Gizmos.color = gridPreviewColor;

            for (int i = 0; i < GetGridCellCount(); i++)
            {
                if(GetGridCellBounds(i, out Bounds bounds))
                {
                    Gizmos.DrawWireCube(bounds.center, bounds.size);
                }
            }
```

```
        }
    }
}
```

MeshCombinerUtility.cs

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.ProBuilder;
using UnityEngine.Rendering;

public static class MeshCombineUtility
{
    public class RenderBatchData
    {
        public class MeshAndTRS
        {
            public Mesh mesh;
            public Matrix4x4 trs;

            public MeshAndTRS(Mesh m, Matrix4x4 t)
            {
                mesh = m;
                trs = t;
            }
        }

        public Material material;
        public int submeshIndex = 0;
        public ShadowCastingMode shadowMode;
        public bool receiveShadows;
        public MotionVectorGenerationMode motionVectors;
        public List<MeshAndTRS> meshesWithTRS = new List<MeshAndTRS>();
    }

    public enum RendererDisposeMethod
    {
        DestroyGameObject,
        DestroyRendererAndFilter,
        DisableGameObject,
        DisableRenderer,
    }

    public static void Combine(List<MeshRenderer> renderers, RendererDisposeMethod dispo-
seMethod, string newObjectName)
```

```
    {
        int renderersCount = renderers.Count;

        List<RenderBatchData> renderBatches = new List<RenderBatchData>();

        // Build render batches for all unique material + submeshIndex combinations
        for (int i = 0; i < renderersCount; i++)
        {
            MeshRenderer meshRenderer = renderers[i];

            if (meshRenderer == null)
                continue;

            MeshFilter meshFilter = meshRenderer.GetComponent<MeshFilter>();

            if (meshFilter == null)
                continue;

            Mesh mesh = meshFilter.sharedMesh;

            if (mesh == null)
                continue;

            Transform t = meshRenderer.GetComponent<Transform>();
            Material[] materials = meshRenderer.sharedMaterials;

            for (int s = 0; s < mesh.subMeshCount; s++)
            {
                if (materials[s] == null)
                    continue;

                int batchIndex = GetExistingRenderBatch(renderBatches, materials[s], meshRen-
derer, s);
                if(batchIndex >= 0)
                {
                    renderBatches[batchIndex].meshesWithTRS.Add(new RenderBatchData.Me-
shAndTRS(mesh, Matrix4x4.TRS(t.position, t.rotation, t.lossyScale)));
                }
                else
                {
                    RenderBatchData newBatchData = new RenderBatchData();
                    newBatchData.material = materials[s];
                    newBatchData.submeshIndex = s;
                    newBatchData.shadowMode = meshRenderer.shadowCastingMode;
                    newBatchData.receiveShadows = meshRenderer.receiveShadows;
```

```
                newBatchData.meshesWithTRS.Add(new RenderBatchData.Me-
shAndTRS(mesh, Matrix4x4.TRS(t.position, t.rotation, t.lossyScale)));

                renderBatches.Add(newBatchData);
            }
        }

        // Destroy probuilder component if present
        ProBuilderMesh pbm = meshRenderer.GetComponent<ProBuilderMesh>();
        if(pbm)
        {
            GameObject.Destroy(pbm);
        }

        switch (disposeMethod)
        {
            case RendererDisposeMethod.DestroyGameObject:
                if (Application.isPlaying)
                {
                    GameObject.Destroy(meshRenderer.gameObject);
                }
                else
                {
                    GameObject.DestroyImmediate(meshRenderer.gameObject);
                }
                break;
            case RendererDisposeMethod.DestroyRendererAndFilter:
                if (Application.isPlaying)
                {
                    GameObject.Destroy(meshRenderer);
                    GameObject.Destroy(meshFilter);
                }
                else
                {
                    GameObject.DestroyImmediate(meshRenderer);
                    GameObject.DestroyImmediate(meshFilter);
                }
                break;
            case RendererDisposeMethod.DisableGameObject:
                meshRenderer.gameObject.SetActive(false);
                break;
            case RendererDisposeMethod.DisableRenderer:
                meshRenderer.enabled = false;
                break;
        }
```

```
        }

        // Combine each unique render batch
        for (int i = 0; i < renderBatches.Count; i++)
        {
            RenderBatchData rbd = renderBatches[i];

            Mesh newMesh = new Mesh();
            newMesh.indexFormat = UnityEngine.Rendering.IndexFormat.UInt32;
            CombineInstance[] combineInstances = new CombineInstance[rbd.mesh-
esWithTRS.Count];

            for (int j = 0; j < rbd.meshesWithTRS.Count; j++)
            {
                combineInstances[j].subMeshIndex = rbd.submeshIndex;
                combineInstances[j].mesh = rbd.meshesWithTRS[j].mesh;
                combineInstances[j].transform = rbd.meshesWithTRS[j].trs;
            }

            // Create mesh
            newMesh.CombineMeshes(combineInstances);
            newMesh.RecalculateBounds();

            // Create the gameObject
            GameObject combinedObject = new GameObject(newObjectName);
            MeshFilter mf = combinedObject.AddComponent<MeshFilter>();
            mf.sharedMesh = newMesh;
            MeshRenderer mr = combinedObject.AddComponent<MeshRenderer>();
            mr.sharedMaterial = rbd.material;
            mr.shadowCastingMode = rbd.shadowMode;
        }
    }

    private static int GetExistingRenderBatch(List<RenderBatchData> renderBatches, Mate-
rial mat, MeshRenderer ren, int submeshIndex)
    {
        for (int i = 0; i < renderBatches.Count; i++)
        {
            RenderBatchData data = renderBatches[i];
            if (data.material == mat &&
                data.submeshIndex == submeshIndex &&
                data.shadowMode == ren.shadowCastingMode &&
                data.receiveShadows == ren.receiveShadows)
            {
                return i;
```

```
            }
        }

        return -1;
    }
}
```

ObjectiveKillEnemies.cs

```
using UnityEngine;

[RequireComponent(typeof(Objective))]
public class ObjectiveKillEnemies : MonoBehaviour
{
    [Tooltip("Chose whether you need to kill every enemies or only a minimum amount")]
    public bool mustKillAllEnemies = true;
    [Tooltip("If MustKillAllEnemies is false, this is the amount of enemy kills required")]
    public int killsToCompleteObjective = 5;
    [Tooltip("Start sending notification about remaining enemies when this amount of ene-
mies is left")]
    public int notificationEnemiesRemainingThreshold = 3;

    EnemyManager m_EnemyManager;
    Objective m_Objective;
    int m_KillTotal;

    void Start()
    {
        m_Objective = GetComponent<Objective>();
        DebugUtility.HandleErrorIfNullGetComponent<Objective, ObjectiveKillEnemies>(m_Ob-
jective, this, gameObject);

        m_EnemyManager = FindObjectOfType<EnemyManager>();
        DebugUtility.HandleErrorIfNullFindObject<EnemyManager, ObjectiveKillEnemies>(m_En-
emyManager, this);
        m_EnemyManager.onRemoveEnemy += OnKillEnemy;

        if (mustKillAllEnemies)
            killsToCompleteObjective = m_EnemyManager.numberOfEnemiesTotal;


        // set a title and description specific for this type of objective, if it hasn't one
        if (string.IsNullOrEmpty(m_Objective.title))
            m_Objective.title = "Eliminate " + (mustKillAllEnemies ? "all the" : killsToComplete-
Objective.ToString()) + " enemies";
```

```
        if (string.IsNullOrEmpty(m_Objective.description))
            m_Objective.description = GetUpdatedCounterAmount();
    }

    void OnKillEnemy(EnemyController enemy, int remaining)
    {
        if (m_Objective.isCompleted)
            return;

        if (mustKillAllEnemies)
            killsToCompleteObjective = m_EnemyManager.numberOfEnemiesTotal;

        m_KillTotal = m_EnemyManager.numberOfEnemiesTotal - remaining;
        int targetRemaning = mustKillAllEnemies ? remaining : killsToCompleteObjec-
tive - m_KillTotal;

        // update the objective text according to how many enemies remain to kill
        if (targetRemaning == 0)
        {
            m_Objective.CompleteObjective(string.Empty, GetUpdatedCounterAmount(), "Objec-
tive complete : " + m_Objective.title);
        }
        else
        {

            if (targetRemaning == 1)
            {
                string notificationText = notificationEnemiesRemainingThreshold >= targetReman-
ing ? "One enemy left" : string.Empty;
                m_Objective.UpdateObjective(string.Empty, GetUpdatedCounterAmount(), notifica-
tionText);
            }
            else if (targetRemaning > 1)
            {
                // create a notification text if needed, if it stays empty, the notification will not be cre-
ated
                string notificationText = notificationEnemiesRemainingThreshold >= targetReman-
ing ? targetRemaning + " enemies to kill left" : string.Empty;

                m_Objective.UpdateObjective(string.Empty, GetUpdatedCounterAmount(), notifica-
tionText);
            }
        }
    }
```

```
    string GetUpdatedCounterAmount()
    {
        return m_KillTotal + " / " + killsToCompleteObjective;
    }

}
```

ObjectiveManager.cs

```
using System.Collections.Generic;
using UnityEngine;

public class ObjectiveManager : MonoBehaviour
{
    List<Objective> m_Objectives = new List<Objective>();

    public bool AreAllObjectivesCompleted()
    {
        if (m_Objectives.Count == 0)
            return false;

        for (int i = 0; i < m_Objectives.Count; i++)
        {
            // pass every objectives to check if they have been completed
            if (m_Objectives[i].isBlocking())
            {
                // break the loop as soon as we find one uncompleted objective
                return false;
            }
        }

        // found no uncompleted objective
        return true;
    }

    public void RegisterObjective(Objective objective)
    {
        m_Objectives.Add(objective);
    }
}
```

OverheatBehavior.cs

```
using UnityEngine;
using System.Collections.Generic;

public class OverheatBehavior : MonoBehaviour
{
    [System.Serializable]
    public struct RendererIndexData
    {
        public Renderer renderer;
        public int materialIndex;

        public RendererIndexData(Renderer renderer, int index)
        {
            this.renderer = renderer;
            this.materialIndex = index;
        }
    }

    [Header("Visual")]
    [Tooltip("The VFX to scale the spawn rate based on the ammo ratio")]
    public ParticleSystem steamVFX;
    [Tooltip("The emission rate for the effect when fully overheated")]
    public float steamVFXEmissionRateMax = 8f;

    //Set gradient field to HDR
    [GradientUsage(true)]
    [Tooltip("Overheat color based on ammo ratio")]
    public Gradient overheatGradient;
    [Tooltip("The material for overheating color animation")]
    public Material overheatingMaterial;

    [Header("Sound")]
    [Tooltip("Sound played when a cell are cooling")]
    public AudioClip coolingCellsSound;
    [Tooltip("Curve for ammo to volume ratio")]
    public AnimationCurve ammoToVolumeRatioCurve;


    WeaponController m_Weapon;
    AudioSource m_AudioSource;
    List<RendererIndexData> m_OverheatingRenderersData;
    MaterialPropertyBlock overheatMaterialPropertyBlock;
    float m_LastAmmoRatio;
    ParticleSystem.EmissionModule m_SteamVFXEmissionModule;
```

```
    void Awake()
    {
        var emissionModule = steamVFX.emission;
        emissionModule.rateOverTimeMultiplier = 0f;

        m_OverheatingRenderersData = new List<RendererIndexData>();
        foreach (var renderer in GetComponentsInChildren<Renderer>(true))
        {
            for (int i = 0; i < renderer.sharedMaterials.Length; i++)
            {
                if (renderer.sharedMaterials[i] == overheatingMaterial)
                    m_OverheatingRenderersData.Add(new RendererIndexData(renderer, i));
            }
        }

        overheatMaterialPropertyBlock = new MaterialPropertyBlock();
        m_SteamVFXEmissionModule = steamVFX.emission;

        m_Weapon = GetComponent<WeaponController>();
        DebugUtility.HandleErrorIfNullGetComponent<WeaponController, OverheatBehav-
ior>(m_Weapon, this, gameObject);

        m_AudioSource = gameObject.AddComponent<AudioSource>();
        m_AudioSource.clip = coolingCellsSound;
        m_AudioSource.outputAudioMixerGroup = AudioUtility.GetAudioGroup(AudioUtility.Audi-
oGroups.WeaponOverheat);
    }

    void Update()
    {
        // visual smoke shooting out of the gun
        float currentAmmoRatio = m_Weapon.currentAmmoRatio;
        if (currentAmmoRatio != m_LastAmmoRatio)
        {
            overheatMaterialPropertyBlock.SetColor("_EmissionColor", overheatGradient.Evalu-
ate(1f - currentAmmoRatio));

            foreach (var data in m_OverheatingRenderersData)
            {
                data.renderer.SetPropertyBlock(overheatMaterialPropertyBlock, data.materialIndex);
            }

            m_SteamVFXEmissionModule.rateOverTimeMultiplier = steamVFXEmissionRate-
Max * (1f - currentAmmoRatio);
```

```
        }

        // cooling sound
        if (coolingCellsSound)
        {
            if (!m_AudioSource.isPlaying
                && currentAmmoRatio != 1
                && m_Weapon.isWeaponActive
                && m_Weapon.isCooling)
            {
                m_AudioSource.Play();
            }
            else if (m_AudioSource.isPlaying
                && (currentAmmoRatio == 1 || !m_Weapon.isWeaponActive || !m_Weapon.isCool-
ing))
            {
                m_AudioSource.Stop();
                return;
            }

            m_AudioSource.volume = ammoToVolumeRatioCurve.Evaluate(1 - currentAmmoRa-
tio);
        }

        m_LastAmmoRatio = currentAmmoRatio;
    }
}
```

Pickup.cs

```
using UnityEngine;
using UnityEngine.Events;

[RequireComponent(typeof(Rigidbody), typeof(Collider))]
public class Pickup : MonoBehaviour
{
    [Tooltip("Frequency at which the item will move up and down")]
    public float verticalBobFrequency = 1f;
    [Tooltip("Distance the item will move up and down")]
    public float bobbingAmount = 1f;
    [Tooltip("Rotation angle per second")]
    public float rotatingSpeed = 360f;

    [Tooltip("Sound played on pickup")]
    public AudioClip pickupSFX;
```

```
    [Tooltip("VFX spawned on pickup")]
    public GameObject pickupVFXPrefab;

    public UnityAction<PlayerCharacterController> onPick;
    public Rigidbody pickupRigidbody { get; private set; }

    Collider m_Collider;
    Vector3 m_StartPosition;
    bool m_HasPlayedFeedback;

    private void Start()
    {
        pickupRigidbody = GetComponent<Rigidbody>();
        DebugUtility.HandleErrorIfNullGetComponent<Rigidbody, Pickup>(pickupRigid-
body, this, gameObject);
        m_Collider = GetComponent<Collider>();
        DebugUtility.HandleErrorIfNullGetComponent<Collider, Pickup>(m_Col-
lider, this, gameObject);

        // ensure the physics setup is a kinematic rigidbody trigger
        pickupRigidbody.isKinematic = true;
        m_Collider.isTrigger = true;

        // Remember start position for animation
        m_StartPosition = transform.position;
    }

    private void Update()
    {
        // Handle bobbing
        float bobbingAnimationPhase = ((Mathf.Sin(Time.time * verticalBobFre-
quency) * 0.5f) + 0.5f) * bobbingAmount;
        transform.position = m_StartPosition + Vector3.up * bobbingAnimationPhase;

        // Handle rotating
        transform.Rotate(Vector3.up, rotatingSpeed * Time.deltaTime, Space.Self);
    }

    private void OnTriggerEnter(Collider other)
    {
        PlayerCharacterController pickingPlayer = other.GetComponent<PlayerCharacterControl-
ler>();

        if (pickingPlayer != null)
        {
```

```
            if (onPick != null)
            {
                onPick.Invoke(pickingPlayer);
            }
        }
    }

    public void PlayPickupFeedback()
    {
        if (m_HasPlayedFeedback)
            return;

        if (pickupSFX)
        {
            AudioUtility.CreateSFX(pickupSFX, transform.position, AudioUtility.Audi-
oGroups.Pickup, 0f);
        }

        if (pickupVFXPrefab)
        {
            var pickupVFXInstance = Instantiate(pickupVFXPrefab, transform.position, Quater-
nion.identity);
        }

        m_HasPlayedFeedback = true;
    }
}
```

PlayerCharacterController.cs

```
using UnityEngine;
using UnityEngine.Events;

[RequireComponent(typeof(CharacterController), typeof(PlayerInputHandler), typeof(Audi-
oSource))]
public class PlayerCharacterController : MonoBehaviour
{
    [Header("References")]
    [Tooltip("Reference to the main camera used for the player")]
    public Camera playerCamera;
    [Tooltip("Audio source for footsteps, jump, etc...")]
    public AudioSource audioSource;

    [Header("General")]
    [Tooltip("Force applied downward when in the air")]
```

```
    public float gravityDownForce = 20f;
    [Tooltip("Physic layers checked to consider the player grounded")]
    public LayerMask groundCheckLayers = -1;
    [Tooltip("distance from the bottom of the character controller capsule to test for grounded")]
    public float groundCheckDistance = 0.05f;

    [Header("Movement")]
    [Tooltip("Max movement speed when grounded (when not sprinting)")]
    public float maxSpeedOnGround = 10f;
    [Tooltip("Sharpness for the movement when grounded, a low value will make the player ac-
celerate and decelerate slowly, a high value will do the opposite")]
    public float movementSharpnessOnGround = 15;
    [Tooltip("Max movement speed when crouching")]
    [Range(0,1)]
    public float maxSpeedCrouchedRatio = 0.5f;
    [Tooltip("Max movement speed when not grounded")]
    public float maxSpeedInAir = 10f;
    [Tooltip("Acceleration speed when in the air")]
    public float accelerationSpeedInAir = 25f;
    [Tooltip("Multiplicator for the sprint speed (based on grounded speed)")]
    public float sprintSpeedModifier = 2f;
    [Tooltip("Height at which the player dies instantly when falling off the map")]
    public float killHeight = -50f;

    [Header("Rotation")]
    [Tooltip("Rotation speed for moving the camera")]
    public float rotationSpeed = 200f;
    [Range(0.1f, 1f)]
    [Tooltip("Rotation speed multiplier when aiming")]
    public float aimingRotationMultiplier = 0.4f;

    [Header("Jump")]
    [Tooltip("Force applied upward when jumping")]
    public float jumpForce = 9f;

    [Header("Stance")]
    [Tooltip("Ratio (0-1) of the character height where the camera will be at")]
    public float cameraHeightRatio = 0.9f;
    [Tooltip("Height of character when standing")]
    public float capsuleHeightStanding = 1.8f;
    [Tooltip("Height of character when crouching")]
    public float capsuleHeightCrouching = 0.9f;
    [Tooltip("Speed of crouching transitions")]
    public float crouchingSharpness = 10f;
```

```
    [Header("Audio")]
    [Tooltip("Amount of footstep sounds played when moving one meter")]
    public float footstepSFXFrequency = 1f;
    [Tooltip("Amount of footstep sounds played when moving one meter while sprinting")]
    public float footstepSFXFrequencyWhileSprinting = 1f;
    [Tooltip("Sound played for footsteps")]
    public AudioClip footstepSFX;
    [Tooltip("Sound played when jumping")]
    public AudioClip jumpSFX;
    [Tooltip("Sound played when landing")]
    public AudioClip landSFX;
    [Tooltip("Sound played when taking damage froma fall")]
    public AudioClip fallDamageSFX;

    [Header("Fall Damage")]
    [Tooltip("Whether the player will recieve damage when hitting the ground at high speed")]
    public bool recievesFallDamage;
    [Tooltip("Minimun fall speed for recieving fall damage")]
    public float minSpeedForFallDamage = 10f;
    [Tooltip("Fall speed for recieving th emaximum amount of fall damage")]
    public float maxSpeedForFallDamage = 30f;
    [Tooltip("Damage recieved when falling at the mimimum speed")]
    public float fallDamageAtMinSpeed = 10f;
    [Tooltip("Damage recieved when falling at the maximum speed")]
    public float fallDamageAtMaxSpeed = 50f;

    public UnityAction<bool> onStanceChanged;

    public Vector3 characterVelocity { get; set; }
    public bool isGrounded { get; private set; }
    public bool hasJumpedThisFrame { get; private set; }
    public bool isDead { get; private set; }
    public bool isCrouching { get; private set; }
    public float RotationMultiplier
    {
        get
        {
            if (m_WeaponsManager.isAiming)
            {
                return aimingRotationMultiplier;
            }

            return 1f;
        }
    }
```

```
    Health m_Health;
    PlayerInputHandler m_InputHandler;
    CharacterController m_Controller;
    PlayerWeaponsManager m_WeaponsManager;
    Actor m_Actor;
    Vector3 m_GroundNormal;
    Vector3 m_CharacterVelocity;
    Vector3 m_LatestImpactSpeed;
    float m_LastTimeJumped = 0f;
    float m_CameraVerticalAngle = 0f;
    float m_footstepDistanceCounter;
    float m_TargetCharacterHeight;

    const float k_JumpGroundingPreventionTime = 0.2f;
    const float k_GroundCheckDistanceInAir = 0.07f;

    void Start()
    {
        // fetch components on the same gameObject
        m_Controller = GetComponent<CharacterController>();
        DebugUtility.HandleErrorIfNullGetComponent<CharacterController, PlayerCharacterCon-
troller>(m_Controller, this, gameObject);

        m_InputHandler = GetComponent<PlayerInputHandler>();
        DebugUtility.HandleErrorIfNullGetComponent<PlayerInputHandler, PlayerCharacterCon-
troller>(m_InputHandler, this, gameObject);

        m_WeaponsManager = GetComponent<PlayerWeaponsManager>();
        DebugUtility.HandleErrorIfNullGetComponent<PlayerWeaponsManager, PlayerCharac-
terController>(m_WeaponsManager, this, gameObject);

        m_Health = GetComponent<Health>();
        DebugUtility.HandleErrorIfNullGetComponent<Health, PlayerCharacterControl-
ler>(m_Health, this, gameObject);

        m_Actor = GetComponent<Actor>();
        DebugUtility.HandleErrorIfNullGetComponent<Actor, PlayerCharacterController>(m_Ac-
tor, this, gameObject);

        m_Controller.enableOverlapRecovery = true;

        m_Health.onDie += OnDie;

        // force the crouch state to false when starting
```

```
        SetCrouchingState(false, true);
        UpdateCharacterHeight(true);
    }

    void Update()
    {
        // check for Y kill
        if(!isDead && transform.position.y < killHeight)
        {
            m_Health.Kill();
        }

        hasJumpedThisFrame = false;

        bool wasGrounded = isGrounded;
        GroundCheck();

        // landing
        if (isGrounded && !wasGrounded)
        {
            // Fall damage
            float fallSpeed = -Mathf.Min(characterVelocity.y, m_LatestImpactSpeed.y);
            float fallSpeedRatio = (fallSpeed - minSpeedForFallDamage) / (maxSpeedForFall-
Damage - minSpeedForFallDamage);
            if (recievesFallDamage && fallSpeedRatio > 0f)
            {
                float dmgFromFall = Mathf.Lerp(fallDamageAtMinSpeed, fall-
DamageAtMaxSpeed, fallSpeedRatio);
                m_Health.TakeDamage(dmgFromFall, null);

                // fall damage SFX
                audioSource.PlayOneShot(fallDamageSFX);
            }
            else
            {
                // land SFX
                audioSource.PlayOneShot(landSFX);
            }
        }

        // crouching
        if (m_InputHandler.GetCrouchInputDown())
        {
            SetCrouchingState(!isCrouching, false);
        }
```

```
        UpdateCharacterHeight(false);

        HandleCharacterMovement();
    }

    void OnDie()
    {
        isDead = true;

        // Tell the weapons manager to switch to a non-existing weapon in or-
der to lower the weapon
        m_WeaponsManager.SwitchToWeaponIndex(-1, true);
    }

    void GroundCheck()
    {
        // Make sure that the ground check distance while already in air is very small, to pre-
vent suddenly snapping to ground
        float chosenGroundCheckDistance = isGrounded ? (m_Controller.skinWidth + ground-
CheckDistance) : k_GroundCheckDistanceInAir;

        // reset values before the ground check
        isGrounded = false;
        m_GroundNormal = Vector3.up;

        // only try to detect ground if it's been a short amount of time since last jump; other-
wise we may snap to the ground instantly after we try jumping
        if (Time.time >= m_LastTimeJumped + k_JumpGroundingPreventionTime)
        {
            // if we're grounded, collect info about the ground normal with a downward cap-
sule cast representing our character capsule
            if (Physics.CapsuleCast(GetCapsuleBottomHemisphere(), GetCapsuleTopHe-
misphere(m_Controller.height), m_Controller.radius, Vector3.down, out RaycastHit hit, chosen-
GroundCheckDistance, groundCheckLayers, QueryTriggerInteraction.Ignore))
            {
                // storing the upward direction for the surface found
                m_GroundNormal = hit.normal;

                // Only consider this a valid ground hit if the ground normal goes in the same direc-
tion as the character up
                // and if the slope angle is lower than the character controller's limit
                if (Vector3.Dot(hit.normal, transform.up) > 0f &&
                    IsNormalUnderSlopeLimit(m_GroundNormal))
                {
```

```
                isGrounded = true;

                // handle snapping to the ground
                if (hit.distance > m_Controller.skinWidth)
                {
                    m_Controller.Move(Vector3.down * hit.distance);
                }
            }
        }
    }

    void HandleCharacterMovement()
    {
        // horizontal character rotation
        {
            // rotate the transform with the input speed around its local Y axis
            transform.Rotate(new Vector3(0f, (m_InputHandler.GetLookInputsHorizontal() * rotationSpeed * RotationMultiplier), 0f), Space.Self);
        }

        // vertical camera rotation
        {
            // add vertical inputs to the camera's vertical angle
            m_CameraVerticalAngle += m_InputHandler.GetLookInputsVertical() * rotationSpeed * RotationMultiplier;

            // limit the camera's vertical angle to min/max
            m_CameraVerticalAngle = Mathf.Clamp(m_CameraVerticalAngle, -89f, 89f);

            // apply the vertical angle as a local rotation to the camera transform along its right axis (makes it pivot up and down)
            playerCamera.transform.localEulerAngles = new Vector3(m_CameraVerticalAngle, 0, 0);
        }

        // character movement handling
        bool isSprinting = m_InputHandler.GetSprintInputHeld();
        {
            if (isSprinting)
            {
                isSprinting = SetCrouchingState(false, false);
            }

            float speedModifier = isSprinting ? sprintSpeedModifier : 1f;
```

```
        // converts move input to a worldspace vector based on our character's transform orientation
        Vector3 worldspaceMoveInput = transform.TransformVector(m_InputHandler.GetMoveInput());

        // handle grounded movement
        if (isGrounded)
        {
            // calculate the desired velocity from inputs, max speed, and current slope
            Vector3 targetVelocity = worldspaceMoveInput * maxSpeedOnGround * speedModifier;
            // reduce speed if crouching by crouch speed ratio
            if (isCrouching)
                targetVelocity *= maxSpeedCrouchedRatio;
            targetVelocity = GetDirectionReorientedOnSlope(targetVelocity.normalized, m_GroundNormal) * targetVelocity.magnitude;

            // smoothly interpolate between our current velocity and the target velocity based on acceleration speed
            characterVelocity = Vector3.Lerp(characterVelocity, targetVelocity, movementSharpnessOnGround * Time.deltaTime);

            // jumping
            if (isGrounded && m_InputHandler.GetJumpInputDown())
            {
                // force the crouch state to false
                if (SetCrouchingState(false, false))
                {
                    // start by canceling out the vertical component of our velocity
                    characterVelocity = new Vector3(characterVelocity.x, 0f, characterVelocity.z);

                    // then, add the jumpSpeed value upwards
                    characterVelocity += Vector3.up * jumpForce;

                    // play sound
                    audioSource.PlayOneShot(jumpSFX);

                    // remember last time we jumped because we need to prevent snapping to ground for a short time
                    m_LastTimeJumped = Time.time;
                    hasJumpedThisFrame = true;

                    // Force grounding to false
                    isGrounded = false;
```

```
                    m_GroundNormal = Vector3.up;
                }
            }

            // footsteps sound
            float chosenFootstepSFXFrequency = (isSprinting ? footstepSFXFrequency-
WhileSprinting : footstepSFXFrequency);
            if (m_footstepDistanceCounter >= 1f / chosenFootstepSFXFrequency)
            {
                m_footstepDistanceCounter = 0f;
                audioSource.PlayOneShot(footstepSFX);
            }

            // keep track of distance traveled for footsteps sound
            m_footstepDistanceCounter += characterVelocity.magnitude * Time.deltaTime;
        }
        // handle air movement
        else
        {
            // add air acceleration
            characterVelocity += worldspaceMoveInput * accelerationSpeedInAir * Time.del-
taTime;

            // limit air speed to a maximum, but only horizontally
            float verticalVelocity = characterVelocity.y;
            Vector3 horizontalVelocity = Vector3.ProjectOnPlane(characterVelocity, Vector3.up);
            horizontalVelocity = Vector3.ClampMagnitude(horizontalVelocity, maxSpeedI-
nAir * speedModifier);
            characterVelocity = horizontalVelocity + (Vector3.up * verticalVelocity);

            // apply the gravity to the velocity
            characterVelocity += Vector3.down * gravityDownForce * Time.deltaTime;
        }
    }

    // apply the final calculated velocity value as a character movement
    Vector3 capsuleBottomBeforeMove = GetCapsuleBottomHemisphere();
    Vector3 capsuleTopBeforeMove = GetCapsuleTopHemisphere(m_Controller.height);
    m_Controller.Move(characterVelocity * Time.deltaTime);

    // detect obstructions to adjust velocity accordingly
    m_LatestImpactSpeed = Vector3.zero;
    if (Physics.CapsuleCast(capsuleBottomBeforeMove, capsuleTopBeforeMove, m_Control-
ler.radius, characterVelocity.normalized, out RaycastHit hit, characterVelocity.magni-
tude * Time.deltaTime, -1, QueryTriggerInteraction.Ignore))
```

```
        {
            // We remember the last impact speed because the fall damage logic might need it
            m_LatestImpactSpeed = characterVelocity;

            characterVelocity = Vector3.ProjectOnPlane(characterVelocity, hit.normal);
        }
    }

    // Returns true if the slope angle represented by the given normal is under the slope an-
gle limit of the character controller
    bool IsNormalUnderSlopeLimit(Vector3 normal)
    {
        return Vector3.Angle(transform.up, normal) <= m_Controller.slopeLimit;
    }

    // Gets the center point of the bottom hemisphere of the character controller capsule
    Vector3 GetCapsuleBottomHemisphere()
    {
        return transform.position + (transform.up * m_Controller.radius);
    }

    // Gets the center point of the top hemisphere of the character controller capsule
    Vector3 GetCapsuleTopHemisphere(float atHeight)
    {
        return transform.position + (transform.up * (atHeight - m_Controller.radius));
    }

    // Gets a reoriented direction that is tangent to a given slope
    public Vector3 GetDirectionReorientedOnSlope(Vector3 direction, Vector3 slopeNormal)
    {
        Vector3 directionRight = Vector3.Cross(direction, transform.up);
        return Vector3.Cross(slopeNormal, directionRight).normalized;
    }

    void UpdateCharacterHeight(bool force)
    {
        // Update height instantly
        if (force)
        {
            m_Controller.height = m_TargetCharacterHeight;
            m_Controller.center = Vector3.up * m_Controller.height * 0.5f;
            playerCamera.transform.localPosition = Vector3.up * m_TargetCharacterHeight * cam-
eraHeightRatio;
            m_Actor.aimPoint.transform.localPosition = m_Controller.center;
        }
```

```
        // Update smooth height
        else if (m_Controller.height != m_TargetCharacterHeight)
        {
            // resize the capsule and adjust camera position
            m_Controller.height = Mathf.Lerp(m_Controller.height, m_TargetCharacter-
Height, crouchingSharpness * Time.deltaTime);
            m_Controller.center = Vector3.up * m_Controller.height * 0.5f;
            playerCamera.transform.localPosition = Vector3.Lerp(playerCamera.transform.localPo-
sition, Vector3.up * m_TargetCharacterHeight * cameraHeightRatio, crouchingSharp-
ness * Time.deltaTime);
            m_Actor.aimPoint.transform.localPosition = m_Controller.center;
        }
    }

    // returns false if there was an obstruction
    bool SetCrouchingState(bool crouched, bool ignoreObstructions)
    {
        // set appropriate heights
        if (crouched)
        {
            m_TargetCharacterHeight = capsuleHeightCrouching;
        }
        else
        {
            // Detect obstructions
            if (!ignoreObstructions)
            {
                Collider[] standingOverlaps = Physics.OverlapCapsule(
                    GetCapsuleBottomHemisphere(),
                    GetCapsuleTopHemisphere(capsuleHeightStanding),
                    m_Controller.radius,
                    -1,
                    QueryTriggerInteraction.Ignore);
                foreach (Collider c in standingOverlaps)
                {
                    if (c != m_Controller)
                    {
                        return false;
                    }
                }
            }

            m_TargetCharacterHeight = capsuleHeightStanding;
        }
```

```
        if (onStanceChanged != null)
        {
            onStanceChanged.Invoke(crouched);
        }

        isCrouching = crouched;
        return true;
    }
}
```

PlayerInputHandler.cs

```
using UnityEngine;

public class PlayerInputHandler : MonoBehaviour
{
    [Tooltip("Sensitivity multiplier for moving the camera around")]
    public float lookSensitivity = 1f;
    [Tooltip("Additional sensitivity multiplier for WebGL")]
    public float webglLookSensitivityMultiplier = 0.25f;
    [Tooltip("Limit to consider an input when using a trigger on a controller")]
    public float triggerAxisThreshold = 0.4f;
    [Tooltip("Used to flip the vertical input axis")]
    public bool invertYAxis = false;
    [Tooltip("Used to flip the horizontal input axis")]
    public bool invertXAxis = false;

    GameFlowManager m_GameFlowManager;
    PlayerCharacterController m_PlayerCharacterController;
    bool m_FireInputWasHeld;

    private void Start()
    {
        m_PlayerCharacterController = GetComponent<PlayerCharacterController>();
        DebugUtility.HandleErrorIfNullGetComponent<PlayerCharacterController, PlayerIn-
putHandler>(m_PlayerCharacterController, this, gameObject);
        m_GameFlowManager = FindObjectOfType<GameFlowManager>();
        DebugUtility.HandleErrorIfNullFindObject<GameFlowManager, PlayerIn-
putHandler>(m_GameFlowManager, this);

        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }

    private void LateUpdate()
```

```
    {
        m_FireInputWasHeld = GetFireInputHeld();
    }

    public bool CanProcessInput()
    {
        return Cursor.lockState == CursorLockMode.Locked && !m_GameFlowMan-
ager.gameIsEnding;
    }

    public Vector3 GetMoveInput()
    {
        if (CanProcessInput())
        {
            Vector3 move = new Vector3(Input.GetAxisRaw(GameConstants.k_AxisNameHorizon-
tal), 0f, Input.GetAxisRaw(GameConstants.k_AxisNameVertical));

            // constrain move input to a maximum magnitude of 1, otherwise diagonal move-
ment might exceed the max move speed defined
            move = Vector3.ClampMagnitude(move, 1);

            return move;
        }

        return Vector3.zero;
    }

    public float GetLookInputsHorizontal()
    {
        return GetMouseOrStickLookAxis(GameConstants.k_MouseAxisNameHorizontal, Game-
Constants.k_AxisNameJoystickLookHorizontal);
    }

    public float GetLookInputsVertical()
    {
        return GetMouseOrStickLookAxis(GameConstants.k_MouseAxisNameVertical, Game-
Constants.k_AxisNameJoystickLookVertical);
    }

    public bool GetJumpInputDown()
    {
        if (CanProcessInput())
        {
            return Input.GetButtonDown(GameConstants.k_ButtonNameJump);
        }
```

```
        return false;
    }

    public bool GetJumpInputHeld()
    {
        if (CanProcessInput())
        {
            return Input.GetButton(GameConstants.k_ButtonNameJump);
        }

        return false;
    }

    public bool GetFireInputDown()
    {
        return GetFireInputHeld() && !m_FireInputWasHeld;
    }

    public bool GetFireInputReleased()
    {
        return !GetFireInputHeld() && m_FireInputWasHeld;
    }

    public bool GetFireInputHeld()
    {
        if (CanProcessInput())
        {
            bool isGamepad = Input.GetAxis(GameConstants.k_ButtonNameGamepadFire) != 0f;
            if (isGamepad)
            {
                return Input.GetAxis(GameConstants.k_ButtonNameGamepadFire) >= triggerAx-
isThreshold;
            }
            else
            {
                return Input.GetButton(GameConstants.k_ButtonNameFire);
            }
        }

        return false;
    }

    public bool GetAimInputHeld()
    {
```

```
        if (CanProcessInput())
        {
            bool isGamepad = Input.GetAxis(GameConstants.k_ButtonNameGamepadAim) != 0f;
            bool i = isGamepad ? (Input.GetAxis(GameConstants.k_ButtonNameGamepa-
dAim) > 0f) : Input.GetButton(GameConstants.k_ButtonNameAim);
            return i;
        }

        return false;
    }

    public bool GetSprintInputHeld()
    {
        if (CanProcessInput())
        {
            return Input.GetButton(GameConstants.k_ButtonNameSprint);
        }

        return false;
    }

    public bool GetCrouchInputDown()
    {
        if (CanProcessInput())
        {
            return Input.GetButtonDown(GameConstants.k_ButtonNameCrouch);
        }

        return false;
    }

    public bool GetCrouchInputReleased()
    {
        if (CanProcessInput())
        {
            return Input.GetButtonUp(GameConstants.k_ButtonNameCrouch);
        }

        return false;
    }

    public int GetSwitchWeaponInput()
    {
        if (CanProcessInput())
        {
```

```
        bool isGamepad = Input.GetAxis(GameConstants.k_ButtonNameGamepadSwitch-
Weapon) != 0f;
        string axisName = isGamepad ? GameConstants.k_ButtonNameGamepadSwitch-
Weapon : GameConstants.k_ButtonNameSwitchWeapon;

        if (Input.GetAxis(axisName) > 0f)
            return -1;
        else if (Input.GetAxis(axisName) < 0f)
            return 1;
        else if (Input.GetAxis(GameConstants.k_ButtonNameNextWeapon) > 0f)
            return -1;
        else if (Input.GetAxis(GameConstants.k_ButtonNameNextWeapon) < 0f)
            return 1;
    }

    return 0;
  }

  public int GetSelectWeaponInput()
  {
    if (CanProcessInput())
    {
        if (Input.GetKeyDown(KeyCode.Alpha1))
            return 1;
        else if (Input.GetKeyDown(KeyCode.Alpha2))
            return 2;
        else if (Input.GetKeyDown(KeyCode.Alpha3))
            return 3;
        else if (Input.GetKeyDown(KeyCode.Alpha4))
            return 4;
        else if (Input.GetKeyDown(KeyCode.Alpha5))
            return 5;
        else if (Input.GetKeyDown(KeyCode.Alpha6))
            return 6;
        else
            return 0;
    }

    return 0;
  }

  float GetMouseOrStickLookAxis(string mouseInputName, string stickInputName)
  {
    if (CanProcessInput())
```

```
        {
            // Check if this look input is coming from the mouse
            bool isGamepad = Input.GetAxis(stickInputName) != 0f;
            float i = isGamepad ? Input.GetAxis(stickInputName) : Input.GetAxisRaw(mouseInput-
Name);

            // handle inverting vertical input
            if (invertYAxis)
                i *= -1f;

            // apply sensitivity multiplier
            i *= lookSensitivity;

            if (isGamepad)
            {
                // since mouse input is already deltaTime-dependant, only scale in-
put with frame time if it's coming from sticks
                i *= Time.deltaTime;
            }
            else
            {
                // reduce mouse input amount to be equivalent to stick movement
                i *= 0.01f;
#if UNITY_WEBGL
                // Mouse tends to be even more sensitive in WebGL due to mouse accelera-
tion, so reduce it even more
                i *= webglLookSensitivityMultiplier;
#endif
            }

            return i;
        }

        return 0f;
    }
}
```

PlayerWeaponsManager.cs

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

[RequireComponent(typeof(PlayerInputHandler))]
public class PlayerWeaponsManager : MonoBehaviour
```

```
{
    public enum WeaponSwitchState
    {
        Up,
        Down,
        PutDownPrevious,
        PutUpNew,
    }

    [Tooltip("List of weapon the player will start with")]
    public List<WeaponController> startingWeapons = new List<WeaponController>();

    [Header("References")]
    [Tooltip("Secondary camera used to avoid seeing weapon go throw geometries")]
    public Camera weaponCamera;
    [Tooltip("Parent transform where all weapon will be added in the hierarchy")]
    public Transform weaponParentSocket;
    [Tooltip("Position for weapons when active but not actively aiming")]
    public Transform defaultWeaponPosition;
    [Tooltip("Position for weapons when aiming")]
    public Transform aimingWeaponPosition;
    [Tooltip("Position for innactive weapons")]
    public Transform downWeaponPosition;

    [Header("Weapon Bob")]
    [Tooltip("Fre-
quency at which the weapon will move around in the screen when the player is in movement")]
    public float bobFrequency = 10f;
    [Tooltip("How fast the weapon bob is applied, the bigger value the fastest")]
    public float bobSharpness = 10f;
    [Tooltip("Distance the weapon bobs when not aiming")]
    public float defaultBobAmount = 0.05f;
    [Tooltip("Distance the weapon bobs when aiming")]
    public float aimingBobAmount = 0.02f;

    [Header("Weapon Recoil")]
    [Tooltip("This will affect how fast the recoil moves the weapon, the bigger the value, the fast-
est")]
    public float recoilSharpness = 50f;
    [Tooltip("Maximum distance the recoil can affect the weapon")]
    public float maxRecoilDistance = 0.5f;
    [Tooltip("How fast the weapon goes back to it's original position after the recoil is finished")]
    public float recoilRestitutionSharpness = 10f;

    [Header("Misc")]
```

```
    [Tooltip("Speed at which the aiming animatoin is played")]
    public float aimingAnimationSpeed = 10f;
    [Tooltip("Field of view when not aiming")]
    public float defaultFOV = 60f;
    [Tooltip("Portion of the regular FOV to apply to the weapon camera")]
    public float weaponFOVMultiplier = 1f;
    [Tooltip("Delay before switching weapon a second time, to avoid recieving multiple inputs from mouse wheel")]
    public float weaponSwitchDelay = 1f;
    [Tooltip("Layer to set FPS weapon gameObjects to")]
    public LayerMask FPSWeaponLayer;

    public bool isAiming { get; private set; }
    public bool isPointingAtEnemy { get; private set; }
    public int activeWeaponIndex { get; private set; }

    public UnityAction<WeaponController> onSwitchedToWeapon;
    public UnityAction<WeaponController, int> onAddedWeapon;
    public UnityAction<WeaponController, int> onRemovedWeapon;

    WeaponController[] m_WeaponSlots = new WeaponController[9]; // 9 available weapon slots
    PlayerInputHandler m_InputHandler;
    PlayerCharacterController m_PlayerCharacterController;
    float m_WeaponBobFactor;
    Vector3 m_LastCharacterPosition;
    Vector3 m_WeaponMainLocalPosition;
    Vector3 m_WeaponBobLocalPosition;
    Vector3 m_WeaponRecoilLocalPosition;
    Vector3 m_AccumulatedRecoil;
    float m_TimeStartedWeaponSwitch;
    WeaponSwitchState m_WeaponSwitchState;
    int m_WeaponSwitchNewWeaponIndex;

    private void Start()
    {
        activeWeaponIndex = -1;
        m_WeaponSwitchState = WeaponSwitchState.Down;

        m_InputHandler = GetComponent<PlayerInputHandler>();
        DebugUtility.HandleErrorIfNullGetComponent<PlayerInputHandler, PlayerWeaponsManager>(m_InputHandler, this, gameObject);

        m_PlayerCharacterController = GetComponent<PlayerCharacterController>();
```

```
        DebugUtility.HandleErrorIfNullGetComponent<PlayerCharacterController, PlayerWeap-
onsManager>(m_PlayerCharacterController, this, gameObject);

        SetFOV(defaultFOV);

        onSwitchedToWeapon += OnWeaponSwitched;

        // Add starting weapons
        foreach (var weapon in startingWeapons)
        {
            AddWeapon(weapon);
        }
        SwitchWeapon(true);
    }

    private void Update()
    {
        // shoot handling
        WeaponController activeWeapon = GetActiveWeapon();

        if (activeWeapon && m_WeaponSwitchState == WeaponSwitchState.Up)
        {
            // handle aiming down sights
            isAiming = m_InputHandler.GetAimInputHeld();

            // handle shooting
            bool hasFired = activeWeapon.HandleShootInputs(
                m_InputHandler.GetFireInputDown(),
                m_InputHandler.GetFireInputHeld(),
                m_InputHandler.GetFireInputReleased());

            // Handle accumulating recoil
            if (hasFired)
            {
                m_AccumulatedRecoil += Vector3.back * activeWeapon.recoilForce;
                m_AccumulatedRecoil = Vector3.ClampMagnitude(m_AccumulatedRecoil, maxRe-
coilDistance);
            }
        }

        // weapon switch handling
        if (!isAiming &&
            (activeWeapon == null || !activeWeapon.isCharging) &&
            (m_WeaponSwitchState == WeaponSwitchState.Up || m_Weap-
onSwitchState == WeaponSwitchState.Down))
```

```
        {
            int switchWeaponInput = m_InputHandler.GetSwitchWeaponInput();
            if (switchWeaponInput != 0)
            {
                bool switchUp = switchWeaponInput > 0;
                SwitchWeapon(switchUp);
            }
            else
            {
                switchWeaponInput = m_InputHandler.GetSelectWeaponInput();
                if (switchWeaponInput != 0)
                {
                    if (GetWeaponAtSlotIndex(switchWeaponInput - 1) != null)
                        SwitchToWeaponIndex(switchWeaponInput - 1);
                }
            }
        }

        // Pointing at enemy handling
        isPointingAtEnemy = false;
        if (activeWeapon)
        {
            if(Physics.Raycast(weaponCamera.transform.position, weaponCamera.transform.for-
ward, out RaycastHit hit, 1000, -1, QueryTriggerInteraction.Ignore))
            {
                if(hit.collider.GetComponentInParent<EnemyController>())
                {
                    isPointingAtEnemy = true;
                }
            }
        }
    }


    // Update various animated features in LateUpdate because it needs to override the ani-
mated arm position
    private void LateUpdate()
    {
        UpdateWeaponAiming();
        UpdateWeaponBob();
        UpdateWeaponRecoil();
        UpdateWeaponSwitching();

        // Set final weapon socket position based on all the combined animation influences
```

```
        weaponParentSocket.localPosition = m_WeaponMainLocalPosition + m_WeaponBobLocalPosition + m_WeaponRecoilLocalPosition;
    }

    // Sets the FOV of the main camera and the weapon camera simultaneously
    public void SetFOV(float fov)
    {
        m_PlayerCharacterController.playerCamera.fieldOfView = fov;
        weaponCamera.fieldOfView = fov * weaponFOVMultiplier;
    }

    // Iterate on all weapon slots to find the next valid weapon to switch to
    public void SwitchWeapon(bool ascendingOrder)
    {
        int newWeaponIndex = -1;
        int closestSlotDistance = m_WeaponSlots.Length;
        for (int i = 0; i < m_WeaponSlots.Length; i++)
        {
            // If the weapon at this slot is valid, calculate its "distance" from the active slot index (either in ascending or descending order)
            // and select it if it's the closest distance yet
            if (i != activeWeaponIndex && GetWeaponAtSlotIndex(i) != null)
            {
                int distanceToActiveIndex = GetDistanceBetweenWeaponSlots(activeWeaponIndex, i, ascendingOrder);

                if (distanceToActiveIndex < closestSlotDistance)
                {
                    closestSlotDistance = distanceToActiveIndex;
                    newWeaponIndex = i;
                }
            }
        }

        // Handle switching to the new weapon index
        SwitchToWeaponIndex(newWeaponIndex);
    }

    // Switches to the given weapon index in weapon slots if the new index is a valid weapon that is different from our current one
    public void SwitchToWeaponIndex(int newWeaponIndex, bool force = false)
    {
        if (force || (newWeaponIndex != activeWeaponIndex && newWeaponIndex >= 0))
        {
            // Store data related to weapon switching animation
```

```
            m_WeaponSwitchNewWeaponIndex = newWeaponIndex;
            m_TimeStartedWeaponSwitch = Time.time;

            // Handle case of switching to a valid weapon for the first time (simply put it up with-
out putting anything down first)
            if(GetActiveWeapon() == null)
            {
                m_WeaponMainLocalPosition = downWeaponPosition.localPosition;
                m_WeaponSwitchState = WeaponSwitchState.PutUpNew;
                activeWeaponIndex = m_WeaponSwitchNewWeaponIndex;

                WeaponController newWeapon = GetWeaponAtSlotIndex(m_WeaponSwitchNew-
WeaponIndex);
                if (onSwitchedToWeapon != null)
                {
                    onSwitchedToWeapon.Invoke(newWeapon);
                }
            }
            // otherwise, remember we are putting down our current weapon for switch-
ing to the next one
            else
            {
                m_WeaponSwitchState = WeaponSwitchState.PutDownPrevious;
            }
        }
    }

    public bool HasWeapon(WeaponController weaponPrefab)
    {
        // Checks if we already have a weapon coming from the specified prefab
        foreach(var w in m_WeaponSlots)
        {
            if(w != null && w.sourcePrefab == weaponPrefab.gameObject)
            {
                return true;
            }
        }

        return false;
    }

    // Updates weapon position and camera FoV for the aiming transition
    void UpdateWeaponAiming()
    {
        if (m_WeaponSwitchState == WeaponSwitchState.Up)
```

```
        {
            WeaponController activeWeapon = GetActiveWeapon();
            if (isAiming && activeWeapon)
            {
                m_WeaponMainLocalPosition = Vector3.Lerp(m_WeaponMainLocalPosition, aim-
ingWeaponPosition.localPosition + activeWeapon.aimOffset, aimingAnimation-
Speed * Time.deltaTime);
                SetFOV(Mathf.Lerp(m_PlayerCharacterController.playerCamera.fieldOfView, active-
Weapon.aimZoomRatio * defaultFOV, aimingAnimationSpeed * Time.deltaTime));
            }
            else
            {
                m_WeaponMainLocalPosition = Vector3.Lerp(m_WeaponMainLocalPosition, default-
WeaponPosition.localPosition, aimingAnimationSpeed * Time.deltaTime);
                SetFOV(Mathf.Lerp(m_PlayerCharacterController.playerCamera.fieldOfView, de-
faultFOV, aimingAnimationSpeed * Time.deltaTime));
            }
        }
    }

    // Updates the weapon bob animation based on character speed
    void UpdateWeaponBob()
    {
        if (Time.deltaTime > 0f)
        {
            Vector3 playerCharacterVelocity = (m_PlayerCharacterController.transform.posi-
tion - m_LastCharacterPosition) / Time.deltaTime;

            // calcu-
late a smoothed weapon bob amount based on how close to our max grounded movement ve-
locity we are
            float characterMovementFactor = 0f;
            if (m_PlayerCharacterController.isGrounded)
            {
                characterMovementFactor = Mathf.Clamp01(playerCharacterVelocity.magni-
tude / (m_PlayerCharacterController.maxSpeedOnGround * m_PlayerCharacterControl-
ler.sprintSpeedModifier));
            }
            m_WeaponBobFactor = Mathf.Lerp(m_WeaponBobFactor, characterMovementFac-
tor, bobSharpness * Time.deltaTime);

            // Calculate vertical and horizontal weapon bob values based on a sine function
            float bobAmount = isAiming ? aimingBobAmount : defaultBobAmount;
            float frequency = bobFrequency;
```

```
            float hBobValue = Mathf.Sin(Time.time * frequency) * bobAmount * m_WeaponBob-
Factor;
            float vBobValue = ((Mathf.Sin(Time.time * frequency * 2f) * 0.5f) + 0.5f) * bobA-
mount * m_WeaponBobFactor;

            // Apply weapon bob
            m_WeaponBobLocalPosition.x = hBobValue;
            m_WeaponBobLocalPosition.y = Mathf.Abs(vBobValue);

            m_LastCharacterPosition = m_PlayerCharacterController.transform.position;
        }
    }

    // Updates the weapon recoil animation
    void UpdateWeaponRecoil()
    {
        // if the accumulated recoil is further away from the current position, make the current po-
sition move towards the recoil target
        if (m_WeaponRecoilLocalPosition.z >= m_AccumulatedRecoil.z * 0.99f)
        {
            m_WeaponRecoilLocalPosition = Vector3.Lerp(m_WeaponRecoilLocalPosition, m_Ac-
cumulatedRecoil, recoilSharpness * Time.deltaTime);
        }
        // otherwise, move recoil position to make it recover towards its resting pose
        else
        {
            m_WeaponRecoilLocalPosition = Vector3.Lerp(m_WeaponRecoilLocalPosition, Vec-
tor3.zero, recoilRestitutionSharpness * Time.deltaTime);
            m_AccumulatedRecoil = m_WeaponRecoilLocalPosition;
        }
    }

    // Updates the animated transition of switching weapons
    void UpdateWeaponSwitching()
    {
        // Calculate the time ratio (0 to 1) since weapon switch was triggered
        float switchingTimeFactor = 0f;
        if (weaponSwitchDelay == 0f)
        {
            switchingTimeFactor = 1f;
        }
        else
        {
            switchingTimeFactor = Mathf.Clamp01((Time.time - m_TimeStartedWeap-
onSwitch) / weaponSwitchDelay);
```

```
    }

    // Handle transiting to new switch state
    if(switchingTimeFactor >= 1f)
    {
        if (m_WeaponSwitchState == WeaponSwitchState.PutDownPrevious)
        {
            // Deactivate old weapon
            WeaponController oldWeapon = GetWeaponAtSlotIndex(activeWeaponIndex);
            if (oldWeapon != null)
            {
                oldWeapon.ShowWeapon(false);
            }

            activeWeaponIndex = m_WeaponSwitchNewWeaponIndex;
            switchingTimeFactor = 0f;

            // Activate new weapon
            WeaponController newWeapon = GetWeaponAtSlotIndex(activeWeaponIndex);
            if (onSwitchedToWeapon != null)
            {
                onSwitchedToWeapon.Invoke(newWeapon);
            }

            if(newWeapon)
            {
                m_TimeStartedWeaponSwitch = Time.time;
                m_WeaponSwitchState = WeaponSwitchState.PutUpNew;
            }
            else
            {
                // if new weapon is null, don't follow through with putting weapon back up
                m_WeaponSwitchState = WeaponSwitchState.Down;
            }
        }
        else if (m_WeaponSwitchState == WeaponSwitchState.PutUpNew)
        {
            m_WeaponSwitchState = WeaponSwitchState.Up;
        }
    }

    // Handle moving the weapon socket position for the animated weapon switching
    if (m_WeaponSwitchState == WeaponSwitchState.PutDownPrevious)
    {
```

```
            m_WeaponMainLocalPosition = Vector3.Lerp(defaultWeaponPosition.localPosition, downWeaponPosition.localPosition, switchingTimeFactor);
        }
        else if (m_WeaponSwitchState == WeaponSwitchState.PutUpNew)
        {
            m_WeaponMainLocalPosition = Vector3.Lerp(downWeaponPosition.localPosition, defaultWeaponPosition.localPosition, switchingTimeFactor);
        }
    }

    // Adds a weapon to our inventory
    public bool AddWeapon(WeaponController weaponPrefab)
    {
        // if we already hold this weapon type (a weapon coming from the same source prefab), don't add the weapon
        if(HasWeapon(weaponPrefab))
        {
            return false;
        }

        // search our weapon slots for the first free one, assign the weapon to it, and return true if we found one. Return false otherwise
        for (int i = 0; i < m_WeaponSlots.Length; i++)
        {
            // only add the weapon if the slot is free
            if(m_WeaponSlots[i] == null)
            {
                // spawn the weapon prefab as child of the weapon socket
                WeaponController weaponInstance = Instantiate(weaponPrefab, weaponParentSocket);
                weaponInstance.transform.localPosition = Vector3.zero;
                weaponInstance.transform.localRotation = Quaternion.identity;

                // Set owner to this gameObject so the weapon can alter projectile/damage logic accordingly
                weaponInstance.owner = gameObject;
                weaponInstance.sourcePrefab = weaponPrefab.gameObject;
                weaponInstance.ShowWeapon(false);

                // Assign the first person layer to the weapon
                int layerIndex = Mathf.RoundToInt(Mathf.Log(FPSWeaponLayer.value, 2)); // This function converts a layermask to a layer index
                foreach (Transform t in weaponInstance.gameObject.GetComponentsInChildren<Transform>(true))
                {
```

```
                t.gameObject.layer = layerIndex;
            }

            m_WeaponSlots[i] = weaponInstance;

            if(onAddedWeapon != null)
            {
                onAddedWeapon.Invoke(weaponInstance, i);
            }

            return true;
        }
    }

    // Handle auto-switching to weapon if no weapons currently
    if (GetActiveWeapon() == null)
    {
        SwitchWeapon(true);
    }

    return false;
}

public bool RemoveWeapon(WeaponController weaponInstance)
{
    // Look through our slots for that weapon
    for (int i = 0; i < m_WeaponSlots.Length; i++)
    {
        // when weapon found, remove it
        if(m_WeaponSlots[i] == weaponInstance)
        {
            m_WeaponSlots[i] = null;

            if (onRemovedWeapon != null)
            {
                onRemovedWeapon.Invoke(weaponInstance, i);
            }

            Destroy(weaponInstance.gameObject);

            // Handle case of removing active weapon (switch to next weapon)
            if(i == activeWeaponIndex)
            {
                SwitchWeapon(true);
            }
```

```
                return true;
            }
        }

        return false;
    }

    public WeaponController GetActiveWeapon()
    {
        return GetWeaponAtSlotIndex(activeWeaponIndex);
    }

    public WeaponController GetWeaponAtSlotIndex(int index)
    {
        // find the active weapon in our weapon slots based on our active weapon index
        if(index >= 0 &&
            index < m_WeaponSlots.Length)
        {
            return m_WeaponSlots[index];
        }

        // if we didn't find a valid active weapon in our weapon slots, return null
        return null;
    }

    // Calculates the "distance" between two weapon slot indexes
    // For example: if we had 5 weapon slots, the distance be-
tween slots #2 and #4 would be 2 in ascending order, and 3 in descending order
    int GetDistanceBetweenWeaponSlots(int fromSlotIndex, int toSlotIndex, bool ascend-
ingOrder)
    {
        int distanceBetweenSlots = 0;

        if (ascendingOrder)
        {
            distanceBetweenSlots = toSlotIndex - fromSlotIndex;
        }
        else
        {
            distanceBetweenSlots = -1 * (toSlotIndex - fromSlotIndex);
        }

        if (distanceBetweenSlots < 0)
        {
```

```
            distanceBetweenSlots = m_WeaponSlots.Length + distanceBetweenSlots;
        }

        return distanceBetweenSlots;
    }

    void OnWeaponSwitched(WeaponController newWeapon)
    {
        if (newWeapon != null)
        {
            newWeapon.ShowWeapon(true);
        }
    }
}
```

PrefabReplacerOnInstance

```
using System.Collections.Generic;
#if UNITY_EDITOR
using UnityEditor;
#endif
using UnityEngine;

[ExecuteInEditMode]
public class PrefabReplacerOnInstance : MonoBehaviour
{
    public GameObject TargetPrefab;

    void Awake()
    {
#if UNITY_EDITOR
        List<GameObject> allPrefabObjectsInScene = new List<GameObject>();
        // Lo ocupo para que evalua si hay un objeto en la escena
        foreach (Transform t in GameObject.FindObjectsOfType<Transform>())
        {
            if (PrefabUtility.IsAnyPrefabInstanceRoot(t.gameObject))
            {
                allPrefabObjectsInScene.Add(t.gameObject);
            }
        }

        foreach (GameObject go in allPrefabObjectsInScene)
        {
            GameObject instanceSource = PrefabUtility.GetCorrespondingObjectFromSource(go);
```

```
            if (instanceSource == TargetPrefab)
            {
                transform.SetParent(go.transform.parent);
                transform.position = go.transform.position;
                transform.rotation = go.transform.rotation;
                transform.localScale = go.transform.localScale;

                // Undo.Register
                Undo.DestroyObjectImmediate(go);

                Debug.Log("Replaced prefab in scene");
                DestroyImmediate(this);
                break;
            }
        }
#endif
    }
}
```

WeaponPickup.cs

```
using UnityEngine;

[RequireComponent(typeof(Pickup))]
public class WeaponPickup : MonoBehaviour
{
    [Tooltip("The prefab for the weapon that will be added to the player on pickup")]
    public WeaponController weaponPrefab;

    Pickup m_Pickup;

    void Start()
    {
        m_Pickup = GetComponent<Pickup>();
        DebugUtility.HandleErrorIfNullGetCompo-
nent<Pickup, WeaponPickup>(m_Pickup, this, gameObject);

        // Subscribe to pickup action
        m_Pickup.onPick += OnPicked;

        // Set all children layers to default (to prefent seeing weapons through meshes)
        foreach(Transform t in GetComponentsInChildren<Transform>())
        {
            if (t != transform)
                t.gameObject.layer = 0;
        }
```

```
    }

    void OnPicked(PlayerCharacterController byPlayer)
    {
        PlayerWeaponsManager playerWeaponsManager = byPlayer.GetComponent<Player-
WeaponsManager>();
        if (playerWeaponsManager)
        {
            if (playerWeaponsManager.AddWeapon(weaponPrefab))
            {
                // Handle auto-switching to weapon if no weapons currently
                if (playerWeaponsManager.GetActiveWeapon() == null)
                {
                    playerWeaponsManager.SwitchWeapon(true);
                }

                m_Pickup.PlayPickupFeedback();

                Destroy(gameObject);
            }
        }
    }
}
```

Resultados

Recursos informáticos

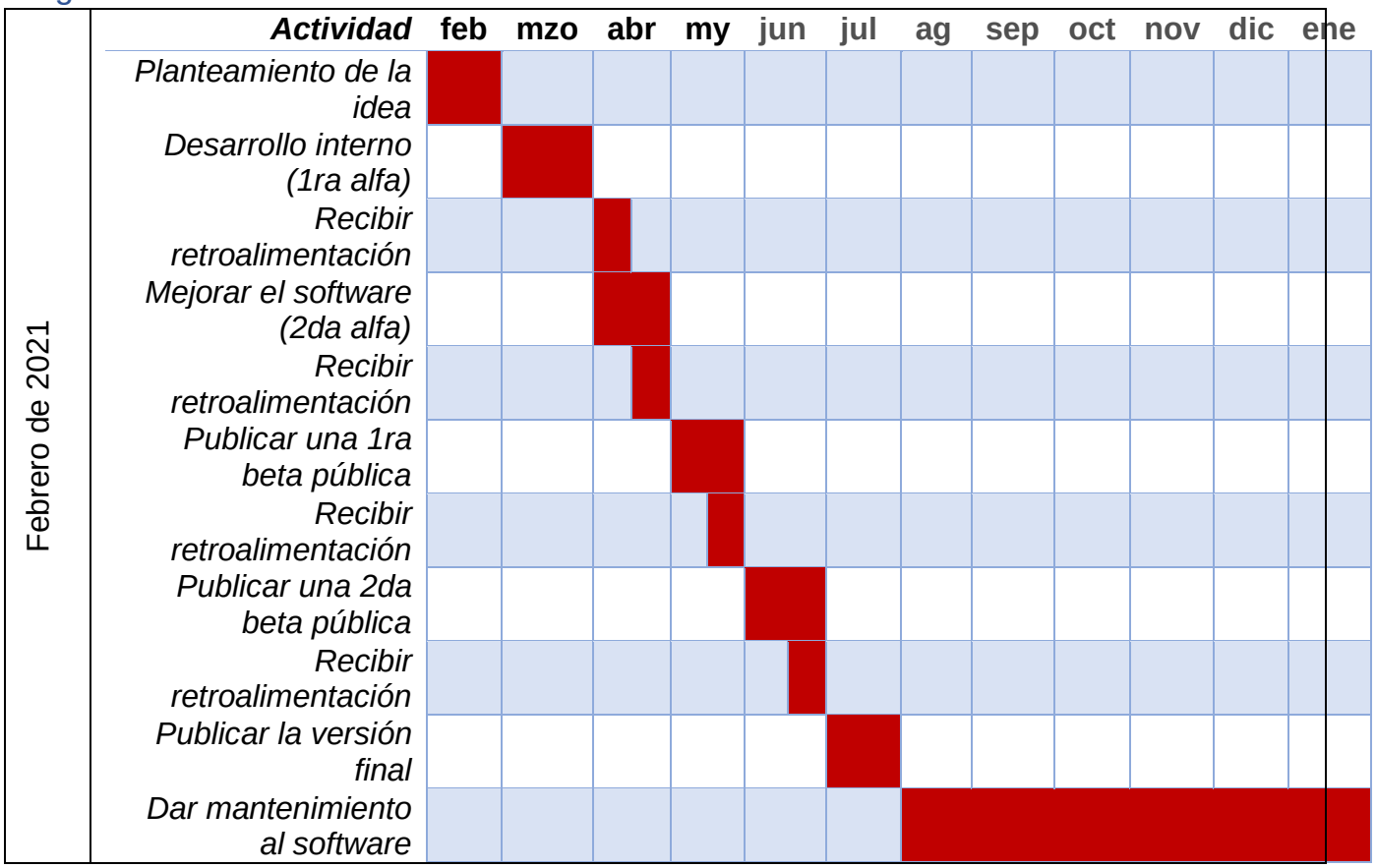
Febrero de 2021	Agosto de 2021
• Una computadora con 8 GB de RAM y con tarjeta gráfica dedicada • Descargar la plantilla	• Una computadora con 8 GB de RAM y con tarjeta gráfica dedicada • Descargar la plantilla

• Y entender cómo funciona el juego base sin modificaciones	• Y entender cómo funciona el juego base sin modificaciones

Costos asociados del proyecto

Febrero de 2020	Febrero de 2021
20000 MXN por una laptop gamer para tener suficientes recursos para desarrollar el proyecto	20000 MXN por una laptop gamer para tener suficientes recursos para desarrollar el proyecto

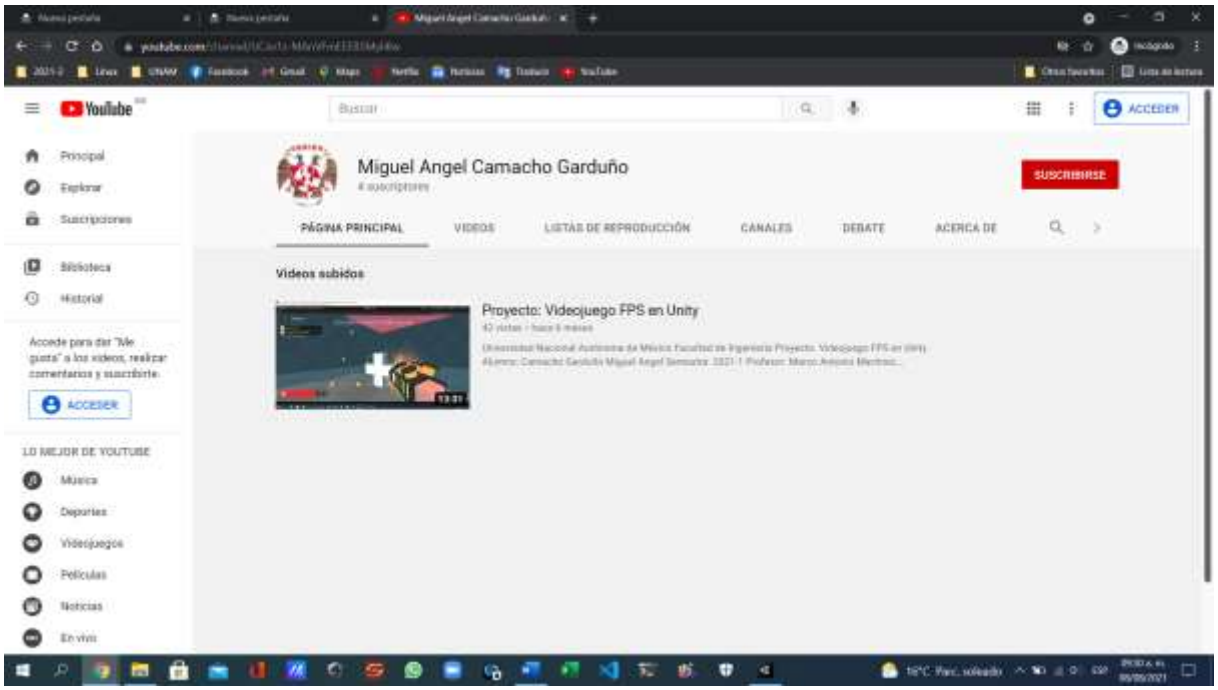
Diagramas de Gantt



Agosto de 2021	Actividad	feb	mzo	abr	myo	jun	jul	ag	sep	oct	nov	dic	enet
	Planteamiento de la idea												
	Desarrollo interno (1ra alfa)												
	Recibir retroalimentación												
	Mejorar el software (2da alfa)												
	Recibir retroalimentación												
	Publicar una 1ra beta pública												
	Recibir retroalimentación												
	Publicar una 2da beta pública												
	Recibir retroalimentación												
	Publicar la versión final												
	Dar mantenimiento al software												

Canal de YouTube

https://www.youtube.com/channel/UCxv1z-MAnVFmEEEBSMyl4lw



Vídeo:

Repositorio de GitHub del proyecto
https://github.com/Mike-Camacho/EDAI/tree/main/Proyecto

Conclusiones

- Es importante realizar algoritmos, ya que dice los pasos a seguir para solucionar un problema en específico, si no existieran los algoritmos, la vida como lo conocemos sería más complicada, por no decir imposible. Una situación parecida existe con respecto a las estructuras de datos, ya que, sin ellas, la recolección de información se haría de forma manual lo que llevaría más tiempo y menor efectividad.
- A diferencia del semestre pasado que al principio no contaba con un equipo potente, ahora todo fue más sencillo ya que desde el inicio del semestre pude realizar con antelación los cambios pertinentes al proyecto del semestre pasado.

Referencias

Facultat d'Informàtica de Barcelona |. (s.f.). *Historia de los videojuegos*. Recuperado el 25 de enero de 2021, de Facultat d'Informàtica de Barcelona: https://www.fib.upc.edu/retro-informatica/historia/videojocs.html

Jensen, K. T. (s.f.). *The Complete History Of First-Person Shooters*. Obtenido de PCMag: https://www.pcmag.com/news/the-complete-history-of-first-person-shooters

Microsoft. (s.f.). *Documentos de C#: inicio, tutoriales y referencias*. Recuperado el noviembre de 2020, de Microsoft Docs: https://docs.microsoft.com/es-mx/dotnet/csharp/

Microsoft. (s.f.). *Floating-Point Parsing and Formatting improvements in .NET Core 3.0*. Recuperado el 25 de enero de 2021, de .NET Blog: https://devblogs.microsoft.com/dotnet/floating-point-parsing-and-formatting-improvements-in-net-core-3-0/

Plan Ceibal. (s.f.). *¿Qué es un videojuego?* Recuperado el 25 de enero de 2021, de Plan Ceibal: https://blogs.ceibal.edu.uy/formacion/faqs/que-es-un-videojuego/

Royal Baloo. (s.f.). *101 Reasons That Video Games Can Be Educational* -. Recuperado el 25 de enero de 2021, de Royal Baloo: https://royalbaloo.com/video-games-educational/#:~:text=Educational%20video%20games%20are%20important,Develops%20hand%2Feye%20coordination.&text=Children%20can%20learn%20programming%2C%20coding,home%2C%20and%20many%20other%20skills.

Unity. (s.f.). *FPS Microgame*. Recuperado el noviembre de 2020, de Unity Learn: https://learn.unity.com/project/fps-template

Universidad Internacional de Valencia. (s.f.). *¿Qué son los juegos fps y por qué son tan conocidos?* Recuperado el 25 de enero de 2021, de VIU: https://www.universidadviu.com/es/actualidad/nuestros-expertos/que-son-los-juegos-fps-y-por-que-son-tan-conocidos

Wikipedia. (s.f.). *Nivel (videojuegos)*. Recuperado el 07 de 08 de 2021, de Wikipedia, la

enciclopedia libre: https://es.wikipedia.org/wiki/Nivel_(videojuegos)
Yelich, L. (2 de marzo de 2020). *Stack Data Structure Explained With Video Games*. Recuperado el 07 de agosto de 2021, de Medium: https://medium.com/@lianyelich14/stack-data-structure-explained-with-video-games-58a5e69b007a

Glosario

- Actor: es el personaje animado que aparecerá en el juego, ya sea que sea amigo o enemigo
- Fotograma: Cada una de las imágenes que se suceden en una película, serie o video.
- GitHub: Es una plataforma para alojar proyectos utilizando el sistema de control de versiones Git
- Jet Pack: Es un dispositivo tipo mochila que se coloca en la espalda y que permite volar al usuario
- Munición: Es el conjunto de proyectiles que dispara un arma de fuego que disparado a algún objeto puede causar daño en él
- Objetivo: Lo que se desea lograr o completar
- Open Source: Es el tipo de software cuyo código fuente puede ser accedido por cualquier persona y puede ser modificado por terceros
- Plantilla: Es un objecto o software que sirve de guía para poder realizar modificaciones si el desarrollador lo desea
- Pseudocódigo: es una descripción de alto nivel compacta e informal del principio operativo de un programa informático u otro algoritmo
- Unity: es un motor de videojuego multiplataforma creado por Unity Technologies y está disponible como plataforma de desarrollo para Microsoft Windows, Mac OS, Linux.

Acrónimos

- EA: Electronic Arts
- FPS: First Person Shooters
- GNU: GNU's Not Unix
- IEEE: *Institute of Electrical and Electronics Engineers*