

Asignatura	Datos del alumno	Fecha
Desarrollo de Videojuegos I	Apellidos: Nombre:	20/02/2026

Actividad 3. Grupal. Top-Down completo

Integrantes

- Jorge Blacio
- Luis Diaz
- Gustavo Totoy

Introducción

El prototipo de top down se desarrolló usando el motor Unity en su versión 6.3.8f1, y se realizó un build web que se encuentra disponible en la plataforma Itchio.

Enlaces del proyecto

- Enlace de GitHub
 - <https://github.com/gtotoy/unir-top-down>
- Enlace de Itchio
 - <https://jorgeblacio.itch.io/the-legend-of-top-down>
- Enlace a video de gameplay y organización de proyecto Unity
 - https://alumnosunir-my.sharepoint.com/:v/g/personal/luis_diaz609_comunidadunir_net/IQ Ae6U6kUtmPSKInf-WncstRAdPOvf9GlgGtoUVLcdL82so?e=kZvz3J

Listado de Implementaciones Extra

- **Menús de juego**
 - Menú principal con acceso directo a la partida.
 - Menú de pausa accesible en cualquier momento durante el juego.

Asignatura	Datos del alumno	Fecha
Desarrollo de Videojuegos I	Apellidos: Nombre:	20/02/2026

- Menú de Game Over con opción de reinicio.
- **Mecánicas adicionales del jugador**
 - Bloqueo (clic derecho): reduce el daño recibido a la mitad y elimina el tinte rojo de impacto.
 - Dash (barra espaciadora): movimiento rápido con cooldown, permite esquivar proyectiles o reposicionarse.
- **Sistema de enemigos con comportamientos diferenciados**
 - Hachero: persigue y ataca en rango corto con mecánica de cuerpo a cuerpo directa.
 - Lancero: mantiene distancia prudencial y ataca en rango medio; retrocede si el jugador se acerca demasiado.
 - Arquero: huye del jugador y dispara flechas como proyectil físico con detección de impacto.
- **Sistema de spawning de enemigos**
 - Puntos configurables de spawn por grupos de combinaciones de tipos de enemigo, parametrizables desde el inspector.
- **Elementos de recuperación**
 - Spawning automático de med kits (representados como filetes) con radio y frecuencia configurables.
- **Diseño sonoro**
 - Música de fondo en loop para cada nivel.
 - Efectos de sonido por acción: ataque, bloqueo, dash, recibir daño, recoger med kit, abrir puerta.
- **Efectos visuales y de retroalimentación**
 - Animaciones de sprites para dash, impacto, consumo de med kit y apertura de puerta.
 - Tinte rojo al recibir daño, tinte alternativo al bloquear.
 - Efecto de cámara (avatar UI) con sacudida y flash al recibir daño.
- **Diseño de niveles**

Asignatura	Datos del alumno	Fecha
Desarrollo de Videojuegos I	Apellidos: Nombre:	20/02/2026

- Nivel 1: tres tipos de enemigos, puerta con cerradura, llave otorgada al eliminar todos los enemigos.
- Nivel 2: jefe final con máquina de estados que combina ataque, bloqueo y dash de forma aleatoria ponderada.

Desarrollo del proyecto

A continuación, se encuentran capturas de la configuración del proyecto de Unity:

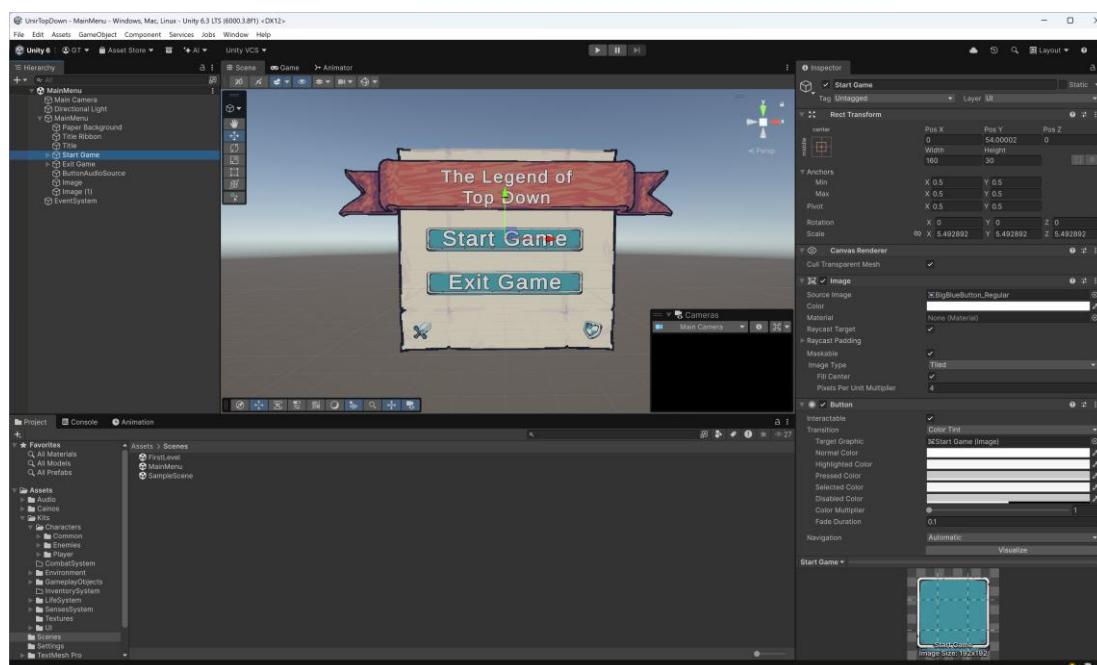


Figura 1. Escena del menú principal. Fuente: Elaboración propia.

Asignatura	Datos del alumno	Fecha
Desarrollo de Videojuegos I	Apellidos:	20/02/2026
	Nombre:	

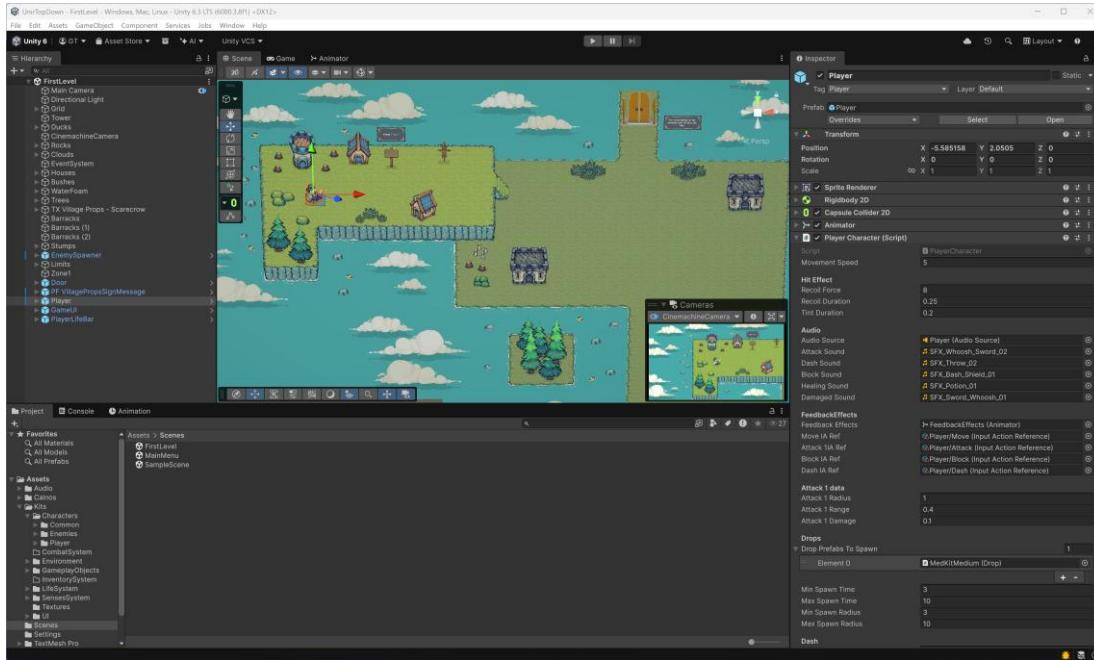


Figura 2. Escena del primer nivel que tiene un spawner con los 3 tipos de enemigos y requiere de eliminarlos a todos para obtener la llave de la puerta al castillo del jefe. Fuente: Elaboración propia.

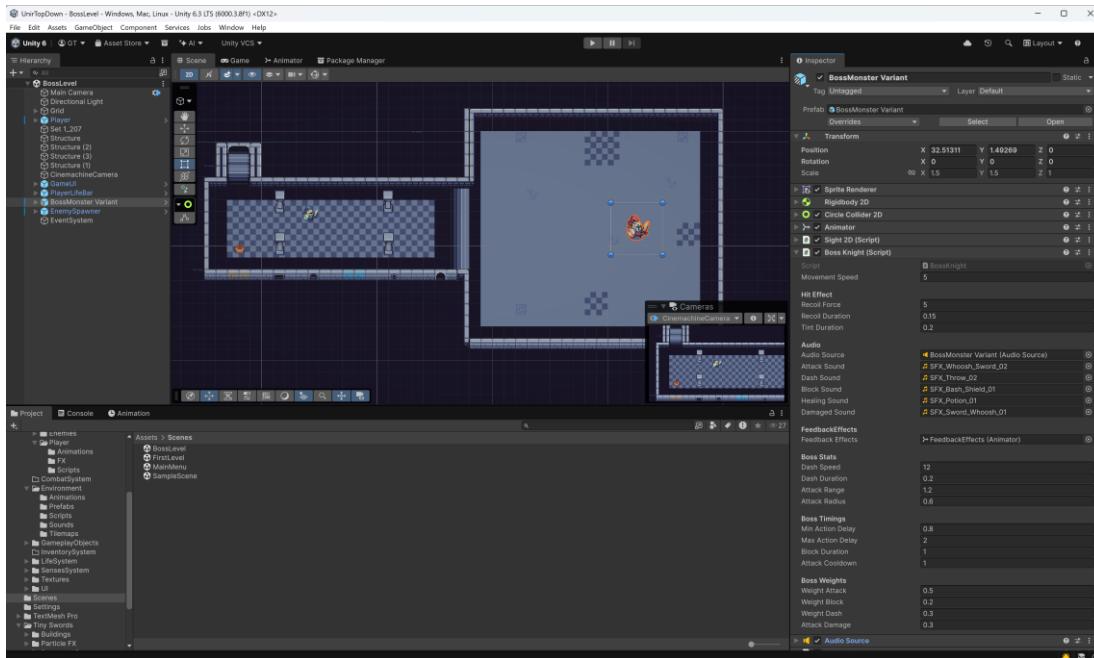


Figura 3. Escena del segundo nivel que se realiza al interior del castillo del jefe final rodeado de sus aliados. Fuente: Elaboración propia.

Asignatura	Datos del alumno	Fecha
Desarrollo de Videojuegos I	Apellidos: Nombre:	20/02/2026

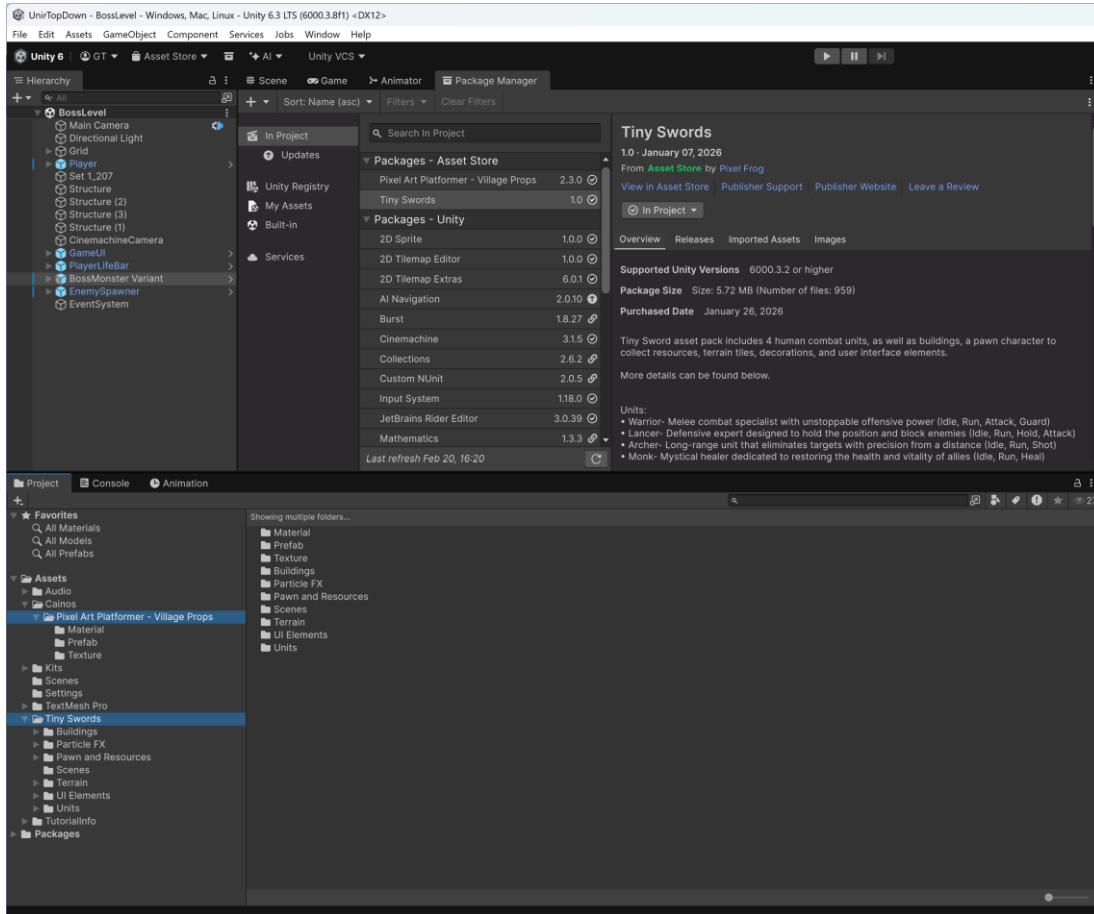


Figura 4. Assets packages obtenidos del Asset Store. Fuente: Elaboración propia.

A continuación, se encuentran capturas del código C# programado:

Asignatura	Datos del alumno	Fecha
Desarrollo de Videojuegos I	Apellidos:	20/02/2026
	Nombre:	

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.InputSystem;

[Header("Attack")]
public class PlayerCharacter : BaseCharacter
{
    [SerializeField] InputActionReference moveIARef;
    [SerializeField] InputActionReference attackIARef;
    [SerializeField] InputActionReference blockIARef;
    [SerializeField] InputActionReference dashIARef;

    [Header("Attack 1")]
    [SerializeField] float attackRadius = 0.5f;
    [SerializeField] float attackDamage = 0.1f;
    [SerializeField] float attackCd = 0.5f;

    [Header("Drop")]
    [SerializeField] Drop[] dropPrefabsToSpam;
    [SerializeField] float minSpamTime = 1f;
    [SerializeField] float maxSpamTime = 5f;
    [SerializeField] float minDashRadius = 1f;
    [SerializeField] float maxDashRadius = 7f;

    [Header("Dash")]
    [SerializeField] float dashForce = 15f;
    [SerializeField] float dashDuration = 0.2f;
    [SerializeField] float dashCd = 1f;

    private float lastDashTime = -999f;

    private void OnEnable()
    {
        moveIARef.action.Enable();
        moveIARef.action.started += HandleMoveInputAction;
        moveIARef.action.performed += HandleMoveInputAction;
        moveIARef.action.canceled += HandleMoveInputAction;

        attackIARef.action.Enable();
        attackIARef.action.performed += HandleAttack1InputAction;

        blockIARef.action.Enable();
        blockIARef.action.started += HandleBlockInputAction;
        blockIARef.action.canceled += HandleBlockInputAction;

        dashIARef.action.Enable();
        dashIARef.action.performed += HandleDashInputAction;
    }

    private void OnDisable()
    {
    }
}

[RequireComponent(typeof(Rigidbody2D))]
public class BaseCharacter : MonoBehaviour
{
    protected SpriteRenderer spriteRend;
    protected Rigidbody2D body;
    protected Animator animator;
    protected Life life;

    [SerializeField] float movementSpeed = 5;

    private Vector2 movementDir;
    private Vector2 lookDir;

    [Header("Hit Effect")]
    [SerializeField] float recoilForce = 5f;
    [SerializeField] float recoilDuration = 0.15f;
    [SerializeField] float tintDuration = 0.2f;

    [Header("Audio")]
    [SerializeField] AudioSource audioSource;
    [SerializeField] AudioClip attackSound;
    [SerializeField] AudioClip dashSound;
    [SerializeField] AudioClip blockSound;
    [SerializeField] AudioClip healingSound;
    [SerializeField] AudioClip damagedSound;

    [Header("Feedback Effects")]
    [SerializeField] Animator feedbackEffects;

    private bool isRecoiling = false;
    private bool isBlocking = false;
    protected bool IsBlocking
    {
        get => isBlocking;
        set
        {
            isBlocking = value;
            animator.SetBool("IsBlocking", value);
        }
    }

    private bool isDashing = false;
    protected bool IsDashing
    {
        get => isDashing;
        set
        {
            isDashing = value;
        }
    }
}

```

Figura 5. Snippet de código del PlayerCharacter que muestra a su lado su clase base BaseCharacter siguiendo la arquitectura definida en clases. Fuente: Elaboración propia.

```

[Serializable]
public class EnemySpawner : MonoBehaviour
{
    public GameObject prefab;
    public int count = 1;

    [System.Serializable]
    public class EnemyGroup
    {
        public string groupName;
        public List<EnemyEntry> enemies;
        public Transform spawnPoint;
    }

    public class EnemySpawner : MonoBehaviour
    {
        [Header("Groups")]
        [SerializeField] List<EnemyGroup> groups;
        [Header("Settings")]
        [SerializeField] bool spawnOnStart = true;
        [SerializeField] int defaultGroupIndex = 0;
        [SerializeField] float spawnScatterRadius = 1f;

        private List<GameObject> activeEnemies = new List<GameObject>();

        private void Start()
        {
            if (spawnOnStart)
                SpawnAllGroupsSequential();
        }

        public void SpawnGroup(int groupIndex)
        {
            if (groupIndex < 0 || groupIndex >= groups.Count)
            {
                Debug.LogWarning($"EnemySpawner: group index {groupIndex} out of bounds");
                return;
            }
            StartCoroutine(SpawnGroupCoroutine(groups[groupIndex]));
        }

        public void SpawnAllGroupsSequential(float delayBetweenGroups = 1f)
        {
            StartCoroutine(SpawnAllGroupsCoroutine(delayBetweenGroups));
        }
    }
}

[Header("Attack")]
public class AxeMonster : BaseMonster
{
    [SerializeField] float attackRange = 1.2f;
    [SerializeField] float attackCd = 1.5f;
    [SerializeField] float attackRadius = 0.6f;
    [SerializeField] float attackDamage = 0.1f;

    private float lastAttackTime;

    protected override void UpdateBehavior()
    {
        var target = sight.GetClosestTargetInSight();
        if (target == null) { SetMovementDirection(Vector2.zero); }

        Vector2 toTarget = target.transform.position - transform.position;
        float dist = toTarget.magnitude;

        if (dist < attackRange)
        {
            SetMovementDirection(toTarget);
            TryAttack(target);
        }
        else
        {
            SetMovementDirection(toTarget.normalized);
        }
    }

    private void TryAttack(Collider2D target)
    {
        if (Time.time < lastAttackTime + attackCd) return;
        lastAttackTime = Time.time;
        playAttackSound();
        animator.SetTrigger("Attack");

        Vector2 dir = (target.transform.position - transform.position).normalized;
        var hits = Physics2D.CircleCastAll(transform.position, a
        foreach (var hit in hits)
        {
            var character = hit.collider.GetComponent<BaseCharacter>();
            if (character != null && character != this)
                character.NotifyAttack(dir, attackDamage);
        }
    }
}

```

Figura 6. Snippet de código del EnemySpawner + AxeMonster. Adicional a AxeMonster también constan LancerMonster y ArcherMonster con comportamientos distintos. Fuente: Elaboración propia.

Asignatura	Datos del alumno	Fecha
Desarrollo de Videojuegos I	Apellidos:	20/02/2026
	Nombre:	

```

using UnityEngine;
[RequireComponent(typeof(Sight2D))]
public class BaseMonster : BaseCharacter
{
    protected Sight2D sight;
    protected override void Awake()
    {
        base.Awake();
        sight = GetComponent<Sight2D>();
    }
    protected override void Update()
    {
        UpdateBehavior();
        base.Update();
    }
    protected virtual void UpdateBehavior()
    {
        var target = sight.GetClosestTargetInSight();
        SetMovementDirection(target != null
            ? (target.transform.position - transform.position).normalized
            : Vector3.zero);
    }
}

```

```

public class BossKnight : BaseMonster
{
    [SerializeField] float dashSpeed = 12f;
    [SerializeField] float dashDuration = 0.2f;
    [SerializeField] float attachRadius = 0.6f;
    [SerializeField] float attackRadius = 0.6f;
    [Header("Boss Timings")]
    [SerializeField] float minActionDelay = 0.8f;
    [SerializeField] float maxActionDelay = 2f;
    [SerializeField] float blockDuration = 1f;
    [SerializeField] float attackCooldown = 1f;
    [Header("Boss Weights")]
    [SerializeField] float weightDash = 0.9f;
    [SerializeField] float weightBlock = 0.2f;
    [SerializeField] float weightAttack = 0.3f;
    private enum BossState { Idle, Deciding, Attacking, Blocking }
    private BossState state = BossState.Idle;
    private float lastAttackTime;
    private Transform player;
    protected override void Awake()
    {
        base.Awake();
    }
    protected override void UpdateBehavior()
    {
        var target = sight.GetClosestTargetInSight();
        if (target != null) player = target.transform;
        if (state == BossState.Idle)
        {
            state = BossState.Deciding;
            StartCoroutine(DecideNextAction());
        }
        if (player != null && state != BossState.Dashing)
        {
            Vector3 toPlayer = player.position - transform.position;
            if (state == BossState.Attacking && state != BossState.Blocking)
                SetMovementDirection(toPlayer.normalized);
        }
    }
    private IEnumerator DecideNextAction()
    
```

Figura 7. Snippet de código del BaseMonster con BossKnight a su lado. Fuente: Elaboración propia.

```

using UnityEngine;
[RequireComponent(typeof(EnemySpawner))]
public class ZoneController : MonoBehaviour
{
    [SerializeField] private EnemySpawner enemySpawner;
    [SerializeField] private GameObject keyFlyPrefab;
    [SerializeField] private Door door;
    private bool isPlayerInZone = false;
    private void Update()
    {
        if (isPlayerInZone)
        {
            if (enemySpawner.AllEnemiesDead())
            {
                //Debug.Log("Player has cleared the zone!");
                Instantiate(keyFlyPrefab, transform.position, Quaternion.identity);
                door.setsUnlocked();
                Destroy(gameObject);
            }
        }
    }
    private void OnTriggerEnter2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            isPlayerInZone = true;
        }
    }
    private void OnTriggerExit2D(Collider2D collision)
    {
        if (collision.CompareTag("Player"))
        {
            isPlayerInZone = false;
        }
    }
}

```

```

using System.Collections;
using UnityEngine;
public class Door : MonoBehaviour
{
    [SerializeField] Gameobject canvasMessage;
    [SerializeField] Controller zoneController;
    private bool isLocked = true;
    private Animator anim;
    private void Awake()
    {
        canvasMessage.SetActive(false);
        anim = GetComponent

```

Figura 8. Snippet de código de ZoneController + Door. Responsables de instanciar la llave y controlar el acceso de la puerta para continuar al siguiente nivel. Fuente: Elaboración propia.

Conclusiones

Asignatura	Datos del alumno	Fecha
Desarrollo de Videojuegos I	Apellidos: Nombre:	20/02/2026

El desarrollo del prototipo ha resultado una experiencia formativa completa que ha obligado al equipo a investigar y resolver problemas reales de diseño y programación de videojuegos.

Arquitectura extensible: la decisión de centralizar la lógica común en BaseCharacter y delegar el comportamiento específico a los subtipos mediante UpdateBehavior() permitió añadir nuevos arquetipos de enemigo sin modificar código existente, siguiendo el principio Open/Closed.

Problemas con la física de projectiles: el principal problema técnico fue la aplicación continua de fuerza al jugador cuando una flecha colisionaba. La causa raíz fue la interacción entre el *Rigidbody2D* del proyectil y el del jugador antes de que Destroy surtiera efecto. La solución fue desactivar body.simulated inmediatamente tras el impacto y usar un flag hasHit para evitar re-entradas.

Sincronización de animaciones y lógica: ajustar el timing del ataque del jugador con el momento real de detección de daño requirió varios ciclos de iteración, resolviéndose con un mejor balance de los tiempos de animación y de los rangos y radios configurables que definimos para el golpe.

Máquina de estados del jefe: un desafío de diseño fue evitar que el jefe se quedara bloqueado en un estado indefinidamente. Se resolvió garantizando que cada corutina de acción siempre establece state = BossState.Idle en su bloque final, incluso si el jugador sale del rango de visión.

Balance de jugabilidad: encontrar el equilibrio correcto en los pesos de las acciones del jefe y los cooldowns de dash y bloqueo del jugador requirió muchas sesiones de prueba. Los parámetros finales se expusieron en el inspector de Unity para facilitar el ajuste sin recomilar.

Asignatura	Datos del alumno	Fecha
Desarrollo de Videojuegos I	Apellidos: Nombre:	20/02/2026

Trabajo en equipo y control de versiones: el uso de GitHub con Unity requirió configurar correctamente el *.gitignore* y establecer convenciones claras para evitar conflictos en archivos de escena y prefabs.

En conjunto, el proyecto ha permitido al equipo consolidar conocimientos sobre arquitectura de software orientada a juegos, inteligencia artificial básica, diseño de niveles y flujo de trabajo colaborativo con control de versiones. Las implementaciones extra han supuesto un reto estimulante que ha impulsado la búsqueda activa de soluciones fuera del temario impartido en clase.