Programación de Juegos 3D en Unity

OLD BLOOD

Juan Alberto Domínguez Vázquez

Saúl Rodríguez Naranjo

Universidad de Huelva | Ingeniería Informática | Programación de Juegos (2024)

ÍNDICE

INTRODUCCIÓN

Old Blood es un videojuego de acción en primera persona basado en la jugabilidad del género FPS (First Person Shooter) o juego de disparos en primera persona. Cuenta con todos los elementos básicos característicos de esta clase de juegos, como son vista del arma en la zona derecha de la pantalla, un HUD en el que se muestra la munición actual del arma y la vida del jugador, además del punto de mira en el centro de la pantalla.

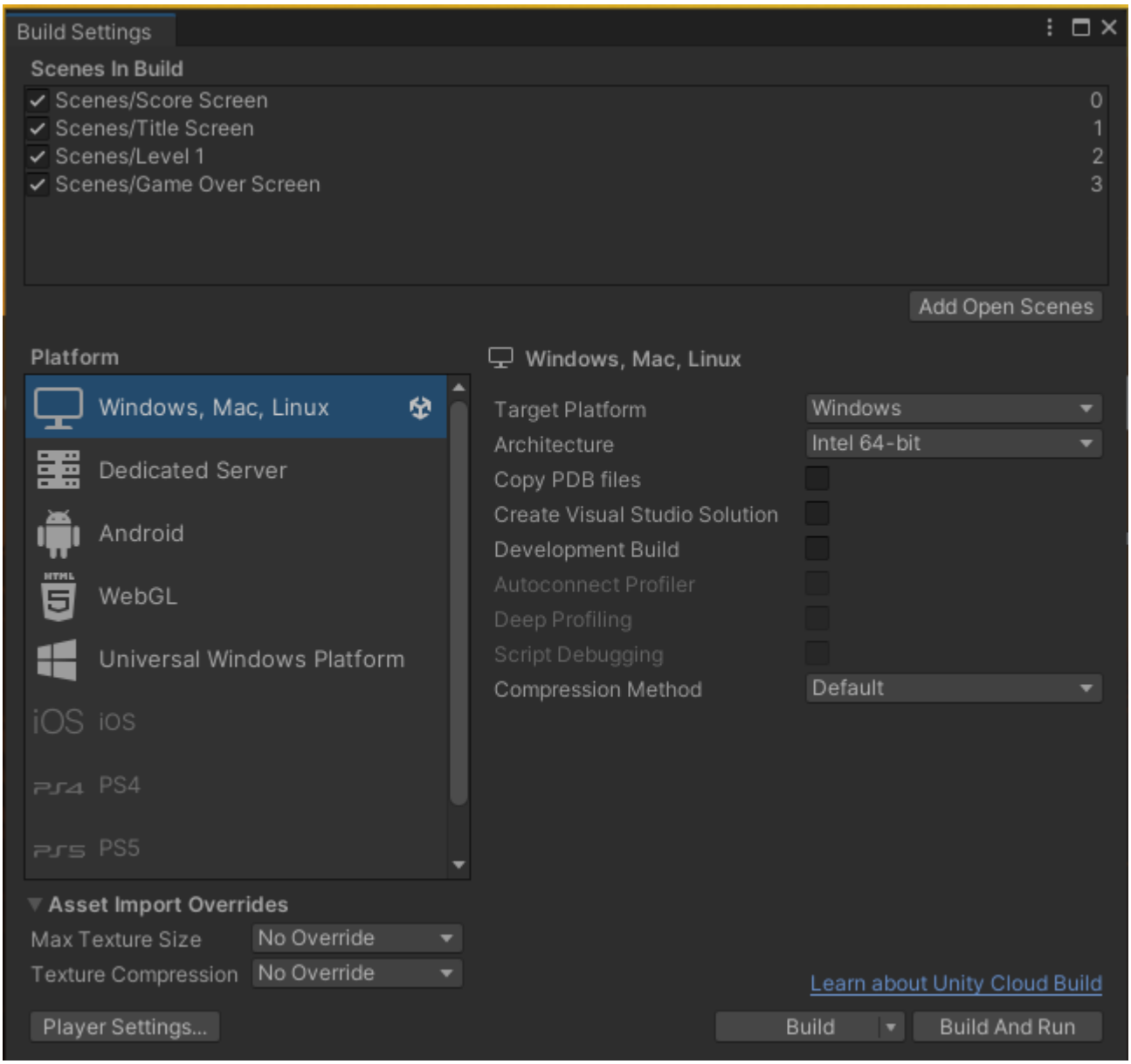
La temática principal consiste en la conquista de una base en un planeta alienígena cuya especie esta en conflicto con los humanos. Los enemigos irán apareciendo por oleadas y el jugador deberá ser capaz de sobreponerse a estas fuerzas enemigas el tiempo indicado en el HUD.

PANTALLA DE TÍTULO

En la pantalla de título podemos observar tres botones, uno para comenzar el juego, otro para poder realizar ciertas configuraciones del juego, como su volumen, tiempo de juego, etc. y otro adicional para salir del juego, que cerrará el programa.

En el menú de configuración se podrá personalizar características tales como el tiempo de juego, el volumen y el número de oleadas de enemigos que aparecerán.

Cabe añadir que para que se produzca el paso de escenas, se deberá configurar las builds setting en unity, añadiendo las respectivas escenas del juego.



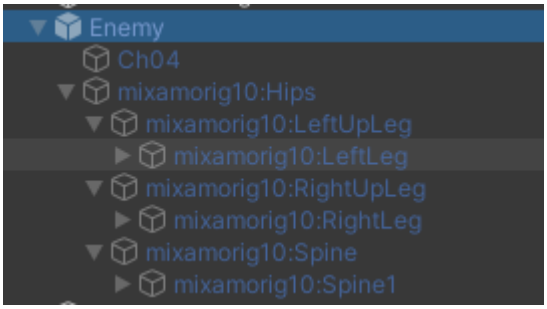
Con respecto al botón de “Salir”, la implementación es tal que cuenta con dos funcionalidades distintas, una para cuando el juego se ejecuta desde el propio editor de Unity y otra para cuando se inicia desde el propio ejecutable compilado.

```
public void exitButtonOnClick()
{
    if(Application.isPlaying)
    {
        UnityEditor.EditorApplication.isPlaying = false;
    }
    else
    {
        Application.Quit();
    }
}
```

PRIMER NIVEL

► IMPLEMENTACIÓN DE ENEMIGOS

Para este proyecto solo contamos con un tipo de enemigo, que en Unity tiene por nombre Enemy. Dicho objeto, es importado de la página de creación de personajes y animaciones Mixamo.com, por tanto, ya incluye ciertos elementos para la correcta visualización de las animaciones y sus físicas.



El script adherido a este objeto que describe los atributos de estos enemigos se llama EnemyCore y en él se define el daño que pueden hacer, su vida y el tiempo que tardan en “despawnear”. También en dicho script, están los objetos de tipo EnemySpawner y GenecticAlgorithm que más tarde se explicarán sus usos.

```
Script de Unity (1 referencia de recurso) | 21 referencias
public class EnemyCore : MonoBehaviour
{
    public Animator animator;
    private EnemySpawner spawner;
    private GenecticAlgorithm ga;

    public int damageGene;
    private int life = 10;
    public float lifeTimeGene;
    private float DespawnTime = 30f;

    Mensaje de Unity | 0 referencias
    void Start()
    {
        spawner = FindObjectOfType<EnemySpawner>();
        ga = FindObjectOfType<GenecticAlgorithm>();
        lifeTimeGene = Time.time;

    }
```

```
1 referencia
public void takeDamage(int damage)
{
    if (life >= damage)
    {
        life -= damage;
    }
    if (life <= 0)
    {
        //Activar animacion de muerte
        animator.SetTrigger("die");
        GetComponent<Collider>().enabled = false;
        if (spawner != null)
        {
            spawner.EnemyDestroyed();
        }
        if (ga != null)
        {
            ga.RemoveFromPopulation(this.gameObject); // Notifica al algoritmo genético que este enemigo ha muerto
        }
        if (animator.transform.gameObject.name == "Enemy")
        {
            animator.transform.gameObject.SetActive(false);
        }
        else
        {
            Destroy(animator.transform.gameObject, DespawnTime);
        }
        PlayerScore.incrementScore(1);
    }
    else
    {
        //Activar animacion de golpe
        animator.SetTrigger("damage");
    }


}

// Añade un método para inicializar los genes
2 referencias
public void InitializeGenes(int damage)
{
    this.damageGene = damage;
}

// Añade un método para obtener el tiempo de vida del enemigo
5 referencias
public float GetLifetime()
{
    return Time.time - lifeTimeGene;
}
```

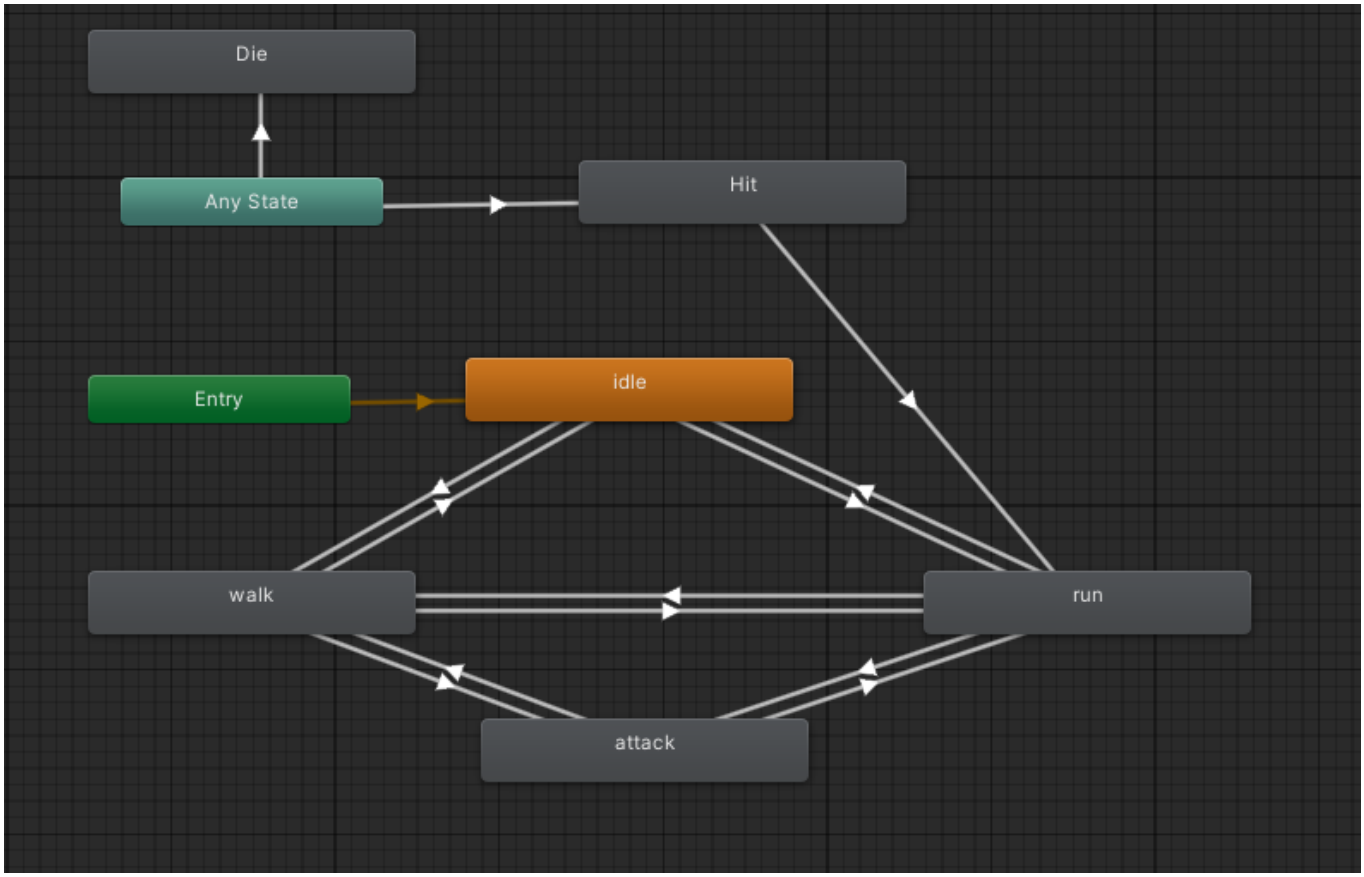
Analizando el script, en primer lugar, en el método start buscamos los objetos asignados al enemigo para su posterior uso. Después vemos el método takeDamage que se encarga de asignar daño a los enemigos en base al parámetro que se le pasa, también es encargado de activar las animaciones de muerte y daño cuando las condiciones se cumplan y de destruir al enemigo cuando el tiempo de “despawnear” se cumpla.

Por último, el método initializeGenes servirá para dotar de los genes necesarios a los enemigos a través del algoritmo genético.

► TÉCNICAS DE IA UTILIZADAS

Para el seguimiento del jugador se ha optado por usar la herramienta de NavMesh de Unity, añadiéndole al objeto Enemy un Nav Mesh Agent y configurando en el mapa un objeto de tipo Nav Mesh Surface donde se configura que zona del mapa puede ser usada por el algoritmo para seguir al enemigo, que en este caso es todo el mapa visible.

Por otro lado, para la gestión del comportamiento del enemigo y de sus animaciones se ha usado una máquina de estados finitos de la siguiente forma:



Estado IDLE:

Si nos centramos en el estado de idle se ha desarrollado el siguiente script para modelar su comportamiento:

```
Script de Unity | 0 referencias
public class IdleState : StateMachineBehaviour
{
    float temporizador;
    Transform player;
    float chaseRange = 8;

    // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    Mensaje de Unity | 0 referencias
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        temporizador = 0;
        player = GameObject.FindGameObjectWithTag("Player").transform;
    }

    // OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
    Mensaje de Unity | 0 referencias
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        temporizador += Time.deltaTime;
        if (temporizador > 5)
        {
            animator.SetBool("isWalking", true);
        }
        float distance = Vector3.Distance(player.position, animator.transform.position);
        if (distance < chaseRange)
        {
            animator.SetBool("isRunning", true);
        }
    }
```

Vemos que se inicia un temporizador para cuando pasen mas de 5 segundos, el enemigo comenzara a caminar, pasando por tanto al estado walk y si la distancia con el jugador es menor a 8 unidades, comenzará a correr hacia él pasando al estado run.

Estado WALK:

Pasemos al estado walk que tiene definido un script llamado PatrolState que modela su comportamiento de la siguiente manera:

```
Script de Unity | 0 referencias
public class PatrolState : StateMachineBehaviour
{
    float temporizador;
    List<Transform> wayPoints = new List<Transform>();
    NavMeshAgent agent;
    Transform player;
    float chaseRange = 8;

    // OnStateEnter is called when a transition starts and the state machine starts to evaluate this state
    Mensaje de Unity | 0 referencias
    override public void OnStateEnter(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        player = GameObject.FindGameObjectWithTag("Player").transform;
        agent = animator.GetComponent<NavMeshAgent>();
        agent.speed = 1.5f;
        temporizador = 0;
        GameObject objects = GameObject.FindGameObjectWithTag("WayPoints");

        foreach (Transform t in objects.transform) {
            wayPoints.Add(t);
        }

        agent.SetDestination(wayPoints[Random.Range(0, wayPoints.Count)].position);
    }

    // OnStateUpdate is called on each Update frame between OnStateEnter and OnStateExit callbacks
    Mensaje de Unity | 0 referencias
    override public void OnStateUpdate(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        if (!agent.pathPending && agent.remainingDistance <= agent.stoppingDistance)
        {
            agent.SetDestination(wayPoints[Random.Range(0, wayPoints.Count)].position);
        }

        temporizador += Time.deltaTime;
        if (temporizador > 3)
        {
            animator.SetBool("isWalking", false);
        }

        float distance = Vector3.Distance(player.position, animator.transform.position);
        if (distance < chaseRange)
        {
            animator.SetBool("isRunning", true);
        }
    }
```

```
    // OnStateExit is called when a transition ends and the state machine finishes evaluating this state
    Mensaje de Unity | 0 referencias
    override public void OnStateExit(Animator animator, AnimatorStateInfo stateInfo, int layerIndex)
    {
        agent.SetDestination(agent.transform.position);
    }
```

Funcionalidad Principal

- **Patrullaje Aleatorio:** El agente selecciona aleatoriamente puntos de referencia (wayPoints) y se mueve hacia ellos.
- **Detección del Jugador:** Si el jugador se encuentra dentro de un rango definido (chaseRange), el agente cambia su comportamiento para perseguir al jugador.
- **Actualización del Animador:** Dependiendo del estado del agente (patrullando o persiguiendo), se actualizan los parámetros del animador (isWalking, isRunning) para reflejar visualmente estos estados.

Detalles de Implementación

- **Variables:**
  - temporizador: Mide el tiempo para controlar las transiciones de comportamiento.
  - wayPoints: Lista de puntos de referencia para el patrullaje.
  - agent: Componente NavMeshAgent que maneja el movimiento del agente.
  - player: Transform del jugador para calcular la distancia.
  - chaseRange: Distancia a la cual el agente comienza a perseguir al jugador.
- **Métodos Sobrecargados:**
  - OnStateEnter: Inicializa las variables y configura el destino inicial del agente.
  - OnStateUpdate: Actualiza el destino del agente, controla el temporizador y cambia los parámetros del animador basado en la distancia al jugador.
  - OnStateExit: Detiene el movimiento del agente cuando el estado se desactiva.

Estado RUN:

Si vemos el estado run, veremos que tiene añadido el script ChaseState es un comportamiento personalizado para agentes de IA en Unity que define cómo el agente persigue al jugador y reacciona cuando está lo suficientemente cerca para atacar. Este script se integra con el sistema de animaciones de Unity mediante la herencia de StateMachineBehaviour, permitiendo transiciones suaves entre estados.

Funcionalidad Principal

- **Persecución del Jugador:** El agente persigue continuamente al jugador.
- **Transición a Estado de Ataque:** Si el agente se acerca lo suficiente al jugador, cambia su comportamiento para iniciar un ataque.
- **Actualización del Animador:** Dependiendo de la distancia al jugador, se actualizan los parámetros del animador (isRunning, isAttacking) para reflejar visualmente estos estados.

Detalles de Implementación

- **Variables:**
  - NavMeshAgent agent: Componente de Unity que maneja el movimiento del agente en una malla de navegación.
  - Transform player: Transform del jugador para calcular la distancia y establecer el destino.
- **Métodos Sobrecargados:**
  - OnStateEnter:
    - Se obtiene el componente NavMeshAgent del agente.
    - Se encuentra la posición del jugador.
    - Se ajusta la velocidad del agente para la persecución (3.5f).
  - OnStateUpdate:
    - Se establece la posición del jugador como el destino del agente.
    - Se calcula la distancia entre el jugador y el agente.
    - Si la distancia es mayor a 15 unidades, se desactiva el parámetro isRunning del animador.
    - Si la distancia es menor a 2.5 unidades, se activa el parámetro isAttacking del animador.
  - OnStateExit:

        - Se detiene el movimiento del agente estableciendo su posición actual como destino.

Estado ATTACK:

Funcionalidad Principal

- Orientación hacia el Jugador: El agente se orienta continuamente hacia el jugador.
- Detección de Proximidad: Si el jugador está lo suficientemente cerca, el agente intentará atacar.
- Ataque con Retraso: El ataque del agente tiene un retraso configurado para no ocurrir continuamente.
- Actualización del Animador: Dependiendo de la distancia al jugador, se actualiza el parámetro del animador (isAttacking) para reflejar visualmente estos estados.

Detalles de Implementación

- Variables:
    - Transform player: Transform del jugador para calcular la distancia y orientar al agente.
    - DateTime lastTimePlayerHit: Almacena la última vez que el jugador fue golpeado.
    - int playerHitDelayInSeconds: Intervalo de tiempo en segundos entre ataques consecutivos.
- Métodos Sobrecargados:
    - OnStateEnter:
        - Se obtiene la posición del jugador al iniciar el estado.
    - OnStateUpdate:
        - El agente se orienta hacia el jugador.
        - Se calcula la distancia entre el jugador y el agente.
        - Si la distancia es mayor a 3.5 unidades, se desactiva el parámetro isAttacking del animador.
        - Si la distancia es menor o igual a 3.5 unidades, se realiza un Raycast para detectar si el jugador está dentro del rango de ataque (2 unidades).
        - Si el Raycast golpea al jugador y ha pasado el tiempo suficiente desde el último ataque, se aplica daño al jugador utilizando el componente PlayerController.
    - OnStateExit:
        - Actualmente vacío, pero puede ser utilizado para limpiar o resetear variables al salir del estado.

Por último, los estados Die y Hurt, definen respectivamente el estado de muerte del enemigo y el estado de dañado cuando recibe daño y se activan sus triggers, controlados ambos en el script EnemyCore.

Por otro lado, también se ha diseñado un algoritmo genético cuyo objetivo es mejorar las características de los enemigos del juego, veámoslo poco a poco:

```
Script de Unity (1 referencia de recurso) | 4 referencias
public class GenecticAlgorithm : MonoBehaviour
{
    public GameObject enemyPrefab;
    private List<GameObject> population = new List<GameObject>();
    public int populationSize = 10;
    private float mutationRate = 0.2f;
    public EnemySpawner spawner;

    Mensaje de Unity | 0 referencias
    void Start()
    {
        spawner = FindObjectOfType<EnemySpawner>();
        for (int i = 0; i < populationSize; i++)
        {
            GameObject enemy = Instantiate(enemyPrefab, new Vector3(Random.Range(-10, 10), 0.5f, Random.Range(-10, 10)), Quaternion.identity);
            enemy.GetComponent<EnemyCore>().InitializeGenes(Random.Range(0, 10)); // Inicializar con genes aleatorios
            enemy.SetActive(true);
            population.Add(enemy);
        }

    }
```

```
Mensaje de Unity | 0 referencias
void Update()
{
    if (population.Count <= 4)
    {
        Debug.Log("Tamaño de la poblacion: " + population.Count);
        List<GameObject> newPopulation = new List<GameObject>();

        while (newPopulation.Count < populationSize)
        {
            GameObject parent1 = SelectParent();
            GameObject parent2 = SelectParent();

            Debug.Log("Genes del padre 1: " + parent1.GetComponent<EnemyCore>().damageGene + ", " + parent1.GetComponent<EnemyCore>().GetLifetime());
            Debug.Log("Genes del padre 2: " + parent2.GetComponent<EnemyCore>().damageGene + ", " + parent2.GetComponent<EnemyCore>().GetLifetime());

            GameObject offspring = CrossOver(parent1, parent2);

            Debug.Log("Genes del hijo antes de la mutación: " + offspring.GetComponent<EnemyCore>().damageGene + ", " + offspring.GetComponent<EnemyCore>().GetLifetime());

            Mutate(offspring);

            Debug.Log("Genes del hijo después de la mutación: " + offspring.GetComponent<EnemyCore>().damageGene + ", " + offspring.GetComponent<EnemyCore>().GetLifetime());

            newPopulation.Add(offspring);
        }

        population = newPopulation;
    }
}

2 referencias
GameObject SelectParent()
{
    int fathersSize = 2; // Definir la cantidad de padres a seleccionar
    List<GameObject> tournament = new List<GameObject>();

    // Seleccionar individuos aleatorios de la población
    for (int i = 0; i < fathersSize; i++)
    {
        tournament.Add(population[Random.Range(0, population.Count)]);
    }

    // Ordena a los individuos por aptitud
    tournament.Sort((x, y) => CalculateFitness(y).CompareTo(CalculateFitness(x)));

    // Devuelve el individuo más apto
    return tournament[0];
}
```

```
2 referencias
float CalculateFitness(GameObject enemy)
{
    // Aquí se define cómo calcular la aptitud de un individuo.
    // En este caso, daño hecho al jugador / tiempo de vida
    EnemyCore genes = enemy.GetComponent<EnemyCore>();
    float fitness = genes.damageGene * genes.GetLifetime();
    Debug.Log("Fitness del enemigo: " + fitness);
    return fitness;
}

1 referencia
GameObject CrossOver(GameObject parent1, GameObject parent2)
{
    GameObject offspring = spawner.SpawnEnemy();
    EnemyCore parent1Genes = parent1.GetComponent<EnemyCore>();
    EnemyCore parent2Genes = parent2.GetComponent<EnemyCore>();
    offspring.GetComponent<EnemyCore>().InitializeGenes((parent1Genes.damageGene + parent2Genes.damageGene) / 2);
    return offspring;
}

1 referencia
void Mutate(GameObject enemy)
{
    if (Random.Range(0f, 2f) <= mutationRate)
    {
        EnemyCore genes = enemy.GetComponent<EnemyCore>();
        genes.damageGene = Random.Range(0, 10);
    }
}

1 referencia
public void RemoveFromPopulation(GameObject enemy)
{
    population.Remove(enemy);
    spawner.EnemyDestroyed();
}
```

El script GeneticAlgorithm implementa un algoritmo genético para la evolución de una población de enemigos en Unity. Utiliza técnicas de selección, cruce y mutación para mejorar las características de los enemigos a lo largo del tiempo. Este script se adjunta a un objeto de Unity y gestiona la creación y evolución de la población de enemigos.

Funcionalidad Principal

- **Inicialización de la Población:** Crea una población inicial de enemigos con genes aleatorios.
- **Evolución de la Población:** Cuando la población disminuye por debajo de un umbral, genera una nueva generación de enemigos mediante selección, cruce y mutación.
- **Selección de Padres:** Selecciona padres basados en su aptitud para generar descendencia.
- **Cruce y Mutación:** Combina los genes de dos padres para crear descendencia y aplica mutaciones aleatorias.

Detalles de Implementación

- **Variables:**
  - GameObject enemyPrefab: Prefab del enemigo utilizado para instanciar nuevos enemigos.
  - List<GameObject> population: Lista que almacena la población actual de enemigos.
  - int populationSize: Tamaño de la población.
  - float mutationRate: Tasa de mutación aplicada a los genes de la descendencia.
  - EnemySpawner spawner: Referencia al spawner de enemigos para instanciar nuevos enemigos.
- **Métodos:**
  - Start(): Inicializa la población con enemigos instanciados con genes aleatorios.
  - Update(): Supervisa el tamaño de la población y genera una nueva generación si es necesario.
  - SelectParent(): Selecciona los padres mediante un torneo de selección basado en la aptitud.
  - CalculateFitness(GameObject enemy): Calcula la aptitud de un enemigo basada en su daño y tiempo de vida.
  - CrossOver(GameObject parent1, GameObject parent2): Crea un nuevo enemigo combinando los genes de dos padres.
  - Mutate(GameObject enemy): Aplica una mutación aleatoria a los genes de un enemigo.
  - RemoveFromPopulation(GameObject enemy): Elimina un enemigo de la población y actualiza el spawner.

Para finalizar este apartado hay que explicar como son spawneados los enemigos generados por el algoritmo genético y para ello usamos el script EnemySpawner:

El script EnemySpawner gestiona la aparición de enemigos en Unity, asegurando que se generen de manera periódica y controlando el número total de enemigos activos en el juego. Se integra con el algoritmo genético (GeneticAlgorithm) para mantener la población de enemigos.

Funcionalidad Principal

- **Generación Periódica de Enemigos:** Genera enemigos a intervalos regulares.
- **Control del Número de Enemigos:** Asegura que el número de enemigos no exceda el tamaño de la población definida.
- **Interacción con Algoritmo Genético:** Utiliza la clase GeneticAlgorithm para conocer el tamaño de la población y ajustarse a ella.

Detalles de Implementación

- **Variables:**
  - GameObject enemyPrefab: Prefab del enemigo utilizado para instanciar nuevos enemigos.
  - Transform[] spawnPoints: Puntos de aparición predefinidos para los enemigos.
  - float spawnInterval: Intervalo de tiempo entre cada aparición de enemigo.
  - GeneticAlgorithm ga: Referencia al algoritmo genético para obtener el tamaño de la población.

- float timer: Temporizador para controlar el intervalo de aparición.
- int currentEnemyCount: Contador del número actual de enemigos en el juego.

- **Métodos:**
  - Start(): Inicializa el temporizador y obtiene la referencia al algoritmo genético.
  - Update(): Actualiza el temporizador y genera un enemigo si se ha alcanzado el intervalo de tiempo y el número de enemigos es menor al tamaño de la población.
  - SpawnEnemy():
    - Selecciona un punto de aparición aleatorio.
    - Instancia un nuevo enemigo en el punto seleccionado.
    - Activa el collider del enemigo y lo marca como activo.
    - Incrementa el contador de enemigos actuales.
  - EnemyDestroyed(): Decrementa el contador de enemigos cuando un enemigo es destruido.

► IMPLEMENTACIÓN DEL ARMA

El arma utilizada por el jugador es el "Fusil Reaper". Sus assets se han descargado desde la Unity Asset Store (https://assetstore.unity.com/packages/3d/props/guns/submachine-gun-reaper-7890#description). El paquete del arma contiene además del modelado en sí del fusil, los prefabs de los casquillos de bala, los cuales se expulsan cada vez que se dispara.

Todo el comportamiento del arma se engloba en la clase WeaponCtrl, siendo que en los métodos Start y Awake es donde se inicializan todos los parámetros necesarios para su correcto funcionamiento.

```
void Awake()
{
    // Determine which components this weapon will have
    _cartridgeSpawPoint = GetComponentInChildren<CartridgeSpawnPoint>();
    if (_cartridgeSpawPoint == null)
    {
        Debug.LogWarning("There is no Cartridge Spawn Point defined in the weapon");
    }
    _shootSpawnPoint = GetComponentInChildren<ShootSpawnPoint>();
    if (_shootSpawnPoint == null)
    {
        Debug.LogWarning("There is no Shoot Spawn Point defined in the weapon");
    }
    _spotlightSpawnPoint = GetComponentInChildren<SpotLightSpawnPoint>();
    if (_spotlightSpawnPoint == null)
    {
        Debug.LogWarning("There is no SpotLight Spawn Point defined in the weapon");
    }
}
```

En el método Awake se obtienen las posiciones físicas desde las cuales se expulsarán los casquillos al disparar, y también desde donde se emitirá el característico fogonazo de un fusil al disparar.

```
void Start()
{
    // init the available components.
    if (_cartridgeSpawPoint != null)
    {
        _cartridgeSpawPoint.Setup(CartridgesPrefab.gameObject);
    }
    if (_shootSpawnPoint != null)
    {
        _shootSpawnPoint.Setup(_cartridgeSpawPoint, ShootPrefab.gameObject);
    }
    if (_spotlightSpawnPoint != null)
    {
        //_spotlightSpawnPoint.Setup(SpotLightPrefab.gameObject);
    }

    weaponStatistics = new WeaponStatistics(ReloadAmount, maximumAmmo, ReloadAmount, infiniteAmmo);
    weaponStatistics.AddListener(this);

    Reload();
}
```

En el método Start, se inicializan los prefabs necesarios para los casquillos y los efectos de sonido y fogonazo del arma. Además, se inicializa un objeto de la clase WeaponStatistics, que es la encargada de controlar los niveles de munición del arma, la capacidad del cargador, etc.

Por defecto los cargadores serán de 30 balas, y la munición infinita. Para poder sincronizar en tiempo real el nivel de munición del arma con el HUD, se utiliza el patrón observer mediante la interfaz WeaponStatusListener, la cual implementa la clase WeaponCtrl. Se añade la propia clase WeaponCtrl como listener de WeaponStatistics. Cada vez que el arma se dispare o recargue, se verá reflejado en el HUD.

```
namespace Assets.Scripts.Player
{
    6 referencias
    public interface WeaponStatusListener
    {
        2 referencias
        void OnWeaponReload();

        2 referencias
        void onWeaponShot();
    }
}
```

```
public void OnWeaponReload()
{
    string weaponHUDText = ReloadAmount.ToString();

    if (infiniteAmmo)
    {
        weaponHUDText += "/\u221E";

    }
    else
    {
        weaponHUDText += "/" + maximumAmmo.ToString();
    }

    weaponHud.SetText(weaponHUDText);
}
```

```
public void onWeaponShot()
{
    string weaponHUDText = weaponStatistics.Ammo.ToString();

    if (infiniteAmmo)
    {
        weaponHUDText += "/\u221E";

    }
    else
    {
        weaponHUDText += "/" + maximumAmmo.ToString();
    }

    weaponHud.SetText(weaponHUDText);
}
```

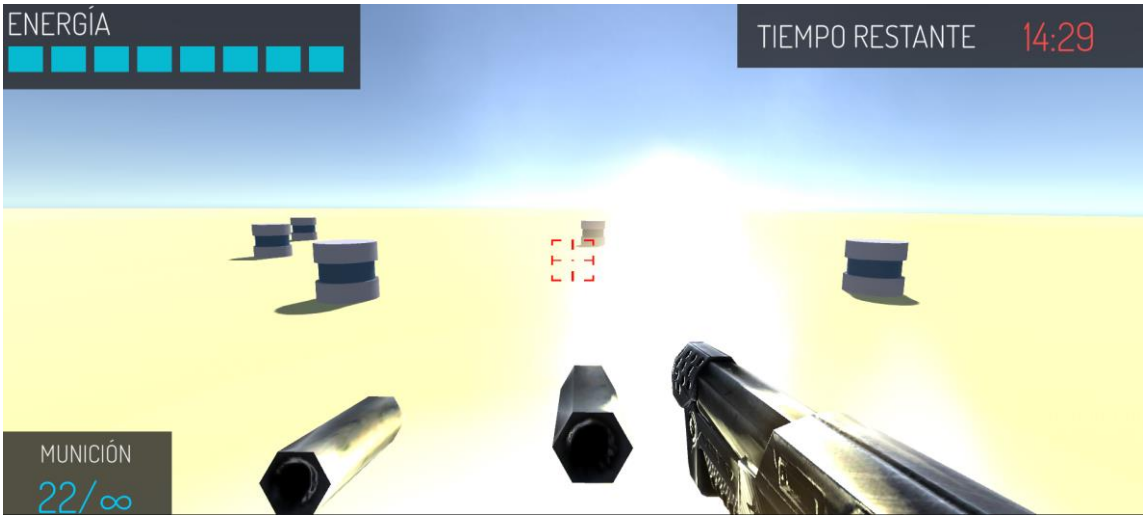
Por último, el arma se recarga, para inicializarla con la munición correspondiente.

```
private void Update()
{
    if(Input.GetKeyDown(KeyCode.R))
    {
        Reload();
    }

    if(Input.GetMouseButtonDown(0))
    {
        DoShoot();
    }
}
```

En cada ciclo Update, se comprueba si el jugador ha presionado la tecla R, para recargar el arma, y también el click derecho del ratón, que disparará una bala por cada vez que se pulse, resultando así la cadencia del arma semiautomática.

El arma cada vez que es disparada, instancia una copia de un prefab de un casquillo de bala, que desaparece a los pocos segundos. Esto aumenta la inmersión ya que proporciona más detalles, y además no merma el rendimiento.



La lógica de disparo funciona mediante un RayCast que se lanza desde el punto medio de la pantalla (El crosshair), si este logra impactar con un enemigo, el enemigo recibirá un punto de daño.

```
public void DoShoot()
{
    if (CurrentAmunition > 0)
    {
        // Consume a bullet
        CurrentAmunition -= 1;
        if (_shootSpawnPoint != null)
        {
            Ray ray = camera.ScreenPointToRay(new Vector3(Screen.width / 2, Screen.height / 2));
            RaycastHit hit;
            float weaponRange = 100;

            if(Physics.Raycast(ray, out hit, weaponRange))
            {
                GameObject hitGameObject = hit.transform.gameObject;

                if (hitGameObject != null)
                {
                    EnemyCore enemyCore = hitGameObject.GetComponent<EnemyCore>();

                    if (enemyCore != null)
                    {
                        //Debug.Log("Enemy hit");
                        enemyCore.takeDamage(1);
                    }
                }
```

► IMPLEMENTACIÓN DE LA VIDA DEL JUGADOR

Del mismo modo que con la lógica del arma, para controlar los puntos de vida del jugador se utiliza la clase llamada CharacterStatistics y la interfaz ICharacterStatusListener, implementando el patrón observer. Cada vez que el jugador reciba daño, muera, o recupere salud, se verá reflejado en la interfaz.

```
namespace Assets.Scripts.Character
{
    7 referencias
    public interface ICharacterStatusListener
    {
        2 referencias
        void OnCharacterDeath();

        2 referencias
        void onCharacterHealed();

        2 referencias
        void onCharacterDamaged();
    }
}
```

El personaje recupera 1 tanque de energía cada 10 segundos. Los enemigos quitarán tantos tanques de energía al jugador, como daño tengan definido en el momento de su instanciación.

Cuando el jugador muera, pierde el juego, y se le lleva a la pantalla de Game Over.

PANTALLA DE PUNTUACIÓN

En la pantalla de puntuación vemos una tabla con las puntuaciones de los 10 primeros jugadores de la siguiente forma:

Como vemos se permite introducir el nombre del jugador para, si supera las puntuaciones por defecto que se cargan a través de un .txt, aparecerá en dicha tabla y luego será redireccionado a la pantalla de inicio.

RECURSOS EXTERNOS

- **Assets para el Nivel 1:** https://assetstore.unity.com/packages/3d/environments/sci-fi/sci-fi-styled-modular-pack-82913
- **Fusil Reaper:** https://assetstore.unity.com/packages/3d/props/guns/submachine-gun-reaper-7890#description
- **Imagen de fondo pantalla de Game Over:** https://www.freepik.com/free-ai-image/war-zone-landscape-with-apocalyptic-destruction_94957863.htm#fromView=search&page=1&position=7&uuid=10799834-6838-4d33-afad-cb88c4a21490%22%3EImage
- **Sonido de recarga del arma:** https://www.fesliyanstudios.com/royalty-free-sound-effects-download/gun-reloading-302
- **Punto de mira (crosshair):** https://assetstore.unity.com/packages/2d/gui/icons/simple-modern-crosshairs-pack-1-79034#content
- **Música:** https://pixabay.com/es/users/u_8ed4bpovet-41953350/?utm_source=link-attribution&utm_medium=referral&utm_campaign=music&utm_content=189720%22%3Eu_8ed4bpovet

  https://pixabay.com/es/users/audiosto-40753689/?utm_source=link-attribution&utm_medium=referral&utm_campaign=music&utm_content=179699%22%3EAudiosto

  https://pixabay.com/es/users/fanchisanchez-5047476/?utm_source=link-attribution&utm_medium=referral&utm_campaign=music&utm_content=127554%22%3EFrancisco

- **Imagen de pantalla de puntuación y título: Generadas mediante Inteligencia Artificial.**