**EXAMEN 1ª Evluación Víctor de la Fuente**

**GDD:**

**1. Título:** The Core Duty School.

**2. Resumen:** Controlas un personaje en primera persona que puede disparar NPCs que interactúan contigo mismo.

**3. Objetivo:** Dispara a todos los objetivos antes de que ellos te alcancen.

**4. Género:** Shooter y Superviviencia

**5. Plataformas:** Ordenador, escalable a otras plataformas.

**6. Estilo visual:** 3D muy básico con pocos assets.

**7. Jugabilidad:** Controles básicos: puedes correr, saltar, moverte en todos los ejes y girar la cámara a todos lados, puedes disparar y recargar aparte de matar objetivos en movimiento.

**8. Sistema de progresión:** Solo un objetivo, matar todos los objetivos sin ser alcanzado.

**9. Desafíos y obstáculos:** Diferentes tipos de NPCS que podrán alcanzarte si te quedas quieto, mátalos antes de que te maten.

**10.Argumento:** Aparecerás en un escenario muy sencillo donde podrás ver enemigos que te seguirán a donde vayas, no dejes que se acerquen demasiado y mátalos con tu arma.

**11.Personajes:** Solo estarás tu como personaje principal y una serie de enemigos a los que disparar. Ninguno tiene más relevancia en el juego que otro.

**12.Ambientación:** Shooter en un espacio muy reducido para que puedas encontrar fácilmente a todos los enemigos y ellos puedan seguirte sin tener que perseguirte por un terreno muy amplio

**13.Estructura de niveles:** Solo un nivel. El juego terminará cuando mates a todos los enemigos o ellos te maten a ti, dificultad muy baja.

**14.Elementos interactivos:** El personaje solo podrá interactuar consigo mismo y con el arma que tiene, disparando a los enemigos.

**15.Motor de juego:** Unity

**16.Requisitos técnicos:**

Requisitos Mínimos

-Sistema Operativo: Windows 7/8/10 (o equivalente en macOS/Linux) -Procesador: Intel Core i3 o equivalente

-Memoria RAM: 4 GB -Tarjeta Gráfica: NVIDIA GeForce 440 o equivalente (o integrada con soporte para OpenGL 2.0)

-DirectX: Versión 9.0c -Espacio en Disco: Al menos 500 MB disponibles

-Resolución de Pantalla: 1280x720 (HD) Requisitos Recomendados

-Sistema Operativo: Windows 10 -Procesador: Intel Core i5 o equivalente -Memoria RAM: 8 GB -Tarjeta Gráfica: NVIDIA GeForce 660 o equivalente

-DirectX: Versión 11 -Espacio en Disco: Al menos 1 GB disponibles

-Resolución de Pantalla: 1920x1080 (Full HD)

**17.Estilo musical:** Ninguno, solo efectos de sonido al disparar.

**18.Referencias:** Call of duty & tutorial de youtube: <https://www.youtube.com/watch?v=wu87qiCWB1o&list=PLvS4Ct-H_3MuwjEm8VZ5W-zvrH8xiNaWy>

**CREACIÓN DEL VIDEOJUEGO:**

**1ºSCRIPTS:**

**1º-> PLAYER CONTROLLER:**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

// Este atributo asegura que el componente "CharacterController" esté siempre presente en el GameObject al que se adjunta este script.

[RequireComponent(typeof(CharacterController))]

public class PlayerController : MonoBehaviour

{

// Encabezado en el Inspector para organizar mejor las variables de referencia.

[Header("References")]

public Camera playerCamera; // Referencia a la cámara del jugador para controlar la rotación de la vista.

// Encabezado en el Inspector para variables generales.

[Header("General")]

public float gravityScale = -20f; // Escala de la gravedad aplicada al jugador.

// Encabezado en el Inspector para variables relacionadas con el movimiento.

[Header("Movement")]

public float walkSpeed = 5f; // Velocidad al caminar.

public float runSpeed = 10f; // Velocidad al correr.

// Encabezado en el Inspector para variables de rotación.

[Header("Rotation")]

public float rotationSensibility = 10f; // Sensibilidad de rotación para la cámara y el jugador.

// Encabezado en el Inspector para variables relacionadas con el salto.

[Header("Jump")]

public float jumpHeight = 1.9f; // Altura máxima alcanzada al saltar.

// Variables privadas.

private float cameraVerticalAngle; // Ángulo vertical de la cámara para limitar la rotación hacia arriba y abajo.

Vector3 moveInput = Vector3.zero; // Vector para almacenar el movimiento del jugador.

Vector3 rotationinput = Vector3.zero; // Vector para almacenar la rotación del jugador.

CharacterController characterController; // Referencia al componente "CharacterController" para manejar el movimiento.

private void Awake()

{

// Obtiene la referencia al componente "CharacterController" del GameObject.

characterController = GetComponent<CharacterController>();

}

private void Start()

{

// Bloquea el cursor dentro de la ventana del juego y lo oculta.

Cursor.lockState = CursorLockMode.Locked;

}

private void Update()

{

// Llama a las funciones de mirar y moverse en cada cuadro.

Look(); // Maneja la rotación de la cámara y el jugador.

Move(); // Maneja el movimiento del jugador.

}

private void Move()

{

if (characterController.isGrounded) // Comprueba si el jugador está en el suelo.

{

// Obtiene la entrada del teclado para movimiento horizontal y vertical.

moveInput = new Vector3(Input.GetAxis("Horizontal"), 0f, Input.GetAxis("Vertical"));

moveInput = Vector3.ClampMagnitude(moveInput, 1f); // Limita la magnitud del vector a 1 para evitar movimientos más rápidos en diagonales.

if (Input.GetButton("Sprint")) // Si se mantiene presionado el botón de correr ("Sprint").

{

// Transforma la dirección local a dirección global y aplica la velocidad de correr.

moveInput = transform.TransformDirection(moveInput) \* runSpeed;

}

else

{

// Transforma la dirección local a dirección global y aplica la velocidad de caminar.

moveInput = transform.TransformDirection(moveInput) \* walkSpeed;

}

if (Input.GetButtonDown("Jump")) // Si se presiona el botón de saltar ("Jump").

{

// Calcula la velocidad vertical necesaria para alcanzar la altura del salto.

moveInput.y = Mathf.Sqrt(jumpHeight \* -2f \* gravityScale);

}

}

// Aplica la gravedad al jugador acumulándola en el eje Y.

moveInput.y += gravityScale \* Time.deltaTime;

// Mueve al jugador según el vector de movimiento calculado.

characterController.Move(moveInput \* Time.deltaTime);

}

private void Look()

{

// Obtiene la entrada del ratón para la rotación en los ejes X (horizontal) e Y (vertical).

rotationinput.x = Input.GetAxis("Mouse X") \* rotationSensibility \* Time.deltaTime; // Rotación horizontal.

rotationinput.y = Input.GetAxis("Mouse Y") \* rotationSensibility \* Time.deltaTime; // Rotación vertical.

// Acumula el ángulo vertical de la cámara.

cameraVerticalAngle = cameraVerticalAngle + rotationinput.y;

// Limita el ángulo vertical de la cámara entre -70 y 70 grados.

cameraVerticalAngle = Mathf.Clamp(cameraVerticalAngle, -70, 70);

// Rota el jugador horizontalmente en el eje Y según el movimiento del ratón.

transform.Rotate(Vector3.up \* rotationinput.x);

// Ajusta la rotación vertical de la cámara.

playerCamera.transform.localRotation = Quaternion.Euler(-cameraVerticalAngle, 0f, 0f);

}

}

**Texto

Descripción generada automáticamente**

**2-PlayerWeponController:**

public class WeaponController : MonoBehaviour

{

[Header("References")]

public Transform weaponMuzzle;

// Referencia al punto donde las balas (o efectos) saldrán del arma.

[Header("Info")]

public string weaponName;

// Nombre del arma para información en la interfaz o sistema de inventario.

public Sprite icon;

// Ícono que representa al arma (por ejemplo, en la interfaz del jugador).

[Header("General")]

public LayerMask hittableLayers;

// Capas que pueden ser impactadas por el arma (evita, por ejemplo, dispararse a sí mismo).

public GameObject bulletHolePrefab;

// Prefab que representa un agujero de bala que se crea al impactar un objeto.

[Header("Shoot Parameters")]

public float fireRange = 200;

// Rango máximo de disparo del arma.

public float recoilForce = 4f;

// Fuerza de retroceso al disparar.

public float fireRate = 0.6f;

// Intervalo mínimo entre disparos (en segundos).

public int maxAmmo = 8;

// Cantidad máxima de munición que el arma puede tener cargada.

[Header("Reload Parameters")]

public float reloadTime = 1.5f;

// Tiempo que tarda en recargar el arma.

public int currentAmmo { get; private set; }

// Munición actual del arma. Es privada para modificarla dentro de este script, pero accesible externamente.

private float lastTimeShoot = Mathf.NegativeInfinity;

// Momento en el que se realizó el último disparo. Inicia en negativo para permitir el primer disparo.

[Header("Sounds & Visuals")]

public GameObject flashEffect;

// Efecto visual que simula el fogonazo al disparar.

public GameObject owner { get; set; }

// Referencia al objeto que posee esta arma (normalmente el jugador).

private void Awake()

{

// Inicializa la munición actual al máximo al comenzar.

currentAmmo = maxAmmo;

}

private void Start()

{

// Método vacío, pero reservado para inicializaciones adicionales si es necesario.

}

private void Update()

{

// Detecta si el jugador presiona el botón de disparar ("Fire1").

if (Input.GetButtonDown("Fire1"))

{

TryShoot();

// Intenta disparar si es posible.

}

// Detecta si el jugador presiona la tecla de recargar (por defecto, "R").

if (Input.GetKeyDown(KeyCode.R))

{

StartCoroutine(Reload());

// Inicia la recarga del arma mediante una corrutina.

}

// Gradualmente devuelve la posición del arma a su posición inicial después del retroceso.

transform.localPosition = Vector3.Lerp(transform.localPosition, Vector3.zero, Time.deltaTime \* 5f);

}

private bool TryShoot()

{

// Comprueba si ha pasado suficiente tiempo desde el último disparo.

if (lastTimeShoot + fireRate < Time.time)

{

// Comprueba si hay munición disponible.

if (currentAmmo >= 1)

{

HandleShoot();

// Maneja el disparo (efectos, detección de impacto, etc.).

currentAmmo -= 1;

// Reduce la munición actual en 1.

return true;

// Indica que se realizó el disparo.

}

}

return false;

// Si no se cumplen las condiciones, no se dispara.

}

private void HandleShoot()

{

// Crea una instancia del efecto de fogonazo en el punto de salida del arma.

GameObject flashClone = Instantiate(flashEffect, weaponMuzzle.position, Quaternion.Euler(weaponMuzzle.forward), transform);

Destroy(flashClone, 1f);

// Destruye el efecto de fogonazo después de 1 segundo.

AddRecoil();

// Aplica el retroceso al arma.

// Realiza un raycast desde la cámara del jugador para detectar impactos.

RaycastHit[] hits;

hits = Physics.RaycastAll(

owner.GetComponent<PlayerController>().playerCamera.transform.position,

owner.GetComponent<PlayerController>().playerCamera.transform.forward,

fireRange,

hittableLayers

);

// Itera a través de todos los objetos impactados.

foreach (RaycastHit hit in hits)

{

Debug.Log(hit.collider.gameObject.name);

// Muestra en la consola el nombre del objeto impactado.

if (hit.collider.gameObject != owner)

{

// Si el impacto no es en el jugador que posee el arma, crea un agujero de bala.

GameObject bulletHoleClone = Instantiate(bulletHolePrefab, hit.point + hit.normal \* 0.001f, Quaternion.LookRotation(hit.normal));

Destroy(bulletHoleClone, 4f);

// El agujero de bala desaparece después de 4 segundos.

}

}

lastTimeShoot = Time.time;

// Registra el tiempo del disparo actual.

}

private void AddRecoil()

{

// Rota el arma hacia atrás según la fuerza de retroceso.

transform.Rotate(-recoilForce, 0f, 0f);

// Mueve el arma ligeramente hacia atrás para simular el retroceso.

transform.position = transform.position - transform.forward \* (recoilForce / 50f);

}

IEnumerator Reload()

{

// Muestra un mensaje de recarga en la consola.

Debug.Log("Recargando...");

// Espera el tiempo de recarga definido.

yield return new WaitForSeconds(reloadTime);

// Restaura la munición actual al máximo.

currentAmmo = maxAmmo;

// Muestra un mensaje indicando que la recarga ha terminado.

Debug.Log("Recargada");

// Aquí se pueden añadir animaciones de recarga (comentadas como "TODO").

}

}

**Texto

Descripción generada automáticamente**

**3-PlayerWeaponManager: Este se añade dentro del prefab del arma**

public class PlayerWeaponManager : MonoBehaviour

{

// Lista de armas iniciales que el jugador tendrá al comenzar el juego.

public List<WeaponController> startingWeapons = new List<WeaponController>();

[Header("References")]

public Transform weaponParentSocket;

// Referencia al transform donde se adjuntarán las armas.

public Transform defaultWeaponPosition;

// Posición por defecto de las armas (cuando no se están apuntando).

public Transform aimingPosition;

// Posición de las armas cuando el jugador está apuntando.

// Índice del arma actualmente activa. Se inicia en -1 (sin arma seleccionada).

public int activeWeaponIndex { get; private set; }

// Array que representa los "slots" para las armas (máximo 5 armas en este caso).

private WeaponController[] weaponSlots = new WeaponController[5];

// Start se ejecuta al inicio del juego.

void Start()

{

activeWeaponIndex = -1; // Inicialmente no hay ninguna arma seleccionada.

// Recorre la lista de armas iniciales y las agrega al inventario.

foreach (WeaponController startingWeapon in startingWeapons)

{

AddWeapon(startingWeapon); // Llama al método que agrega un arma.

}

}

// Update se ejecuta en cada frame.

void Update()

{

// Comprueba si el jugador presiona la tecla "1" para seleccionar el arma en el slot 0.

if (Input.GetKeyDown(KeyCode.Alpha1))

{

SwitchWeapon(0); // Cambia al arma en el slot 0.

}

}

// Método para cambiar de arma al especificar su índice.

private void SwitchWeapon(int p\_weaponIndex)

{

// Si el índice del arma seleccionada es diferente del actual y es válido (>= 0)...

if (p\_weaponIndex != activeWeaponIndex && p\_weaponIndex >= 0)

{

// Activa el arma del índice especificado.

weaponSlots[p\_weaponIndex].gameObject.SetActive(true);

// Actualiza el índice del arma activa.

activeWeaponIndex = p\_weaponIndex;

}

}

// Método para agregar un arma al inventario del jugador.

private void AddWeapon(WeaponController p\_weaponPrefab)

{

// Coloca el transform del "padre" de las armas en la posición por defecto.

weaponParentSocket.position = defaultWeaponPosition.position;

// Recorre el array de slots de armas para encontrar un espacio vacío.

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

{

if (weaponSlots[i] == null) // Si el slot está vacío...

{

// Instancia un clon del arma y lo asigna al padre (weaponParentSocket).

WeaponController weaponClone = Instantiate(p\_weaponPrefab, weaponParentSocket);

// Asigna al arma el jugador como propietario.

weaponClone.owner = gameObject;

// Desactiva el arma para que no sea visible inicialmente.

weaponClone.gameObject.SetActive(false);

// Almacena el arma clonada en el slot vacío.

weaponSlots[i] = weaponClone;

return; // Finaliza el método una vez que se agrega el arma.

}

}

}

}

**4-SWAY:**

public class Sway : MonoBehaviour

{

private Quaternion originLocalRotation;

// Almacena la rotación local original del objeto (arma, cámara, etc.) al inicio del juego.

private void Start()

{

// Guarda la rotación local inicial del objeto para utilizarla como punto de referencia durante el efecto de "sway".

originLocalRotation = transform.localRotation;

}

void Update()

{

// Llama a la función que actualiza el efecto de sway cada frame.

updateSway();

}

private void updateSway()

{

// Obtiene la entrada del mouse en los ejes X (horizontal) y Y (vertical).

float t\_xLookInput = Input.GetAxis("Mouse X");

float t\_yLookInput = Input.GetAxis("Mouse Y");

// Calcula el ajuste de la rotación en el eje horizontal (X).

// Se usa un ángulo negativo para que el movimiento del mouse tenga un efecto opuesto en la rotación.

Quaternion t\_xAngleAdjustment = Quaternion.AngleAxis(-t\_xLookInput \* 2f, Vector3.up);

// Calcula el ajuste de la rotación en el eje vertical (Y).

Quaternion t\_yAngleAdjustment = Quaternion.AngleAxis(t\_yLookInput \* 2f, Vector3.right);

// Calcula la rotación objetivo combinando la rotación original con los ajustes calculados.

Quaternion t\_targerRotation = originLocalRotation \* t\_xAngleAdjustment \* t\_yAngleAdjustment;

// Mueve suavemente hacia la rotación objetivo usando Lerp (Interpolación Lineal).

// La velocidad de movimiento se controla multiplicando por `Time.deltaTime \* 10f` para suavizar el movimiento.

transform.localRotation = Quaternion.Lerp(transform.localRotation, t\_targerRotation, Time.deltaTime \* 10f);

}

}

**Texto

Descripción generada automáticamente**

**EnemyController:**

using UnityEngine;

using UnityEngine.AI;

public class EnemyChase : MonoBehaviour

{

[Header("Enemy Settings")]

public float speed = 3.5f; // Velocidad del enemigo

public float stoppingDistance = 2f; // Distancia mínima para detenerse cerca del jugador

public float chaseRange = 10f; // Distancia máxima a la que el enemigo comienza a perseguir

[Header("References")]

public Transform player; // Referencia al transform del jugador

private NavMeshAgent navMeshAgent;

private bool isChasing = false;

private void Awake()

{

// Obtener el componente NavMeshAgent del enemigo

navMeshAgent = GetComponent<NavMeshAgent>();

}

private void Start()

{

// Configurar la velocidad y la distancia mínima en el NavMeshAgent

navMeshAgent.speed = speed;

navMeshAgent.stoppingDistance = stoppingDistance;

}

private void Update()

{

// Comprobar la distancia al jugador

float distanceToPlayer = Vector3.Distance(transform.position, player.position);

if (distanceToPlayer <= chaseRange)

{

StartChasing();

}

else

{

StopChasing();

}

if (isChasing && player != null)

{

navMeshAgent.SetDestination(player.position);

}

}

private void StartChasing()

{

isChasing = true;

navMeshAgent.isStopped = false;

}

private void StopChasing()

{

isChasing = false;

navMeshAgent.isStopped = true;

}

private void OnDrawGizmosSelected()

{

// Dibujar el rango de persecución en la escena

Gizmos.color = Color.red;

Gizmos.DrawWireSphere(transform.position, chaseRange);

}

}

**Captura de pantalla de computadora

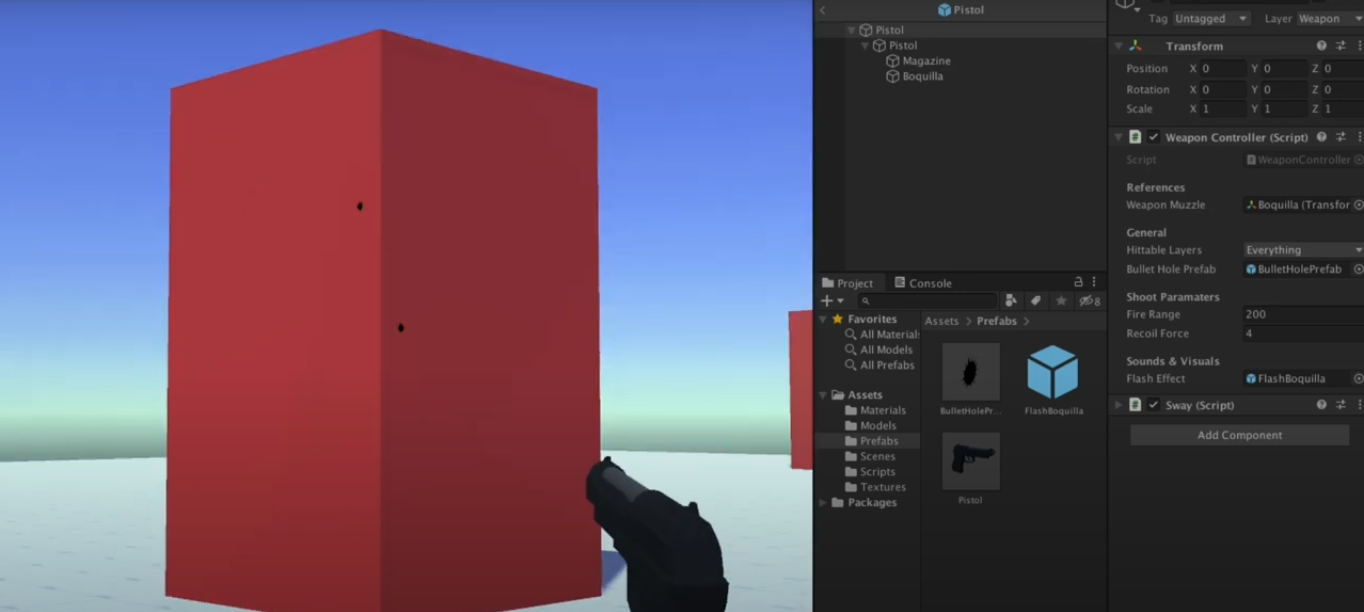
Descripción generada automáticamente**

Este Script se añade a los enemigos y te persiguen mientras que tu tienes que intentar escapar y matarlos. Este Script está hecho en su totalidad con ChatGPT

**PREFABS:**

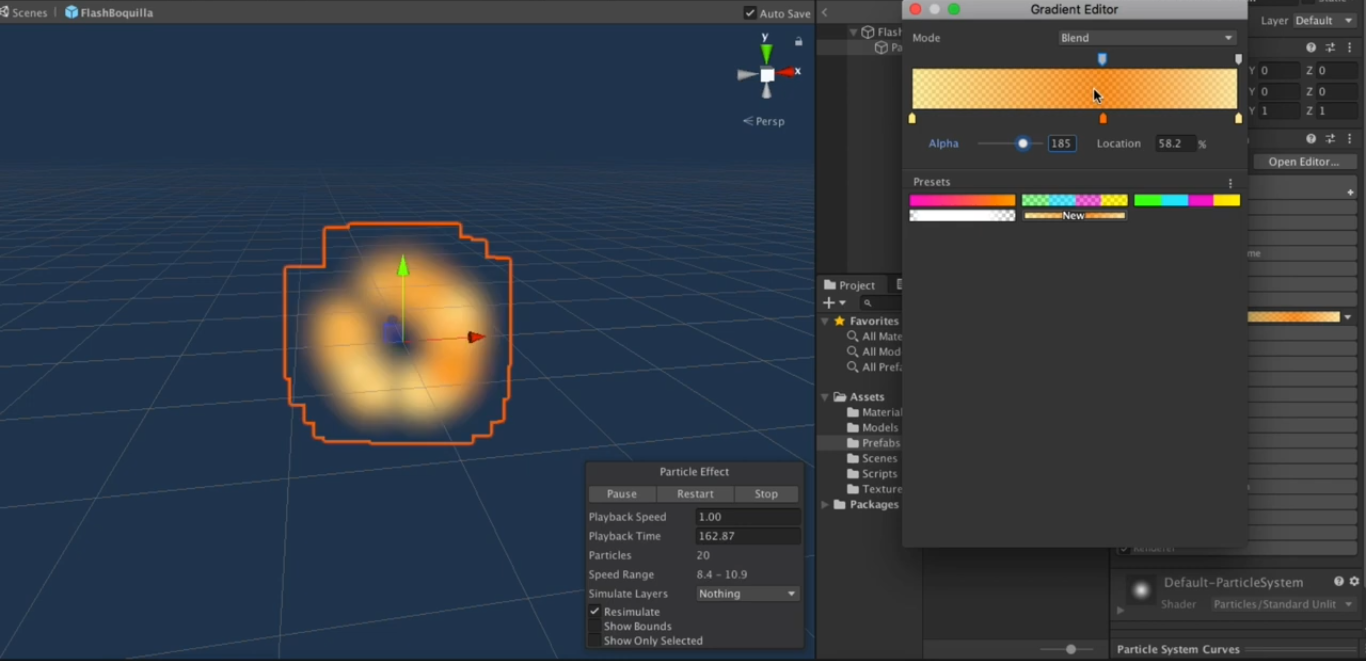
**Agujero de bala:** He creado un prefab, que es un agujero de bala para que cuando impacte con un asset, este agujero se quede marcado durante unos segundos:

Simplemente he importado una imagen de Google y la he ajustado para que se vea un pequeño agujero negro donde dispares. Solo puede haber un máximo de 5 agujeros en los assets ya que si los agujeros sobrepasan ese número se van borrando en orden, estos agujeros también desaparecen con el tiempo, a los pocos segundos.

****

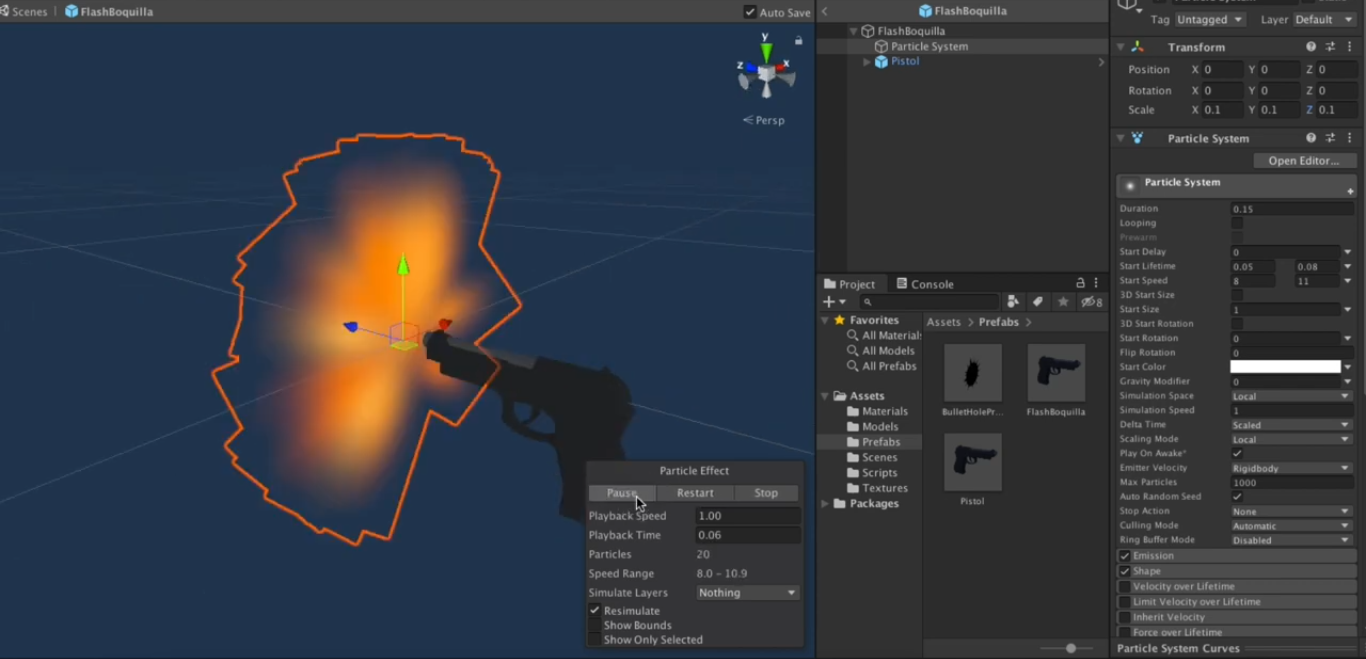
**Pistola:**

Tiene un emptiObject dentro del prefab que hace de boquilla, de donde salen las balas.

****

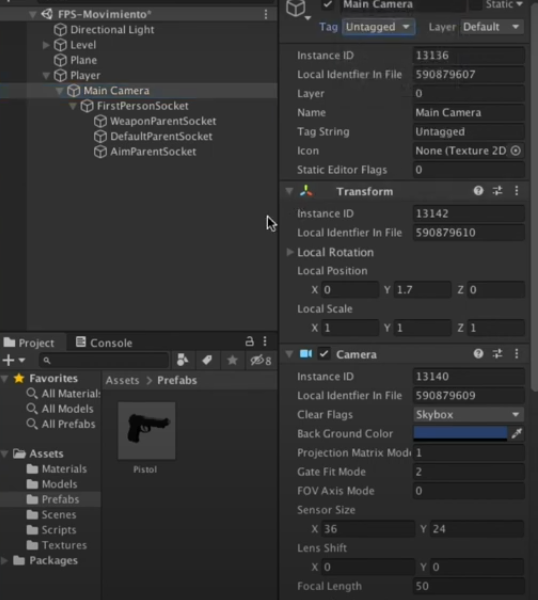
La pistola solo puede disparar 8 balas (Las que he puesto por predeterminado) y luego tienes que recargar. No puedes disparar de forma seguida ya que le he metido un pequeño tiempo entre disparo y disparo. También hay un tiempo para que el arma se recargue. El prefab(el arma) no aparece a la hora de iniciar el juego, en cuanto pulsas cualquier boton se crea una copia de esta arma (un clon) y aparece justo enfrente de la cámara. He generado en los scripts, un array de cantidad de armas en el juego, por predeterminado es solo una para que solo aparezca una pistola, pero se podría ampliar a la cantidad de armas que se quiera, simplemente habría que hacer un prefab con cada una de ellas y cambiar de arma durante el juego, el clon 1 será la pistola siempre y los siguientes clones serían los otros prefabs de otras armas.

En el script que controla todo lo que hace el arma, le he metido que cada vez que dispares el arma se mueva un poco hacia la dirección de la cámara( que como es un juego en primera persona es hacia atrás) para que de la sensación de que al disparar hay un pequeño retroceso.

****

Este prefab, genera partículas en un área muy reducida, de color anaranjado y que desaparecen enseguida tras disparar, da la sensación de que dispara. Este prefab se incorpora dentro de la MainCamera para que salga justo en frente de la pantalla pareciendo que somos nosotros los que tenemos la pistola en las manos.

El arma atraviesa todos los assets si no le pones a la MainCamera “untagged”. La jerarquía del contenido del proyecto funciona así:



**Animaciones:**

Tutorial en formato de guía para realizar las animaciones en Unity:

Para las animaciones, tenemos que ir a la siguiente página web: <https://www.mixamo.com/> y buscar las animaciones que queramos en nuestro proyecto. Busca y descarga todas las animaciones de un avatar, por ejemplo una animación del avatar caminando con un arma en la mano, otra del mismo avatar caminando marcha atrás con el arma, otra caminando hacia los lados, otra estando en reposo y otra corriendo. Descarga y descomprime todas las animaciones. Crea una carpeta nueva en el proyecto y mete todas las animaciones en ella. Selecciona las animaciones y cambia el Animation Type de General a Humanoid. Selecciona todas las animaciones y selecciona Avatar Definition y cámbialo a Copy From Other Avatar. Arrastra el esqueleto de avatar a Source para que todas las animaciones tengan el mismo esqueleto del avatar. Activar en todas las animaciones el Loop Time, el Bake into pose en todos los ejes y marcar Based upon como Original, esto te prevendrá de fallos dentro del juego. Una vez introducido el avatar, esté necesita un RigidBody y un Capsule Colider, que hay que ajustarlo al avatar y congelamos la rotación en x, y, z para que no atraviese el suelo cuando choque con un objeto. Después de que nuestro personaje se mueva por el plano, creamos un Animation Controller, ‘Animation player’. Lo abrimos y seleccionamos nuestro Blend Tree ( que combina una animación con otra para que ésta no cambie bruscamente dándole realismo ), hay que cambiar el tipo de 1D a 2D Freeform Directional, dentro parámetros creamos los parámetros X e Y que marcarán la posición del avatar y la fuerza. Por ejemplo, si el avatar está en la posición 0,0 la animación que hará será la de Idle (estar quieto respirando) y si el avatar camina hacia atrás la posición del eje y será negativa entonces cogerá la animación del avatar caminando de espaldas, si la posición del eje y esta cambiando rápidamente en positivo significa que el avatar está corriendo y cambiará la animación a la que corre. Tienes que crear Motions y asignarle a cada una la animación y la posición donde se da para que cambie la animación. Para que funcionen hay que hacer referencia al animator en el código (en el PlayerController)

**Aplicando las Animaciones:**

Captura de pantalla de computadora

Descripción generada automáticamente

Pantalla de celular con imagen de la pantalla de una computadora

Descripción generada automáticamente

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica

Descripción generada automáticamente

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media para que no se caiga cuando choque con un asset

Captura de pantalla de un celular

Descripción generada automáticamente

Interfaz de usuario gráfica

Descripción generada automáticamente

Captura de pantalla de computadora

Descripción generada automáticamente

Captura de pantalla de computadora

Descripción generada automáticamente

Así queda nuestro personaje totalmente animado, después de animarlo hay que introducir los enemigos, los prefabs y el resto de assets para tener completado el juego.