Medieval Hell Shooter

Evidencia 2
Actividad Integradora Gráficas

Modelación de sistemas multiagentes
con gráficas computacionales

Alejandra Arredondo

August 29, 2025

Introducción

Durante estas tres semanas, hemos explorado el entorno de desarrollo de videojuegos en Unity. A través de diversos laboratorios, aprendimos conceptos fundamentales que resultaron de gran utilidad para el desarrollo de la actividad integradora de medio término. Entre estos temas destacan el manejo de objetos, la importación de librerías y la gestión de la cámara, habilidades que me permitieron avanzar de manera más eficiente en el desarrollo de mi videojuego.

Para la actividad integradora que se nos planteó, decidí enfocar la temática de mi Bullet Hell Shooter en un ambiente medieval. Para ello, seleccioné un escudo medieval de los assets importados, que sería el objeto principal a lanzar dentro del juego.

Nivel Escogido

Nivel: **FÁCIL**

Modo de Juego

Debido a que seleccioné el nivel fácil, la implementación corresponde a una versión más básica del juego, pero conserva los elementos esenciales que caracterizan a un **Bullet Hell Shooter**, adaptados a la temática elegida. En esta versión, no se contempla aún la interacción directa de un jugador que controle un personaje en pantalla, pero sí se incluyen distintos aspectos que pueden ser ajustados por el usuario a través del **Inspector de Unity**, lo cual permite experimentar con la dinámica del juego.

El juego está diseñado para ofrecer flexibilidad al jugador, permitiéndole escoger el objeto que desea utilizar como bala, siempre y cuando este se encuentre configurado como prefab. Además, es posible personalizar distintos parámetros del sistema de disparo como:

- La velocidad de la bala
- El número de direcciones de disparo que tendrá el spawner
- El tipo de movimiento que seguirá al disparar
- El intervalo de tiempo entre cada lanzamiento.

Implementación

- **Técnica de las balas**

  Para organizar de manera eficiente la lógica del proyecto, creé una carpeta llamada `Scripts` dentro de la vista de `Projects`. En esta carpeta desarrollé un archivo basado en `MonoBehaviour`, el cual concentra toda la lógica relacionada con el comportamiento de las balas (representadas en este caso por los escudos).

  Este script se encarga de manejar aspectos fundamentales como:

  - La velocidad de cada bala.

- Su posición inicial al ser lanzada.
- Una referencia al objeto desde el cual fue disparada.
- El tiempo transcurrido desde su creación para calcular su movimiento de manera precisa.

En esta clase, los objetos se desplazan de manera recta gracias a la función `Movement`, la cual calcula la posición a la que debe moverse cada bala utilizando `transform.right.x` multiplicado por la `speed` y el `timer`. Esto permite que las balas tengan un movimiento lineal desde su creación.

```
Vector2 Movement(float timer){
    float x = timer * speed * transform.right.x;
    float y = timer * speed * transform.right.y;
    return new Vector2(x+spawnPoint.x, y+spawnPoint.y);
}
```

El vector `transform.right` apunta siempre hacia la "derecha" local del objeto, según su rotación inicial. Multiplicarlo por `speed` y `timer` calcula el desplazamiento acumulado a lo largo de esa dirección.

A diferencia de lo que podría parecer, este desplazamiento no depende del eje X o Y, sino de la orientación del objeto. Por ejemplo:

- Si el objeto tiene rotación 0°, `transform.right` coincide con el eje X global y el movimiento es horizontal.
- Si el objeto está rotado 90°, `transform.right` apunta hacia el eje Y global y el movimiento es vertical.

De esta manera, la bala siempre se mueve en línea recta siguiendo la dirección "derecha" del objeto desde su creación, sin importar su orientación inicial.

Cuando las balas alcanzan ciertas coordenadas definidas en el método `Update`, se realiza una referencia al `Spawner` para decrementar el contador de balas activas en pantalla y, finalmente, se destruye el objeto creado.

```
void Update()
{
    if (transform.position.x < -17 || transform.position.x > 17
        || transform.position.y < -8.5 || transform.position.y > 8.5){
            spawner.BulletCleanup();
            Destroy(this.gameObject);
        }
    timer += Time.deltaTime;
    transform.position = Movement(timer); /
}
```

- **Técnica de cada patrón**

  Los patrones de disparo implementados en esta actividad son completamente personalizables gracias a las variables configuradas en el sistema. Esto permite ajustar la dinámica de las balas de acuerdo con las preferencias del usuario y experimentar con diferentes combinaciones para obtener resultados variados.

  Para el diseño de dichos patrones, tomé como referencia diversos materiales audiovisuales, principalmente tutoriales y ejemplos encontrados en YouTube y TikTok, los cuales me sirvieron de guía para recrear y adaptar los estilos de disparo al contexto de mi propio juego.

  - **Patrón Straight:**

    Este patrón se refleja en el movimiento de las balas definido en su clase. Cuando se dispara cualquier bala desde el `Spawner`, se crean múltiples objetos. Lo que hace único al método `FireAllAngles` en la clase `Skull Spawner` es que, a partir del valor de `number arms` establecido en el Inspector de Unity, se generan varias balas distribuidas en los ángulos correspondientes al instanciar cada objeto.

```
void FireAllAngles(){
    for (int i = 0; i < number_arms; i++){
        float bulletAngle = i * (360 / number_arms);
        GameObject bulletObject = Instantiate(bullet,
            transform.position, Quaternion.Euler(0,0, bulletAngle));
        bulletObject.GetComponent<Shield_Bullet>().speed = speed;
        bulletObject.GetComponent<Shield_Bullet>().spawner = this;
        numberBullets++;
        bulletCountText.text = "Contador de balas: " + numberBullets;
    }
}
```

    Cada bala se instancia con una rotación específica determinada por `bulletAngle`, de manera que queden distribuidas uniformemente alrededor del `Spawner`. Además, se asignan la velocidad y la referencia al `Spawner` correspondiente, y se actualiza el contador de balas activas en pantalla.

  - **Patrón Spin:**

    Este patrón se implementa haciendo que el spawner realice una rotación continua en el eje Z, lo que genera un desplazamiento giratorio. La rotación se realiza a una velocidad de 70 unidades por segundo mediante la siguiente instrucción:

```
transform.Rotate(0, 0, 70f * Time.deltaTime);
```

Además, cuando se instancia una bala, esta hereda la rotación actual del spawner, asegurando que su movimiento inicial siga la orientación giratoria del objeto que la dispara:

```
void FireSpin(){
  GameObject spawnedBullet = Instantiate(bullet,
    transform.position, Quaternion.Euler(0,0,0));
  spawnedBullet.GetComponent<Shield_Bullet>().speed = speed;
  spawnedBullet.GetComponent<Shield_Bullet>().spawner = this;
  spawnedBullet.transform.rotation = transform.rotation;

  numberBullets += 1;
  bulletCountText.text = "Contador de balas: " + numberBullets;
}
```

De esta manera, cada bala mantiene la rotación del spawner en el momento de ser disparada, permitiendo que el patrón Spin se refleje correctamente en el movimiento de todas las balas generadas.

– **Patron Snake:**

Este patrón se realizá mediante una variable auxiliar que me ayuda a llevar el tiempo y poder oscilar entre el angulo de rotaciones que hace es spawner

```
if (rotationTimer<1){
  transform.Rotate(0, 0, 70f * Time.deltaTime);
  rotationTimer += Time.deltaTime;
} else {
  transform.Rotate(0, 0, -70f * Time.deltaTime);
  rotationTimer += Time.deltaTime;
  if (rotationTimer>2) rotationTimer=0;
}
```

Al momento de instanciar el objeto, se utiliza el mismo componente con una ligera modificación: en la rotación se asigna el ángulo de la bala sumado al ángulo actual del spawner, convirtiéndolo a un valor entero. Esto permite que las balas se dispersen en todas las direcciones de disparo definidas por el patrón. Si no se aplicara esta suma, todas las balas saldrían con el mismo ángulo inicial, lo que generaría un comportamiento similar al patrón Spin, pero concentrando todas las balas en una única dirección en lugar de distribuirlos correctamente.

```
GameObject bulletObject = Instantiate(
  bullet,
  transform.position,
  Quaternion.Euler(0,0,bulletAngle+transform.rotation.eulerAngles.z));
```

- **Justificación de por qué cada patrón es diferentes**

  Cada patrón es diferente debido a que sus formas de oscilar son distintas. Si bien existe la posibilidad de ajustar la velocidad, el período de lanzamiento de las balas o el número de direcciones de disparo que tendrá el spawner, los movimientos que realiza cada patrón son únicos. Por ejemplo, el patrón **Snake** se caracteriza por un cambio continuo de ángulo que hace que las balas sigan una trayectoria ondulada, mientras que el patrón **Straight** lanza las balas en línea recta, y el patrón **Spin** combina un movimiento rotatorio constante del spawner para dispersar las balas de manera circular. Estas diferencias permiten ofrecer al jugador diversos patrones visualmente distintos y mecánicamente únicos.

Reflexión

Durante esta tercera semana, me encontré con retos que al principio pensé que serían un impedimento para realizar esta actividad, pero con el tiempo logré sobrellevarlos y organizarme para completar todo en tiempo y forma, sin necesidad de desvelarme ni experimentar estrés constante. Algunos de los desafíos enfrentados fueron:

- Siempre que inicio una actividad, siento un poco de ansiedad porque no sé cómo empezar. Sin embargo, después de buscar tutoriales en YouTube, revisar la documentación de Unity e incluso seleccionar los assets adecuados para la temática que quería implementar, logré superar este obstáculo que en ocasiones me generaba algo de miedo.
- Desarrollar el mapa en Unity, componente por componente, asegurando que las capas fueran correctas y que todo se viera visualmente agradable, fue un desafío. Los bloques los agregué manualmente uno por uno hasta completar el tablero y asegurarme de que todo estuviera en su lugar.
- Entender la lógica detrás de mi idea para implementar el videojuego, luego de investigar e incluso fracasar en algunos intentos, fue algo interesante pero desafiante, ya que nunca había trabajado con un motor como Unity.
- Establecer mi spawner fue un reto, porque al asignarlo a mi boss este giraba debido a la rotación necesaria para lanzar los escudos. Por ello, decidí tener un spawner oculto detrás del boss, haciendo que el boss permanezca quieto y que únicamente el spawner realice la rotación.

Video de demostración y reporte

- **Video demostrativo** En este video se puede encontrar una demostración de como funciona el proyecto que implementé
- **Link del repositorio** En esta URL se puede encontrar el repositorio al cual se realizaron todos los commits con las modificaciones implementadas

Futuras Mejoras

Para futuras mejoras, se podrían implementar los siguientes aspectos:

- Movimiento de un personaje controlado por el jugador.

- Objetos destructibles por detección de colisiones.
- Patrones de disparo más complejos.

Referencias

- **Assets**
  - Dungeon Tale - Fantasy RPG Sprites FX Tileset. (2025). Unity Asset Store. Link
  - Fake Shadow For 2D. (2025). Unity Asset Store. linkcolor Link
  - Free - Casual & Relaxing Game Music Pack. (2025). Unity Asset Store. Link
- **Videos**
  - Video 1
  - Video 2
  - Video 3
- **Documentation**
  - Unity Technologies. (2025). Unity - Scripting API: Time.deltaTime. Link
  - Unity Technologies. (2022). Cannot implicitly convert type “string” to “TMPro.TextMeshProUGUI.” Link
  - Destroy an instantiated prefab Link
  - Instantiated a prefab Link