

Modelación de sistemas multiagentes con gráficas computacionales

Evidencia 1. Actividad Integradora de Gráficas

Estudiante:Aksel Deneken Maldonado A01711966

Profesores: Alejandro Fernández Vilchis, Denisse Lizbeth Maldonado Flores, Pedro Oscar Pérez Murueta

TC2007B

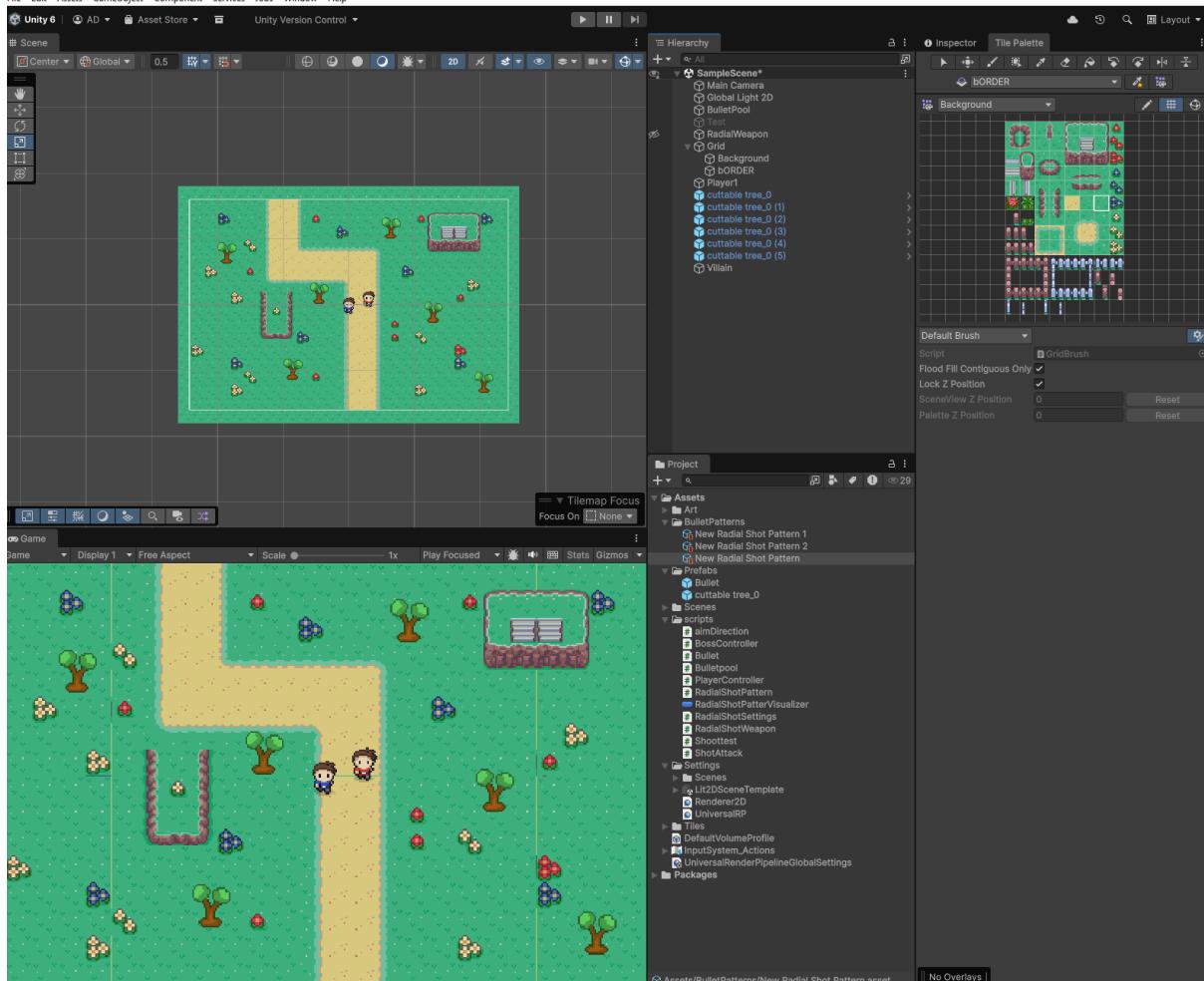
Grupo 602

25/11/2025

## Reporte del Proyecto – Bullet Hell en Unity (Fácil)

### Proceso de construcción del proyecto

Inicié el proyecto construyendo el entorno del juego con assets tipo pixel-art inspirados en Pokemon y armé el mapa usando Tilemaps para separar el piso de los jugadores.



Después implementé el movimiento del jugador con teclado (WASD) usando el Input System. El movimiento se aplica con Rigidbody2D usando MovePosition dentro de FixedUpdate para que sea estable con físicas y colisiones.

```

private void Update()
{
    if (Keyboard.current == null) return;

    move = Vector2.zero;

    if (Keyboard.current.wKey.isPressed) move.y += 1;
    if (Keyboard.current.sKey.isPressed) move.y -= 1;
    if (Keyboard.current.dKey.isPressed) move.x += 1;
    if (Keyboard.current.aKey.isPressed) move.x -= 1;

    if (move.sqrMagnitude > 1f) move.Normalize(); // diagonal parejito
}

private void FixedUpdate()
{
    rb.MovePosition(rb.position + move * moveSpeed * Time.fixedDeltaTime);
}

```

Finalmente implementé el sistema de balas usando un Object Pool para optimizar el performance. En vez de instanciar y destruir balas en cada disparo, reutilizo objetos del pool y solo los activo cuando se necesitan. Esto permite que el boss dispare muchas balas sin caídas fuertes de FPS.

```

public Bullet RequestBullet()
{
    for (int i = 0; i < bulletPool.Count; i++)
    {
        if (!bulletPool[i].gameObject.activeSelf)
        {
            bulletPool[i].gameObject.SetActive(true);
            return bulletPool[i];
        }
    }
    AddBulletsToPool(1);
    bulletPool[bulletPool.Count - 1].gameObject.SetActive(true);
    return bulletPool[bulletPool.Count - 1];
}

```

### Explicación Técnica

Los disparos del boss se basan en una arquitectura modular con cuatro piezas clave que son: Bulletpool, ShotAttack, RadialShotSettings/RadialShotPattern y una extensión para rotar vectores. El Bulletpool mantiene una lista de balas desactivadas y al disparar se pide una bala con RequestBullet.

ShotAttack es la clase que centraliza la lógica de disparo. Con el método SimpleShot toma una bala del pool, la coloca en el origen y le asigna una velocidad; RadialShot calcula el ángulo entre balas (360 / numberOfBullets), genera direcciones rotando Vector2.up y dispara todas las balas necesarias.

```
using UnityEngine;

public class ShotAttack
{
    public static void SimpleShot (Vector2 origin, Vector2 velocity)
    {
        Bullet bullet = Bulletpool.Instance.RequestBullet();
        bullet.transform.position = origin;
        bullet.Velocity = velocity;
    }

    public static void RadialShot (Vector2 origin, RadialShotSettings settings)
    {
        float angleStep = 360f / settings.numberOfBullets;
        Vector2 baseDirection = Vector2.up;

        if (settings.angleOffset != 0f || settings.PhaseOffset != 0f)
        {
            baseDirection = Vector2.up.Rotate(settings.angleOffset + (settings.PhaseOffset * angleStep));
        }

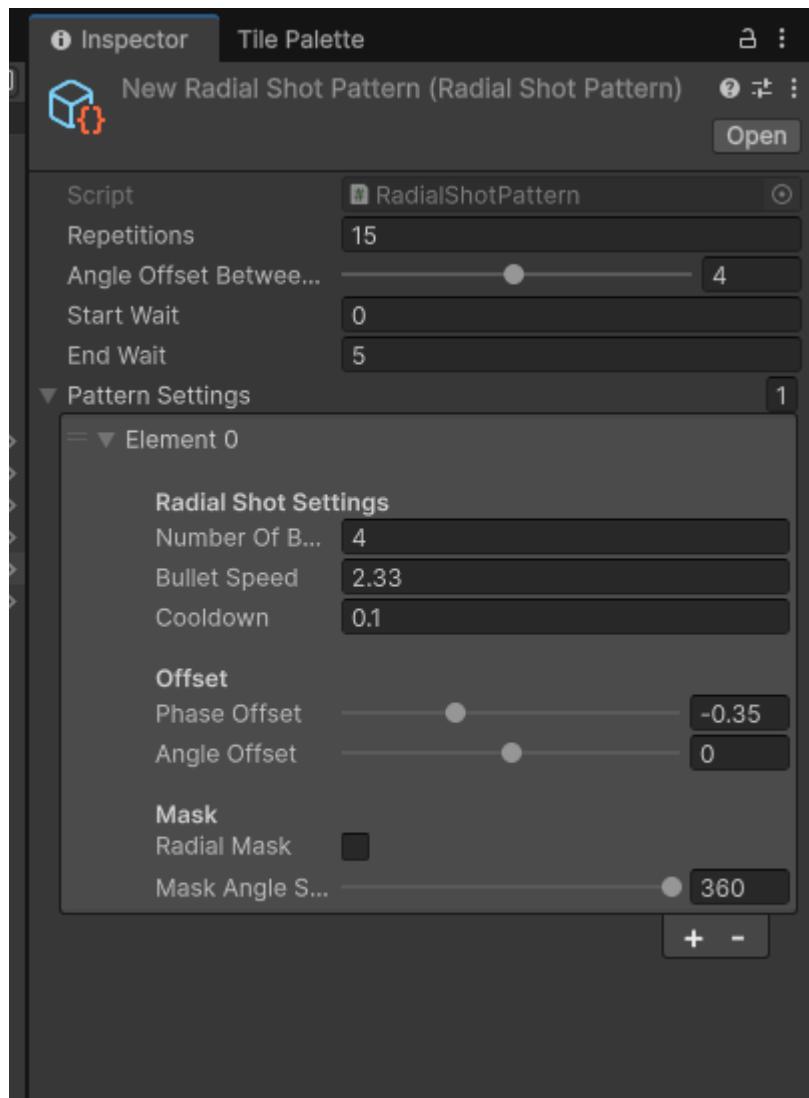
        for (int i = 0; i < settings.numberOfBullets; i++)
        {
            float bulletAngle = i * angleStep;

            if(settings.RadialMask && bulletAngle > settings.MaskAngleStart)
                break;

            Vector2 bulletDirection = baseDirection.Rotate(bulletAngle);
            SimpleShot(origin, bulletDirection * settings.bulletSpeed);
        }
    }
}
```

Para rotar direcciones uso una extensión Vector2.Rotate basada en Quaternion.AngleAxis, lo que simplifica la construcción de patrones radiales de forma limpia.

Los patrones del boss están encapsulados como ScriptableObjects (RadialShotPattern) que guardan Repetitions, AngleOffsetBetweenLaps, startWait/endWait y un arreglo de RadialShotSettings. Esto permite editar ataques desde el Inspector sin tocar el código.



En ejecución, el patrón se dispara por medio de una coroutine que itera laps, aplica rotación acumulada entre laps y dispara cada elemento de PatternSettings respetando el Cooldown.

```

private IEnumerator ExecuteRadialShotPattern(RadialShotPattern pattern)
{
    isShooting = true;
    int lap = 0;
    Vector2 aimDirection = transform.up;
    Vector2 center = transform.position;

    yield return new WaitForSeconds(pattern.startWait);

    while (lap < pattern.Repetitions)
    {
        if (lap > 0 && pattern.AngleOffsetBetweenLaps != 0f)
        {
            aimDirection = aimDirection.Rotate(pattern.AngleOffsetBetweenLaps);
        }

        for (int i = 0; i < pattern.PatternSettings.Length; i++)
        {
            ShotAttack.RadialShot(center, pattern.PatternSettings[i]);
            yield return new WaitForSeconds(pattern.PatternSettings[i].Cooldown);
        }
        lap++;
    }
    yield return new WaitForSeconds(pattern.endWait);
    isShooting = false;
}

```

Para cumplir el requerimiento del contador de balas activas en pantalla, implementé un conteo global usando una variable estática: Bullet.ActiveCount. Esta se incrementa automáticamente en OnEnable() y se decrementa en OnDisable(), lo cual funciona perfecto con el pooling porque cada vez que una bala se activa/desactiva el contador se actualiza de inmediato.

```

using UnityEngine;

public class Bullet : MonoBehaviour
{
    private const float maxLifeSpan = 10f;
    private float lifeSpan = 0f;
    public Vector2 Velocity;

    // Conteo global de balas activas
    public static int ActiveCount { get; private set; }

    private void OnEnable()
    {
        lifeSpan = 0f;
        ActiveCount++;
    }

    private void OnDisable()
    {
        if (ActiveCount > 0)
            ActiveCount--;
    }

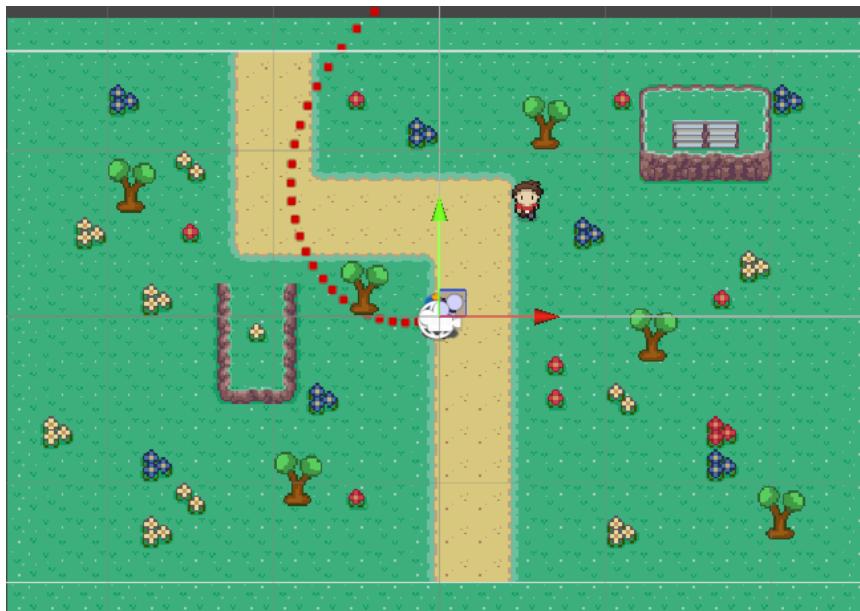
    void Update()
    {
        transform.position += (Vector3)(Velocity * Time.deltaTime);
        lifeSpan += Time.deltaTime;
        if (lifeSpan >= maxLifeSpan)
        {
            Disable();
        }
    }

    void Disable()
    {
        lifeSpan = 0f;
        gameObject.SetActive(false); // aquí se dispara OnDisable
    }
}

```

### Patrón 1 – “Espiral”

En este patrón se genera una espiral de una sola línea. El patrón se repite múltiples veces y cada repetición aplica un offset angular entre laps, por eso la espiral rota. Este patrón obliga al jugador a reposicionarse porque los huecos cambian constantemente.



El disparo se calcula con  $\text{angleStep} = 360 / \text{numberOfBullets}$  y cada bala sale al rotar un vector base (`Vector2.up`), y su velocidad depende de `bulletSpeed`.

#### Patrón 2 – “Doble ráfaga”

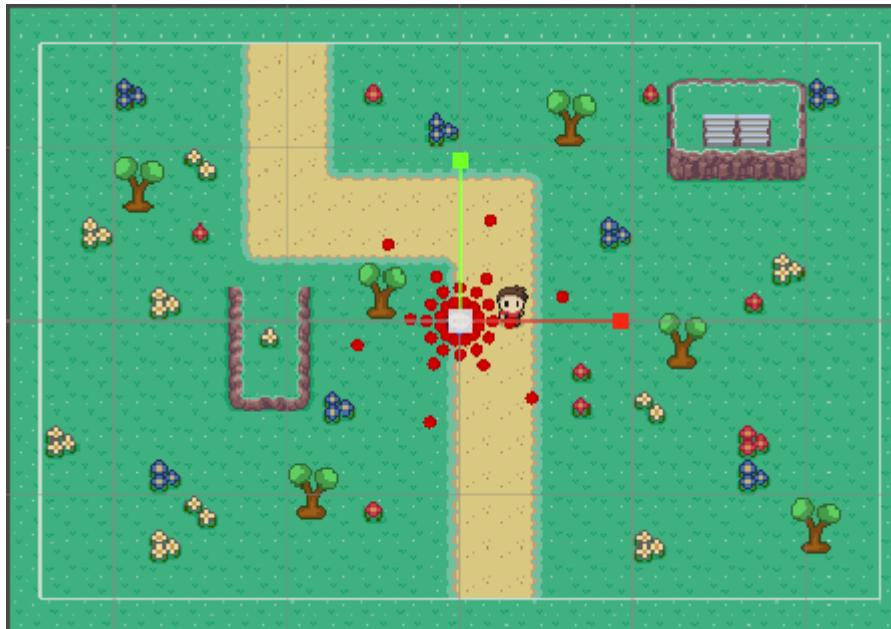
Este patrón es mecánicamente distinto porque combina dos disparos en secuencia dentro del mismo `PatternSettings`, primero una ráfaga amplia (por ejemplo 15 balas) que llena el espacio, y después una segunda ráfaga más pequeña pero rápida (por ejemplo 2 balas con mayor `bulletSpeed`) con un `PhaseOffset` diferente para que salga desfasada respecto al primer anillo. Esto funciona como “cobertura + castigo”: el primer anillo condiciona dónde se puede mover el jugador y la segunda ráfaga castiga zonas que se ven seguras.



Aunque comparte la idea radial, su diferencia mecánica es que el patrón no es una sola explosión uniforme sino que se construye por alternancia de offsets entre disparos.

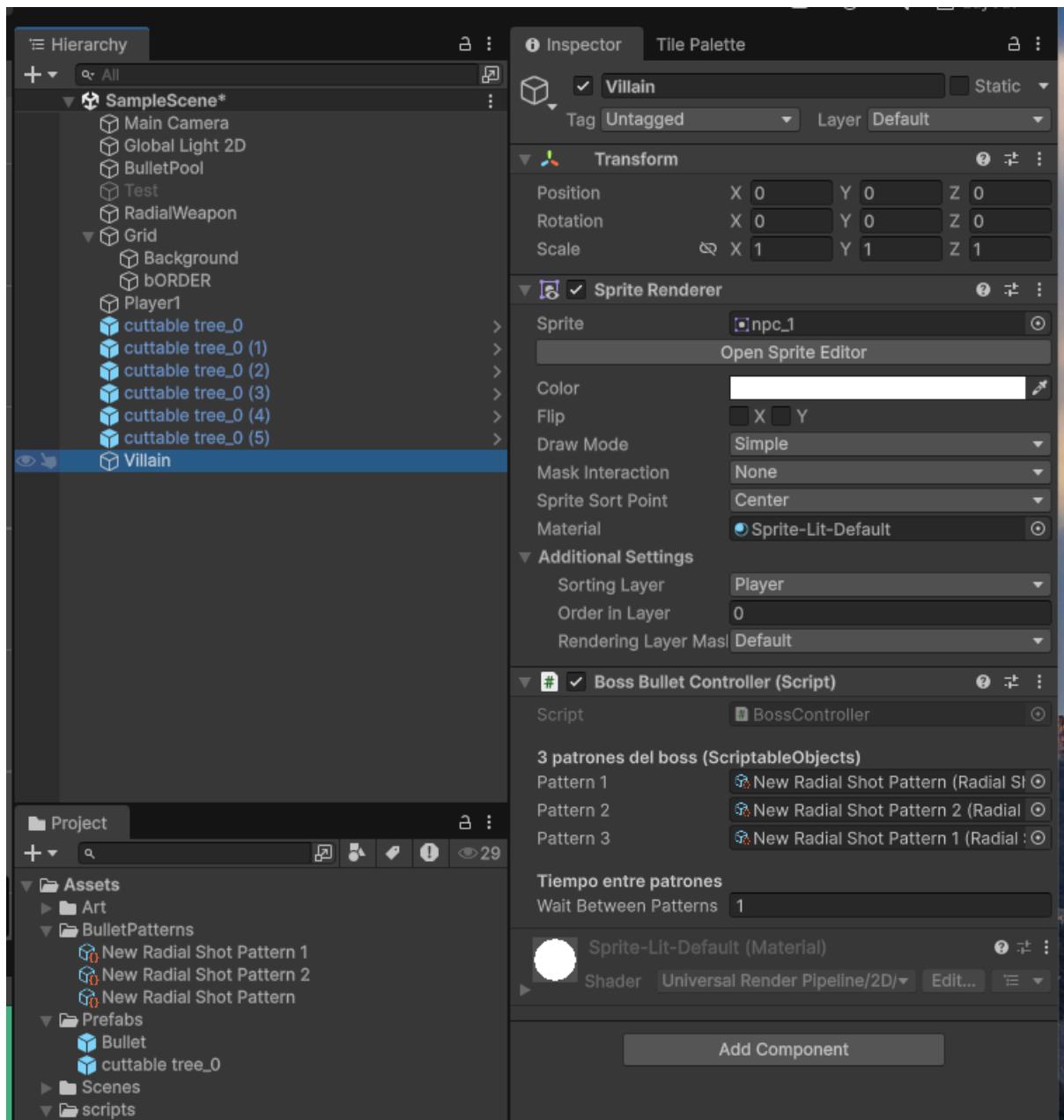
#### Patrón 3 – “Stream”

Para este disparo se genera un stream de balas que gira progresivamente. En mi implementación no necesito una función totalmente nueva, logro el efecto haciendo que el patrón dispare muy pocas balas por ráfaga con un Cooldown corto, y usando AngleOffsetBetweenLaps para que cada disparo salga con un ángulo ligeramente diferente al anterior. Como el ángulo se va acumulando con el tiempo, el resultado visual es una como espiral.



La clave mecánica es que la geometría nace del tiempo (acumulación del ángulo) y no de un disparo simultáneo. Esto es posible porque RadialShot permite ajustar angleOffset y PhaseOffset para rotar la dirección base antes de disparar.

Los tres patrones son mecánicamente distintos, no solo cambios de velocidad y dirección. Esto se debe a que en mi sistema el patrón depende del número de balas por ráfaga (numberOfBullets), composición de ráfagas (múltiples elementos en PatternSettings), offsets (PhaseOffset / angleOffset) y rotación acumulada por repetición (AngleOffsetBetweenLaps), además de los tiempos de Cooldown y Repetitions.



## Retos y Aprendizaje

Implementar colisiones del mapa, el personaje se sale del área jugable, así que intenté asegurar que el Tilemap de borde tuviera collider funcional y se comportara como límite real del escenario pero, no lo logré. También aprendí que mover el jugador con Rigidbody2D (MovePosition) hace el movimiento estable y evita comportamientos raros.

Orden de render: otro detalle importante fue el orden visual; si el tilemap o la capa superior está por encima, las balas pueden verse “atrás” del mapa, lo que se resuelve ajustando Sorting Layer en los SpriteRenderers (especialmente el prefab de bala).

## Conclusión

Este proyecto me ayudó a integrar varias herramientas de Unity en un solo juego. Tilemaps

y colisiones, físicas 2D, control por teclado, y sobre todo el diseño de patrones de disparo con lógica matemática. También aprendí a optimizar usando Object Pooling para poder mantener muchos proyectiles en pantalla sin sacrificar rendimiento, y a diseñar ataques de boss editables desde el Inspector usando ScriptableObjects, lo cual hace el sistema escalable para agregar más patrones en el futuro.

URLs:

Repositorio: <https://github.com/akseldeneken/Actividad-Integradora-de-Gr-ficas>

Video: <https://www.loom.com/share/f4e6766a9dda42a3bac2602efaf8c5df>