JAIRO ARANDA - GAME DEVELOPER

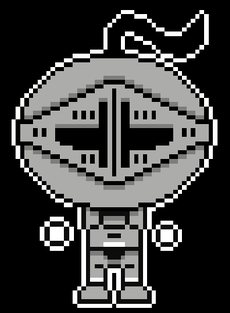
# DUNGEON SURVIVOR

DOCUMENTO DEL PROYECTO

ROGUELIKE DE ACCIÓN

# PROGRAMA

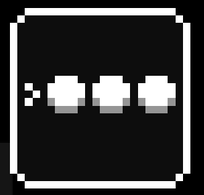
## TEMÁTICAS ABORDADAS



PROPUESTA INICIAL

SCRIPTABLE OBJECTS

HERENCIAS Y OBJECT POOLS



INTERFACES Y EVENTOS

# PROPUESTA INICIAL

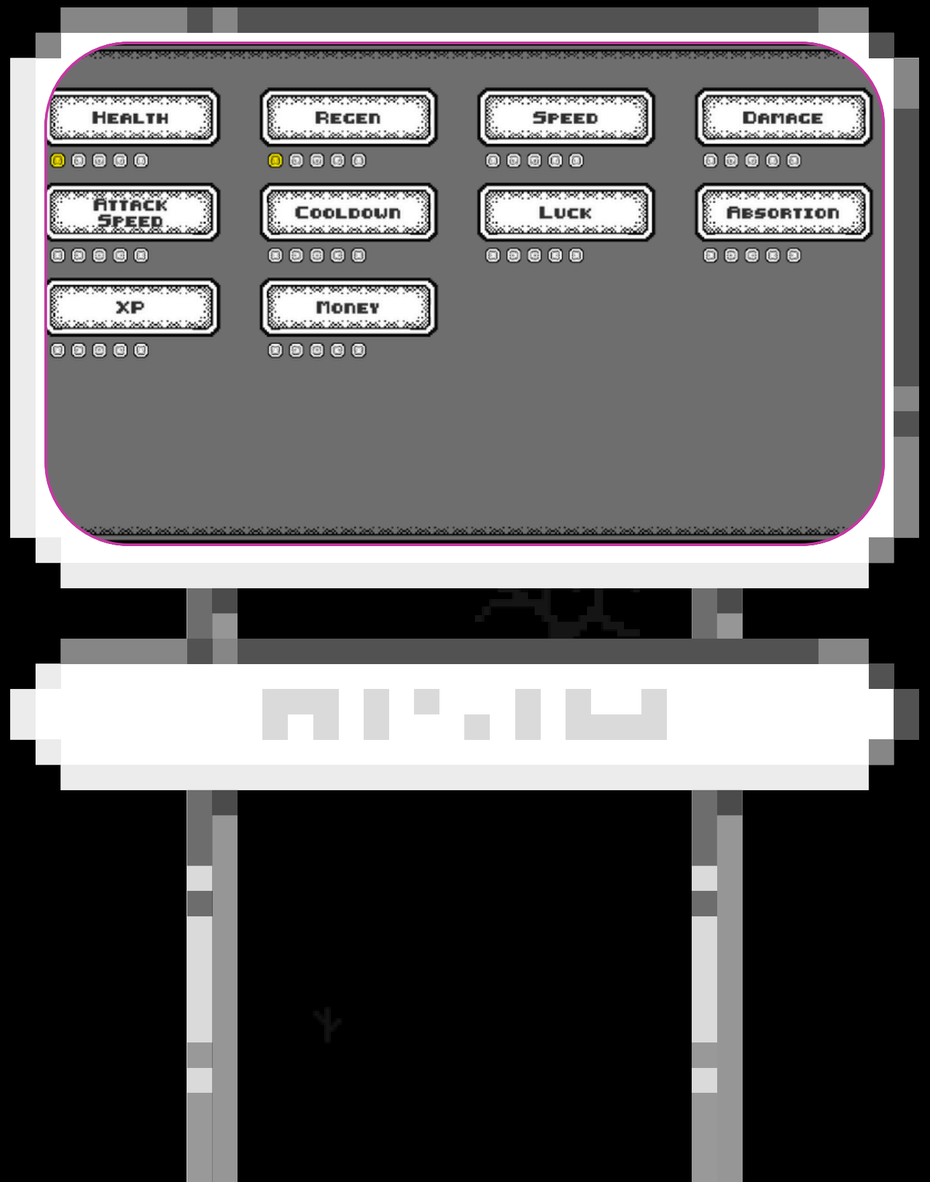
- EL JUGADOR AVANZA POR PISOS LLENOS DE ENEMIGOS, GANANDO EXPERIENCIA PARA SUBIR DE NIVEL Y MEJORAR UNA ESTADÍSTICA BASE.

- A MEDIDA QUE PROGRESA, ENCUENTRA COFRES CON LOOT VARIADO QUE LE AYUDA EN SU AVENTURA.

# PROPUESTA INICIAL

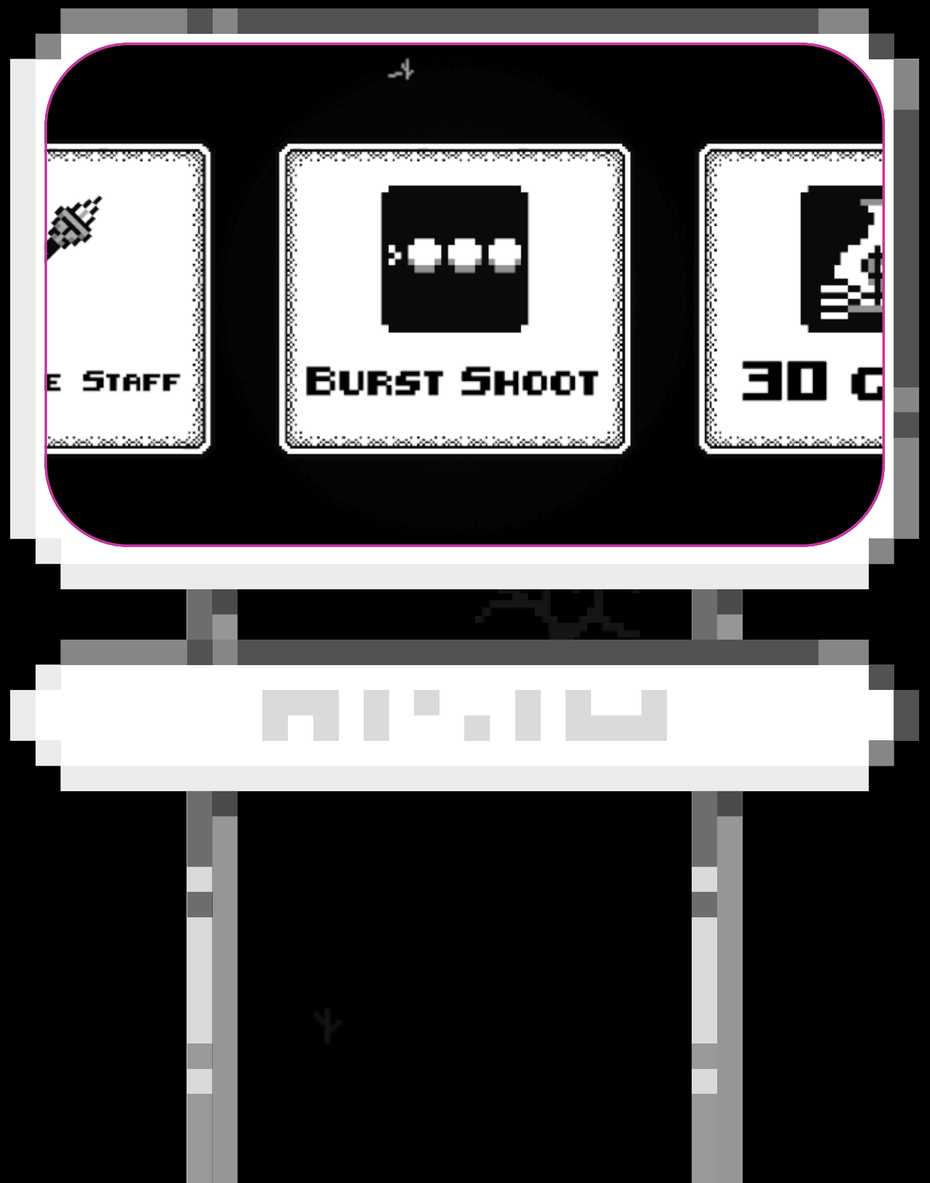
- SI MUERE, EL DINERO ACUMULADO SE GUARDA Y PUEDE GASTARSE EN EL MENÚ PRINCIPAL PARA MEJORAR PERMANENTEMENTE LAS ESTADÍSTICAS DEL PERSONAJE.

- CADA PARTIDA ES UN NUEVO INTENTO DE LLEGAR MÁS LEJOS Y ENFRENTAR ENEMIGOS MÁS FUERTES.



# PROPUESTA INICIAL

- LA IDEA FINAL NO ERA HACER UN JUEGO COMPLETO, ES HACER UNA BASE LA CUAL SE PUEDA EXPANDIR, AÑADIENDO ENEMIGOS, HABILIDADES, ARMAS ETC. DE MANERA SENCILLA SIN NECESIDAD DE EDITAR EL CODIGO BASE



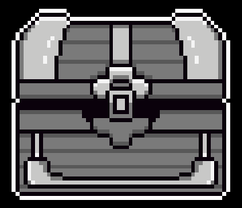
# SCRIPTABLE OBJECTS

## JUGADORES

Permiten centralizar y gestionar las estadísticas del jugador sin necesidad de instancias en escena, mejorando la organización y rendimiento. Facilitan la reutilización de datos entre escenas y simplifican las modificaciones

## ENEMIGOS

Esto facilita la reutilización de datos entre diferentes enemigos y simplifica el ajuste de parámetros sin necesidad de modificar múltiples instancias. Además, mejora la organización del proyecto al separar datos de la lógica del juego.



## LOOT

permite definir de manera centralizada el rango de cantidades mínimas y máximas, así como las probabilidades de aparición. Esto facilita el ajuste y el balance de las mecánicas de loot sin alterar el código.

# HERENCIAS Y OBJECT POOLS

- Las object pools son beneficiosas porque mejoran el rendimiento al reducir la sobrecarga de la creación y destrucción de objetos en tiempo de ejecución.

- Facilitan la gestión de recursos en juegos y aplicaciones, permitiendo un manejo más eficiente de objetos que se utilizan con frecuencia, como proyectiles o enemigos, lo que resulta en una experiencia más fluida para el usuario.

```
Script de Unity | 6 referencias
public class GeneralPool : MonoBehaviour
{
    [Header("Pool Data")]
    [Space]
    [Range(1, 50)]
    [Tooltip("Número total de instancias del objeto en el pool.")]
    public int poolSize = 20;
    [Tooltip("Tipos de objetos que se instanciarán en el pool.")]
    [SerializeField] protected GameObject[] types;

    [HideInInspector]
    public GameObject[] typesInstances; // Instancias de los objetos en el pool.

    Mensaje de Unity | 3 referencias
    protected virtual void Start()
    {
        typesInstances = new GameObject[poolSize];

        for (int i = 0; i < poolSize; i++)
        {
            // Instancia un objeto aleatorio del array de tipos y lo coloca en la posición (0, 0).
            typesInstances[i] = Instantiate(types[Random.Range(0, types.Length)], new Vector2(0, 0f), Quaternion.identity);
            // Ajusta el indice del transform del objeto para mantener el orden en la jerarquía.
            typesInstances[i].transform.SetSiblingIndex(gameObject.transform.GetSiblingIndex() + 1);
        }
    }
}
```

# HERENCIAS Y OBJECT POOLS

- Utilizar herencias en programación es útil porque permite crear una jerarquía de clases que comparten características y comportamientos comunes, lo que promueve la reutilización de código.

- Al extender clases base, se pueden definir nuevas funcionalidades en las subclases sin duplicar código, lo que facilita la mantenibilidad y la expansión del sistema.

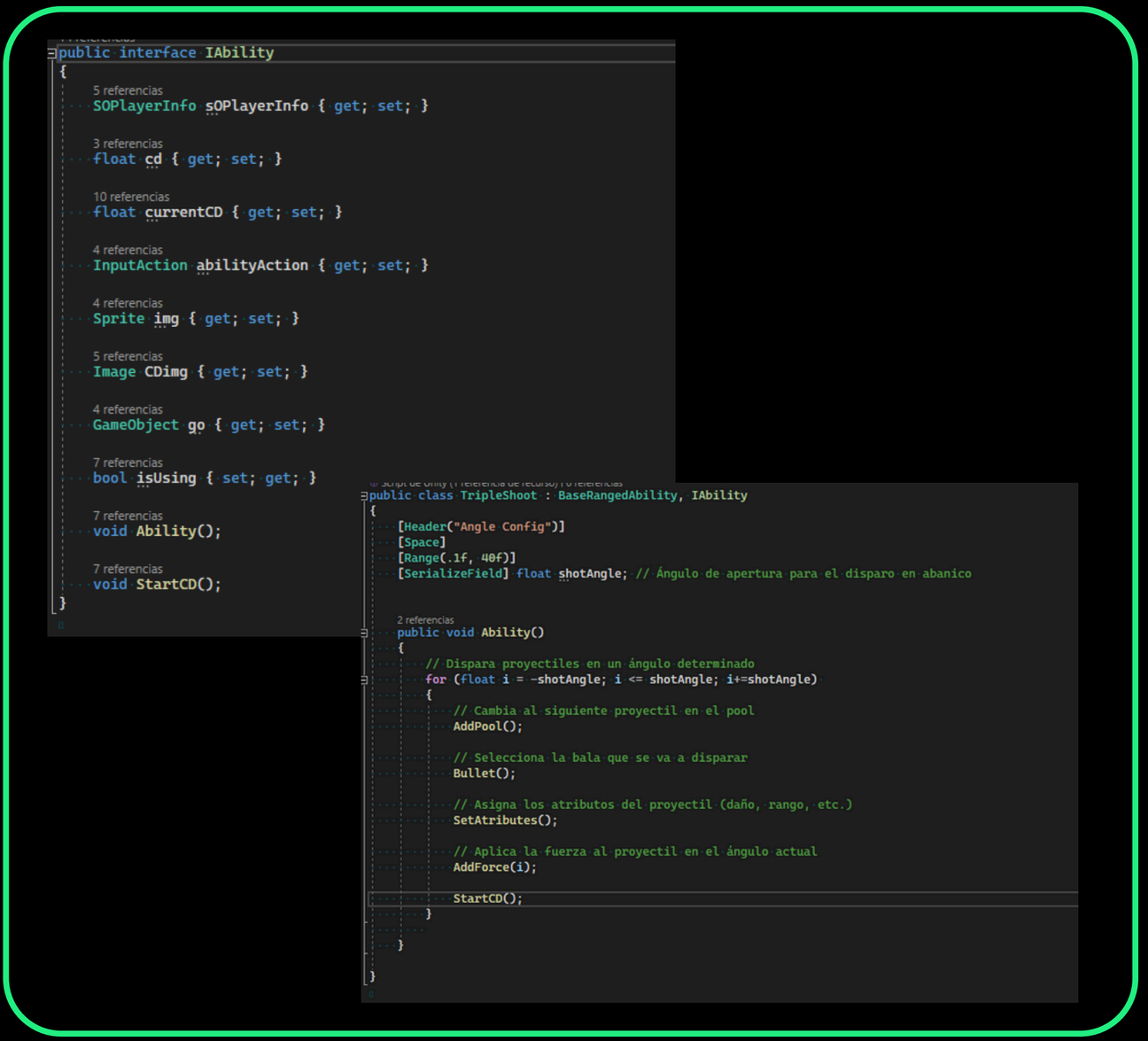
```
public class ExplosionParticlePool : GeneralPool
{
    public static ExplosionParticlePool instance;

    int explosionNumber = -1; // Índice para rastrear el siguiente objeto de partículas de explosión a usar

    Mensaje de Unity | 0 referencias
    private void Awake()
    {
        if (instance == null)
        {
            instance = this;
        }
        else
        {
            Destroy(gameObject);
        }
    }


    1 referencia
    public void SpawnExplosion(GameObject go, float _dmg, LayerMask hitMask)
    {
        explosionNumber++;

        // Si el índice excede el tamaño del pool, reinícialo a 0 para reutilizar objetos
        if (explosionNumber > poolSize - 1)
        {
            explosionNumber = 0;
        }

        // Obtiene el siguiente objeto de partículas de explosión del pool
        GameObject explosion = typesInstances[explosionNumber];

        // Establece la posición del objeto de partículas de explosión en la posición del objeto afectado
        explosion.transform.position = go.transform.position;

        // Reproduce el sistema de partículas
        explosion.GetComponent<ParticleSystem>().Play();

        // Configura el componente ExplosionTrigger del objeto de partículas de explosión
        ExplosionTrigger explosionTrigger = explosion.GetComponent<ExplosionTrigger>();
        explosionTrigger.hitLayer = hitMask; // Establece la máscara de capa para los objetos que serán afectados
        explosionTrigger.dmg = _dmg; // Establece el daño de la explosión
        explosionTrigger.enabled = true; // Habilita el componente para que pueda detectar colisiones

        // Habilita el componente Collider2D del objeto de partículas de explosión
        explosion.GetComponent<CircleCollider2D>().enabled = true;

    }
}
```

# INTERFACES Y EVENTOS

- Usar interfaces en programación es beneficioso porque promueve la separación de preocupaciones y el desacoplamiento entre componentes.

- Al definir contratos que las clases deben implementar, se facilita la reutilización y la extensión del código sin modificar clases existentes.

*(las parte de herencia para las interfaces se explica mejor en el video)*



```
public interface IAbility
{
    5 referencias
    SOPlayerInfo sOPlayerInfo { get; set; }

    3 referencias
    float cd { get; set; }

    10 referencias
    float currentCD { get; set; }

    4 referencias
    InputAction abilityAction { get; set; }

    4 referencias
    Sprite img { get; set; }

    5 referencias
    Image CDimg { get; set; }

    4 referencias
    GameObject go { get; set; }

    7 referencias
    bool isUsing { set; get; }

    7 referencias
    void Ability();

    7 referencias
    void StartCD();
}
```

```
public class TripleShoot : BaseRangedAbility, IAbility
{
    [Header("Angle Config")]
    [Space]
    [Range(.1f, 40f)]
    [SerializeField] float shotAngle; // Ángulo de apertura para el disparo en abanico


    2 referencias
    public void Ability()
    {
        // Dispara proyectiles en un ángulo determinado
        for (float i = -shotAngle; i <= shotAngle; i+=shotAngle)
        {
            // Cambia al siguiente proyectil en el pool
            AddPool();

            // Selecciona la bala que se va a disparar
            Bullet();

            // Asigna los atributos del proyectil (daño, rango, etc.)
            SetAtributes();

            // Aplica la fuerza al proyectil en el ángulo actual
            AddForce(i);

            StartCD();
        }

    }
}
```

# INTERFACES Y EVENTOS

- El uso de eventos en programación es valioso porque permite una comunicación flexible entre componentes sin crear dependencias directas.

- Esto fomenta el desacoplamiento, ya que un objeto puede notificar a otros sobre cambios o acciones sin saber quién los está escuchando.

```
// Eventos que se disparan en momentos clave: recibir daño, muerte, regeneración de salud y subir de nivel
public static event Action<GameObject> EventTriggerHitPlayer, EventTriggerDeathPlayer, EventTriggerHealthRegen;
public static event Action EventTriggerLevelUp;
```

```
public class HitStop : MonoBehaviour
{
    [Range(0.01f, 1f)]
    [SerializeField] float duration = 0.1f; // Duración del efecto de pausa en segundos

    bool waiting; // Indica si la pausa está en curso

    Mensaje de Unity | 0 referencias
    private void OnEnable()
    {
        PlayerStats.EventTriggerHitPlayer += StopTime;
    }

    Mensaje de Unity | 0 referencias
    private void OnDisable()
    {
        PlayerStats.EventTriggerHitPlayer -= StopTime;
    }


    2 referencias
    void StopTime(GameObject go)
    {
        // Inicia la corutina si no se está esperando actualmente
        if (!waiting)
        {
            StartCoroutine(HitStopCoroutine());
        }
    }

    1 referencia
    private IEnumerator HitStopCoroutine()
    {
        Time.timeScale = 0f;

        waiting = true;

        // Espera durante la duración especificada en tiempo real (ignora el `Time.timeScale`)
        yield return new WaitForSecondsRealtime(duration);

        Time.timeScale = 1f;

        waiting = false;
    }
}
```

# ¡MUCHAS GRACIAS!

DUNGEON SURVIVOR