# Práctica sobre generación aleatoria de enemigos mediante listas

## Introducción

Esta práctica tiene como objetivo la investigación de dos métodos distintos de generación aleatoria de enemigos. Para ello, utilizaremos las recién aprendidas listas para seleccionar posibilidades de entre un conjunto (por ejemplo, spawnear un enemigo aleatorio de entre 10 posibilidades). También necesitaremos recurrir a un generador en área horizontal-vertical, como hicimos en el proyecto del Monkey Coins.

Los objetivos de esta práctica son 4:

- Utilizar lo aprendido en los proyectos del FlappyBird y MonkeyCoins de forma autónoma para asentar conocimientos.
- Comprender y practicar la utilidad de la aleatoriedad en videojuegos.
- Comprender y practicar la listas en Unity.
- Comprender el concepto de "Generación procedural".

A continuación se explican ambos métodos, así como las instrucciones para implementarlos en Unity. Las explicaciones prácticas están basadas en el proyecto "PaacjFarm", el de los pollos que chillan. Podéis descargar un paquete básico del proyecto desde el aula virtual.

## Método 1: spawn aleatorio de entre enemigos prediseñados.

### ¿En qué consiste este método?

Como su nombre indica, este método consiste en crear numerosos prefabs de posibles enemigos. Entonces, en lugar de asignar al script Generador un solo prefab, se le asignan los prefabs en una lista, de forma que el Generador spawnee uno u otro prefab aleatoriamente.

El ejemplo más parecido de este tipo de generación aleatoria lo encontramos en las rutas de Pokémon, ya que cada ruta tiene predefinidos una serie de pokémons que pueden aparecer en ella.

## ¿Cómo aplicarlo en Unity?

Para ello, las tareas pertinentes serían:

- Hacer un script Generador que reciba como variable pública una lista de prefabs.
- Crear en el proyecto varios prefabs de enemigos bien diferenciados entre ellos (cámbiales el sprite y el

audio que emiten).
- En lugar de asignarle al Generador un único prefab, asignarle los prefabs creados.

## Método 2: spawn de enemigos generados proceduralmente.

### Pero primero: ¿Qué es la generación procedural?

Entendemos por generación procedural aquella donde los contenidos (en este caso los enemigos) no están diseñados de antemano, ni vienen almacenados como archivos dentro del propio juego, sino que **se crean de manera aleatoria** en base al código programado por los desarrolladores. De esta forma, los elementos generados proceduralmente serán únicos y prácticamente ilimitados, abriendo así el abanico de posibilidades hasta niveles impensables para juegos donde todo venga predefinido, aunque sacrificando para ello la precisión en el diseño y el gusto por el máximo detalle.

En la mayor parte de los casos, el principal elemento procedural es el mapa del juego en sí, que se genera aleatoriamente en cada partida. Ejemplos de juegos con generación procedural son:

- **Minecraft**: el mapa es procedural.
- **No Man's Sky**: el mapa, las naves, y las criaturas del juego son procedurales.
- **Borderlands**: las armas que looteas en baúles son procedurales.

Pero la generación procedural no es una tecnología reciente. El primer juego con generación procedural, en este caso de mazmorras, fue **Rogue** (1980), padre del género *Roguelike*.

La mayoría de juegos con factores aleatorios combinan de algún modo generación procedural y aleatoriedad prediseñada, ya que si no el diseño del juego sería incontrolable. Por ejemplo, en el caso de Pokémon, si bien los Pokémons que spawnean en cada ruta están predefinidos, las características de cada pokémon son procedurales, generadas aleatoriamente al spawnear el Pokémon (su sexo, su naturaleza son aleatorias, e incluso sus estadísticas pueden variar arriba o abajo levemente, permitiendo así que todos los Pikachus no sean idénticos).

Por tanto, como ves, la generación procedural no es exclusiva de los mapas o enemigos, se puede aplicar a cualquier cosa; y, además, puede aparecer combinada con aleatoriedad prediseñada para lograr así mayor control en el diseño.

> ¡DISCLAIMER! En realidad, esto último no es del todo cierto, ya que, estrictamente hablando, la generación procedural **siempre** aparece combinada con, al menos, un mínimo de aleatoriedad prediseñada, ya que si no el juego sería un completo caos.
>
> Para entenderlo, prueba a preguntarte lo siguiente: Si los árboles del mundo de Minecraft son procedurales, ¿por qué sus texturas son siempre las mismas y varían solamente entre 3 o 4 posibilidades? En realidad, la *forma* de los árboles es lo único procedural en ellos (así como su posición en el mundo).
>
> Igual ocurre con los Pokémons. Su porcentaje de aparición en las rutas es aleatoriedad prediseñada,

> pero sus características son levemente procedurales. Y digo *levemente* porque varían aleatoriamente un poco, pero no demasiado, para así equilibrar el juego.

**EN RESUMEN:**
Una buena forma de identificar cuando un elemento procedural es preguntarse: ¿este elemento está aquí en todas y cada una de las partidas porque *así lo decidió un diseñador*? O, en su defecto, ¿este elemento está aquí casualmente porque *así lo codificó un programador*?

### ¿Cómo aplicar la generación procedural en Unity?

En este caso, ya no necesitamos un montón de prefabs: nos basta con uno que se autogenerará a sí mismo al inicio aleatoriamente. Por tanto, **el Generador ya no necesita una lista de prefabs que spawnear**, ahora el que necesitará las listas será el propio prefab del pollo.

El prefab del pollo estará compuesto por los siguientes componentes:

- Transform.
- Sprite Renderer (para mostrar la apariencia del pollo).
- Collider (para detectar que ha sido clicado).
- AudioSource (para emitir el sonido del gruñido).
- Script Pollo.cs (puedes llamarlo como quieras). Encargado de matar al pollo cuando clickes sobre él.

El prefab del pollo tendrá uno (o varios, a elección del programador) scripts que recibirán como variable pública en el inspector una lista de audios y otra de sprites. Y, al inicio (en el método `Start()`), accederán a los componentes SpriteRenderer y AudioSource con `GetComponent<>()` y les asignarán un sprite y un audio aleatorios de los que hayan recibido en la lista.

De esta forma, nada más spawnear el pollo y antes de que pueda llegar siquiera a verlo el jugador, éste tomaría una nueva apariencia y un nuevo sonido aleatorios, convirtiéndose así en un pollo único, y procedural.

## Práctica

Por parejas, divide la tarea con tu compañero: uno implementará la generación de los pollos mediante aleatoriedad prediseñada, y otro implementará la generación de los pollos proceduralmente.

Descargad ambos el paquete “Ejercicio Arrays” del aula virtual, e importadlo, cada uno, en su propio proyecto de Unity.
Nombrad al proyecto como “Nombre1_Procedural” o “Nombre2_Prediseñado”.

Antes de nada, **comenzad la sesión diseñando juntos unos cuantos sprites de pollos alternativos** (al menos 5). Deben ser imágenes en formato .png, con el fondo transparente, y deben ser del mismo tamaño que las del pollo original (28x40) para que no haya diferencias de tamaño entre unos y otros. Toma como base el sprite del pollo original del paquete de Unity descargado del aula virtual. Cada tipo de pollo reproducirá un gruñido diferente para mayor diferenciación.

Una vez tengáis ambos proyectos funcionando y generando pollos aleatorios, explicaos el uno al otro cómo funciona vuestro método. Por ejemplo, cómo tendríais que hacer para introducir un nuevo tipo de pollo con un gruñido diferente en el juego. Entonces, una vez comprendáis ambos métodos, redactad un escrito (en PDF) en qué saquéis conclusiones de cada uno de los métodos sobre cuál es mejor en unos casos y en otros. Como guía, podéis pensar en las siguientes preguntas:

- ¿Qué ventajas e inconvenientes aprecias en cada uno de los métodos?
- ¿Cuál de los dos métodos facilita más el crear enemigos en cantidad?
- ¿Cuál de los dos métodos permite mayor control sobre los enemigos que aparecerán en el juego?

## Entrega

Fecha de entrega: **Domingo día 10**.

Forma de entrega: carpeta comprimida en .zip con los dos proyectos de Unity y el texto de conclusiones, que deberá subirla solamente 1 de los integrantes del equipo. El .zip debe llamarse, al igual que la carpeta raíz, Nombre1_Nombre2.zip. Abriré la tarea en el aula virtual.

Más explícitamente:

- Nombre1_Nombre2.zip
    - Nombre1_Prediseñado (proyecto de Unity con el método 1 implementado).
    - Nombre2_Procedural (proyecto de Unity con el método 2 implementado).
    - Conclusiones.pdf.

# Apuntes

La mayoría de cosas necesarias para este proyecto las hemos utilizado anteriormente en Flappy Bird o Monkey Coins, por lo que recurre a esos proyectos para fijarte en cómo programar un Generador o como crear un Prefab.

La mayor novedad es la utilización de las listas junto con números aleatorios. Para ello, fíjate en los apuntes sobre listas del aula virtual,

## Detectar click un game object

Para detectar si se ha hecho click en un Game Object, primero necesitamos que éste tenga un Collider. Si lo tiene, nos basta con programar lo que queremos que suceda en caso de haber hecho click dentro del siguiente método:

```
void OnMouseDown()
{
 // Programa aquí lo que quieres que suceda cuando se haga click sobre el objeto.
}
```

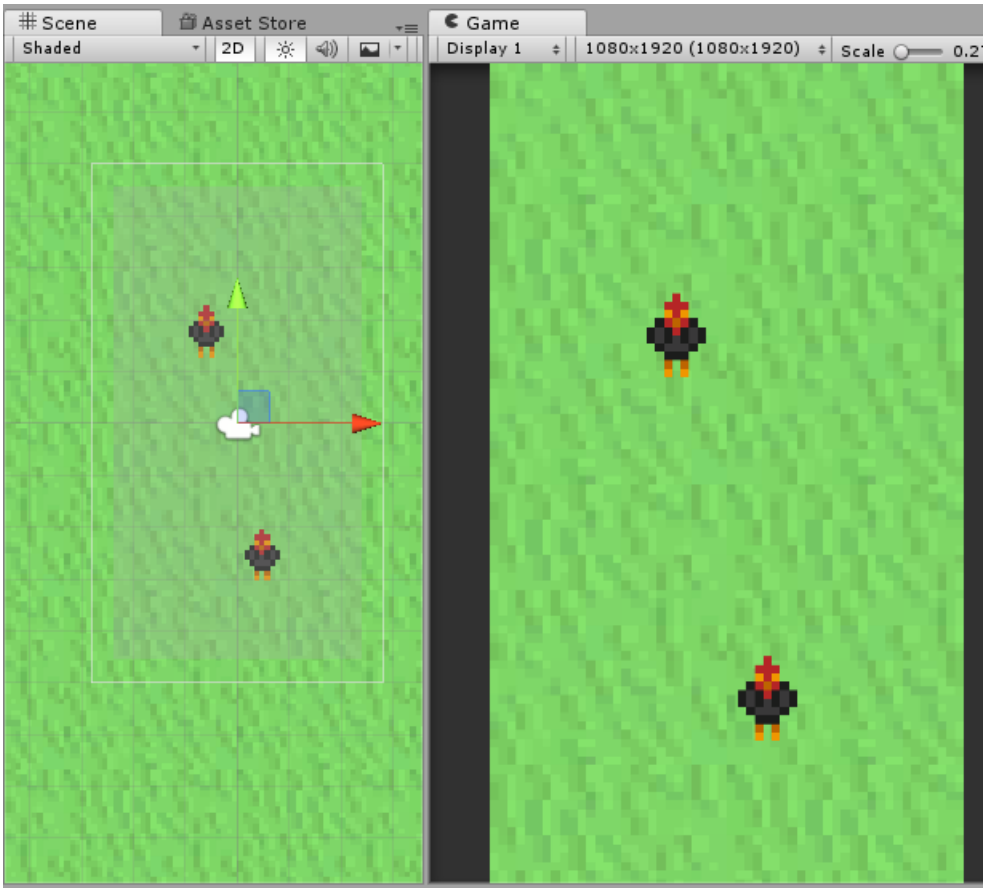
## Visualizar área de spawn en el editor

Para configurar un área donde spawnean enemigos, lo ideal es permitir modificar el rango aleatorio desde el inspector, al igual que hicimos en el proyecto de Flappy Bird. Pero, ¿cómo sabemos en el editor si los valores que estamos introduciendo son los correctos y los enemigos no se saldrán de la pantalla? Hasta ahora, teníamos que hacerlo a ojo, ir probando y moviendo objetos por la pantalla a ver con qué valores se quedaban en el borde y con cuáles centrados.

Con el siguiente método, podemos dibujar en la vista de la escena un área translúcida que nos indique dicho área, lo cual será especialmente útil para que el diseñador del nivel pueda configurar fácil y visualmente dónde aparecerán los enemigos:

```
public class Generador : MonoBehaviour
{
    public float horizontalArea, verticalArea;

 // [ ... ]

    void OnDrawGizmosSelected()
    {
        Color areaColor = Color.white;
        areaColor.a = 0.25f;

        Vector3 areaSize;
        areaSize.x = horizontalArea * 2;
        areaSize.y = verticalArea * 2;
        areaSize.z = 0f;

        Gizmos.color = areaColor;
        Gizmos.DrawCube(transform.position, areaSize);
    }
}
```

Eso se verá de la siguiente manera:




Como puedes observar no tienes por qué preocuparte por su apariencia, ya que solo se ve en el Editor y no dentro del juego.