

Juego Roguelike 2D con Unity

-

Proof of Concept (POC - Prueba de concepto)

ÍNDICE

1 - Introducción.....	2
1.1 - Pruebas de concepto.....	2
2 - Base de datos.....	3
2.1 - Plugin para Unity.....	3
2.2 - Script de gestión.....	5
2.3 - Convertir a objetos de C#.....	6
2.4 - Resultado final.....	7
3 - Mapa de juego.....	8
3.1 - Estructurar el mapa.....	8
3.1.1 - Concepto del mapa.....	8
3.1.2 - Tiles.....	9
3.1.3 - Clase Map en C#.....	10
3.2 - Dibujar el mapa.....	13
3.2.1 - El problema de las coordenadas.....	13
3.2.2 - Solución.....	16
3.3 - Movimiento.....	18
4 - Interactuar con el entorno.....	20
4.1 - Leer contenido adyacente.....	20
4.1.1 - Código.....	20
4.1.2 - Unity.....	21
4.2 - Interfaz gráfica.....	22
5 - Panel de estados.....	23
5.1 - Definición.....	23
5.2 - Adaptabilidad de la interfaz.....	24
5.3 - Modificar los contadores.....	25
6 - Turnos y eventos.....	26
6.1 - Mecánica de turnos.....	26
6.2 - Tabla eventos.....	27
6.3 - Finalizar turno.....	28
7 - Inventario.....	29
7.1 - Iconos.....	29
7.2 - Prefab de ITEM.....	30
7.3 - Singleton.....	30
7.4 - Sección de inventario.....	31
8 - Idiomas.....	33
8.1 - Tablas de localización.....	33
8.2 - Textos en la base de datos.....	34
9 - Referencias y fuentes.....	35

1 - Introducción

1.1 - Pruebas de concepto

Antes de empezar con nuestra prueba de concepto (de ahora en adelante POC) vamos a definir qué es exactamente una POC y en qué márgenes se moverá en nuestro proyecto.

Definición de **Wikipedia**:

Una prueba de concepto o PoC (del inglés proof of concept) es una implementación, a menudo resumida o incompleta, de un método o de una idea, realizada con el propósito de verificar que el concepto o teoría en cuestión es susceptible de ser explotada de una manera útil.

La PoC se considera habitualmente un paso importante en el proceso de crear un prototipo realmente operativo. En seguridad informática, se usan pruebas de concepto para explicar cómo se pueden explotar vulnerabilidades de día cero. Se trata de vulnerabilidades de las cuales se desconoce su funcionamiento exacto, y por tanto se utilizan PoC para intentar entender cómo pueden explotarse en un sistema o equipo.

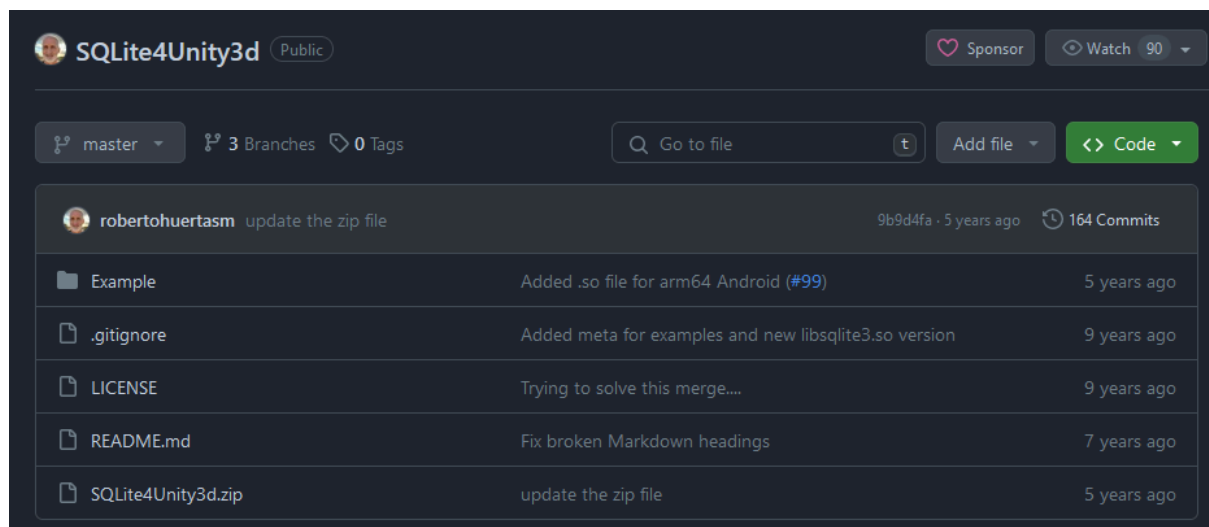
Con eso en mente definimos los límites impuestos a nuestra POC. El objetivo de esta etapa del proyecto es definir qué se puede hacer y qué no dentro de la idea inicial así como enfrentar problemas derivados de la falta de experiencia del alumno o cambios constantes en los conceptos de los videojuegos y tecnologías asociadas.

Un gran ejemplo de ello es el cambio de sistema gestor de base de datos mencionado en el documento **Elección de la base de datos** anterior. Gracias a la prueba de concepto nos hemos topado con este problema al inicio y sin estar directamente en el proyecto real lo cual nos ha permitido solventar esa etapa de decisión inicial de forma correcta. La alternativa, no haber realizado una POC y directamente trabajar sobre el proyecto final, habría terminado topando con problemas como el de la base de datos y otros que se mencionan en este documento y se tendrían que solventar de forma rápida y poco eficiente. Gracias a la POC realizamos una especie de pre-proyecto donde tendremos una carcasa funcional con todas las mecánicas y flujo de trabajo que debe tener nuestro proyecto pero de forma resumida y no necesariamente estética. Es decir, queremos ver **que todo funciona o podrá funcionar** de forma real no solo basarnos en que el trabajo se hará más adelante. Una vez todo funciona se arrancará el desarrollo del proyecto definitivo y real sobre unas bases sólidas y estructuradas.

2 - Base de datos

2.1 - Plugin para Unity

Gracias a la POC pudimos detectar un problema importante en la base de datos original que se había elegido para el proyecto (Firebase) y se decidió a cambiar a **SQLite**.



Con ello en mente y después de diferentes pruebas con servidores en la nube, bases de datos distribuidas, bases de datos en local en otra máquina y un largo sinfín de prueba realizadas ya detalladas en anteriores documentos lo siguiente que toca hacer es ver como conectar correctamente la base de datos SQLite a Unity. Para ello haremos uso del plugin **SQLite4Unity3d** de **Roberto Huertas**.

En principio existe mucha documentación y vídeos que respaldan su uso pero tiene un gran inconveniente. Lleva 5 años sin actualizarse. Aquí vemos una de las utilidades de elaborar una POC antes de lanzarnos al proyecto final. Nuestro primer paso será probar si la base de datos se crea correctamente, se conecta y se gestiona bien usando dicho plugin.

Crearemos un proyecto nuevo en Unity eligiendo el entorno 2D y lo llamaremos **SQLiteTest**. Este será el nombre general de la POC sobre la que desarrollaremos todo.

Vamos a usar la base de datos simple que ya mostramos en el documento **Elección de la base de datos** y crearemos un script .sql para crear la base de datos, varios script en C# para que Unity se conecte y gestione la base de datos, diferentes archivos "insert.txt" donde tendremos la información relevante de nuestras tablas para rellenar. Estos archivos serán una especie de CSV que cargaremos en la POC. Y un script en C# que lance los anteriores scripts, rellene la base de datos y recupere la información a objetos.

Si todo se desempeña correctamente debemos tener una base de datos con varias tablas, siendo la tabla más completa y que más nos interesa la de “places” que vemos a continuación:

DB Browser for SQLite - C:\Users\Fernando Esra\Desktop\OneLastTime\UnityBDTest\SQLiteTest\Assets\StreamingAssets\oltbedtw.db										
Archivo Editar Ver Herramientas Ayuda										
Nueva base de datos Abrir base de datos Guardar cambios Deshacer cambios Abrir proyecto Guardar proyecto										
Estructura Hoja de datos Editar pragmas Ejecutar SQL										
Tabla: places Filtrar en cualquier columna										
	idPlace	Name	Description	PlaceType	Option1	Item1	Option2	Item2	Option3	Item3
	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	1	Place1	Description1	empty	Place1_option1	1	Place1_option2	2	NULL	NULL
2	2	Place2	Description2	empty	Place2_option1	NULL	NULL	NULL	NULL	NULL
3	3	Place3	Description3	forest	Place3_option1	4	Place3_option2	5	Place3_option3	NULL
4	4	Place4	Description4	empty	Place4_option1	NULL	Place4_option2	NULL	Place4_option3	NULL
5	5	Place5	Description5	forest	Place5_option1	NULL	NULL	NULL	NULL	NULL
6	6	Place6	Description6	forest	Place6_option1	7	Place6_option2	8	Place6_option3	9
7	7	Place7	Description7	empty	Place7_option1	10	Place7_option2	NULL	NULL	NULL
8	8	Place8	Description8	empty	Place8_option1	11	Place8_option2	12	Place8_option3	3
9	9	Place9	Description9	forest	Place9_option1	NULL	NULL	NULL	NULL	NULL
10	10	Place10	Description10	forest	Place10_option1	7	Place10_option2	4	NULL	NULL
11	11	Place11	Description11	forest	Place11_option1	3	NULL	NULL	NULL	NULL
12	12	Place12	Description12	forest	Place12_option1	1	Place12_option2	9	NULL	NULL
13	13	Place13	Description13	forest	Place13_option1	17	Place13_option2	NULL	NULL	NULL
14	14	Place14	Description14	forest	Place14_option1	4	NULL	NULL	NULL	NULL
15	15	Place15	Description15	forest	Place15_option1	12	Place15_option2	14	NULL	NULL
16	16	Place16	Description16	forest	NULL	NULL	NULL	NULL	NULL	NULL
17	17	Place17	Description17	forest	Place17_option1	16	NULL	NULL	NULL	NULL
18	18	Place18	Description18	forest	Place18_option1	6	Place18_option2	7	NULL	NULL
19	19	Place19	Description19	forest	Place19_option1	15	Place19_option2	14	NULL	NULL
20	20	Place20	Description20	forest	Place20_option1	3	NULL	NULL	NULL	NULL
21	21	Place21	Description21	forest	Place21_option1	1	NULL	NULL	NULL	NULL
22	22	Place22	Description22	forest	Place22_option1	7	NULL	NULL	NULL	NULL
23	23	Place23	Description23	forest	Place23_option1	8	NULL	NULL	NULL	NULL
24	24	Place24	Description24	forest	Place24_option1	9	NULL	NULL	NULL	NULL
25	25	Place25	Description25	forest	Place25_option1	12	NULL	NULL	NULL	NULL
26	26	Place26	Description26	forest	Place26_option1	13	NULL	NULL	NULL	NULL
27	27	Place27	Description27	forest	Place27_option1	14	Place27_option2	NULL	NULL	NULL
28	28	Place28	Description28	forest	Place28_option1	16	Place28_option2	NULL	NULL	NULL
29	29	Place29	Description29	forest	Place29_option1	10	NULL	NULL	NULL	NULL

2.2 - Script de gestión

Siguiendo la guía que tenemos en GitHub, la documentación y diferentes tutoriales de youtube citados en fuentes (NEERACADEMY) se logró, después de casi una semana de trabajo continuo, localizar algunos errores de desfase entre el plugin y Unity para así poder subsanarlos correctamente y que la base de datos funcionase tanto en Android como en Windows.

```
#if UNITY_EDITOR
    // var dbPath = Path.Combine(Application.streamingAssetsPath, DatabaseName);
    var dbPath = string.Format(@"Assets/StreamingAssets/{0}", DatabaseName);
    // DeleteIfExists(dbPath);
```

El problema general se ubica en la forma en la que el script estaba accediendo a la base de datos y almacenando la ruta donde se encontraba. Dicha gestión estaba algo anticuada en lo referente a las versiones más nuevas y daba error fuera del editor. Es decir, al lanzar en juego en modo prueba desde Unity todo iba bien pero al lanzar el juego compilado tanto desde Windows como desde Android el programa no era capaz de ubicar correctamente la base de datos.

Ha sido una labor ardua de depuración de código en parte por desconocimiento de cómo se realiza esta gestión en Android en parte por tener que trabajar con un código tan complejo como es el gestionado por el plugin además de comprender, en el camino, intrincados aspectos de funcionamiento tanto de Unity como de SQLite. Pero después de mucha investigación se logró dar con la solución.

```
Debug.Log("Insert Items");
string filePath = Path.Combine(Application.streamingAssetsPath, "insertItems.txt");
Debug.Log("Existe \"insertItems.txt\" en " + filePath + "\n\n");
WWW www = new WWW(filePath);
```

Para ello se usa el objeto **WWW** el cual nos permite gestionar diferentes tipos de direcciones tanto https como locales. Con ello logramos acceder mucho más eficientemente a las carpetas persistentes del sistema (principalmente en Android) y además tenemos una capa de seguridad extra. Si bien mientras se estaba desarrollando la POC y se realizaba el proceso de documentación y formación para el correcto uso de WWW se ha visto que se podría usar, de forma más eficiente, el módulo **UnityWebRequest**. Después de varias pruebas con UnityWebRequest se deja ese cambio para futuras iteraciones del proyecto dado que actualmente no se dispone de mucho tiempo.

Una vez solventado este problema podemos empezar a trabajar con una base de datos SQLite en Unity.

2.3 - Convertir a objetos de C#

El objetivo final de nuestra base de datos es tener objetos en C# “descargados” de la base de datos para trabajar con ellos en Unity.

```
2 references
public List<Item> ReadItems(List<Condition> conditionsList)
{
    Debug.Log("ReadItemsStart");
    List<Item> itemList = new List<Item>();

    // using (var connection = new SQLiteConnection(ShowPath(), SQLiteOpenFlags.ReadWrite | SQLiteOpenFlags.Create))
    using (var connection = ods.GetConnection())
    {
        var items = connection.Query<TempItem>("SELECT items.Name AS ItemName, items.Description AS ItemDescription, conditions.Name AS ConditionName, " +
        "items.Icon AS ItemIcon FROM items LEFT JOIN conditions ON items.Effect = conditions.idCondition;");
        foreach (var item in items)
        {
            // Debug.Log($"Name: {item.ItemName}, Description: {item.ItemDescription}, Effect: {item.ConditionName}");
            Condition foundCondition = conditionsList.FirstOrDefault(condition => condition.Name == item.ConditionName);
            Item actual = new Item(item.ItemName, item.ItemDescription, foundCondition, item.ItemIcon);
            itemList.Add(actual);
        }
    }
    Debug.Log("ReadItemsEnd");
    return itemList;
}

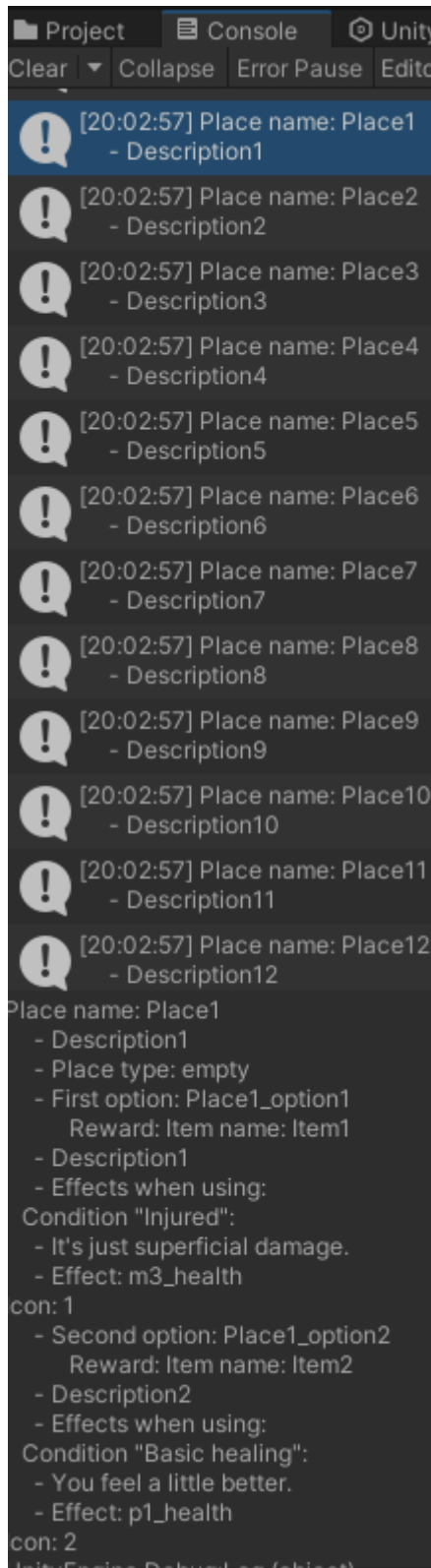
1 reference
public class TempItem
{
    1 reference
    public string ItemName { get; set; }
    1 reference
    public string ItemDescription { get; set; }
    1 reference
    public string ConditionName { get; set; }
    1 reference
    public int ItemIcon { get; set; }
}
```

Mediante conexiones gestionadas por las clases gestoras que hemos definido y arreglado para SQLite4Unity3d nos es posible realizar sentencias SELECT y traer los datos a código. Dichos datos los vamos a ir almacenando en clases temporales cuyos datos luego usaremos para almacenar en nuestros datos finales.

Este paso intermedio se hace, por ejemplo, para buscar relaciones. Vemos en el código superior que rescatamos la tabla de objetos (items) pero los objetos tienen una relación, una asociación con la tabla condiciones que ya fue recuperada anteriormente y almacenada en un ArrayList de condiciones llamado conditionsList. Mediante la sentencia select recuperamos un nombre de condición ya que en el SELECT tenemos un LEFT JOIN para disponer del nombre en lugar de un INT que es el valor que tiene la columna Effect dentro del objeto en la base de datos. Con este dato recuperado hacemos una búsqueda en el array conditionsList y podremos encontrar nuestro objeto de C# para vincular con el nuevo recuperado.

A medida que la base de datos crece en cantidad de tablas, referencias y complejidad de su estructura también crece la dificultad a la hora de recuperar los datos. Gracias a la POC podemos definir mejor la base de datos final que vamos a usar conociendo bien los entresijos del código que tendremos que afrontar una vez queremos traer a Unity esos objetos. Este paso ha sido de vital importancia para asegurar una estructura sólida en la base de datos real de nuestro proyecto y en caso de haberse realizado durante la realización del proyecto y no en la POC podría haber creado una deficiencia en el resultado final reseñable.

2.4 - Resultado final



Lo que queremos principalmente es comprobar que hemos logrado una conexión exitosa tanto en Windows como en Android entre nuestra base de datos SQLite y Unity, para ello nos bastará (por ahora) con imprimir por pantalla la información de todos los objetos. Si logramos hacer eso, la integridad y flujo de trabajo de la base de datos de la POC cumpliría su función.

En orden, lo que hemos hecho ha sido:

- Crear la base de datos usando un script (fichero) que interpreta el plugin
- Rellenar la tabla **conditions** usando un fichero .txt con toda la información dentro, mediante código.
- Rellenar la tabla **items** usando un fichero .txt con toda la información dentro, mediante código.
- Rellenar la tabla **places** usando un fichero .txt con toda la información dentro, mediante código.
- Rellenar la tabla **events** usando un fichero .txt con toda la información dentro, mediante código.
- Conectar a la tabla **conditions** mediante código, traer sus datos a un objeto de C# llamado Condiciones y almacenarlo en un ArrayList de Condiciones.
- Conectar a la tabla **items** y leer todos sus datos, la tabla items está relacionada con conditions para lo cual hacemos una consulta con JOIN, recuperamos el nombre de la condición asignada (si existe) y la buscamos en el ArrayList de Condiciones, de esta forma obtenemos un objeto llamado Items en C# que guardaremos en un ArrayList de Items.
- Similar a lo hecho con Items conectamos con **places** y recuperamos todos los lugares. Los lugares a su vez pueden tener dentro Items que recuperaremos gracias a otro JOIN en la sentencia SELECT.

Como resultado final realizamos una iteración con un bucle foreach sobre el ArrayList de Places. La imagen mostrada a la izquierda es el resultado. Podemos ver que todos los datos han sido rescatados con éxito y plasmado en objetos de C# que Unity interpreta y con los cuales será posible construir nuestro mundo, nuestras interacciones, mecánicas y, en general, nuestro juego.

3 - Mapa de juego

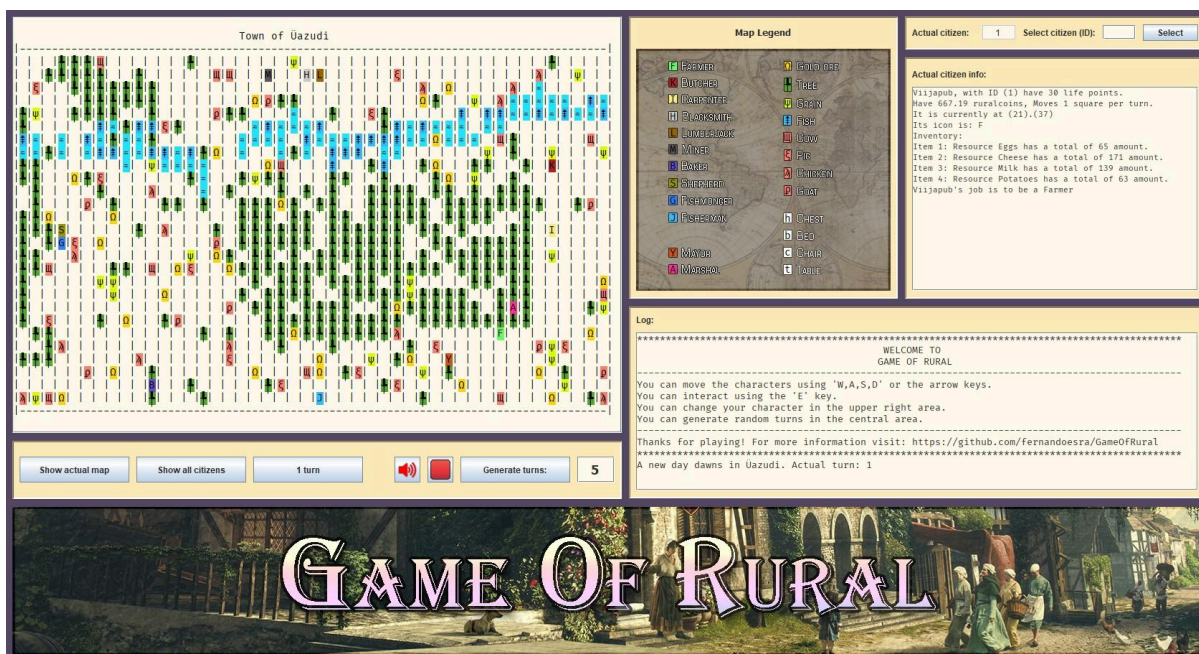
3.1 - Estructurar el mapa

En el documento de **Análisis Iniciales** se habló superficialmente de la idea del mapa. Tendremos un Array de objetos “lugar” de dos dimensiones (matriz) por el cual nuestro jugador (un puntero) se moverá. Dicho puntero debe poder interactuar con los lugares que tiene a su alrededor y mostrar por pantalla información.

El primer obstáculo al que se enfrenta la POC es llevar esto a la pantalla. Si bien la construcción del mapa (matriz) de Places no es algo complejo en exceso dado que todo se hará mediante código en C# su posterior dibujo es algo más elaborado.

3.1.1 - Concepto del mapa

Para tener una referencia de que se pretende hacer vamos a tomar de base un juego que realicé el año pasado al finalizar el curso. Game of Rural (enlace a GitHub en referencias).



Este juego tiene un Array de 2 dimensiones (matriz) con objetos de tipo Persona y cada persona puede ser de una profesión diferente. Lo importante que tenemos es que el método toString() de Persona nos devuelve su “icono” que no es más que un chara para definirlo y asociarlo a algo. Podemos ver así cómo se construye la ilusión jugable de que tenemos todo un escenario con diferentes personas. En el centro arriba podemos ver una leyenda

con diferentes asociaciones en cuanto a caracteres y colores. Con ello el jugador puede relacionar, por ejemplo, la letra F en verde al Farmer (granjero) y saber donde se encuentra en el mapa. Una vez más, este mapa no es más que una matriz de objetos en Java y somos nosotros, mediante código, los que decidimos su comportamiento y como se dibujara.

Es interesante el ejemplo por una razón, es un ejemplo sencillo. Con las “celdas” dibujadas y con un interior no muy largo (solo un carácter) no es difícil ver lo que estamos haciendo. Se asemeja mucho a imprimir una matriz de chars mediante un doble bucle for. Eso es lo que realiza el programa “de forma muy resumida” pero usando el toString() de la Persona que hay dentro.

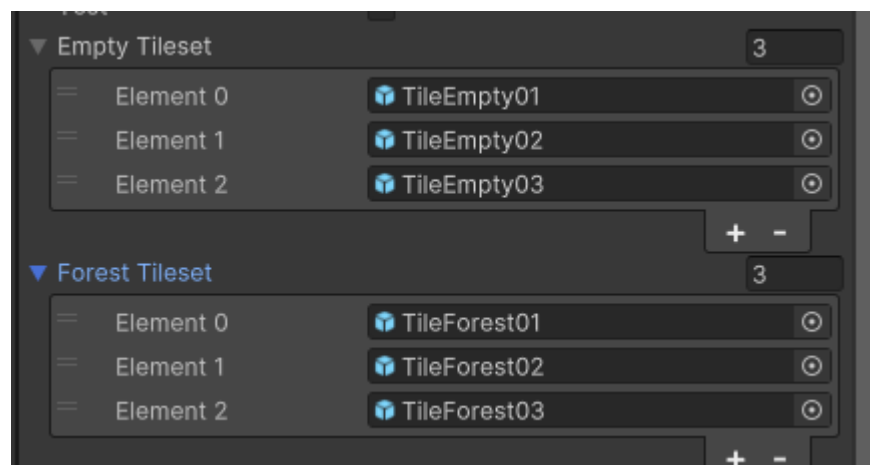
1	PlaceType	
	Filtro	Fi
1	empty	Pl
2	empty	Pl
3	forest	Pl
4	empty	Pl

Nuestro objetivo en Unity es similar. Tenemos una matriz de Places (lugares) y queremos dibujarlos dependiendo de, en el caso de la POC, su PlaceType (tipo de lugar) que puede ser empty (vacío) o forest (bosque, aún que se usa en la POC como “no vacío”).

Esto nos permite jugar un poco con cómo interpretamos los objetos al venir de la base de datos y nos permite controlar el tipo de lugar que será desde dicha base. El objetivo a lograr es que en la versión definitiva tengamos muchos tipos de lugares, por ejemplo cuevas, ríos, ciudades, ruinas, encuentros de algún tipo (peligrosos, misteriosos, milagrosos), los bosques ya mencionados, quizás encuentros con npcs (aún que no se plantean npcs debido a la trama, pero sería una posibilidad sencilla de añadir), animales salvajes y en general todo lo que queramos.

3.1.2 - Tiles

Vamos a crear dos arrays de GameObjects en Unity llamados Empty y Forest y dentro añadiremos diferentes tiles que, de momento, serán de un Asset pack de Kenney (ya mencionado en el primer documento de Análisis iniciales).



Nuestro código usará de forma aleatoria uno de los tiles que se encuentran en el array para, de paso, probar un poco con la generación aleatoria. Ya que vamos a tener, presumiblemente, muchos lugares de un mismo tipo (por ejemplo bosques) y al posicionarlos con los futuros algoritmos de generación de terreno no queremos que todos los tiles sean iguales. Por ejemplo dibujar 3 o 4 variaciones de cada tile para darle algo de vida al mundo.



Los objetos añadidos al Array serán **prefabs** de Unity, lo cual quiere decir que son objetos del editor que podemos instanciar. Cada objeto puede tener propiedades dentro, al igual que los objetos de las clases de los lenguajes de programación y después de crear nuevas instancias podemos crear referencias a esas propiedades y editarlas. Si bien esto es algo que actualmente no nos interesa en un primer vistazo dado que lo único que necesitamos es que nuestro prefab tenga un icono asignado. Es decir, cada prefab tendrá un **componente Sprite Renderer** con un Sprite asignado que corresponderá al tile que queramos. Como vemos en la imagen superior hemos elegido 3 tiles para los lugares vacíos y 3 tiles para los lugares de bosque.

3.1.3 - Clase Map en C#

Nuestro objeto “Mapa” no tiene excesivo misterio en su construcción si bien es importante definir su manejo. Dispondremos de una matriz de objetos Place que el constructor iniciará con unos datos dados (los cuales manejaremos desde nuestro código principal o desde Unity para debug) y también iniciará el **puntero del jugador usando un Vector2** a forma de representar las coordenadas en ejes X e Y en el mapa.

```
public class Map
{
    15 references
    public Place[,] map;
    10 references
    public int height { get; set; }
    10 references
    public int width { get; set; }
    8 references
    public Vector2 PlayerPointer = new Vector2(0, 0);
    3 references
    private int[,] Directions = new int[,] { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };

    2 references
    public Map(int height, int width)
    {
        this.height = height;
        this.width = width;
        map = new Place[height, width];
    }
}
```

Creando una instancia de Map con el constructor **map(20x20)**, por ejemplo, crearemos un mapa totalmente cuadrado de 200 Places que, importante, serán todos Null.

```

2 references
public void fillMap(List<Place> places)
{
    System.Random random = new System.Random();
    foreach (var item in places)
    {
        int x = random.Next(height);
        int y = random.Next(width);

        while (map[x, y] != null)
        {
            x = random.Next(height);
            y = random.Next(width);
        }

        map[x, y] = item;
    }
}

```

Una vez nuestra matriz está creada lo que vamos a hacer será rellenarla. Le pasaremos mediante nuestro código principal el ArrayList de lugares que nos descargamos de la base de datos e iremos creando de forma 100% aleatoria posiciones de la matriz. En caso de que la posición esté “vacía” (su contenido sea null) se pondrá dentro una referencia al objeto y se pasará al siguiente.

En un futuro existirán algoritmos de posicionamiento y generación semi-aleatorios para cada tipo de terreno. Por

ejemplo las montañas estarán separadas unas de otras por una cantidad definida de tiles, las ruinas igual, los bosques tendrán densidad, los ríos serán unas “líneas rectas” que cruzan el mapa etc. Esto se puede ver ligeramente en el ejemplo del Game of Rural en la [sección 3.1.1](#). Allí vemos que la representación visual del mapa tiene sentido, los bosques parecen bosques y los ríos parecen ríos.

Pero para nuestra POC lo único que queremos es posicionar los lugares de formas totalmente aleatorias, con eso bastará para mostrar que funciona. Con ello ya tenemos algo, hemos logrado traer desde nuestra base de datos todos los lugares correctamente vinculados con sus items, correctamente asociados a sus condiciones. Y una vez recuperados estamos rellenando una matriz con ellos para asemejar un mapa.

Los siguiente será definir condiciones para dar algo de sentido a las reglas de nuestro mapa. Los métodos **InBounds(x, y)** y **Empty(x, y)** comprueban si la coordenada buscada no está fuera del Array y que sea null, respectivamente. Esto nos da el método **Valid(x, y)** con el cual podemos comprobar si una coordenada específica es válida. Esto se usa, por ejemplo, si queremos mover al personaje. Con ello hemos creado un sistema rústico de colisiones mediante el cual nuestro

```

4 references
public bool Valid(int x, int y)
{
    if (InBounds(x, y) && Empty(x, y))
    {
        return true;
    }
    else
    {
        return false;
    }
}

```

puntero (un Vector2) podrá moverse por la matriz. Imaginemos por ejemplo que **nuestro puntero (de ahora en adelante personaje)** se encuentra en (0,0) y queremos movernos

Por último y de vital importancia para del debug, tenemos que sobrescribir el método ToString() para escribir en consola el contenido de la matriz. Usaremos el atributo Name de los Place y para el personaje la palabra clave PLAYER.

[illegible]

Se podría concluir con que una vez tenemos en código todo esto el principal propósito de la POC a nivel de código con la base de datos está cumplido. Se ha logrado conectar a la base de datos y transformar todo su contenido a un mapa jugable. Esto recuerda en gran parte a las antiguas aventuras de exploración (de las cuales se hablará en siguientes documentos) donde todo era texto plano y apenas teníamos dibujos más allá de algunas interpretaciones imaginativas en arte ASCII. Si ahora mismo quisiéramos, podríamos programar un juego completo de ese estilo, desprovisto de gráficos más elaborados.

Proyecto fin de grado. Fernando Tarriño del Pozo. DAM-2023-2024. **Prueba de concepto** - Pág 12

3.2 - Dibujar el mapa

Nuestra siguiente meta será lograr dibujar el mapa en pantalla. Todo lo que hemos visto hasta ahora son mensajes en la consola de debug que las clases de C# arrojan y pueden ser interpretadas por un programador en Unity, pero dichos mensajes no son vistos por un usuario final. Es decir, el método ToString() que tenemos de nuestro mapa es muy elaborado pero no muestra información relevante al jugador humano que controla al personaje.

Esa es la razón por la que estamos usando Unity. Este editor nos permite rapidez a la hora de mostrar gráficos en pantalla en comparación con, por ejemplo, librerías como libGDX de Java (que he usado en el pasado, se puede ver en mi GitHub) donde todo el código debe ser rehecho desde cero. No todos los proyectos tienen por qué reinventar la rueda. Ya nos enfrentamos a muchos desafíos a la hora de gestionar el proyecto y construir el juego es por ello que una decisión sensata es relegar algunos aspectos del código (gestión gráfica, renderizado, streams de música) a un editor 3D.

3.2.1 - El problema de las coordenadas

Pero eso no quiere decir que carezca de complejidad. Es de vital importancia entender lo que estamos haciendo muy bien tanto a nivel de código como a nivel de editor 3D. El primer problema que encontramos cuando queremos dibujar nuestro mapa tiene que ver con las coordenadas.

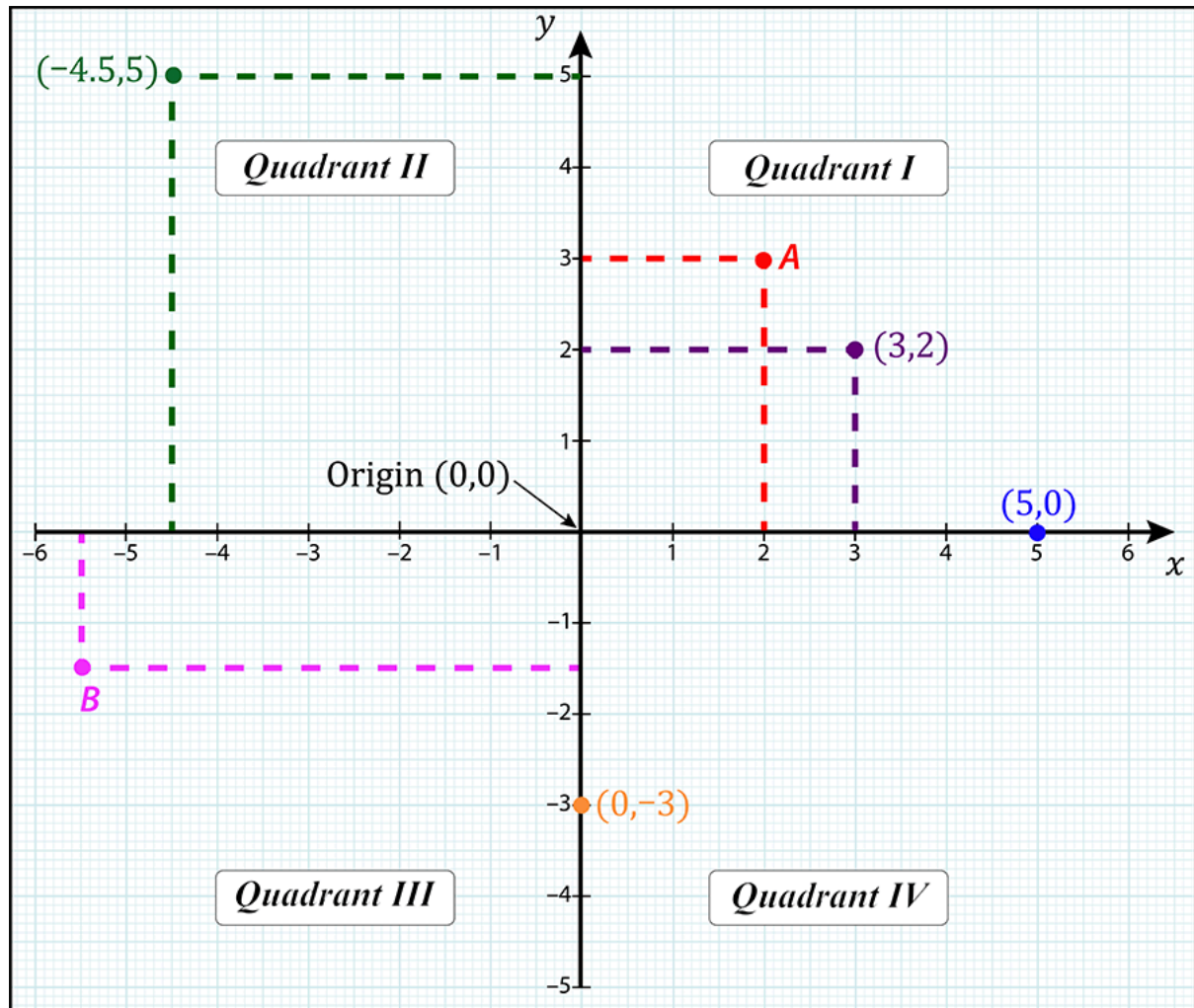
	Col1	Col2	Col3	Col4
Row1	Arr[0][0]	Arr[0][1]	Arr[0][2]	Arr[0][3]	
Row2	Arr[1][0]	Arr[1][1]	Arr[1][2]	Arr[1][3]	
Row3	Arr[2][0]	Arr[2][1]	Arr[2][2]	Arr[2][3]	
Row4	Arr[3][0]	Arr[3][1]	Arr[3][2]	Arr[3][3]	
⋮					

La figura que vemos a la izquierda representa como algunos lenguajes de programación entienden las coordenadas de una matriz. Una matriz puede ser definida en realidad como un sistema de mapeado en 2D donde situar cosas, como es el caso de nuestro mapa. El punto de origen de la matriz (0.0) se sitúa en la esquina superior derecha y el **primer valor**, que solemos llamar **X**, es la **altura** la cual se mueve de forma

creciente en cantidad pero ascendente en posición (cuanto más abajo en la matriz más alto el índice). Por otro lado, el **segundo valor**, conocido como **Y**, es la **anchura**. Dicho valor se mueve de forma creciente de izquierda a derecha. Si por ejemplo nos referimos a la coordenada (2,1) quiere decir que está tres posiciones hacia abajo (dado que 0 es una coordenada válida) y dos posiciones a la derecha, en la foto de referencia sería la (Row3,Col2). De nuevo cabe remarcar que **así es como se comportan las**

“coordenadas” en un espacio 2D dentro de una matriz en algunos lenguajes de programación como pueden ser **Java o C#** que es el lenguaje que estamos usando.

Por otro lado tenemos Unity. Unity es un motor 3D que usa principalmente un sistema de coordenadas cartesianas para casi todo lo que proyecta y renderiza en su entorno. Esto quiere decir que el sistema que usamos para ubicar un punto en el espacio es muy diferente al que usa una matriz en el lenguaje de programación.



Este sistema mucho más extendido a niveles no tan técnicos como pueden ser los que maneja un programador tiene el origen de sus coordenadas en la esquina inferior izquierda (ignorando los valores negativos). Su **eje X** se mueve de forma creciente de **izquierda a derecha** y su **eje Y** se mueve de forma creciente de **abajo a arriba**. Esto es muy diferente a como lo hace el lenguaje de programación.

Si superponemos un sistema a otro veremos que tenemos un problema. Las coordenadas no se corresponden. Imaginemos por ejemplo que nuestro personaje se inicia en 0.0. Esto indica que en la matriz de programación estará en la esquina superior izquierda y en el entorno 2D estará en la esquina inferior izquierda. Ahora nuestro jugador pedirá al personaje que se mueva **una posición hacia abajo** (modificando en la **matriz el eje X** pero en el **entorno 2D el eje Y**). Lo cual dará como resultado en la matriz la posición (1.0) y en el

entorno 2D la posición (0,-1). Como vemos la posición de la coordenada cambia tanto en el signo indicando que es una coordenada negativa como en la posición del valor.

Ahora vamos a intentar mover a nuestro personaje desde dicha posición a una coordenada a la derecha. En la matriz pasaremos a estar en la posición (1.1) y en el entorno 2D en la posición (1,-1).

INICIO	MOVIMIENTO	FIN
Matriz (1,0) - 2D (0,-1)	DERECHA	Matriz (1,1) - 2D (1,-1)

Si continuamos dando pasos veremos que ocurre:

INICIO	MOVIMIENTO	FIN
Matriz (1,1) - 2D (1,-1)	DERECHA	Matriz (1,2) - 2D (2,-1)
Matriz (1,2) - 2D (2,-1)	ABAJO	Matriz (2,2) - 2D (2,-2)
Matriz (2,2) - 2D (2,-2)	ABAJO	Matriz (3,2) - 2D (2,-3)

Imaginemos que tenemos una matriz de 10x10 y nuestro personaje inicia en la posición superior derecha Matriz(0,9) - 2D(9,9):

INICIO	MOVIMIENTO	FIN
M(0,9) - 2D(9,9)	ABAJO	M(1,9) - 2D(9,8)
M(1,9) - 2D(9,8)	ABAJO	M(2,9) - 2D(9,7)
M(2,9) - 2D(9,7)	IZQUIERDA	M(2,8) - 2D(8,7)
M(2,8) - 2D(8,7)	IZQUIERDA	M(2,7) - 2D(7,7)
M(2,7) - 2D(7,7)	ABAJO	M(3,7) - 2D(7,6)

En este caso es más complejo seguir la traza del movimiento. Si bien es posible dado que lo que hacemos es mover el puntero en dos sitios en paralelo lo que buscamos es ser capaces de recrear esto en pantalla de forma sencilla sin necesidad de tanta transformación.

Nuestro objetivo es, al igual que con la tabla, mostrar en pantalla por un lado la representación gráfica (coordenadas cartesianas) de forma intuitiva al jugador y por otro lado controlar nosotros en código donde se encuentra el puntero. Como hemos visto el movimiento es el mismo, si nos movemos abajo da igual donde se encuentre el puntero o el personaje, ambas entidades van a desplazarse una casilla hacia abajo. La dificultad real viene a la hora de pintar el mapa. Una vez el mapa ha sido interpretado por el código y dibujado de forma consecuente con la matriz nos dará igual a donde se mueva el personaje, la representación gráfica en Unity se moverá siempre que la matriz lo permita. Es decir,

nuestro objetivo es dibujar el mapa de forma correcta, no traducir las coordenadas en cada movimiento del personaje.

Los gráficos de esta sección han sido sacados de varios artículos instructivos de diferentes webs que se encuentran listadas en la sección de referencias.

3.2.2 - Solución

```
public void GenerateGrid()
{
    GridWidth = map.width;
    GridHeight = map.height;

    int UnityX = 0;
    for (int y = GridHeight - 1; y >= 0; y--)
    {
        int UnityY = 0;
        for (int x = 0; x < GridWidth; x++)
        {
            GameObject NewTile;
            if (map.Empty(y, x)) ...
            else ...

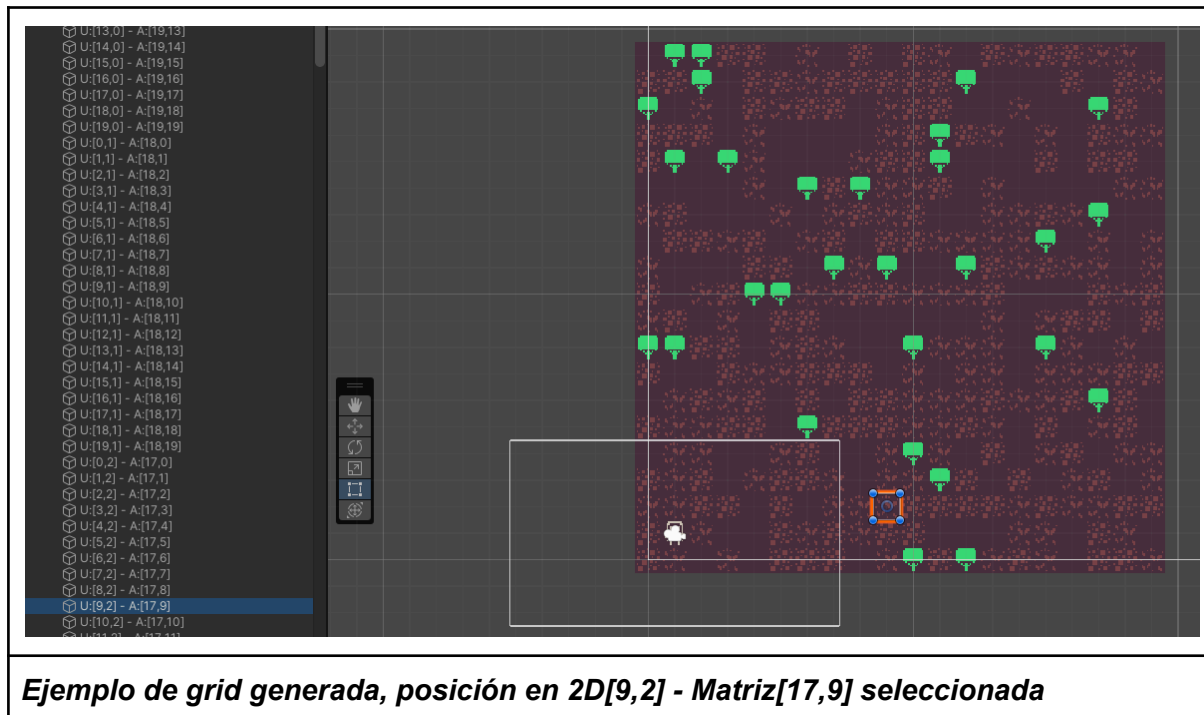
            float posX = x * TileSize;
            float posY = (GridHeight - 1 - y) * TileSize;

            NewTile.transform.position = new Vector2(posX, posY);
            NewTile.name = "U:[" + UnityY + "," + UnityX + "] - A:[" + y + "," + x + "]";
            UnityY++;
        }
        UnityX++;
    }
}
```

Ignoramos la parte colapsada del IF que es para saber que tipo de tile generar.

Lo que buscamos es dibujar el mapa de forma correcta. Como hemos visto en las tablas de la sección anterior, existe una relación matemática clara entre las coordenadas. Vamos a usar las coordenadas cartesianas de referencia. Primero, el eje X (anchura) se mantiene inalterable en su valor pero cambia de posición. En la matriz de programación vemos el eje X de la anchura en el segundo valor dado que la anchura es medida por la columna en que nos encontramos mientras que en la interpretación 2D vemos el eje X en el primer valor. Luego tenemos la dificultad de mapear correctamente el eje Y para la altura. En el caso de las coordenadas cartesianas es el lugar donde nos encontramos (por ejemplo 2 para indicar que estamos tres posiciones para arriba) vemos que en la matriz es el primer valor pero corresponde a la operación matemática de (total - posición cartesiana de Y - 1). Es decir, si nuestra matriz es de 20x20 (va de 0.0 a 19.19) y nos encontramos en las coordenadas cartesianas para (9,2) indicando 10 posiciones a la derecha y 3 posiciones para arriba, estaremos en las coordenadas de la matriz (17,9) viendo que el del eje X se respeta, movido de sitio, y el eje Y es (20-2-1). 20 el total de medida del mapa, 2 la posición actual y 1 que se resta del total por que las posiciones empiezan en 0.0.

Con eso en mente podemos ejecutar el código que vemos en el inicio de esta sección. Vamos a recorrer la *grid* que es nuestra representación en 2D del entorno gráfico. Dicha malla tiene unos valores definidos de altura y anchura. Con estos valores podremos movernos correctamente y mapear las posiciones que nos traemos de la matriz. Por cada posición comprobaremos su valor en la matriz con `IsValid(x, y)` y en función de lo que tenga crearemos nuevas instancias de los prefabs asignados y los dibujaremos en el entorno 2D. Vamos a llevar un registro de su posición con `U:[x,y]` para Unity y `A[x,y]` para la matriz dando con ello referencias exactas en cada una de sus medidas.



Ejemplo de grid generada, posición en 2D[9,2] - Matriz[17,9] seleccionada

Gracias a este sistema podemos pintar el mapa al inicio de la creación del escenario. Una vez pintado tenemos dos referencias a nuestro personaje.

Por un lado tenemos una referencia visual que el jugador entiende e interpreta de forma sencilla. El dibujo del personaje que vemos en la esquina inferior izquierda en la imagen superior. Dicha representación vale para que el jugador sepa dónde se encuentra dentro del mundo. Esta representación está controlada por unos métodos de nuestra clase `TileController`, dicho método actualiza la cámara y el tile del personaje si es posible moverlo.

Por otro lado tenemos el puntero que definimos en la clase `Map`. Dicho puntero es el que controla realmente si puede moverse el personaje pero nuestro jugador no ve eso en ningún momento, lo que ve es la representación gráfica equivalente dentro del entorno 2D.

Es decir, vamos a mover en paralelo tanto el cursor en la matriz como el personaje en el entorno 2D lo cual permite controlar qué sucede mediante código de forma eficiente y sencilla dentro de unas clases controladas y a su vez darle la sensación al jugador de que todo sucede en un entorno renderizado.

3.3 - Movimiento

```
void Update()
{
    CheckActions();

    if (HaveActions && !DoingSomething)
    {
        if (Input.GetKeyDown(KeyCode.W))
        {
            MoveUp();
        }
        else if (Input.GetKeyDown(KeyCode.S))
        {
            MoveDown();
        }
        else if (Input.GetKeyDown(KeyCode.A))
        {
            MoveLeft();
        }
        else if (Input.GetKeyDown(KeyCode.D))
        {
            MoveRight();
        }
        else if (Input.GetKeyDown(KeyCode.E))
        {
            Interact();
        }
    }
}
```

Dentro del método Update() de nuestra clase TileController crearemos referencias a cuatro métodos separados para el movimiento. A su vez estos métodos estarán condicionados a que sea posible realizar acciones (a definir más adelante) o que el personaje no esté haciendo algo actualmente como puede ser interactuando con un lugar (a definir más adelante).

De esta forma podemos controlar bien en el código que sucede al pulsar una tecla. Cabe reseñar que en la POC el movimiento está controlado por teclas de forma que sea más sencillo y rápido hacer debug. En el proyecto final todo el movimiento será táctil y controlado mediante botones. Es decir, el método Update() apenas se usará en realidad. Esto es una decisión tanto a nivel jugable como a nivel técnico. Una de las dificultades de la creación de videojuegos (si

bien en nuestro proyecto no iba a ser de gran impacto) es la sincronización de frames. Unity tiene dos métodos, Update() y FixedUpdate() que se comportan de maneras diferentes dependiendo de valores como la cantidad de frames y otros aspectos complejos de desarrollo de videojuegos. Para evitar tener que centrarnos en esos problemas se hará todo mediante botones y se relega el uso de las funciones Update() y FixedUpdate() a otras tareas de mantenimiento y no a procesos tan importantes como pueden ser el control del personaje.

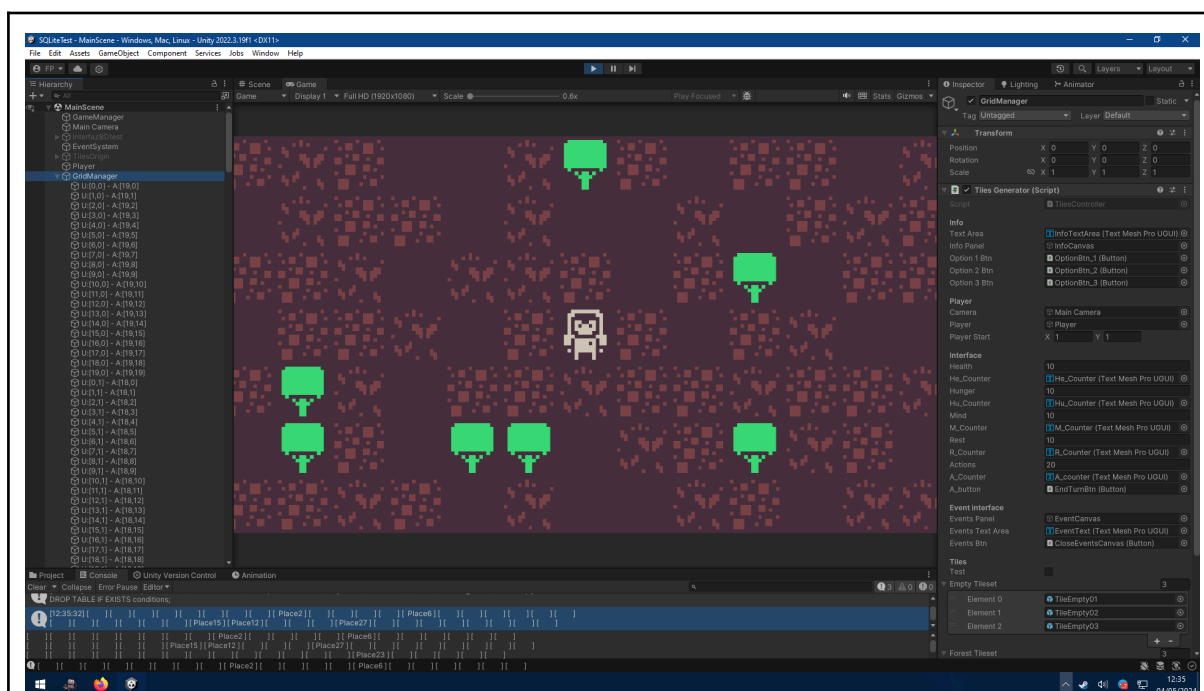
Los métodos lo que harán será tomar el atributo **transform.position** tanto de la cámara de juego como del tile del jugador y actualizar su posición 16 píxeles en la dirección indicada por el movimiento.

Para ello, antes de realizar dicha acción, se usará el método IsValid(x, y) del objeto Map donde nos movemos. Si por ejemplo nos encontramos en (4,7) y queremos movernos una posición hacia arriba (3,7) se preguntará a la matriz si dicha posición es válida. En caso de

que nos devuelva una afirmación con el método actualizaremos tanto la cámara como el tile como el puntero de referencia.

```
public void MoveUp()
{
    int TargetX = map.PlayerPositionX() - 1;
    if (map.Valid(TargetX, map.PlayerPositionY()))
    {
        RestOneAction();
        Camera.transform.position += Vector3.up * MovementUnit;
        Player.transform.position += Vector3.up * MovementUnit;
        map.SetPlayerPosition(TargetX, map.PlayerPositionY());
    }
}
```

En este caso vemos que estamos haciendo uso del método PlayerPositionY() de nuestro Map. Existen dos métodos para recuperar el eje X y el eje Y del vector donde se encuentra el puntero del jugador. Gracias a que hemos dibujado el mapa correctamente al inicio de la partida y a este tipo de interacciones entre el puntero y el entorno gráfico será sencillo dar la sensación al jugador de que está moviendo al personaje dentro del entorno gráfico. En realidad lo que está haciendo es mover el puntero y en función de si es posible moverlo, mover también junto a dicho puntero la cámara de Unity y el tile del personaje.



Resultado de dibujar el mapa visto en el editor de Unity en modo juego

4 - Interactuar con el entorno

4.1 - Leer contenido adyacente

Ya hemos logrado varios de los objetivos claves de la POC, crear y conectar con la base de datos, recuperar toda su información y traerla a objetos de Unity. Una vez tenemos dichos objetos hemos logrado también dibujar en pantalla el mapa y hacer que un personaje se mueva por dicho mapa. Lo siguiente que necesitamos es que nuestro personaje pueda interactuar con lo que tiene alrededor. Es decir, si nos encontramos adyacentes a una localización (lugar) será capaz de recuperar y mostrar de alguna manera el contenido de dicha localización.

4.1.1 - Código

Para lograr lo que buscamos lo primero que tenemos que hacer será añadir un nuevo método a nuestra clase **Map()** el cual nos permite buscar en los alrededores del personaje.

```
public Place SearchForPlaces()
{
    for (int i = 0; i < Directions.GetLength(0); i++)
    {
        int SearchX = (int)PlayerPointer.X + Directions[i, 0];
        int SearchY = (int)PlayerPointer.Y + Directions[i, 1];
        if (InBounds(SearchX, SearchY))
        {
            if (map[SearchX, SearchY] != null)
            {
                return map[SearchX, SearchY];
            }
        }
    }
    return null;
}
```

Este método toma como valores la posición del puntero de nuestro jugador y mira que hay alrededor usando un array de coordenadas.

```
3 references
private int[,] Directions = new int[,] { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };
```

Si bien este código no es demasiado correcto dado que en el momento que encuentre algo devolverá lo encontrado. Es decir, si tenemos por ejemplo dos lugares adyacentes, uno

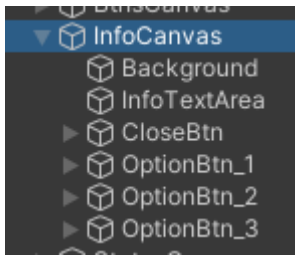
4.1.2 - Unity

Dentro del método `Update()` vamos a añadir un método llamado `Interact()` que se lanzará al pulsar la tecla E, en el futuro esto también será controlado todo por botones como ya mencionamos a la hora de definir el movimiento.

The image is a screenshot of a Unity development environment. The top half shows a game scene with a dark, pixelated forest background. In the foreground, there are two white rectangular UI elements with black text: "Place10_option1" and "Place10_option2". A "CLOSE" button is visible in the bottom right corner of the scene. The bottom half of the image shows the Unity console, which is displaying several log messages. The messages include database path information, a warning about a missing "create.sql" file, and a list of place names and descriptions. The console also shows the Unity version control and animation tabs. The overall layout is typical of a game development IDE.

De momento todo lo que estamos haciendo es por consola pero el objetivo de la POC va más allá, queremos ver si es posible crear un flujo entendible por el jugador igual que se hizo a la hora de dibujar el mapa y transformar el sistema de puntero-matriz en un sistema de mapa-personaje.

4.2 - Interfaz gráfica



Vamos a crear un panel nuevo y dentro definiremos diferentes componentes. Un área de texto para mostrar la información de nuestro lugar y tres botones para las posibles acciones que puede tener cada lugar del juego.

Al inspeccionar un lugar se activará mediante el código de `Interact()` este panel y se mandará la información del objeto para ser impresa en el área de texto. Además vamos a hacer uso de la variable

DoingSomething mencionada en el [punto 3.3](#) para bloquear el movimiento del jugador mientras explora.

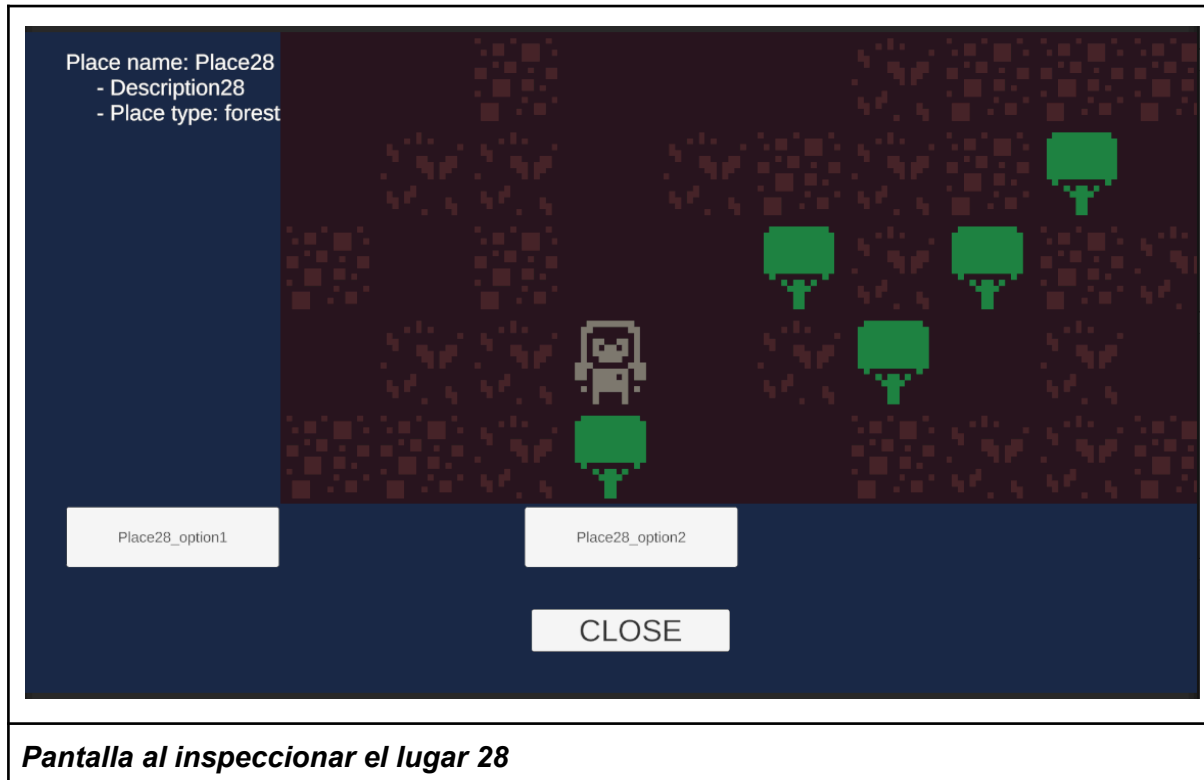
```
if (NearPlace.Option1 != null)
{
    Option1Btn.gameObject.SetActive(true);
    TextMeshProUGUI Text1 = Option1Btn.GetComponentInChildren<TextMeshProUGUI>();
    Text1.SetText(NearPlace.Option1);
    Item item = NearPlace.Item1;
    Option1Btn.onClick.RemoveAllListeners();
    Option1Btn.onClick.AddListener(() => PickItem(NearPlace, item));
}
```

El código para añadir las opciones es ligeramente más complejo. Lo que hacemos es comprobar si las opciones no son null y en caso de no serlas asignaremos su texto a un botón. Además de ello vemos si tienen item asignado y dicho item lo pasamos junto al método `PickItem()` como un nuevo Listener al botón correspondiente. Esto hace que se vayan generando una sucesión lógica de eventos que da la sensación al jugador de estar explorando el lugar y relacionando todo.

En este punto vemos ya cómo se va construyendo todo desde nuestra base de datos.

26	26	Place26	Description26	forest	Place26_option1	13	NULL	NULL	NULL	NULL
27	27	Place27	Description27	forest	Place27_option1	14	Place27_option2	NULL	NULL	NULL
28	28	Place28	Description28	forest	Place28_option1	16	Place28_option2	NULL	NULL	NULL
29	29	Place29	Description29	forest	Place29_option1	10	NULL	NULL	NULL	NULL

Aquí vemos los lugares del 26 al 29. Vamos a sentarnos en el **lugar 28**. Por ahora los textos son un “placeholder” que solo usamos de referencia sencilla para ver en la interfaz que todo está correcto. Vemos el nombre (Place28) la descripción (Description28) y que tiene dos opciones, la option1 y la option2. De estas opciones solo la opcion1 nos dará un objeto (el item 16). Gracias a cómo hemos estructurado la base de datos es posible rellenar esa información sin necesidad de acceder al código para nada.



5 - Panel de estados

5.1 - Definición

Como ya vimos en el documento de Análisis Iniciales, nuestro juego tendrá 4 diferentes marcadores de estado cuyo comportamiento será definido en un futuro. De momento nos vamos a centrar en que los marcadores son numéricos y empezarán todos a 10. Como extra tenemos el marcador de acciones que empezará en 20 y define la cantidad de acciones que podemos hacer en un turno (mover, interactuar).

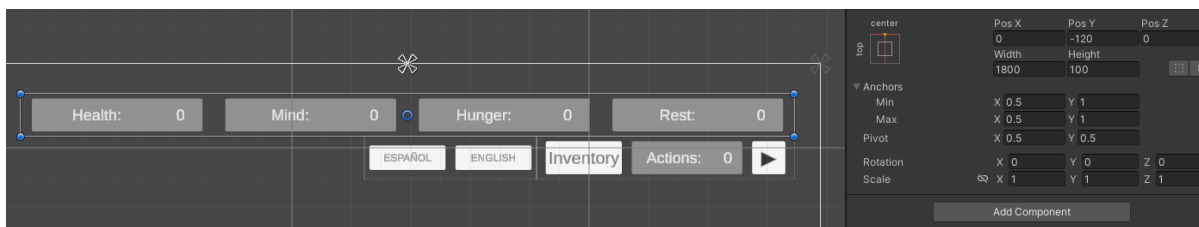
Los estados del juego son **Vida**, **Cordura**, **Hambre** y **Cansancio**. Todos empezarán a 10 y si cualquiera de ellos llega a 0 terminará la partida.

En siguientes secciones de la POC se establecerá un sistema de inventario para interactuar con los objetos pero lo primero que tenemos que definir es que todo funciona.

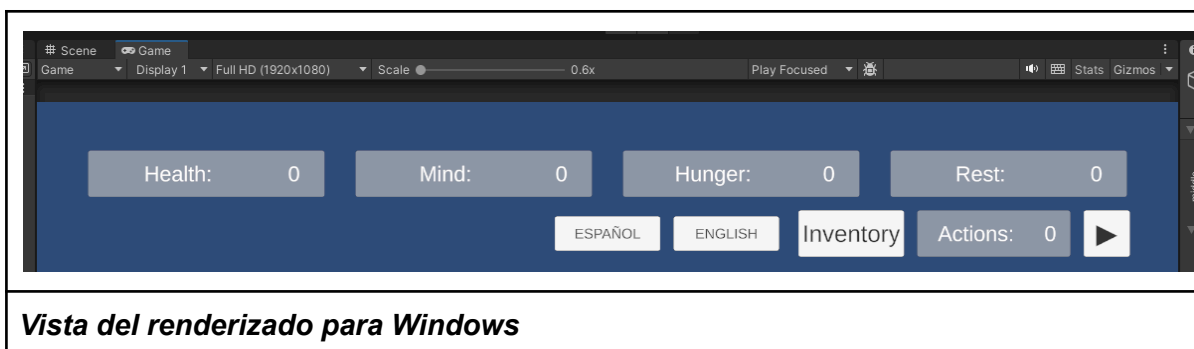
Vamos a aprovechar también esta sección para probar correctamente los componentes **Transform** de cara a la **adaptabilidad** (responsive) de los objetos en el editor. Hasta ahora no nos hemos preocupado demasiado en la POC de donde quedan nuestros componentes en cuanto a renderización.

5.2 - Adaptabilidad de la interfaz

Se tendrán en cuenta las posiciones de cada elemento en la interfaz usando valores numéricos más exactos y un correcto posicionamiento tanto de las anclas de referencia como del pivote.



Como ejemplo tenemos el frame donde se engloban los estados. Hemos creado diferentes componentes de texto, imágenes etc para generar la composición y todos los componentes se encuentran en un objeto “frame” para posicionarlos. Como vemos en el inspector del editor el ancla es centro arriba (punto de referencia del objeto padre) y el pivote se encontrará en el centro del objeto hijo. Luego nos posicionamos 120 píxeles para abajo en el eje Y en referencia a nuestro ancla. Todos los componentes internos han sido posicionados de igual manera.



Podemos apreciar que todos los componentes mantienen tanto sus dimensiones como su posición.

5.3 - Modificar los contadores

Lo siguiente que queremos es que los contadores respondan a la interacción con los objetos.

De momento lo que vamos a hacer es que al interactuar con un lugar y elegir una acción tenga lugar directamente lo que diga la condición del objeto asociado a dicha acción, si tiene. Con esto además vamos a gestionar un sistema rápido para ver como interpretar las acciones.

Este paso es muy importante en la POC dado que en el futuro cuando asentamos los atributos de nuestra base de datos real tenemos que tener muy claro cómo funcionará todo para que tenga sentido al traducirse desde tablas de SQLite y atributos a objetos de C# para su tratado en Unity y cohesión con el juego.

En la base de datos de la POC tenemos los atributos de los efectos (conditions) como un text “x_y” donde “x” es el modificador y la cantidad e “y” el atributo al que afecta. La **m** significa restar y la **p** sumar.

```
2 references
public void UseCondition(Condition actual)
{
    string[] split = actual.Effect.Split('_');
    string status = split[0].Trim();
    string action = split[1].Trim();

    char modifier = status[0];
    string numberStr = status[1..];
    numberStr = numberStr.Trim();
    int quantity = int.Parse(numberStr);

    switch (action)
    {
        case "health":
            if (modifier == 'm')
            {
                for (int i = 0; i < quantity; i++)
                {
                    MinusOne(He_Counter);
                }
            }
            else if (modifier == 'p')
            {
                for (int i = 0; i < quantity; i++)
                {
                    PlusOne(He_Counter);
                }
            }
            break;
        case "mind":
            if (modifier == 'm')
```

```
s > StreamingAssets > insertConditions.txt > data
NAME;DESCRIPTION;EFFECT
Injured;It's just superficial damage.;m3_health
Basic healing;You feel a little better.;p1_health
Horried;You feel like something is chasing you, but there is nothing here but you.;m3_mind
```

Por ejemplo los tres primeros efectos que añadimos. El primero restará 3 de vida, el segundo sumará 1 de vida y el tercero restará 1 de vida.

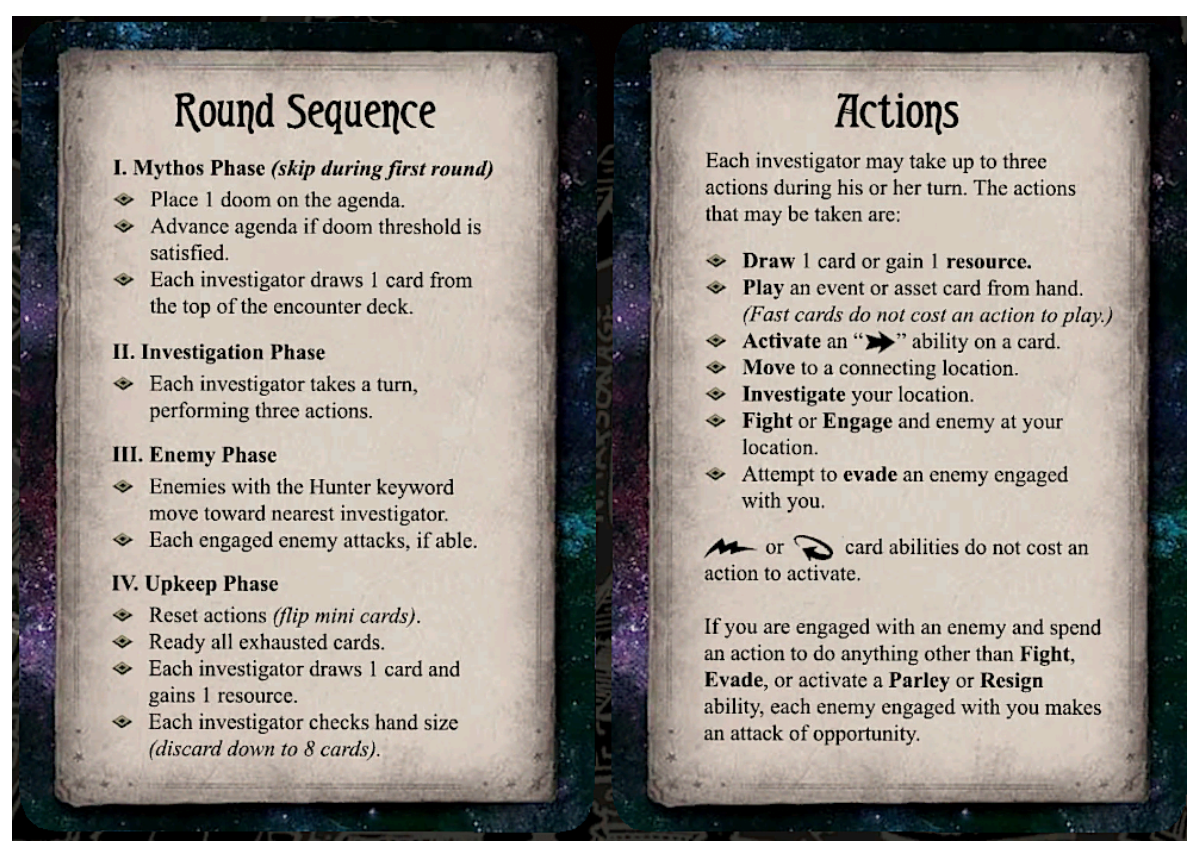
El código que vemos arriba recuperará la cadena Effect asociada al objeto, la separará e interpretará de forma que se vea reflejado en la interfaz. Los métodos PlusOne() y MinusOne() alteran el texto en pantalla de los contadores.

6 - Turnos y eventos

6.1 - Mecánica de turnos

Como ya se ha mencionado en el pasado, nuestro juego será una mezcla entre juego de mesa y juego de ordenador. Estamos viendo, de momento, muchas mecánicas que asociamos a los juegos de ordenador tales como un amplio mapa, generación aleatoria, contadores etc. Llega la hora de entrar un poco más en lo que serán las partes de “juego de mesa”.

Usualmente los juegos de mesa van por rondas o turnos, cada juego tiene un objetivo ya sea competitivo (los jugadores compiten entre sí para tener un ganador) o cooperativo (los jugadores cooperan para ganar al tablero) y dichas rondas o turnos se estructuran normalmente en fases. Por ejemplo fase de mantenimiento, fase de movimiento, fase de acciones, fase de enemigos, repetir hasta finalizar el juego.



Round Sequence

I. Mythos Phase (*skip during first round*)

- Place 1 doom on the agenda.
- Advance agenda if doom threshold is satisfied.
- Each investigator draws 1 card from the top of the encounter deck.

II. Investigation Phase

- Each investigator takes a turn, performing three actions.

III. Enemy Phase

- Enemies with the Hunter keyword move toward nearest investigator.
- Each engaged enemy attacks, if able.

IV. Upkeep Phase

- Reset actions (*flip mini cards*).
- Ready all exhausted cards.
- Each investigator draws 1 card and gains 1 resource.
- Each investigator checks hand size (*discard down to 8 cards*).

Actions

Each investigator may take up to three actions during his or her turn. The actions that may be taken are:

- Draw** 1 card or gain 1 **resource**.
- Play** an event or asset card from hand. (*Fast cards do not cost an action to play.*)
- Activate** an “➡” ability on a card.
- Move** to a connecting location.
- Investigate** your location.
- Fight** or **Engage** and enemy at your location.
- Attempt to **evade** an enemy engaged with you.

⚡ or 🔄 card abilities do not cost an action to activate.

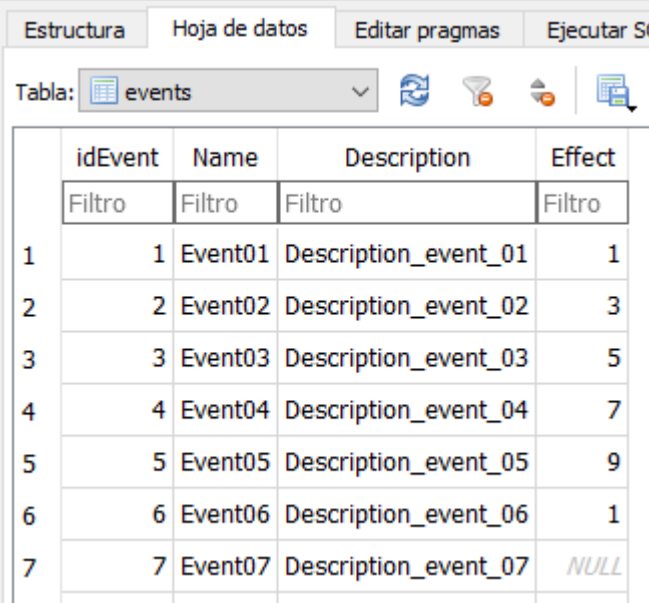
If you are engaged with an enemy and spend an action to do anything other than **Fight**, **Evade**, or activate a **Parley** or **Resign** ability, each enemy engaged with you makes an attack of opportunity.

Secuencia de ronda de Arkham Horror LCG (FFG)

Esta secuencia forma un intrincado mecanismo a la par de los objetivos de juego (derrotar a un enemigo, cumplir una misión) de forma que los jugadores la siguen para tener un flujo de juego y lograr la victoria o perder la partida de forma estructurada.

El objetivo de nuestro juego es simular algo parecido. Tendremos una “ronda de juego” que es como podemos definir a la sección de tiempo durante la cual el jugador disponga de acciones para moverse/interactuar. En paralelo a dicha ronda tenemos una “ronda de gestión” dado que las acciones como, por ejemplo, gestionar el inventario o usar el mapamundi no consumen acciones. Una vez que el jugador se ha quedado sin acciones y no quiere realizar nada más podrá pulsar el botón de “terminar ronda” y sucederán diversas cosas.

6.2 - Tabla eventos



	idEvent	Name	Description	Effect
	Filtro	Filtro	Filtro	Filtro
1	1	Event01	Description_event_01	1
2	2	Event02	Description_event_02	3
3	3	Event03	Description_event_03	5
4	4	Event04	Description_event_04	7
5	5	Event05	Description_event_05	9
6	6	Event06	Description_event_06	1
7	7	Event07	Description_event_07	NULL

Algo que no se tuvo en cuenta a la hora de diseñar la base de datos de pruebas fue, precisamente, los eventos.

Queremos que cuando nuestro personaje termine la ronda salte un evento aleatorio con algo de información. Imaginemos que tenemos varios tipos de evento (por ejemplo en función de lo cansado que está) o cualquier otro sistema que se nos ocurra diseñar. Gracias a la elaboración de una POC hemos podido llegar a encontrar un error de falta de datos enorme. Necesitamos otra tabla. Con ello la tabla evento será creada y tendremos en cuenta esta información

de cara a la creación de la base de datos real de nuestro juego definitivo.

Esta base de datos tiene información de los eventos con un nombre, texto descriptivo y la posibilidad de que el evento esté sujeto a un efecto. Por ejemplo, al finalizar la ronda nuestro personaje intenta descansar pero no lo logra al tener pocas provisiones y esto le quita un punto de cordura (la sanidad mental es muy importante en el juego).

Por ahora, al igual que en muchos otros apartados de la POC, esta información será sencilla y directa. Vamos a asociar un evento a un solo efecto y a no crear diferentes tipos de eventos. En el juego real se estudiará implementar diferentes eventos con, incluso, algo de interacción por parte del jugador. Aunque esto es una idea superficial, lo que se pretende conseguir es esa sensación de aleatoriedad que tienen los juegos de mesa a la hora de resolver la fase de eventos “de los malos”. El momento en que una carta aleatoria sacada de una baraja del caos puede desestabilizar toda la partida, todos tus buenos planes y

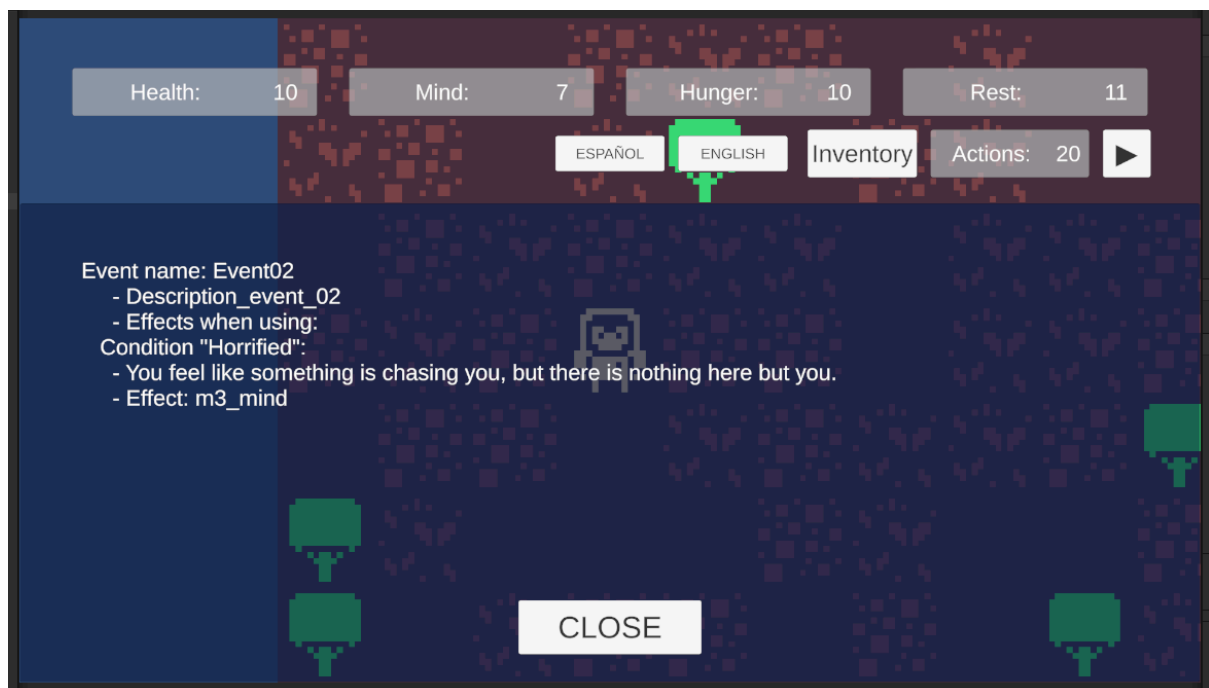
jugadas y a la vez mostrar lo duro que es el mundo del juego y que no todo está bajo control.

6.3 - Finalizar turno

Hemos retocado nuestros códigos gestores de base de datos para traer los eventos a objetos de C# y crear un array con todos nuestros eventos. También crearemos una clase llamada RandomEvents() que es donde almacenaremos dicho array. Esta clase tendrá, al menos en la POC, un solo método llamado GenerateEvent() que nos devolverá un evento de la lista y **lo eliminará de la lista**. De esta forma no se repiten los eventos.

```
public void EndTurn()  
{  
    DoingSomething = true;  
    EventsPanel.SetActive(true);  
    PhaseEvent actual = randomEvents.GenerateEvent();  
    EventsTextArea.text = actual.ToString();  
    if (actual.Effect != null)  
        UseCondition(actual.Effect);  
}
```

Al pulsar sobre el botón de terminar turno se lanzará un evento aleatorio en un panel nuevo que hemos creado, aplicaremos su condición (si la tiene) y luego restauraremos las acciones del jugador para que siga jugando.



Aquí vemos que el jugador se ha encontrado con el Event02 el cual le ha restado 3 de cordura. Antes tuvo un evento que le dio 1 más de descanso. Con estos eventos se verá modificado todo el juego y son uno de los principales hilos narrativos/jugables que dispondrá el proyecto final.

7 - Inventario

7.1 - Iconos

Gracias a la elaboración de la POC hemos descubierto otra carencia que tenía todo nuestro sistema. La base de datos y más concretamente la columna de ítem no disponía de un atributo para hacer referencia al icono que tendrá dicho ítem. Si queremos tener un inventario también queremos tener alguna manera de mostrar los objetos y, con ello, mostrar sus iconos.



	idItem	Name	Description	Effect	Icon
	Filtro	Filtro	Filtro	Filtro	Filtro
1	1	Item1	Description1	1	1
2	2	Item2	Description2	2	2
3	3	Item3	Description3	3	3
4	4	Item4	Description4	4	4

Al igual que con todo lo que estamos creando, por ahora vamos a hacer un tileset sencillo y descriptivo que valga para comprobar fácilmente las relaciones entre el índice que escribimos en la tabla y el icono.

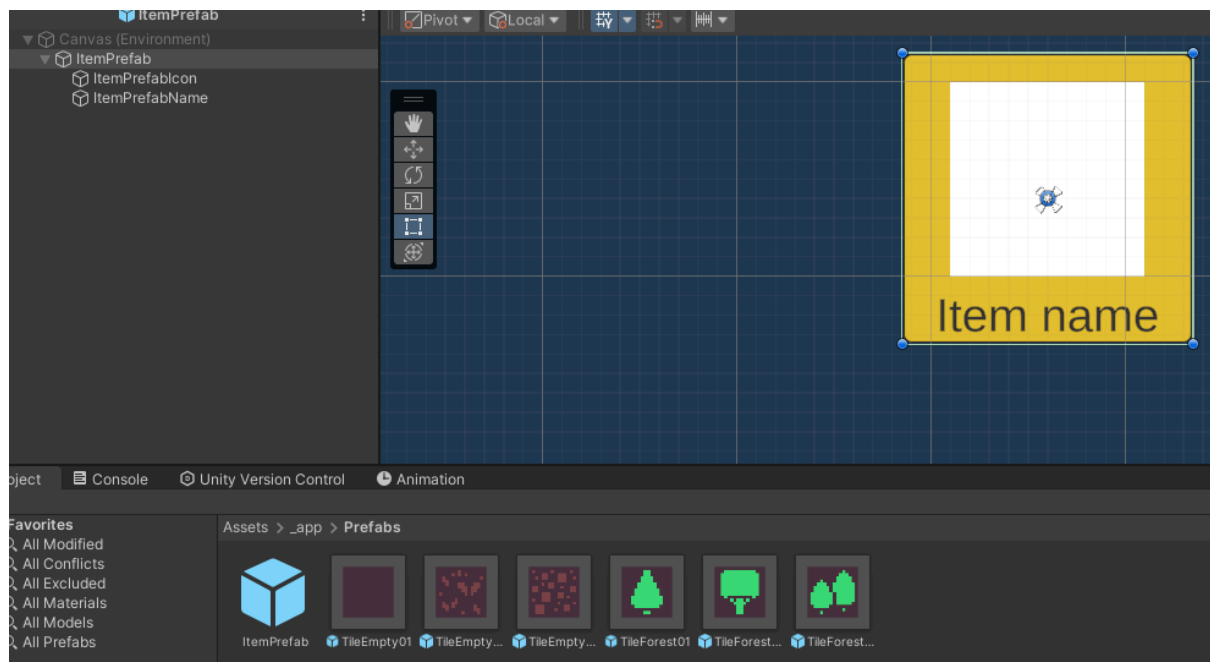
A cada item de la tabla le hemos añadido un valor entero llamado Icon que no tiene por que ser único, este valor hará

referencia a un icono dentro de nuestro tileset y posteriormente lo usaremos para dibujar el inventario.

De nuevo esto nos muestra claramente el valor de crear una POC. Imaginemos que ya hemos empezado nuestro proyecto real, el juego final. De repente llegamos hasta aquí después de haber realizado infinidad de trabajo y tener diseñada la interfaz, el estilo gráfico, la paleta de colores y otro largo etcétera de apartados. Hasta ese momento no nos damos cuenta de la carencia y debemos reestructurar, posiblemente, todo el sistema que hemos

realizado. Esto se traduce en una gran cantidad de tiempo de trabajo y, posiblemente, como suele suceder en estos casos, abre la posibilidad de crear errores en el camino. Es lo que se le llama “parchear” algo en lugar de solucionarlo desde sus inicios. Con esto en mente, cuando diseñemos la base de datos real de nuestro juego final podremos tener en cuenta todas las tablas y atributos que necesitamos desde el inicio.

7.2 - Prefab de ITEM



Una de las cosas más interesantes que nos permite Unity como motor de juego es la creación de **Prefabs**. Esto lo vimos en la [sección 3.1.2](#) al crear prefabs para los tiles que componen el mapa en el entorno 2D.

Un prefab es algo muy similar a un objeto en un lenguaje de programación. Creamos la carcasa y definimos algunos valores para posteriormente instanciar dicho objeto y modificarlo a placer. De esta manera disponemos de una “plantilla” que replicar. En el caso de nuestro objeto en particular vamos a crear un botón con una imagen de fondo, otra imagen para el icono y un texto para el nombre. Posteriormente vamos a, desde código, ir recuperando los objetos que tiene el jugador en inventario y dibujando dichos objetos en un canvas instanciando para ello diferentes prefabs de ItemPrefab y asignando los valores nombre e icono a dicho prefab.

7.3 - Singleton

Un singleton es una clase de C# que Unity instancia una sola vez y podemos tener activa entre diferentes escenas. Sus métodos serán estáticos y los podemos llamar desde diferentes secciones del código de forma que tengamos acceso a ella globalmente.

Un singleton se usa comúnmente para definir objetos o mecánicas que deben permanecer inmutables en los cambios de escena como puede ser, en nuestro caso, el inventario del jugador, el jugador mismo o un control de sonido. El ejemplo del control de sonido es el más comúnmente usado cuando se busca información en internet acerca de los singleton y una sencilla forma de entenderlo. Pongamos que nuestro jugador abre una puerta, pero no una puerta que comunica dos salas dentro de la misma escena y se mueve entre habitaciones si no una puerta con animación que llevará a una pantalla de carga y nos moverá entre escenas. Si tenemos un objeto de audio de fondo siendo reproducido, al salir de la escena nuestro objeto se destruirá y dejaremos de escuchar la música. Lo que se pretende es que la música no pare de reproducirse de manera intermitente entre cambios de escenas. Para ello son los singleton.

```
3 references
public static InventoryManager Instance
{
    get
    {
        if (_instance == null)
        {
            _instance = FindObjectOfType<InventoryManager>();

            if (_instance == null)
            {
                GameObject inventoryObj = new GameObject("InventoryManager");
                _instance = inventoryObj.AddComponent<InventoryManager>();
            }
        }

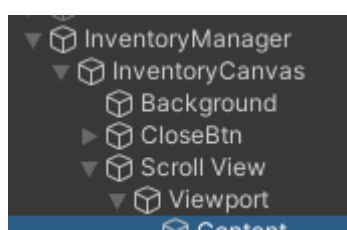
        return _instance;
    }
}
```

Todos los singleton comienzan con este método para generar la instancia. Si no existe, la crean. Si existe, la buscarán y la asignan a la clase del singleton para hacer recuperar la referencia.

En nuestro caso en particular hemos hecho que el InventoryManager sea el singleton pero después de trabajar con la POC hemos podido revisar diferentes opciones para el juego final que, de nuevo, nos van a ahorrar mucho tiempo cuando tengamos que implementar.

Seguramente el único singleton que tengamos en el juego final sea la clase del jugador como tal y dentro guardaremos información como su inventario, los valores de sus estados y otros datos relevantes similares.

7.4 - Sección de inventario



Crearemos una composición para el inventario con un nuevo panel y dentro del panel estableceremos una sección con un ScrollView, un Viewport y un panel Content.

Dentro de toda esta composición se configura el comportamiento visual del inventario. Parámetros tales como los bordes, el tipo de scroll etc.

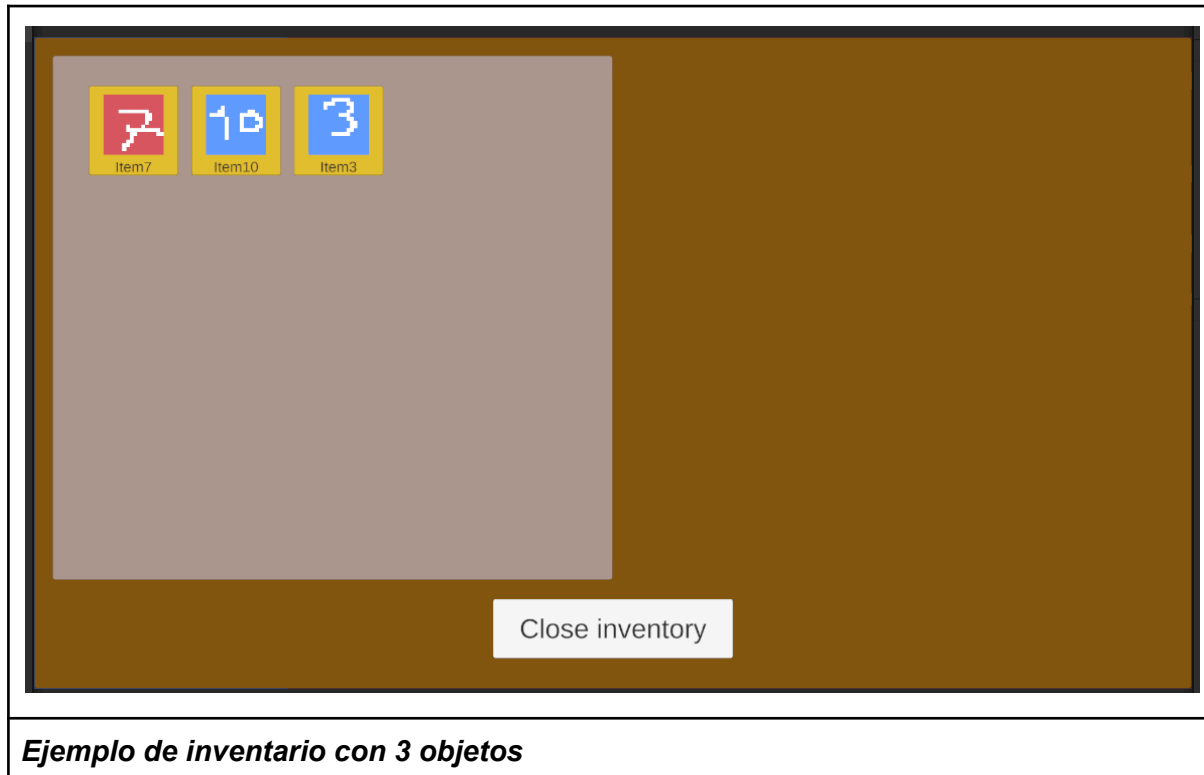
Cambiaremos nuestro código para que en lugar de aplicar los efectos de los items al explorar un lugar se llame al método AddItem(Item) del inventario y lo añada a un array de Items.

```
foreach (Item i in items)
{
    Debug.Log(i);
    GameObject NewPrefab = Instantiate(Prefab);
    NewPrefab.transform.SetParent(content, false);
    TMP_Text newItemText = NewPrefab.GetComponentInChildren<TMP_Text>();
    newItemText.text = i.Name;
    Transform itemIconTransform = NewPrefab.transform.Find("ItemPrefabIcon");
    var itemIcon = itemIconTransform.GetComponentInChildren<Image>();

    Sprite tileSprite = TileSet[i.Icon - 1];

    itemIcon.sprite = tileSprite;
}
```

Cuando abrimos el inventario recorreremos el array de items e iremos instanciando prefabs de ItemPrefab por cada item dentro del array, luego añadiremos dichos prefab al Content que se encontraba dentro del ScrollView. Recuperamos los valores de nombre e icono del prefab y los asignamos a los valores del objeto que toca pintar en cada iteración.



8 - Idiomas

8.1 - Tablas de localización

Idiomas ☆ Guardado en Drive

Archivo Editar Ver Insertar Formato Datos Herramientas

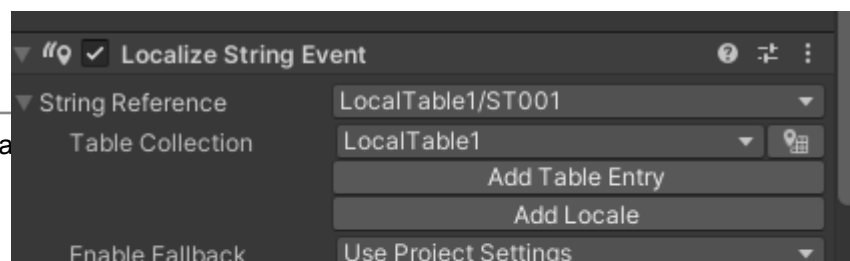
Menús

	A	B	C	D
1	Key	Id	English(en)	Spanish(es)
2	ST001	1	Health	Vida
3	ST002	2	Sanity	Cordura
4	ST003	3	Hunger	Hambre
5	ST004	4	Rest	Cansancio
6	ST005	5	Action	Acción
7	ST006	6	Inventory	Inventario
8				

Una de las principales ventajas que tiene trabajar con un motor 3D como Unity es la posibilidad de trabajar con tablas de localización.

Una tabla de localización es un archivo comúnmente en formato .csv que tiene una key, la cual usaremos en el editor, y una id que deben ser únicos. Luego tendrá una columna para cada uno de los idiomas. En un futuro si se quiere modificar se añadirá una columna más y tendremos disponible ese

idioma.



Una vez en Unity podemos ir a cualquier componente y en lugar de escribir los datos del interior del componente vamos a hacer una referencia a la tabla de localización. Pondremos en **Entry Name** la Key asignada a esa columna y listo. Con esto los componentes se traducen automáticamente.



Texto en inglés, datos en tabla de localización

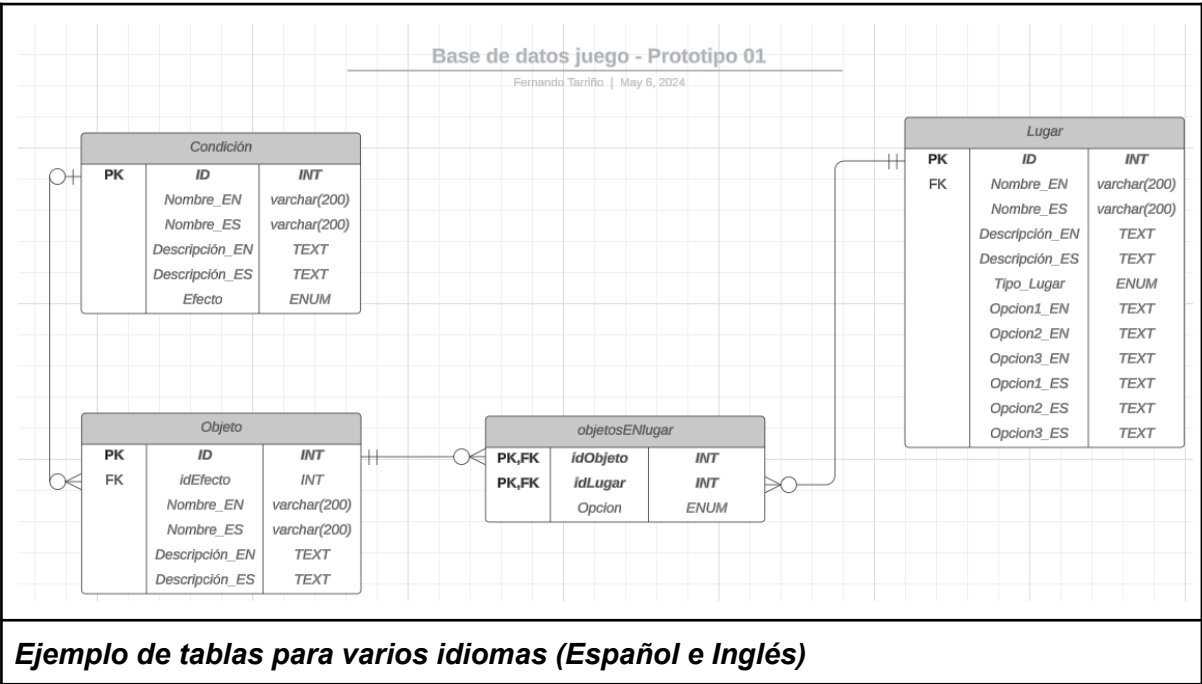


Texto en castellano, datos en tabla de localización

8.2 - Textos en la base de datos

Las tablas de localización son herramientas de gran potencia siempre que trabajemos con datos internos pero nos fallan a la hora de tomar datos externos. En el caso de nuestro proyecto vamos a tener todos los datos de texto, nombres etc en tablas. Por ello debemos crear unas tablas específicas o campos para los diferentes idiomas.

La idea para el proyecto final será que nuestras tablas tengan campos de, por ejemplo, Name_ES y Name_EN para el nombre, Description_ES y Description_EN para la descripción etc. De esa forma y al igual que el resto de objetos relacionados con la base de datos, se podrán añadir y modificar datos sin necesidad de código.



9 - Referencias y fuentes

En esta sección se encuentran listados, sin orden en particular, todos los recursos usados para la realización tanto de este documento como del trabajo que implica.

[01]: <https://www.sqlite.org/docs.html>

[02]: <https://github.com/robertohuertasm/SQLite4Unity3d>

[03]: <https://www.youtube.com/watch?v=u-BOb69lhAI>

[04]:

<https://medium.com/@kevinjulu/2d-arrays-a-comprehensive-guide-for-programmers-40337aa3c77f>

[05]: <https://www.skillsyouneed.com/num/cartesian-coordinates.html>

[06]:

https://www.reddit.com/r/arkhamhorrorlcg/comments/oj7f65/keywords_for_cards_and_deckbuilding/

[07]: https://www.youtube.com/watch?v=AoD_F1fSFFg