Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Programación del juego de rol de tablero multijugador Versus Guild

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Torres Badia, Guillem

Tutor: Ramos Peinado, Enrique

2014/2015

Resumen

En este documento se describe el proceso de desarrollo de un videojuego multijugador local del género *RPG* (*Role Playing Game*), concretamente un *RPG Tactics*. Algunos juegos conocidos de este subgénero son: *Advance Wars*, *Final Fantasy Tactics* o *Fire Emblem*.

Generalmente, los *RPG* suelen ser juegos por turnos, lo que resta importancia a la eficiencia temporal de los algoritmos empleados. No obstante, su lógica suele ser más compleja que en otros géneros, debido a la variedad de personajes y ataques que estos suelen poseer. Esto es especialmente remarcable en los *Tactics*, ya que además de los elementos habituales de los *RPG*, existe un tablero por el que se mueven los personajes, añadiendo así una dimensión más al juego.

El entorno empleado para el desarrollo ha sido *Unity3D*, usando C# como lenguaje de programación.

Palabras clave: videojuego, *RPG*, *Tactics*, *Unity3D*, C#

Abstract

This document describes the development process of a local multiplayer RPG videogame, more specifically, a Tactic RPG. Some games of this subgenre are *Advance Wars*, *Final Fantasy Tactics* or *Fire Emblem*.

Usually, RPGs are turn-based games, which make time efficiency less important than in other kinds of games. However, their logic rules are also more complex than in other genres, due to the great variety of characters and attacks. This is especially remarkable in Tactic RPGs, as they do not only posses the typical RPG elements, but also a board where characters can move.

The tool used for the development is *Unity3D*, using *C#* as programming language.

Keywords: videogame, RPG, Tactic RPG, *Unity3D*, *C#*

Tabla de contenidos

1. Introducción

1.1 Descripción del proyecto

Versus Guild es un *RPG Tactics* en el que cada jugador controla un gremio compuesto por varios personajes, cada uno de los cuales pertenecerá a una Clase (a escoger entre seis) con unas habilidades determinadas.

Esta versión solo soporta partidas para dos jugadores humanos, de modo que cada uno controlará dos personajes y el objetivo será eliminar a los personajes del rival.

El sistema multijugador estará implementado de forma local, teniendo el control del computador el jugador del cual sea el turno. Esto es posible debido a que se trata de un juego por turnos, por lo que ambos jugadores no necesitan interactuar con el sistema simultáneamente.

Además, el proyecto se ha realizado teniendo en cuenta la ampliabilidad del mismo, estando preparado para que se añadan Clases, Habilidades y Técnicas con efectos que no aparecen en ninguna de las existentes. También es posible añadir efectos no previstos con facilidad.

1.2 Motivación

Versus Guild está basado en un juego de rol del foro miscelaneous.net, el cual tenía las mismas mecánicas que el videojuego desarrollado, solo que gestionadas por un *game master* humano.

Además de su detallada ambientación y lo bien definidas que estaban las clases de mercenario, el rol destacaba por la gran variedad de efectos que poseían sus técnicas o ataques, lo que unido a un sistema de combate por parejas, hacía que las posibilidades del juego fueran prácticamente ilimitadas.

Debido a ello, el autor de este proyecto decidió implementar un videojuego basado en el rol y vio en el TFG una oportunidad de hacerlo de una forma académica y disciplinada.

1.3 Elección de las herramientas de trabajo

La elección de las herramientas empleadas ha estado basada principalmente en la familiaridad del autor con las mismas.

El entorno, *Unity3D*, es un motor de videojuegos que se caracteriza por tener una interfaz muy amigable. Aunque el programa es más adecuado para el desarrollo de videojuegos en tiempo real, el paquete *UnityEngine.UI* cuenta con multitud de elementos de interfaz como botones, imágenes, etcétera, permitiendo el desarrollo de videojuegos como el que nos ocupa.

Unity3D tiene compatibilidad con tres lenguajes de programación diferentes: *Javascript*, *Boo* y *C#*. De ellos, el escogido fue *C#*, ya que la orientación a objetos es sumamente adecuada para el desarrollo de un proyecto que requiere gran estructuración, como es el caso de *Versus Guild* (especialmente en la capa de lógica). Además, *C#* es similar a *Java* –lenguaje con el que el autor está muy familiarizado–, lo que facilitó en gran medida el aprendizaje del mismo.

Por último, los programas empleados para la edición de los gráficos y del audio han sido *Gimp* y *Audacity*, seleccionados por ser abiertos y gratuitos.

1.4 Relación con las asignaturas cursadas durante el grado

Para la realización de este proyecto se han empleado múltiples habilidades adquiridas a lo largo del grado en Ingeniería Informática.

En cuanto a los conocimientos más básicos, las competencias en programación orientada a objetos fueron adquiridas en asignaturas como *Introducción a la Informática y a la Programación* (IIP), *Programación* (PRG) y *Estructuras de Datos y Algoritmos* (EDA). Por otra parte, los conocimientos y habilidades referentes al desarrollo de interfaces fueron aprendidos en *Interfaces Persona-Computador* (IPC) e *Ingeniería del Software* (ISW).

Respecto a los conocimientos más avanzados, los principios de estructuración de código seguidos –tales como la elusión de valores literales en el código o la separación de los datos y los algoritmos en ficheros diferentes– provienen de *Introducción a la Programación de Videojuegos* (IPV) y, en menor medida, de *Ingeniería del Software* (ISW). Los conocimientos algorítmicos empleados, por otra parte, fueron adquiridos en *Algorítmica* (ALT), *Competición de Programación* (CACM) y *Grafos, Modelos y Aplicaciones* (GMA).

Por último, las herramientas empleadas también fueron usadas durante el transcurso de la carrera: *Unity3D y C#* fueron empleados para un proyecto conjunto de *Introducción a la Programación de Videojuegos* (IPV) y *Arquitectura y Entornos de Videoconsolas* (AEV), mientras que *Gimp* y *Audacity* se utilizaron en *Fundamentos de Sistemas Multimedia* (FSM).

Competencias	**Asignaturas**
Programación orientada a objetos	IIP, PRG
Principios de estructuración de código	IPV, ISW
Conocimientos sobre algorítmica	ALM, CACM, GMA
Creación de interfaces	IPC, ISW
Unity3D	IPV, AEV
Gimp y Audacity	FSM

2. Reglas del juego

Antes de entrar en detalles sobre la implementación del juego es necesario conocer en profundidad sus reglas, para poder comprender fácilmente las decisiones tomadas. En primera instancia se explicarán algunas reglas básicas, como el objetivo del juego o las acciones que existen. A continuación, se ahondará un poco con reglas sobre los ataques, la experiencia, etc. Por último, se detallarán las características de las clases de mercenario y los mapas disponibles.

2.1 Reglas básicas

Cuál es el objetivo

El objetivo del juego es acabar con los personajes del contrincante. Para ello, los personajes de ambos jugadores podrán moverse por el mapa y atacarse entre ellos, respetando los turnos.

Cómo moverse

El número de casillas que podrá moverse un personaje en cada turno está limitado por un dado, el cual se da al principio del mismo.

- Se permite mover menos casillas que las que indique el dado, incluyendo no moverse en absoluto.
- También es posible intercalar moverse con otras acciones (por ejemplo, se permite mover-atacar-mover).
- El dado oscilará entre 3 y 5, sujeto a modificadores.
- Una misma casilla puede ser ocupada por varios personajes.

Cómo atacar y qué acciones pueden realizarse

Cada turno, cada personaje podrá realizarse una única acción principal. Éstas son las siguientes:

- Atacar físicamente.
- Usar una técnica.
- Utilizar un edificio.

Las características de los personajes

Las características básicas de cada personaje vienen dadas por la clase a la que pertenece, aunque algunas de estas pueden mejorarse subiendo de nivel. Las características son:

- Vida: puntos que el personaje va perdiendo al recibir ataques enemigos. Al llegar a cero, el personaje es eliminado.
- Fuerza: determina el daño que causan los ataques físicos del personaje.
- Velocidad: determina el orden de movimiento. Las subidas o bajadas de velocidad afectarán al turno siguiente, nunca al actual. Adicionalmente, los personajes tendrán un 1% más de acertar (o fallar) sus ataques por cada punto de velocidad que tengan sobre su enemigo, con un máximo de ±10%.
- Arma: determina los efectos secundarios del ataque físico.

- Habilidad: característica pasiva del personaje. Pueden hacer que el personaje tenga más fuerza según determinadas situaciones o que sea inmune a ciertos estados, entre otros.
- Técnicas: listado de técnicas del personaje. Todas las clases tienen una técnica al nivel 1, con posibilidad de aprender más al subir de nivel.

2.2 Los ataques

Existen dos tipos de ataque: los ataques físicos y las técnicas.

Los ataques físicos

Ataques realizados usando el arma del personaje. El daño causado es igual a la fuerza del usuario -sujeto a modificadores- y tienen algunos efectos especiales dependiendo de las características del arma. Su precisión por defecto es de un 90% sujeto a modificadores.

Las técnicas

Son ataques especiales de cada clase, con efectos más variados que los de los ataques físicos. Tienen una cantidad de usos limitada y su precisión varía según la técnica en cuestión. Se pueden clasificar en tres tipos:

- Ofensivas directas: su principal objetivo es causar daño, el cual será igual al poder de la técnica, sujeto a modificadores. También pueden tener algún efecto adicional.
- Ofensivas indirectas: se basan en causar estados o disminuciones de características al contrincante.
- Defensivas/apoyo: se usan sobre el usuario o sobre el aliado. Sus efectos suelen ser curaciones o aumentos de características.

Por otro lado, estas formas de ataque tienen sus contrapartes defensivas: la Defensa en el caso de los ataques físicos y la Resistencia en el caso de las técnicas. Una combinación de ambas se denomina Protección. Éstas pueden ser aumentadas mediante el uso de técnicas o habilidades.

Al margen de estos factores, todos los ataques tendrán un 10% de resultar en golpes críticos, lo que multiplicará por 1.5 el daño causado por los mismos

Los elementos

Algunas clases y ataques tendrán afinidad hacia ciertos Elementos, que desarrollan relaciones de efectividad entre ellos. Así, si el Elemento del ataque es fuerte contra el Elemento del personaje que lo recibe, el daño se multiplicará por 1.5. Análogamente, si el ataque fuera débil contra el personaje, se multiplicará por 0.5. También se considera que un Elemento es débil contra sí mismo.

En esta versión solo existen dos elementos: Natura y Veneno, los cuales son fuertes entre sí.

El rango

El rango es una característica de los ataques (físicos o técnicos) que determina a que objetivos puede alcanzar. En esta versión existen los siguientes rangos:

- Rango base: el atacante tiene que estar en la misma casilla que el objetivo.
- +X: el atacante tiene que estar a X casillas o menos del objetivo para poder atacarle.
- =X: el atacante tiene que estar exactamente a X casillas del objetivo para poder atacarle.
- Campo: el ataque afecta a toda la casilla. Puede combinarse con rangos +X o =X, si bien por el momento no existe ningún ataque que lo haga.

2.3 Experiencia y niveles

Cada personaje acumulará puntos de experiencia a medida que vaya luchando, de modo que al alcanzar cierta cantidad subirá de nivel. Al subir de nivel, el personaje aumentará sus Vida y Velocidad, además de dar la posibilidad de aumentar 2 puntos de fuerza o aprender una nueva técnica. El nivel máximo es el 3, en el cual las técnicas disponibles serán dos, aunque solo se podrá elegir una de ellas o subir fuerza.

Las formas de ganar experiencia son las siguientes:

- Atacar o usar técnicas sobre el contrario: 10 puntos de experiencia.
- Usar técnicas sobre uno mismo o sobre el aliado: 8 puntos de experiencia.
- Recibir un ataque del enemigo: 4 puntos de experiencia.
- Fallar un ataque: 1 punto de experiencia.

La experiencia requerida para alcanzar el nivel 2 es de 40 y para el nivel 3 es 90.

Los aumentos de Vida y Velocidad al subir de nivel vienen dados por el horóscopo, el cual se asigna aleatoriamente al crear al personaje.

	Acua	Capri	Sagi	Scor	Libra	Virgo	Piscis	Aries	Tau	Gem	Can	Leo
Vida	3-4	2-4	1-4	2-3	1-3	1-3	2-2	2-2	1-2	1-2	0-2	0-1
Vel.	0-1	0-2	1-2	1-2	2-2	1-3	2-2	1-3	2-3	1-4	2-4	4-3

2.4 Los estados alterados

Los estados alterados, generalmente, afectan negativamente a los personajes y suelen ser provocados por el adversario. Existen dos tipos de estado alterado: los primarios y los secundarios.

Los estados primarios

Un personaje solo puede ser afectado por un estado primario a la vez. Tienen un 20% de probabilidad de curarse al inicio del turno. Los estados primarios que aparecen en esta versión son:

- Envenenamiento: el afectado pierde un 10% de su Vida máxima cada turno. Los personajes de elemento Veneno son inmunes.
- Parálisis: resta 1 al dado del afectado. Los personajes de elemento Natura son inmunes.
- Quemaduras: el afectado tiene un 25% menos de Defensa.
- Silencio: el afectado no puede realizar técnicas.

Los estados secundarios

Un personaje puede ser afectado por múltiples estados secundarios simultáneamente. Permanecen un número determinado de turnos, especificado en la técnica que los cause. Los estados secundarios que aparecen en esta versión son:

- Picor: 30% de fracaso al realizar una acción.
- Desarme: impide atacar físicamente.
- Contaminación: las curaciones recibidas causan daño en lugar de sanar.
- Cambio de características: aumentos o disminuciones temporales de características (fuerza, defensa, evasión, etc.)
- Escondido: para atacar a un personaje escondido será necesario estar en la misma casilla que este, buscarlo y encontrarlo (30%). “Buscar” consume la acción principal, aunque si el enemigo es encontrado, se permite encadenar la acción con un ataque sobre este.

2.5 Las clases

A continuación se describen las seis clases que existen en esta versión, indicando con detalle las características de las mismas.

Samurai

Los Samurais son espadachines especializados en el combate cuerpo a cuerpo. Su principal baza es su fuerza física.

- Vida: 44
- Fuerza: 9
- Velocidad: 9
- Elemento: -
- Arma: Katana
- Habilidad: Ying Yang (si sufre un estado primario, fuerza +1)
- Técnicas:

Nv	Técnica	Elemento	Poder	Rg	Prec	U	Efecto	Descr.
1	¡En guardia!	-	-	-	95%	5	Devuelve la mitad del daño físico recibido. Uso propio.	Adopta una posición defensiva que le permite contraatacar.
2	Mizuchi	-	7	+1	80%	5	20% crítico	Onda cortante que se desplaza por el aire a gran velocidad.
3	Yabuki Tsurugi	-	7	-	80%	6	Causa Desarmado [2-3T] (No se puede atacar físicamente)	Hace chocar su katana con el arma del rival, inutilizándola temporalmente.
3	Hara-Kiri	-	-	-	100%	1	El Samurái pierde toda su vida. Su aliado recupera el doble de la vida que le quedara.	Honorable sacrificio necesario para lograr mayores fines.

Pistolero

Los Pistoleros son precisos tiradores especializados en armas de fuego. Destacan en el uso de ataques con rango.

- Vida: 42
- Fuerza: 7
- Velocidad: 8
- Elemento: -
- Arma: Revólver [rango =1]
- Habilidad: Ojo de Halcón (precisión de ataques físicos +10%)
- Técnicas:

Nv	Técnica	Elemento	Poder	Rg	Prec	U	Efecto	Descr.
1	Bala Etérea	-	5	+2	75%	9	10% de causar Silencio.	Proyectil de energía que afecta al sistema nervioso.
2	Fogueo	-	9	-	80%	5	40% de causar Quemaduras	Ensordecedor disparo a quemarropa cargado de pólvora.
3	Recarga	-	-	-	90%	4	Recupera 3 usos de una técnica. Uso propio o ajeno. No puede usarse sobre Recarga.	Se abastece de munición adicional.
3	Zoom	-	-	-	-	1	El rango de su arma aumenta en 1 permanentemente.	Acopla una mirilla de largo alcance a su pistola.

Espía

Los Espías son maestros en el arte de pasar inadvertidos. Alcanzarlos puede ser realmente difícil.

- Vida: 38
- Fuerza: 6
- Velocidad: 13
- Elemento: -
- Arma: Daga [crítico +5%]
- Habilidad: Sabotaje (anula la habilidad de los que estén en su casilla)
- Técnicas:

Nv	Técnica	Elemento	Poder	Rg	Prec	U	Efecto	Descr.
1	Camuflaje	-	-	-	90%	6	+50% Evasión [1T] Uso propio.	Utiliza sus trucos de camuflaje para pasar inadvertido.
2	Imitación	-	-	+1	85%	6	Copia un ataque (arma o técnica) del objetivo. El ataque copiado solo puede usarse una vez.	Analiza el estilo de combate del rival, mimetizándolo a su costa.
3	Mordaza	-	-	-	90%	5	Causa Silencio.	Silencia al rival usando una mordaza.
3	Daga Oculta	-	6	-	95%	5	Se esconde después de atacar.	Se aproxima al rival sin que nadie le vea y le asesta una veloz puñalada trapera.

Chef

Los Chefs usan elementos culinarios para luchar. Son guerreros equilibrados, capaces de realizar funciones ofensivas, de apoyo o de *statuser*.

- Vida: 40
- Fuerza: 7
- Velocidad: 6
- Elemento: -
- Arma: Cuchillo de Cocina
- Habilidad: Gourmet (las curaciones recibidas aumentan un 10%)
- Técnicas:

Nv	Técnica	Elemento	Poder	Rg	Prec	U	Efecto	Descr.
1	Plato del Día	-	-	-	90%	8	Sana de 4 a 10 PS. Uso propio o ajeno.	Prepara una receta con ingredientes de primera calidad.
2	Afilar	-	-	-	90%	6	Fuerza x1,5 en el siguiente golpe. Uso propio.	Todo cuchillo necesita estar afilado…
3	Cuisine Fatale	Veneno	8	-	90%	6	Envenena. Hay un 15% de que siente bien (no envenena y cura en lugar de dañar)	Suculenta receta condimentada con toxinas naturales.
3	Sopa Hirviendo	-	9	-	90%	7	25% de Quemar.	Lanza sopa extremadamente caliente, ideal para los inviernos más fríos.

Gorgona

Las Gorgonas son expertas en el uso de técnicas basadas en las serpientes. Se les da bien causar estados alterados, especialmente los relacionados con el veneno. Luchan principalmente mediante desgaste.

- Vida: 42
- Fuerza: 6
- Velocidad: 9
- Elemento: Veneno
- Arma: Daga de la Cobra [Elemento Veneno] [20% envenenar]
- Habilidad: Piel Escamosa [causa 2 de daño al ser atacado físicamente]
- Técnicas:

Nv	Técnica	Elemento	Poder	Rg	Prec	U	Efecto	Descr.
1	Toxina	Veneno	-	-	75%	6	Causa Envenenado.	Inyecta un potente veneno en el cuerpo del rival.
2	Áspid	Veneno	7	-	85%	6	Drena 50% del daño. Si el rival está envenenado, +3 de Poder pero lo cura.	Extrae la vitalidad con sus colmillos de serpiente.
3	Infección Viperina	Veneno	6	+1	80%	6	Causa Contaminado [3T] (Cuando intenta curarse, daña)	Lanza un chorro de extraño veneno, mermando el sistema inmunológico.
3	Boa Constrictor	-	10	-	85%	7	30% de causar Parálisis.	Abrazo asfixiante que comprime el cuerpo de su presa.

Entomólogo

Los Entomólogos son expertos en el uso de técnicas basadas en los invertebrados. Son una clase con una gran movilidad.

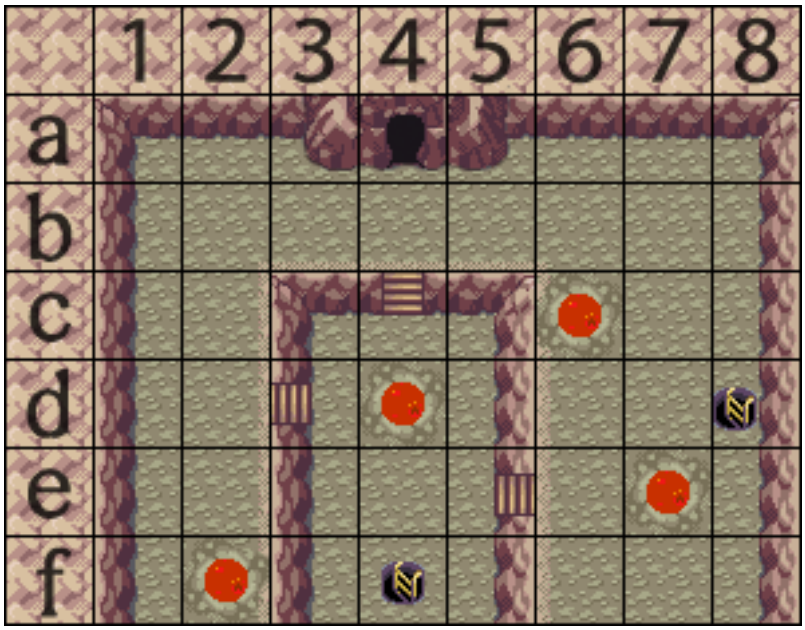
- Vida: 41
- Fuerza: 6
- Velocidad: 7
- Elemento: Natura
- Arma: Hilos de Seda [Elemento Natura] [Rango Campo]
- Habilidad: Migración [el terreno no influye en su movimiento, salvo agua]
- Técnicas:

Nv	Técnica	Elemento	Poder	Rg	Prec	U	Efecto	Descr.
1	Crisálida	Natura	-	-	90%	7	Resistencia +40% [1T] Uso propio o ajeno.	Se envuelve a sí mismo o a su compañero en un resistente hilo de seda.
2	Pulgas	Natura	-	-	80%	5	Causa Picor [3T] (50% de fracaso al realizar una acción primaria)	Libera un frasco de incordiosas pulgas sobre el adversario.
3	Telaraña	Natura	9	-	90%	6	Causa Parálisis	Teje una tela de araña que atrapa a su presa.
3	Saltamontes	Natura	5	=3	90%	7	Desplaza al usuario a la casilla afectada Puede usarse sin objetivo. Si falla o se usa sin objetivo, el usuario pierde 3 PS.	Da un tremendo salto horizontal, que le desplaza varios metros de distancia.

2.6 Los mapas

En este apartado se hará una descripción de los mapas o tableros que habrá en esta versión, resaltando las casillas especiales.

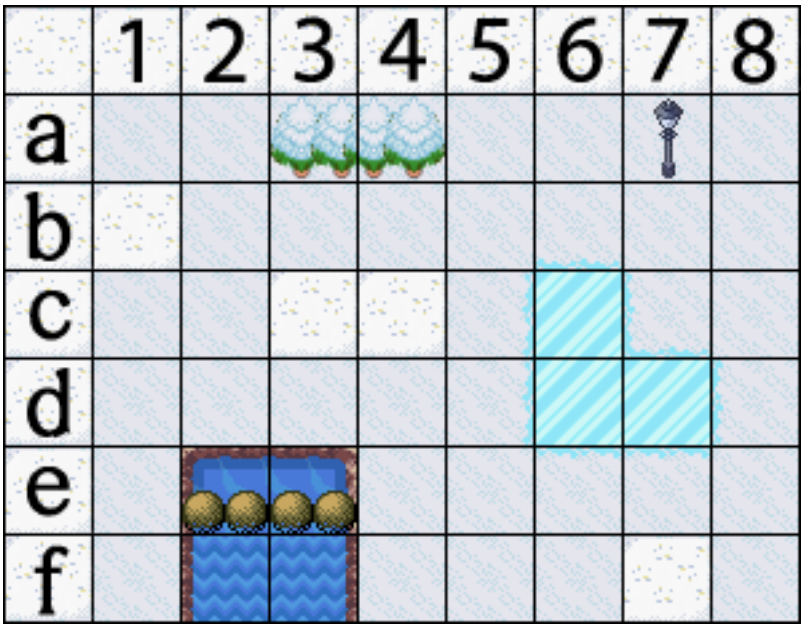
Cráter de Palop



Casillas especiales:

- Cueva (A4): permite al usuario esconderse, consumiendo su acción principal.
- Charco de lava (F2, D4, C6, E7): entrar en estas casillas y empezar el turno en ellas hace que el usuario pierda 2 PS.
- Túnel (F4, D8): permite al usuario transportarse de una a la otra, consumiendo la acción principal y 3 puntos de movimiento.
- Muro (C3, E3, F3, C5, D5, F5): impide acceder a las casillas de “arriba”.

Llanura Invernal



Casillas especiales:

- Bosque (A3, A4): permite al usuario esconderse, consumiendo su acción principal.
- Nieve profunda (B1, C3, C4, F7): salir de estas casillas consume un punto de movimiento extra.
- Hielo (C6, D6, D7): entrar en esta casilla te desplaza automáticamente a la siguiente casilla. Por ejemplo: si estás en C5 y entras a C6, pasas automáticamente a C7.
- Farola (A7): estar en esta casilla otorga +10% de evasión ante rango.
- Cascada (E2, E3, F1, F2): casillas acuáticas. Es imposible acceder a ellas.

3. Jerarquía de la escena

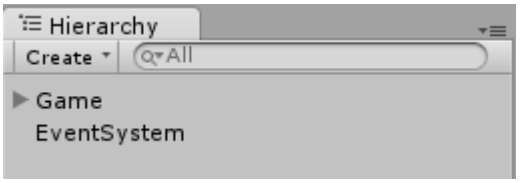
En *Unity3D*, el paradigma de trabajo está orientado alrededor de los *GameObject*, tanto en lo que respecta a la programación como a la edición gráfica. Éstos suelen ser elementos visibles del juego como personajes, escenografía, elementos de UI, etcétera. Además, también pueden ser objetos vacíos que contengan componentes –como *scripts*, elementos de audio...– o que actúen como padres de otros *GameObject*.

Los *GameObject* se sitúan en una o varias escenas (*Scene*), que típicamente hacen las veces de niveles y menús. Dentro de una escena, los elementos se organizan de forma jerárquica, mejorando así la organización del proyecto. Más importante aún, esta jerarquía permite que los objetos hijo tomen como sistema de referencia al objeto padre, de modo que al mover o redimensionar al padre, también lo harán sus hijos.

Habitualmente, los proyectos en *Unity3D* suelen contar con múltiples escenas, de modo que durante la ejecución se realiza un gran número de transiciones entre estas. Esto implica la creación y destrucción reiterada de objetos, lo que puede producir *memory leaks* si el juego está ejecutándose durante períodos de tiempo prolongados. Para evitar esto, el proyecto que nos ocupa posee una única escena muy jerarquizada, de modo que para simular un cambio de escena se activarán los *GameObject* pertinentes y se desactivarán el resto.

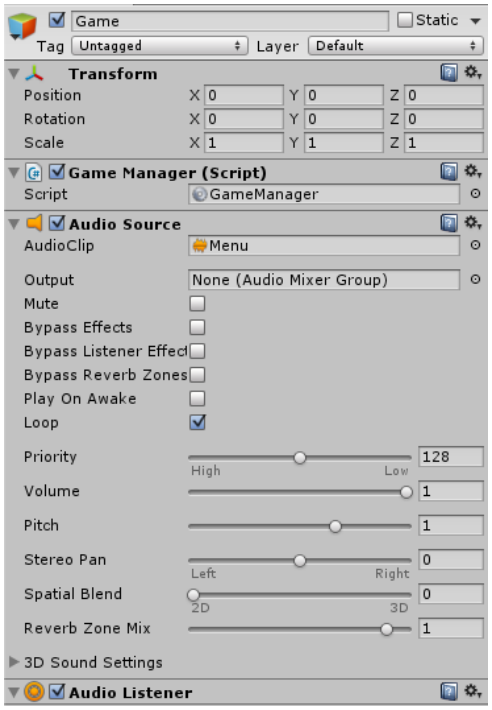
3.1 Primer nivel de jerarquía: *Game* y *EventSystem*

El primer nivel de la jerarquía –es decir, aquellos *GameObject* que no son hijos de ninguno– está compuesto por dos *GameObject*: *EventSystem* y *Game*.



EventSystem es un *GameObject* que *Unity3D* genera automáticamente cuando se crean elementos que funcionen mediante eventos –cómo por ejemplo un botón de interfaz– ya que sin éste dichos eventos no funcionarán.

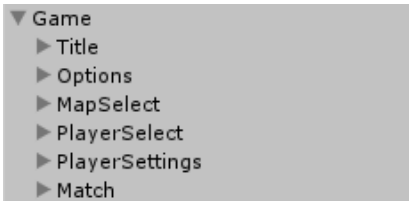
Game, por otro lado, es un *GameObject* vacío que representa al juego en sí mismo. Actúa como padre de los *GameStates* –*GameObjects* que representan a lo que tradicionalmente serían escenas–, además de contener los componentes correspondientes al audio (*Audio Source* y *Audio Listener*) y un *script* llamado *GameManager* que gestiona los cambios de *GameState* y la reproducción de pistas de sonido.



Como apunte, las escenas en *Unity3D* suelen poseer *GameObject* de tipo *Camera*, a través del cual se percibe el mundo recreado por el videojuego. Este no es el caso de *Versus Guild*, puesto que como todos los elementos son de *UI*, la cámara es innecesaria.

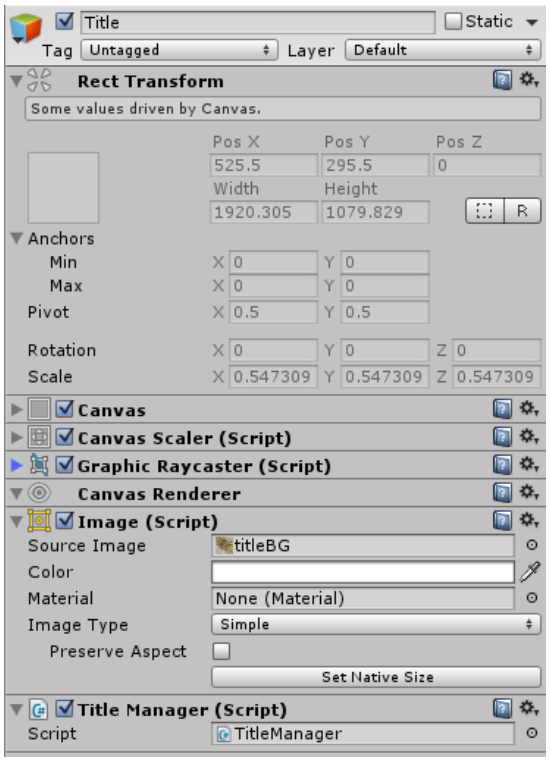
3.2 Los hijos de *Game*: los *GameState*

Se ha denominado *GameStates* a los *GameObject* que hacen la función que habitualmente haría una escena. Todos ellos son hijos de *Game*.

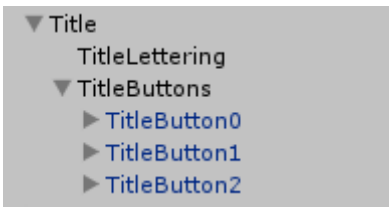


Estos objetos toman como base un *Canvas*. Este tipo de *GameObject* son contenedores del resto de elementos del paquete *UI*, de modo que todos estos objetos tienen que ser hijos –directa o indirectamente– de un *Canvas*. La resolución que se ha empleado en todos ellos es 1920x1080.

Además de los componentes que llevan los *Canvas* por defecto, a los *GameStates* se les ha agregado dos componentes más: un *script* y una imagen. Los *script* empleados son herederos de la clase *GenericGameStateManager*. Además son únicos para cada *GameState* (por ejemplo, el *GameState Title* tiene el *script TitleManager*). La imagen, por su parte, es un elemento del tipo *Image* que determina la imagen de fondo del estado.



Los *GameState* tienen como hijos a los elementos que formarán parte de la escena. Estos, a su vez, pueden ser *GameObjects* vacíos que actúen como padres de un conjunto de elementos con características similares. Por ejemplo, en *Title* podemos ver que los botones de menú están agrupados mediante *TitleButtons*, mientras que el letrero de título –*TitleLettering*– es hijo directo de *Title*.



Actualmente existen seis *GameStates*, si bien en el futuro se añadirán dos más (*Instructions* y *Credits*). Estos son:

- *Title*: representa la pantalla de título, desde donde se puede empezar a configurar una partida e ir al menú de opciones. En el futuro se añadirá la posibilidad de añadir un menú de instrucciones y otro de créditos –si bien el botón del primero ya está situado en la escena aún no está operativo.

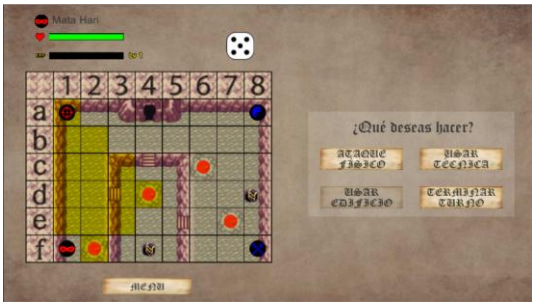
- *Options*: menú de opciones. Actualmente solo permite cambiar el idioma del juego, entre español, inglés o catalán. Tanto "Adelante" como "Atrás" devuelven al menú de título, solo que con el segundo no se guardarán los cambios realizados.

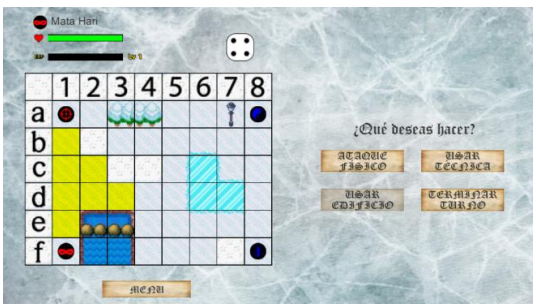
- *MapSelect*: pantalla en la que se selecciona el mapa en el que transcurrirá la partida. También muestra información sobre el mapa al pinchar sobre él, si bien esto aún no está completo (se planea que, cuando se esté mostrando un mapa, al pinchar sobre la casilla, aparezca información de la misma). Clicar en "Adelante" dirige a *PlayerSelect* -no es posible hacerlo hasta que se haya seleccionado el mapa en cuestión– y hacerlo en "Atrás" vuelve al menú de título.

- *PlayerSelect*: muestra los personajes que van a participar en la partida. Inicialmente todos están vacíos (como el último personaje en la imagen inferior), pudiendo rellenarse mediante el botón "Crear Personaje", que dirige al *GameState PlayerSettings*. Pinchar en "Modificar" un personaje ya creado también tiene efectos similares. "Adelante" lleva a *Match* –de nuevo, no es posible seleccionarlo hasta haber creado a todos los personajes– y atrás devuelve al menú de selección de mapa.

- *PlayerSettings*: permite elegir el nombre y clase de tu personaje. No permite escoger la misma clase que tu compañero de equipo. "Adelante" y "Atrás" dirigen a *PlayerSelect*, pero solo el primero guarda los cambios. No es posible pinchar en "Adelante" si la clase o el nombre no han sido escogidos. El espacio vacío entre los botones de clase y Adelante/Atrás se reserva para mostrar información sobre la clase en cuestión, algo que no está implementado por el momento.

- *Match*: *GameState* en el que se produce la partida. Debido a su gran complejidad, sus componentes serán explicados con más detalles en el siguiente apartado.





3.3 Los hijos de *Match*: los *MatchState* y *MatchBehaviour*

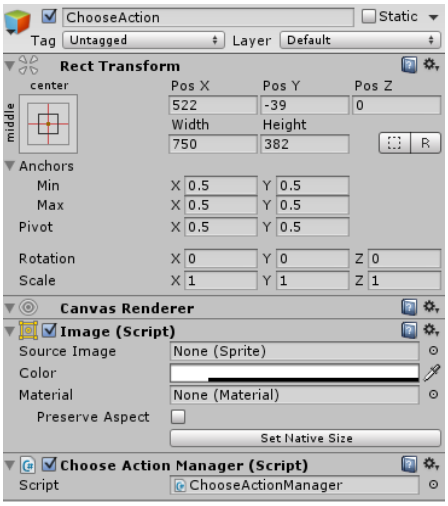
Como se ha mencionado anteriormente, el *GameState Match* cuenta con una complejidad muy superior a la del resto. Esto se debe, principalmente, a que atraviesa una serie de estados internos llamados *MatchStates*. Adicionalmente, los cambios en la situación de la partida deben verse reflejados en algunos elementos que reciben el nombre de *MatchBehaviour*.

MatchStates

Los *MatchState* representan los diferentes menús por los que atraviesa la partida para solicitar y mostrar información al jugador. Así, *Match* controla las transiciones entre ellos de forma análoga a cómo hace *Game* con los *GameState*.

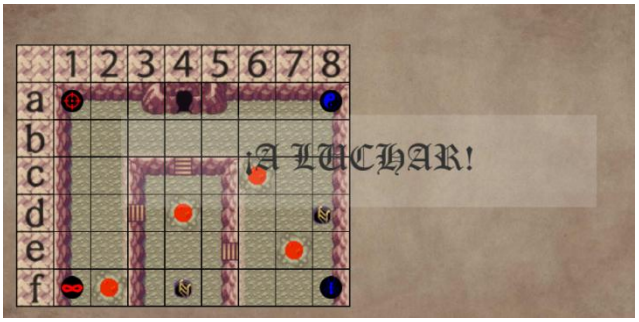
Salvo una excepción, están implementados mediante *GameObject* tipo *Image* con un color de fondo asociado, que delimita gráficamente el área del menú. Además, son los padres de los elementos correspondientes, típicamente botones y texto.

Su único componente adicional es un *script* heredero de *GenericMatchStateManager*, el cual será único para cada *MatchState* (por ejemplo, *ChooseActionManager* para el *MatchState ChooseAction*). Dicho *script* gestionará todas las acciones que pueden hacerse en el estado en cuestión. Esto incluye las acciones asociadas a clicar sobre una casilla o ficha de jugador, ya que aunque estas no forman parte del *MatchState* tienen unas funciones diferentes según el estado.

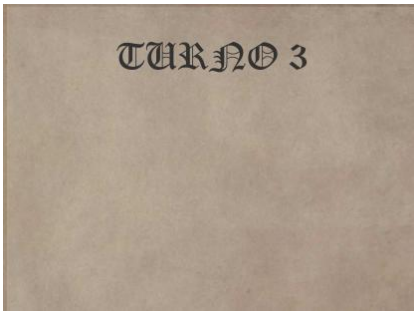


Los diferentes *MatchState* que existen son los siguientes:

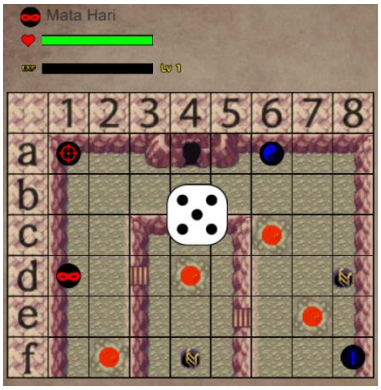
- *MatchBegin*: mensaje que aparece al comenzar la partida. Al hacer clic en cualquier sitio, cambia a *TurnBegin*.



- *TurnBegin*: al principio de cada turno muestra el contador del mismo. El espacio vacío está destinado para futuras versiones, en las que cada turno tendrá características tales como clima u horario (día y noche). Al clicar en cualquier sitio, dirige a *PlayerTurn*.



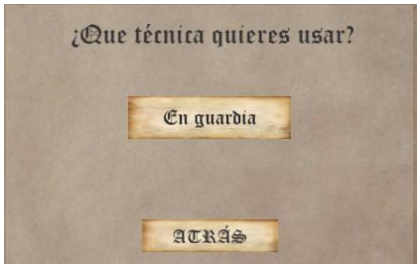
- *PlayerTurn*: se muestra al principio del turno de un jugador. En primera instancia, aparece un dado que cambia de número continuamente, el cual deberá lanzar el jugador haciendo clic en cualquier sitio. A continuación, se muestran unas pequeñas animaciones correspondientes a los estados alterados, entre otros. Una vez se han mostrado, transita automáticamente a *ChooseAction*. En caso de que el jugador muriera durante esta fase, se transitaría a *TextDisplay* para informar de su muerte. Actualmente, esto solo puede ocurrir debido a charcos de lava (ya que el estado Envenenamiento nunca remata a la víctima).



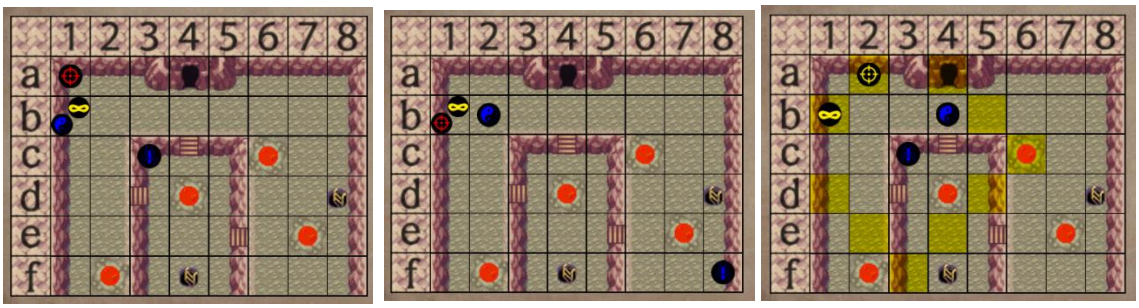
- *ChooseAction*: estado en el que el jugador puede escoger que acción realizar. Las acciones que se pueden realizar son:
    - Moverse por el mapa, haciendo clic sobre una casilla alcanzable (resaltada en amarillo). Tras ello, el jugador se moverá a dicha casilla y el dado se reducirá como sea pertinente.

    - Atacar físicamente, clicando en el botón "Ataque Físico". Esto cambia el estado a *ChooseTarget*. No es posible hacerlo si el jugador ya ha consumido la acción principal o si tiene un estado que le impida usar el arma (Desarmado).
    - Utilizar una técnica, clicando en el botón "Usar Técnica". Esto cambia el estado a *ChooseSkill*. El botón estará desactivado si el jugador ya ha consumido la acción principal o si tiene un estado que le impida usar técnicas (Silencio).
    - Utilizar el edificio de la casilla en la que se encuentra el jugador, pulsando el botón "Usar Edificio", transitando al *MatchState UseBuilding*. No es posible hacerlo si el jugador ha consumido su acción principal, si la casilla no tiene edificio o si no puede realizar la acción por otros motivos (por ejemplo, intentar usar el Túnel si el dado restante es inferior a 3).
    - Terminar la jugada, haciendo clic en "Terminar Turno". Esto cambia al estado *TurnBegin* o *PlayerTurn*, según si el jugador actual es el último o no.

- *ChooseSkill*: en este estado el jugador puede escoger una técnica de las que ha aprendido, incluyendo los ataques copiados mediante la técnica Imitación. La disposición de los botones cambia según el número de técnicas. Al hacer clic sobre un botón de técnica, dirige a *ChooseTarget* y al hacerlo sobre "Atrás" vuelve a *ChooseAction*.



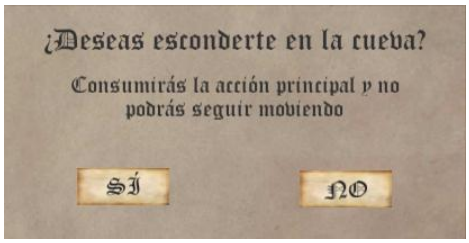
- *ChooseTarget*: estado en el que el jugador debe escoger el objetivo de su ataque, ya sea el mismo, otro mercenario o una casilla. Los objetivos resaltados también dependerán del rango del ataque. Desde este estado se puede transitar a:
  - P*reviousEffect*: al seleccionar un objetivo, si el ataque es una técnica con efectos que requieran consultar al jugador sobre ellos.
  - *AskSeek*, si no se dan las condiciones de *PreviousEffect* y se selecciona un objetivo escondido.
  - *TextDisplay*, si al seleccionar un objetivo no se dan las condiciones anteriores.

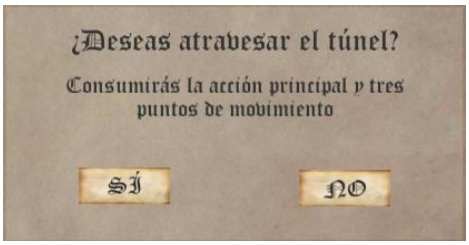


A la izquierda, el Samurai usa su arma, la *Katana*, la cual no tiene rango
Al centro, el Espía usa la técnica *Camuflaje*, que solo puede emplear sobre sí mismo.
A la derecha, el Entomólogo usa *Saltamontes*, técnica con rango =3 y que puede usarse tanto sobre casillas como sobre mercenarios.

  - *ChooseAction*: al pulsar "Atrás"

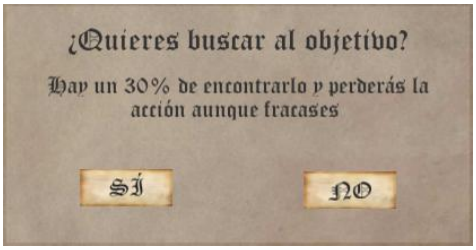
- *UseBuilding*: menú al que se accede al pulsar "Usar Edificio" en *ChooseAction* y pide la confirmación de la acción, explicando las consecuencias de la misma. Tanto al pulsar "Sí" como "No" transita a *ChooseAction*, solo que en el primer caso también se aplicarán los efectos del edificio. Excepcionalmente, si al pulsar en el botón "Sí" fracasa la acción, transitará a *TextDisplay* para informar de ello (actualmente esto solo puede ocurrir debido al estado secundario Picor).



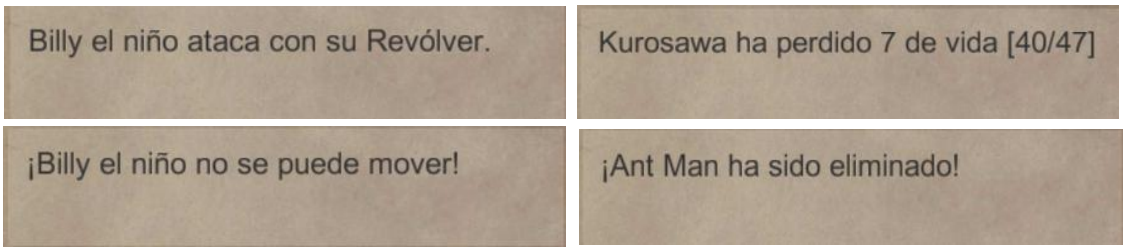


- *PreviousEffect*: estado al que se accede tras seleccionar el objetivo de una técnica con efectos que requieran consultar al jugador sobre ellos. Esto sucede con las técnicas *Imitación* (en la que el jugador debe seleccionar que ataque desea copiar) y *Recarga* (en la que el jugador debe seleccionar a qué técnica quiere recuperar usos). Al igual que en *ChooseSkill*, la disposición de los botones cambia según el número de técnicas. Además, en el caso de *Recarga*, los botones correspondientes a técnicas con los usos al máximo estarán desactivados. Al pulsar sobre uno de los botones principales, se transita a *AskSeek* si el objetivo seleccionado anteriormente está escondido o a *TextDisplay* en caso contrario. Al clicar "Atrás" vuelve a *ChooseAction*.

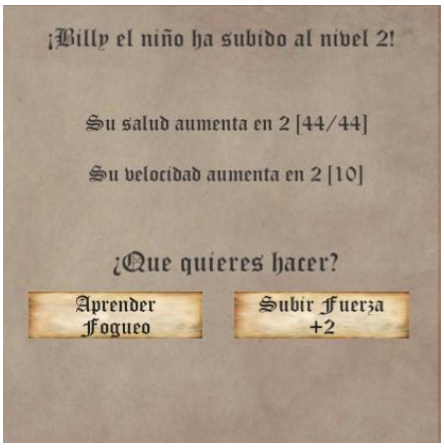
- *AskSeek*: estado que se visita cuando el objetivo seleccionado está escondido y pide la confirmación de la acción. Al pulsar "Sí" realiza el ataque y transita a *TextDisplay* y al pulsar "No" vuelve a *ChooseAction*.



- *TextDisplay*: en este estado se muestra información al jugador en forma de texto. Habitualmente esto está relacionado con el resultado de un ataque, aunque también puede ser debido a la muerte de un personaje o al fracaso de una acción. Hacer clic en cualquier sitio puede desencadenar una de las siguientes acciones:
  - Mostrar el siguiente mensaje en la lista.
  - Transitar a *LevelUp*, si el último mensaje mostrado informa sobre la subida de nivel de algún personaje.
  - Transitar a *ChooseAction*, si no quedan mensajes que mostrar.
  - Transitar a *GameOver*, si no quedan mensajes que mostrar y además se dan las condiciones para que termine la partida.



- *LevelUp*: pantalla que se aparece cuando un personaje sube de nivel. En ella se muestran los aumentos de vida y velocidad, además de permitir al jugador escoger entre subir fuerza o aprender una técnica. Al pulsar en cualquiera de estos botones, se efectúan las operaciones necesarias y transita a *ChooseAction* si no quedan mensajes que mostrar o a *TextDisplay* en caso contrario.

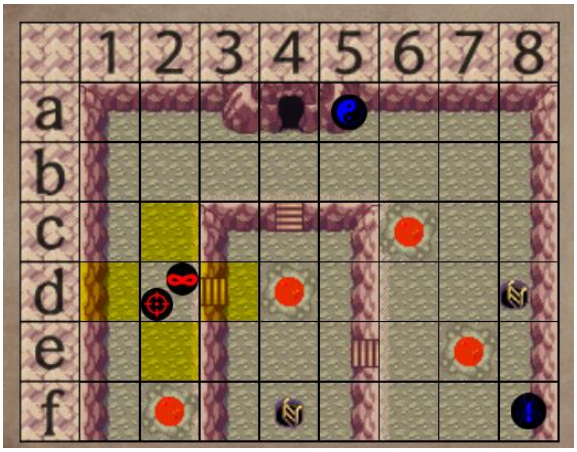


- *GameOver*: estado que se alcanza cuando termina la partida, indicando al ganador. Es capaz de reconocer empates. Al pulsar en el botón "Volver al menú", el *GameState* cambia a *Title*. También hace sonar una fanfarria de victoria en caso de que gane alguien o una música triste en caso de empate.

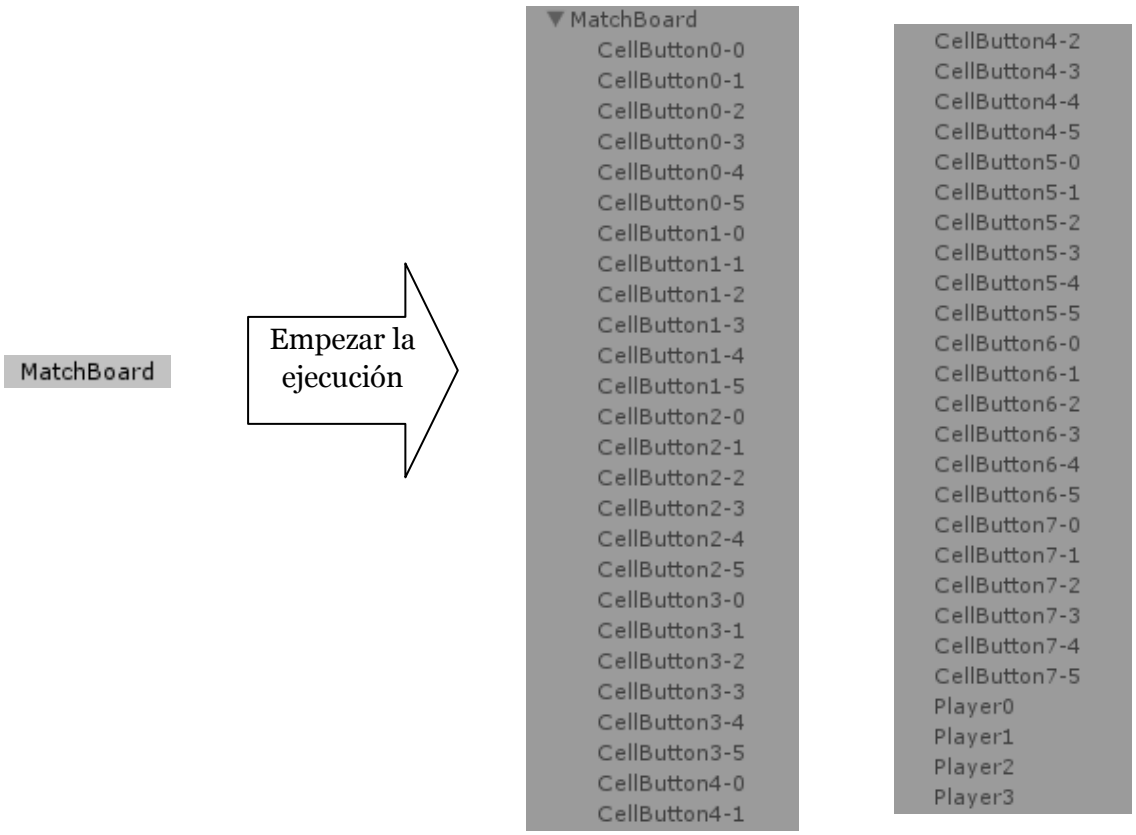
MatchBehaviour

Los *MatchBehaviour* son aquellos *GameObject* que reflejan la situación de la partida. Tienen como componente un *script* de la carpeta *MatchBehaviour* heredero de la clase de *Unity3D MonoBehaviour*. Hay dos *MatchBehaviour*: *MatchBoard* y *StatsBar*.

MatchBoard es el tablero en el que se indican las características del mapa y la posición de los mercenarios, además de poder interactuar con él tal y como se ha explicado en el apartado 3.2.1.



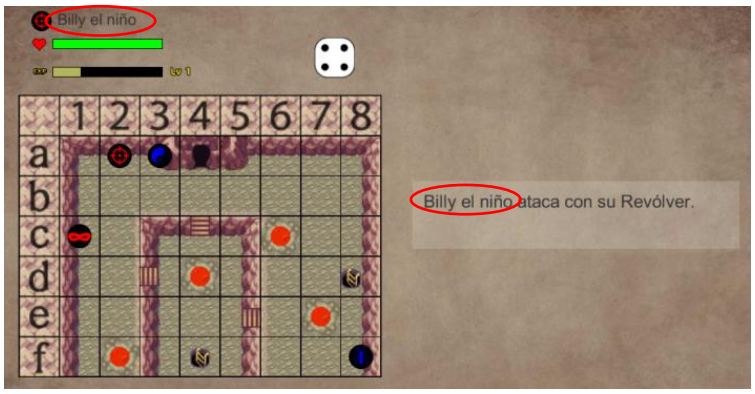
El tablero es una *Image* que cuyo *sprite* se rellenará con el borde del mapa seleccionado. En la vista de edición, *MatchBoard* no tiene ningún hijo, debido a que tanto las casillas como las fichas de jugador se generan al principio de la ejecución, lo que permite añadir mapas más grandes o con más jugadores con mayor facilidad. Los mapas más pequeños desactivarán las casillas que no necesiten. Tanto las casillas como las fichas de jugador están implementadas con *Buttons*, para que sea posible clicar sobre ellos.

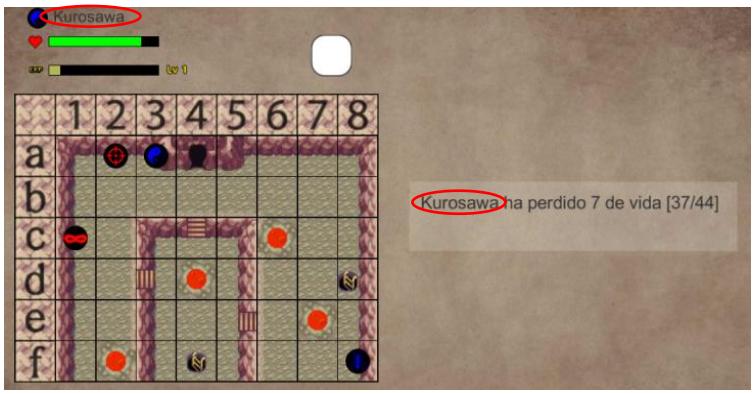


La actualización del tablero se produce mediante la función *updateBoard*(), que sitúa a cada ficha en la casilla en la que realmente está y desactiva las fichas correspondientes a jugadores muertos. La función, además, arregla las casillas con más de un jugador, reordenando las fichas que se encuentren en ella.

Por su parte, *StatsBar* muestra información sobre un mercenario de manera gráfica e intuitiva, incluyendo nombre, equipo, clase, vida, estado primario, experiencia, nivel y dado. El color de la barra de vida varía en función de cuánta quede.

El mercenario del que se muestra la información es, habitualmente, el jugador que tiene el control de la máquina. No obstante, en el *MatchState TextDisplay*, la barra mostrada corresponde al personaje al que alude el mensaje. Por ejemplo:





Adicionalmente, el *script StatsBarManager* permite la actualización de la barra "por partes", de modo en *TextDisplay* esta puede ir actualizándose a medida que van saliendo los mensajes (por ejemplo, no reducir la vida hasta que no se muestre el mensaje que indica cuánta ha perdido)

4. Jerarquía del directorio *Assets*

Los recursos empleados en un proyecto de *Unity3D* –imágenes, audio, *scripts*, etc. – se encuentran en el directorio *Assets*. Éste, a su vez, se encuentra en el directorio del proyecto, si bien dentro del programa *Assets* actúa como raíz.

Dicho directorio suele contener subcarpetas que permiten una mejor organización del proyecto. En este caso, *Assets* cuenta con cuatro directorios:

- *Resources* contiene recursos variados como imágenes, audio, animaciones…
- *Scenes* contiene las escenas del proyecto, que, tal y como se explicó en el apartado anterior, en este caso solo es una.
- *Scripts* como su nombre indica, contiene los ficheros de código del proyecto.
- *Standard Assets* contiene una serie de recursos que vienen por defecto con *Unity3D*, con posibilidad de importar más. En este caso solo existen los más básicos.

Debido a la simplicidad de *Scenes* y *Standard Assets*, tan solo se entrará en detalles con respecto a *Resources* y *Scripts*.

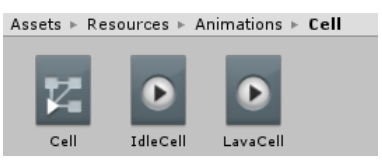
4.1 El directorio *Resources*

Como se ha mencionado anteriormente, *Resources* contiene una gran variedad de recursos. El nombre del directorio no es casual, dado que para instanciar mediante código a ciertos recursos, es necesario el método *Load* de la clase *Resources*, el cual recibe como argumento un *string* que representa la ruta del fichero tomando como raíz *Assets/Resources*.

```
cells[i,j]=(GameObject)Instantiate(Resources.Load("Prefabs/Cell"));
background.sprite = Resources.Load <Sprite>("Sprites/UI/Backgrounds/"+keyword);
```

Debido a la gran variedad de recursos almacenados en el proyecto que nos ocupa, *Resources* contiene una serie subcarpetas que ayudan a su organización. Estas son:

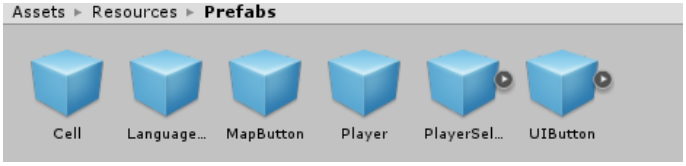
- *Animations*: contiene las animaciones y *Animators* –para animar un *GameObject*, este necesita tener un *Animator* como componente, el cual contendrá las animaciones .Está organizada en subcarpetas, según el *GameObject* al que se aplican dichas animaciones.



- *Audio*: almacena las melodías empleadas en el juego.

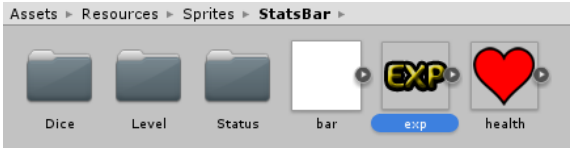
- *Fonts*: guarda las fuentes de texto que se utilizan.

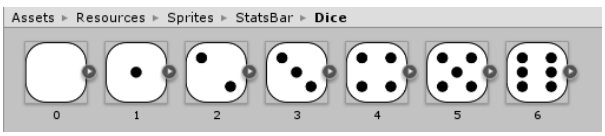
- *Prefabs*: contiene los *prefabs* empleados. Un *prefab* es un *GameObject* que se guarda como modelo, de modo que es posible instanciarlo cuantas veces sea necesario. Además, al modificar un *prefab*, también se modifican sus instancias.

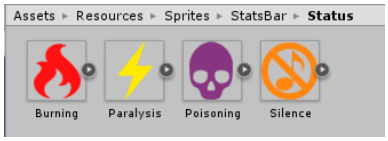


- *Sprites*: almacena las imágenes utilizadas. Es el directorio más jerarquizado de *Resources*, de modo que contiene las siguientes subcarpetas:
  - *ClassIcons*: almacena los iconos de clase, los cuales se emplean tanto para la partida (haciendo la función de ficha) como durante la creación de personajes.

  - *StatsBar*: contiene las imágenes que aparecen en la barra de estado. Tiene subcarpetas con las imágenes correspondientes a los valores del dado, del nivel y del estado primario.



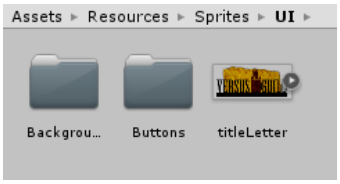


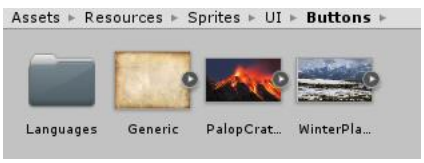


- *Tiles*: contiene las imágenes de las casillas y el margen del mapa. Está organizado en subcarpetas, cada una de las cuales hace referencia a un mapa. Como puede verse en la imagen inferior, el nombre de las casillas está numerado, de modo que para que cada *GameObject* que representa a una casilla tenga la imagen correspondiente, tan solo es necesario ejecutar un bucle doble.

- *PlayerSettings*: contiene el icono de escritorio del juego y el cursor.

- *UI*: contiene imágenes que se usan para elementos de interfaz. Está organizada en subcarpetas según la naturaleza de las imágenes:





4.2 El directorio *Scripts*

El directorio *Scripts* contiene, como su nombre indica, todos los archivos de código que se emplean en el proyecto –un total de 99. Casi todos ellos están situados en tres subdirectorios o paquetes:

- *Logic*: contiene representaciones abstractas de entidades del juego, tales como los mercenarios, los mapas, los ataques, etc.
- *UI*: almacena los ficheros que gestionan el comportamiento de la entrada y salida, ya sea de los *GameState*, de los *MatchState* o de los *MatchBehaviour*.
- *Encyclopedias*: en él se sitúan clases estáticas que almacenan datos de distinta índole: características de cada clase, cadenas de texto (en los tres idiomas) o reglas generales (por ejemplo, probabilidad de golpe crítico, dado máximo y mínimo…)

Debido a la gran complejidad de los tres paquetes, se entrará en detalles sobre ellos en los apartados 5, 6 y 7, respectivamente.

Además de dichos subdirectorios, *Scripts* contiene cuatro ficheros de código: *VGUtil*, *LogicEnums*, *UIEnums* y *TermsEnums*.

VGUtil es una clase estática que contiene una serie de métodos con propósitos diversos. Sus utilidades abarcan, entre otras cosas, generación números y eventos aleatorios (*intDice* y *eventDice*), barajar un *array* (*shuffle*) o limitar un número a un máximo y mínimo (*clamp*). También contiene otras funciones más específicas del juego, como obtener el equipo de un mercenario dado su número de jugador (*getTeam*) o la cantidad de experiencia necesaria para subir a un nivel determinado (*getNxtLv*).

```
public static bool eventDice(float prob)
{
    if (UnityEngine.Random.Range (0.0f, 1.0f) <= prob)
        return true;
    return false;
}
```

```
public static CTeam getTeam(int player){
    CTeam res = CTeam.VOID;
    switch (player) {
    case 0:
    case 1:
        res = CTeam.RED;
        break;
    case 2:
    case 3:
        res = CTeam.BLUE;
        break;
    }
    return res;
}
```

Además de esto, también contiene variables estáticas sobre diferentes constantes del juego, como el nivel máximo, las dimensiones del mapa más grande o la longitud de los *enums*.

```
public static int MAX_LEVEL = 3;
public static int MAX_MAP_WIDTH = 8;
public static int MAX_MAP_HEIGHT = 6;

public static int NUM_CLASSES = Enum.GetValues (typeof(CClass)).Length - 1;
```

El resto de ficheros almacenan tipos enumerados o *enums*, esto es tipos básicos definibles por el usuario con un número finito de posibles valores. Están organizados según el propósito de los *enums* que almacenan:

- *LogicEnums*: contiene enumerables relativos al paquete de lógica, con tipos que representan las clases de mercenario (*CClass*), las técnicas (*CSkill*), las habilidades (*CAbility*), etc. También hay otros que representan conceptos algo menos intuitivos, como *CTrigger* (que representa las posibles situaciones en las que se activa un efecto) o

```
public enum CClass
{
    VOID=0,
    SAMURAI=1,
    GUNNER=2,
    SPY=3,
    CHEF=4,
    GORGON=5,
    ENTOMOLOGIST=6,
    MAX_VALUE
}
```

```
public enum CExpRise
{
    ATTACK=0,
    SUPPORT=1,
    RECIVE_ATTACK=2,
    MISS=3,
    MAX_VALUE
}
```

CExpRise (los diferentes eventos que hacen que un mercenario suba experiencia).

Además, al final del fichero está la clase estática *LogicEnumsMethods*, en la que se definen métodos estáticos que se pueden aplicar sobre los *enum* con la misma sintaxis que cuando se aplica un método sobre un objeto. Para ello solo hay que escribir la palabra reservada *this* delante del argumento sobre el que queremos que se aplique el método.

Por ejemplo, el método *affinity*, que determina la afinidad elemental (o efectividad) del elemento de un ataque (que será tratado como *this*) sobre el elemento de un mercenario (que será tratado como parámetro):

```
public static CElementAffinity affinity(this CElement atk, CElement trg){
    switch (atk)...

    return CElementAffinity.NEUTRAL;
}
```

Un ejemplo de uso para determinar la afinidad del elemento Veneno sobre el elemento Natura:

```
CElement atk = CElement.POISON;
CElement def = CElement.NATURE;
CElementAffinity affinity = atk.affinity(def);
```

- *UIEnums*: almacena *enums* relativos a los ficheros del paquete *UI*. Incluyen códigos para los *GameState* y *MatchState*, botones de diferentes menús, etc.

```
public enum GameState{
    TITLE=0,
    OPTIONS=1,
    MAP_SELECT=2,
    PLAYER_SELECT=3,
    PLAYER_SETTINGS=4,
    MATCH=5,
    MAX_VALUE
}
```

```
public enum TitleButtons{
    NEW_GAME=0,
    HOW_TO_PLAY=1,
    OPTIONS=2,
    MAX_VALUE
}
```

```
public enum Language{
    ESP=0,
    ENG=1,
    CAT=2,
    MAX_VALUE
}
```

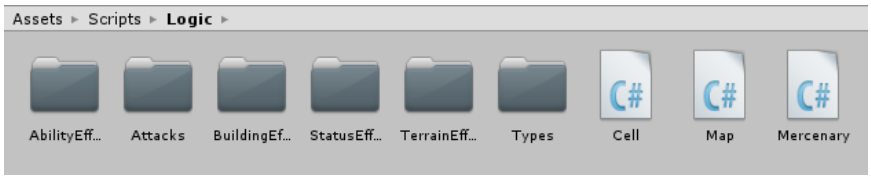
- *TermsEnums*: contiene *enums* que representan términos del juego, ya sean generales (*Turno*, *Técnica*, etc.) como específicas de algún otro elemento (*Fuerza*, *Vida*, etc. relativos a los mercenarios; *Poder*, *Rango*, etc. relativos a los ataques...). Su propósito es determinar cuál es el nombre de dichos términos para mostrárselo al jugador cuando sea pertinente. Debido a que en la versión actual no es posible consultar mucha información en el juego, tan solo existen dos *enums* en este fichero: *CTerm* (que actúa como cajón de sastre, conteniendo tanto términos generales como otros que en el futuro serán movidos a *enums* más específicos) y *CMapInfo* (que contiene los diferentes parámetros que se muestran sobre un mapa en la pantalla de selección del mismo):

```
public enum CTerm{
    VOID=0,
    TURN=1,
    NAME=2,
    CLASS=3,
    STRENGTH=4,
    HEALTH=5,
    SPEED=6,
    ELEMENT=7,
    WEAPON=8,
    SKILL=9,
    SKILLS=10,
    ABILITY=11,
    LEARN=12,
    RISE_STRENGTH=13,
    // MORE WILL BE ADDED
    MAX_VALUE
}
```

```
public enum CMapInfo
{
    NAME=0,
    COUNTRY=1,
    SIZE=2,
    DESCRIPTION=3,
    MAX_VALUE
}
```

5. El paquete *Logic*

El paquete *Logic* contiene una serie de clases y estructuras que representan de forma abstracta a los diferentes elementos del juego.



Existen tres archivos que se sitúan directamente sobre esta clase, ya que representan a los principales conceptos del juego: *Mercenary*, *Cell* y *Map*. El resto de ficheros sirven como apoyo a estos tres elementos y están distribuidos en subdirectorios:

- *Attacks*: contiene los ficheros relacionados con los ataques: la clase abstracta *Attack*, sus herederos *Weapon* y *Skill* (representación de armas y técnicas) y *Effect*, que representa los efectos secundarios de un ataque. También contiene un subdirectorio llamado *Metaeffects*, que contiene "efectos unitarios", de modo que un *Effect* está compuesto por varios *Metaeffects*. Esto será explicado con detalle más adelante.

- *AbilityEffects*: almacena clases que representan de los diferentes efectos que pueden tener las habilidades. Esto está implementado mediante herencia, para dotarlo de una mayor flexibilidad.

- *BuildingEffects*: contiene la representación de los efectos de los edificios, es decir, elementos de las casillas con los que el jugador puede elegir si interactuar o no (cueva, túnel, bosque). De modo similar a *AbilityEffects*, está implementado mediante herencia.

- *StatusEffects*: guarda clases que representan los efectos de estados alterados, ya sean primarios o secundarios. De nuevo, está implementado mediante herencia.

- *TerrainEffects*: contiene la representación de los efectos de los terrenos, es decir, elementos de las casillas por los que el jugador se ve afectado solo por estar en ellas (lava, nieve, etc.). También ha sido implementado mediante herencia.

- *Types*: almacena *structs* que sirven para los atributos de otras clases del paquete.

5.1 Las clases principales: *Mercenary*, *Cell* y *Map*

Mercenary, *Cell* y *Map* pueden considerarse las clases más importantes y representativas del paquete de lógica.

Esto puede verse con claridad incluso desde el punto de vista del usuario: desde la selección de mapa y mercenarios como por el hecho de que son los únicos elementos representados gráficamente en el tablero de *Match*.

Además, los objetos de la clase *Mercenary* van siendo modificados durante el combate para reflejar cambios de vida, estado, posición, etc. Por su parte, los objetos de tipo *Map* y *Cell* también podrán modificarse en versiones futuras, cuando se introduzcan mecánicas como el clima –que será almacenado en el mapa– o efectos que actúan sobre una casilla. Esto no es concebible para ningún otro objeto de *Logic*.

Por ello, estas tres clases aparecen como atributos en *MatchLogicManager* (ver apartado 6), mientras que el resto se guardan en la *DataEncyclopedia* (ver apartado 7) y son referenciadas por los objetos de las clases principales. De esto podría deducirse que los objetos de las clases que nos ocupan se crean y se destruyen cada vez que se inicia una partida, pero como se ha expuesto al inicio del apartado 3, el proyecto se ha implementado evitando la creación y destrucción de objetos reiterada.

De este modo, los objetos solo se crean una vez (al inicio de la ejecución) y cada vez que comienza una partida se llama a un método inicializador, que, tomando unos parámetros determinados, "crea" a un nuevo objeto. Por ejemplo, estos son el constructor e inicializador de la clase *Mercenary*:

```
// CONSTRUCTOR
public Mercenary(){
    skills=new SkillReference[VGUtil.MAX_SKILLS_PER_MERCENARY];
    secondaryStatuses=new StatusCounter[VGUtil.NUM_SECONDARY_STATUSES];
}

// INITALIZER
public void init(string n, CClass c, int ind)...
```

Del inicializador solo se muestran los parámetros, debido a la gran extensión de su código.

La clase *Mercenary*

Los objetos de la clase *Mercenary* son la abstracción lógica de los mercenarios o personajes que los jugadores controlarán. Debido a la cantidad de datos necesarios tanto para la definición de un mercenario como para almacenar su estado, esta clase requiere un gran número de atributos. Esto, sumado a una no menos extensa colección de métodos, hace de *Mercenary* la clase más extensa del proyecto.

Los atributos contienen, entre otros, información relativa al nombre, la clase, las técnicas, el estado, la habilidad, el nivel, la casilla, etc. Respecto a los atributos que almacenan información relativa a objetos de otra clase (*weapon*, *skills*, *ability*), no se guardan los objetos directamente, sino un *enum* que permite acceder a ellos en *DataEncyclopedia*:

```
protected SkillReference[] skills;

protected CAbility ability;     protected CWeapon weapon;
```

Como apunte, el atributo *skills* no es un *array* de *enums*, sino de estructuras que almacenan tanto el *enum* que referencia a la técnica como los usos que restan de ella.

Respecto a los métodos, existe una gran variedad de ellos: constructor e inicializador, métodos *get/set*, métodos simples (como *damage* o *heal*) y métodos más complejos como *achievableCells*, *useStatMod* o *attackMercenary*. Si bien algunos de estos últimos tienen pocas líneas de código, esto es porque delegan muchas operaciones en otros métodos, ya sean de *Mercenary* o de otra clase.

Método de Mercenary: UseStatMod

El método *useStatMod* calcula las modificaciones de una característica con un operador determinado (por ejemplo, aumentos multiplicativos de defensa). Además, en caso de que una modificación tenga usos (como la técnica *Afilar*, del *Chef*), éstos se verán reducidos. La función tiene en cuenta todas las posibles fuentes de modificadores de características: estados primarios, estados secundarios, terreno de la casilla en la que se encuentra el usuario y su habilidad.

```
float useStatMod(CStat stat, CStatOperator so, Mercenary other, Attack atk, CStatAcumulator acum){
    float sum=acum.baseAcumulator ();
    sum=acum.massAcumulate(sum,primaryStatusStatMod(stat,so,other,atk,acum));
    sum=acum.massAcumulate(sum,secondaryStatusStatMod(stat,so,other,atk,acum));
    sum=acum.massAcumulate(sum,terrainStatMod(stat,so,other,atk,acum));
    sum=acum.massAcumulate(sum,abilityStatMod(stat,so,other,atk,acum));
    return sum;
}
```

Los parámetros utilizados representan lo siguiente:

- *CStat stat*: característica sobre la que se van a calcular los modificadores. Puede ser: poder, protección, precisión, evasión, crítico, contraataque, movimiento o rango.
- *CStatOperator op*: operador del que se buscan modificaciones. Puede ser suma, multiplicación o división.
- *Mercenary other*: atacante (si se buscan *stats* defensivos) u objetivo (si se buscan *stats* ofensivos). Sirve para determinar si los modificadores que dependen del rango del ataque entran en juego, comparando su casilla con la de *this*. Actualmente, solo es relevante para el terreno *Farola*, que aumenta la evasión ante ataques efectuados a distancia.
- *Attack atk*: ataque sobre el que se calculan los modificadores de características. Sirve para determinar si los modificadores que dependen de si el ataque es físico o técnico se activan o no. Por ejemplo, la técnica *Crisálida* aumenta la Resistencia (o protección ante técnicas), por lo que sus modificaciones no deberán ser tenidas en cuenta si el ataque es físico. Otros ejemplos son la técnica *Afilar* (que aumenta la Fuerza o poder físico) o la habilidad *Ojo de Halcón* (que aumenta la precisión del arma).
- *CStatAcumulator acum*: forma en la que se acumulan los modificadores. Existen dos: suma y multiplicación del inverso. Estos se usan según el caso:

Caso	Acumulador	Ejemplo
Operador suma	Suma	+3 ataque – 2 defensa = +5 daño
Stats relacionados con probabilidad (prec, evas, crit)	Suma	-20% prec + 50% evas = -70% probabilidad de acierto
Aumentos de daño multiplicativos	Suma	+50% ataque – 50% defensa = daño x2
Reducciones de daño	Multiplicación del inverso	+40% defensa – 25% ataque = 0.6 * 0.75 = daño x0.45

Además, existen dos versiones sobrecargadas de éste método: sin *other* y sin *atk* ni *other*, dado que dichos parámetros no tienen sentido para los modificadores de rango y movimiento, respectivamente.

```
float useStatMod(CStat stat, CStatOperator so, Attack atk, CStatAcumulator acum){
    return useStatMod(stat, so, this, atk, acum);
}

float useStatMod(CStat stat, CStatOperator so, CStatAcumulator acum){
    return useStatMod(stat, so, this, null, acum);
}
```

Método de Mercenary: AchievableCells

El método *achievableCells* calcula qué casillas puede alcanzar un mercenario (teniendo en cuenta su posición y el dado restante) e indica el coste óptimo para alcanzar cada una de ellas. Puesto que en el mapa *Cráter de Palop* existen casillas de lava que dañan al jugador al atravesarlas, el algoritmo tiene en cuenta tanto el coste de vida como el de movimiento, priorizando el primero.

Esta función es muy similar al algoritmo de Dijkstra empleado en teoría de grafos, solo que con dos costes diferentes (vida y movimiento) y con una cuadrícula en lugar de un grafo. Así, cada iteración del bucle principal, se encolan las casillas adyacentes de la casilla que está siendo explorada en ese momento.

Tiene en cuenta elementos del mapa como muros, casillas inalcanzables (acuáticas) o que provocan desplazamientos (hielo).

La clase *Cell*

Los objetos de la clase *Cell* son la representación lógica de las casillas que forman parte del tablero y por las que se van moviendo los jugadores.

Debido a que no tiene tantos atributos como *Mercenary*, sí se entrará en detalles sobre cada uno de ellos:

- *Coord coord*: coordenadas de la casilla. *Coord* es una estructura que almacena dos enteros, *x* e *y*.

- *CTerrain terrain*: terreno de la casilla. Se almacena como un *enum* que permite acceder al objeto *TerrainEffect* en *DataEncyclopedia*.

- *BuildingReference building*: edificio de la casilla. *BuildingReference* es una estructura que guarda tanto un *enum* que permite acceder al objeto *BuildingEffect* en *DataEncyclopedia* como un *Coord* que representa una casilla relacionada con el edificio (solo se usa para el edificio *Túnel*, de forma que representa la casilla destino)

- *bool[] wall*: muros que posee la casilla. Tiene longitud 4, y cada posición representa la dirección indicada por el *enum CCardinalDirection*:

```
public enum CCardinalDirection
{
    NORTH=0,
    SOUTH=1,
    EAST=2,
    WEST=3,
    MAX_VALUE
}
```

- *Mercenary[] mercenaries*: mercenarios que están en la casilla. Es una doble referencia, ya que los objetos *Mercenary* también tienen una referencia a la casilla en la que se encuentran.

- *int numMercenaries*: número de mercenarios que se encuentran en la casilla.

- *Map map*: mapa al que pertenece la casilla. De nuevo, es una doble referencia, ya que los objetos *Map* poseen referencias a todas sus casillas.

También posee algunos métodos, a destacar:

- *addMercenary* y *delMercenary*: utilizados para añadir y borrar mercenarios de la casilla.
- *healthCost* y *diceCost*: indican el coste de vida y movimiento requeridos para cruzar la casilla, teniendo en cuenta el terreno.
- *isAchievable*: indica si una casilla es accesible directamente desde otra. Es decir, devolverá *true* salvo que no sean adyacentes, si hay un muro entre ellas o si la casilla destino es acuática.
- *distances*: determina la distancia al resto de casillas, ignorando el terreno. Se emplea para determinar qué casillas son alcanzables mediante el rango de un ataque.

La clase *Map*

Los objetos de la clase *Map* representan lógicamente al mapa o tablero en el que transcurre una partida. Entendemos como mapa no solo al conjunto de casillas, sino también a los jugadores.

Por ello, los atributos de la clase son los siguientes:

- *Cell[,] cells*: conjunto de casillas del mapa. Sus dimensiones reales equivalen a la máxima anchura por la máxima altura, si bien las casillas que superen los límites del mapa en cuestión nunca serán accedidas.
- *Mercenary[] mercenaries*: mercenarios que luchan en el mapa. El *array* tiene una longitud equivalente al número máximo de mercenarios en un mapa.
- *int width*: anchura del mapa. Los *cells* cuyo primer índice iguale o supere a este valor no pueden ser accedidos.
- *int width*: altura del mapa. Los *cells* cuyo segundo índice iguale o supere a este valor no pueden ser accedidos.
- *int nplayers*: número de jugadores del mapa. Los *mercenaries* cuyo índice iguale o supere a este valor no pueden ser accedidos.

Respecto a los métodos, no hay ninguno a destacar, ya que todos ellos responden a los patrones constructor, inicializador o *get/set*.

5.2 El directorio *Attacks*

El directorio *Attacks* contiene las clases relacionadas con los ataques, tanto aquellas que representan a los ataques en sí mismos como las que hacen lo propio con los efectos secundarios. Como se ha mencionado anteriormente, esto incluye a la clase abstracta *Attack*, sus herederas *Weapon* y *Skill*, la clase *Effect* y el directorio *Metaeffects*.

La clase abstracta *Attack*

Attack es una clase abstracta que actúa de padre de *Weapon* y *Skill*, de modo que almacena las características comunes de ambas.

Sus atributos son los siguientes:

- *CElement element*: elemento del ataque.
- *Range range*: rango del ataque. Está implementado con una estructura.
- *float crit*: probabilidad de que el ataque sea golpe crítico, en tanto por uno.
- *Effect[] effects*: efectos secundarios del ataque.
- *PowerBoost[] boosts*: se usa cuando el ataque se vuelve más fuerte bajo ciertas circunstancias. Actualmente solo se usa para la técnica *Áspid*, cuya potencia aumenta si el objetivo está envenenado.

Como es habitual, la clase posee una serie de métodos constructores, *get* y *set*. Además, posee algunos métodos virtuales que devuelven valores vacíos o por defecto, de modo que algunas de las clases herederas lo implementarán con funciones algo más complejas.

Un ejemplo de ello es el método *fails*, que determina si un ataque falla porque el objetivo no puede verse afectado por ella. Esto solo puede darse en técnicas (por ejemplo, si una técnica es curativa y el objetivo tiene la vida al máximo), por lo que supondremos que los ataques, en general, no pueden fallar salvo que se especifique lo contrario. Así, tenemos el método genérico en *Attack* y una implementación alternativa en *Skill*:

```
// ATTACK
public virtual bool fails(Mercenary trg){
    return false;
}

// SKILL
public override bool fails (Mercenary trg){
    if (isDirectDamaging ())
        return false;

    if (effects [0] == null ||
        (!(effects [0].onlyCausesStatus () && trg.isInmuneToStatus (effects [0].getPrimaryStatus ())) &&
        !(effects [0].onlyHeals () && trg.getHealth () == trg.getMaxHealth () && !trg.damagedOnHeal ()))
        )
        return false;

    return true;
}
```

Además, la clase *Attack* posee una serie de métodos para la ejecución de los ataques. Éstos son:

- *attackMercenary*: método que se lanza al efectuar un ataque sobre un mercenario. Efectúa todas las operaciones que hacen referencia al usuario del ataque, como su experiencia, daño por retroceso, transporte a otra casilla, etc. Además, llama al método *attack* para el objetivo principal y los secundarios.
- *attackCell*: método que se lanza al efectuar un ataque sobre una casilla. Efectúa todas las operaciones que hacen referencia al usuario del ataque, como su experiencia, daño por retroceso, transporte a otra casilla, etc.
- *attack*: conjunto de operaciones que deben ejecutarse sobre cada mercenario objetivo. Incluye causar daños, efectos secundarios, etc.

Estos métodos destacan por su gran complejidad, debido al gran número de operaciones que deben ejecutarse en un ataque.

Cabe destacar que la pérdida de usos (en caso de que el ataque sea una técnica) se efectúa en los métodos *attackMercenary* y *attackCell* de la clase *Mercenary*, ya que los objetos de tipo *Skill* representan modelos de técnica, y no instancias de la misma.

La clase *Weapon*

La clase *Weapon* es una de las dos herederas de *Attack*. Los objetos de esta clase son la representación de las armas que poseen los mercenarios –si bien estos no guardan un objeto *Weapon*, si guardan un *enum* que les permite acceder a éstos.

Notablemente más simple que *Skill*, no cuenta con ningún atributo a parte de los que ya posee *Attack*. Sus métodos, por su parte, tan solo incluyen diversos constructores y *getAttackBase* (método abstracto de *Attack* que permite obtener si el ataque en cuestión es *Arma* o *Técnica*).

Dada la gran semejanza entre *Attack* y *Weapon*, podría haberse decidido prescindir de la primera, de modo que *Skill* heredaría de *Weapon*. No obstante, esta representación sería poco fiel a la realidad (ya que las habilidades no son un tipo de armas). Además, impediría que en el futuro se añadan características exclusivas de las armas, lo cual está proyectado.

La clase *Skill*

Los objetos de la clase *Skill* son la representación lógica de las técnicas. Igual que con *Weapon*, los objetos *Mercenary* no almacenan los objetos de esta clase, sino un *enum* que les permite acceder a ellas en *DataEncyclopedia*.

A diferencia de *Weapon*, *Skill* posee algunos atributos además de los heredados de *Attack*. Estos son:

- *Power power*: poder de la técnica. A pesar de que en la versión actual hubiera sido suficiente con almacenar un entero, en versiones futuras aparecerán técnicas de poder variable (dependiente, por ejemplo, de la fuerza del usuario o de la salud del objetivo), por lo que se ha implementado mediante una estructura.
- *int maxUses*: usos máximos de la técnica. Nótese que los usos restantes se almacenan en *Mercenary*.
- *float accuracy*: precisión de la técnica, es decir, probabilidad de acertar expresada en tanto por uno. Nótese que este atributo no aparece en *Weapon* porque todas las armas tienen la misma precisión (90%).
- *bool[] possibleTargets*: indica si la técnica puede usarse sobre el usuario (*user*), sobre otros mercenarios (*merceanry*) o sobre las casillas (*terrain*). Esto viene dado por el *enum CTarget*.

```
// possible targets of a skill, effect, etc.
public enum CTarget
{
    MERCENARY=0,
    USER=1,
    TERRAIN=2,
    MAX_VALUE
}
```

- *Recoil recoil*: daño que sufre el mercenario al emplear la técnica bajo ciertas condiciones. El tipo *Recoil* es una estructura que contiene las condiciones en las que se dispara y el daño que causa. Las técnicas con daño de retroceso son *Saltamontes* y *Hara Kiri*.
- *CTeleport teleport*: transporte asociado a la técnica. Puede ser *VOID* (sin transporte asociado, usado para casi todas las técnicas), *USER* (transporta al usuario a la casilla destino, usado para *Saltamontes*) o *TARGET* (transporta al objetivo a la casilla del usuario, no se usa actualmente).
- *PreviousEffect previousEffect*: efecto sobre el que hay que preguntar al usuario. Se usa para las técnicas *Imitación* (pregunta qué ataque copiar) y *Recarga* (pregunta que técnica recupera usos).
- *CAttackBase attackBase*: indica si el daño de la técnica se ve afectado por modificadores de *Ataque* y *Defensa* o de *Poder* y *Resistencia*. Todas las técnicas actuales pertenecen al segundo caso.

Respecto a los métodos, todos son constructores, *get/set* u *overrides* de los métodos abstractos y virtuales de *Attack*, de modo que no se entrará en detalles.

La clase *Effect*

Los objetos de la clase *Effect* representan a los efectos secundarios que tienen tanto técnicas como armas, excluyendo aquellos que requieren ser tratados a parte (*Teleport*, *PreviousEffect*, *Recoil*).

Para el diseño de esta clase, es necesario establecer qué patrones de técnica existen o pueden existir, de modo que el diseño de la clase que nos ocupa pueda abarcarlos a todos ellos. Así, los casos a tratar son:

Caso	Ejemplo(s)
Ataque sin efectos secundarios	Arma *Katana* Arma *Revólver*
Ataque con un único efecto que se activa siempre	Técnica *Plato del Día*: cura entre 4 y 9 de vida Técnica *Telaraña*: causa Parálisis Técnica *Camuflaje*: aumenta la Evasión en 50%
Ataque con un único efecto que puede activarse con cierta probabilidad	Arma *Daga de la Cobra*: 20% de Envenenamiento Técnica *Bala Etérea*: 10% de causar Silencio
Ataque con varios efectos que no se producen a la vez	Técnica *Cuisine Fatale*: 85% de envenenar, si esto no se produce cura 8 de vida
Ataque con varios efectos que se producen a la vez	Técnica *Áspid*: el atacante recupera la mitad del daño causado y cura el Envenenamiento del objetivo
Ataque con varios efectos con probabilidades de activación diferentes, que pueden producirse o no a la vez	No existe ningún ejemplo actualmente Ejemplo inventado: 30% de causar Quemaduras al objetivo, 20% de causar Parálisis al usuario

De este modo, la implementación de la clase *Effect* necesaria para cada caso es:

- Primer caso: no es necesaria la clase *Effect*.
- Segundo caso: clase abstracta con distintas herederas según el tipo de efecto que se produce. Los efectos del primer caso se representan con un objeto nulo.
- Tercer caso: se añade un atributo *float* que representa la probabilidad en tanto por uno. En los efectos del segundo caso, dicho atributo es igual a 1.
- Cuarto caso: remodelación de la clase. Ahora *Effect* ha dejado de ser una clase abstracta, de modo que posee tres atributos:
  - *mainEffect*, que representa al efecto principal.
  - *disjEffect*, que representa al estado alternativo.
  - *chance*, probabilidad de que se produzca el efecto principal y no el efecto alternativo.

  Tanto *mainEffect* como *disjEffect* serian objetos similares al que se describe en el segundo caso, a los cuales llamaremos de ahora en adelante *efectos atómicos* o *metaefectos*. En los efectos del tercer caso, *disjEffect* será nulo.
- Quinto caso: tanto *mainEffect* como *disjEffects* pasan a ser *arrays* de metaefectos. Los objetos innecesarios del *array* serán nulos.
- Sexto caso: no afecta a la clase *Effect*. No obstante, los objetos *Attack* pasan a guardar un *array* de *Effect* en lugar de un solo objeto. Así, efectos con probabilidades de activación independientes se consideran distintos efectos.

Así pues, los atributos de *Effect* son dos *arrays* de *Metaeffects* (*mainEffects* y *disjEffects*) y un *float* (*chance*).

Además de constructores y *get/set*, esta clase posee una serie de métodos que determinan si un mercenario se ve afectado por el efecto en cuestión, ayudando a determinar si un ataque falla (método *fails* de *Attack*).

Las únicas situaciones en las que un ataque puede fallar (*fail;* nótese que no es lo mismo que *miss*) son que el ataque no dañe directamente y además:

- El único metaefecto de la técnica sea un estado primario y el objetivo sea inmune a este, ya sea debido al elemento (Veneno es inmune a Envenenamiento y Natura a Parálisis) o porque el objetivo ya sufre otro estado alterado.
- El único metaefecto de la técnica sea una sanación y el objetivo tenga la salud al máximo, además de no sufrir el estado secundario *Contaminado*, el cual hace que las curaciones quiten salud en lugar de sumarla.

Por ello, la clase *Effect* posee los métodos *onlyCausesStatus* y *onlyHeals*.

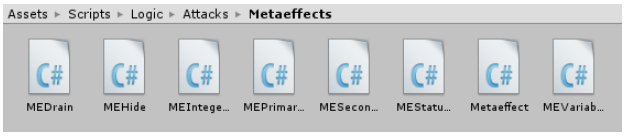
Pero probablemente, el método más importante de la clase es *affect*, que determina si se produce el efecto y, según el resultado de esto, aplica los efectos principales o los alternativos:

```
public AttackMercenaryOutput[] affect(AttackMercenaryOutput[] amo, Mercenary usr, Mercenary trg, int dam){

    if(VGUtil.eventDice(chance)){
        for(int i=0;i<VGUtil.MAX_MAIN_PER_EFFECT && mainEffect[i]!=null;i++){
            amo=mainEffect[i].affect (amo, usr, trg, dam);
        }
    }

    else{
        for(int i=0;i<VGUtil.MAX_DISJOINED_PER_EFFECT && disjEffect[i]!=null;i++){
            amo=disjEffect[i].affect (amo, usr, trg, dam);
        }
    }

    return amo;
}
```

Este método se llama desde *attack*, de la clase *Attack*.

El directorio *Metaeffects*

Como se ha explicado con anterioridad, los *metaefectos* son los componentes atómicos de los efectos propiamente dichos. Estos responden a un amplio abanico de modelos: estados primarios, estados secundarios, curaciones, etc.



Por ello, se ha optado por representar a los metaefectos mediante una clase abstracta de la cual heredarán todos los modelos posibles. Esta clase, llamada *Metaeffect*, posee un único atributo (*CTarget target*), que determina si el efecto afecta al usuario o al objetivo del ataque que causa el metaefecto.

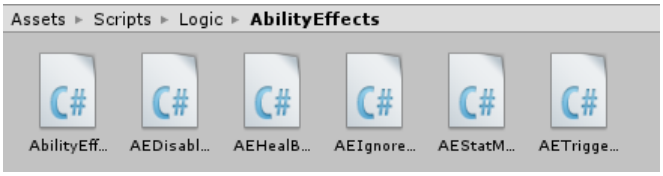
La clase también posee un método abstracto llamado *affect*, que será implementado en cada uno de los hijos de *Metaeffect* y que aplicará las operaciones pertinentes en cada caso. Este método es llamado desde el método *affect* de la clase *Effect*.

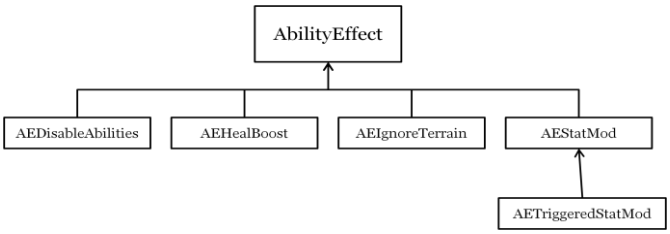
Las clases herederas de *Metaeffect* son las siguientes:

Nombre	Descripción	Atributos adicionales	Técnicas
MEDrain	El usuario recupera vida en función del daño causado.	*float value*: porcentaje de salud absorbida	Áspid
MEHide	Esconde al objetivo del metaefecto.	*float chance*: probabilidad de ser encontrado.	Daga oculta
MEIntegerHeal	Curación independiente de otros valores (como la salud actual)	*int min*: mínimos puntos de vida sanados *int max*: máximos puntos de vida sanados	Plato del día, Cuisine fatale
MEPrimaryStatus	Causa un estado primario	*CPrimaryStatus code*: estado causado	Toxina, Mordaza, Telaraña, Fogueo
MESecondaryStatus	Causa un estado secundario	*CSecondaryStatus code*: estado causado *int turns*: turnos que dura el estado *int times*: veces que afecta el estado	Yabuki Tsurugi, Pulgas, Zoom, Afilar, Crisálida
MEStatusHeal	Sana algunos estados primarios	*bool[] status*: qué estados cura	Áspid
MEVariableHeal	Curación que depende de la salud del usuario o del objetivo	*int min*: mínimo porcentaje sanado *int max*: máximo porcentaje sanado *CTarget healthBase*: si toma como referencia al usuario o al objetivo.	Hara Kiri

5.3 El directorio *AbilityEffects*

El directorio *AbilityEffects* contiene todos los ficheros relacionados con los efectos de las habilidades. Al igual que en *Metaeffects*, se han implementado mediante herencia, con una clase abstracta *AbilityEffect* y otras que heredan directa o indirectamente de ella.





AbilityEffect es una clase prácticamente vacía: tan solo tiene un método virtual (sobrecargado dos veces) llamado *ignoresTerrain*, el cual determina si una habilidad ignora un terreno concreto. En su implementación en esta clase siempre devuelve *false*, siendo sobreescrita únicamente en la clase *AEIgnoreTerrain*.

Las clases herederas de *AbilityEffect* son:

- *AEDisableAbilities*: desactiva las habilidades de los mercenarios que compartan su casilla. No tiene atributos adicionales. La única habilidad de esta clase es *Sabotaje*, del *Espía*.
- *AEHealBoost*: mejora porcentualmente las curaciones recibidas. Su único atributo es *float value*, que indica el porcentaje de mejora en tanto por uno. La única habilidad es *Gourmet*, del *Chef* (*value*=0.1).
- *AEIgnoreTerrain*: ignora el terreno de su casilla bajo ciertas condiciones. Sus atributos son:
    - *bool[] attributes*: indica que atributos del terreno ignora, según el *enum CTerrainAttribute*. Sus posiciones representan reducciones de vida (*Lava*), reducciones de de movimiento (*Nieve Profunda*), cambios de *stats* (*Farola*) y desplazamientos (*Hielo*).
    - *bool[] terrains*: indica qué terrenos ignora y cuáles no, según en *enum CTerrain*.
    - *bool[] moment*: indica en qué momento ignora a los terrenos, según el *enum CTerrainMoment*. Esto puede ser al principio del turno o durante el movimiento.

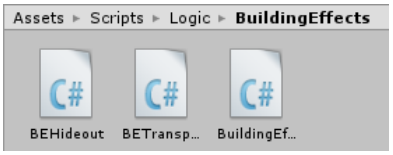
  La única habilidad de esta clase es *Migración*, que ignora pérdidas de vida y movimiento (*attributes* = {*true*, *true*, *false*, *false*}) durante el movimiento (*moment* = {*false*, *true* }).

- *AEStatMod*: modifica una característica del poseedor de la habilidad. Sus atributos son:
    - *CStat stat*: estadística que se modifica. Puede ser poder, resistencia, precisión, evasión, crítico, contraataque, rango y movimiento.
    - *CStatOperator oper*: operación mediante la que se modifica la técnica. Puede ser suma, multiplicación o división.
    - *float value*: valor de la modificación. Si el operador es "suma", debe ser un número entero. Si el operador es "multiplicación" o "división", expresa un porcentaje en tanto por uno.
    - *StatModConditions conditions*: determina si la modificación actúa en ataques físicos o técnicos (o ambos) y en ataques a distancia o cuerpo a cuerpo (o ambos).

  Existen dos habilidades pertenecientes a esta clase:
    - *Ojo de Halcón*, del *Pistolero*. Aumenta la precisión de los ataques físicos en 50%.
    - *Piel Escamosa*, de la *Gorgona*. Contraataca 2 puntos de daño ante ataques físicos.
- *AETriggeredStatMod*: clase heredera de *AEStatMod*, que simboliza modificaciones de características que tienen lugar bajo ciertas condiciones. Añade el atributo *boostConditions* (perteneciente al tipo *BoostConditions*), que representa tanto el disparador de la técnica como quién debe tener las condiciones necesarias, si el poseedor de la habilidad o el enemigo.
  La habilidad de esta clase es *Yin Yang*, del *Samurái*, que aumenta el Poder de las técnicas si el usuario tiene un estado primario.

5.4 El directorio *BuildingEffects*

El directorio *BuildingEffects* almacena los ficheros relacionados con los efectos de los edificios, es decir, elementos del mapa con los que el jugador puede interactuar deliberadamente. De nuevo, se ha optado por emplear la herencia.



La clase raíz es *BuildingEffect*, una clase abstracta que contiene los siguientes atributos:

- *bool consumeMainAction*: indica si al usar el edificio, el jugador consume su acción principal.
- *int diceCost*: reducción de movimiento al emplear el edificio. Si supera al dado restante de un jugador, éste no podrá usarlo. Si vale -1, puede usarse siempre pero consume todos los puntos de movimiento.
- *int healthCost*: reducción de vida al emplear el edificio. Actualmente, no existe ningún edificio que reduzca salud, pero se ha incluido por si fuera necesario en el futuro.

Además, la clase tiene un método virtual que ejecuta las operaciones necesarias sobre un objeto *Mercenary* que representa al jugador que use el edificio. Este método será sobreescrito en las clases herederas, de modo que estas implementaciones también ejecutarán las operaciones de la clase padre.

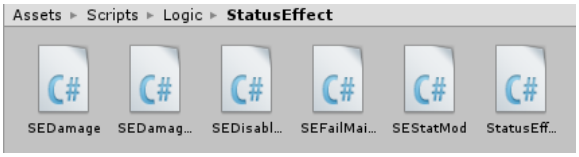
```
public virtual void useBuilding (Mercenary m, Coord c){
    m.reduceDice (diceCost);
    m.damage (healthCost);
    if (consumeAction)
        m.consumeAction ();
}

public override void useBuilding(Mercenary mercenary, Coord c){
    base.useBuilding(mercenary,c);
    mercenary.setHidden (findChance);
}
```

Las clases herederas son dos:

- *BEHideout*: edificios que permiten esconderse. Su único atributo adicional es *float findChance*, que representa la probabilidad de ser encontrado en dicho edificio. Los edificios de esta clase son *Cueva*, del mapa *Cráter de Palop* y *Bosque*, de *Llanuras Invernales*.
- *BETransport*: edificios que permiten transportarse a otra casilla. No tienen atributos adicionales, ya que la casilla destino se guarda en el atributo *building* de *Cell*. El edificio de esta clase es *Túnel*, del *Cráter de Palop*.

5.5 El directorio *StatusEffects*

El directorio *StatusEffects* guarda los ficheros que determinan el comportamiento de los estados alterados, tanto primarios como secundarios. El directorio comprende la clase abstracta *StatusEffect* y sus herederas.



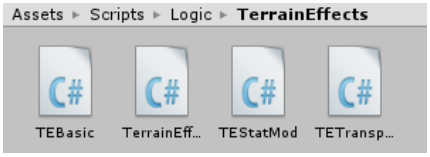
La clase raíz *StatusEffect* no posee ningún atributo y tan solo un método. Este es *bool isOffensive*, que determina si un estado afecta durante el turno del afectado o del resto de jugadores, lo cual afecta al momento en el que el contador de turnos restantes debe disminuirse. Dado que todos los estados son ofensivos excepto los cambios en las características resistencia, evasión y contraataque, este método virtual siempre devuelve *true* y solo será implementado en *SEStatMod*.

Los herederos de *StatusEffect* son:

- *SEDamage*: causa daño al afectado al principio del turno. Sus atributos son *float value* (porcentaje de vida que quita) y *bool killUser* (si el afectado puede ser eliminado por este estado). El único estado de esta clase es el *Envenenamiento*, un estado primario.
- *SEDamageHealing*: cuando el afectado reciba una sanación, perderá vida en lugar de recuperarla. No tiene atributos. Sirve para el estado secundario *Contaminación*.
- *SEDisableAttack*: impide al afectado realizar ciertos tipos de ataque. Su único atributo es *bool[] atkBase*, que determina si el afectado no puede realizar ataques físicos o técnicos. Los dos estados que implementan esta clase son *Silencio* (primario, impide usar técnicas) y *Desarme* (secundario, impide realizar ataques físicos).
- *SEFailMainAction*: el usuario puede fallar al realizar una acción principal. Su único atributo es *float chance*, que determina la probabilidad de que esto suceda. El estado secundario *Picor* implementa esta clase.
- *SEStatMod*: modificaciones de características. Sus atributos son los mismos que en *AEStatMod*: *CStat stat*, *CStatOperator oper*, *float value* y *StatModConditions conditions*.

5.6 El directorio *TerrainEffect*

El directorio *TerrainEffects* contiene los ficheros relacionados con el efecto de los terrenos, es decir, características de las casillas que afectan a los jugadores que se encuentren en ellas. Siguiendo la tónica habitual, se ha implementado mediante herencia.



La clase padre, *TerrainEffect*, es una clase abstracta con los siguientes atributos:

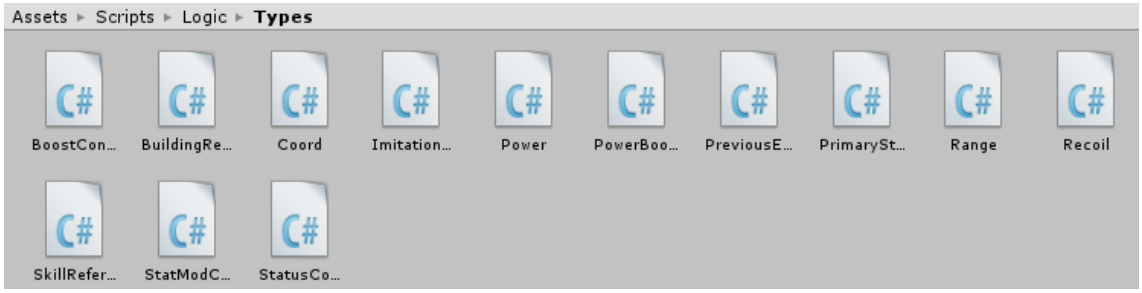
- *int diceCost*: coste de movimiento de salir de la casilla. Por defecto es 1.
- *int damageCost*: daño que causa entrar en la casilla o empezar el turno en ella. Por defecto es 0.
- *bool transitable*: determina si la casilla es transitable o no.
- *bool[] damageMoment*: determina si el daño indicado en *damageCost* se produce al empezar el turno en la casilla, al entrar en ella o en ambos casos. Sus posiciones vienen determinadas por el *enum CTerrainMoment*.

Los hijos de *TerrainEffect* son:

- *TEBasic*: terreno que no tiene ninguna característica no especificada en la clase abstracta. Las casillas neutras, la *Lava* y la *Nieve Profunda* se representan mediante objetos de esta clase.
- *TEStatMod*: estar en la casilla produce modificaciones de características. Sus atributos son, de nuevo, los mismos que en *AEStatMod* y *SEStatMod*: *CStat stat*, *CStatOperator oper*, *float value* y *StatModConditions conditions*. El terreno *Farola* es de este tipo.
- *TETransport*: al entrar en casillas con este terreno, el jugador se mueve una casilla más. No tiene atributos adicionales. Sirve para implementar el *Hielo*.

5.7 El directorio *Types*

El directorio *Types* es una especie de cajón de sastre en el que se almacenan estructuras que sirven como atributos al resto de clases del paquete *Logic*, pero que no tienen entidad lógica como para ser representadas mediante clases.



Las estructuras almacenadas son las siguientes:

- *BoostConditions*: condiciones necesarias para que se dispare un potenciador, el cual puede hacer referencia al aumento de poder de un ataque (*PowerBoost*) o a la activación de una habilidad que aumenta las características (*AETriggeredStatMod*). Tiene dos atributos:
  - *CTrigger trigger*: la condición en sí misma.
  - *CTarget target*: quien debe cumplir la condición, si el jugador o el otro mercenario.

- *BuildingReference*: almacena la referencia a un edificio, así como una coordenada asociada al mismo. Se emplea en la clase *Cell*. Contiene dos atributos:
  - *CBuilding building*: edificio al que se referencia.
  - *Coord coord*: coordenada asociada. Solo se usa para el edificio *Túnel*, de modo que representa la casilla a la que se llega al atravesarlo. Guardar este atributo aquí en lugar de en el modelo de edificio evita tener que crear un objeto diferente para cada túnel.

- *Coord*: coordenada de una casilla en un sistema de referencia cartesiano. Contiene dos atributos enteros (x e y).

- *Imitation*: ataque que ha copiado un mercenario mediante la técnica *Imitación*. Se emplea como atributo en la clase *Mercenary*. Tiene tres atributos:
  - *CSkill skill*: referencia a la técnica copiada. Si no se ha copiado ningún ataque o si este es un arma, su valor es nulo.
  - *CWeapon weapon*: referencia al arma copiada. Si no se ha copiado ningún ataque o si este es una técnica, su valor es nulo.
  - *int strength*: fuerza del mercenario del que se ha copiado el ataque. Solo se usa si este es un arma.

- *Power*: potencia de una técnica (*Skill*). A pesar de que todas las técnicas actuales tienen una potencia fija, en el futuro pueden aparecer técnicas que dependan de otros factores, por lo que se ha implementado mediante un *struct* en lugar de cómo un simple entero. Sus atributos son:
    - *float value*: valor numérico de la potencia. En el caso de que esta sea fija (como ocurre con todas las técnicas actuales), su valor debe equivaler a un entero.
    - *CPowerBase pbase*: factores tomados en cuenta para calcular la potencia. Si la potencia es fija, su valor es *CPowerBase.DIRECT*.

- *PowerBoost*: se emplea cuando un ataque puede verse potenciado por factores externos. Sus atributos son:
    - *BoostConditions conditions*: condiciones en las que se produce. Esto está detallado en la página anterior.
    - *float value*: valor de la potenciación del ataque.
    - *CStatOperator oper*: determina si la potenciación es aditiva o multiplicativa. En el caso de ser aditiva, *value* debe ser un entero. En caso de ser multiplicativa, representará el porcentaje de mejora en tanto por uno.

- *PreviousEffect*: representa los efectos de una técnica por los que hay que preguntar al usuario. Esto se usa para las técnicas *Imitación* y *Recarga*. Sus atributos son:
    - *CPreviousEffect code*: representa al efecto en sí mismo. Puede ser “Imitación” o “Recuperación de usos”.
    - *int value*: se emplea para la recuperación de usos, indicando el número de usos recuperados.

- *PrimaryStatus*: guarda información relativa al estado primario de un mercenario. Se usa como atributo en *Merceanry*. Sus atributos son:
    - *CPrimaryStatus code*: estado primario que sufre el mercenario.
    - *bool active*: indica si el estado está activo. En el momento en el que se causa un estado, este no está activo hasta que el turno del mercenario.

- *Range*: rango de los ataques. Sus atributos son:
    - *int minScope*: alcance mínimo. En rangos +X equivale a 0, y en rangos =X equivale a *maxScope*.
    - *int maxScope*: alcance máximo. En rangos +X e =X equivale a la X. Los aumentos de rango de un mercenario solo hacen referencia a este valor.
    - *CRangeArea area*: área del ataque. Puede ser “Defecto” o “Campo”.

- *Recoil*: representan el daño de retroceso de las técnicas, es decir, el que causan al usuario de las mismas. Es un atributo de *Skill*. Sus atributos son:
    - *float value*: valor que se usa para calcular el daño de retroceso.
    - *CRecoilBase rbase*: dicta como se interpreta *value*. Puede ser un daño fijo, relacionado con el daño causado, relacionado con la vida del usuario, etc.
    - *CRecoilTrigger trigger*: representa el disparador del daño de retroceso. Puede ser al acertar el ataque o al fallarlo.

- *SkillReference*: se usa en la clase *Mercenary*, como representación de las habilidades que este posee. Sus atributos son:
    - *CSkill code*: técnica a la que se referencia.
    - *int uses*: usos restantes de la técnica.

- *StatModConditions*: determina las condiciones en las que afecta un cambio de características. Sus atributos son:
    - *bool[] attackBase*: determina si la modificación afecta a los ataques físicos, técnicos o ambos. Sus posiciones se basan en el *enum CAttackBase*.

```
public enum CAttackBase
{
    WEAPON=0,
    SKILL=1,
    MAX_VALUE
}
```

    - *bool[] scope*: determina si la modificación afecta a ataques que tengan origen en la misma casilla, en otra diferente o en ambos casos. Sus posiciones se basan en el *enum CScope*.

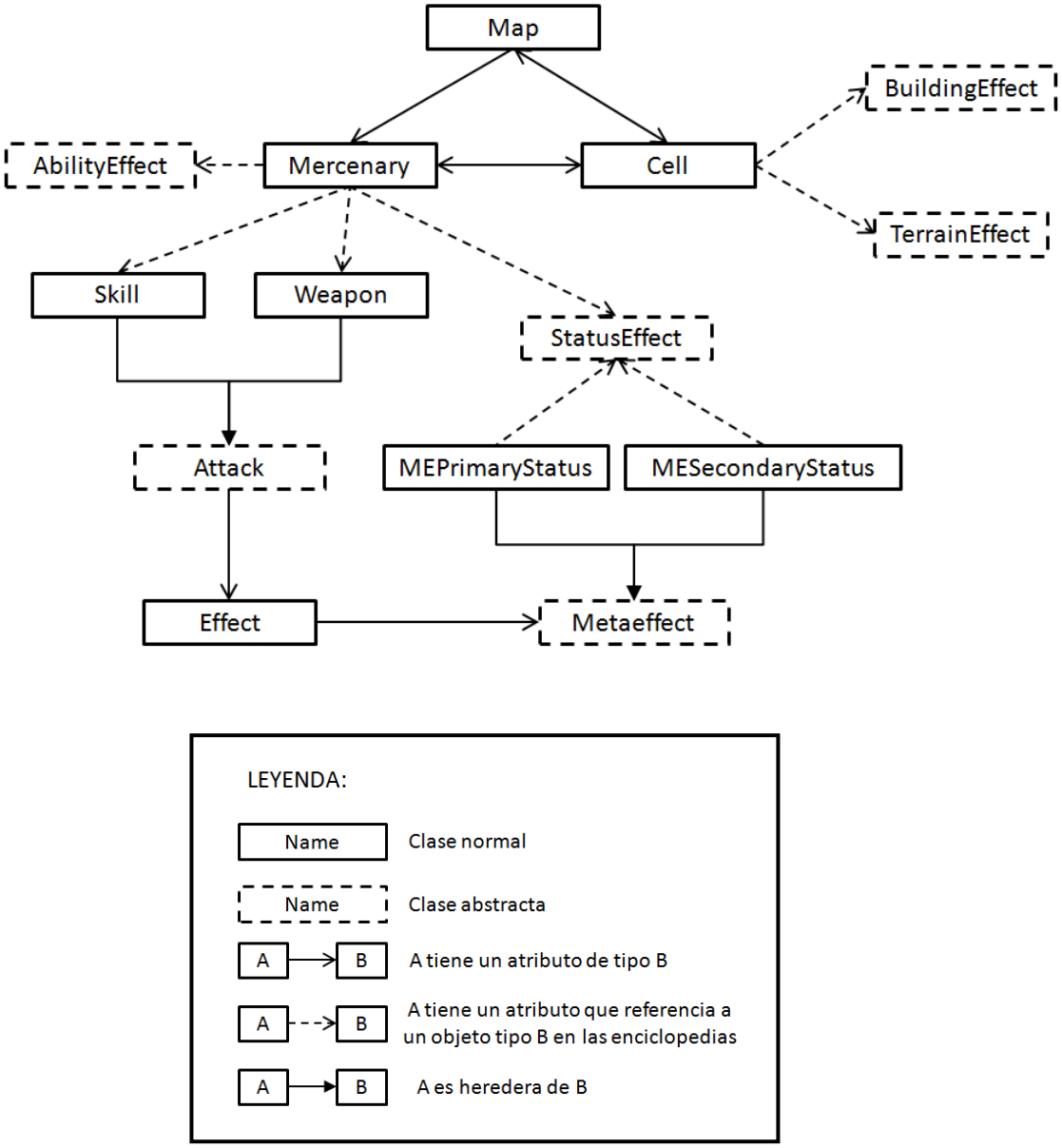
```
public enum CScope
{
    UNRANGED=0,
    RANGED=1,
    MAX_VALUE
}
```

- *StatusCounter*: contador para estados secundarios, almacenado como atributo en la clase *Mercenary*. Sus atributos son:
    - *bool active*: determina si el estado está activo. Al igual que con los estados primarios, en el momento en que se causa el estado, este es *false* y su valor pasa a ser *true* en el siguiente turno del afectado.
    - *int turns*: determina cuantos turnos quedan al estado. En cuanto desciende a cero, este se desactiva.
    - *int times*: determina cuantas veces puede producirse el estado. En cuanto desciende a cero, este se desactiva. Se usa en la técnica *Afilar*, que aumenta el ataque hasta el siguiente golpe.

  Nótese que esta estructura no guarda ninguna referencia al estado secundario del cual se lleva la cuenta. Esto se debe a que los *StatusCounter* se almacenan en un *array* ordenado, de modo que cada posición de éste referencia a un estado secundario.

5.8 Diagrama de clases

En este apartado se resume la relación de las clases del paquete *Logic* con un diagrama de clases. Éste es el siguiente:



Las clases herederas de *AbilityEffect*, *BuildingEffect*, *TerrainEffect* y *StatusEffect* y *Metaeffect* no han sido incluidas porque esto aumentaría la complejidad del diagrama sin aportar mucha información, ya que no están relacionadas con ninguna otra clase. La única excepción a esto son dos de las herederas de *Metaeffect* (*MEPrimaryStatus* y *MESecondaryStatus*), ya que estas están relacionadas con otra clase. Los tipos estructurados tampoco han sido incluidos.

6. El paquete *UI*

El paquete *UI* contiene los ficheros de código referentes a la interfaz de usuario del juego. Las clases de este paquete suelen llevar el sufijo *Manager* en su nombre, debido a que la mayoría de ellas actúan como gestores de los *GameState*, *MatchState*, *MatchBehaviour* o incluso del juego en sí mismo. Salvo excepciones, las clases del paquete heredan de *MonoBehaviour* (una clase abstracta que proporciona *Unity3D* para aquellos *scripts* que deban situarse como componente de un *GameObject* y regular su comportamiento).

Como puede verse en la imagen inferior, los ficheros se encuentran agrupados en directorios según la naturaleza del *GameObject* al que controlan.

6.1 La clase *GameManager*

La clase *GameManager* es el único fichero que se sitúa directamente sobre el directorio *UI*. Actúa como componente del *GameObject Game* (ver apartado 3.1). Sus principales funciones son determinar qué *GameState* tiene el control en cada momento, cambiar la música y almacenar variables que los *GameState* pueden acceder, haciendo posible la comunicación necesaria entre ellos.

Los atributos de esta clase son los siguientes:

- *GameObject[] gameStates*: *array* que contiene los diferentes *GameState* del juego, permitiendo activarlos y desactivarlos. Sus posiciones se basan en el *enum GameState*:

```
public enum GameState{
    TITLE=0,
    OPTIONS=1,
    MAP_SELECT=2,
    PLAYER_SELECT=3,
    PLAYER_SETTINGS=4,
    MATCH=5,
    MAX_VALUE
}
```

- *AudioSource music*: componente de *Game* que reproduce la música.
- *CClass[] clases*: contiene las clases escogidas por los jugadores. Su valor puede ser modificado en *PlayerSettings* y se lee en *PlayerSelect* y *Match*. Se representan mediante un *enum*.
- *string[] names*: contiene los nombres de los mercenarios. Al igual que *clases*, se escribe en *PlayerSettings* y se lee en *PlayerSelect* y *Match*.
- *CMap map*: mapa escogido para la partida. Se modifica en *MapSelect* y se lee en *Match*. Se representa mediante un *enum*.
- *int editMercenary*: representa el mercenario que está siendo editado en *PlayerSettings*. Se escribe en *PlayerSelect*, al pulsar el botón que dirige a *PlayerSettings*.

- *Language language*: *enum* que representa el idioma del juego. Se escribe en *Options*.

Los métodos de esta clase se dividen en tres categorías: inicialización y *get/set* de propósito general. Nótese que la clase no tiene constructor, debido a que los herederos de *MonoBehaviour* no lo necesitan.

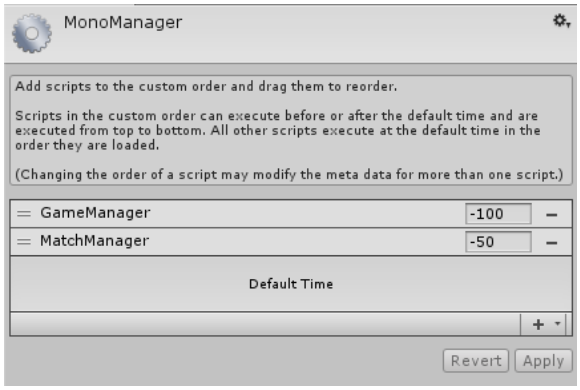
Métodos de inicialización

Los métodos de inicialización son dos: *Awake* y *Start*. El nombre de estos métodos no es casual, ya que éste permite que se ejecuten automáticamente al arrancar la aplicación.

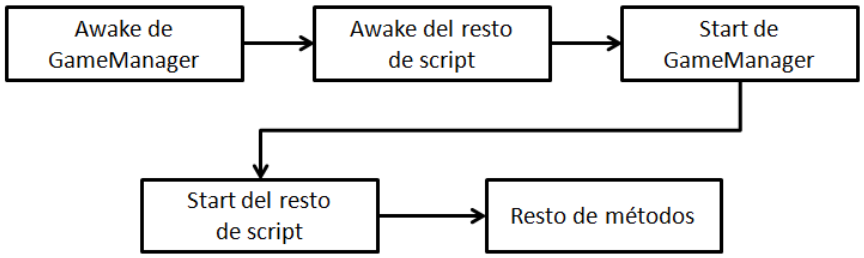
Awake tiene más prioridad que *Start* y se ejecuta tanto en el arranque como al activar el *GameObject* que lo contiene (cosa que no sucederá con este *script*, ya que el *GameObject Game* no se activa ni desactiva). En este caso, *Awake* contiene las instrucciones que inicializa sus variables y las enciclopedias (ver apartado 7).

Start, por su parte, se ejecuta cuando los *Awake* de todas las clases han sido ejecutados. Su función, en este caso, es ceder el control al *GameState Title*.

Además, la clase *GameManager* tiene más prioridad que el resto, lo que ha sido determinado en el menú *Script Execution Order*:



De este modo, el orden de ejecución es:



Métodos de get/set

Por lo general, los métodos *get/set* son tremendamente simple, ya que tan solo tienen una instrucción (leer o escribir un atributo), motivo por el cual no se ha ahondado en ellos en ninguna otra clase. La mayoría de los *get/set* de esta clase cumplen dichas condiciones, aunque algunos se salen de la norma.

En primer lugar tenemos a los métodos *del* (del inglés *delete*). Su función es borrar o reiniciar el valor de ciertos atributos. Existen dos:

- *void delMap()*: olvida que mapa ha elegido el usuario. Se emplea cuando termina una partida o al visitar un *GameState* anterior a *MapSelect* (como el menú de título *Title*).
- *void delPlayers()*: olvida el nombre y clase de los mercenarios. Se emplea cuando termina una partida o al visitar un *GameState* anterior a *PlayersSelect* (como *MapSelect*).

Otro método del segmento *get/set* que se sale de la norma es *setLanguage*. Éste, además de modificar la variable *language*, ejecuta los métodos *setLanguage* de las enciclopedias de texto, modificando sus variables para que contengan las cadenas del idioma especificado.

```
public void setLanguage(Language l){
    UITextEncyclopedia.setLanguage (l);
    LogicTextEncyclopedia.setLanguage (l);
    language = l;
}
```

Métodos de propósito general

Por último, tenemos los métodos de propósito general, es decir, los que implementan la funcionalidad de la clase propiamente dicha.

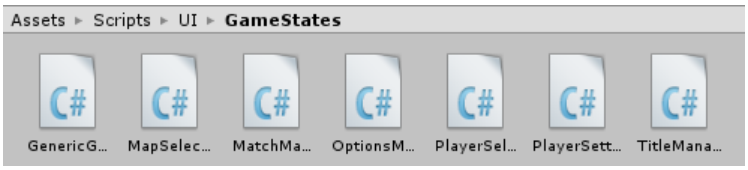
El método *changeGameState*, como su nombre indica, modifica el *GameState* que se encuentra activo, activando el *GameObject* que lo representa y desactivando el resto.

Por otro lado, tenemos el método *play*, que reproduce un fichero del directorio *Resources/Audio*. Toma como argumentos el nombre del audio y dos *bools* que representan, respectivamente, si la melodía se reproduce en bucle (*loop*) y si espera a que termine la melodía que esté reproduciendo en ese momento (*wait*). También tiene una versión sobrecargada que solo toma como argumento la melodía, de modo que al ejecutar esta versión *loop* es *true* y *wait* es *false*.

En último lugar tenemos la función *Update*, la cual, debido a su nombre, se ejecuta continuamente. En este caso, detecta si se pulsa la tecla *escape*, lo cual termina la ejecución del programa.

6.2 El directorio *GameStates*

Como su nombre indica, en este directorio se reúnen aquellos archivos que gestionan a los diferentes *GameStates*.



La clase abstracta *GenericGameStateManager*

La clase más importante de este directorio es *GenericGameStateManager*, una clase abstracta de la cual heredan el resto de clases de clases de la carpeta.

Tiene dos atributos:

- *GameManager gameManager*: referencia al gestor del juego. Como es un *singleton*, el *GameManager* referenciado siempre es el mismo. Con este atributo, los *GameState* pueden realizar transiciones entre ellos o cambiar la melodía en ciertas condiciones (como al pulsar un botón, por ejemplo).
- *bool init*: indica si la clase ya ha sido inicializada.

Esta clase tiene tres métodos abstractos que deben ser implementados en las clases herederas. Estos son:

- *void initialize()*: debe ejecutarse una única vez, al arrancar la aplicación. Generalmente, su propósito suele ser buscar los *GameObject* que deben actuar como atributos de la clase.
- *void enable()*: debe ejecutarse cada vez que el *GameState* se active.
- *void setUIText()*: pone textos de los elementos de UI (botones, texto plano, etc.). Debido a los cambios de idioma que pueden suceder durante la ejecución, también debe ejecutarse al acivar el *GameState*.

Estos tres métodos se llaman en *OnEnable*, un método reservado por *Unity3D*. Al igual que *Awake*, este método se llama tanto al iniciar la ejecución como al activar el *GameObject* pertinente, pero tiene una menor prioridad de ejecución que este (similar a la de *Start*). Así, el método tiene el siguiente aspecto:

```
void OnEnable(){
    gameManager = GameObject.Find ("Game").GetComponent <GameManager>();
    if (!init) {
        initialize();
        init=true;
    }
    else {
        enable ();
        setUIText ();
    }
}
```

Como puede verse, *OnEnable* no es abstracto ni virtual, por lo que su estructura será la misma en todas las clases herederas. La funcionalidad cambiará según la implementación de los tres métodos abstractos.

Clases herederas de *GenericSceneManager*

En este paquete, existe una clase heredera de *GenericGameStateManager* por cada *GameState* que hay. Evidentemente, estas tendrán atributos y métodos adicionales.

Generalmente, los atributos representarán a los diferentes *GameObject* que componen al *GameState*, aunque también pueden representar *flags* u otros conceptos abstractos.

De los métodos es necesario destacar a los métodos de clic, es decir, aquellos que se activan al pinchar sobre un botón de interfaz. La forma en que se asocia un método a un evento de clic es la siguiente:

```
okButton.onClick.AddListener(() => { onOKButtonClicked(); });
```

Siendo *okButton* un objeto de tipo *Button* y *onOKButtonClicked()* un método en el que se sitúan las acciones pertinentes.

La clase *MatchManager*

De todas las herederas de *GenericGameStateManager*, *MatchManager* es la más relevante. Esto se debe a que actúa como coordinadora de los *MatchState*, de un modo similar al que actúa *GameManager* con los *GameState*. Además, también almacena algunos *MatchBehaviour*.

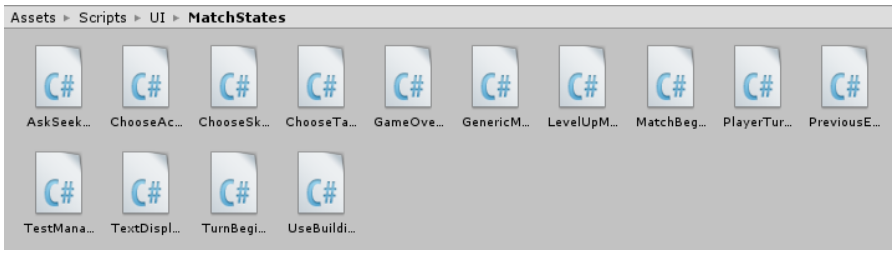
Los atributos de la clase son:

- *GameObject[] matchStates*: almacena los *MatchState*. Sus posiciones se determinan por el *enum MatchStates*.
- *bool[] showStatsBar*: determina si la barra de estado se muestra en cada *MatchState*.
- *bool[] showMenuButton*: determina si el botón de volver al menú se muestra en los diferentes *MatchState*.
- *MatchLogicManager logicManager*: gestor lógico. Almacena los objetos del paquete *Logic* de las clases *Mercenary*, *Map* y *Cell*.
- *GameObject statsBar*: barra de estado. Su función en la clase es ser activada o desactivada según determine la posición *showStatsBar* del *MatchState* actual.
- *MatchBoardManager boardManager*: gestor de la barra de estados. Cuando *Match* se activa (método *enable*),
- *GameObject menuButton*: botón de volver al menú. Se activa o desactiva en función de *showMenuButton* y el *MatchState* actual. También tiene una función asociada.

Respecto a los métodos, la clase posee *initialize*, *enable* y *setUIText* (como el resto de herederos de *GenericGameStateManager*), métodos *get/set* y *changeMatchState*, que activa y desactiva los *MatchState*, de modo similar a *changeGameState* del *GameManager*.

6.3 El directorio *MatchStates*

Este directorio almacena aquellos archivos que gestionan los *MatchStates*.



La clase abstracta *GenericMatchStateManager*

Como es habitual, los gestores de los *MatchState* se han implementado mediante herencia, de modo que todos ellos son hijos de la clase abstracta *GenericMatchStateManager*. Esta clase, a su vez, hereda de *GenericGameStateManager*, ya que comparte mucha funcionalidad con esta.

Así, a los atributos de *GenericGameStateManager* se añaden:

- *MatchManager matchManager*: gestor de la partida. Permite a los *MatchManager* ceder el control a otro.
- *MatchLogicManager logicManager*: gestor de la lógica. Realmente es innecesario, ya que *matchManager* también lo almacena, pero de este modo se acortan las sentencias que lo involucren.
- *StatsBarManager statsBarManager*: gestor de la barra de estado. Permite a los *MatchManager* actualizarla en momentos concretos.
- *MatchBoardManager boardManager*: gestor del tablero. Permite a los *MatchManager* actualizar la situación del tablero, así como captar eventos de clic sobre los botones del mismo (jugadores y casillas)
- *HUDAnimationManager hudAnimation*: animador. Permite a los *MatchBoardManager* llamar a las animaciones.

Respecto a los métodos *initialize* y *enable*, han sido implementados como métodos virtuales. La función del primero será inicializar el valor de los atributos descritos anteriormente. El segundo, que recordemos que se ejecuta al activar el *GameObject* que contenga al *script*, elimina los eventos asociados a los botones de tablero que estén asociados al *MatchState* anterior y añade los del actual. Por supuesto, los herederos de la clase podrán reescribir estos métodos, pero en todos los casos en los que se hace se ha llamado al método de la clase padre, de modo que las operaciones descritas se producen en todos ellos.

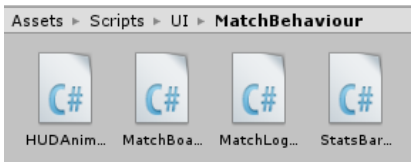
Los eventos de clic asociados a los botones del tablero que se asocian mediante la función *enable* se realizan mediante los métodos abstractos *onCellClicked* y *onPlayerClicked*, los cuales se implementan en cada clase heredera con la funcionalidad necesaria.

Por último, tenemos algunos métodos que están en esta clase por el mero hecho de que se usan en varias herederas. Esto incluye *checkIfDead* (que comprueba si el jugador actual está muerto y si lo está transita a *TextDisplay*), *backButtonClicked* (que transita a *ChooseAction* y borra la información sobre los ataques almacenada) o *attack* (que ejecuta el método *attack* de *logicManager*, haciendo que el jugador actual ejecute el ataque según los parámetros guardados).

Respecto a las clases herederas, existe una por cada *MatchState* y se emplean como componentes de estos. Por supuesto, en ellas se definen nuevos atributos y métodos.

6.4 El directorio *MatchBehaviour*

En este directorio encontramos los archivos que reflejan el estado de la partida, tanto internamente como de cara al usuario. Estos pueden estar asociados a un *GameObject* o no.



La clase *StatsBarManager*

StatsBarManager es una clase heredera de *MonoBehaviour* que se emplea como componente del *GameObject StatsBar*, controlando todas sus funciones.

Sus atributos corresponden a los *GameObject* hijos de *StatsBar*, así como algunos componentes del tipo *Image* o *Text* de los mismos. También contiene referencias al gestor de la partida (*MatchManager*) y al gestor de lógica (*MatchLogicManager*).

El método que actualiza toda la barra de estado (*updateAll*) tiene tres versiones sobrecargadas:

- Sin argumentos: muestra el estado del jugador actual. Quien es este se consulta en el gestor de lógica.
- Un argumento *Mercenary*: muestra el estado del mercenario.
- Un argumento *Mercenary* y otro *AttackMercenaryOutput*: muestra el estado del mercenario, teniendo en cuenta el *AMO* para no actualizar información que aún no ha sido procesada de cara al usuario, lo que permite mostrar los efectos de un ataque progresivamente.

El resto de métodos son actualizaciones de un solo elemento de la barra, de modo que se llaman desde *updateAll* o desde otra clase. También está el

playStatus, que reproduce animaciones (esto se llama desde *HUDAnimationManager*)

La clase *MatchBoardManager*

De naturaleza similar a *StatsBarManager*, esta clase heredera de *MonoBehaviour* se sitúa como componente de *MatchBoard* y lo controla.

Sus atributos son:

- *GameObject[,] cells*: *GameObjects* que representan a las casillas.
- *GameObject[,] players*: *GameObjects* que representan a las fichas de jugador.
- *GameObject board*: el marco del tablero. Se almacena, entre otros, para poder cambiar la imagen de éste, según el mapa.
- *Image background*: fondo de *Match*. Aunque no pertenece al tablero, cambia según el mapa, por lo que se gestiona en esta clase.
- *GameManager GameManager*: gestor del juego.
- *MatchLogicManager logicManager*: gestor de la lógica. La situación del tablero refleja los datos almacenados en este.

Respecto a los métodos, podemos clasificarlos en varios tipos: inicialización, *get/set*, coloración de fichas, arreglo de posiciones, activación de casillas y animaciones. No se entrará en detalles sobre esto.

La clase *HUDAnimationManager*

Si bien esta clase hereda de *MonoBehaviour*, su naturaleza es algo diferente de la de *StatsBarManager* y *MatchBoardManager*. Si bien se sitúa como componente de *Match*, no controla ningún *GameObject* como lo hacían las dos anteriores.

Sus funciones son realizar las animaciones correspondientes al tablero y la barra de estado. Si esto no se ha incluido en sus respectivos gestores, es debido a que algunas animaciones involucran elementos de ambos objetos.

Las animaciones deben almacenarse en métodos de tipo *IEnumerator*, los cuales se llaman con *StartCorroutine("NombreDelMetodo")*. Por ello, cada animación requiere dos métodos: el método en sí mismo y un "envoltorio" que permite llamarlo desde otras clases.

```
public void levelUp(){
    StartCoroutine ("levelUpCorrutine");
}

public void turnInfo(){
    StartCoroutine ("turnInfoCorrutine");
}

IEnumerator levelUpCorrutine()...


IEnumerator turnInfoCorrutine()...
```

La clase *MatchLogicManager*

MatchLogicManager es una clase atípica, no solo dentro del directorio *MatchBehaviour*, sino también del paquete *UI*. Un indicio de esto es que es la única clase del paquete que no hereda –directa o indirectamente– de *MonoBehaviour*, por lo que no se sitúa como componente de ningún *GameObject*.

La función de esta clase es almacenar los objetos principales del paquete *Logic* (objetos de las clases *Mercenary*, *Cell* y *Map*), de modo que actúa como puente entre la interfaz de usuario y la lógica. Además, también almacena algunos datos que deben preservarse entre los diferentes *MatchStates*.

Los atributos relacionados directamente con las clases principales y el estado de la partida son:

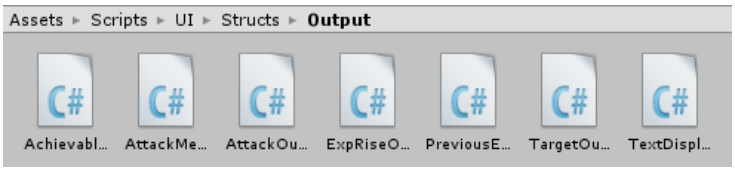
- *Map map*: mapa en el que se desarrolla la partida. Permite consultar la información sobre las casillas y sobre los mercenarios (ordenados por su índice).
- *int alvPlayers*: número de jugadores que siguen vivos.
- *Mercenary[] speedOrder*: mercenarios ordenados por la velocidad sin modificadores. El orden de sus elementos se altera cuando un mercenario sube de nivel (con su consiguiente subida de velocidad).
- *Mercenary[] turnSpeedOrder*: cada turno, copia el orden de *speedOrder* y lo altera según modificadores. Su existencia permite que cuando un mercenario sube de nivel, el orden del turno en el que lo ha hecho permanezca intacto. Además, permite gestionar alteraciones artificiales en la velocidad, cuando sean añadidas en futuras versiones.
- *int playerTurn*: indica a quien le toca. Representa la posición dentro de *turnSpeedOrder*.
- *int turn*: número de turno. Es decir, cuantas veces han movido todos los mercenarios vivos.

Los atributos que intercomunican a los *MatchState* están relacionados principalmente con los ataques: referencia al ataque empleado, al objetivo seleccionado, al ataque sobre el que se va a efectuar el *previous effect* (si lo hubiere). También los hay relacionados con la salida de los ataques, ya sea con la experiencia (para decidir cuándo transitar al *MatchState LevelUp*) o los *strings* que deben mostrarse en *TextDisplay*.

Por último, la clase posee una gran variedad de métodos que obedecen a propósitos como gestión de turnos, eliminar mercenarios cuya vida haya bajado a cero, mover a los mercenarios por el mapa, determinar si la partida ha terminado, etc. Debido a su gran variedad, entrar en detalles sobre esto podría alargar considerablemente la longitud del documento, por lo que se ha decidido no hacerlo.

6.5 El directorio *Structs*

En este directorio encontramos estructuras de datos que sirven como apoyo a los ficheros del paquete, principalmente a *MatchLogicManager*. Contiene una subcarpeta llamada *Output* que contiene los tipos que se usan como salida para ataques, casillas alcanzables, etc.



Hay dos estructuras situadas directamente sobre este directorio:

- *AttackReference*: contiene la referencia a un ataque del mercenario, empleada como atributo en el gestor de lógica. Sus atributos son *CAttackType type* (indica si el ataque es el arma, una técnica o el ataque copiado mediante imitación) y *int numSkill* (si el ataque es una técnica, guarda el índice de esta).
- *AttackTarget*: contiene la referencia al objetivo seleccionado para un ataque. También se emplea como atributo en el gestor de lógica. Sus atributos son *CTarget type* (que indica si el objetivo es un mercenario o una casilla) y dos enteros *x* e *y* (que sirven para identificar qué mercenario o casilla ha sido seleccionado; y no se usa para el mercenario).

Respecto a las estructuras del directorio *Output*, son las siguientes:

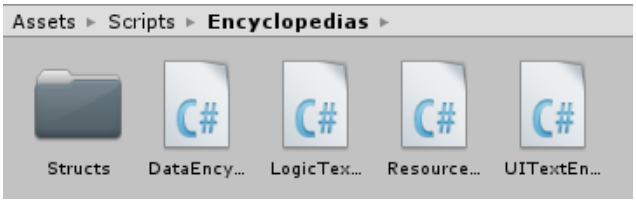
- *AchievableCellsOutput*: representa el coste de alcanzar una casilla. Sus atributos son *int movement* (coste en el dado) e *int health* (coste de vida). Es el tipo de retorno del método *achievableCells*, de la clase *Mercenary*, el cual devuelve una matriz de esta estructura de las mismas dimensiones que el mapa.
- *TargetOutput*: representa los objetivos seleccionables por un ataque. Sus atributos son *bool[] mercenaries* y *bool[,] cells*, representando que mercenarios y casillas pueden ser seleccionados o no.
- *TextDisplayOutput*: tipo que el *MatchState TextDisplay* lee y procesa, reflejándolo mediante texto, animaciones y alteraciones en la barra de estado. Sus atributos son:
    - *string text*: texto mostrado.
    - *CTextDisplayAction action*: acción a realizar, tanto en lo que respecta a realizar animaciones como a actualizar la barra de estado.
    - *CTextDisplayTarget target*: a que jugador hace referencia el TDO. Puede ser al usuario, al objetivo principal o a un objetivo secundario (en ataques con área).
    - *int index*: índice del objetivo, si este es secundario.

- *AttackOutput*: representa el resultado de un ataque, de modo que se procesa para convertirse en un *array* de *TextDisplayOutput*. Es la salida de los métodos *attackMercenary* y *attackCell*, de la clase *Attack*. Sus atributos son:
    - *AttackMercenaryOutput user*: efectos que tiene el ataque sobre el usuario.
    - *AttackMercenaryOutput mainTarget*: efectos que tiene el ataque sobre el objetivo principal (es decir, sobre quien se clica al seleccionar objetivo).
    - *AttackMercenaryOutput[] secondaryTargets*: efectos que tiene el ataque sobre los objetivos secundarios (en ataques con área, aquellos que estén bajo el área de efecto del ataque, pero no hayan sido seleccionados).
    - *bool cannotMove*: representa si el usuario ha fallado al realizar la acción, debido al estado secundario *Picor*.
    - *bool hidden*: indica si el objetivo principal estaba escondido antes de realizar el ataque.
    - *bool found*: si *hidden* es verdadero, indica si el objetivo principal ha sido encontrado.
- *AttackMercenaryOutput*: efectos que tiene un ataque sobre un mercenario. Sus atributos incluyen daño recibido, curación recibida, estados primarios y secundarios, experiencia, *previous effects*, etc.
- *ExpRiseOutput*: representa la subida de experiencia de un mercenari, normalmente debido a un ataque. Sus atributos son:
    - *int expRise*: cantidad de experiencia subida.
    - *bool lvUp*: indica si el jugador ha subido de nivel debido a la subida de experiencia.
    - *int health*: aumento de vida debido a la subida de nivel. Si *lvUp* es falso, este atributo valdrá cero.
    - *int speed*: aumento de velocidad debido a la subida de nivel. Si *lvUp* es falso, este atributo valdrá cero.
- *PreviousEffectOutput*: representa los efectos previos que han afectado a un mercenario durante un ataque. Sus atributos son:
        - *CPreviousEffect code*: representa qué *previous effect* ha afectado al mercenario. Puede ser "Imitación" o "Recuperación de usos".
        - *AttackReference attack*: referencia al ataque afectado por el *previous effect*.
        - *int value*: se emplea para recuperaciones de usos, representando la cantidad de usos recuperados.

7. El paquete *Encyclopedias*

El paquete *Encyclopedias* contiene una colección de clases estáticas que almacenan los datos del juego. Esto incluye datos de tan diversa índole características de las clases, reglas del juego o textos de interfaz.

El paquete contiene cuatro clases, además de un directorio con estructuras de apoyo:



7.1 Las enciclopedias

Las enciclopedias que se han empleado son cuatro:

- *DataEncyclopedia*: contiene datos técnicos del juego. Se podría decir que es una traducción a C# de algunas de las reglas descritas en el apartado 2 de este documento. Sus atributos pueden clasificarse en varias categorías.
    - Reglas generales: a esta categoría solo pertenece *ruleset*, una estructura que contiene datos como dado máximo y mínimo, probabilidad de crítico por defecto, etc.
    - Datos sobre objetos *Logic* principales: estructuras de datos que permiten inicializar los objetos *Mercenary*, *Cell* y *Map* con el método *init* de sus respectivas clases. También podemos incluir en esta categoría al *learnset*, es decir, las técnicas que cada clase puede aprender.
    - Objetos de clases *Logic* no principales: objetos como ataques, terrenos, habilidades, etc. Dado que no se modifican durante la partida, se almacenan en las enciclopedias, de modo que los objetos principales acceden a estos mediante *enums* que los identifican.
    - Otros: probabilidad de curación de los estados alterados, subida de experiencia en cada situación y efectos de los horóscopos.
- *ResourcesEncyclopedia*: almacena datos que se emplean para los recursos. Esto incluye dos categorías:
    - *Keywords*: palabras clave que permiten identificar a un elemento en concreto, como mapas, estados alterados, clases, etc. Por ejemplo, todos los directorios y recursos que tienen que ver con el mapa *Cráter de Palop* tienen en su nombre *PalopCrater*.
    - Dimensiones del mapa: dimensiones en píxeles de las casillas y margen de un mapa. Se emplean para crear los *GameObject* que representan a las casillas, lo cual se hace de forma dinámica.

- *LogicTextEncyclopedia*: textos relacionado con los objetos de la capa de lógica. Esto incluye nombres de clase, mapas, técnicas, etc.
- *UITextEncyclopedia*: textos mostrados en elementos de interfaz, como botones o texto plano.

Los atributos de las enciclopedias son, habitualmente, *arrays* ordenados. En ellos, las posiciones de sus objetos vienen dadas por los *enums*. Por ejemplo, en el caso de las características de las clases, el *array* viene dado por *CClass*:

```
public enum CClass
{
    VOID=0,
    SAMURAI=1,
    GUNNER=2,
    SPY=3,
    CHEF=4,
    GORGON=5,
    ENTOMOLOGIST=6,
    MAX_VALUE
}
```

De modo que al inicializarlo se tienen en cuenta los valores del *enum*, ignorando a qué posición numérica referencia cada valor.

```
classData = new ClassData[VGUtil.NUM_CLASSES];

classData [(int)CClass.SAMURAI] = new ClassData (9,44,9,CWeapon.KATANA,CAbility.YIN_YANG,CElement.NEUTRAL);
classData [(int)CClass.GUNNER] = new ClassData (7,42,8,CWeapon.REVOLVER,CAbility.SHARPSHOOTER,CElement.NEUTRAL);
classData [(int)CClass.SPY] = new ClassData (6,38,13,CWeapon.DAGGER,CAbility.SABOTAGE,CElement.NEUTRAL);
classData [(int)CClass.CHEF] = new ClassData (7,40,6,CWeapon.KITCHEN_KNIFE,CAbility.GOURMET,CElement.NEUTRAL);
classData [(int)CClass.GORGON] = new ClassData (6,42,9,CWeapon.COBRA_DAGGER,CAbility.SCALED_SKIN,CElement.POISON);
classData [(int)CClass.ENTOMOLOGIST] = new ClassData (6,41,7,CWeapon.SILK_STRINGS,CAbility.MIGRATION,CElement.NATURE);
```

Los métodos, por su parte, se componen de un inicializador (que crea los *array* y rellena sus valores) y métodos *get*. En algunos casos, el inicializador se compone de varios métodos auxiliares, facilitando la comprensión del código.

Mención aparte merecen las enciclopedias de texto *LogicTextEncyclopedia* y *UITextEncyclopedia*. En ellas, el inicializador tan solo crea los *array*, pero no rellena su valor. Esto lo hace el método *setLanguage*, que rellena con los *string* según el idioma pasado como parámetro. Este método se llama al inicio de la ejecución y al cambiar el idioma en el *GameState Options*.

7.2 El directorio *Structs*

El directorio *Structs* contiene estructuras de datos que se usan como tipos de algunos atributos de las enciclopedias.

Estas estructuras son:

- *Ruleset*: conjunto de valores numéricos que representan valores por defecto. Incluye dado máximo y mínimo, precisión por defecto, probabilidad de golpe crítico por defecto, multiplicadores por afinidad elemental y golpe crítico, etc.
- *ClassData*: rasgos que distinguen a una clase de otra, excluyendo el *learnset* (técnicas que puede aprender cada nivel). Esto es, Fuerza, Vida, Velocidad, Arma, Habilidad y Elemento.
- *CellData*: datos de una casilla. Sus atributos representan: terreno, edificio y muros.
- *MapData*: datos que distinguen un mapa, excluyendo los de sus casillas (ya que estos se almacenan en *CellData*). Incluye dimensiones, número de jugadores y posición inicial de los mismos.
- *HoroscopeData*: valores máximos y mínimos que las características suben debido al horóscopo cada nivel.
- *MapDimensions*: dimensiones gráficas del mapa, representadas en píxeles. Incluye tamaño de las casillas, márgenes y líneas, así como la posición de los márgenes.

8. Conclusión

Tras la realización del proyecto que nos ocupa, podemos concluir que lo más importante a la hora de desarrollar un videojuego es la coordinación entre sus partes. En el caso de *Versus Guild*, la delimitación de qué funciones desempeña cada paquete ha sido vital, así como la relación tanto entre ellos como con los *GameObject* de la escena.

El esfuerzo requerido, por su parte, ha sido algo superior al esperado, con un total de 9680 líneas de código, repartidas entre 100 ficheros de código distintos.

8.1 Objetivos cumplidos

Finalmente, la mayoría de los objetivos planteados inicialmente han sido cumplidos.

Lo más destacable es la funcionalidad relacionada con las reglas del juego, las cuales han sido implementadas por completo. Esto incluye algunas mecánicas complejas como:

- Técnicas con efectos sobre los que hay que preguntar al usuario, como *Recarga* –que requiere preguntar qué técnica recupera usos– o *Imitación* –que requiere preguntar qué ataque copiar. Esto complicaba el diseño del juego, ya que requería un estado de la partida (*MatchState*) y almacenar información extra.
- Técnicas con varios efectos, que se activan conjunta o disjuntamente, como *Cuisine Fatale* o *Áspid*. Esto ha añadido complejidad a las clases *Effect* y *Skill*.
- Efectos de las casillas, tanto activos (*Edificios*) como pasivos (*Terrenos*).

Otro de los objetivos iniciales en el que se puso mucho énfasis era maximizar la organización del código, permitiendo futuras ampliaciones con facilidad. Dicho objetivo no solo ha sido cumplido, sino que ha mejorado las expectativas: los objetos de los paquetes *BuildingEffects*, *TerrainEffects*, *AbilityEffects* y *StatusEffects* no iban a existir en un principio, de modo que sus funcionalidades iban a ser implementadas de forma más exhaustiva.

Otros objetivos menores cumplidos incluyen poder jugar en tres idiomas (español, inglés y catalán), la implementación de una barra de estado, pequeñas animaciones al principio de cada turno, etc.

El único objetivo no cumplido es la implementación de un sistema que permita consultar información al jugador, lo cual no se ha realizado por falta de tiempo. Esto incluye el menú de instrucciones, consultar información sobre las clases y mapas durante su selección y consultar el estado de la partida, lo cual está limitado actualmente a la barra de estado y al *text display* que aparece al realizar un ataque.

8.2 Futuras ampliaciones

Como se ha explicado anteriormente, el proyecto está estructurado de modo que añadir ampliaciones es relativamente sencillo. Así pues, algunas de ellas ya están proyectadas para futuras versiones.

La mayoría de ellas hacen referencia a la inclusión de nuevas clases y mapas, ampliando las mecánicas con habilidades, técnicas, terrenos, etc. con efectos nuevos. Algunas de estas novedades incluyen armas con aumentos de características pasivos, técnicas que alteran el terreno de una casilla, ataques con rangos de área variados, etc.

Otra ampliación proyectada es sustituir el *text display* por una pantalla en la que aparecerían unos *sprites* representando a los personajes y animaciones de los ataques. Esto también incluiría un editor de personajes, pudiendo personalizar su aspecto.

9. Bibliografía

Al no tratarse de un proyecto de investigación, sino del desarrollo autónomo de una aplicación, no ha sido necesario consultar un gran número de fuentes. No obstante, se citarán enlaces a los sitios oficiales de las herramientas utilizadas, así como la documentación y recursos empleados.

9.1 Programas empleados

- *Unity 3D*: http://unity3d.com/
- *The* Gimp: http://www.gimp.org/
- Audacity: http://audacity.es/

9.2 Documentación

- *Unity Answers*: http://answers.unity3d.com/
- *Unity Community*: http://forum.unity3d.com/
- Tutoriales C#: https://msdn.microsoft.com/es-es/library/aa288436(v=vs.71).aspx

9.3 Recursos

- Compositores Derek y Brandon Fietcher: https://www.youtube.com/channel/UCjMZjGhrFq_4llVS_x2XJ_w
- Open Game Art: http://opengameart.org/