CEDV
Fighters

Juan Montero Román
Ricardo Guzmán Velasco

Memoria completa

Índice

Diseño de la solución

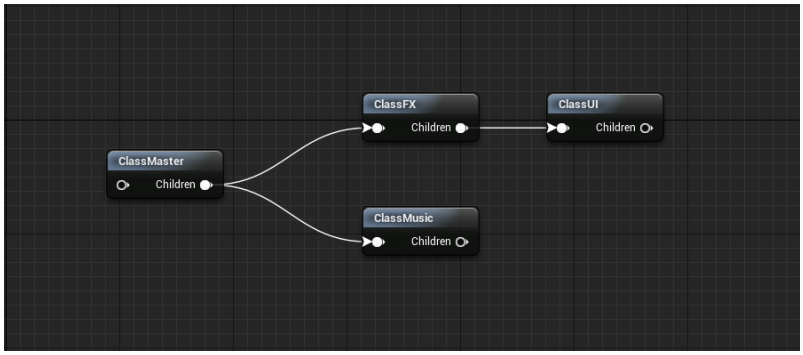
Gestión del audio

Clases de sonido

Unreal Engine proporciona una entidad de naturaleza jerárquica llamada *SoundClass*. Mediante su uso bien emparentado, es posible reducir a muy poco esfuerzo la gestión recurrente de sonido que los videojuegos requieren. Aunque Unreal ya trae una taxonomía por defecto de clases de sonido, para este proyecto se ha optado por crear una propia anexa por, entre otras, las siguiente razones:

- **La visualización de las clases de sonido por defecto que trae Unreal es defectuosa en el editor.** En ocasiones, si se quiere trabajar con estas clases más en profundidad, el editor no es capaz de enlazar a ellas (ni siquiera activando en el selector de clase el contenido propio del motor, lo cual hay que hacer en cualquier caso en que quiera usarse una de estas clases de sonido).
- **La clasificación de las clases por defecto es menos eficiente de lo que podría para este proyecto.** Si bien todas las clases de sonido que Unreal proporciona derivan de la que denomina *Master*, lo cual es de agradecer, no ocurre lo mismo entre otras clases como *UI* respecto a *FX*. Por supuesto esta es una solución plausible, pero ralentiza las operaciones de gestión de audio en todos aquellos proyectos en los que los sonidos de interfaces siempre se consideren efectos de sonido.

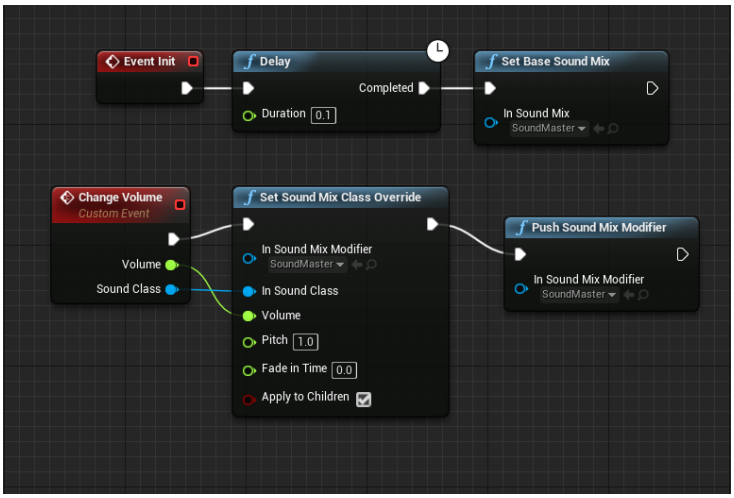
En esta entrega basta con cuatro clases, como muestra la imagen. Han sido prefijadas todas con la palabra *Class* para diferenciarlas fácilmente de las clases por defecto en el editor (no sería necesario si se tiene el contenido interno del motor desactivado en la visualización). La madre de ellas es *ClassMaster*, análoga a su homónima por defecto. Cualquier tratamiento de esta clase tendrá por tanto efecto en las demás. La siguen *ClassMusic* y *ClassFX* para usos obvios y, como se ha dicho, de la última hereda *ClassUI*. Esta ha sido marcada con la propiedad `isUISound`, de manera que Unreal autogestionará que cualquier sonido perteneciente a ella sea reproducido sin problemas incluso cuando el juego esté en pausa (el resto de sonidos sí se detendrán hasta que se restablezca la ejecución).



Mezcladores de audio

La forma adecuada de gestionar estas clases, o, mejor dicho, aquellos sonidos que contienen, es el uso de mezcladores de audio, llamados *SoundMix*. Aunque no se ha usado para tal en esta entrega, otros usos de un mezclador de audio son suavizar la transición entre distintos sonidos, aplicar efectos

conjuntos, realizar desvanecimientos de entrada y salida, etc. Por la magnitud del proyecto, ha bastado con un solo mezclador de audio encargado de manipular las clases.



La figura muestra cómo el mezclador de sonido creado en el proyecto (*SoundMaster*) es configurado como mezclador por defecto. El retardo imperceptible añadido antes evita problemas de inicialización. En la traza de código restante se hace uso de un evento propio para el cambio de volumen de una determinada clase de sonido. Gracias a la aplicación a las clases hijas (parámetro `Apply to Children`), la jerarquía vista en el apartado anterior cobra sentido y minimiza el trabajo de esta gestión.

Hay que decir que el código de *Blueprints* de la imagen no puede situarse a la ligera en el proyecto. A fin de cuentas, será donde este código se ubique el lugar desde el que podrá modificarse el audio. Si se quiere manejar solamente desde un nivel, como por ejemplo de opciones, bastaría con incluirlo ahí. Sin embargo, si traspasa tal límite porque, póngase el caso, también puede modificarse durante la partida de juego, entonces lo propio es situarlo en la instancia de juego (*GameInstance*). Es el caso de este proyecto. Para facilitar el manejo de estas funciones y su potencial expansión, se ha optado por basar un *Blueprint* en la instancia de juego propia que se había creado en C++. Así, basta con lanzar el evento propio que recogerá la instancia de juego (recuérdese, disponible desde cualquier nivel) sea desde donde sea.

Inciso: propiedad intelectual

Todos los efectos de sonidos que se han importado al proyecto provienen de los bancos de audio que, cada año, Sonniss libera con motivo de la GDC. Se permite su uso no comercial en cualquier circunstancia.

No obstante, más allá de los efectos de sonido se encuentran las dos canciones usadas durante el desarrollo de la partida. Estas son un bucle de pocos segundos obtenido de una canción con licencia propietaria y otro, este más largo, traído de una versión aficionada de otra canción de la misma naturaleza. Por el carácter académico de la entrega, su condición no lucrativa y la ínfima duración de los cortes usados se considera lícito de derecho legal su uso.

Interfaces

Todas las interfaces han sido construidas con recursos gráficos de un paquete de terceros, usando UMG para posicionamiento y animaciones. Todas las interfaces del juego han sido construidas a partir de envolver el lienzo principal de un lienzo escalado para prevenir una visualización incorrecta en distintos ratios de aspecto.

Una conclusión es clara: el sistema con que UMG proporciona la capacidad de animar componentes ofrece menos generalidad de la que debería. Al realizarse estas de manera absoluta, no es fácil crear animaciones genéricas tan recurrentes como por ejemplo desvanecimientos de opacidad o de escala, sino que debe crearse una de ellas por cada componente sobre el que quiera aplicarse.

Una posible solución a esto es el uso de componentes personalizados (*widgets* propios que son instanciados dentro de otros), pero se genera un nuevo problema: los componentes propios aparecen en el editor como un elemento de caja negra. Puede modificarse parte de sus detalles, como posicionamiento, anclaje, etc., pero más relacionados con su ranura de ubicación que con su interior. Por ejemplo, si se quiere crear un texto con una animación en bucle y guardar ese componente para después instanciarlo, no es posible mediante el editor cambiar el texto que cada una de las instancias tendrá. Debe hacerse por código (con la consecuente pérdida de productividad y flexibilidad intolerables) o usar las ranuras nombradas (*NamedSlot*), componentes que dejan un hueco preparado para incluirle nuevos miembros en cada instancia.

Otra conclusión es que usar un nivel para cada pantalla del menú no es una buena decisión de diseño *software*. Unreal proporciona medios para crear distintos *widgets* en pantalla y, dependiendo de las condiciones que sean necesarias, añadir a la visualización o sacar de ella. Esto eliminaría transiciones innecesarias entre niveles y por ende tiempos de carga, si bien ínfimos, visibles al jugador. Además, permitiría mantener los sonidos ambientales y/o la música sin necesidad de meterse en el uso de niveles persistentes o en una codificación innecesariamente compleja (como el guardado en la instancia de juego del momento exacto en que los audios se encuentran al momento de cambiar de pantalla, lo cual igualmente provocaría un parón momentáneo).

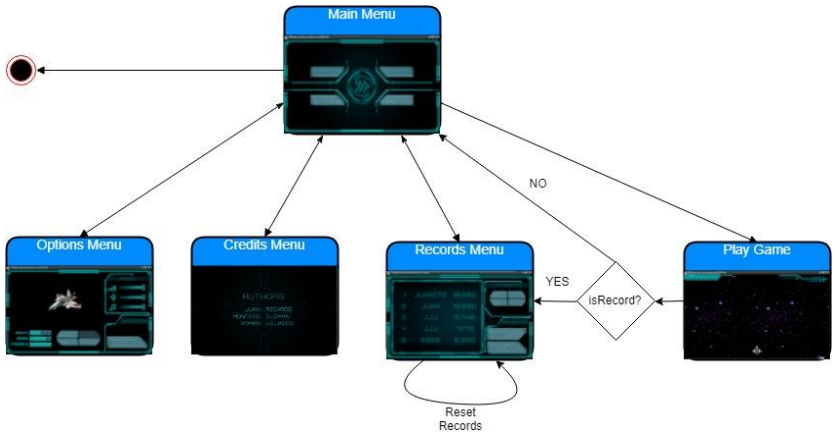
El anterior es el esquema que se usa, sin ir más lejos, cuando aparece una ventana flotante. De hecho, hablando de las ventanas flotantes, aparece de nuevo el problema de la poca generalidad que Unreal ofrece con su UMG: no es trivial (y debería) conseguir una ventana flotante genérica cuyo comportamiento varíe según qué instancia. Por ejemplo, no es fácil generar una ventana con respuesta SÍ/NO y usarla para cerrar el juego si se pulsa el botón de salir, mientras que para reiniciar el archivo siguiendo ambas con la Ley del Esfuerzo Proporcionado. La causa es que recoger los eventos de pulsación de botones es algo interno al componente, por lo que no es sencillo comunicarse con el exterior genéricamente (ya que el componente no conoce *per se* dónde está enmarcado para pasar la información). Habría que hacer uso de los eventos propios.

Flujo de ventanas

Existen los siguientes niveles en el juego:

- menú principal;
- records;
- opciones;
- créditos;
- nivel jugable.

Dado el requisito del enunciado por el cual cada ventana debía ser un nuevo nivel de Unreal, el flujo de carga de niveles se resume en la siguiente imagen:



Persistencia

Instancia de juego

La clase *GameInstance* se encarga de la persistencia entre niveles. Su código guarda los datos que requieren evitar la volatilidad:

- **Puntuación y nivel actuales:** dado que los niveles se van recargando a base de reiniciar un nivel en Unreal, es necesario que a ese nivel sobrevivan datos sobre los puntos que lleva el jugador y el nivel en el que se encuentra. Así, con cada siguiente nivel es tan sencillo como actualizar la instancia de juego y cargar esos datos al comienzo del mismo. Esto evita duplicar innecesariamente niveles, lo cual sólo sería adecuado si el crecimiento de la cantidad de hordas en los mismos no siguiese una fórmula determinada (en este caso $2N+1$, donde N es el nivel actual) o si las hordas estuviesen prefijadas por el diseño lúdico.
- **Nave elegida:** gracias a mantener entre niveles este dato es muy sencillo permitir que el jugador elija entre distintas naves en opciones y después la pantalla de juego pueda modificar el *pawn* del jugador acorde con esa nave. Como podrá verse en el diagrama de clases, se creó para este fin una enumeración propia con capacidad de exposición a *Blueprint*.

Además de eso, la instancia de juego base en código es tomada como base por un *Blueprint* que es en realidad la instancia de juego usada en la ejecución. Este *Blueprint* facilita sobremanera el control de los niveles de volumen que tienen las distintas clases de audio, como se dijo antes.

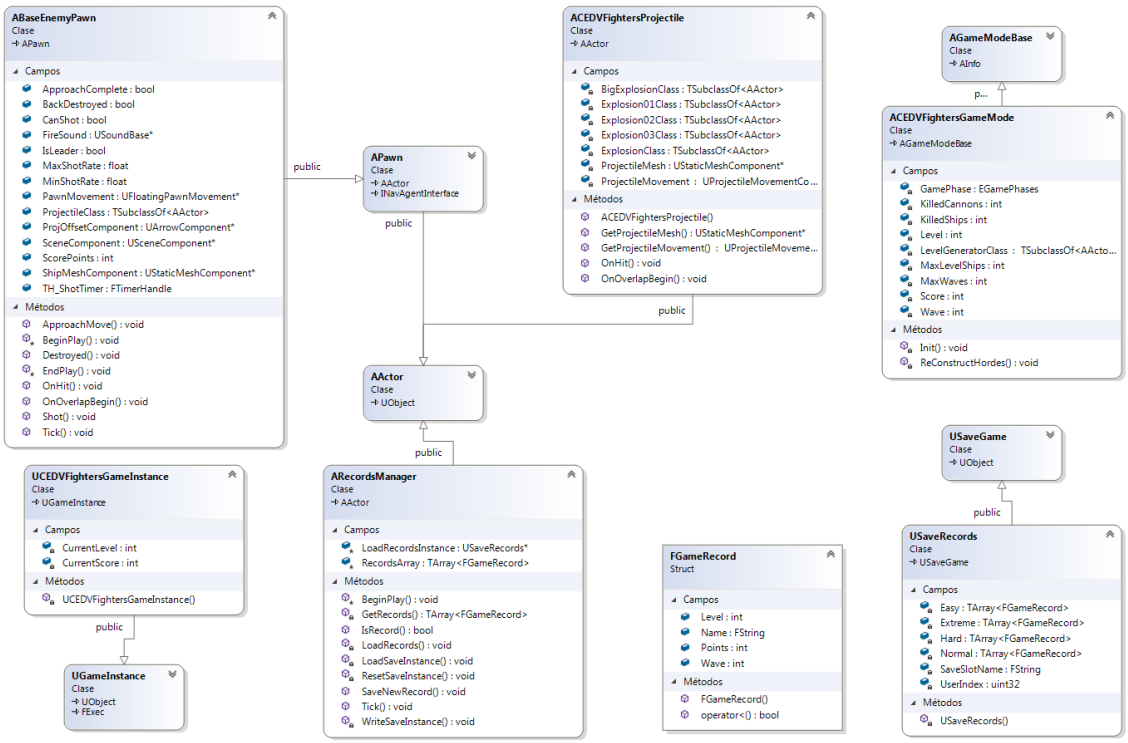
Datos guardados

Para el guardado y recuperación de puntuaciones se siguió el mismo esquema que en la entrega anterior. Una estructura permite almacenar los datos relacionados con un récord. Una clase para el guardado almacena colecciones de esas estructuras según cada dificultad disponible (y haciendo así uso de una sola ranura de guardado, facilitando las comunicaciones entre clases por ser las ranuras transparentes al exterior del gestor de guardado). El gestor de guardado, como actor, permite recuperar y sobreescribir cualquiera de esos vectores.

Estructura de clases

Código

A continuación se muestra un resumen autogenerado por Visual Studio.



Las anteriores son las clases de código en C++ que conforman el proyecto (sin contar los tipos de enumeraciones propios como las fases de juego, las dificultades…). Pueden diferenciarse fácilmente por un lado aquellas que heredan de la instancia de juego, el modo de juego y el guardado, que se encargan de gestión más allá de la jugabilidad en sí, y por otro las que son actores o peones, a excepción de *RecordsManager* que siendo actor es un gestor de guardado (se profundizará más adelante). A estas últimas, en esencia personajes enemigos y proyectiles, habría que sumar el peón propio del jugador, que Visual Studio deja fuera del diagrama por alguna razón, y el enemigo jefe, desarrollado en *Blueprints*.

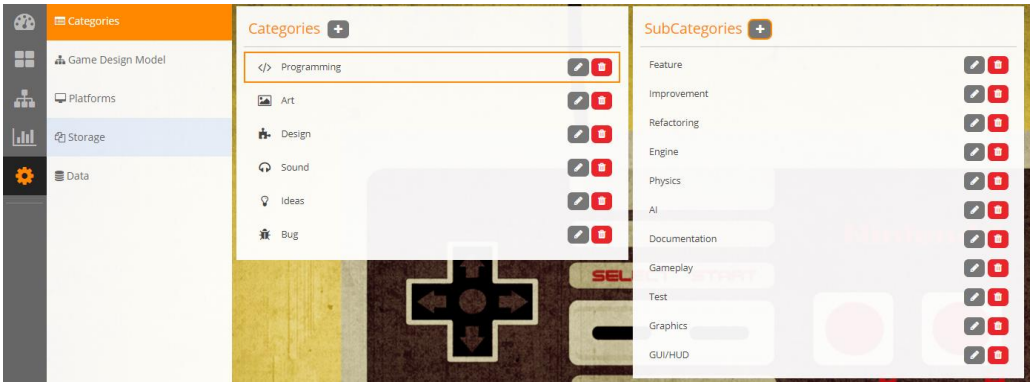
Además, existen árboles de comportamiento y manejadores de inteligencia artificial, estructurados tal y como se ha visto en el contenido del curso. Y también un generador de niveles en tiempo de compilación y/o ejecución que permite fácilmente cambiar la cantidad de oleadas enemigas a superar en cada nivel. Este generador se ha implementado de manera que sea genérico y aleatorio, eligiendo entre todos los tipos de horda (convoy, colmena, línea de vanguardia y tríada, derivando todas de un tipo base para extenderse fácilmente) cada vez que se cambia la cantidad de oleadas que debe generar. Es una decisión difícil de entender en un proyecto real de videojuego con aspiraciones comerciales, ya que el apartado de diseño lúdico queda muy limitado. Sin embargo, centrándose la entrega en el desarrollo tal como se indica en el curso, es mucho más interesante el desarrollo de un generador aleatorio que la disposición directa de hordas desde el editor, en el rol que manejaría un diseñador de niveles.

Herramientas de gestión

A modo de acercamiento de cara al proyecto final, se han usado para este desarrollo algunas herramientas de gestión de forma relajada.

Hack & Plan

Hack & Plan es un gestor de tareas análogo a otros como Trello, sólo que enfocado en proyectos relacionados con la industria videolúdica. Así, guarda peculiaridades como una estructura de columnas propia del flujo de trabajo de proyectos de videojuegos, categorías y subcategorías ligadas a tales y, lo más importante, un modelo de diseño de juego. Este modelo, jerárquico, permite asociar tareas a cada elemento lúdico, como una mecánica, un actor, un mundo o un objeto.



Toggl

Toggl es una conocidísima herramienta para gestionar tiempos. Sus temporizadores se asocian a tareas concretas dentro de proyectos, con una cantidad indeterminada de etiquetas. Lamentablemente, Hack & Plan aún no ofrece conexión con Toggl, como sí hace por ejemplo Trello, consiguiendo que las tareas, sus estimaciones de tiempo y su trabajo efectivo se enlace automáticamente.

Git/GitHub

Si la anterior es una herramienta popular, qué decir de Git y GitHub. Permiten, en dupla, el control de versiones y el repositorio remoto. Su uso ha facilitado el trabajo conjunto en el proyecto, pese a que aquellos archivos que Unreal marca como recursos (*assets*) son binarios y por tanto hay que tener especial cuidado con ellos, no siendo posible mezclarlos tras su actualización concurrente. Esta es quizá la desventaja más importante de no usar el sistema de control de versiones y repositorio que incluye Unreal internamente, así como la posesión de recursos por parte de los distintos desarrolladores (que, esencialmente, relaja la cantidad de recursos del explorador de Unreal para cada uno de los miembros del proyecto). No obstante, las limitaciones de este sistema interno no compensan tales ventajas.

Enlaces

Elemento	Enlace	Descripción
Vídeo	YouTube	Breve vídeo del proyecto en funcionamiento
Registro de tareas	Hack&Plan	Proyecto en servicio de gestión colaborativa enfocado a videojuegos
Repositorio	GitHub	Estado actual del código con control de versiones histórico desde el inicio

Otra información

Controles para pruebas y depuración

Controles del juego

El jugador se mueve con el **esquema WASD** y el disparo puede realizarse con la **barra espaciadora** o bien con la **flecha hacia arriba** (remanente de unas primeras pruebas con una plantilla de Unreal).

Mediante la **tecla P** se accede al menú de pausa siempre y cuando el jugador esté en una partida activa (no en la transición de principio de nivel ni ya en la pantalla de muerte).

Tecla M: modo invencible

El jugador no recibe daños. Alterna entre activada y desactivada.

Tecla N: muerte instantánea

Simula que el jugador ha perdido toda la vida. Útil para ir hacia las transiciones de muerte de una manera muy rápida (por ejemplo, para crear algún récord rápidamente tras reinicio de los datos guardados).

Tecla U: enemigos restantes

Visualiza un mensaje de depuración para que el jugador sepa cuántos enemigos quedan antes de llamar al jefe del nivel.

Tecla X: cursor animado propio

En el menú principal, mientras cargan las animaciones iniciales, esta tecla permite que se visualice un cursor propio creado y animado expresamente para la entrega, pero descartado después por un fallo irreparable: en ciertas pantallas, el *viewport* en que se encuentra el cursor parece desplazarse respecto al resto de *widgets* y dar un funcionamiento incorrecto.

Tecla C: cambio de cámara

Permutación entre la cámara normal, una a la espalda del jugador y otra alejada lo suficiente como para visualizar las hordas fuera de pantalla.

Errores conocidos

Cambio de dimensiones durante ejecución

Si bien las interfaces hacen uso de componentes de escalado y se ajustan a ellos, evitando así que un cambio en el ratio de aspecto o en la resolución pueda afectar a la visualización de la interfaz, llevar a cabo estos cambios mediante redimensionamiento de la ventana durante la ejecución puede hacer que las animaciones fallen. Esto se debe a lo comentado anteriormente sobre el carácter absoluto de los valores que las animaciones usan: cambiar el ratio de aspecto durante la animación es

generar una inconsistencia entre tales valores y por tanto al finalizar su ejecución puede dar con un estado final distinto del deseado.

Enemigos fuera del alcance del jugador

En muy raras ocasiones los enemigos con árboles de comportamiento determinados pueden situarse al borde de su espacio de navegación y avanzar sin fin, haciendo imposible al jugador acabar con ellos. A fin de evitar estos y otros errores se han colocado fuera del *viewport* los típicos colisionadores recolectores de basura. En este caso, mandan la destrucción voluntaria de los actores enemigos que le llegan, almacenándose como enemigo eliminado y permitiendo la correcta ejecución del resto del nivel (el jefe no aparece hasta que el número de naves eliminado sea igual al total que había al principio). Los mismos "recolectores de basura" se han colocado en otros límites del espacio de juego para hacer lo propio con proyectiles (que además también tienen tiempo de vida o *lifespan*). Esta técnica es muy recurrente en desarrollo de videojuegos, evitando mantener en memoria instancias que ya no participan del escenario de juego activo.

Trabajo futuro

Interfaces genéricas

Como se ha dicho, trabajar con interfaces genéricas es complejo en UMG, pero viable a través de las ranuras nombradas y el uso de componentes propios con sus animaciones ya preparadas. Así, próximamente se hará uso de ellos y se tratará de encapsular las animaciones o los comportamientos repetidos en lienzos propios (*custom widgets* con *canvas*). Otra opción es el peligroso uso del evento *PreConstruct*, el cual se aconseja en la memoria usar con mucha cautela debido a su doble naturaleza (está presente tanto en ejecución como en tiempo de edición).

Mejor encapsulamiento del guardado

Aunque la clase creada para tramitar la persistencia tiene un buen diseño, su condición de actor puede no ser la mejor opción si hay muchas pantallas distintas en que se requiera recuperar o sobreescribir una ranura de guardado.

Último récord remarcado

Por motivos de tiempo y prioridad ha quedado fuera del proyecto el remarcar el récord recién conseguido por el jugador. Había algunas ideas para diseñarlo, como usar la instancia de juego para guardar un puntero a récord y compararlo con aquellos que la pantalla de marcadores fuese cargando para mostrar, cambiando el color de la tipografía al que coincidiese con ese récord guardado.

Tras esta muestra se eliminaría la referencia del puntero para evitar que se quedase en caso de que en la siguiente partida el jugador no consiguiera un nuevo récord.

Otra idea era guardar directamente en la instancia de juego un valor lógico sobre si se ha conseguido recientemente un récord o no y, anexamente, la posición de ese récord y su dificultad. De la misma forma podría remarcarse con un color distinto la tipografía del texto que coincidiese con esa dupla.

Prefabricados de jugador

En esta entrega se trabajó con distintas opciones para que el jugador escogiera su dificultad y, con ello, su nave asociada, teniendo cada una distintas estadísticas y por supuesto aspecto. Tales datos se cargan directamente al comienzo del nivel según la dificultad que guarda la instancia de juego. Habría

sido mucho más adecuado asociar a cada dificultad unos datos por defecto. De este modo podría reusarse tal conjunto de datos prefabricados tanto en la muestra de aspecto en la pantalla de opciones como al cargar el nivel de juego. También facilitaría la escalabilidad del juego, de manera que añadir nuevos aspectos/dificultades se reduciría a crear un nuevo elemento a la enumeración y asociarle otro prefabricado.

Nota: Unreal no trabaja con prefabricados de entidad/componente de la forma en que lo hacen otros motores, ni tampoco con objetos prediseñados (*Scriptable Objects*, por ejemplo, en motores como Unity). Una posible solución a este diseño en Unreal consistiría en la creación de tablas estáticas de las cuales se leyesen los datos.