

Informe de Práctica Profesional

CC5901 - Optimización y Mejoras de Sistemas para AOne Games

Empresa: AOne Games S.p.A.
Alumno: Daniel Soto G.
Carrera: Ingeniería Civil
Especialidad: Ciencias de la Computación
RUT: 19.308.472-9
E-Mail: danielsoto.3004@gmail.com
Teléfono: +56 9 4735 6543
Realización: Enero 2019

Fecha de entrega: 13 de abril de 2019
Santiago, Chile

Índice de Contenidos

1. Resumen	3
2. Introducción	4
3. Problemas Abordados	5
4. Objetivos de la Práctica	5
5. Metodología	6
6. Descripción de las Soluciones	7
6.1. Optimización del modo arcade	7
6.2. Refactorización de la visualización de los personajes	8
6.3. Rediseño de la creación de niveles para el modo historia	9
7. Discusión y Reflexión sobre la Práctica	11
8. Conclusiones	12
Referencias	13
Anexo A. Mapa del Modo Arcade	14
Anexo B. Modo Historia	16
B.1. Documentación del sistema implementado	19

Listado de Figuras

A.1 Mapa del modo arcade dentro del juego.	14
A.2 Implementación original del mapa en blueprints.	14
A.3 Rutina de construcción final del mapa de arcade.	15

B.1	Interfaz para editar las timelines anteriores.	16
B.2	Una timeline lineal.	16
B.3	Una timeline con distintas ramas de ejecución.	16
B.4	Una función que define un evento simple.	17
B.5	Una función que define un evento complejo, con más de una salida.	17
B.6	Tres eventos que definen un loop para hacer una pelea mejor de N .	17
B.7	Definición del primer evento de un mejor de N .	17
B.8	Definición del segundo evento de un mejor de N .	18
B.9	Definición del último evento de un mejor de N .	18

1. Resumen

En este informe de práctica profesional se detalla todo el proceso y los resultados obtenidos luego del mes de trabajo en AOne Games. Esto incluye el porqué se contrató un practicante en la empresa, definición de los problemas a resolver, los objetivos que se intentaron cumplir, las metodologías usadas, descripciones detalladas de las soluciones y discusión y conclusiones.

Los resultados obtenidos fueron buenos, cumpliendo satisfactoriamente los objetivos planteados. A pesar de esto, las metodologías utilizadas no fueron óptimas, se repitió trabajo en ciertas partes del proceso, lo que quizás pudo haber sido optimizado. Se aprendió sobre cómo trabajar en un grupo relativamente grande, y la importancia de tener reuniones periódicas con el equipo de trabajo.

2. Introducción

Esta práctica profesional fue realizada en AOne Games, una compañía de videojuegos chilena. La organización desarrolla el videojuego de peleas Omen of Sorrow, lanzado a fines del año 2018. Al momento de la práctica se encontraban en el proceso de adaptar el juego para lanzarlo en nuevos sistemas, y optimizar distintos módulos de este.

La compañía necesitaba un practicante para ayudar en la optimización de estos sistemas, pues muchos se encontraban implementados de maneras sub-óptimas. Esto se debió a que el motor del videojuego, Unreal Engine, soporta un sistema de programación visual el cual es fácil de aprender. Requeriendo menor entrenamiento, miembros del equipo que no eran programadores pudieron definir ciertas funcionalidades del juego. La desventaja es que este sistema es menos eficiente que una funcionalidad equivalente en código C++.

Durante la práctica, los superiores viajaron a Japón para coordinar el traslado del juego a las máquinas de arcade de TAITO. Debido a esto se pasaría parte del período de práctica con supervisión relativamente baja. Durante este período la tarea era encontrar debilidades en sistemas actuales que se podrían reparar.

El trabajo realizado consistió en analizar archivos del juego buscando puntos que pudiesen ser optimizados. Luego se diseñaron distintas metodologías para reparar estos puntos. Estas metodologías fueron puestas en práctica, y los resultados fueron verificados con otros miembros del equipo. Otro aspecto central del trabajo fue analizar procesos inefficientes dentro del motor, y diseñar nuevas herramientas para mejorarlos. Similarmente, se desarrolló una metodología y los resultados fueron validados con los superiores.

Se trabajó con tres herramientas distintas, una para diseñar el juego, otra para programarlo y la última para coordinar el trabajo entre miembros del equipo. El diseño de Omen of Sorrow fue realizado mediante Unreal Engine, un motor de videojuegos creado por Epic Games [1]. La programación fue hecha en código C++ con el editor Visual Studio Community. Finalmente, la coordinación de versiones fue realizada con Perforce, un programa de control de revisiones centralizado [2].

3. Problemas Abordados

Al ser desarrollado en Unreal Engine 4, muchos sistemas de Omen of Sorrow se encontraban implementados con los Blueprints [3] del motor. El precio de ser intuitivos de programar es que la ejecución se ve ralentizada, y es propensa a memory leaks. Por otra parte, algunas herramientas fueron desarrolladas en código C++, pero las interfaces que usarían los diseñadores para utilizarlas eran muy complejas de usar.

Debido a que la ejecución debe ser fluida, es fundamental que todas las componentes del juego sean eficientes y económicas en espacio. Un sistema anterior encargado de manejar la interfaz de usuario tenía un memory leak, lo que causaba que el juego fuera progresivamente más lento, hasta que se acababa memoria. En contraste, otros módulos bien optimizados pero con malas interfaces son problemáticos, pues lentifican demasiado la utilización de estos.

A pesar de que el videojuego ya se encuentre lanzado, arreglar estos errores no pierde importancia. El juego puede ser actualizado con las optimizaciones realizadas. Incluso la mejora de algunas interfaces para el equipo facilita agregar contenido extra, a un menor costo de horas hombre.

4. Objetivos de la Práctica

El objetivo propuesto al llegar a la práctica trataba sobre optimizar los Blueprints existentes del videojuego. Esto consistiría en buscar debilidades en los sistemas actuales, y arreglar las fallas de diseño en código. Además se consideraba la mejora de algunas herramientas, para lograr una mayor facilidad de uso.

Un desafío técnico relevante fue aprender el uso de C++ en conjunto con Unreal Engine y sus Blueprints. Por otra parte, fue necesario trabajar con Perforce, una herramienta de versionamiento centralizada. Organizacionalmente fue esencial hablar con otros miembros del equipo para averiguar el funcionamiento de ciertos sistemas, y poder descubrir el impacto que tendrían algunos cambios.

Lo producido en esta práctica será utilizado tanto por usuarios finales como por empleados de AOne Games. La optimización de los distintos módulos de la aplicación va a ser aprovechada por los jugadores, teniendo un mejor rendimiento del juego. La mejora de herramientas va orientada hacia equipo de diseñadores de la compañía, simplificando procesos ya existentes.

5. Metodología

Debido a que se resolvieron dos problemas bastante distintos, se trabajó con dos metodologías distintas. En el lente de optimizar, el proceso consistía en analizar las Blueprints defectuosas y buscar dos cosas en el código: repetición y partes optimizables por estructuras de datos. Luego estos cambios y mejoras serían escritos en C++, y configurados para ser llamados desde el flujo original, en el lugar de los segmentos repetidos o sub-óptimos. Esto se hizo sin eliminar la funcionalidad anterior de los archivos. Al confirmar que el funcionamiento era el mismo, se eliminaban las funciones deficientes.

En cambio, desde la perspectiva de mejorar herramientas el método se centró en analizar el sistema actual para encontrar puntos donde podría haber un mejor proceso. Al descubrir una debilidad, se diseñaba una implementación que la supliera. Luego esta era implementada y el contenido antiguo se pasaba a la versión nueva. Si el funcionamiento era correcto, se mostraba el resultado a los miembros del equipo que utilizarían la interfaz creada y con el feedback obtenido se iteraba sobre la solución. Se continuó así hasta conseguir un resultado satisfactorio.

La primera metodología fue sugerida por el supervisor y obtuvo resultados bastante buenos. El trabajo fue rápido y eficiente de esta manera. La segunda fue de fabricación propia, con una solución final satisfactoria, pero un proceso no muy efectivo. El problema de esta era que iterar implicaba repetir tareas. A esto se suma que la traducción de contenido entre versiones era muy lenta pues era labor manual y repetitiva.

Evaluar los resultados obtenidos resultó bastante simple. En el caso de la optimización de blueprints, bastaba con comprobar que el comportamiento original se mantuviese. No es necesario verificar que el rendimiento es mejor, debido a que inherentemente la implementación en el sistema de Unreal Engine será más lenta que una equivalente en C++. Para la mejora de usabilidad se hacía suficiente confirmar con quienes utilizarían la nueva versión si efectivamente era ventajosa sobre la previa.

6. Descripción de las Soluciones

Todo el trabajo hecho en AOne Games se centró alrededor del uso de una componente particular de Unreal Engine 4, sus blueprints [3]. Este sistema consistía en un lenguaje de programación visual, donde las funciones y valores se representan con nodos, y el orden de las llamadas se define uniendo estos nodos con un cable blanco. Los parámetros y resultados de las funciones se pasan a través de líneas de otros colores. Una particularidad importante de este sistema es que existen un tipo especial de métodos, llamados puros. Estos no necesitan que se defina la secuencia de ejecución, pues su resultado no muta la clase.

6.1. Optimización del modo arcade

Este problema en particular trataba de optimizar la blueprint que se encargaba de la visualización del mapa del modo arcade (Figura A.1). La modalidad de juego consistía en una serie de 7 u 8 combates consecutivos, elegidos aleatoriamente. La pantalla muestra es una mesa fichas de la etapa en la cual se llevará a cabo la próxima batalla y contra quién para el combate actual y los anteriores. Es mostrado al comenzar, y después de cada lucha. Inicialmente el jugador se encuentra donde se jugará primero, pero luego se reproduce una animación moviéndose al nivel correspondiente adonde su contrincante lo espera.

Esta blueprint fue desarrollada originalmente por alguien sin experiencia programando, por lo que no se usaron estructuras de datos, y al elegir las curvas de movimiento del jugador y las texturas que se debían mostrar para cada enemigo, se hacían comparaciones con todos los valores posibles. Como consecuencia la ejecución tenía muchas ramas distintas, lo que implicaba código duplicado, y aumentaba considerablemente el riesgo de cometer un error al editarla (Figura A.2). Las variables aleatorias eran definidas al comenzar el modo, y se guardaban en un struct que almacenaba todas las batallas, en qué orden, y sus estados actuales.

La solución intuitiva fue utilizar diccionarios donde se almacenarían las texturas de los enemigos y las curvas necesarias. Así modo el resultado final simplemente obtenía el elemento del diccionario relacionado con la llave entregada, en orden constante. Se usó el valor del personaje contra el que se pelearía con el objetivo de hallar los componentes del enemigo. Similarmente, en el caso de los niveles se utilizó el identificador de este. Cabe notar que para cada mapa había más de una sola variable asociada. Estos conjuntos almacenados fueron representados por distintos structs, que permitieron un fácil acceso a ellas.

El almacenamiento de las curvas era más complicado, pues además de necesitar los puntos de comienzo

y el fin, estas eran bidireccionales. Se traduce en que al conseguirlas de la estructura de datos, se debía reconocer la dirección en que se tenían que recorrer. Se almacenó la trayectoria en un struct, el cual tenía definido un inicio y un fin, junto a un campo booleano que describía el sentido de esta. Debido a que la elección no era tan simple como extraer un elemento de un diccionario, se optó por un arreglo adonde la extracción consistió en un método que buscaba la línea correspondiente, y asignaba el hacia donde recorrerla a base del orden descrito en el contenedor y el camino buscado.

Por último, se extrajo del blueprint la rutina de actualización de la posición de la cámara y el jugador. Se debieron almacenar referencias a funciones que entregaban el progreso a lo largo de la curva, dependiente del tiempo (representado como un número entre 0 y 1). Estas fueron usadas para crear timelines que eran actualizadas todos los frames, y que definían la ubicación de las entidades necesarias.

Soportar todo este comportamiento nuevo requirió configurar en el blueprint las referencias a las texturas, curvas y animaciones del mapa para que pudiesen ser utilizadas por el código. El resultado final en el motor fue un archivo vacío a excepción de su rutina de construcción (Figura A.3), donde simplemente se asignan los objetos relevantes a variables en C++. A pesar de verse tan grande, es bastante comprimido, pues se configuran cientos de valores distintos, ordenados en structs representando niveles, enemigos y caminos.

6.2. Refactorización de la visualización de los personajes

Comparativamente fue un problema simple. Cada personaje del juego tiene efectos visuales que se reproducen dependiendo de su estado actual. Esto incluye sus sombras y partículas. La visualización se encontraba implementada en blueprints específicos de los luchadores. Así llegamos al conflicto, entre las versiones particulares se repiten muchos segmentos de código.

Se analizaron los blueprints de los personajes para encontrar puntos comunes. Estos fueron traducidos a C++, y se insertaron llamadas al código dentro de cada implementación. Esto fue luego probado con todos los luchadores, y no se encontraron diferencias con la versión anterior.

Cabe notar que durante esta refactorización, se encontró un bug sobre un personaje en específico. Este no era capaz de aplicarle sus debuffs al otros. Se buscó donde estaban las líneas de código responsables por el error, se notificaron a los programadores adecuados, y fue arreglado.

6.3. Rediseño de la creación de niveles para el modo historia

La herramienta modificada fue una creada con el objetivo de poder programar una secuencia de eventos que se ejecutarían en el modo historia, antes y después de las peleas de los niveles. Esta consistía almacenar varias listas de **timeline assets** (Desde ahora equivalentes a **instrucciones**, **comandos** y **acciones**. Ver figura B.1). Cada uno de éstos contenía instrucciones específicas para ejecutar sobre el juego, programada en C++, definiendo un *Command Pattern*. Todas tienen una referencia a su nombre y el de la acción que vendrá a continuación. Al completar su ejecución se buscan en la lista completa todos los comandos con el mismo nombre que la siguiente acción, y se activan para que ejecuten su código.

El sistema tenía varios problemas. Mirándolo desde la perspectiva de la eficiencia, cada frame se hacían llamadas a la totalidad de los comandos en la lista activa y todas las instrucciones de todas las listas eran instanciadas al comenzar el juego, lo que gastaba cerca de 500KB de memoria RAM. Por el lente de la usabilidad tenemos aún más defectos, entre ellos que se almacenaba un número arbitrario de listas, las cuales no tenían ningún orden interno y no tomaban en cuenta cuantas se utilizaban, lo que causaba que fuese complejo de usar y desordenado. Además, la manera en la que se referencian las instrucciones es poco robusta, pues con un simple error de tipeo los assets que uno quería que se ejecutaran, no lo harán.

Se comenzó atacando a los problemas de eficiencia. Evitar llamadas innecesarias requirió un sistema de contenedores llamados **eventos** (Desde ahora equivalentes a **conjuntos**, **colecciones** y **agrupaciones**). Equivalen a una colección de comandos asociados al mismo nombre en la versión anterior. Estos eran los encargados de recibir los mensajes cada frame, y sólo si se encontraban activos, repetirlos a sus assets correspondientes. Cuando una de las instrucciones contenidas termina su ejecución, envía un mensaje de activación a la agrupación configurada como siguiente. Al completar todas las acciones, se desactiva el conjunto completo y se paran de distribuir las notificaciones.

Para la usabilidad, se cambiaron las llamadas mediante nombres iguales a activations directas mediante referencias de un asset al siguiente evento. Al completar su llamada, una instrucción enviaba un mensaje de activación al conjunto que viene a continuación. De este modo se evitaron errores de secciones no reproducidas por escribir mal un identificador.

Para el orden, se eliminó la necesidad de tener más de una lista. Se incluyó el manejo del flujo de las peleas, es decir la entrada y salida de secciones pre-programadas y combate, a la herramienta. Se agregaron comandos que pausaban la ejecución de la timeline hasta que se cumpliera cierta condición, como una

batalla ganada. Así se logró que un nivel quedase expresado en una única linea de tiempo, con periodos donde el jugador puede controlar a su personaje, y de reproducción de animaciones y diálogo que exponen la historia.

Para soportar estos cambios anteriores se movió la definición de los eventos desde los campos del blueprint en el motor, a los nodos definidos en este. Se diseñó un patrón de implementación tal que las timelines se pueden visualizar ordenadas temporalmente (Figuras B.2 y B.3). Concretamente, representa un conjunto de llamadas a funciones puras, cuyos resultados son utilizados en una función final, que inicializa todo el árbol. Como un efecto lateral se evitó la instanciación de los comandos al lanzar el juego, gracias a que la creación de esta estructura, junto a todos sus conjuntos y instrucciones, espera a que se llame la función previamente mencionada.

El patrón de implementación lo podemos definir de la siguiente manera. Se definen funciones puras dentro del blueprint, que reciben un input y tienen uno o más outputs, hechas con la intención de crear un nuevo evento. Estas salidas y entradas son assets, los cuales activan a la agrupación actual y activarán a las siguientes, respectivamente (Figuras B.4 y B.5). En cada conjunto se guardan referencias a la instrucción que lo activó. Se necesitan guardar referencias a los comandos anteriores para poder encontrar la raíz del árbol, navegando desde una hoja hasta llegar a un nodo sin un elemento que lo active. La flexibilidad en la construcción permite diseñar estructuras avanzadas, como unir distintas ramas en una misma mediante una acción *MergeAssets* y separar ramas dependientes de las condiciones cumplidas, mediante un conjunto de assets especiales llamados *TriggerAssets*. Un ejemplo práctico de esto son las peleas mejor de tres (Figuras B.6, B.7, B.8, B.9), que no eran implementables mediante las líneas de tiempo anteriormente. Una descripción más detallada de este sistema se puede encontrar en el documento B.1.

Por último, la definición de assets dentro de los blueprints consistió en crear una librería de métodos estáticos puros que crean los eventos e instancias de comandos y configuran las referencias necesarias. Estas son importantes debido a que no se pueden entregar clases en lugar de argumentos fácilmente. Todas las funciones se definen puras para evitar que sea necesario definir un orden de ejecución (Ver [4]). Esto simplifica visualmente las timelines, pero su precio es que se vuelve crucial asegurar que la creación de cada evento sea llamada. La solución es que todos los hijos del árbol deben tener por lo menos un output, el cual debe ser almacenado en un arreglo que se pasa como argumento a una función que sólo utilizará un elemento de este.

7. Discusión y Reflexión sobre la Práctica

Llegar a soluciones satisfactorias fue en gran parte gracias a los conocimientos adquiridos en ramos como Metodologías de Diseño y Programación, para poder reconocer patrones de diseño ya implementados y diseñar soluciones alrededor de ellos. Algoritmos y Estructuras de Datos, y Diseño y Análisis de algoritmos fueron útiles para reconocer debilidades en los sistemas actuales, e implementar nuevo algoritmos y estructuras de datos para optimizarlos. Por último, Ingeniería de Software ayudó a trabajar con el equipo, y poder ponerse de acuerdo en qué cosas era crítico hacer primero.

Hizo falta un mayor conocimiento programando en C++, en conjunto a Unreal Engine. Debido a esta falta de experiencia, los primeros días de la práctica se pasaron estudiando el código relevante y aprendiendo las prácticas comunes utilizando aquellas herramientas.

El trato con el resto del equipo fue bastante bueno cuando se trataba de relaciones de a pocos. Un problema importante observado en la oficina fue que no existían reuniones para coordinar las tareas que se debían llevar a cabo. Esto implicaba que uno debía ir y preguntarle a un miembro en particular con cual tarea se debiera continuar. Fue una buena experiencia para mejorar la proactividad en la oficina, y reforzar la importancia de tener reuniones constantes con el resto del equipo.

Se aprendió a tomar los comentarios del resto del equipo sobre las soluciones desarrolladas, y aplicarlos para obtener una mejor versión. Por ejemplo, para el rediseño del modo historia el sistema desarrollado pasó por cerca de 4 iteraciones, cada una fue mostrada a los miembros del equipo que las utilizarían después con la intención de obtener sus opiniones sobre ellas. Así se logró desarrollar un sistema que cumplía lo pedido inicialmente, e incluía nuevas funcionalidades.

Por último, se obtuvo una gran cantidad de experiencia analizando herramientas existentes para encontrar posibles mejoras a hacer. Esto se debió a que el supervisor no siempre estuvo disponible para pedir consejos. Como consecuencia se debió tomar una actitud más independiente para buscar las tareas, y proponer cambios sobre ellas.

8. Conclusiones

Los objetivos iniciales lograron ser cumplidos satisfactoriamente. Las blueprints del modo arcade fueron simplificadas por un factor considerable, y consecuentemente recibieron una optimización importante. El funcionamiento de las visualizaciones de los personajes del juego se logró acoplar a una clase base sin cambiar el funcionamiento externo. Por último, el sistema diseñado para crear niveles del modo historia logró representar la misma funcionalidad, e incluso algunas extra, como definir ramas sujetas a una condición, ciclos y unir ramas distintas en una sola.

El tiempo en la compañía enseñó lecciones sobre organización de los equipos y procesamiento de críticas constructivas. Se aprendió a diseñar sistemas amigables para sus usuarios. Finalmente, se obtuvo experiencia sobre cómo trabajar en sistemas con *garbage collectors* personalizados, como el de Unreal Engine.

El conocimiento adquirido en la carrera fue de especial utilidad para resolver los problemas planteados en la práctica. Este permitió entender ciertas metodologías y desarrollar nuevas. Tanto como poder optimizar sistemas ya existentes y tener buen trabajo en equipo dentro de la oficina.

Personalmente la práctica logró brindar bastante conocimiento útil sobre espacios de trabajo en compañías pequeñas. A pesar de esto, la falta de organización dentro de AOne Games causó un poco de decepción, pues se esperaba un mayor nivel de coordinación dentro del equipo. En general fue una experiencia positiva, con aprendizajes de que cosas se deberían hacer al diseñar sistemas, y cuales no al manejar una compañía.

Referencias

[1] Página web de Unreal Engine.

<https://www.unrealengine.com/en-US/>

[2] Página web de Perforce.

<https://www.perforce.com/>

[3] Documentación del sistema de blueprints de Unreal Engine.

<https://docs.unrealengine.com/en-us/Engine/Blueprints>

[4] Documentación de las funciones en los blueprints de Unreal Engine.

<https://docs.unrealengine.com/en-us/Engine/Blueprints/UserGuide/Functions>

Anexo A. Mapa del Modo Arcade

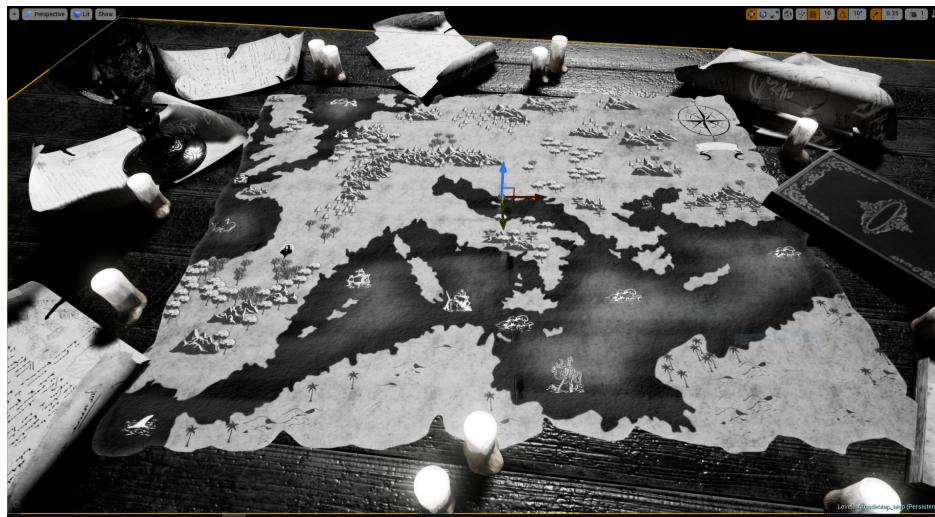


Figura A.1: Mapa del modo arcade dentro del juego.

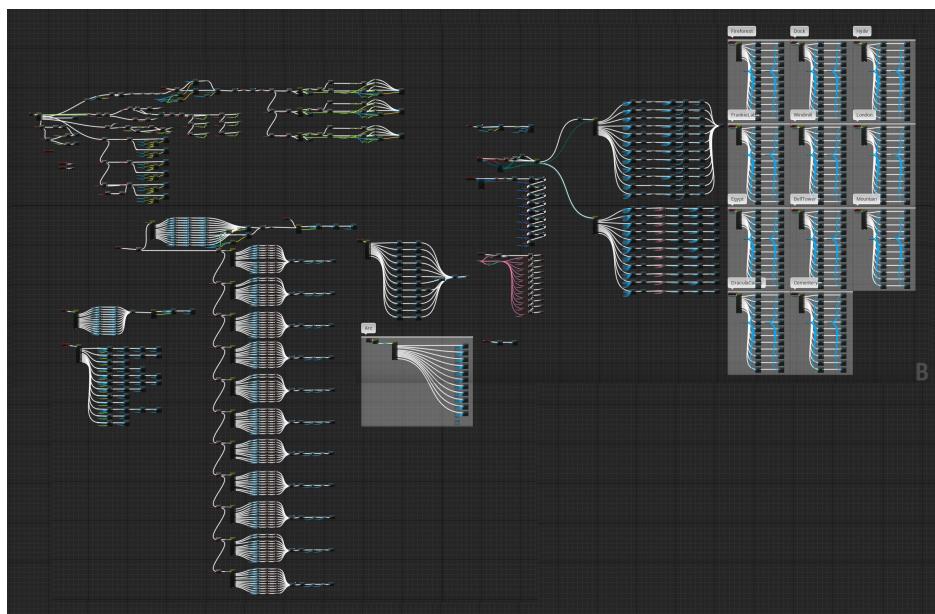


Figura A.2: Implementación original del mapa en blueprints.

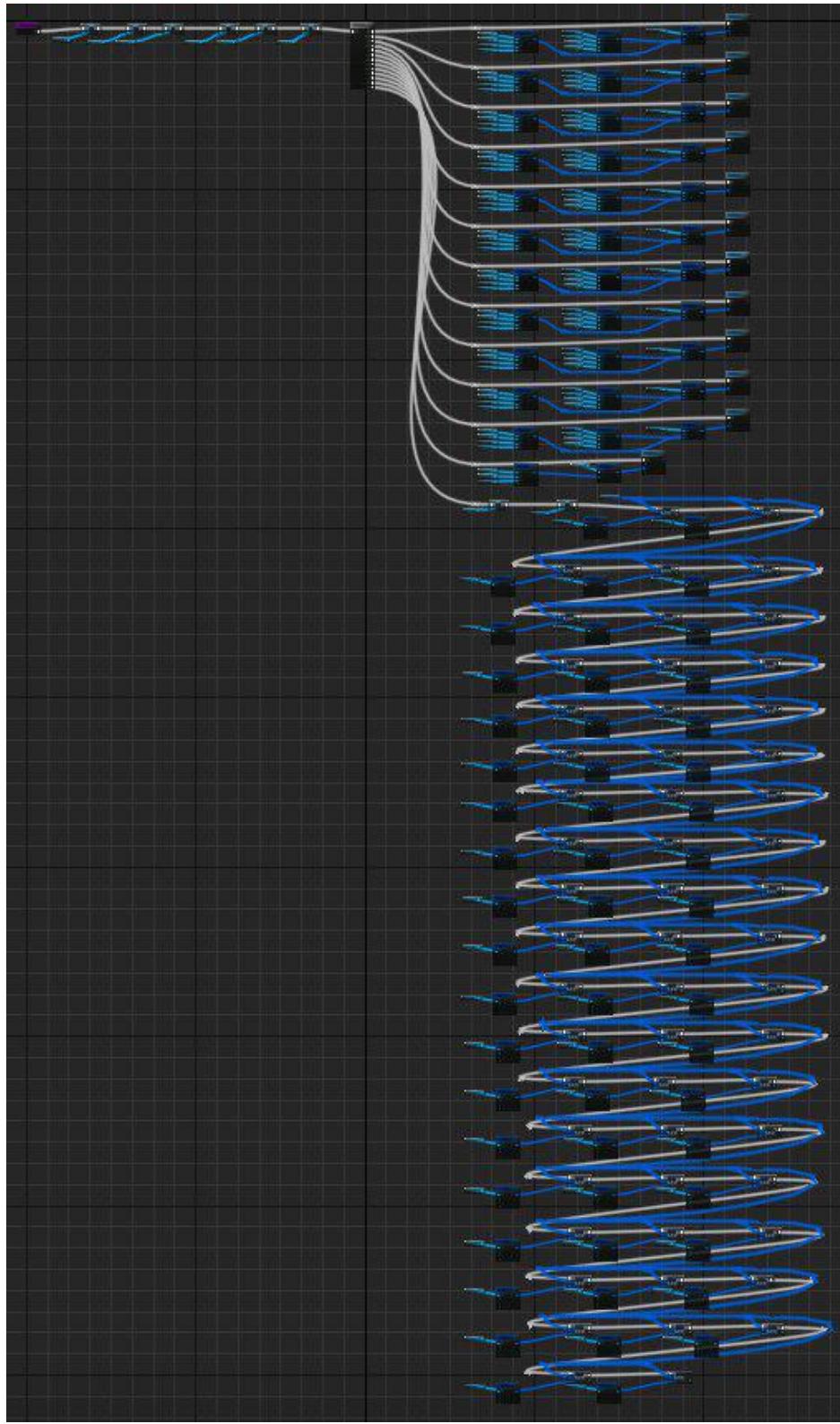


Figura A.3: Rutina de contrucción final del mapa de arcade.

Anexo B. Modo Historia

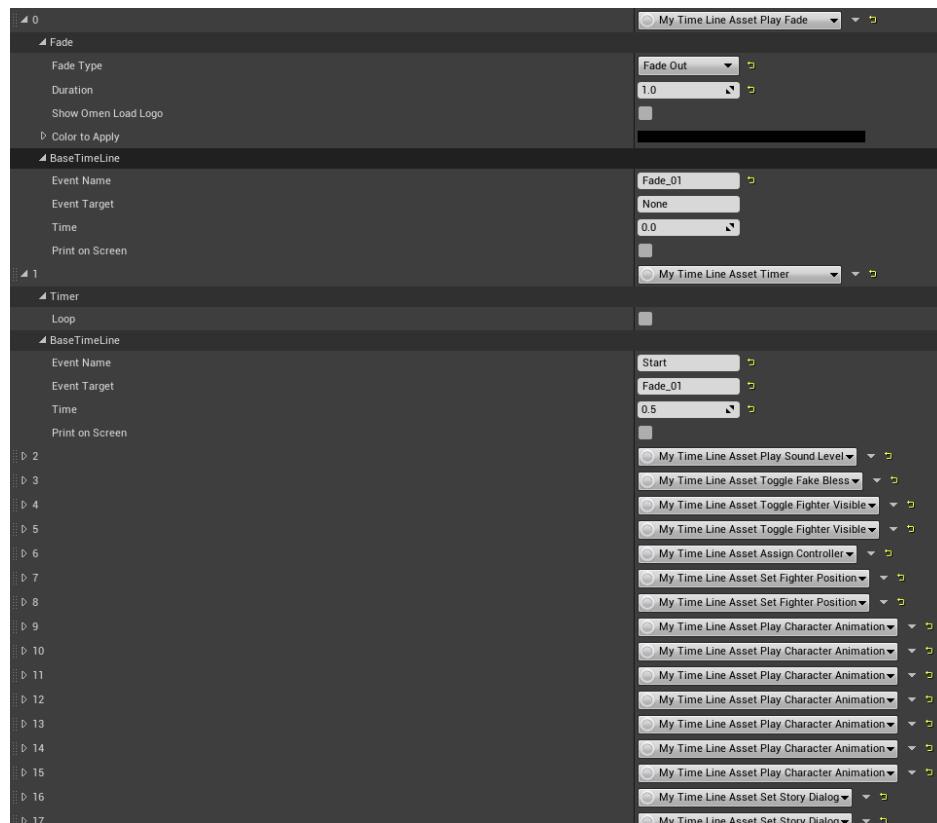


Figura B.1: Interfaz para editar las timelines anteriores.



Figura B.2: Una timeline lineal.



Figura B.3: Una timeline con distintas ramas de ejecución.

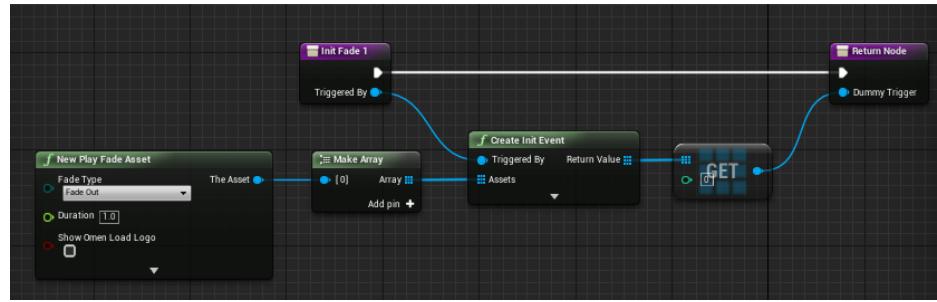


Figura B.4: Una función que define un evento simple.

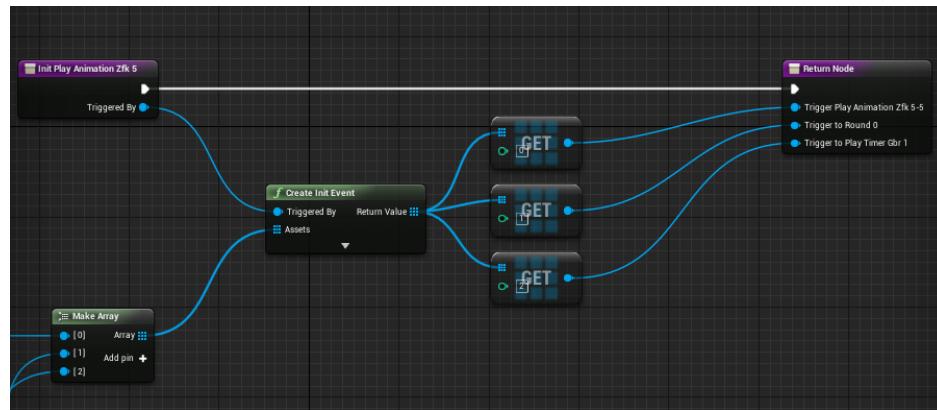


Figura B.5: Una función que define un evento complejo, con más de una salida.

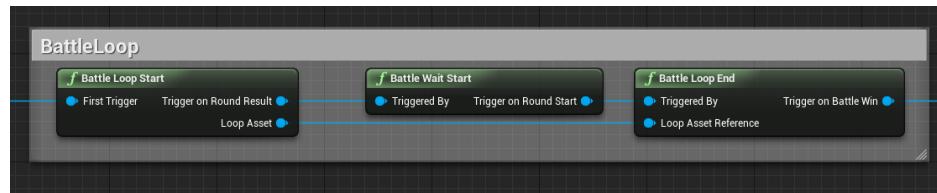


Figura B.6: Tres eventos que definen un loop para hacer una pelea mejor de N.

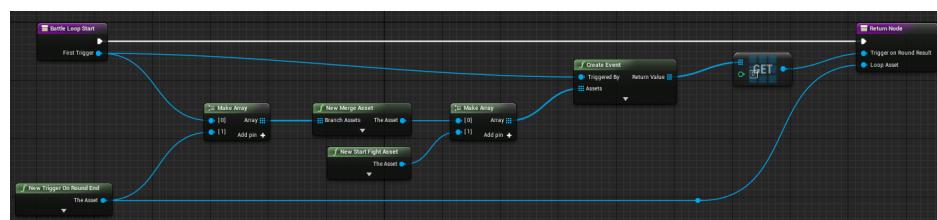


Figura B.7: Definición del primer evento de un mejor de N.

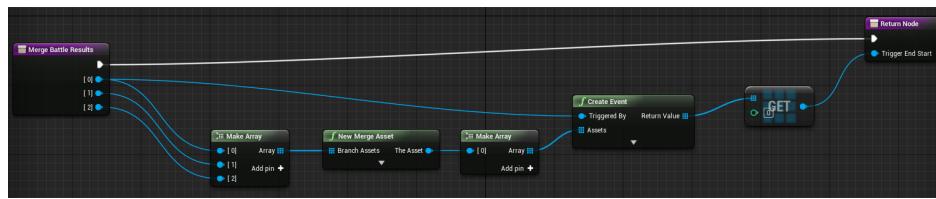


Figura B.8: Definición del segundo evento de un mejor de N .

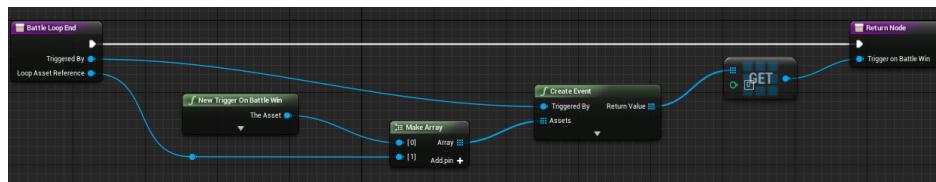


Figura B.9: Definición del último evento de un mejor de N .

B.1. Documentación del sistema implementado

Documentación Modo Historia

Daniel Soto

Enero 2019

1 Introducción

La nueva implementación del modo historia está basada en blueprints, para mejorar la visualización de los eventos llamados, y cuándo son llamados relativo a los otros eventos. Esta implementación organiza las timelines anteriores en eventos y assets. Cuando un evento es llamado, éste llama a todos sus assets al mismo tiempo, hasta que se completen todos. Para definir cuándo se llama un evento, se debe asignar un asset de otro evento como un trigger para el siguiente evento. Hay un tipo de assets especiales, llamados triggers, los que pausan la ejecución del timeline hasta que se cumple cierta condición.

2 Configuración

Para comenzar a crear un nivel con esta nueva implementación, se debe crear un blueprint nuevo a partir de la clase *MyStoryBattleConfigAsset*. Los class defaults se deben configurar del mismo modo que para la implementación anterior, con las siguientes diferencias:

- Los campos en StoryBattleConfig → Timelines se dejan en blanco.
- Se debe marcar el valor Use Event Handlers, en StoryBattleConfig → Blueprint como verdadero.

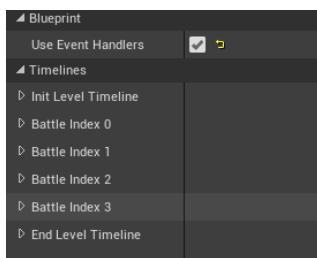


Figura 1: Parte de la configuración que cambió en esta implementación nueva.

En el blueprint se debe implementar el evento *Init* para configurar el nivel. El nivel pasa por tres estados. El comienzo, las peleas y el final. Naturalmente, el primer estado es el comienzo. Para avanzar a las peleas, se debe configurar la batalla y la ronda que toca a continuación, y luego llamar al evento *End*. Para avanzar hasta el final, se debe llamar el evento *Stop Battles* antes de que se termine la última pelea. En el final, al llamar *End* se termina la ejecución del timeline y se avanza al siguiente nivel.

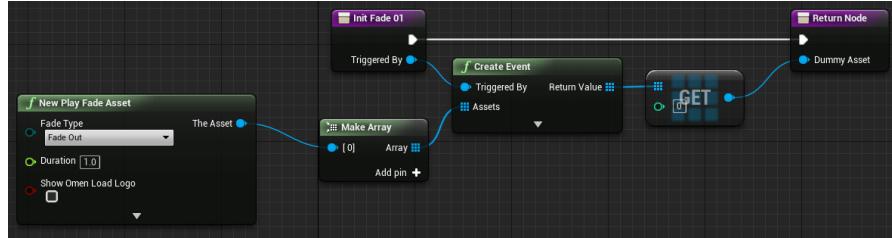


Figura 2: Un evento simple, que no triggereea otros eventos.

3 Creación de Eventos

Para cada evento se debe crear una nueva función, con **un asset de input** que actuará como trigger de este evento¹ y **por lo menos un asset de output**, que actuará como trigger de otro evento². Los inputs y los outputs son todos del tipo *MyTimelineAsset*. **Es importante marcar la función como Pure en el editor.**

Luego se crean los assets del evento usando alguna de las funciones ubicadas en StoryTimelines → Assets. Luego de crear los assets deseados para el evento. Se deben agrupar todos los assets en el mismo arreglo, y entregar ese arreglo a la función *Create Event*. El input de la función que se está creando debe ser entregado a la función de creación del evento mediante el parámetro *Triggered By*. **Si esto no se hace, se corre el riesgo de que ningún evento de más adelante sea llamado, o de que ningún evento anterior sea llamado.**

La función de creación del evento retorna el mismo arreglo de assets que recibe. De este arreglo se deben obtener los assets que actuarán como triggers para los siguientes eventos.

En la figura 2 se puede ver cómo se define un evento simple, con un único asset, que además no triggereea otros eventos. En la figura 3 se define un evento más complejo, con tres assets que triggereean eventos distintos.

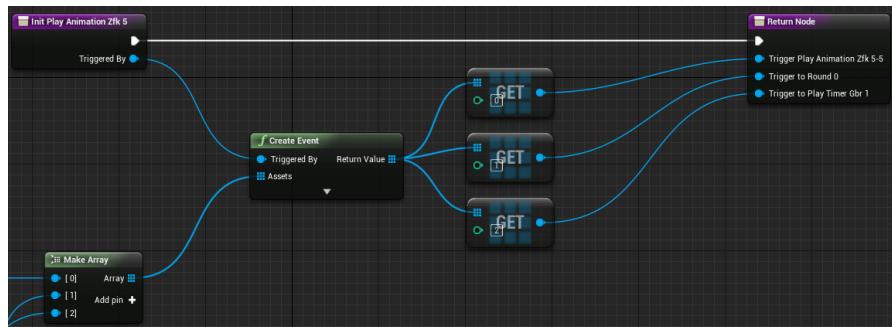


Figura 3: Un evento más complejo, que triggereea tres eventos distintos.

¹Excepto para el primer evento llamado, donde no es necesario que hayan inputs

²Si ningún asset del evento actúa como trigger para otro evento, seleccionar un asset cualquiera como output de la función

4 Conexión de eventos

Para configurar el flujo de los eventos se deben conectar todos las funciones donde se definen eventos, conectando al lado izquierdo el asset que triggere a un evento dado, y el asset de la derecha al evento triggereado por aquél asset (Si es que triggere a algún evento). Luego de conectar todos los eventos, se deben tomar los assets finales de cada rama, y hacer un arreglo con todos ellos. **Si alguna rama no se conecta al arreglo del final, es como si no existiese para el programa.** En las figuras 4 y 5 se ven ejemplos de cómo se conectan los eventos.



Figura 4: Serie de eventos sin ramas. Notar que se igual se forma un arreglo con el evento final.



Figura 5: Serie de eventos con dos ramas. El evento final de cada rama se agrega al arreglo.

5 Flujo de los eventos

La timeline creada en los blueprints comienza su ejecución desde el nodo que no es triggereado por nadie (el de más a la izquierda). Hay un par de factores importantes que se deben configurar en la organización de los eventos, para que el sistema sepa que acción es la que se debe tomar a continuación.

5.1 Pausas en la timeline

Para el funcionamiento correcto de la timeline, se deben utilizar assets que frenen la ejecución de una rama hasta que se cumpla cierta condición. En particular, la condición más importante se encuentra implementada en el evento ***Trigger On Round Win***. Esta condición espera a que la pelea entregada termine su ejecución. Este evento debe aparecer después de cada llamada a *Start Fight*, y antes de la primera llamada de la siguiente batalla. **Si no se incluye, entonces los eventos que siguen a una pelea se ejecutarán durante la pelea.**

5.2 Configuración de las batallas

En el caso de las batallas, se deben usar los assets *Config Battle* y *Config Round* para ajustar que tipo de batalla y que tipo de ronda será la que viene a continuación (se puede omitir *Config Battle* si es que fue configurado anteriormente, y la ronda que viene a continuación es parte de la misma batalla). Estos assets **deben ser configurados antes de que se complete el llamado de *Trigger On Round End* de la ronda anterior**. Si no se configuran antes de que termine la ronda anterior, en la próxima batalla se utilizarán las configuraciones de la ronda o batalla anterior. En general, se recomienda configurarlo en el mismo evento donde se llama a *Trigger On Round End*. Para el caso de la primera batalla, se debe hacer la configuración de esta antes de la llamada de *End* en el comienzo de la timeline.

5.3 Última batalla

Para que el timeline pase al estado final correctamente **se debe ejecutar Stop Battles antes de que la última batalla termine**. Si no se hace, entonces el timeline esperará una llamada *Start Fight*, en vez de una *End*, y el juego quedará *soft-locked*.

Figura 6: Implementación de un *Best of 3* con ramas distintas para cada resultado.

Figura 7: Merge de las distintas ramas de la figura 6.

5.4 Batallas con flujos especiales

Para niveles donde se pelean batallas del tipo *Best of N*, donde el jugador puede perder una o más partidas sin reiniciar el nivel, se debe especificar el flujo de las rondas en el blueprint. Hay un par de maneras posibles de hacer esto. Los ejemplos mostrados son de la implementación del nivel 32 con este sistema.

5.4.1 Ramas para cada resultado

Esta estrategia simplemente espera al resultado de una ronda utilizando *Trigger On Round Win* y *Trigger On Round Lose* para crear distintas ramas en la timeline, dependiente del resultado obtenido en cada ronda. Esta implementación permite tener eventos personalizados después de cada ronda. Es importante recordar terminar las batallas con *Stop Battles*. Como cada ronda en el *BoN* es parte de la misma batalla, es válido llamar *Stop Battles* una vez antes de que comience la batalla.

Al comenzar una batalla, se debe usar como trigger para el siguiente evento a un *Trigger On Round Start*. Si no se hace, entonces el timeline se salta los eventos que vienen a continuación, pensando que el resultado actual es válido para las siguientes rondas.

Para controlar el flujo de la ejecución se usa un evento que tiene dos assets, *Trigger On Round Win* y *Trigger On Round Lose*. Cuando se cumple el resultado de uno, se continúa por su rama y se detiene la rama del otro. En casos donde al perder se perdería la batalla se usa un evento que sólo tiene a *Trigger On Round Win* como asset.

Al salir de la batalla es importante tener un evento con un *Merge Asset*, el cual recibe todas las ramas creadas durante la batalla. Luego de este evento se puede continuar con el resto del nivel. Esto se debe hacer para que cualquiera de las ramas continúe por el mismo camino más adelante.

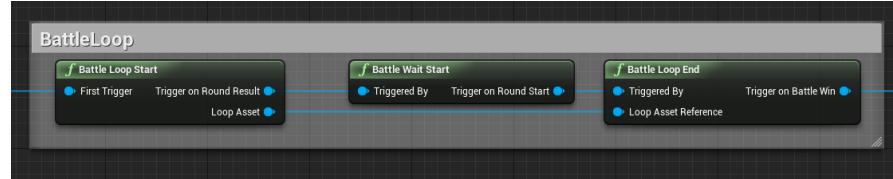
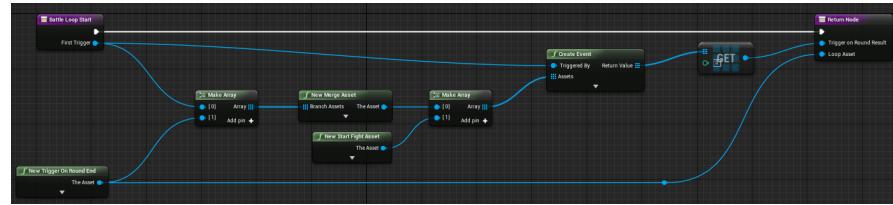
5.4.2 Loop hasta ganar

Esta estrategia se ve más limpia, pero no permite eventos personalizados entre rondas³. Esta versión está compuesta por tres eventos, donde el primero y el último son distintos a los usuales.

Para el último evento, el asset que triggere el loop es un *Trigger On Round End* y es recibido como un argumento. Además no se usa como salida. **Este asset es creado en el primer evento** (Figura 10).

En el primer evento, el asset recibido por *Triggered By* se entrega como argumento para un *Merge Assets*, junto al *Trigger On Round End*. Las salidas de este evento son el *Merge Assets* creado y el *Trigger On Round*

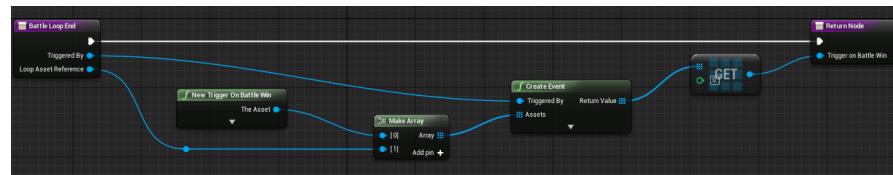
³Puede repetirse el mismo evento entre todas las rondas, pero se complica mucho la implementación, por lo que se recomienda hacer ramas para usar eventos personalizados entre las rondas

Figura 8: Implementación de un *Best of 3* con un loop.Figura 9: Primer evento del loop del *Best of 3*.

Finalmente, el evento intermedio es sólo un evento que espera al comienzo de la pelea con *Trigger On Round Start*.

5.5 Pausas implementadas

- *Trigger On Battle Win*: Continúa la ejecución de la rama si es que la batalla actual se gana. Si se pierde, se detiene la ejecución de la rama.
- *Trigger On Fighter HP*: Recibe el fighter que se quiere observar, y un número entre 0 y 1. Continúa cuando el fighter especificado alcanza un porcentaje de vida menor o igual al número entregado.
- *Trigger On Round End*: Continúa la ejecución de la rama si es que la ronda actual se termina. Si la ronda se ganó, y en consecuencia se termina la batalla, entonces no se continúa y se detiene la ejecución de la rama.
- *Trigger On Round Lose*: Continúa la ejecución de la rama si es que la ronda actual se pierde. Si la ronda se ganó, se cancela la ejecución de esta rama.
- *Trigger On Round Win*: Continúa la ejecución de la rama si es que la ronda actual se gana. Si la ronda se perdió, se cancela la ejecución de esta rama. Por defecto también se activa si es que la batalla se gana, pero tiene un parámetro avanzado que permite ignorar este caso.

Figura 10: Último evento del loop del *Best of 3*.

- *Trigger On Round Start:* Continúa la ejecución de la rama cuando el jugador consigue el control del fighter en la ronda actual.

6 Guardado de los eventos

Guardar los eventos es muy simple, basta con entregar el arreglo resultante de la conexión de los eventos a la función *Set Events*.

7 Configuraciones para testeo

7.1 Eventos

Para testear y poder debuggear el flujo de un nivel, todas las funciones de crear evento tienen 2 argumentos avanzados, uno es el nombre del evento, y otro es un flag para imprimirlo en pantalla al ser llamado, y cuando todos sus assets terminan su ejecución.

7.2 Assets

Además, todos los assets tienen en sus parámetros avanzados dos argumentos análogos, que hacen que se imprima en pantalla un mensaje cuando se comienza a ejecutar el asset, y cuando se termina de ejecutar el asset.