Simonéame

Memoria de la creación de un Simón 3D en Unreal Engine

Alicia Guardeño Albertos

28 de febrero de 2018

Introducción

Simonéame es un clon con gráficos 3D del juego clásico llamado Simón, hecho en Unreal Engine 4. Contiene un único modo de juego que es el clásico de añadir un elemento a memorizar más por cada ronda de secuencias acertadas. Como añadidos extras, se ha incorporado un pequeño contador de las cinco mejores puntuaciones que el jugador ha obtenido en el juego.

Para jugar, el jugador tiene esperar a que el juego muestre visualmente la secuencia y una vez finalizada, se podrán seleccionar los bloques del Simón con el ratón. Al ser seleccionado, se reproduce el color y sonido de dicho bloque y posteriormente puede suceder una de estos casos:

- Si el bloque seleccionado es correcto y además era el último a adivinar de la secuencia, se gana la ronda y comienza otra ronda con un bloque más a memorizar.
- Si el bloque seleccionado es incorrecto, se pierde el juego y después sale la pantalla de final de partida y se guarda la puntuación obtenida (si es mejor que alguna de las guardadas anteriormente). El jugador ahí puede seleccionar si volver a empezar una nueva partida o al menú inicial.

Luego en lo que respecta a desarrollo, la mayoría de las decisiones tomadas han sido teniendo en cuenta sobretodo dos factores:

- **Ser fiel al material del curso:** Siempre que ha sido posible, he intentado reutilizar los ejemplos que hemos dado en el curso para desarrollar diversas partes del juego aunque se haya sacrificado algo de legibilidad para ello.
- **Reducida disponibilidad de horas de trabajo:** A pesar de quedar claro desde el principio que había un mes aproximadamente para realizar el juego, debido a diversos factores como terminar la entrega anterior, problemas de salud y otras de índole personal, el tiempo estimado para trabajar se acabó reduciendo a 10 días.

Dicho todo esto, se procede a hablar del desarrollo en cuestión más detalladamente.

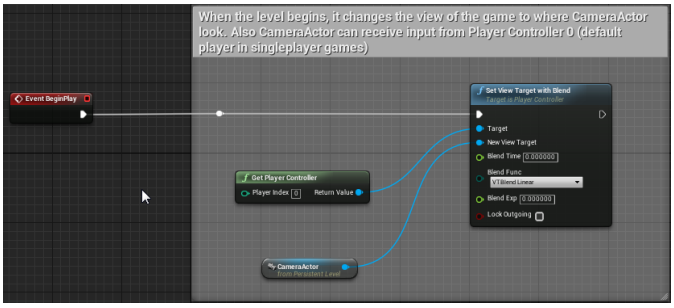
Primeros pasos

En la creación del proyecto nuevo en Unreal Engine 4, se decidió el uso de una plantilla de C++ vacía para evitar contenidos y configuraciones superfluas. No obstante, varios elementos básicos del juego fueron asimilados de la plantilla de puzzles de C++ de Unreal Engine 4:

- Se reutilizaron el `GameMode`, el `Pawn` y el `PlayerController` para generar `SimoneameGameModeBase`, `MouseInputHandlerPawn` y `SimoneamePlayerController` respectivamente. Esto es para permitir al juego recibir input del ratón del jugador sin la necesidad de un `PlayerCharacter` en el juego.

- Los métodos `BlockClicked` y `OnFingerPressedBlock` en la clase `SimonBlock`. Más adelante hablaremos de ella.

Aparte de esto, en todos los `LevelBlueprint` de los mapas, se ha añadido el siguiente código en Blueprints.



Esto es porque al no existir ningún `PlayerCharacter`, el juego no sabe cuál cámara elegir para mostrar el juego y por defecto pone una cámara libre. Como lo que queremos es tener la cámara fija, haciendo esto le decimos al mapa cuál usar.

Lógica del juego

Una vez fijada la cámara y que el juego permitiese usar el ratón sin problemas, procedí a empezar con la lógica del juego.

Lo primero fue crear un tipo enumerado de la clase `UEnum` llamado `SimonBlockColorType` con los colores a haber en el juego. Así después al crear los actores clase `SimonBlock` se podían asignar que tipo de color tiene cada bloque, esto también sirve para luego posteriormente inicializar el color de material y el sonido específicos de cada uno.

Luego tenemos los métodos `BlockClicked` y `OnFingerPressedBlock` que nos sirven para detectar si han sido clickeados por el jugador, dentro de esta varios métodos evalúan si ese click es válido dependiendo de si se está mostrando una secuencia nueva en el simón, se ha ganado o perdido, o de si otro bloque está siendo iluminado. En el caso de que si se pueda, el bloque se ilumina y reproduce su sonido, el tiempo que está iluminado depende de la duración del sonido y se controla mediante un booleano y el método `Tick`. Cuando se termina de iluminar, el booleano que lo controla se vuelve al estado `false` y chequea si, en el caso de que el juego esté esperando input por parte del jugador, le pasa a la clase `SimonManager` el color de dicho bloque.

A continuación, programé la clase `SimonManager`. Es una clase compleja que se encarga tanto de la lógica y reproducción visual del simón, además de cómo gestionar el estado de las rondas y el juego. En resumidas cuentas, sus tareas se desglosan en:

- **Generar las secuencias del simón:** Al inicio del juego, crea una lista `SimonSequence` a la que se le añade una cantidad específica de elementos del tipo `SimonBlockColorType` dependiendo del valor de `StartingSequenceLength` (configurable desde el editor de Unreal Engine 4). Luego por cada ronda de simón ganada por el jugador, añadirá un elemento más a esta lista.

- **Reproducir visualmente las secuencias del simón:** Por cada ronda, se genera una cola `DisplayingSimonSequence` a partir de los elementos de la lista `SimonSequence`. Luego coge el elemento en cabeza de la cola y dependiendo del tipo, le indica a un `SimonBlock` concreto que reproduzca su color y sonido. Una vez termina el `SimonBlock`, este avisa a `SimonManager` de que coja el siguiente elemento de la cola y así sucesivamente hasta que la cola se quede vacía. Cuando ocurre eso,

- **Gestionar el input del jugador:** Una vez termina de reproducirse visualmente el simón, `SimonManager` avisa a todos los `SimonBlock` que pueden ser pulsados con el ratón por el jugador. Cuando uno es pulsado, se reproduce visualmente y le pasa su color a `SimonManager` para evaluar si es idéntico con el elemento en cabeza de la cola `EvaluatePlayerInputSimonSequence` (creada de la misma manera que la cola `DisplayingSimonSequence`). Si el valor es el mismo, el jugador puede pulsar otro `SimonBlock` y así hasta que se vacíe la cola o falle el jugador y pase un elemento que no sea igual al de la cabeza de la cola.

- **Gestión del estado de la partida:** Una ronda puede acabar de dos diferentes maneras, en ambos casos `SimonManager` le hará saber al jugador cuando ocurre eso mediante la reproducción visual y sonora de los `SimonBlock`. Además de eso, dependiendo de si el jugador ha ganado o no:

  - Si el jugador acierta toda la secuencia, `SimonManager` le indicará a `ScoreCountManager` que actualice el marcador de rondas ganadas.
  - Si el jugador falla al introducir un elemento de la secuencia, `SimonManager` creará dinámicamente el *widget* de Game Over (final de partida) donde el jugador podrá elegir si desea volver a empezar a jugar al simón o al menú de inicio. También se llama a `SaveGameManager` para que recoja la puntuación obtenida y, si está entre las cinco mejores, la guarda para ser mostrada en el mapa **Scoreboard**.

Sistema de guardado de puntuaciones

Heredando de la clase `SaveGame`, se creó una clase `SimoneameSaveGame` encargada de guardar en un atributo público llamado `Scores` de tipo lista las cinco mejores puntuaciones conseguidas en el juego. Luego para gestionar correctamente los datos de `SimoneameSaveGame`, se creó otra clase llamada `SaveGameManager`.

`SaveGameManager` es la encargada de crear un *SaveSlot*(fichero de partida guardada) de tipo `SimoneameSaveGame`, si no existía previamente, e inicializa todos los valores de `Scores` a cero. Luego para comunicarse con las demás clases están los métodos `LoadScores` y `SaveScore`.

`LoadScores` es trivial, devuelve el valor de `Scores` del *SaveSlot* actual.

`SaveScore` es algo más compleja. Mediante el uso de una lista auxiliar, cogemos la nueva puntuación, las ordenamos en la lista y luego mediante un bucle `for` los asignamos

uno a uno en el atributo `Scores` del *SaveSlot* actual.

Esta última decisión fue la escogida debido a que si se intenta reasignar los valores de manera dinámica, producen un error de violación de acceso.

Manejo dinámico de elementos de la interfaz

La interfaz es estática en la mayoría de los mapas del juego, y por lo tanto sólo contienen pequeños scripts básicos en Blueprintsque permite a los menús cambiar de mapa. Pero en el mapa **Game**, y de manera parcial en **Scoreboard** no.

En **Game** tenemos en la interfaz un contador de veces que hemos acertado la secuencia del simón, esto es gestionado mediante el actor de la clase `ScoreCountManager`. Este instancia el widget que contiene los gráficos del contador que se va actualizando cuando es llamado por la clase `SimonManager`.

Y en el mapa **Scoreboard**, tenemos el actor de clase `ScoreboardManager`. Cuando carga el mapa, este actor crea dinámicamente widgets anidados en el panel de la interfaz de este mapa con la información de las puntuaciones obtenidas con la clase `SaveGameManager`.

Conclusiones

En general se ha conseguido llegar a desarrollar los requisitos establecidos con un diseño relativamente claro en el reducido tiempo disponible. Sin embargo, siempre hay aspectos que son mejorables, aunque los más a destacar son:

- Usar la clase `FTimer` en vez de booleanos y el método `Tick` para controlar los tiempos en la clase `SimonManager`.
- Separar en dos la clase `SimonManager` de manera que una se dedicara exclusivamente a la lógica del juego y otra para controlar el aspecto gráfico y sonoro.

Anexo

Diagrama de la arquitectura resultante