MEMORIA

Práctica 2: Clon de OneLine

Autores

Adrián Alcántara Delgado
Daniel Quintero Bernal

Estructura del proyecto

Ficheros

Desde el punto de vista de ficheros, el proyecto se organiza de la siguiente manera.
Dentro del directorio **Assets** encontramos distintas carpetas con los recursos empleados en el desarrollo del videojuego.

- **Editor**: Assets de UnityAds para el editor de Unity.

- **Niveles:** Contiene un *json* con todos los mapas que vamos a utilizar, en formato fijo. Sería posible añadir nuevos mapas siempre que se respete la sintáxis.

- **Plugins:** Plugins del proyecto.
    - Android e iOS: Plugins de UnityAds necesarios para la visualización de anuncios
    - SimpleJSON: Plugin para la lectura de ficheros *json*. Lo usamos para leer arrays de pares de valores, ya que los parsers que proporciona Unity no nos dan esa funcionalidad.

- **Prefab**: Prefabs del proyecto. Tomamos como prefabs elementos que se van a replicar a lo largo del videojuego: tiles, el cursor, y las marcas en el camino.

- **Scenes:** Escenas del proyecto. Disponemos de tres escenas.
    - **Title**: Título de la escena, desde ella se puede acceder a la selección de niveles o al nivel challenge.
    - **Selección Nive**l: Desde esta escena el jugador puede lanzar un nivel que tenga desbloqueado, o puede regresar al menú.
    - **Juego**: Se comparte la escena de juego entre un nivel normal y un nivel challenge, debido a que son pocos los cambios entre las mismas: principalmente son cambios en la interfaz. Con un parámetro de GameManager sabemos distinguir si el jugador está jugando un nivel de desafío o uno normal y hacer los cambios pertinentes.

- **Scripts:** Contiene todo el código que usamos en el juego. Está estructurada de manera que separamos los scripts “por escenas” y dentro de los mismos por utilidad.
    - **Juego:** Aquí se encuentran los scripts de la escena de juego.
        - Gameplay: Scripts que definen comportamientos de gameplay, como la clase Tile
        - Managers: Contiene todos los managers que usamos en el juego.
        - Menú: Clases encargadas de gestionar el canvas de cada escena. Además, tiene un script que obtiene valores necesarios para escalar el juego en función del Canvas, y tiene el código encargado de lanzar anuncios recompensados.
    - **Menú:** Scripts encargadas del Canvas en las demás escenas, la de título y selección de nivel. Entre sus clases, encontramos la encargada de todos los callbacks del título o la encargada de “poblar” el grid de niveles en la escena de selección de nivel, marcando adecuadamente los que haya jugado el usuario.

- **Sprites**: Elementos gráficos del juego. En esta carpeta encontramos los sprites que utilizan ordenados por carpetas. Además de los sprites, encontramos los *scriptable objects* de las **skins del juego**, constituidas por sprites.
    - **Gameplay**: Dentro de esta carpeta encontramos sprites que se usan dentro de la escena de juego. Aquí encontramos las skins, así como los elementos de interfaz separados de los elementos propios del gameplay.
    - **Menú:** En esta carpeta encontramos los assets que hemos usado para las distintas escenas de menú, especialmente la escena principal.

- **UnityAds**: Dentro de esta carpeta encontramos las *dll* de UnityAds, encargadas de mostrar los anuncios con los que recompensamos al jugador.

Clases

Se encuentran dentro del directorio scripts. Cada fichero contiene una única clase. El directorio de Scripts se divide en dos partes. Los cs del juego y los del menú.

- Menú:
    - **ActivarScroll**: Es una clase muy sencilla que se encarga de activar la función de scroll del menú de selección de niveles y de activar el componente encargado de rellenar el Viewport del Scroll View con los distintos niveles según la dificultad.
    - **BotonNivel**: Es una clase que va ligada a cada botón de nivel de la escena y se encarga de administrar qué nivel tiene que cargar al ser pulsado su botón. Tanto jugado como bloqueado.
    - **CanvasMenu**: es un “manager” del canvas de la escena Title. Administra todos los elementos del canvas de esa escena. Actualiza la información del jugador que se ve reflejada y también oculta o visualiza los elementos del botón de challenge.
    - **CanvasSeleccionNivel**: es una clase sencilla que administra los objetos UI de la escena de selección de niveles y se encarga de actualizar el texto que determina la dificultad de los niveles presentes.
    - **PopulateGrid**: Instancia y asigna el componente BotonNivel, comentado más arriba, a cada botón dentro del viewport de Scroll View en la escena.
    - **ScaleEffect**: Se encarga de darle una pequeña animación al regalo de Login diario.
- Juego:
    - GamePlay
        - **Cursor**: Representa en la escena la posición del cursor.
        - **Tile**: Administra la información del Tile. Qué skin tiene, si tiene pista, si tiene ya un camino dibujado, si ha sido pulsado.
        - **TileSkin**: ScriptableObject que define los atributos que tiene una skin del juego.
    - Managers
        - **TimeManager**: Clase que se encarga de controlar el tiempo del juego para que pueda seguir funcionando con la aplicación cerrada o en segundo plano.
        - **ProgressManager**: Es la clase que serializa los datos del jugador y los carga y guarda.
        - **LectorNivel**: De un archivo json obtenemos todos los niveles del juego y los guardamos en un diccionario.
        - InputManager: Guarda la información del cursor en la pantalla al hacer touch o click
        - **GameManager**: Gestiona los elementos esenciales del juego. Define cualquier tipo de recompensa en el juego, el progreso del juego, el guardado de los datos del juego. El número de niveles por dificultad. Información del nivel actual.

        - **DatosJugador**: Clase contenedora de los datos serializables. Monedas, medallas de challenge, niveles disponibles, el tiempo restante de challenge y de recompensa diaria.
        - **BoardManager**: Administra la información del tablero, cómo se tiene que escalar la cámara según la plantilla utilizada. El camino de la pista, movimientos posibles, deshacer el camino cuando no es el correcto, etc.
    - Menú:
        - **Timer**: hace la cuenta atrás del tiempo que haya sido asignado.
        - **SeePresent**: lógica para mostrar u ocultar el regalo de Login diario cuando sea necesario.
        - **RewardedAdButton**: Lanza un anuncio al ser pulsado el botón que debe tener asociado. Cuando el anuncio ha sido visualizado completamente, se llamará al método delegado que debe haber sido previamente asignado.
        - **GameScale**: Obtiene el espacio de pantalla disponible para el tablero. Restando los recuadros verticales. Calcula el margen lateral del tablero.
        - **CanPlayChallenge**: Tiene una funcionalidad parecida al de SeePresent pero para el tiempo de espera hasta volver a jugar a un challenge.
        - **CanvasJuego**: Administra lo que aparece en la escena durante el nivel y lo que aparece una vez acabado. Sea o no un nivel de tipo challenge.
        - **CountDown**: Contabiliza el tiempo posible de juego para el challenge.

Requisitos de implementación

- Adaptar correctamente la visualización en pantalla en función de la relación de aspecto (ancho-alto) del dispositivo
  Esto se hace desde el **BoardManager** con la información proporcionada por el **GameScale**. Sabiendo lo que ocupa un tile y cuantos Tiles queremos meter a lo alto en el espacio disponible para el tablero, siempre respetando margen lateral, somos capaces de corregir el orthographicSize de la cámara.

- Leer los mapas de ficheros de texto:
  Leemos todos los datos de cada nivel en un único Json una vez al inicio de la aplicación en **LectorNiveles.** Cuando se requiera la información de un nivel, se le pedirá a esta clase con un índice específico.

- Disponer de varias pieles como datos, de modo que añadir nuevas pieles no suponga cambiar código
  Mediante el uso de **scriptable objects,** tenemos un sistema en el que es sencillo crear desde el editor nuevas "pieles" para los tiles. El **BoardManager** escoge una piel aleatoria de una lista, modificable desde el editor, y la mantiene a lo largo de toda la partida.

- Mantener el progreso del usuario
  Los datos del jugador se cargan, o se crean, al inicio de la aplicación en **GameManager**. Todos los cambios que se realizan se hacen de forma "local" en un atributo privado, y sólo cuando sea necesario se guardará. El único capaz de cargar y guardar una partida es **GameManager** a través de la clase estática **Progress Manager**. Serializamos los datos del jugador y calculamos la hash. Esa hash se guarda con los datos en una ruta persistente en el dispositivo destino.
  Profundizando en el **encriptado:**
  Nuestra hash se calcula serializando el objeto y hallando la hash del mismo. La serialización se descarta, y esa hash se guarda en el objeto con los datos del jugador. Se serializa de nuevo, y se guarda en la ruta persistente. Además tenemos una ***sal***, guardada en el objeto, que calculamos sumando a un número base el número de monedas que tenga el jugador. De esta manera, si ese número no fuera el mismo al cargar los datos sabríamos que se ha modificado desde fuera y no se pueda predecir que ese número son las monedas sin más. Además, la hash del objeto también debe coincidir al deserializar, haciendo el proceso inverso.

- Mostrar anuncios al usuario
  Mediante la clase **RewardedAdsButton** tenemos botones que son capaces de lanzar un anuncio. Si ese anuncio se ha visualizado completamente, esta clase se encargará de notificar a un método delegado que contendrá la recompensa adecuada en función del tipo de botón. Es decir, un botón de duplicar tendrá que definir un *callback* que duplique una

cierta cantidad, y debe ser encargado de notificar a su componente **rewaredAdsButton** del callback al que quiere llamar. En caso que el usuario no visualice el anuncio en su totalidad, no habrá consecuencias y tendrá la opción alternativa de pagar, u obtener una recompensa sin duplicarla. De esta manera, no pierde la oportunidad de ganar algo por haber saltado el anuncio.
