Memoria Práctica 1 Videojuegos en Dispositivos Móviles

Pablo Martín García, Aaron Reboredo Vázquez.

En este documento vamos a explicar la organización de clases que hemos seguido para estructurar nuestro proyecto de forma que pueda ejecutarse tanto en dispositivos móviles como ordenadores.
De la misma manera también explicaremos algunas decisiones que hemos tomado a la hora de implementar nuevas clases.

La estructura general del proyecto se divide en 6 módulos:

Módulos AndroidGame y PCGame:

Contiene las respectivas clases Main que nos permiten lanzar el juego tanto en plataforma Android como en la plataforma PC.

Módulo Engine:

Define las **interfaces** básicas que van a usar los respectivos motores de Android y PC. Para implementar estas interfaces hemos seguido las recomendaciones dadas en el guion de la práctica y en clase. Las 4 interfaces son las siguientes:

- Image: contiene una imagen como mapa de bits.
- Graphics: proporcionará funcionalidades gráficas para renderizar imágenes en pantalla.
- Input: será la encargada de recoger la entrada del usuario.
- Game: será el encargado de mantener las instancias de Graphics e Input principalmente, para que puedan ser usados por otras clases.

En este módulo también hemos creado unas clases que nos serán de gran utilidad para optimizar y facilitar la programación del proyecto, las hemos incluido en un directorio llamada **Utils**, y son las siguientes:

- Pool: esta clase nos aporta la funcionalidad de un pooler, permitiéndonos reutilizar objetos en vez de instanciarlos y destruirlos. La usaremos para reutilizar los eventos del input del jugador.
- Rect: esta clase tiene la función de ofrecer los atributos básicos de un rectángulo, como son el Top, Bottom, Right, Left, Width y Height, permitiéndonos acceder a ellos con total libertad.

- Sprite: esta clase nos permitirá trabajar en un futuro con instancias de tipo Sprite, lo que nos ayudará a la hora de programar, ya que contienen una imagen y un srcRect propios. Además tiene implementado los métodos para pintarse en pantalla directamente.

Por último, tenemos tres **clases abstractas** definidas, que nos ayudarán sobre todo con el reescalado de la aplicación:

- AbstractGraphics: Clase intermedia entre la interfaz y las clases Graphics correspondientes a cada plataforma. Contiene la lógica necesaria para reescalar las imágenes en función del tamaño de la pantalla y nos permite traducir las dimensiones y coordenadas lógicas a las dimensiones y coordenadas físicas correspondientes. Tiene dos métodos genéricos DrawImageResized y DrawImageResizedAlpha que nos permiten pintar en pantalla una imagen con unas dimensiones (lógicas) diferentes a las de la imagen original, con o sin transparencia respectivamente. Esos dos métodos utilizan a su vez un método genérico (resizedDest) que es el que hace el reescalado propiamente dicho y modifica el rectángulo destino, traduciéndose, como hemos mencionado, de coordenadas lógicas a físicas (en pantalla). El resto de los métodos facilitan o automatizan ciertas operaciones, como posicionar una imagen directamente en el centro de la pantalla y permitir el desplazamiento únicamente en vertical, por ejemplo, haciendo uso de los dos primeros métodos mencionados, en función de si queremos o no imagen con transparencias. Esto nos permite que siempre se mantenga la relación ancho alto de nuestro juego en todo momento independientemente de las dimensiones de la pantalla donde se esté mostrando.
  Así mismo contiene la lógica (métodos drawImageAsBottomRightBand y drawImageAsUpperLeftBand) necesaria para plasmar las bandas negras que servirán como embellecedor y que cubrirán toda la parte de la pantalla no útil, dejando visible solo lo correspondiente a nuestro tablero o pantalla lógica de juego.

- AbstractInput: Clase intermedia entre la interfaz input y las clases Input correspondientes a cada plataforma. Contiene la lógica necesaria para reescalar o traducir las coordenadas de cada uno de los eventos, de los valores recibidos en dimensiones físicas a su correspondiente en coordenadas lógicas. Para ello recogemos la lista de eventos que nos llegan desde el Input de cada una de las clases, modificamos los valores para que coincidan con su correspondiente en coordenadas lógicas y los vamos almacenando en otra lista de eventos que es la que devolvemos finalmente y que recibirán las clases Estado de juegos cuando llamen a la lista de eventos.

- State: esta clase define un estado de juego, definiendo métodos para actualizar, pausar, reanudar o renderizar dicho estado. Nos permitirá dividir el juego en diferentes estados con funcionalidades únicas. Aunque no los hemos usado, los métodos resume(), pause() y dispose() están declarados por si quisiéramos en un futuro implementar alguna funcionalidad para la que fueran necesarios. Nos permitirán interactuar directamente sobre los elementos plasmados en la pantalla (lógica) con exactitud.

Módulo Android Engine :

Redefine las Interfaces definidas en el módulo Engine y las adapta a la plataforma Android.

- Image: contiene una imagen como mapa de bits.
- Graphics: proporcionará funcionalidades gráficas para renderizar imágenes en pantalla. Contiene los métodos que permiten pintar o renderizar en pantalla las imágenes a partir de su Bitmap.
- Input: será la encargada de recoger la entrada del usuario.
- Game: será el encargado de mantener principalmente las instancias de Graphics e Input de la plataforma, para que puedan ser usados por otras clases.

Además, se define la interfaz TouchHandler que contiene los métodos básicos que serán usados en las clases que gestionan el input por pantalla táctil en la plataforma android.

Así mismo, define las clases propias para la gestión de Input en esta plataforma (MultiTouchHandler y SingleTouchHandler) y la que gestiona el ciclo de juego (MySurfaceView).

- MultiTouchHandler: Contiene los métodos y la lógica necesaria para gestionar eventos simultáneos de interacción con la pantalla y nos brinda información sobre qué evento se está recogiendo y el id del dedo que lo genera.
- SingleTouchHandler: Contiene los métodos y la lógica necesaria para gestionar eventos únicos de interacción con la pantalla.
- MySurfaceView: Hereda de runnable y contiene los métodos y la lógica necesaria para gestionar el bucle principal de la aplicación. Es el que gestiona el bucle principal de juego, lanzando el estado de juego seleccionado, actualizándolo y presentándolo. Así mismo lleva la información de tiempo del juego y nos permite lanzar los métodos de gestión de bucle de juego acorde con las velocidades del procesador de la plataforma.

Módulo PC Engine :

Redefine las Interfaces definidas en el módulo Engine y las adapta a la plataforma PC.

- Image: contiene una imagen como mapa de bits.
- Graphics: proporcionará funcionalidades gráficas para renderizar imágenes en pantalla. Contiene los métodos que permiten pintar o renderizar en pantalla las imágenes a partir de su Bitmap.
- Input: será la encargada de recoger la entrada del usuario.
- Game: será el encargado de mantener principalmente las instancias de Graphics e Input de la plataforma, para que puedan ser usados por otras clases.

Además, se define la interfaz TouchHandler que contiene los métodos básicos que serán usados en las clases que gestionan el input por pantalla táctil en la plataforma android.

Este módulo también define las clases propias para la gestión de Input en esta plataforma (PCMouseHandler) y la que gestiona el ciclo de juego (PCSurfaceView).

- PCMouseHandler: contiene los métodos y la lógica necesaria para gestionar los eventos del ratón. Crea el pooler de eventos al que estos se irán añadiendo según se produzcan.
- MySurfaceView: hereda de JFrame y contiene los métodos y la lógica necesaria para gestionar el bucle principal de la aplicación. Es el que gestiona el bucle principal de juego, lanzando el estado de juego seleccionado, actualizándolo y presentándolo. Así mismo lleva la información de tiempo del juego y nos permite lanzar los métodos de gestión de bucle de juego acorde con las velocidades del procesador de la plataforma.

Módulo Logic:

Este módulo es el encargado de gestionar toda la parte lógica del juego, por lo que va a ser común para ambas plataformas.

Contamos con un directorio de **Superclases** que contiene las clases base que hemos usado para hacer herencia ó bien clases con una funcionalidad muy particular. Son las siguientes:

- GameObject: clase que define los atributos y métodos básicos para cualquier gameObject, de ella irán heredando el resto de GameObjects que tendremos en nuestro juego. La idea es que cada GameObject tenga su propio método update() y present() para poder llamarlos después con ganeObject.update() o gameObject.present() desde cualquier estado y no estar preocupados de su lógica o renderizado.

- SwitchDashObject: clase que define atributos y métodos básicos para un GameObject perteneciente a SwitchDash. Implementa el atributo de color de cada objeto, ya que todos pueden ser blancos o negros. La idea era tener una clase que siguiese siendo general, pero esta vez para GamObjects del juego específico que estamos haciendo.

- GameManager: clase encargada de gestionar todo lo relacionado con la puntuación y las velocidades de los gameObjects. En ella se definen los puntos necesarios para aumentar la velocidad del juego y el aumento de velocidad que se va a aplicar a cada GameObject. También se encarga de guardar la velocidad y puntuación para poder disponer de ellas en la pantalla de GameOver.
- Assets: clase que contiene todas las imágenes, rectángulos y sprites que pueden ser usados en el juego.

- ParticleSystem: es la clase encargada de generar las instancias del objeto Particle, actualizarlas y pintarlas.

En el directorio **GameObjects** encontramos todos los gameObjects que se crearán en los estados del juego, tenemos las siguientes clases:

- Arrows: lleva la lógica de las flechas de fondo que van bajando, para producir el efecto de descenso constante vamos recolocando la imagen a su posición inicial cada vez que baja una determinada distancia, haciendo que coincida el corte de tal manera que no se observen efectos no deseados a la hora de imprimir la imagen en pantalla.

- BackgroundColor: se encarga de gestionar el color de fondo, permite salvar el color para poder repetirlo al pasar de la pantalla de menú principal a la de instrucciones.

- Ball: lleva la lógica de las pelotas, todo lo referente a generarlas con un color aleatorio sesgado y que vayan cayendo continuamente. Para realizar el aleatorio sesgado tenemos dos arrays de 10 colores, cada uno con 7 posiciones del color predominante (blanco o negro) y las 3 posiciones restantes con el color contrario, de manera que generamos un número aleatorio del 0 al 9 y escogemos el color en esa posición del array de negros predominante o blancos predominantes, basándonos en el color de la bola anterior.

- BlackBands: clase usada a modo de embellecedor. Se encarga de crear una capa de color plano a modo de bandas negras que cubren todo el espacio no útil o todo aquello que queda fuera del espacio de tablero o juego. Hace uso de los métodos correspondientes en la clase AbstractGraphics para pintarse acorde con el escalado de la pantalla.

- Button: contiene atributos y métodos generales que usaremos en los botones de juego, funciona como una clase base de la que heredará cada botón.

- ExitButton: tiene la lógica del botón para volver al menú principal cuando se pulsa.

- OptionsButton: tiene la lógica del botón para ir al menú de instrucciones cuando se pulsa.

- Particle: contiene los atributos y métodos para las partículas que generará el sistema de partículas.

- Player: contiene los atributos del player y su lógica, que puede resumirse en cambiar su color.

- PointsString: clase que permite dibujar en pantalla la palabra POINTS para mostrarla en el estado de Game Over.

- Score: contiene todo lo relacionado con la puntuación como la forma de posicionarla y el contador de puntos. Lleva la cuenta de la puntuación y se comunica con el GameManager para guardar las puntuaciones al final de la partida y poder plasmarla en la pantalla de fin de partida.

- SoundButton: tiene la lógica del botón para activar o desactivar el sonido del juego. Puesto que no tenemos sonido, su única funcionalidad es la de actualizar el sprite del botón cuando se pulsa.

- TapToPlay: contiene la lógica para pintar aumentando y disminuyendo el alpha la palabra Tap To Play, esto creará un efecto de parpadeo.

- WhiteFlash: clase encargada de la lógica del flash que aparece al cambiar de estado, que hacemos disminuyendo el alpha de la imagen.

En otro directorio **States** almacenamos los diferentes estados de juego que tendrá nuestra aplicación. Son los siguientes:

- GameOverState: se lanza cuando el jugador pierde, muestra los puntos obtenidos y da la opción de iniciar otra partida. Además, las flechas de fondo mantienen la velocidad que tenían en el estado de juego.

- GameState: el estado principal del juego. Contiene todos los gameobjects necesarios y su lógica para poder jugar una partida. Desde aquí se lleva la lógica de detección de colisiones (comprobando posiciones de imágenes) y fin de partida (salto al estado de Game Over bajo una condición determinada) así como la lógica de aumento de dificultad (velocidad de Arrows y Balls). Esto último lo hacemos comunicándonos con el GameManager para comprobar el número de puntos necesarios para aumento de velocidad y con las clases correspondientes a las que se le aplica el incremento.

- InstructionsState: se lanza tras pulsar en la pantalla desde el menú principal o accediendo desde el botón de opciones. Muestra en pantalla las instrucciones para jugar y mantiene el color de fondo que había en el menú principal.

- LoadingState: es el primer estado que se lanza al iniciar la aplicación. Su función es crear todas las instancias de imágenes, rectángulos y sprites para cada gameObject que pueda ser utilizado en el juego.

- MainMenuState: se lanza tras de forma automática tras el loading state, muestra un sprite con el nombre del juego y otro parpadeando que anima al jugador a pulsar la pantalla.