SWITCH DASH

NÉSTOR CABRERO MARTÍN

KELVIN COMPER DIAS

CONTENIDO

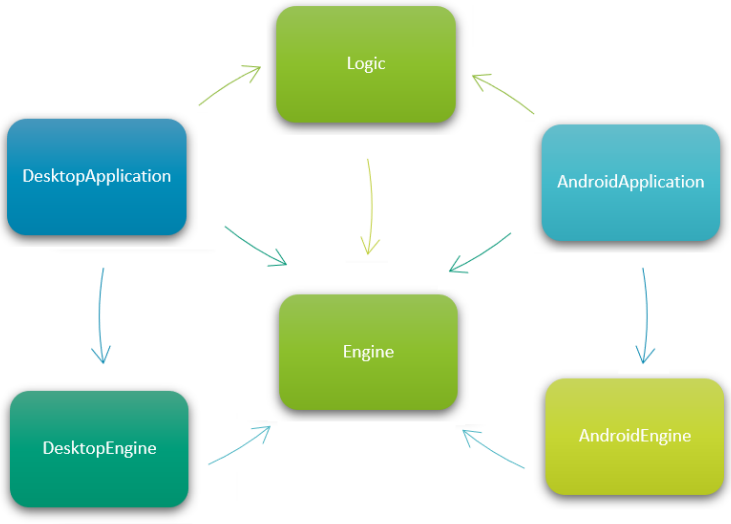
SWITCH DASH

En este documento de memoria de la primera práctica de la asignatura explicaremos de forma concisa y resumida la estructura general del proyecto de Android Studio, así como algunas de las funcionalidades más destacadas de nuestro *framework* y las extensiones opcionales implementadas.

El proyecto *Switch Dash 3.4* (por la versión de Android Studio sobre la que se ha desarrollado, v3.4.2) cuenta con seis **módulos**:

1. **Engine**: librería de Java que proporciona, a través de interfaces, la funcionalidad básica común a todas las implementaciones del motor. Cuenta con un paquete de utilidades multiplataforma.
2. **AndroidEngine**: librería de Android que implementa las interfaces del módulo Engine para dispositivos Android.
3. **DesktopEngine**: librería de Java que implementa las interfaces del módulo Engine para ordenadores.
4. **AndroidApplication**: aplicación de Android que lanza el juego en Android.
5. **DesktopApplication**: librería de Java que lanza el juego en ordenador.
6. **Logic**: librería de Java que contiene la lógica de nuestro juego, *Switch Dash*. Cuenta con un paquete de estados de juego y con un paquete de objetos de juego.

Las **dependencias** que se establecen entre los seis módulos del proyecto se resumen en este gráfico:



Los **paquetes** en los que se organiza el proyecto son:

- *es.ucm.vdm.engine*
  - *es.ucm.vdm.engine.android*
  - *es.ucm.vdm.engine.desktop*
  - *es.ucm.vdm.engine.utils*
- *es.ucm.vdm.app*
  - *es.ucm.vdm.app.android*
  - *es.ucm.vdm.app.desktop*
- *es.ucm.vdm.logic*
  - *es.ucm.vdm.logic.objects*
  - *es.ucm.vdm.logic.states*

PIXMAP

La interfaz Pixmap es un *wrapper* para objetos de tipo imagen. Podemos acceder a sus dimensiones y, si lo deseamos, liberar manualmente los recursos que utiliza de la plataforma.

ANDROID PIXMAP

La implementación para Android de Pixmap tiene como atributo un objeto de tipo Bitmap de la librería gráfica de Android. Podemos acceder al objeto Bitmap con un *getter*.

DESKTOP PIXMAP

La implementación para PC de Pixmap tiene como atributo un objeto de tipo Image de la librería AWT de Java. Podemos acceder al objeto Image con un *getter*.

GRAPHICS

La interfaz Graphics gestiona el apartado gráfico del motor. Nos permite crear nuevos objetos de la clase Pixmap, conocer las dimensiones de la superficie de renderizado y renderizar imágenes y formas a través de varios métodos.

SCALED GRAPHICS

Esta clase abstracta actúa como paso intermedio entre la interfaz Graphics y sus implementaciones en cada plataforma. Siguiendo el patrón de diseño *strategy*, ScaledGraphics nos permite redimensionar los rectángulos de destino de los objetos que vamos a pintar en función de la relación de aspecto que queramos mantener para nuestra superficie de renderizado lógica; los objetos de juego llaman al método de dibujado de ScaledGraphics que necesiten, e internamente ScaledGraphics aplicará el reescalado apropiado y llamará a métodos privados que implementarán las versiones de Android y PC. **Si queremos cambiar el tamaño lógico al que se reescalan todos los objetos del juego, debemos acudir a esta clase.** La clase cuenta con métodos públicos para reescalar rectángulos de forma individual o la superficie de renderizado al completo.

ANDROID GRAPHICS

La implementación para Android de Graphics, que hereda de ScaledGraphics. En su constructora recibe un *framebuffer* que utiliza para generar la superficie de renderizado de tipo Canvas. Además, utiliza tan solo dos rectángulos de Android (destino y fuente) durante toda la ejecución de la aplicación, agilizando el renderizado.

DESKTOP GRAPHICS

La implementación para PC de Graphics, que hereda de ScaledGraphics. En su constructora recibe un objeto JFrame que permite crear una estrategia de cambio de *buffer* adecuada; la clase cuenta con métodos específicos para acceder y modificar esta estrategia y el objeto Graphics de la librería AWT de Java asociado, ambos necesarios durante el bucle principal de PC.

GRAPHIC UTILS

La clase **Rect** representa un rectángulo genérico que podemos utilizar indistintamente en Android o PC. La clase **Sprite** encapsula un Pixmap y un Rect que hace las veces de rectángulo fuente, facilitando la gestión y renderizado de *sprite-sheets*. La clase **PixmapManager** es un *singleton* que nos permite gestionar objetos Pixmap de un modo sencillo, cargando las imágenes una sola vez al inicio del juego.

INPUT

La interfaz Input gestiona los eventos de entrada. Define dos clases internas: **KeyEvent** y **TouchEvent**. Estas dos clases generalizan los atributos de los eventos de entrada de Android y PC. También se define un tipo enumerado para categorizar los eventos de entrada: *PRESSED, RELEASED* o *MOVED*. La interfaz Input brinda la oportunidad de gestionar los eventos al gusto del programador, con métodos de consulta directa, o mediante *polling*, gracias al uso de listas que habilitan el *buffered input*.

ANDROID INPUT

La implementación para Android de Input. Cuenta con un manejador de input de teclado y con un manejador de input táctil, que puede ser *single-touch* o *multi-touch*. La interfaz **AndroidTouchHandler** encapsula la funcionalidad de ambos tipos de manejadores de input táctil.

DESKTOP INPUT

La implementación para Android de Input. Cuenta con un manejador de input de teclado y con un manejador de input de ratón.

KEYBOARD HANDLER

Tanto la implementación de PC como la de Android siguen el patrón *observer*, registrándose como *listeners* de la ventana de la aplicación para poder registrar los eventos de entrada. El manejador de teclado gestiona cuatro listas: un array de booleanos que indica si cada una de las 128 teclas se ha pulsado en el último frame, una lista con todos los eventos de teclado registrados en el frame anterior, una lista con todos los eventos de teclado registrados en el frame actual y un *pool* de eventos que benefician al rendimiento global de la aplicación.

MOUSE HANDLER / MULTITOUCH HANDLER

Tanto la implementación de PC como la de Android siguen el patrón *observer*, registrándose como *listeners* de la ventana de la aplicación para poder registrar los eventos de entrada. El manejador de ratón de PC y el manejador de *multitouch* de Android manejan tres listas principales: una lista con todos los eventos de teclado registrados en el frame anterior, una lista con todos los eventos de teclado registrados en el frame actual y un *pool* de eventos; y cuatro listas auxiliares, que facilitan la consulta directa: un array de booleanos que informa de si se ha interactuando con la pantalla, dos arrays de enteros que informan de la posición en la que se ha interactuado con la pantalla, y un array de identificadores que se utilizan para diferenciar los botones del ratón en PC, y los dedos en Android.

El manejador de *multitouch* de Android, además, cuenta con dos atributos reales que indican el factor de escala que debe aplicarse para calcular la posición de interacción con la pantalla, por lo que **los cambios de tamaño afectan también al sistema de Input**, ya que debido a las distintas densidades de píxel de los dispositivos móviles pueden producirse errores.

INPUT UTILS

La clase **Pool** puede utilizarse con cualquier objeto, dado que es paramétrica, pero es especialmente útil a la hora de gestionar los eventos de entrada. En Android, en particular, el rendimiento puede resentirse notablemente si no se utiliza un *pool* de eventos. Así, en cada frame, en vez de crear nuevos eventos, se accede a los que estén disponibles del *pool*, para liberarse al final del siguiente frame. La interfaz **PoolObjectFactory** nos permite, utilizando el patrón de diseño *factory*, definir cómo se crean nuestros objetos para añadirlos al *pool.* Algunos objetos de la lógica también se benefician del uso de la clase Pool.

SOUND

La interfaz Sound es un *wrapper* para objetos de tipo sonido. Podemos reproducir el sonido, pausarlo, detenerlo, modificar parámetros como el volumen o el estado de reproducción en bucle, y, si lo deseamos, liberar manualmente los recursos que utiliza de la plataforma.

La arquitectura original diferenciaba entre las interfaces de sonido y de música, pero debido a varias complicaciones nos ha sido imposible incluir de forma satisfactoria y plena sonidos en Android y PC, por lo que **hemos decidido mantener sólo la música bajo la etiqueta genérica de la interfaz Sound**.

ANDROID SOUND

La implementación para Android de Sound tiene como atributo un objeto de tipo MediaPlayer de la librería multimedia de Android. Este objeto es el encargado de reproducir, pausar y parar la reproducción de música en Android, y trabaja en un hilo distinto al principal, por lo que muchos de los métodos cuentan con protección de sincronización.

DESKTOP SOUND

La implementación para PC de Sound tiene como atributo un objeto de tipo Clip de la librería de sonido de JavaX. Este objeto tiene poca información, por lo que la clase DektopSound debe suplirlo con un mayor número de atributos que suplan dicha funcionalidad.

AUDIO

La interfaz Audio gestiona el apartado sonoro del motor. Sin embargo, su utilidad es bastante reducida, dado que su único método se utiliza para crear nuevos objetos de la clase Sound. Debido a las limitaciones de la librería de PC, el motor sólo permite la carga de archivos con extensión *wav*, tanto en la implementación **DesktopAudio** como en la implementación **AndroidAudio**.

GESTORES DE RECURSOS

Una vez vistos todos los recursos multimedia que va a utilizar el motor del juego, podemos introducir los gestores de recursos. Estas herramientas son útiles para agilizar la carga de imágenes, sonidos, etc., así como para poder acceder a ellas de un modo rápido y eficaz en el módulo de la lógica. Las clases **AudioManager** y **PixmapManager** integran sendos diccionarios, con el nombre de archivo como clave, y el recurso específico (Sound o Pixmap) como valor. Así, a la vez que cargamos los recursos con los métodos de Graphics y Audio, podemos registrarlos en los gestores para que sean más fácilmente accesibles. Idealmente la lógica incorporará identificadores (en nuestro caso, parejas de tipo enumerado con cadenas de texto registradas en la clase **Assets** del módulo Logic) que coincidan con la clave de los diccionarios.

OTRAS UTILIDADES

El paquete de utilidades del módulo Engine cuenta con otras dos utilidades interesantes. Por un lado, una clase abstracta **Random**, que encapsula la funcionalidad de obtener números aleatorios. Aunque está construida *ad hoc* para resolver las necesidades puntuales de la lógica, es fácilmente ampliable y resulta bastante útil. Por otra parte, la clase **Pair** suple la inexistencia en Java del tipo pareja de valores; al ser una clase paramétrica podemos construir parejas con el tipo de objetos que queramos. Esta clase también se creó expresamente para suplir una necesidad de la lógica, pero podría ser aprovechada por los gestores de recursos, especialmente si se amplía a tupla.

STATE

La clase abstracta State representa un estado de nuestro juego. Cuenta con métodos para actualizar la lógica y para renderizar, así como métodos que intervienen en el ciclo de vida de la aplicación. El juego utiliza estados para llevar a cabo la lógica, por lo que todas las implementaciones de State se verán en el módulo de Lógica.

GAME

La interfaz Game agrupa todos los gestores y controla el estado global del juego. **DesktopGame** implementa la interfaz Game y utiliza los parámetros de su constructora para crear una ventana de JFrame; en el módulo DesktopApplication se declara una clase que hereda de DesktopGame, redefiniendo el método que devuelve el estado inicial del juego e implementando la función *main* que lanza la ejecución. **AndroidGame** además de implementar la interfaz Game hereda de Activity, por lo que pone a nuestra disposición todos los métodos del ciclo de vida de la actividad de Android; en concreto, en el módulo de AndroidApplication se declara una clase que redefine el método que devuelve el estado inicial del juego, y como este es llamado al finalizar el método *onCreate*, se lanza la ejecución del juego.

ANDROID RENDER VIEW

La clase AndroidRenderView hereda de SurfaceView e implementa la interfaz Runnable, por lo que incluye un método *run* que se ejecuta en una hebra distinta a la de la actividad principal. Es en esta hebra en la que ejecutaremos el bucle principal del juego. Tras actualizar toda la lógica del juego con el *delta time* calculado, el bucle principal procede al renderizado: **si el nivel de API es superior a 25 utilizará renderización por hardware**.

DESKTOP RENDER VIEW

La clase DesktopRenderView hereda de JFrame, y dado que la hebra de Swing se lanza en segundo plano no será necesario que nosotros lo hagamos explícitamente. Sin embargo, contaremos con un método *run* al que se llamará después de inicializar el juego, y que gestionará el bucle principal. No se indica ninguna parada de ejecución alfinal del bucle, por lo que el framerate en PC permanece desbloqueado. La clase DesktopRenderView además se registra como *listener* de ComponentAdapter, de modo que recibe información cuando la ventana se redimensiona: esto nos permite reescalar la superficie de renderizado en tiempo real. **Los bordes de la ventana de JFrame, sin embargo, interfieren en el reescalado del juego; al quitarlos, por otra parte, perdemos la capacidad de redimensionar la ventana**.

LÓGICA

En el módulo Logic se implementa la lógica de Switch Dash. En el paquete de estados contamos con las clases **LoadingState** (que se encarga de cargar todos los recursos), **MenuState** (que gestiona el menú principal), **HowToPlayState** (que gestiona el menú de ayuda), **MainGameState** (que gestiona la pantalla de juego) y **GameOverState** (que gestiona la pantalla de final de juego); todos heredan de la clase **GameState**, que a su vez hereda de State, aglutinando cierta funcionalidad común.

En el paquete objects contamos con las clases que heredan de **GameObject**, tales como **Player, BallsManager**, **Background** o **ScoreBaord**. Hemos añadido clases auxiliares como **Button**, **FontMapper** o **ParticleEmitter**, que facilitan la ejecución de ciertas secciones de la lógica y que se aprovechan de utilidades de nuestro motor como Random o ScaledGraphics.

La funcionalidad específica de cada una de estas clases de la lógica, así como detalles más extensos sobre la implementación de los demás módulos, **puede consultarse directamente en el JavaDoc del proyecto.**