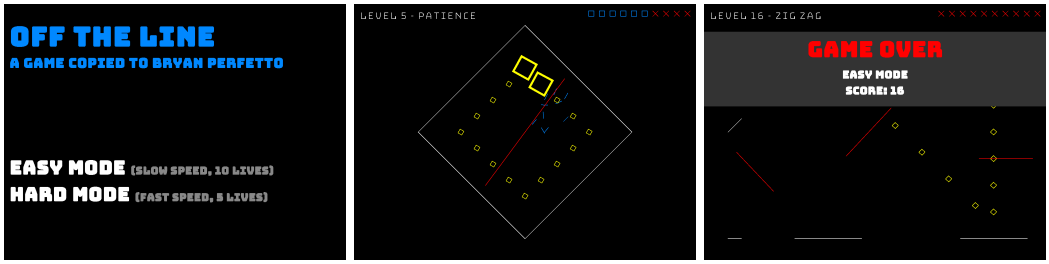
Práctica 1: Clon de *Off the line*

Fecha de entrega: 6 de diciembre de 2020, 23:55

Objetivo: Iniciación a la programación en Android y Java. Arquitectura para desarrollo multiplataforma



Off the line es un juego en HTML5 disponible en muchos sitios web de juegos para el navegador[1]. La mecánica es muy sencilla. Cada nivel está compuesto por uno o más *caminos* a través de los que se desplaza el "jugador". Al pulsar sobre la pantalla, éste salta de manera perpendicular a su desplazamiento hasta caer en otro camino. El objetivo es conseguir todas las "monedas" sin golpear ningún obstáculo (barras rojas) ni salirse de la pantalla.

1. El juego

La práctica consiste en la implementación de un clon de *Off the line* para móvil y ordenadores de escritorio. Al lanzarse, el juego mostrará una pantalla de menú donde se puede elegir entre dos modalidades de juego, que determinan la velocidad del jugador y el número de vidas iniciales.

Una vez elegida, el juego va mostrando los sucesivos niveles que el usuario deberá superar. Si en algún momento el jugador golpea un obstáculo o, por un salto, se sale de los límites de la pantalla se perderá una vida y se reiniciará el mismo nivel. Este reinicio ocurre dos segundos después de que ocurra el suceso que ha ocasionado la muerte. Un

[1] Por ejemplo en `https://armorgames.com/play/18514/off-the-line`

nivel se supera si se recogen toas las "monedas" y el jugador se mantiene vivo un segundo después de que esto suceda. El juego termina cuando se acaban todas las vidas o cuando se supera el último de los 20 niveles.

A continuación se indican posibles valores para algunos de los parámetros más importantes del juego. En los casos donde es necesario, se asume un tamaño de referencia de pantalla de 640×480 píxeles:

- En el modo fácil, el jugador se desplaza por los caminos a 250 píxeles por segundo y tiene 10 vidas.
- En el modo difícil, el jugador se desplaza a 400 píxeles por segundo y tiene 5 vidas.
- Durante los saltos, el jugador se desplaza a 1500 píxeles por segundo independientemente del nivel de dificultad.
- Tanto el jugador como las "monedas" giran sobre sí mismas 180 grados por segundo.
- El jugador es representado por un cuadrado de 12 píxeles de lado y las monedas por cuadrados de 8 píxeles de lado.
- Los principales colores usados en el juego son (0, 136, 255) para el jugador, (255, 0, 0) para los obstáculos y (255, 255, 255) para los caminos. El fondo es negro.
- Cuando el jugador golpea contra un obstáculo se desintegra en 10 partículas (líneas) de 6 píxeles.

Cuando el jugador salta y llega a un trazo, el *sentido* con el que se desplaza en él es lo más parecido posible al que llevaba originalmente. Ten en cuenta que eso puede hacer que, si ha caído en un segmento del trazo por el que iba, cambie su *sentido de giro* sobre la polilínea.

2. Requisitos de la implementación

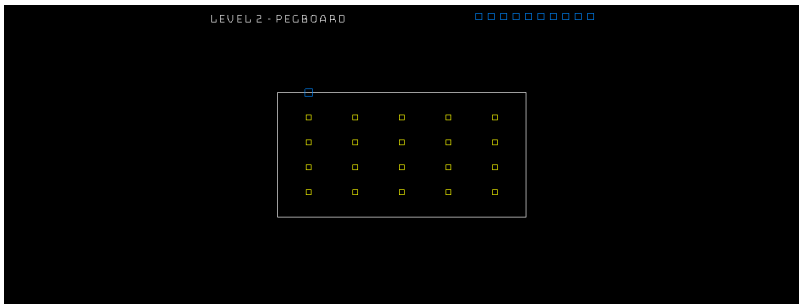
El juego se implementará en Java, y deberán poderse generar dos versiones diferentes, una para Android y otra para sistemas de escritorio (Windows o GNU/Linux). El código deberá estar correctamente distribuído en paquetes, ser comprensible y estar suficientemente documentado.

Para el desarrollo se hará uso de Android Studio, utilizando un único proyecto con varios *módulos*. La mayor parte del código *deberá ser común* y aparecer una única vez, compartida entre ambas versiones. Deberá existir así una *capa de abstracción* de la plataforma, que se implemente dos veces, una para Android y otra para escritorio. La implementación del juego deberá hacer uso únicamente de la capa de abstracción y de las funcionalidades del lenguaje que estén disponibles en ambas plataformas.

Dado que el punto de entrada de la aplicación es diferente en cada plataforma (en Android se necesita una `Activity` y en escritorio un `main()`) se permitirá también la existencia de dos módulos distintos (minimalistas) de "arranque del juego".

La aplicación se debe adaptar a *cualquier* resolución de pantalla. Independientemente de su relación de aspecto, el juego *no* deberá deformarse, sino aparecer *centrado* en la pantalla con "bandas" arriba y abajo o a izquierda y derecha. A modo de ejemplo, la figura siguiente muestra cómo debe verse si el dispositivo tuviera una resolución (muy alargada) de 2560×960 píxeles. La lógica espera una resolución de 640×480 por lo que se

añaden bandas negras a los lados para que la zona de juego quede en el centro ocupando todo el espacio a lo alto.



3. Consejos de implementación

Para abstraer la plataforma, podéis definir los siguientes interfaces:

- `Font`: envuelve un tipo de letra para ser utilizado al escribir texto.
- `Graphics`: proporciona las funcionalidades gráficas mínimas sobre la ventana de la aplicación:
  - `Font newFont(filename, size, isBold)`: crea una nueva fuente del tamaño especificado a partir de un fichero `.ttf`. Se indica si se desea o no fuente en negrita.
  - `void clear(color)`: borra el contenido completo de la ventana, rellenándolo con un color recibido como parámetro.
  - Métodos de control de la *transformación* sobre el canvas (`translate(x,y)`, `scale(x,y)`, `rotate(angle)`; `save()`, `restore()`). Las operaciones de dibujado se verán afectadas por la transformación establecida.
  - `void setColor(color)`: establece el color a utilizar en las operaciones de dibujado posteriores.
  - `void drawLine(x1, y1, x2, y2)`: dibuja una línea.
  - `void fillRect(x1, y1, x2, y2)`: dibuja un rectángulo relleno.
  - `void drawText(text, x, y)`: escribe el texto con la fuente y color activos.
  - `int getWidth()`, `int getHeight()`: devuelven el tamaño de la ventana.
- `Input`: proporciona las funcionalidades de entrada básicas. El juego no requiere un interfaz complejo, por lo que se utiliza únicamente la pulsación sobre la pantalla (o *click* con el ratón).
  - `class TouchEvent`: clase que representa la información de un toque sobre la pantalla (o evento de ratón). Indicará el tipo (pulsación, liberación, desplazamiento), la posición y el identificador del "dedo" (o botón).
  - `List<TouchEvent> getTouchEvents()`: devuelve la lista de eventos recibidos desde la última invocación.
- `Engine`: interfaz que aglutina todo lo demás. En condiciones normales, `Graphics` e `Input` serían *singleton*. Sin embargo, al ser *interfaces* y no existir en Java precompilador no es tan sencillo. El interfaz `Engine` puede ser el encargado de mantener las instancias y otros métodos útiles de acceso a la plataforma:

- `Graphics getGraphics()`: devuelve la instancia del "motor" gráfico.
- `Input getInput()`: devuelve la instancia del gestor de entrada.
- `InputStream openInputStream(filename)`: devuelve un *stream* de lectura de un fichero.

Para independizar la lógica de la resolución del dispositivo (o de la ventana) podéis ampliar la clase `Graphics` para que reciba un *tamaño lógico* (de "canvas") y que todas las posiciones se den en ese *sistema de coordenadas*. También podéis plantear el desarrollo de clases adicionales que proporcionen un mayor nivel de abstracción. En particular, para facilitar la puesta en marcha de la aplicación en cada plataforma, es posible que queráis ampliar `Engine` para incorporar la idea de *estado* de la aplicación.

Al ser interfaces, observad que *no* se indica cómo se crearán las instancias. Así, por ejemplo, la clase que implemente `Graphics` para la versión de escritorio podría necesitar recibir en su constructor la ventana de la aplicación, y la versión de Android el `SurfaceView` y `AssetManager`. La puesta en marcha de la aplicación tendrá que ser diferente en cada plataforma y estar encapsulada, en la medida de lo posible, en los módulos correspondientes. No obstante, la *carga* de los recursos no debe programarse dos veces, y debe formar parte del módulo de lógica.

Para implementar la lógica lo más cómodo es guardar el trazo sobre el que se está desplazando el jugador y a qué distancia está de su primer vértice. En cada *frame* se actualiza esa distancia en función del tiempo transcurrido y se calcula la nueva posición.

La "detección de colisiones" entre el jugador y los obstáculos (o los trazos durante un salto) se realizará mirando si el segmento se cruza con el segmento formado por la posición previa y la nueva del jugador. Para detectar cuando el jugador "toca" una moneda se mirará la distancia entre ese mismo segmento y el centro de la moneda. Si es menor que un umbral se asumirá que el jugador ha recolectado la moneda incluso aunque visualmente no hayan llegado a tocarse. Cuanto mayor sea ese umbral más fácil será el juego. La implementación original utiliza una distancia de 20.

Para implementarlo, en el módulo de lógica es interesante hacer una clase `Utils` con métodos estáticos:

- `segmentsIntersection(...)`: recibe dos segmentos y devuelve el punto donde se cruzan (si lo hacen).
- `sqrDistancePointSegment(...)`: recibe un segmento y un punto y devuelve el cuadrado de la distancia del punto al segmento. Tened en cuenta que esto *no es lo mismo* que la distancia del punto a la recta definida por el segmento.

Para cargar los niveles se utilizará un *parser* de JSON. Android incorpora uno de serie, pero no ocurre lo mismo en Java. Por simplicidad, se usará en ambas plataformas una librería externa, json-simple, que se incluirá como dependencia del módulo *de la lógica*.

Si en la versión para móvil tenéis problemas de rendimiento, minimizad la creación de objetos (analizad la posibilidad de cachearlos con *pools*) y usad, si es posible, *superficies hardware* (`SurfaceHolder::lockHardwareCanvas()`).

4. Recursos suministrados

Se proporcionan un par de ficheros de fuentes de letra (`.ttf`) para ser usadas en las etiquetas y "botones", y en el rótulo con el nombre del nivel actual. También se proporciona

un fichero JSON con la definición de los niveles, destilado del código fuente del juego original en HTML.

El JSON contiene un array con un elemento por cada nivel, donde se definen los trazos, los obstáculos y las "monedas". Todas las posiciones se proporcionan asumiendo que el nivel *está centrado en la pantalla* (es decir la posición (0,0) está en el centro), la coordenada Y crece *hacia arriba* y el espacio total disponible es de 640×480.

Un nivel tiene los siguientes campos:

- `name`: indica el nombre del nivel. Se muestra en la esquina superior izquierda de la ventana mientras se juega.
- `paths`: contiene los trazos o caminos por los que se mueve el jugador. Es un array, con un elemento por trazo. Cada trazo contiene los siguientes campos:
  - `vertices`: array de objetos con campos `x` e `y` con los vértices de la polilínea del trazo. El último vértice se considera unido con el primero.
  - `directions` (opcional): cada par de vértices adyacentes crea un *segmento*. Por defecto, si el "vector director" de un segmento es (x, y), el jugador saltará en dirección $(y, -x)$. En algunos trazos ese comportamiento por defecto debe ser sobreescrito con este array. Incluye un objeto *por segmento* con los campos `x` e `y` indicando la dirección del salto.
- `items`: array de objetos, uno por cada "moneda" del nivel. Incluyen los campos siguientes:
  - `x` e `y`: posición del centro del ítem en el nivel.
  - `radius`, `speed` y `angle` (opcional): si se especifican, el ítem *gira* alrededor de la posición (x, y) anterior en lugar de estar en ella. Lo hará a una distancia de `radius` píxeles. La velocidad de giro es `speed` grados por segundo y al principio está en el ángulo `angle`.
- `enemies`: array con objetos indicando los obstáculos. Tienen los siguientes campos:
  - `x`, `y`, `length` y `angle`: indica la posición (x, y) del *centro* del segmento, su longitud total (la mitad hacia cada lado) y la orientación en grados. Un obstáculo totalmente horizontal tendrá una orientación de 0 grados. Uno que apunte de la esquina inferior izquierda a la superior derecha, tendrá la orientación en 45 grados.
  - `speed` (opcional): velocidad de giro sobre su posición, en grados por segundo.
  - `offset`, `time1` y `time2` (opcional): el obstáculo *se desplaza* desde su posición inicial, (x, y) a esa misma posición desplazada tanto como indica el objeto `offset`. Al principio empieza desplazándose desde el punto (x, y). Para el movimiento de un punto a otro utiliza `time1` segundos, y espera `time2` segundos antes de reaundar la marcha en sentido opuesto.

El jugador siempre comienza en el primer vértice del primer trazo.

5. Partes opcionales

Siempre que los requisitos básicos de la práctica funcionen correctamente y estén bien implementados, se valorará positivamente la incorporación de características adicionales como por ejemplo:

- Inclusión de efectos de sonido.
- En la versión de escritorio:
  - Uso de pantalla completa.
  - Uso del teclado con el mismo efecto que pulsar con el ratón.
  - Pausa del juego (sin consumir recursos) cuando la aplicación pierda el foco.

6. Entrega de la práctica

La práctica debe entregarse utilizando el mecanismo de entregas del campus virtual, no más tarde de la fecha y hora indicada en la cabecera de la práctica. Es *indispensable* que la representación se adapte al tamaño de la ventana/pantalla. No hacerlo así supone el suspenso directo, al igual que la existencia de errores de compilación (en PC o en Android).

Sólo un miembro del grupo deberá realizar la entrega, que consistirá en un archivo `.zip` con el proyecto completo de Android Studio eliminando los ficheros temporales. Se añadirá también un fichero `alumnos.txt` con el nombre completo de los alumnos y un pequeño `.pdf` indicando la arquitectura de clases y módulos de la práctica y una descripción de las partes opcionales desarrolladas, si ha habido alguna.

El `.zip` deberá tener como nombre los nombres de los integrantes del grupo con la forma `Apellidos1_Nombre1-Apellidos2_Nombre2.zip`. Por ejemplo para el grupo formado por Miguel de Cervantes Saavedra y Santiago Ramón y Cajal, el fichero se llamará `DeCervantesSaavedra_Miguel-RamonYCajal_Santiago.zip`.

Bibliografía

- Beginning Android Games, Third Edition, Mario Zechner and J. F. DiMarzio, Apress, 2016.
- Developing games in Java, David Brackeen, Bret Barker, Lawrence Vanhelsuwe, New Riders.