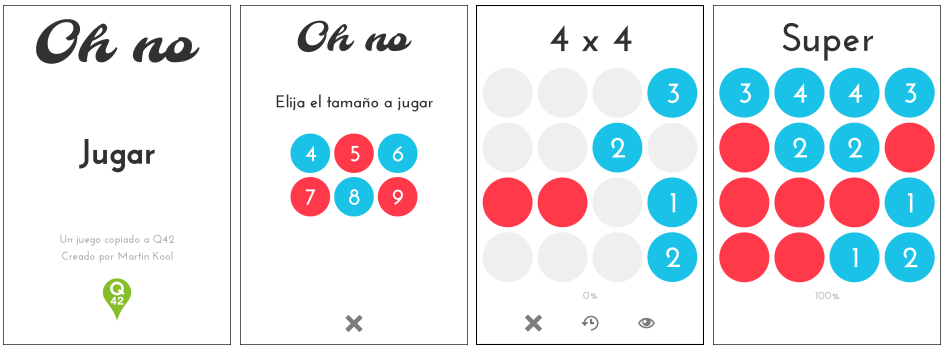
Práctica 1: Clon de *0h n0!*

Fecha de entrega: 14 de noviembre de 2021, 23:55

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



0h n0! es un juego para HTML5[1] y dispositivos móviles[2] creado por Martin Kool inspirado en los Kuromasu japoneses. Es un solitario en el que el jugador se enfrenta a un tablero cuadrado o rectangular dividido en celdas en las que puede haber puntos azules o rojos. Los círculos azules "ven" a otros círculos azules que estén en su misma fila o columna (en ambos sentidos). Los círculos rojos bloquean esa visión.

Al principio, en el tablero se proporcionan algunos círculos (rojos y/o azules), y para estos últimos se indica cuántos círculos ven con un número dentro del círculo. El jugador debe deducir el contenido del resto de celdas no rellenas. Las capturas muestran un ejemplo. Ten en cuenta que el borde negro se incluye por claridad en el guión, pero no debe aparecer en el juego.

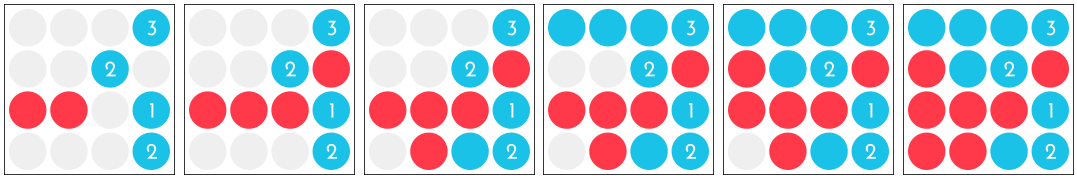
1. El juego

La práctica consiste en la implementación de un clon de *0h n0!* para móvil y ordenadores de escritorio usando Java. Al lanzarse, el juego mostrará una pantalla de bienvenida

[1] `https://0hn0.com/`
[2] `https://play.google.com/store/apps/details?id=com.q42.ohno`

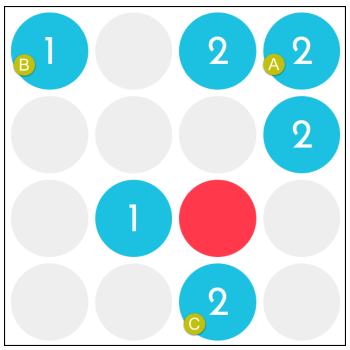
con un "menú" con una única opción de jugar. En la implementación original se muestra por separado una ventana de bienvenida y un menú con más opciones (tutorial, configuración, acerca de, etcétera) que no son necesarias para la práctica. Tampoco es necesario implementar el sistema de puntos.

Tras el menú, se permite seleccionar el tamaño del tablero (cuadrado) y finalmente se muestra el estado inicial del tablero que el usuario deberá completar. Por ejemplo, en el puzzle de la captura, el 1 ya tiene un azul visible (el 2 que tiene debajo), por lo que sus celdas superior e izquierda deben ser paredes. Eso hace que el 2 de la esquina inferior derecha solo pueda resolverse añadiendo un azul a su izquierda y, además, no puede tener otro tras él. Con razonamientos equivalentes llegamos a completar prácticamente todo el tablero. Para acabar, hay que saber que todos los deben ver al menos a otro, por lo que finalmente la esquina inferior izquierda será irremediablemente .



El jugador puede cambiar el estado de las celdas pulsando sobre ellas, lo que las hará "ciclar" en los estados vacío , punto y pared . Solo pueden modificarse las celdas que, al empezar la partida, estuvieran vacías. Si se intenta modificar una de las celdas preestablecidas, el juego avisa del error haciéndola oscilar en tamaño, y mostrando un candado sobre las que son pared. Un instante después de completar el tablero, el juego felicita al usuario y termina la partida (si es correcto) o resalta una celda incorrecta y muestra un mensaje con su error.

Mientras se juega, es posible rendirse ✖, deshacer el último movimiento (pudiendo llegar a deshacerlos todos) y pedir una pista . Las pistas dan indicaciones sobre un posible razonamiento a seguir para continuar con la resolución del puzzle. Se consiguen casi siempre a partir de *casillas con número* y algunos ya los hemos mencionado[3]:



1. Si un número tiene ya visibles el número de celdas que dice, entonces se puede "cerrar", es decir, poner paredes en los extremos. Ocurre en el 2 de la posición `A` de la figura, o en el primer movimiento del ejemplo anterior.

2. Si pusiéramos un punto azul en una celda vacía, superaríamos el número de visibles del número, y por tanto, debe ser una pared. Ocurre en el 1 de la posición `B`; no podemos poner un punto en la posición de su derecha, porque pasaría a tener tres visibles.

[3]La enumeración siguiente es más fácil de comprender si se han jugado varias partidas al juego original.

3. Si *no* ponemos un punto en alguna celda vacía, entonces es imposible alcanzar el número. Ocurre en el 2 de la posición `C`. Obligatoriamente tenemos que poner un punto en la celda de su izquierda, porque por la derecha (que es la única otra forma de añadir visibles) no vamos a conseguir llegar al 2 nunca.

Hay algunas pistas que pueden proporcionarse si el jugador *se ha equivocado*:

4. Un número tiene más casillas azules visibles de las que debería.
5. Un número tiene una cantidad insuficiente de casillas azules visibles y sin embargo ya está "cerrada" (no puede ampliarse más por culpa de paredes).

Además, hay pistas que ocurren sobre casillas que no son números:

6. Si una celda está vacía y cerrada y no ve ninguna celda azul, entonces es pared (todos los puntos azules deben ver al menos a otro).
7. En sentido opuesto, si hay una celda punto puesta por el usuario que está cerrada y no ve a ninguna otra, entonces se trata de un error por el mismo motivo.

Es posible añadir pistas adicionales que resulten más explicativas, aunque no sean estrictamente necesarias porque están cubiertas por las anteriores:

8. Un número que no ve suficientes puntos no está aún cerrado y solo tiene abierta una dirección. Está cubierta por la pista 3.
9. Un número no está cerrado y tiene varias direcciones, pero la suma alcanzable es el valor que hay que conseguir. Basta con llenar el resto de celdas vacías para resolverlo. Está también cubierta por la pista 3.
10. En sentido opuesto, una celda de tipo número no está cerrada pero si se ponen en punto el resto de celdas vacías que tiene alcanzables no llegará a su valor, por lo que es un futuro error. Si no se incluye esta pista, el programa dará incorrectamente varias veces la pista 3 para al final terminar indicando el error de la 5.

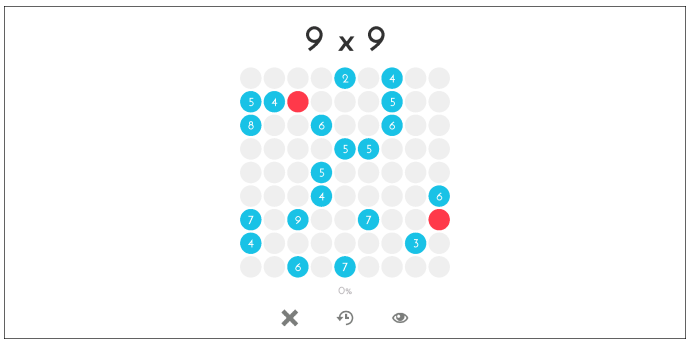
2. Requisitos de la implementación

El juego se implementará en Java, y deberán poderse generar dos versiones diferentes, una para Android y otra para sistemas de escritorio (Windows, GNU/Linux o Mac). 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, compartido 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 se ejecutara en un móvil en apaisado con una resolución de 2220 × 1080 (relación 18.5:9). La lógica espera una relación de 2:3 (por ejemplo 400 × 600) por lo que se añaden bandas blancas a los lados para que la zona de juego quede en el centro ocupando todo el espacio a lo alto. Como en las figuras anteriores, el marco negro se incluye como referencia, pero no se mostrará en el juego:



Los puzzles que genere el juego deben ser siempre resolubles, tener una única solución y no tener números que superen el tamaño (ancho o alto). Para eso, el programa deberá intentar resolver los puzzles que genere con la aplicación reiterada de las pistas, y descartar aquellos que no consiga resolver.

La solución deberá estar cerrada desde el punto de vista del usuario. El interfaz deberá ser fluído, mostrar pequeñas animaciones (*fade-in* y *fade-out*) en los cambios del tipo de celda o al mostrar una pista, dar *feedback* al usuario cuando pulsa sobre una celda que no puede modificar, etcétera.

3. Consejos de implementación

Para abstraer la plataforma, podéis definir los siguientes interfaces:

- `Image`: envuelve una imagen de mapa de bits para ser utilizada a modo de *sprite*:
  - `int getWidth()`: devuelve el ancho de la imagen.
  - `int getHeight()`: devuelve el alto de la imagen.
- `Font`: envuelve un tipo de letra para ser utilizado al escribir texto.
- `Graphics`: proporciona las funcionalidades gráficas mínimas sobre la ventana de la aplicación:
  - `Image newImage(String name)`: carga una imagen almacenada en el contenedor de recursos de la aplicación a partir de su nombre.
  - `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(int 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)`; `save()`, `restore()`). Las operaciones de dibujado se verán afectadas por la transformación establecida.
  - `void drawImage(Image image, ...)`: recibe una imagen y la muestra en la pantalla. Se pueden necesitar diferentes versiones de este método dependiendo de si se permite o no escalar la imagen, si se permite elegir qué porción de la imagen original se muestra, etcétera.
  - `void setColor(color)`: establece el color a utilizar en las operaciones de dibujado posteriores.
  - `void fillCircle(cx, cy, r)`: dibuja un círculo relleno del color activo.
  - `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:

  - `Graphics getGraphics()`: devuelve la instancia del "motor" gráfico.
  - `Input getInput()`: devuelve la instancia del gestor de entrada.

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, cread un "modelo" del juego con clases para las celdas, el tablero o las pistas que *sean independientes de la representación en pantalla*. Esas clases deberían poder usarse para implementar el mismo juego sin interfaz gráfica (para un terminal). Luego implementad clases que se encarguen de la representación visual de esos elementos (la "vista").

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 los ficheros de fuentes de letra (`.ttf`) usados en el juego original. También se proporcionan las imágenes con los iconos de los botones de volver, deshacer y pedir pista, un candado para sobreimpresionar sobre las paredes fijas si el usuario intenta cambiarlas, y el logotipo del estudio que creó el juego original.

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.
  - 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.