Programación Multimedia y Dispositivos Móviles

UD 16. KorGE

Javier Carrasco

Curso 2024 / 2025

KorGE

16. KorGE

Esta unidad tratará la programación multimedia de manera superficial, aprovechando el recién adquirido conocimiento de Kotlin, se explorará su potencial para el desarrollo de un videojuego multiplataforma haciendo uso de **KorGE**[1].

¿Qué es KorGE?

KorGE Game Engine es un motor para videojuegos moderno, creado en Kotlin y totalmente diseñado para ser portable y fácil de utilizar. Además, es multiplataforma, por lo que pueden crearse videojuegos para escritorio, móvil y/o web. Como curiosidad, el nombre de KorGE viene de **K**otlin c**OR**utines **G**ame **E**ngine.

16.1. Instalar KorGE

El primer paso será instalar el IDE con el que se trabajará, si no lo tienes ya. Se instalará para ello **IntelliJ IDEA Community Edition**[2], que puedes descargar desde la página de *JetBrains*.

Una vez instalado el IDE, el siguiente paso será añadir el *plugin* para poder utilizar KorGE.

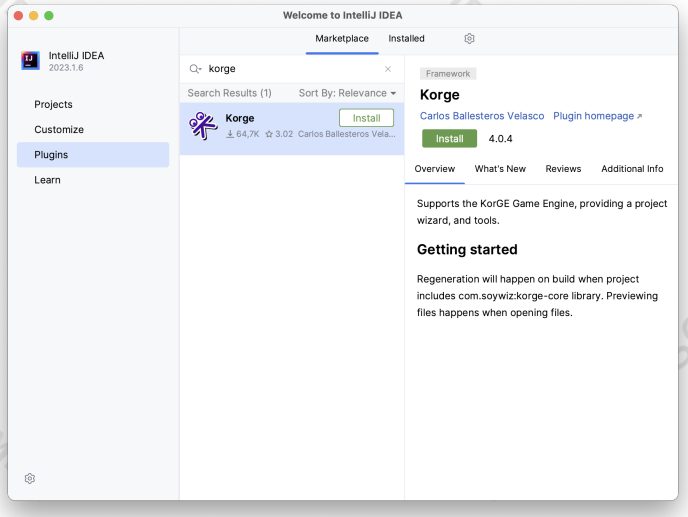

Figura 1

Una vez añadido el *plugin* ya podría crearse un proyecto KorGE desde el sección *Projects* de IntelliJ. En la sección *Generators* podrás ver que aparece la opción **KorGE Game**, en la que podrás encontrar diferentes *Starter Kits* y algunos *Showcases*. Como curiosidad, verás que puedes probar

1 KorGE (https://korge.org/)
2 IntelliJ IDEA Community Edition (https://www.jetbrains.com/idea/download/)

los diferentes *templates* con la opción *Playable here* que encontrarás en la definición.

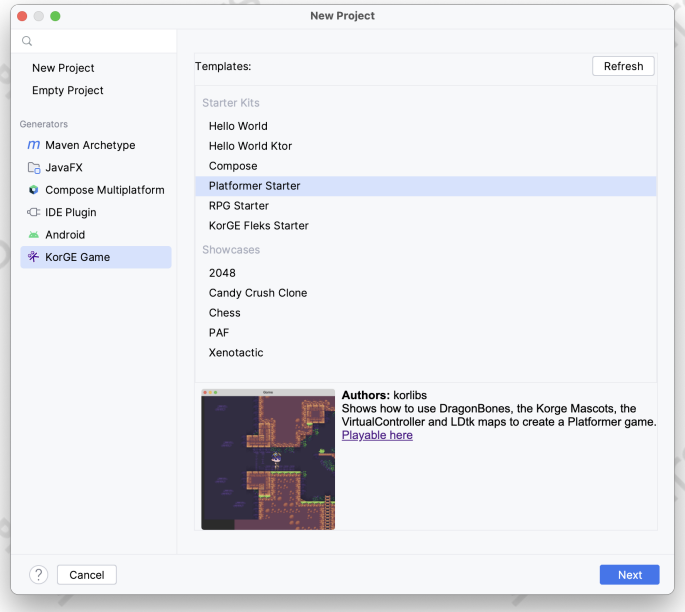

Figura 2

16.2. Entendiendo el entorno

Creando un proyecto inicial mediante el *template* ***Hello World*** podrás tener la primera toma de contacto. La primera carga deberá descargar bastante, por lo que puede tardar. En primer lugar, el código fuente, lo encontrarás en la ruta **<proyecto>/src/commonMain/kotlin**, dónde estará la clase de entrada, **main.kt**.

Figura 3

Para lanzar el proyecto se dispone de la opción **runJvmAutoreload** que puedes encontrar en el *Gradle*, o directamente desde *Run/Debug Configurations* de la barra superior.

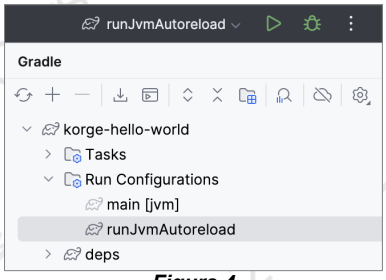

Figura 4

Esta opción te permite lanzar el juego y, además, te permitirá hacer pruebas en caliente, de forma que si modificas código y guardas los cambios, se verá reflejado casi de inmediato.

Figura 5

Prueba a modificar la escala, cambiando el valor 0.8 que viene por defecto por 0.4 y guarda el proyecto, verás reflejado el cambio en el juego.

```
val image = image(resourcesVfs["korge.png"].readBitmap()) {
 rotation = maxDegrees
 anchor(.5, .5)
 scale(0.4)
 position(256, 256)
}
```

Durante la ejecución del proyecto, puedes utilizar la tecla **F7** en la ventana ***Game*** para mostrar una ventana de *debug*, la cual te permitirá interactuar con los elementos en pantalla y modificar valores para ver como se aplican los cambios. También muestra información como los FPS, máquina virtual sobre la que está corriendo, etc.

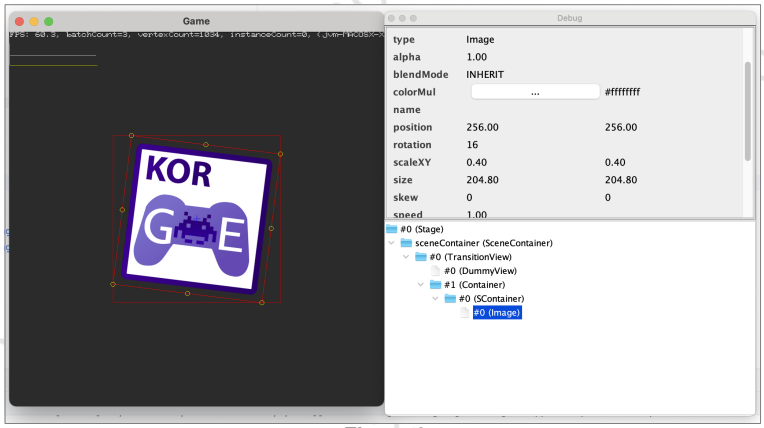

Figura 6

El *plugin* de KorGE instalado también añade dos nuevos botones a la barra superior del IDE.

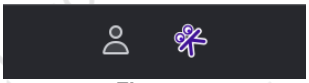

Figura 7

El primero te permite hacer *login* si te registras en la web oficial de KorGE, pero lo más interesante es que te da acceso a la documentación y al canal de *Discord*.

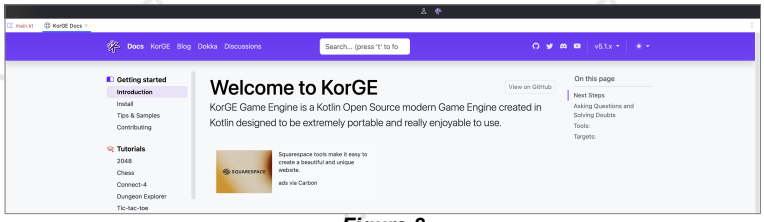

Figura 8

El segundo te da acceso a la **KorGE Store**, dónde podrás encontrar una gran cantidad de recursos para añadir a tus proyectos.

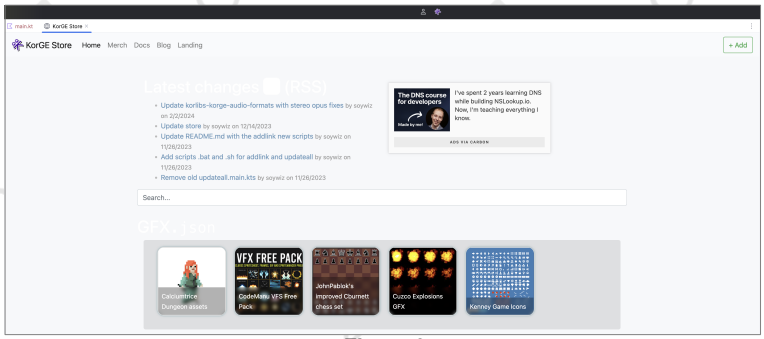

Figura 9

16.3. Tutorial para crear un 2048

16.3.1. Configuración del proyecto

El primer paso será crear un nuevo proyecto KorGE utilizando la plantilla ***Hello World***. El nombre para este proyecto será **m2048Game**. Una cargado, se configurará el juego en el fichero **build.gradle.kts**, dónde encontrarás las siguientes líneas:

```
korge {
 id = "com.sample.demo"
```

Modifícalas y añade la nueva línea de la siguiente forma, sincroniza el *Gradle*.

```
korge {
    id = "com.sample.m2048game"
    name = "2048"
```

El bloque *korge* permite especificar cierta información del juego, es este caso se ha indicado el identificador y el nombre del juego.

Ahora, en la clase `main.kt` que encontrarás en la ruta **src/commonMain/kotlin/** verás que aparece el código inicial de la plantilla. De esta parte, lo que más interesa es la llamada a Korge, dónde se especifica el tamaño de la ventana, el color de fondo y otros elementos. A continuación, se definirá el tamaño de la pantalla y su color de fondo, también se eliminará lo innecesario para este ejemplo.

```
suspend fun main() = Korge(
    windowSize = Size(480, 640),
    title = "2048",
    bgcolor = RGBA(253, 247, 240)
) {
    // TODO: Código para el juego.
}
```

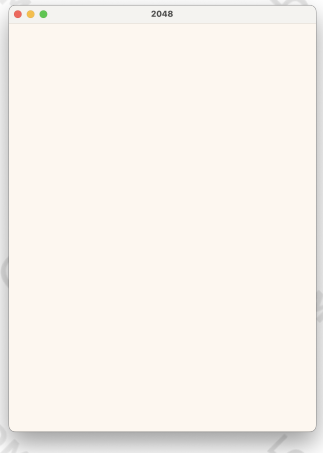

Figura 10

Si lanzas el proyecto con *runJvmAutoreload* podrás ver la nueva ventana del juego. KorGE permite lanzar la ejecución en diferentes plataformas, pero para las pruebas, la mejor suele ser la de JVM. También puedes lanzar la ejecución de la aplicación sin *Autoreload* escribiendo en la línea de comandos de *Gradle* la siguiente línea.

```
gradle runJvm
```

De esta forma quedaría registrada en el lanzador del IDE para futuras ejecuciones.

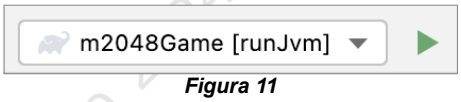

Figura 11

Como habrás observado, la forma en la que se ha especificado el color de fondo es diferente a la que aparecía en el código de ejemplo.

Existen diferentes maneras de especificar colores en KorGE.

- `RGBA(0x00, 0x00, 0xFF, 0xFF)`
- `RGBA(0, 0, 255)`
- `Colors[“#0000FF”]`
- `Colors.BLUE`

16.3.2. Añadir y ubicar vistas

La estructura de vistas de KorGE se basa en dos tipos de elementos, las vistas y los contenedores. Los contenedores son vistas que pueden contener otras vistas como hijos. Cuando se renderiza la escena KorGE comienza desde el contenedor raíz y desciende por la estructura según se han ido añadiendo los elementos. El efecto que se obtiene es que se pintan primero los elementos del fondo y va avanzando, de esta manera, los elementos de las primeras líneas solaparán a los del fondo.

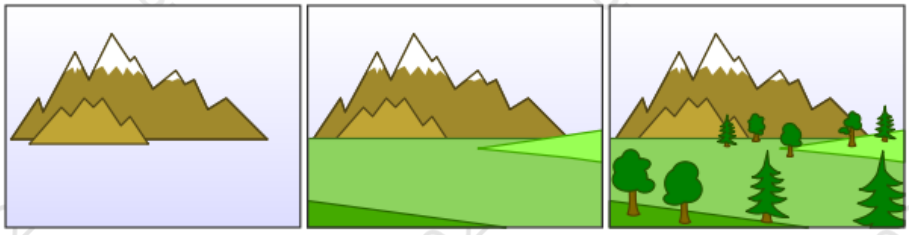

Figura 12

Cada vista tendrá sus propias propiedades de tamaño (***width*** y ***height***) y posición (***x*** e ***y***). Las propiedades para indicar la posición determinan su posición dentro del contenedor padre.

Para ajustar las celdas del juego, como se dispondrá de una rejilla de 4x4 celdas del mismo tamaño de alto y ancho, más espacio entre ellas, los más fácil será ajustar su tamaño al tamaño de la ventana. Puede dividirse el tamaño de la ventana entre 5 y así obtener el tamaño justo de la celda.

Crea la siguiente variable en el método ***main***.

```
1  val cellSize = views.virtualWidth / 5.0
```

El objeto *views* permite acceder a las propiedades de la ventana. La siguiente variable determinará el tamaño de la zona de las celdas.

```
1  val fieldSize = 50 + 4 * cellSize
```

Las dos siguientes variables establecerán los márgenes izquierdo y superior.

```
val leftIndent = (views.virtualWidth - fieldSize) / 2
val topIndent = 150.0
```

A continuación, se añadirá la primera vista a la escena. Los elementos principales en KorGE son de tipo ***Graphics*** y sus derivados, como ***RoundRect*** y ***Circle***. Para crear el terreno dónde se ubicarán las celdas se utilizará *RoundRect*, al que se le indicará el tamaño y la posición.

```
val bgField = roundRect(
    size = Size(fieldSize, fieldSize),
    radius = RectCorners(5.0),
    fill = Colors["#b9aea0"]
) {
    position(leftIndent, topIndent)
}
```

También puedes utilizar las propiedades ***x*** e ***y*** en lugar de *position* si lo ves más claro. Una vez creado habrá que añadirlo a la vista contenedora, puede hacerse de dos formas…

```
bgField.addTo(this)
```

… o

```
addChild(bgField)
```

Aunque en las últimas versiones no siempre es necesario, ya que por defecto añade los elementos creados al contenedor inicial.

Para añadir una celda se utilizará el objeto ***Graphics***, estos permiten dibujar formas simples y complejas. Además, tienen su propio DSL (*Domain Specific Language*), y sus funciones principales son ***fill***, ***stroke*** y ***fillStroke***, aunque también se dispone de otras como ***rect***, ***rectHole***, ***roundRect***, ***arc***, ***circle*** y ***ellipse***.

Se comenzará por crear la primera celda con las siguientes líneas.

```
graphics {
    it.position(leftIndent, topIndent)
    fill(Colors["#cec0b2"]) {
        roundRect(10.0, 10.0, cellSize, cellSize, 5.0)
    }
}
```

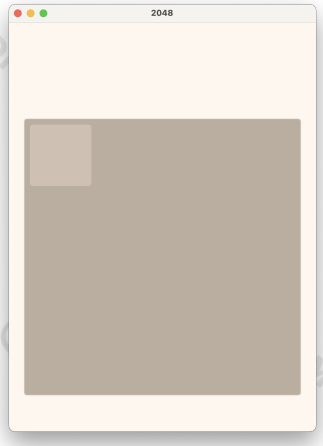

Figura 13

Pero como hacen falta 16 celdas, lo mejor será montar un doble bucle para pintarlas todas.

```
graphics {
    it.position(leftIndent, topIndent)
    fill(Colors["#cec0b2"]) {
        for (i in 0..3) {
            for (j in 0..3) {
```

```
                roundRect(
                    10 + (10 + cellSize) * i,
                    10 + (10 + cellSize) * j,
                    cellSize,
                    cellSize,
                    5.0
                )
            }
        }
    }
}
```

Ahora se añadirá una nueva celda fuera del campo de juego, esta representará el logo del juego.

```
val bgLogo = roundRect(
    Size2D(cellSize, cellSize),
    RectCorners(5.0),
    fill = Colors["#edc403"]
) {
    x = leftIndent
    y = 30.0
}
```

También es posible hacer uso de posiciones relativas para ubicar elementos. KorGE proporciona métodos como ***alignRightToLeftOf***, ***alignTopToBottomOf***, ***centerBetween*** y ***centerOn***.

Ahora se crearán dos nuevas vistas para establecer la mejor puntuación y la puntuación actual, y se ubicarán haciendo uso de las posiciones relativas.

```
val bgBest = roundRect(
    Size2D(cellSize * 1.5, cellSize * 0.8),
    RectCorners(5.0), fill = Colors["#bbae9e"]
) {
    alignRightToRightOf(bgField)
    alignTopToTopOf(bgLogo)
}

val bgScore = roundRect(
    Size2D(cellSize * 1.5, cellSize * 0.8),
    RectCorners(5.0), fill = Colors["#bbae9e"]
) {
    // alinea su derecha con la izquierda de.
    alignRightToLeftOf(bgBest, 24)
    alignTopToTopOf(bgBest)
}
```

En el siguiente punto se verá como añadir textos a la escena.

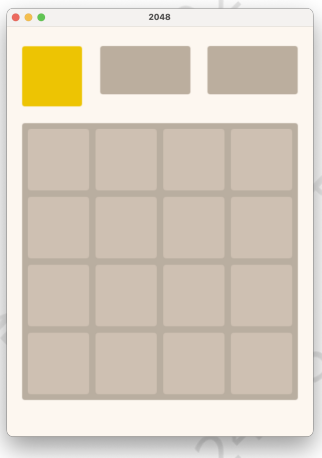

Figura 14

16.3.3. Añadir textos

KorGE permite añadir texto de una manera sencilla, pero hay que añadir la fuente que se quiera utilizar. Descarga los archivos para la fuente (*clear_sans.fnt* y *clear_sans.png*) desde los recursos proporcionados para el tema. Una vez descargados, deberás añadirlos al proyecto, a la carpeta ***resources*** que hay dentro de la carpeta ***commonMain*** dentro de ***src***.

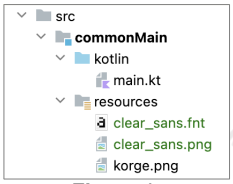

Figura 15

En código, en la clase ***main.kt***, se añadirá una nueva propiedad llamada ***font*** que se encargará de cargar la fuente.

```
// Fuente del texto.
val font = resourcesVfs["clear_sans.fnt"].readBitmapFont()
```

Ahora se añadirá el primer texto, el del logo, para ello se utilizará el método ***text*** dónde se indicará el texto a mostrar, el tamaño, el color y la fuente a utilizar. El método ***centerOn*** se utilizará para indicar la posición del texto sobre el elemento, el fondo del logo.

```
// Texto logo.
text("2048", cellSize * 0.5, Colors.WHITE, font).centerOn(bgLogo)
```

Si quieres añadir una fuente tipo TTF, basta con indicar el nombre del archivo importado al proyecto y el método de lectura de la fuente.

```
val font2 = resourcesVfs["crotah_italic.ttf"].readTtfFont()
```

El tamaño de la fuente puede verse afectado, en este caso por ejemplo, el factor de corrección para que quede bien sería 0.25. Ahora se añadirá el resto de textos para los marcadores de puntuación y mejor puntuación.

```
text("BEST", cellSize * 0.25, Colors.WHITE, font) {
    centerXOn(bgBest)
    alignTopToTopOf(bgBest, 5.0)
}
text("0", cellSize * 0.5, Colors.WHITE, font) {
    alignment = TextAlignment.MIDDLE_CENTER
    alignTopToTopOf(bgBest, 20.0)
    centerXOn(bgBest)
}
text("SCORE", cellSize * 0.25, Colors.WHITE, font) {
    centerXOn(bgScore)
    alignTopToTopOf(bgScore, 5.0)
}
text("0", cellSize * 0.5, Colors.WHITE, font) {
    alignment = TextAlignment.MIDDLE_CENTER
    centerXOn(bgScore)
    alignTopToTopOf(bgScore, 20.0)
}
```

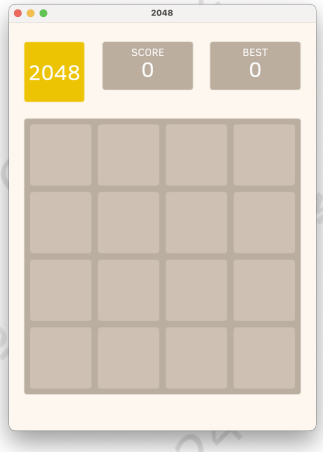

Figura 16

Con este código ya estaría preparado el panel de puntuaciones del juego y el logo.

16.3.4. Añadir imágenes

Al igual que con las fuentes, KorGE suporta perfectamente el uso de imágenes. A continuación, se verá una ligera introducción, ya que este punto requeriría un capítulo a parte. Descarga las imágenes (*restart.png* y *undo.png*) desde los recursos proporcionados para el tema. Una vez descargados, deberás añadirlos al proyecto, a la carpeta ***resources*** que hay dentro de la carpeta ***commonMain*** dentro de ***src***.

Ya en código, en la clase ***main.kt***, se añadirán dos nuevas propiedades para cargar las imágenes desde los recursos.

```
val restartImg = resourcesVfs["restart.png"].readBitmap()
val undoImg = resourcesVfs["undo.png"].readBitmap()
```

Ahora que ya se dispone de la imagen cargada en los objetos creados, se crearán los botones en los puntos indicados.

```
// Mostrar y ubicar las imágnes.
val btnSize = cellSize * 0.3
val restartBlock = container {
    val bg = roundRect(Size(btnSize, btnSize), RectCorners(5.0), fill = Colors.DARKGRAY)
    image(restartImg) {
        size(btnSize * 0.8, btnSize * 0.8)
        centerOn(bg)
    }
    alignTopToBottomOf(bgBest, 5)
    alignRightToRightOf(bgBest)
}
val undoBlock = container {
    val bg = roundRect(Size(btnSize, btnSize), RectCorners(5.0), fill = Colors.DARKGRAY)
    image(undoImg) {
        size(btnSize * 0.6, btnSize * 0.6)
        centerOn(bg)
    }
    alignTopToTopOf(restartBlock)
    alignRightToLeftOf(restartBlock, 5)
}.onClick {
    println("click")
}
```

Para crear botones se utilizar el bloque ***container***, en el que se define el tamaño e imagen del botón, así como su ubicación. Este bloque, como puedes ver, dispone del evento ***onClick***, que se detallará más adelante.

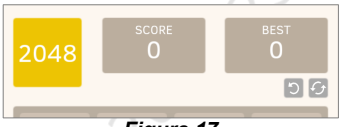

Figura 17

16.3.5. Estado del juego e interacción

Llegados a este punto, ya están dispuestos en pantalla todos los elementos estáticos que se necesitan para representar el juego. El objetivo de este apartado es añadir bloques dinámicos, control de la posición en el mapa para el estado del juego y definir los controles que permitan al usuario mover los bloques.

16.3.5.1. Números

En primer lugar se definirán los número que intervienen en el juego. El tablero contiene 16 celdas, en dos de ellas pueden haber dos cuatros, y el usuario las podría unir para obtener el siguiente número.

Se creará el siguiente el fichero **Number.kt** (junto a main.kt) que contendrá una clase **enum** llamada **Number**. Esta clase contendrá dos propiedades, valor y color de la celda (RGBA). El valor definirá el número a mostrar, y el color indicará el tono de la celda.

```
enum class Number(val value: Int, val color: RGBA) {
    ZERO(2, RGBA(240, 228, 218)),
    ONE(4, RGBA(236, 224, 201)),
    TWO(8, RGBA(255, 178, 120)),
    THREE(16, RGBA(254, 150, 92)),
    FOUR(32, RGBA(247, 123, 97)),
    FIVE(64, RGBA(235, 88, 55)),
    SIX(128, RGBA(236, 220, 146)),
    SEVEN(256, RGBA(240, 212, 121)),
    EIGHT(512, RGBA(244, 206, 96)),
    NINE(1024, RGBA(248, 200, 71)),
    TEN(2048, RGBA(256, 194, 46)),
    ELEVEN(4096, RGBA(104, 130, 249)),
    TWELVE(8192, RGBA(51, 85, 247)),
    THIRTEEN(16384, RGBA(10, 47, 222)),
    FOURTEEN(32768, RGBA(9, 43, 202)),
    FIFTEEN(65536, RGBA(181, 37, 188)),
    SIXTEEN(131072, RGBA(166, 34, 172))
}
```

16.3.5.2. Bloques

Ahora se creará una vista especial llamada **Block** que se moverá por el tablero cuando el usuario interactúe con este. Para ello se creará la clase **Block.kt** que extenderá de *Container* con una única propiedad *Number*, creada en el paso anterior.

Como se verá a continuación, esta clase hará uso de propiedades creadas en la función *main* del fichero *main.kt*, para poder acceder a ellas deberán declararse e inicializarse fuera del método, por lo que habrá que modificarlo de la siguiente forma.

```
var cellSize: Double = 0.0
```

```
var fieldSize: Double = 0.0
var leftIndent: Double = 0.0
var topIndent: Double = 0.0
var font: BitmapFont by Delegates.notNull()

suspend fun main() = Korge(
    windowSize = Size(480, 640), title = "2048", bgcolor = RGBA(253, 247, 240)
) {
    font = resourcesVfs["clear_sans.fnt"].readBitmapFont()

    cellSize = views.virtualWidth / 5.0
    fieldSize = 50 + 4 * cellSize
    leftIndent = (views.virtualWidth - fieldSize) / 2
    topIndent = 150.0
    ...
```

Al tratarse de propiedades de nivel superior, no puede utilizarse *lateinit*, pero para las propiedades de tipo *Double* no hay problema, basta con inicializarlas a cero. Pero, para la propiedad de tipo *BitmapFont*, es necesario utilizar un ***Delegates.notNull***, lo que requiere que se inicialicen las propiedades en las primeras líneas del *main*.

Ahora, la clase **Block.kt** quedará de la siguiente forma.

```
class Block(val number: Number) : Container() {
    init {
        roundRect(Size2D(cellSize, cellSize), RectCorners(5.0), fill = number.color)
        val textColor = when (number) {
            ZERO, ONE -> Colors.BLACK
            else -> Colors.WHITE
        }
        text(number.value.toString(), textSizeFor(number), textColor, font) {
            centerBetween(0.0, 0.0, cellSize, cellSize)
        }
    }

    private fun textSizeFor(number: Number): Double = when (number) {
        ZERO, ONE, TWO, THREE, FOUR, FIVE -> cellSize / 2
        SIX, SEVEN, EIGHT -> cellSize * 4 / 9
        NINE, TEN, ELEVEN, TWELVE -> cellSize * 2 / 5
        THIRTEEN, FOURTEEN, FIFTEEN -> cellSize * 7 / 20
        SIXTEEN -> cellSize * 3 / 10
    }
}
```

En el constructor *init* se crea el bloque tal como se ha visto en puntos anteriores y se asigna el texto que le corresponda. El método *textSizeFor* se utiliza para resolver el tamaño del texto según la longitud del número que se deba mostrar.

Para terminar con la clase *Block*, se creará la siguiente función de extensión sobre *Container* que permitirá agilizar el código.

```
fun Container.block(number: Number) = Block(number).addTo(this)
```

Concretamente, esta función creará un nuevo bloque con el número indicado y se añadirá al contenedor en cuestión.

16.3.5.3. Crear nuevos bloques

Para gestionar los bloques del juego, será necesario añadir dos nuevas variables al fichero **main.kt**. Una será **blocks**, que contendrá un *HashMap* con los *ids* de los bloques, y la otra será **freeId**, para indicar el siguiente bloque libre disponible.

```
val blocks = mutableMapOf<Int, Block>()
var freeId = 0

suspend fun main() = Korge(...
```

También se añadirán los siguientes métodos fuera de la función **main**. Estos dos primeros métodos, ***columnX(number)*** y ***rowY(number)***, dado el número pasado, devolverán la posición real que ocupa.

```
fun columnX(number: Int): Double = leftIndent + 10 + (cellSize + 10) * number
fun rowY(number: Int): Double = topIndent + 10 + (cellSize + 10) * number
```

El siguiente método se encargará de crear un nuevo bloque y añadirlo al mapa junto con su identificador y posición.

```
fun Container.createNewBlockWithId(id: Int, number: Number, position: Position) {
    blocks[id] = block(number).position(columnX(position.x), rowY(position.y))
}
```

Observa como se hace uso de la función de extensión ***block*** creada anteriormente, añadiendo el nuevo bloque al contenedor. También se hace uso de la clase **Position** que se detallará en el siguiente punto.

El último método que se añadirá calculará el identificador del nuevo bloque que se añadirá, llamará a *createNewBlockWithId* y devolverá el identificador asignado.

```
fun Container.createNewBlock(number: Number, position: Position): Int {
    val id = freeId++
    createNewBlockWithId(id, number, position)
    return id
}
```

16.3.5.4. La clase PositionMap

Se creará un nuevo fichero llamado **PositionMap.kt** que contendrá las clases **Position** y **PositionMap**. La primera, *Position*, se utilizará para indicar las coordenadas x e y. La segunda clase, *PositionMap*, se utilizará para gestionar el estado del tablero, creando en el constructor una matriz de 4x4 con los identificadores de los bloques, o -1 para indicar que no está en uso.

```
class Position(val x: Int, val y: Int)

class PositionMap(private val array: IntArray2 = IntArray2(4, 4, -1)) {}
```

Fíjate en el tipo utilizado para crear el array, ***IntArray2***, este tipo pertenece a la biblioteca **korlibs**. Ahora se añadirán los siguientes métodos a la clase *PositionMap*.

```
class PositionMap(private val array: IntArray2 = IntArray2(4, 4, -1)) {
    private fun getOrNull(x: Int, y: Int) = if (array.get(x, y) != -1)
        Position(x, y)
    else null

    private fun getNumber(x: Int, y: Int) = array.tryGet(x, y)?.let {
        blocks[it]?.number?.ordinal ?: -1
    } ?: -1

    operator fun get(x: Int, y: Int) = array[x, y]

    operator fun set(x: Int, y: Int, value: Int) {
        array[x, y] = value
    }

    fun forEach(action: (Int) -> Unit) {
        array.forEach(action)
    }

    override fun equals(other: Any?): Boolean {
        return (other is PositionMap) && this.array.data.contentEquals(other.array.data)
    }

    override fun hashCode() = array.hashCode()
}
```

A continuación, se detalla la utilidad de cada uno de los métodos creados:

- **getOrNull(x, y)**: devuelve la posición del objeto si hay un bloque en la posición, en caso contrario devuelve nulo.
- **getNumber(x, y)**: devuelve el número del objeto *Number* que haya en la posición indicada, o -1 en caso de no existir bloque en esa posición.
- **get(x, y)**: devuelve del identificador del bloque en la posición indicada.
- **set(x, y, value)**: asigna el identificador de bloque a la posición indicada.
- **forEach(action)**: llama a la acción *forEach* para cada identificador de la matriz.
- **equals(other)**: comprueba si el objeto pasado es un objeto *PositionMap* y si su posición el el mapa es la misma.
- **hashCode()**: delega el cálculo del *hash* al array.

Ya solo quedará crear una instancia de la clase **PositionMap** en el fichero **main.kt**.

```
var map = PositionMap()
val blocks = ...
```

16.3.5.5. Generando nuevos bloques

Tras lanzar el juego, deberá crearse un bloque inicial para lanzar la partida. Se creará la función de extensión **generateBlock** para la clase *Container*. Esta función seleccionará una posición al azar en el tablero y se le asignará un número inicial.

```
fun Container.generateBlock() {
    val position = map.getRandomFreePosition() ?: return
    val number = if (Random.nextDouble() < 0.9) Number.ZERO else Number.ONE
    val newId = createNewBlock(number, position)
    map[position.x, position.y] = newId
}
```

El número aleatorio está preparado para obtener en un 90% un 2 (*ZERO*) y en un 10% un 4. Para conseguir una posición aleatoria se deberá crear el método **getRandomFreePosition()** en la clase **PositionMap**.

```
fun getRandomFreePosition(): Position? {
    val quantity = array.count { it == -1 }
    if (quantity == 0) return null
    val chosen = Random.nextInt(quantity)
    var current = -1
    array.each { x, y, value →
        if (value == -1) {
            current++
            if (current == chosen) {
                return Position(x, y)
            }
        }
    }
    return null
}
```

Por último, en el método **main**, se llamará a **generateBlock** para que aparezca el bloque inicial.

```
suspend fun main() = Korge(...) {
    ...
    generateBlock()
}
```

El resultado será un bloque inicial, probablemente con valor 2, en una posición aleatoria del tablero cada vez que se lance el juego.

16.3.5.6. Interacción con el usuario

En KorGE, la interacción se gestiona mediante eventos, por ejemplo, la pulsación de una tecla, pasar el ratón por un punto o la pulsación del ratón. El motor genera los eventos y habrá que crear los *listeners* para capturarlos y adaptarlos a las necesidades. Algunos de estos eventos son:

- *onKeyDown*, *onKeyUp*, *onKeyTyped*
- *onOver*, *onOut*, *onDown*, *onUp*
- *onMove*, *onClick*, *onMouseDrag*, *onSwipe*
- etc.

Existen muchas formas para gestionar la interacción con el usuario, pero se centrará la atención en la función DSL (*Domain Specific Language*) que ofrece el motor. Si quieres ampliar esta información puedes consultar la documentación[3] de KorGE.

Se comenzará por definir los *listeners* para los eventos, en el fichero **PositionMap.kt** añade la siguiente clase *enum* para simplificar.

```
enum class Direction {
    UP, DOWN, LEFT, RIGHT
}
```

Ahora, en la función **main**, al final, se añadirá el *listener* para la pulsación de la tecla (**onKeyDown**).

```
keys {
    down {
        when (it.key) {
            Key.LEFT → moveBlocksTo(Direction.LEFT)
            Key.RIGHT → moveBlocksTo(Direction.RIGHT)
            Key.UP → moveBlocksTo(Direction.UP)
            Key.DOWN → moveBlocksTo(Direction.DOWN)
            else → Unit
        }
    }
}
```

La librería que se utilizará para implementar la clase **Key** es **korlibs.event**. Dentro del bloque **down** se evalúa la pulsación de la tecla (it). Las propiedades del evento **Key.Type.DOWN** son **id**, **key**, **keyCode** y **character**. En este caso se evalúa **key** para conocer la tecla pulsada. El método **moveBlockTo(direction)**, que se define a continuación como función de extensión sobre **Stage**, de momento mostrará en consola la dirección obtenida tras la captura del evento.

```
fun Stage.moveBlocksTo(direction: Direction) {
    println(direction)
}
```

3 Input (https://docs.korge.org/views/input/)

Otro evento que se añadirá será para deslizar el ratón (**onSwipe**), este es algo más complejo que el anterior. Se considerará deslizar cuando el usuario pulse con el ratón en un punto y arrastre soltando en otro punto distinto. Se capturan las cuatro direcciones posibles y se indicará el umbral, que será la cantidad de píxeles que debe moverse.

```
onSwipe(20.0) {
    when (it.direction) {
        SwipeDirection.LEFT → moveBlocksTo(Direction.LEFT)
        SwipeDirection.RIGHT → moveBlocksTo(Direction.RIGHT)
        SwipeDirection.TOP → moveBlocksTo(Direction.UP)
        SwipeDirection.BOTTOM → moveBlocksTo(Direction.DOWN)
    }
}
```

Si lanzas el juego, podrás ver en la consola como se muestra la dirección elegida según la tecla o el arrastre que hagas del ratón.

16.3.6. Animando el juego

Ya se dispone del bloque inicial del juego, en este punto se añadirá el movimiento básico para el funcionamiento del juego y la unión de los bloques.

16.3.6.1. Comprobaciones iniciales

El primer paso será añadir dos nuevas variables al inicio del fichero **main.kt** para controlar si se está reproduciendo una animación y si es el final del juego.

```
var isAnimationRunning = false
var isGameOver = false

suspend fun main() = Korge(...
```

En la clase **PositionMap** se añadirán dos nuevos métodos encargados de comprobar si se dispone de movimientos posibles.

```
class PositionMap(private val array: IntArray2 = IntArray2(4, 4, -1)) {
    ...

    private fun hasAdjacentEqualPosition(x: Int, y: Int) = getNumber(x, y).let {
        it == getNumber(x - 1, y)
            || it == getNumber(x + 1, y)
            || it == getNumber(x, y - 1)
            || it == getNumber(x, y + 1)
    }

    fun hasAvailableMoves(): Boolean {
        array.each { x, y, _ →
            if (hasAdjacentEqualPosition(x, y)) return true
        }
```

```
        return false
    }
    ...
```

El método **hasAdjacentEqualPosition()** comprueba la existencia de alguna celda adyacente con el mismo valor numérico, devolviendo -1 en caso contrario o si está fuera del campo, y devolviendo el número si es la hay. El método **hasAvailableMoves()** hace uso del método anterior para devolver verdadero o falso por cada posición del tablero.

Ahora, en el método **moveBlocksTo()** creado previamente en el fichero **main.kt**, se añadirán las comprobaciones para estas nuevas variables.

```
fun Stage.moveBlocksTo(direction: Direction) {
    if (isAnimationRunning) return
    if (!map.hasAvailableMoves()) {
        if (!isGameOver) {
            isGameOver = true
            println("Game Over")
        }
    }
    return
}
```

16.3.6.2. Texto “Game Over” superpuesto

El siguiente paso será añadir la función de extensión showGameOver a la clase **Container** para mostrar el texto superpuesto en el escenario. Además, se añadirá un *callback* para devolver el clic sobre la opción *“Try again”* y eliminar así el texto superpuesto.

```
fun Container.showGameOver(onRestart: () -> Unit) = container {
    fun restart() {
        this@container.removeFromParent()
        onRestart()
    }

    position(leftIndent, topIndent)

    roundRect(Size2D(fieldSize, fieldSize), RectCorners(5.0), fill = Colors["#FFFFFF33"])
    text("Game Over", 60.0, Colors.BLACK, font) {
        centerBetween(0.0, 0.0, fieldSize, fieldSize)
        y -= 60
    }
    text("Try again", 30.0) {
        centerBetween(0.0, 0.0, fieldSize, fieldSize)
        y += 20
        textSize = 30.0
        font = font
        color = RGBA(0, 0, 0)
        onOver { color = RGBA(90, 90, 90) }
        onOut { color = RGBA(0, 0, 0) }
```

```
        onDown { color = RGBA(120, 120, 120) }
        onUp { color = RGBA(120, 120, 120) }
        onClick { restart() }
    }

    keys.down {
        when (it.key) {
            Key.ENTER, Key.SPACE → restart()
            else → Unit
        }
    }
}
```

Observa como se añade un efecto de cambio de color sobre el texto *"Try again"* utilizando los método *onOver*, *onOut*, *onDown* y *onUp*, así como el evento *onClick*. También se añade la posibilidad de intentarlo de nuevo utilizando las teclas *Enter* o *Space*. También se añadirá la función de extensión **restart()** a la clase **Container** para reiniciar el tablero.

```
fun Container.restart() {
    map = PositionMap()
    blocks.values.forEach { it.removeFromParent() }
    blocks.clear()
    generateBlock()
}
```

El método **moveBlocksTo()** quedará ahora de la siguiente manera:

```
fun Stage.moveBlocksTo(direction: Direction) {
    if (isAnimationRunning) return
    if (!map.hasAvailableMoves()) {
        if (!isGameOver) {
            isGameOver = true
            showGameOver {
                isGameOver = false
                restart()
            }
        }
    }
    return
}
```

Y como ya se dispone del método **restart()**, se añadirá esta funcionalidad al botón creado para reiniciar la partida.

```
val restartBlock = container {
    ...
    onClick {
        this@Korge.restart()
    }
}
```

Para poder ver el texto superpuesto en este punto, deberás comentar el *if* que controla si hay movimientos disponibles para que se ejecute el código que contiene, de tal manera que, con un movimiento aparecerá y podrás comprobar que funciona correctamente.

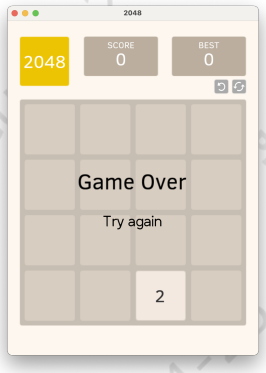

Figura 18

16.3.6.3. Cambios de posición en el mapa

Desde el método **moveBlocksTo()**, tras comprobar que no se está ejecutando una animación y que no es el final del juego, deberá actualizarse el mapa según la acción realizada por el usuario.

Primero deberá calcularse el nuevo mapa, y los movimientos y fusiones de los bloques en el tablero, tras lo cual, si hay diferencias entre el mapa nuevo y el antiguo, se animarán los cambios.

Por partes, en el fichero **main.kt** se añadirán los siguientes métodos que faciliten la obtención del número de la celda y eliminar una celda.

```
val blocks = mutableMapOf<Int, Block>()
var freeId = 0

fun numberFor(blockId: Int) = blocks[blockId]!!.number
fun deleteBlock(blockId: Int) = blocks.remove(blockId)!!.removeFromParent()
```

En la clase **PositionMap** se añadirán los dos métodos siguientes, el primero se encargará de devolver una instancia de **Position** si se encuentra una posición vacío y nulo en caso contrario.

```
fun getNotEmptyPositionFrom(direction: Direction, line: Int): Position? {
    when (direction) {
        Direction.LEFT -> for (i in 0..3) getOrNull(i, line)?.let { return it }
        Direction.RIGHT -> for (i in 3 downTo 0) getOrNull(i, line)?.let { return it }
        Direction.UP -> for (i in 0..3) getOrNull(line, i)?.let { return it }
        Direction.DOWN -> for (i in 3 downTo 0) getOrNull(line, i)?.let { return it }
    }
    return null
}
```

La segunda, para facilitar la copia del mapa, útil más adelante.

```
fun copy() = PositionMap(array.copy(data = array.data.copyOf()))
```

También se añadirá a la clase **Number** el siguiente método.

```
fun next() = entries[(ordinal + 1) % entries.size]
```

Este método se encargará de obtener el número que debe mostrarse tras unir dos bloques. De vuelta al fichero **main.kt**, se añadirán los siguientes métodos.

Este primer método se encargará de las animaciones, se buscará el efecto rebote cuando dos bloque se unan.

```
fun Animator.animateScale(block: Block) {
    val x = block.x
    val y = block.y
    val scale = block.scale
    tween( // scale up
        block::x[x - 4], block::y[y - 4], block::scale[scale + 0.1],
        time = 0.1.seconds,
        easing = Easing.LINEAR
    )
    tween( // return to original position
        block::x[x], block::y[y], block::scale[scale],
        time = 0.1.seconds,
        easing = Easing.LINEAR
    )
}
```

Para hacer uso del método **tween** será necesaria la librería **korlibs.korge.tween.***. El método *tween* permite realizar animaciones de forma fácil.

El siguiente método, **showAnimation()**, se encarga de ejecutar la animación del movimiento de los bloques, así como obtener las uniones y actualizar la lista de *merges* y crear el bloque nuevo que salga como resultado.

```
fun Stage.showAnimation(
    moves: List<Pair<Int, Position>>,
    merges: List<Triple<Int, Int, Position>>,
    onEnd: () -> Unit
) = launchImmediately {
    animate {
        parallel {
            moves.forEach { (id, pos) ->
                moveTo(
                    blocks[id]!!, columnX(pos.x), rowY(pos.y),
                    0.15.seconds, Easing.LINEAR
                )
            }
            merges.forEach { (id1, id2, pos) ->
                sequence {
                    parallel {
                        moveTo(
                            blocks[id1]!!, columnX(pos.x), rowY(pos.y),
                            0.15.seconds, Easing.LINEAR
                        )
                        moveTo(
                            blocks[id2]!!, columnX(pos.x), rowY(pos.y),
                            0.15.seconds, Easing.LINEAR
                        )
                    }
                    block {
                        val nextNumber = numberFor(id1).next()
```

```
                        deleteBlock(id1)
                        deleteBlock(id2)
                        createNewBlockWithId(id1, nextNumber, pos)
                    }
                    sequenceLazy {
                        animateScale(blocks[id1]!!)
                    }
                }
            }
        }
        block {
            onEnd()
        }
    }
}
```

Observa que se crea el bloque cuando se hace la unión, pero la animación se ejecuta dentro del bloque **sequenceLazy**, de esta forma, el efecto rebote se ejecuta sobre el nuevo bloque creado tras eliminar los dos que han producido la unión, y esto se hará cuando termine, no al inicio de la animación.

Por último, se creará el método **calculateNewMap()** para obtener el nuevo mapa resultante con cada movimiento que haga el usuario.

```
fun calculateNewMap(
    map: PositionMap,
    direction: Direction,
    moves: MutableList<Pair<Int, Position>>,
    merges: MutableList<Triple<Int, Int, Position>>
): PositionMap {
    val newMap = PositionMap()
    val startIndex = when (direction) {
        Direction.LEFT, Direction.UP -> 0
        Direction.RIGHT, Direction.DOWN -> 3
    }
    var columnRow = startIndex

    fun newPosition(line: Int) = when (direction) {
        Direction.LEFT -> Position(columnRow++, line)
        Direction.RIGHT -> Position(columnRow--, line)
        Direction.UP -> Position(line, columnRow++)
        Direction.DOWN -> Position(line, columnRow--)
    }

    for (line in 0..3) {
        var curPos = map.getNotEmptyPositionFrom(direction, line)
        columnRow = startIndex
        while (curPos != null) {
            val newPos = newPosition(line)
            val curId = map[curPos.x, curPos.y]
```

```
            map[curPos.x, curPos.y] = -1

            val nextPos = map.getNotEmptyPositionFrom(direction, line)
            val nextId = nextPos?.let { map[it.x, it.y] }
            //two blocks are equal
            if (nextId != null && numberFor(curId) == numberFor(nextId)) {
                //merge these blocks
                map[nextPos.x, nextPos.y] = -1
                newMap[newPos.x, newPos.y] = curId
                merges += Triple(curId, nextId, newPos)
            } else {
                //add old block
                newMap[newPos.x, newPos.y] = curId
                moves += Pair(curId, newPos)
            }
            curPos = map.getNotEmptyPositionFrom(direction, line)
        }
    }
    return newMap
}
```

En primer lugar se crea una instancia de **PositionMap**, se define un índice de inicio que se basa en la dirección, que será el número de columna o fila por la que se comenzará el cálculo, este índice se almacenará en la variable **columnRaw**.

Se crea el método **newPosition()**, al cual se le indicará el número un línea, y según la dirección tomada, devolverá la nueva posición.

Por último, se recorrerán todas las filas, se obtendrá la posición actual del bloque y se obtiene su id, se obtiene la siguiente posición que tenga bloque y se obtiene su id, se comprueban los números para ver si son iguales o no. Si son iguales, se fusionan, borrando la posición siguiente del mapa y añadiendo el id al nuevo mapa. Si no son iguales, se mueve el número actual en la nueva posición al nuevo mapa y se añade a la lista de movimientos. Una vez recorridas todas las posiciones se devuelve el nuevo mapa.

Por último, el método **moveBlocksTo()** quedará de la siguiente manera:

```
fun Stage.moveBlocksTo(direction: Direction) {
    if (isAnimationRunning) return
    if (!map.hasAvailableMoves()) {
        if (!isGameOver) {
            isGameOver = true
            showGameOver {
                isGameOver = false
                restart()
            }
        }
    }

    val moves = mutableListOf<Pair<Int, Position>>()
```

```
14      val merges = mutableListOf<Triple<Int, Int, Position>>()
15      val newMap = calculateNewMap(map.copy(), direction, moves, merges)
16
17      if (map != newMap) {
18          isAnimationRunning = true
19          showAnimation(moves, merges) { // when animation ends
20              map = newMap
21              generateBlock()
22              isAnimationRunning = false
23          }
24      }
25  }
```

Comprueba el funcionamiento ejecutando el juego, ya deberían fusionarse los bloques y tener el juego en marcha.

16.3.7. Gestión de la información

En este punto ya se dispone de un juego totalmente "jugable", para terminar de completarlo, falta añadir la puntuación y la posibilidad de almacenar la mejor puntuación.

La puntuación del juego funcionará de la siguiente manera; el jugador acumulará puntos según vaya uniendo bloques, de tal forma que, si une dos bloque de 2, dando como resultado uno de 4, la puntuación a sumar es 4, si une dos bloque de 4, dando como resultado uno de 8, sumará a la puntuación 8 puntos más. Además, cuando la puntuación actual supera la mejor puntuación, esta se irá actualizando. Finalmente se guardarán las dos puntuaciones, esto se podrá ver algo más adelante.

16.3.7.1. Propiedades para la puntuación

Las propiedades para la puntuación deben ser accesibles desde cualquier punto del fichero **main.kt**, por lo tanto, deberán definirse como propiedades de nivel superior.

Además, hay que elegir bien el tipo de datos de estas propiedades. Si se piensa en el uso de estar propiedades, primero hay que establecer el valor inicial de las propiedades. Segundo, deben actualizarse los valores de las propiedades desde el método **main** con los valores almacenados en el histórico (si existe). Tercero, deberán actualizarse según vayan fusionándose los bloques. Y cuarto, cuando se actualicen las propiedades, deberán actualizarse los marcadores y almacenar los datos.

La primera aproximación que viene la cabeza es definirlas como de tipo **Int**, pero esta aproximación hace que el acceso al almacenamiento, a la actualización de los marcadores, a la fusión de los bloques y a los propios datos deba hacerse en el mismo lugar. Esto no es viable, ya que si observas la estructura del proyecto, no todos es accesible desde cualquier punto.

Para este caso concreto, la mejor opción es utilizar la clase **ObservableProperty** proporcionada por la librería **korlibs.io.async.ObservableProperty**. Esta clase dispone de un constructor inicial para indicar un valor de inicio y dos métodos, **observe(handler)** y **update(value)**.

El primero permite añadir un manejador para observar y manejar los valores de la propiedad cuando esta se actualice. La segunda permite actualizar el valor de la propiedad. Haciendo uso de esta clase se podrán actualizar las propiedades desde cualquier punto, incluso desde fuera de main.kt.

16.3.7.2. Iniciando las puntuaciones.

```
val score: ObservableProperty<Int> = ObservableProperty(0)
val best: ObservableProperty<Int> = ObservableProperty(0)

suspend fun main() = Korge(...
```

Con estas líneas se instancian la propiedades para la puntuación actual y mejor puntuación con el valor inicial. Ahora se establecerán los observadores de la propiedad **score** que actualizará **best** si fuese necesario, y el segundo se encargará de la actualización de **best** en el almacenamiento.

```
suspend fun main() = Korge(
    windowSize = Size(480, 640), title = "2048", bgcolor = RGBA(253, 247, 240)
) {
    // Fuente del texto.
    font = resourcesVfs["clear_sans.fnt"].readBitmapFont()

    score.observe {
        if (it > best.value) best.update(it)
    }
    best.observe {
        // TODO: Se actualizará en el storage.
    }
    ...
```

Ahora, en la vista que muestra la puntuación…

```
text("BEST", cellSize * 0.25, Colors.WHITE, font) {
    ...
}
text("0", cellSize * 0.5, Colors.WHITE, font) {
    alignment = TextAlignment.MIDDLE_CENTER
    alignTopToTopOf(bgBest, 20.0)
    centerXOn(bgBest)
}
```

… deberá inicializarse el valor inicial para la vista de la mejor puntuación, y se añadirá el observador para actualizar el dato cuando cambie la puntuación.

```
text("BEST", cellSize * 0.25, Colors.WHITE, font) {
    ...
}
text(best.value.toString(), cellSize * 0.5, Colors.WHITE, font) {
    alignment = TextAlignment.MIDDLE_CENTER
    alignTopToTopOf(bgBest, 20.0)
```

```
    centerXOn(bgBest)

    best.observe {
        text = it.toString()
    }
}
```

Y se repetirá la misma operación para la vista que contiene la puntuación actual de la partida.

```
text("SCORE", cellSize * 0.25, Colors.WHITE, font) {
    ...
}
text(score.value.toString(), cellSize * 0.5, Colors.WHITE, font) {
    alignment = TextAlignment.MIDDLE_CENTER
    centerXOn(bgScore)
    alignTopToTopOf(bgScore, 20.0)

    score.observe {
        text = it.toString()
    }
}
```

16.3.7.3. Actualizando puntuaciones

La actualización de las puntuaciones a medida que vayan fusionándose se bloques deberá realizarse desde el método **moveBlocksTo()**.

```
fun Stage.moveBlocksTo(direction: Direction) {
    ...

    if (map != newMap) {
        isAnimationRunning = true
        showAnimation(moves, merges) {
            // update score
            val points = merges.sumOf {
                numberFor(it.first).value
            }
            score.update(score.value + points)

            // when animation ends
            map = newMap
            generateBlock()
            isAnimationRunning = false
        }
    }
}
```

Con este código ya empezarás a sumar puntos a medida que vayas fusionando bloques, verás que también se actualiza la mejor puntuación.

16.3.7.4. Reiniciando la puntuación

Otra acción sobre la puntuación será el reinicio de la misma, para ello, se añadirá la siguiente línea en el método **restart()** de la clase Container.

```
fun Container.restart() {
    map = PositionMap()
    blocks.values.forEach { it.removeFromParent() }
    blocks.clear()
    // restart score
    score.update(0)
    generateBlock()
}
```

Ahora se reiniciará la puntuación y se mantendrá la mejor puntuación obtenida, pero si cierras el juego, la máxima puntuación se pierde. Esto es algo que se solucionará a continuación.

16.3.7.5. Clase NativeStorage

Para guardar información de manera persistente, KorGE facilita la clase **NativeStorage**[4], la cual permite el almacenamiento de información mediante clave-valor de manera sencilla.

Como particularidad, no puedes instanciar un objeto de esta clase fuera de **main**, debes utilizar para ello la función de extensión **Views.storage**, esto permitirá acceder a cualquier instancia de esta desde cualquier punto siempre que se disponga de un **Stage**, ya que contiene las *views*.

Una vez instanciada dentro del método **main** se recogerá el valor almacenado, controlando si existe o no. También deberá actualizarse el método **observe** sobre el objeto **best** para que se guarde la información cada vez que esta cambie, utilizando para toda la operación la clave *"best"*.

```
suspend fun main() = Korge(
    windowSize = Size(480, 640), title = "2048", bgcolor = RGBA(253, 247, 240)
) {
    // Fuente del texto.
    font = resourcesVfs["clear_sans.fnt"].readBitmapFont()

    // Storage.
    val storage = views.storage
    // Recuperar la mejor puntuación.
    best.update(storage.getOrNull("best")?.toInt() ?: 0)

    score.observe {
        if (it > best.value) best.update(it)
    }
    best.observe {
        storage["best"] = it.toString()
    }
```

4 Preferences (https://docs.korge.org/preferences/)

16.3.7.6. Clase History

En este punto se creará la clase **History**, que se utilizará para almacenar cada uno de los movimientos del jugador. Recuerda que se dispone de un tablero de 4x4, que contiene 16 bloques, y estos bloques cambiarán con cada movimiento. Estos movimientos se llamarán **Element** en el historial. Crea en un nuevo fichero la clase **History**.

```
class History {
    class Element {

    }
}
```

Como la clase **Element** debe conocer el estado del tablero, deberá contener una propiedad que sea un *IntArray* de 16 elementos, que contendrá los *ids* de los bloques, y también deberá conocerse la puntuación de ese movimiento.

```
class History {
    class Element(val numberIds: IntArray, val score: Int)
}
```

Para instanciar la clase **History**, deberá recuperarse el estado en el que quedó el juego la última vez, para recuperarlo de utilizando la clase **NativeStorage** se requiere un *String*, que puede ser nulo. También se creará un *callback* que se llamará cada vez que el historial se actualice.

```
class History(from: String?, private val onUpdate: (History) -> Unit) {
    ...
}
```

Mediante la propiedad **from** se obtendrá el historial que se haya guardado, y como el historial es una lista de elementos **Element**, deberá definirse una propiedad para contenerlos.

```
class History(from: String?, private val onUpdate: (History) -> Unit) {
    class Element(val numberIds: IntArray, val score: Int)

    private val history = mutableListOf<Element>()
}
```

Ahora será necesario inicializar la propiedad **history** con los valores que se pasen a través del parámetro **from**. Recuerda que serán 17 elementos, los 16 que forman el array más la puntuación.

```
class History(from: String?, private val onUpdate: (History) -> Unit) {
    class Element(val numberIds: IntArray, val score: Int)

    private val history = mutableListOf<Element>()

    init {
        if (from!!.isNotEmpty()) {
            from.split(";").forEach {
                history.add(elementFromString(it))
```

```
            }
        }
    }

    private fun elementFromString(string: String): Element {
        val numbers = string.split(",").map { it.toInt() }
        if (numbers.size != 17) throw IllegalArgumentException("Incorrect history")
        return Element(IntArray(16) { numbers[it] }, numbers[16])
    }
}
```

La idea es utilizar la coma (,) para separar el entero del *array*, y el punto y como (;) para separa cada representación en formato *String* de cada objeto **Element**. También se añadirá la operación inversa sobrecargando el método ***toString()***.

```
override fun toString(): String {
    return history.joinToString(";") {
        it.numberIds.joinToString(",") + "," + it.score
    }
}
```

Y para terminar con la clase **History**, se añadirán los siguientes métodos con los que gestionar los datos:

- Añadir un nuevo Element (ids y puntuación) al histórico.
- Deshacer el último movimiento, devolviendo el elemento actual.
- Eliminar el historial (al reiniciar el juego).
- Comprobar si el histórico está vacío.

También se añadirá la propiedad **currentElement** que devolverá el último elemento del historial.

```
class History(...) {
    ...

    val currentElement: Element get() = history.last()

    ...

    override fun toString(): String {
        return history.joinToString(";") {
            it.numberIds.joinToString(",") + "," + it.score
        }
    }

    fun add(numberIds: IntArray, score: Int) {
        history.add(Element(numberIds, score))
        onUpdate(this)
```

```
    }

    fun undo(): Element {
        if (history.size > 1) {
            history.removeAt(history.size - 1)
            onUpdate(this)
        }
        return history.last()
    }

    fun clear() {
        history.clear()
        onUpdate(this)
    }

    fun isEmpty() = history.isEmpty()
}
```

16.3.7.7. Aplicando el histórico

En primer lugar se definirá una variable histórico antes de la función **main**, y una vez dentro, se comprueba si hay información para el histórico almacenada.

```
var history: History by Delegates.notNull()

suspend fun main() = Korge(...) {
    // Fuente del texto.
    font = resourcesVfs["clear_sans.fnt"].readBitmapFont()

    // Storage.
    val storage = views.storage
    history = History(storage.getOrNull("history")) {
        storage["history"] = it.toString()
    }
    ...
}
```

Ahora, dentro del método **main**, hay creado un bloque llamado **undoBlock** para **Container**, cuando el jugador pulse el botón se deshará el último movimiento.

```
val undoBlock = container {
    val bg = ...
}.onClick {
    this@Korge.restoreField(history.undo())
}
```

El método **restoreField()**, asociado a **Container**, se encargará de actualizar la puntuación, limpiar el tablero y representar las posiciones previas.

```
fun Container.restoreField(history: History.Element) {
```

```
    map.forEach { if (it != -1) deleteBlock(it) }
    map = PositionMap()
    score.update(history.score)
    freeId = 0

    val numbers = history.numberIds.map {
        if (it >= 0 && it < Number.entries.size)
            Number.entries[it]
        else null
    }
    numbers.forEachIndexed { index, number →
        if (number != null) {
            val position = Position(index % 4, index / 4)
            val newId = createNewBlock(number, position)
            map[position.x, position.y] = newId
        }
    }
}
```

También habrá que actualizar dentro del método **main** la creación del tablero al lanzar el juego, comprobando si es un juego nuevo o debe cargarse una partida sin terminar desde el histórico.

```
...
val undoBlock = container {
    ...
}

if (!history.isEmpty())
    restoreField(history.currentElement)
else generateBlock()

// Evento de teclado.
keys {
    down {
    ...
```

En el método **restart()** también se limpiará el histórico.

```
fun Container.restart() {
    map = PositionMap()
    blocks.values.forEach { it.removeFromParent() }
    blocks.clear()
    score.update(0)
    history.clear()
    generateBlock()
}
```

Para finalizar, para tener siempre guardado el histórico cada vez que se genere un movimiento, deberá actualizarse el historial cada vez que se genere un bloque. De esta forma, siempre se guardará el estado cada vez que el usuario haga un movimiento, se fusionen bloques y abra o reinicie el juego.

Se renombrará el método **generateBlock()** por **generateBlockAndSave()**, y se añadirá una nueva línea al método.

```
fun Container.generateBlockAndSave() {
    val position = map.getRandomFreePosition() ?: return
    val number = if (Random.nextDouble() < 0.9) Number.ZERO else Number.ONE
    val newId = createNewBlock(number, position)
    map[position.x, position.y] = newId
    history.add(map.toNumberIds(), score.value)
}
```

El método **toNumberIds()** será un nuevo método de la clase **PositionMap**, encargado de devolver un array con los ids de los números colocados.

```
class PositionMap(...) {
    ...

    fun toNumberIds() = IntArray(16) { getNumber(it % 4, it / 4) }
}
```

Llegados a este punto, ya dispones de un juego 2048 totalmente funcional y listo para disfrutar de unas partidas.