



ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS

**Juego tipo "videoaventura de plataformas" para
móviles con J2ME**

MÁLAGA, SEPTIEMBRE 2005

ESCUELA TÉCNICA SUPERIOR
DE INGENIERÍA INFORMÁTICA

INGENIERO TÉCNICO EN INFORMÁTICA DE SISTEMAS

**JUEGO TIPO "VIDEOAVENTURA DE PLATAFORMAS" PARA
MÓVILES CON J2ME**

Realizado por

DANIEL RODRÍGUEZ CASTILLO

Dirigido por

FRANCISCO ROMÁN VILLATORO MACHUCA

Departamento

**DEPARTAMENTO DE LENGUAJES Y CIENCIAS DE LA
COMPUTACIÓN**

UNIVERSIDAD DE MÁLAGA

MÁLAGA, SEPTIEMBRE 2005

UNIVERSIDAD DE MÁLAGA

**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
INFORMÁTICA**

INGENIERO TÉCNICO EN INFORMÁTICA DE SISTEMAS

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente D^o;/D^a. _____

Secretario D^o;/D^a. _____

Vocal D^o;/D^a. _____

para juzgar el proyecto Fin de Carrera titulado:

**JUEGO TIPO "VIDEOAVENTURA DE PLATAFORMAS" PARA MÓVILES CON
J2ME**

del alumno D^o;/D^a. **Daniel Rodríguez Castillo**

dirigido por D^o;/D^a. **Francisco Román Villatoro Machuca**

ACORDÓ POR _____ OTORGAR LA CALIFICACIÓN DE

**Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS
COMPARECIENTES DEL TRIBUNAL, LA PRESENTE DILIGENCIA**

Málaga, a _____ de _____ de _____

El Presidente _____ El Secretario _____ El Vocal _____

Fdo: _____ Fdo: _____ Fdo: _____

Índice

Capítulo 1. Introducción.....1

1.1. Objetivos del proyecto.....	1
1.1.1. Antecedentes.....	1
1.1.2. Objetivos.....	3
1.1.3. Metodologías y fases de trabajo.....	3
1.2. Contenidos de la memoria.....	3

Parte 1: Programación de juegos en J2ME.....7

Capítulo 2. Introducción a J2ME.....9

2.1. ¿Qué es J2ME?.....	9
2.2. ¿Qué es un MIDlet?.....	10

Capítulo 3. Interfaces de usuario.....13

3.1. Introducción.....	13
3.2. Interfaz de usuario de alto nivel.....	14
3.2.1. La clase Alert.....	15
3.2.2. La clase List.....	16
3.2.3. La clase TextBox.....	17
3.2.4. La clase Form.....	17
3.2.4.1. La clase TextField.....	19
3.3. Interfaz de usuario de bajo nivel.....	20
3.3.1. Primitivas gráficas. Colores.....	20
3.3.2. Primitivas gráficas. Primitivas.....	21
3.3.3. Primitivas gráficas. Texto.....	21
3.3.4. Primitivas gráficas. Imágenes.....	23

Capítulo 4. Animación del personaje.....25

4.1. Sprites.....	25
4.2. Lectura de teclado.....	26
4.3. Threads.....	27
4.4. Game loop.....	28
4.5. Inteligencia artificial (IA).....	29

Capítulo 5. Record Management System (RMS).....31

Capítulo 6. Sonido y Música.....33

6.1. Sonido.....33

6.2. Música.....33

Parte 2: Juego Cómeme.....35

Capítulo 7. Introducción al juego Cómeme.....37

7.1. La historia de Cómeme.....37

7.2. Tipo de juego.....37

7.3. Objetivos del juego.....38

Capítulo 8. Mapa del juego.....39

8.1. Pantallas. Bloques gráficos y bloques de decorados.....39

8.2. La clase Sprite.....39

8.3. La clase Bloque.....43

8.4. La clase Bloque en el Canvas.....46

8.4.1. Tabla de bloques de decorados.....46

8.4.2. Inicialización de la tabla de bloques de decorados.....46

8.4.3. Generación de pantallas.....48

8.4.4. Pintando la pantalla.....50

8.5. El mapa de pantalla.....51

Capítulo 9. Los bichos.....55

9.1. Tipos de bichos.....55

9.2. La clase Bicho.....55

9.3. La clase Bicho en el Canvas.....59

9.3.1. Tabla de bichos.....59

9.3.2. Inicialización de la tabla de bichos.....60

9.3.3. Generación de bichos.....63

9.3.4. Pintando los bichos.....65

Capítulo 10. Los Objetos.....67

10.1. Tipos de objetos.....67

10.2. La clase Objeto.....67

10.3. La clase Objeto en el Canvas.....68

10.3.1. Tablas de objetos.....68

10.3.1.1. pantallas.....68

10.3.1.2. pantallasAsignadas.....68

10.3.1.3. tablaPlatos.....69

10.3.1.4. tablaGalletas.....	69
10.3.1.5. tablaVidasExtras.....	69
10.3.2. Inicialización de las tablas de objetos.....	69
10.3.3. Generación de objetos.....	73
10.3.4. Pintando los objetos.....	75
Capítulo 11. El personaje (JaimeNu).....	77
11.1. La clase JaimeNu.....	77
11.2. La clase JaimeNu en el Canvas.....	79
11.2.1. Inicialización del personaje.....	79
11.2.2. Actualización del personaje.....	81
11.2.3. Pintando al personaje.....	100
Capítulo 12. Pié de pantalla.....	103
12.1. Inicialización del marcador.....	103
12.2. Pintando el marcador.....	104
Capítulo 13. Scroll, teclas y otros.....	107
13.1. Scroll.....	107
13.2. Interfaz con el jugador (teclas).....	107
13.3. Colisiones entre objetos.....	111
13.4. Restauración de pantalla.....	114
Capítulo 14. El menú.....	115
14.1. La clase Comeme.....	115
14.2. La clase Menu.....	119
14.3. La clase Dificultad.....	120
14.4. La clase Resultado y la clase NuevoRecord.....	121
14.4.1. La clase NuevoRecord.....	121
14.4.2. La clase Resultado.....	123
14.5. La clase Musica.....	127
14.6. La clase Alerta.....	128
Capítulo 15: La música.....	131
15.1. La clase EfectosDeSonido.....	131
15.2. La clase EfectosDeSonido en el Canvas.....	133
Capítulo 16. Conclusiones.....	135
16.1. Conclusiones finales.....	135

16.2. Herramientas utilizadas.....	136
16.3. Líneas futuras.....	136

Apéndices.....139

Apéndice A. Listados de pantallas y bichos.....141

Apéndice B. Manual de usuario.....143

B.1. Paseo por el menú.....	143
B.2. Los movimientos de JaimeNu.....	146
B.3. Los objetos de Cómemme.....	147

Apéndice C. Contenidos del CD y manual de instalación.....149

C.1. Contenidos del CD.....	149
C.2. Manual de instalación.....	150
C.2.1. Instalación de herramientas y ejecución en un PC.....	150
C.2.2. Transferencia de la aplicación a un dispositivo móvil.....	152

Apéndice D. Mapa de Cómemme.....155

Bibliografía y Referencias.....157

CAPÍTULO 1: INTRODUCCIÓN

1.1. Objetivos del proyecto

1.1.1. Antecedentes

La programación de aplicaciones para teléfonos móviles compatibles con Java no reviste especiales dificultades más allá de las propias limitaciones de la máquina virtual de Java que las ejecuta: poca memoria y pantalla de reducidas dimensiones. Una de las aplicaciones más interesantes en dicho entorno son los juegos de ordenador. Éstos deben ser similares a los primeros juegos de ordenador que se desarrollaron para microordenadores a principio de los años 1980 ya que estaban sujetos a las mismas restricciones. Los juegos para ZX-81 (16 Kb de memoria y pantalla en blanco y negro), Spectrum (48 Kb y con 16 colores), y más tarde para Commodore, MSX, Amiga, Atari, etc., que tenían mayores capacidades de memoria y gráficos, y las técnicas de programación utilizadas para su desarrollo, se pueden extender fácilmente al entorno de la programación de teléfonos móviles.

Los juegos de videoaventura (*arcade*) basadas en plataformas tuvieron mucho éxito durante los 1980s (*Manic Miner*, *Jet Set Willy*, *Profanation*, *Phantom*, etc., y más recientemente los famosos *Mario Bros.*). Se basan en un personaje plano que se mueve en un mapa de pantallas distribuidas verticalmente, caminando horizontalmente de izquierda a derecha, subiendo y bajando por escaleras y plataformas, y saltando entre plataformas cuando sea necesario. El objetivo del juego es encontrar una serie de objetos que recoger o enemigos a los que vencer. Durante el transcurso del juego hay que evitar y/o luchar con múltiples enemigos, algunos situados en determinadas habitaciones con movimientos fijos que hay que evitar y otros que nos persiguen. Asimismo hay habitaciones ocultas o cerradas que requieren llaves u conjuros para poder acceder a ellas.

Para un programador novel las mayores dificultades a la hora de realizar un juego de estas características no son de programación: la elaboración de la idea (historia) en la que se basará el juego, el diseño gráfico de las pantallas, de los personajes y sus movimientos, la composición del sonido y la música, y la selección de las acciones necesarias para superar cada pantalla o concluir el juego. Con objeto de reducir al mínimo todas estas dificultades, dejando sólo las propias de la programación del mismo, la mejor opción es “copiar” un juego ya existente. Para evitar todo tipo de conflictos con derechos comerciales hemos decidido “copiar” el juego “Cómeme”, desarrollado por los programadores de la compañía española Dinamic para la sección “Aprende a programar tu propio juego” escrita por Pablo Ariza y publicada en diez números de la revista MicroHobby, revista independiente para usuarios de ordenadores Sinclair, programado para un ZX Spectrum con sólo 16 Kb de memoria [3-12]. Los códigos presentados en dicha revista están en formato hexadecimal representando código máquina para el procesador de 8 bits Z-80 (Zilog), luego no pueden ser re-utilizados en este proyecto y sus algoritmos deberán ser reprogramados en J2ME.

En “Cómeme”, el gran magnate Jaime Nu, que se gasta todo lo que gana en su pequeño y único vicio, comer, está sumergido en una horrible pesadilla en la que tiene que comerse un pequeño banquete de diez platos antes de que el resto de los alimentos comestibles se le coman a él [3].

El mapa del juego está formado por 36 pantallas (6x6) formadas por sólo 14 bloques gráficos, todos ellos fácilmente “escaneables” de la propia revista [3]. Cada pantalla está constituida por 20 filas de 32 bloques gráficos que se agrupan en combinaciones denominadas “decorados”: repeticiones de bloques consecutivos. El mapa completo del juego ocupa 640 bytes. Cada decorado se almacena con 4 bytes, sus dos coordenadas, color, código de bloque de decorado, y longitud de repetición. El juego utiliza un total de 20 bloques de decorados distintos, cada uno almacenado con 10 bytes. En el juego a desarrollar se utilizará la misma codificación utilizada en el juego “Cómeme”, ya que es muy compacta y los valores de sus bytes (en hexadecimal) puede ser “escaneada” de la revista con un OCR [4].

Los “enemigos” del juego serán gráficos de 16x16 píxeles con un solo color y animados con un recorrido constante de ida y vuelta. Cada pantalla tendrá sus propios “enemigos”. Cada uno se almacena con 5 bytes que indican su posición, velocidad y tipo de movimiento [5]. El juego tiene 4 melodías bitonales (con dos notas simultáneas) que se escucharán durante el mismo almacenadas como grupos 2 bytes por nota y su duración [5]. Estas músicas se podrán reproducir tanto en un móvil monotonó (simulando las dos notas con un fichero .wav adecuado) como multitono (directamente).

Para el dibujo y movimiento del personaje y de los “enemigos” evitando posibles parpadeos, el juego “Cómeme” utiliza rutinas basadas en interrupciones que le permiten sincronizar el movimiento de los personajes con el refresco de la pantalla y evitar conflictos entre el acceso a la memoria de video por el procesador y el DMA del ZX Spectrum [6]. Sin embargo, estos detalles técnicos no serán necesarios durante la programación del juego para un móvil, para lo que usaremos directamente una rutina de manejo de *sprites* estándar [0]. En cualquier caso, los detalles del (tipo de) movimiento de los “enemigos” serán los mismos que los usados en “Cómeme” [6].

La pantalla de presentación del juego, así como la fase de inicialización del mismo, que incluye la situación aleatoria de los objetos especiales (los 10 platos a buscar, las vidas extra, y las galletas de la suerte), codificados con 5 bytes asociados a cada una de las 36 pantallas, se presentan en el quinto artículo de la serie [7]. El bucle principal del juego, que utiliza programación orientada a eventos, que incluye la lectura del teclado, la gestión de la puntuación, de las vidas, etc., se detalla en el artículo [8].

Una de las rutinas más importantes del juego es la relativa al movimiento de nuestro personaje, que Pablo Ariza nos presenta en ensamblador del Z-80 en el artículo séptimo [9], y nos comenta con todo lujo de detalles en tres artículos sucesivos [9-11]. Los detalles finales relativos a lo que ocurrirá cuando el juego finaliza tras haber recogido las diez comidas, así como algunos consejos sobre cómo jugar aparecen en décimo y último artículo de la serie [12].

1.1.2. Objetivos

Desarrollar un juego de ordenador para teléfonos móviles de tipo videoaventura de plataforma utilizando el lenguaje de programación y la tecnología J2ME (Java 2 Micro Edition) [0]. Se implementará el juego “Cómeme” publicado en la revista MicroHobby para ZX Spectrum, cuyos gráficos, personajes, mapa, y sonido serán *escaneados* directamente a partir de la revista, en la que aparecen en código hexadecimal [3-12].

1.1.3. Metodología y fases de trabajo

La presentación del juego “Cómeme” realizada por Pablo Ariza en su serie de diez artículos en MicroHobby [3-12] es bastante lógica y puede ser utilizada como metodología para el desarrollo de este proyecto fin de carrera. En cada artículo se expone una fase de desarrollo del juego, incluyendo códigos en hexadecimal, a veces también en ensamblador del Z-80, de cada una de las rutinas, junto con una explicación detallada y suficiente de todas las estructuras de datos y algoritmos utilizados. Así que podemos resumir las fases del trabajo como:

- 1) Aprendizaje de la programación en J2ME con énfasis en el manejo de gráficos (objeto *Canvas*), gráficos animados (*sprites*), control de teclado y generación de sonido [0,1,14,15,16].
- 2) Estudio de los diez artículos que describen y explican el juego “Cómeme” publicado en MicroHobby [3-12].
- 3) “Escaneo” de todos los gráficos del juego y de su música.
- 4) Implementación de todos los algoritmos del juego con Java (J2ME).
- 5) Prueba del juego y mejora de los detalles necesarios para facilitar su “jugabilidad”.
- 6) Análisis del juego resultante, y exposición de los problemas y conclusiones oportunas sobre las técnicas de programación de juegos en el entorno de teléfonos móviles utilizando J2ME como lenguaje.

1.2. Contenidos de la memoria

Esta memoria está dividida en dos partes:

Parte 1: Programación de juegos en J2ME, la cual consta de la explicación de todos los métodos [16] y conocimientos de este lenguaje que hemos utilizado en nuestro proyecto [0,1]. Incluye los siguientes capítulos:

- **Capítulo 2. Introducción a J2ME.** Se explica brevemente qué es J2ME y qué es un *midlet*.
- **Capítulo 3. Interfaces de usuario.** Se describen los dos tipos de interfaces que vamos a utilizar, tanto la de alto nivel como la de bajo nivel.
- **Capítulo 4. Animación del personaje.** Este capítulo, como su propio nombre

indica, está dedicado a la animación de los *sprites* en general, dando también una noción sobre la inteligencia artificial de nuestros enemigos.

- **Capítulo 5. Record Managent Sistema (RMS).** Explicamos en qué consisten los *rms* de J2ME y los métodos necesarios para la creación y utilización de estos.
- **Capítulo 6. Sonido y Música.** Mostramos los distintos métodos para generar tanto sonidos como música en J2ME.

Parte 2: Juego Cómeme, en la que explicamos todo lo relacionado con nuestra aplicación (mapa, bichos, objetos...). Explicamos las partes de nuestro código que creemos que son más importantes: variables globales, tablas, clases y métodos, con una serie de figuras para que el lector pueda comprenderlo con la mayor facilidad posible. Incluye los siguientes capítulos:

- **Capítulo 7. Introducción al juego Cómeme.** Comentamos la historia, el tipo y los objetivos de nuestro juego.
- **Capítulo 8. Mapa del juego.** Damos unas definiciones que vamos a utilizar durante toda la memoria, y explicamos las clases y estructuras para la generación de nuestras pantallas.
- **Capítulo 9. Los bichos.** Vemos los distintos tipos de bichos que nos encontramos en nuestro juego, y además explicamos la clase `Bicho` y las estructuras utilizadas para la generación de estos.
- **Capítulo 10. Los objetos.** Igual que el capítulo anterior pero referido en este caso a los objetos.
- **Capítulo 11. El personaje (JaimeNu).** Igual que los dos capítulos anteriores pero referente a nuestro personaje principal.
- **Capítulo 12. Pié de pantalla.** Explicamos como creamos nuestro marcador (puntuación, indicador de vidas y nombre de pantalla).
- **Capítulo 13. Scroll, teclas y otros.** Explicamos todo lo relacionado con nuestro *scroll* de pantalla, la interfaz con el usuario (teclas), las colisiones entre objetos y la forma en que restauramos las pantallas al cambiar de una a otra o al perder una vida.
- **Capítulo 14. El menú.** Describimos las clases que forman nuestro menú principal, así como la pantalla en la que introducimos nuestro nombre tras lograr un nuevo récord.
- **Capítulo 15. La música.** Vemos la clase que nos permite escuchar la música en todo momento.
- **Capítulo 16. Conclusiones.** En este capítulo damos las conclusiones finales a las que hemos llegado tras finalizar el proyecto. Además damos las herramientas principales utilizadas y unas líneas futuras dedicadas a mejorar esta aplicación.

Finalmente nos encontramos con la bibliografía pertinente y tres apéndices:

Apéndice A: Explicamos, con un ejemplo, como sacamos los datos de todas las pantallas y bichos a partir de un listado hexadecimal [3-12] que encontramos en la revista MicroHobby [3-13].

Apéndice B: Contiene un manual de usuario donde explicamos todas las opciones y

posibilidades de nuestra aplicación.

Apéndice C: Este apéndice está compuesto por los contenidos del CD-ROM que acompaña a la memoria y por un manual de instalación en el que explicamos como instalar nuestra aplicación en un dispositivo móvil [0].

Apéndice D: Mostramos en dos figuras (Figura D.1. Y Figura D.2.) el mapa completo de nuestro juego.

PARTE 1

Programación de juegos en J2ME

CAPÍTULO 2: INTRODUCCIÓN A J2ME

2.1. ¿Qué es J2ME?

Cuando Sun creó el paquete Java 2 lo dividió en tres. Esta división está pensada para satisfacer a los distintos tipos de usuarios. La división es la siguiente:

- J2SE (Java Standard Edition), orientada al desarrollo de aplicaciones independientes de la plataforma, algo así como la base de la tecnología Java.
- J2EE (Java Enterprise Edition), creada para el entorno empresarial.
- J2ME (Java Microedition Edition), para dispositivos de poca capacidad de cómputo y de memoria (ver Figura 2.1).

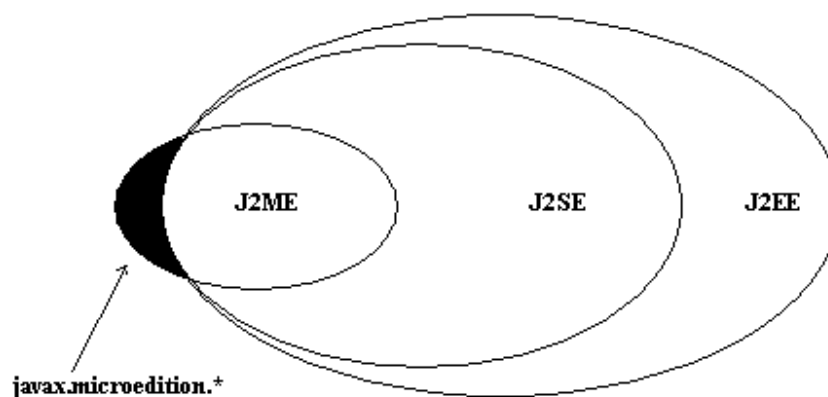


Figura 2.1. Relación entre las diferentes API's de Java 2

Las principales características de J2ME son:

- Una máquina virtual diferente a la de J2SE llamada KVM (Kilo Virtual Machine) mucho más pequeña que la JVM (Java Virtual Machine). Esto es debido principalmente a las restricciones que nos encontramos en los dispositivos móviles a los que va dirigido J2ME.
- Se basa en configuraciones (característica mínima de configuración hardware y software). La que utiliza J2ME para dispositivos de poca memoria y procesamiento es la CLDC (Connected Limited Device Configuration) que define:
 - 1.Las características del lenguaje Java incluidas.
 - 2.La funcionalidad que será incluida en la máquina virtual.

3.Las API's a utilizar.

4.Los requerimientos hardware de los dispositivos.

- Se basa en bibliotecas Java de alto nivel para dispositivos específicos, denominadas perfiles.

Como conclusión decir que en todo entorno de ejecución de J2ME nos encontraremos:

- Máquina Virtual.
- Configuraciones.
- Perfiles.
- Paquetes Opcionales.

2.2. ¿Qué es un MIDlet?

Un *midlet* es para J2ME lo que un *applet* para J2SE. Al igual que un *applet*, un *midlet* no tiene método `main()`. Un *applet* es subclase de la clase `Applet`, mientras que un *midlet* lo es de la clase `MIDlet`. Ambos tienen que implementar unos métodos concretos: `init()`, `start()`, `stop()`, `destroy()`, en el caso de un *applet* y `startApp()`, `pauseApp()`, `destroyApp()`, en el de un *midlet*. Y mientras un *applet* tiene que ser ejecutado sobre un navegador *web*, un *midlet* lo tiene que hacer en un dispositivo con soporte J2ME (teléfonos móviles, PDA's,...).

Un *midlet*, cuando está ejecutándose, puede estar en tres estados diferentes:

- Activo, si se está ejecutando. Ocupa en memoria principal todos los recursos que necesite. Puede pasar al estado de pausa, llamando al método `MIDlet.pauseApp()`, o al estado destruido, llamando al método `MIDlet.destroyApp()`.
- Pausa, no está actualmente en ejecución. En este estado el *midlet* no utiliza ningún recurso compartido. Para volver a estar en ejecución, estado activo, tiene que hacer una llamada al método `MIDlet.startApp()`. Desde este estado también se puede pasar al estado destruido.
- Destruído, se liberan todos los recursos ocupados por el *midlet*. A este estado se pasará si se finaliza la ejecución del *midlet* o si se necesitan los recursos ocupados por éste para una aplicación de mayor prioridad (ver Figura 2.2).

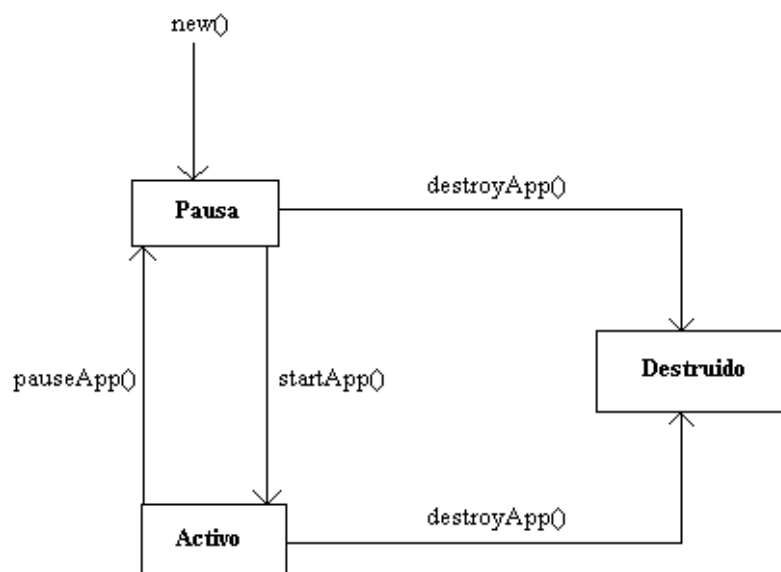


Figura 2.2. Estados de un midlet en ejecución.

CAPÍTULO 3: INTERFACES DE USUARIO

3.1. Introducción

La API que nos permite crear interfaces de usuario en J2ME es MIDP. Esta API nos suministra los siguientes paquetes (exclusivos de J2ME):

- `javax.microedition.midlet`
- `javax.microedition.lcdui`
- `javax.microedition.io`
- `javax.microedition.rms`

El paquete más importante de todos es `javax.microedition.midlet`, el cual nos provee una única clase, `MIDlet`, que nos permite la ejecución de aplicaciones en dispositivos móviles.

El paquete `javax.microedition.lcdui` nos suministra una serie de clases e interfaces que nos permitirán crear las interfaces de usuarios (tanto en alto como en bajo nivel). Es similar al paquete `AWT` de J2SE pero muy reducido. Este paquete nos permite trabajar en alto nivel, con *screens*, creando menús, editores de texto,... y en bajo nivel, *canvas* (trabajamos a nivel gráfico).

Ambas clases, `Screen` y `Canvas`, heredan de la clase `Displayable` (ver Figura 3.1.).

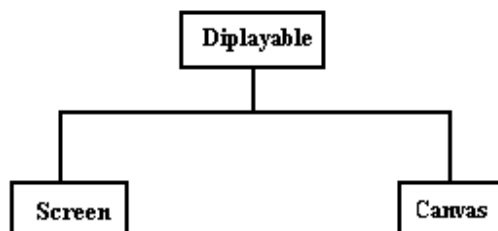


Figura 3.1. Relación de clases.

Todo lo que mostremos por la pantalla del dispositivo hereda de `Displayable`. Veamos unos métodos de esta clase utilizados en el proyecto:

<code>public void addCommand (Command comando)</code>
<code>public int getHeight ()</code>
<code>public int getWidth ()</code>

El primer método coloca en pantalla un comando. El segundo y el tercero nos devuelve la altura y anchura de nuestra pantalla. Y con el tercero indicamos que clase va a ser la que atienda los eventos de comandos.

Para este proyecto, y en general para el desarrollo de videojuegos, el objeto `Canvas` es el más importante, quedando el `Screen` para menús y cosas similares.

3.2. Interfaz de usuario de alto nivel

Para crear interfaces de alto nivel, J2ME nos aporta cuatro clases. Estas clases heredan de `Screen` y son las siguientes: `Alert`, `Form`, `List`, `TextBox` (ver Figura 3.2.).

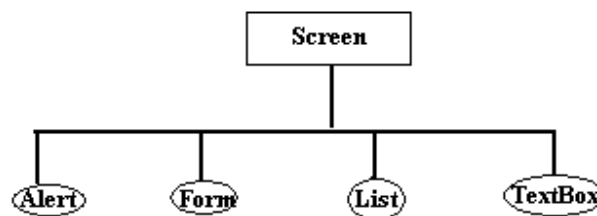


Figura 3.2. Interfaces de alto nivel.

Un *midlet* puede estar compuesto de varios elementos de este tipo, pero sólo podemos mostrar uno cada vez ya que ocupan la pantalla entera. Para cambiar de un elemento a otro utilizamos el método `setCurrent()` de la clase `Display`.

```
void setCurrent (Displayable siguienteInterfaz)
```

El único parámetro de ésta función es el interfaz al que queremos cambiar (ver Figura 3.3).

En nuestro proyecto en particular sólo vamos a utilizar las clases `Alert`, `Form` y `List`, por lo que nos explayaremos más en éstas, dando sólo una breve explicación de la clase `TextBox`.

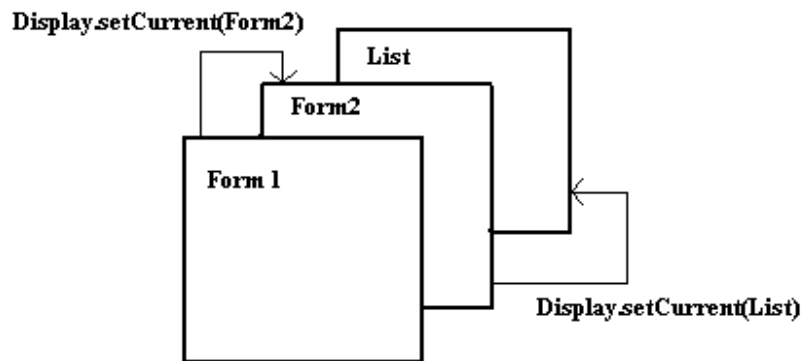


Figura 3.3. Cambio de interfaz.

3.2.1. La clase Alert

Esta clase nos permite mostrar información por pantalla al usuario, normalmente de errores, mediante un texto o imagen, o ambas, durante un tiempo determinado o hasta que se produzca un evento de comando. El constructor de la clase `Alert` es el siguiente:

```
Alert (String titulo)
```

El único parámetro es el nombre de la alerta, el cual se mostrará en la cabecera de la pantalla. Para asignar un texto y una imagen a nuestra alerta utilizaremos los siguientes métodos respectivamente:

```
public void setImage (Image img)
```

```
public void setString (String str)
```

Para asignar un tipo de alarma, sólo se diferencian en el tipo de sonido que emiten, utilizamos el siguiente método:

```
public void setType (AlertType tipo)
```

Cuyo argumento puede ser una de las siguientes constantes:

- `ALARM`
- `CONFIRMATION`
- `ERROR`
- `INFO`
- `WARNING`

Si queremos mostrar la alerta durante un determinado tiempo, le pasaremos como parámetro, al método `setTimeout(int tiempo)`, el tiempo en milisegundos. Si lo que queremos es que se muestre la alerta hasta que se presione un determinado comando

utilizaremos la constante `FOREVER` (ver Figura 3.4.).

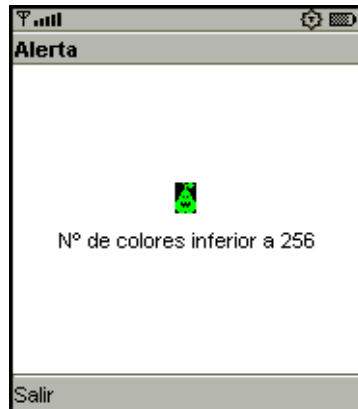


Figura 3.4. Ejemplo de la clase `Alert`.

3.2.2. La clase `List`

Con esta clase podemos realizar listas de opciones para la creación de menús. Su constructor es el que se muestra a continuación:

```
List (String titulo, int tipoLista)
```

El primer parámetro es el mismo que para la clase `Alert` anteriormente explicada. El segundo es el tipo de lista a crear, a elegir entre las siguientes constantes:

- `EXCLUSIVE`: Sólo se puede seleccionar un elemento.
- `IMPLICIT`: Se selecciona el elemento que está marcado.
- `MULTIPLE`: Permite la selección múltiple.

Para los dos primeros tipos de listas anteriores, los que vamos a utilizar en nuestro proyecto, tenemos un método que nos permite saber cuál ha sido la opción elegida por el usuario. Este método es `getSelectedIndex()`, que nos devuelve un entero, siendo este valor, comenzando por cero, la posición del valor elegido. Si este valor lo pasamos como parámetro al método `getString()`, nos devolverá el texto de la opción elegida.

Para añadir opciones a la lista nos encontramos con el método `append(...)`:

```
public int append (String nombreOpcion, Image imagenOpcion)
```

Añade un elemento a la lista, siendo el primer parámetro el texto y el segundo una imagen que acompaña a éste (si no queremos imagen le pasaremos el valor `null`).

Para borrar una opción de la lista tenemos el método `delete(...)`:

```
public void delete (int opcion)
```

El único argumento a pasarle como parámetro es la opción de la lista que queremos borrar.

Otros métodos que vamos a utilizar son: `setTitle (String nombre)` para cambiar el nombre de la lista, `size()` que nos devuelve el número de opciones de la lista y `setSelectedIndex(int opción, boolean seleccionada)` que nos permite seleccionar la opción dada como argumento en el primer parámetro si colocamos el segundo como `true` (ver Figura 3.5).

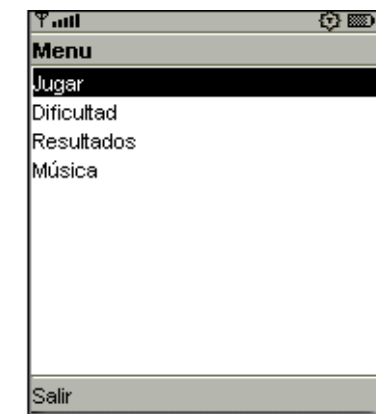


Figura 3.5. Ejemplo de la clase List.

3.2.3. La clase TextBox

Como esta clase no la vamos a utilizar en nuestro proyecto, utilizamos una similar que veremos próximamente dentro de la clase `Form` (ver apartado 3.2.4.1), sólo diremos que esta interfaz básicamente nos permite mostrar por pantalla un pequeño editor de texto.

3.2.4. La clase Form

Con la clase `Form` podemos realizar interfaces de usuario más complejas que las anteriores. Esta clase es de tipo contenedor, por lo que podemos incluir más de un elemento en pantalla. El constructor con el que crearemos un *form* vacío es el siguiente:

```
Form (String titulo)
```

Los posible elementos que puede contener un *form* son:

- StringItem
- ImageItem
- TextField
- DateField
- ChoiceGroup
- Gauge

En la figura 3.6. podemos ver de una forma global como se relacionan todas las clases de la rama de alto nivel de la clase `Displayable`.

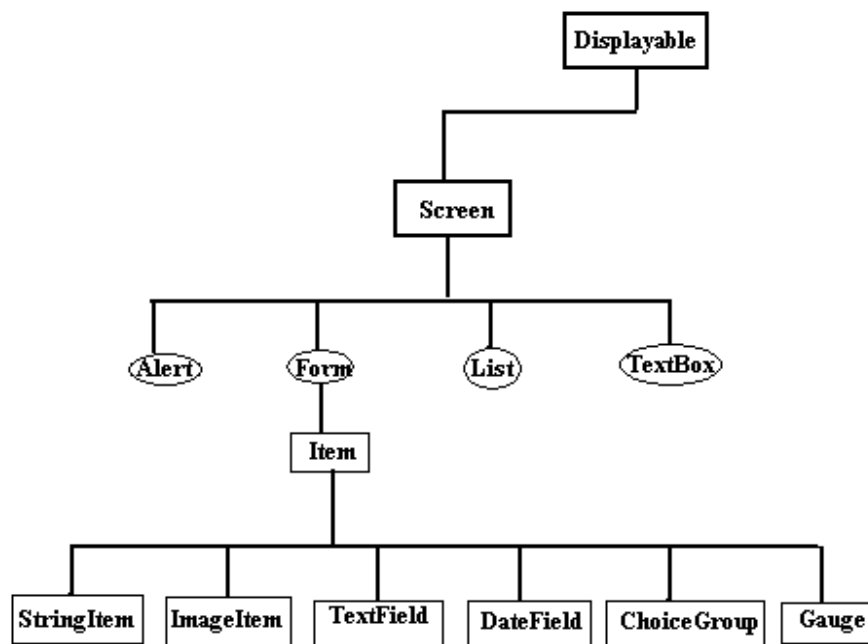


Figura 3.6. Relación completa de la rama de alto nivel de `Displayable`.

Todos estos elementos heredan de la clase `Item` y éste a su vez de la clase `Form`. Un *item* es un elemento visual que no ocupa toda la pantalla, por lo que nos permite una mayor variedad a la hora de hacer interfaces de usuario. En nuestro proyecto el único elemento que vamos a utilizar de la clase `Item` es `TextField`.

Los métodos principales para la creación de interfaces de la clase `Form` son:

```
int append (Item elemento)
```

Con este método añadimos un nuevo elemento al formulario. Nos devuelve, si todo va bien, el índice que se le ha asignado a dicho elemento dentro del formulario (al primero se le asigna el valor 0). Este es el único método, de la clase `Form`, que vamos a utilizar.

Otros métodos interesantes, aunque no los utilizemos en el proyecto, son:

```
int size ()
```

Este método nos permite saber en todo instante cuántos elementos hay dentro de un formulario.

```
void delete (int indice)
```

Este método elimina un elemento del formulario. Este elemento será el indicado por el índice, único parámetro.

```
void insert (int indice, Item elemento)
```

Nos permite introducir un elemento al formulario en la posición deseada.

```
void set (int indice, Item elemento)
```

Nos permite sustituir un elemento por otro en el formulario.

3.2.4.1. La clase **TextField**

Ésta clase es similar a la vista con anterioridad `TextBox`. La principal diferencia es que esta no ocupa toda la pantalla. El constructor sería de la siguiente forma:

```
TextField (String etiqueta, String texto, int tamañoMax, int limitacion)
```

El primer parámetro es la etiqueta del campo de texto, como nombre, teléfono, dirección, etc. El segundo y tercero son, respectivamente, el cuerpo y el tamaño máximo del texto a escribir. Y el cuarto y último, son las restricciones a las que sometemos al texto (sólo correo electrónico, sólo número, etc). Ver figura 3.7. para un ejemplo gráfico utilizado en el proyecto.



Figura 3.7. Ejemplo de TextField.

3.3. Interfaz de usuario de bajo nivel

Para la realización de juegos necesitamos algo más, algo que nos permita pintar a más bajo nivel, a nivel de píxel. Esto es lo que nos ofrece la clase *Canvas*. Las operaciones que nos permite ésta clase, y que vamos a utilizar en nuestro proyecto, son las siguientes:

- Dibujar primitivas gráficas (puntos, rectas,...)
- Escribir texto.
- Dibujar imágenes (PNG, BMP,...)

En el *canvas*, cada vez que necesitamos pintar o repintar algo, llamamos con los métodos `repaint()` y `serviceRepaints()` al código donde se encuentra todas las instrucciones que se encargan de dibujar en pantalla. Éste código se encuentra en el método:

```
public paint (Graphics g)
```

Al parámetro *g* se le denomina contexto gráfico, es de tipo *Graphics* y nos suministra las primitivas necesarias para poder pintar por pantalla.

3.3.1. Primitivas gráficas. Colores

Para poder formar los distintos colores necesitamos una gama de colores básicos. Dependiendo de las aportaciones de cada color básico obtenemos un color diferente. En informática, y en nuestro caso en particular, se utilizan como colores básicos el rojo, verde y azul o lo que también conocemos como RGB (Red, Green, Blue). Algunos ejemplos se muestran en la tabla 3.1.

RGB	Color generado
255, 0, 0	Rojo
0, 255, 0	Verde
0, 0, 255	Azul
128, 0, 0	Rojo oscuro
255, 255, 0	Amarillo
0, 0, 0	Negro
255, 255, 255	Blanco
128, 128, 128	Gris

Tabla 3.1. Colores.

La clase `Graphics` nos suministra el método `setColor()` para poder utilizar los distintos colores.

```
void setColor (int rojo, int verde, int azul)
```

Los tres parámetros son la gama de los distintos colores básicos. Estos parámetros tienen el rango `[0 , 255]`.

3.3.2. Primitivas Gráficas. Primitivas

Aunque no se utilizan mucho para la creación de juegos, en nuestro proyecto utilizamos un método que nos permite pintar la pantalla del color anteriormente asignado con `setColor(...)`. El método es el siguiente:

```
void fillRect (int x, int y, int ancho, int alto)
```

Con esta primitiva podemos crear rectángulos rellenos con el color asignado anteriormente. Los dos primeros parámetros nos indican las esquina superior izquierda del rectángulo en el eje de coordenadas. Y el tercer y cuarto representan el ancho y el alto del rectángulo a pintar.

3.3.3. Primitivas Gráficas. Texto

Aunque no utilicemos texto con mucha frecuencia en nuestro proyecto (sólo lo utilizaremos para el nombre de cada pantalla, el marcador de vidas y la puntuación), podemos pintar texto en un *canvas*. El método que nos permite hacer esto es `drawString(...)`.

```
void drawString (String texto, int x, int y, int ancla)
```

Con el primer parámetro indicamos el texto que queremos escribir por pantalla. Con el segundo y tercero respectivamente, posicionamos el texto. Y con el último posicionamos los dos valores anteriores dentro del texto. Los valores posibles para nuestro último parámetro son:

- TOP
- BASELINE
- BUTTOM

para el eje vertical y:

- LEFT
- HCENTER
- RIGHT

para el eje horizontal (ver Figura 3.8.).

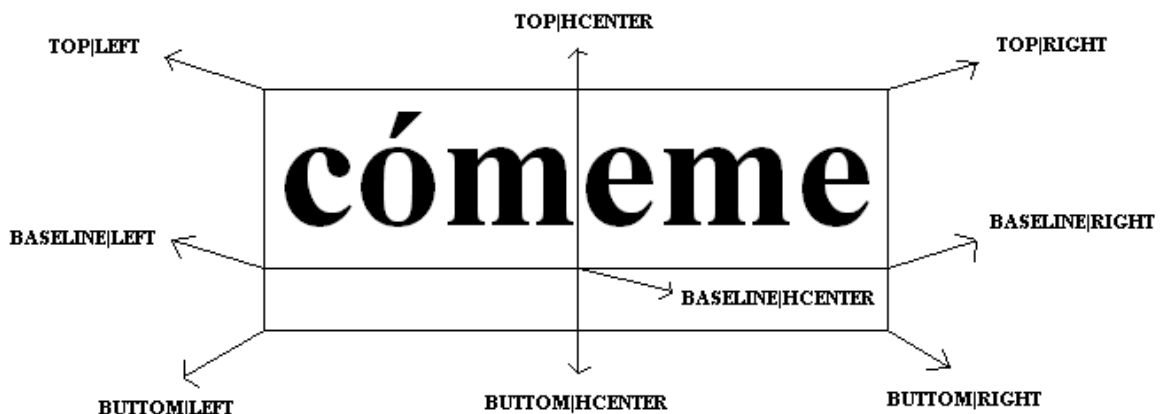


Figura 3.8. Tipos de ancla para un mismo texto.

Además de posicionar el texto dentro de la pantalla, tenemos la posibilidad de elegir entre distintas fuentes y tamaños de texto, además del espaciado entre palabras. Con el método `getFont(...)` obtenemos un objeto de tipo `Font` que se lo podemos pasar al método `setFont(...)` para aplicar el tipo de fuente deseada.

```
static Font getFont (int espaciado, int estilo, int tamaño)
void setfont (Font fuente)
```

Los valores posibles para los argumentos del método `getFont(...)` son los de la tabla 3.2.

Para pintar texto en nuestro proyecto hemos utilizado tamaño `SIZE_MEDIUM`, estilo `STYLE_BOLD` y espaciado `FACE_PROPORTIONAL`.

Tamaño	<code>SIZE_SMALL</code> , <code>SIZE_MEDIUM</code> , <code>SIZE_LARGE</code>
Estilo	<code>STYLE_PLAIN</code> , <code>STYLE_ITALICS</code> , <code>STYLE_BOLD</code> , <code>STYLE_UNDERLINED</code>

Tabla 3.2. Opciones de texto.

3.3.4. Primitivas gráficas. Imágenes

En nuestro proyecto para poder cargar imágenes vamos a utilizar el método `createImage(...)`.

```
public static Image createImage (String nombre) throws
IOException
```

El parámetro `nombre` es el directorio donde tenemos almacenada la imagen. Este directorio se dará a partir del directorio *res*, ya que todos los recursos irán almacenados en esta carpeta, tanto imágenes como sonido. El valor que nos devuelve es la imagen que hemos creado. Además de esto, este método lanza las siguientes excepciones que tendremos que capturar:

- `NullPointerException`, si el nombre es nulo.
- `IOException`, si la imagen no existe o no puede ser cargada o decodificada.

En un principio podemos cargar imágenes de tipo PNG, sin embargo dependiendo del dispositivo podremos cargar otros tipos de imágenes como las BMP.

Con el método anterior hemos cargado la imagen, pero todavía nos queda pintarla. Esto se hace con el método `drawImage(...)`.

```
public void drawImage (Image, int x, int y, int ancla)
```

El primer parámetro pasado es la imagen creada anteriormente con el método `createImage(...)`. El segundo, tercero y cuarto son similares a los explicados anteriormente para el dibujo de texto, con la única diferencia de los valores que puede tomar el argumento `ancla`. Estos son:

- `TOP`
- `VCENTER`
- `BOTTOM`

para el eje vertical y:

- LEFT
- HCENTER
- RIGHT

para el eje horizontal (ver Figura 3.9).

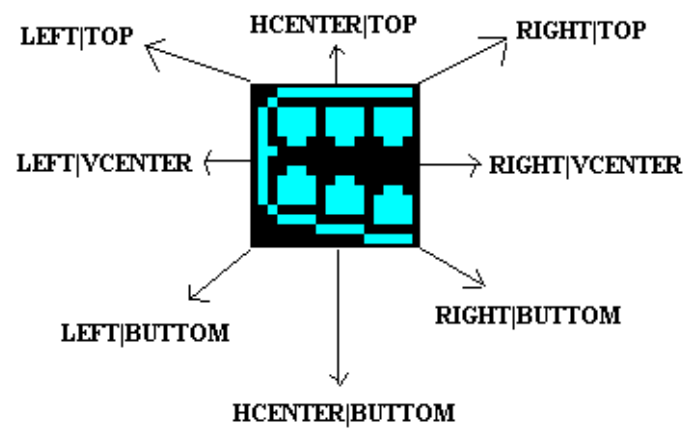


Figura 3.9. Tipos de ancla en una imagen.

CAPÍTULO 4: ANIMACIÓN DEL PERSONAJE

En este capítulo vamos a ver todo lo relacionado con la animación de los *sprites*. Primero veremos qué es un *sprite* y todo lo relacionado con él (posicionamiento, colisiones,...), luego veremos cómo dar órdenes a nuestro personaje, para que se mueva mediante la lectura de teclado, la definición de *thread* y para que lo vamos a utilizar, qué es un *game loop* y su forma general, y por último unas nociones de inteligencia artificial (la poca que van a tener nuestros bichos).

4.1. Sprites

Una de las partes más importantes en la creación de juegos para móviles, y para juegos en dos dimensiones (2D) en general, es la concerniente a los *sprites*. Un *sprite* lo podemos definir como un elemento gráfico, en nuestro caso una imagen *PNG*, que tiene entidad propia sobre el que podemos definir y modificar ciertos atributos, como el número de *frames* que forman el gráfico, posición de éste en el eje de coordenadas, etc.

En los *sprites* podemos diferenciar movimiento interno y movimiento externo. El primero es la animación en sí (por ejemplo nuestro personaje caminando o subiendo escaleras), mientras que el movimiento externo es la posición relativa del *sprite* en el eje de coordenadas o desplazamiento de éste.

El eje de coordenadas para J2ME y en general para gráficos por ordenador es el que mostramos en la figura 4.1.



Figura 4.1. Eje de coordenadas en J2ME.

En este eje de coordenadas los incrementos positivos del eje Y son hacia abajo en el eje vertical al contrario que en la habitual definición matemática, donde estos incrementos son hacia arriba en el eje vertical.

Además de poder controlar el movimiento interno y externo de nuestro *sprite*, podemos detectar colisiones entre estos de una manera fácil y muy eficiente aunque no del todo correcta. Diremos que dos *sprites* o imágenes han colisionado si el rectángulo que bordea nuestra imagen solapa el rectángulo de la otra imagen. Aunque esto no es totalmente cierto, es la manera más rápida y fácil y si minimizamos este rectángulo será cierto casi siempre (ver Figura 4.2).

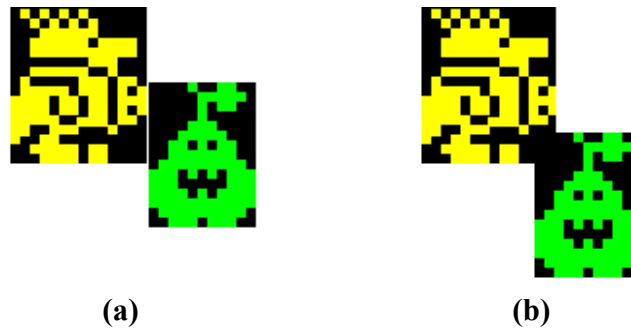


Figura 4.2. Ejemplos de colisiones.

En la primera imagen (a) no hay colisión, y nuestro método considera también que no la hay. Aunque en la segunda imagen (b), tampoco hay colisión y nuestro método la considera como tal ya que se solapan los recuadros que rodean a ambas imágenes.

4.2. Lectura de teclado

Para la lectura de teclado tenemos tres métodos heredados de la clase `Canvas` (si utilizáramos la clase `GameCanvas` sería de otra forma).

- `KeyPressed()`
- `KeyReleased()`
- `KeyRepeated()`

En nuestro juego sólo vamos a utilizar los dos primeros métodos. El método `KeyPressed()` es invocado al pulsar una tecla mientras que `KeyReleased()` lo es cuando la soltamos. `KeyRepeated()` es invocado de forma continuada mientras mantenemos una tecla pulsada. Este método, aunque parezca el más adecuado, no lo vamos a utilizar ya que no todos los dispositivos aceptan la auto repetición de teclas, incluido el emulador de Sun utilizado en este proyecto.

Estos métodos recogen como argumento un entero, cuyo valor es el código unicode de la tecla pulsada. Para solucionar los posibles problemas con los distintos códigos utilizados por cada fabricante, la clase `Canvas` nos permite convertir estos códigos a unas constantes que son independientes del fabricante. Esto lo podemos hacer con el método `getGameAction()`.

```
public int getGameAction (int codigo)
```

A este método le pasamos el código de la tecla pulsada, o soltada, y nos devuelve la constante correspondiente a ésta (ver Tabla 4.1.).

CONSTANTES DEVUELTAS	TECLAS PULSADAS/SOLTADAS
KEY_NUM0, KEY_NUM1, KEY_NUM2, KEY_NUM3, KEY_NUM4, KEY_NUM5, KEY_NUM6, KEY_NUM7, KEY_NUM8, KEY_NUM9	Teclas numéricas
KEY_POUND	Tecla almohadilla
KEY_STAR	Tecla asterisco
GAME_A, GAME_B, GAME_C, GAME_D	Teclas especiales de juego
UP	Tecla arriba
DOWN	Tecla abajo
LEFT	Tecla izquierda
RIGHT	Tecla derecha
FIRE	Tecla disparo

Tabla 4.1. Constantes de teclas.

4.3. Threads

En nuestro juego vamos a tener un bucle, llamado *game loop* que explicaremos en el próximo apartado, que tiene que estar activo durante toda la aplicación. Para hacer que esto sea posible no podemos utilizar el *thread* principal de la aplicación, sino que tendremos que ejecutar el cuerpo donde se encuentra nuestro *game loop* en uno diferente. Si no hiciéramos esto la animación sería más brusca y habría veces en la cual nuestro programa no respondería a la pulsación de teclas.

Un *thread* o hilo puede estar en cuatro estados posibles al igual que cualquier proceso: ejecutándose, preparado para ser ejecutado, suspendido a la espera de un evento externo y terminado (ver Figura 4.3).

El paquete que nos permite utilizar *threads* es `java.lang.Thread`. Para que nuestra clase principal, el *canvas*, se ejecute como un *thread* tiene que implementar la interfaz `java.lang.Runnable`, exactamente el método `run()`. Este método será el que se ejecute cuando lancemos nuestro *thread*.

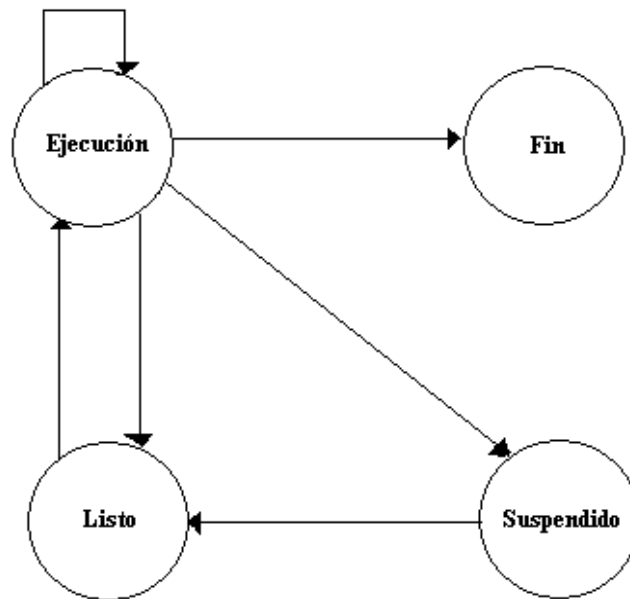


Figura 4.3. Vida de un thread.

Veamos los métodos más interesantes de esta clase. Aunque no lo vamos a utilizar, con el método `currentThread()` tendremos acceso al *thread* que se está ejecutando en todo momento. Para que nuestro *thread* sea ejecutado tenemos que utilizar el método `start()` de éste. Otro método que vamos a utilizar es `sleep(int tiempo)`, que detiene nuestro *thread* los milisegundos indicados en el argumento `tiempo`. Este tiempo dependerá de la velocidad de cada dispositivo.

4.4. Game Loop

El *game loop* es el bucle más importante de nuestro proyecto. Está incluido en nuestra clase principal, la que se ejecuta en un *thread* aparte, y con él haremos que parezca que nuestro personaje camina, que haya *scroll*.... En cada vuelta del *game loop* tenemos que actualizar todas las variables necesarias (como puntuación, posición del personaje, *scroll*, etc) y una vez hecho esto pintar todo por pantalla. Esta fase se denomina fase de *render*. Con esto, en cada vuelta del *game loop* creamos un *frame* distinto, y la sucesión de *frames* a una alta velocidad, nos da la sensación de animación.

Antes de entrar en el *game loop* tenemos que inicializar todas las variables y estructuras a utilizar en éste (puntuación, número de vidas, posición del personaje, etc). Un gráfico de un *game loop* general sería el que se muestra en la figura 4.4.

Para pintar por pantalla utilizamos el método `repaint()` seguido de `serviceRepaints()`. Cada vez que repintamos la pantalla, antes de volver a ejecutar el cuerpo del *game loop*, tendremos que dormir al *thread* durante un tiempo determinado. Este tiempo dependerá de cada dispositivo y de la complejidad del *frame* a pintar.

4.5. Inteligencia Artificial (IA)

Hay una gran diversidad de métodos de inteligencia artificial para la programación de juegos (grafos, redes neuronales,...). Nosotros en este proyecto vamos a utilizar una técnica muy sencilla y a la vez potente para crear rutinas de movimiento para nuestros bichos. Esta técnica está basada en un conjunto de estados y reglas y se denomina máquina de estado. El objeto al que queremos dotar de cierta inteligencia artificial estará en todo momento en uno de estos estados, y pasará a otro estado si y sólo si el objeto cumple una de estas reglas. En nuestro caso los bichos no son muy inteligentes y nos basta con un par de estados y un par de reglas (ver Figura 4.5).

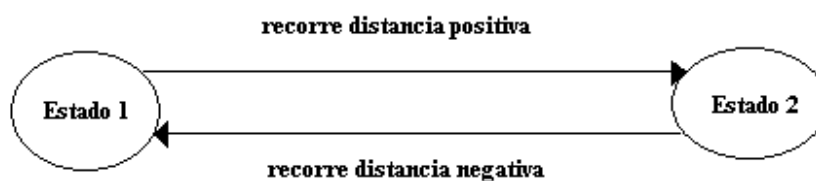


Figura 4.5. Máquina de estado de los bichos de Cómeme.

Si nos encontramos en el estado 1 nuestro bicho se moverá o en el sentido positivo del eje X o en el sentido positivo del eje Y. Cuando termine su recorrido, entra en el estado 2, volverá hacia donde empezó su animación, con lo que se moverá en el sentido negativo del eje X o del Y dependiendo de cual sea su dirección.

También habrá movimientos con incrementos en el eje X y el eje Y a la vez. Estos bichos se moverán en diagonal y aunque su máquina de estados es la misma que la anteriormente explicada, distinguimos sus estados como estado 3 y estado 4 (ver Figura 4.6).



Figura 4.6. (a) Bicho en estado 1 e incrementos positivos sobre el eje X.
(b) Bicho en estado 2 e incrementos negativos sobre el eje Y.

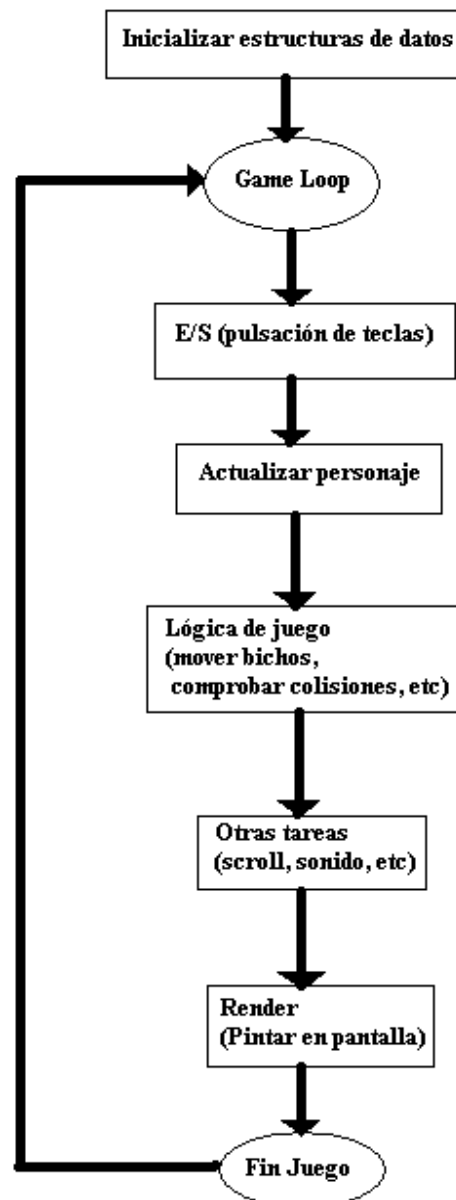


Figura 4.4. Forma general de un game loop.

CAPÍTULO 5: RMS (Record Management System)

Hasta ahora, en todo lo explicado, utilizábamos la memoria principal del dispositivo, la RAM, para guardar todos los datos. Todos estos datos al destruirse el *midlet* desaparecían, entonces cómo, por ejemplo, almacenar las puntuaciones de los jugadores. J2ME nos permite guardar datos como si de un disco duro se tratara. Este sistema se denomina RMS y es similar a una pequeña base de datos donde la unidad básica de almacenamiento es el registro. Estos registros son almacenados en un *recordStore*, el cual puede considerarse como una colección de registros. Para poder utilizar *rms* tenemos que importar el paquete `javax.microedition.rms`. Este paquete está compuesto de la clase `RecordStore` y de cuatro interfaces. Lo que vamos a utilizar de este paquete en nuestro proyecto son los siguientes métodos:

Lo primero es abrir y cerrar un *recordStore*. Esto se realiza con los métodos `openRecordStore(...)` y `closeRecordStore()`.

<pre>static RecordStore openRecordStore (String nombre, boolean crearSiEsNecesario)</pre>
<pre>void closeRecordStore()</pre>

El método para crear o abrir un *rms* tiene los siguientes parámetros: el primero es el nombre de la base de datos, cuya longitud no puede sobrepasar los 32 caracteres. El segundo nos dice si creamos la base de datos, si esta no existía anteriormente, con `true`, o si no la creamos, con `false`. Por último, al método que nos permite cerrar el *rms* no se le pasa ningún parámetro.

Para añadir registros utilizamos dos métodos distintos. Uno nos permite introducir el dato en el registro dado como parámetro, mientras que el otro no. Veámoslos:

<pre>public int addRecord (byte[] dato, int desplazamiento, int numBytes)</pre>
<pre>public void setRecord (int recordId, byte[] dato int desplazamiento, int numBytes)</pre>

El primer parámetro, del método `addRecord(...)`, es el dato que queremos almacenar. El segundo es la posición, a partir de la cual, colocamos el dato (posición dentro del *array* de bytes). Y el tercero es el número de bytes de los que consta el dato. Este método nos devuelve un entero, este entero es la posición que ocupa el dato dentro de la base de datos (identificador único).

El segundo método, `setRecord(...)`, es lo único que se diferencia del anterior es en que le damos la posición que queremos que ocupe el dato (identificador único). Además no nos devuelve nada, al contrario que `addRecord(...)`.

Para leer registros nos encontramos con el método `getRecord(...)`. Con este método podemos acceder al registro que queramos, siempre y cuando sepamos su identificador.

```
public byte[] getRecord (int identificador)
```

Su único parámetro es el identificador del registro que queremos leer, cuyo valor es devuelto como un *array* de bytes.

Todos los métodos anteriormente explicados lanzan la excepción `RecordStoreException`, con lo que tenemos que capturarla.

Otro método utilizado en el proyecto es `getNumRecords()` que nos devuelve el número de registros actuales que forman la base de datos. Para este método tendremos que capturar la excepción `RecordStoreNotOpenException`.

CAPÍTULO 6: SONIDO Y MÚSICA

6.1. Sonido

Para la reproducción de sonidos, en J2ME, nos encontramos con una API denominada *MIDP 2.0 Media API*. Esta API está compuesta, principalmente, por la clase `Manager` y las interfaces `Player` y `Control`.

La clase `Manager` nos permite crear objetos de tipo `Player`, y con éste podremos reproducir información multimedia, tanto música como vídeo. El paquete que nos suministra estos dos objetos es `javax.microedition.media`. Por último la clase `Control` nos permite controlar y gestionar un objeto de tipo `Player`. El paquete que nos permite utilizar éste último objeto es `javax.microedition.media.control`.

El método que nos permite crear un *player* es `createPlayer(...)`, y deberemos capturar las excepciones `IOException` y `MediaException`.

```
static Player createPlayer (InputStream stream, String type)
```

El primer parámetro es un flujo de entrada que nos permite leer el fichero que contiene el sonido o música. Mientras que en el segundo indicamos el tipo de fichero (wav, midi,...).

El método que nos permite reproducir un *player* es `start()`, y deberemos de capturar la excepción `MediaException`.

```
public void start()
```

Otros métodos utilizados en el proyecto son: `stop()`, nos permite parar un *player* actualmente en reproducción, `getDuration()`, nos devuelve la duración del *player* en microsegundos y `setLoopCount(int contador)`, el cual repite un sonido o melodía las veces que indicamos en su único parámetro (si es `-1` se repetirá indefinidamente).

6.2. Música

Además de utilizar sonidos *muestreados*, podemos componer nuestra propia música mediante la reproducción de tonos. Nuestra API nos permite componer nuestra música tono a tono o mediante secuencias de tonos, nosotros vamos a utilizar esta última.

Una secuencia de tonos será un *array* de bytes con la siguiente estructura:

1. Versión del formato de secuencia. Esto lo indicaremos en el primer par de bytes: (`ToneControl.VERSION`, <nº de version>).

2. Tiempo de la melodía. Lo indicamos también con un par de bytes: (ToneControl.TEMPO, <tiempo>).

3. Y por último una serie de bloques, cuantos queramos, que contienen las notas de la melodía. Cada bloque consta de un número par de bytes, en el que cada par de estos son una nota. Estas notas serán sacadas a partir de la nota Do Central (ToneControl.C4), sumándole o restándole a la frecuencia de ésta. La estructura de cada bloque es la siguiente:

```
//comienzo del bloque
ToneControl.BLOCK_START, 0,

//notas del bloque
C4, 8, C4+3, 9, ...

//fin del bloque
ToneControl.BLOCK_END, 0,

//reproduce bloque
ToneControl.PLAY_BLOCK, 0,
```

Una vez hayamos creado la secuencia, tendremos que crear un *player* que nos permita tocarla.

```
try{
    p = Manager.createPlayer (Manager.TONE_DEVICE_LOCATOR);
    p.realize();
    ToneControl c=(ToneControl)p.getControl("ToneControl");
    c.setSequence(<nombre de la secuencia creada>);
    p.setLoopCount(-1);
    p.start();
}catch (IOException ioe){
}catch (MediaException me){
}
```

PARTE 2

EL JUEGO CÓMEME

CAPÍTULO 7: Introducción al juego Cómeme

Vamos a comenzar esta segunda parte de nuestra memoria con una introducción a nuestro juego. Lo primero que vamos a ver es la historia, luego al tipo de juego que pertenece Cómeme y por último los objetivos, como jugador, de éste.

7.1. La historia de Cómeme

"Jaime Nu es un gran magnate de las finanzas; sabe sacar dinero de las piedras. Incluso el negocio más ruinoso se convierte en próspero y floreciente cuando llega a sus manos. Sin embargo, en contra de lo que cabría esperar, Jaime no posee una gran fortuna. En realidad, sus únicas posesiones personales son una pequeña mansión en las afueras y un coche que usa para trasladarse a la ciudad. La causa a la que se debe esto es que todo lo que gana se lo gasta en un pequeño y único vicio: comer.

Todos los días llegan a su mesa los más exquisitos manjares importados de todo el mundo: caviar de Rusia, naranjas de la China, ensaimadas de Mallorca, paellas de Valencia, perritos calientes de Estados Unidos, etc. Además, todos los fines de semana invita a todos sus amigos y organiza grandes fiestas que dejan a las comilonas romanas a la altura del betún.

Una noche, tras una de aquellas fiestas, Jaime comenzó a sentirse mal. Tal vez fuera debido al cochinitillo asado, o al pato a la naranja, o a los chuletones de Ávila, o tal vez le hubiera sentado mal alguna de las truchas que comió. En cualquier caso, todo parecía indicar que el causante había sido uno de los segundos platos, ya que ninguna de las treinta clases de sopas y cremas que probó parecían estar en mal estado.

Se tomó un somnífero y se acostó algo más temprano de lo habitual. Seguro de que cuando despertara se encontraría mucho mejor. Pero nada más comenzar a dormir, se encontró sumergido en una horrible pesadilla..."

MicroHobby nº97 [3]

7.2. Tipo de juego

El tipo de juego planteado es el de *plataformas*. En este género, que podemos englobar juegos como *Súper Mario Bros* o *Sonic the Hedgehog*, manejamos un personaje que tendrá que recoger una serie de objetos y pasar de pantalla antes de que los enemigos nos eliminen. Para esto nuestro personaje JaimeNu podrá ayudarse de varios movimientos como andar, saltar o incluso subir y bajar escaleras.

7.3. Objetivos del juego

Nuestro objetivo será el de recoger, en orden, 10 comidas repartidas por las 36 pantallas de que consta el juego. Estas comidas no estarán siempre en las mismas pantallas, sino que a cada partida cambiarán de lugar. Si cogemos una comida que no toca, nuestro personaje perderá una vida. Además consta de dos niveles de dificultad, normal y difícil. En este último no podremos tocar los corazones de manzana si no perderemos una vida.

CAPÍTULO 8: Mapa del juego

En este capítulo vamos a ver unas definiciones sobre los decorados (bloques gráficos y bloques de decorados) y por último todo lo relacionado con la construcción del mapa del juego, tanto las clases como los métodos utilizados para este cometido.

8.1. Pantallas. Bloques gráficos y bloques de decorados

Para la creación de las 36 pantallas (ver apéndice D) de que se compone el juego nos ayudamos de lo que llamaremos bloques gráficos y bloques de decorados. Cada pantalla está compuesta de una serie de bloques de decorados y estos a su vez son una repetición, en una dirección determinada, de un mismo bloque gráfico. Por ejemplo, el bloque gráfico número 11 (hueso) repetido verticalmente formaría una escalera o los bloques gráficos 1 ó 2 el suelo, el techo o una pared (ver Figura 8.1. y 8.2.).

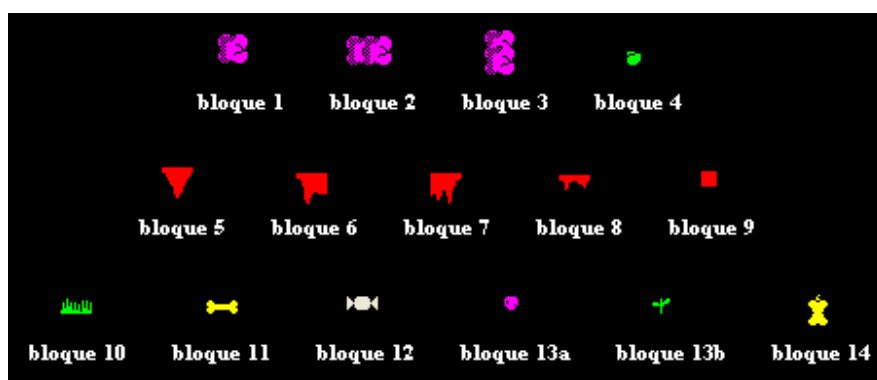


Figura 8.1. Bloques gráficos.



Figura 8.2. Ejemplos de bloques de decorados.

8.2. La clase Sprite

Vamos a comenzar con la primera clase del proyecto, la clase *Sprite*. Vamos a considerar como un *sprite* todo lo que vayamos a pintar por pantalla como los bloques, bichos, objetos y personaje.

Para cada *sprite* tendremos los atributos:

```
private int posX, posY;  
private boolean activo;  
private int frame, nframes;  
private Image[] sprites;
```

- `posX` y `posY`, que nos indicarán las coordenadas, en píxeles, del *sprite* a dibujar.
- `activo`, nos indica si el *sprite* está siendo utilizado actualmente en pantalla.
- `nframes` y `frame`, nos informa del número de imágenes que forman el *sprite* y la imagen actual a pintar.
- `[]sprites`, que almacena las imágenes que forman el *sprite*.

Su constructor:

```
public Sprite(int nframes){  
    activo=false;  
    frame=1;  
    this.nframes=nframes;  
    sprites=new Image[nframes+1];  
}
```

Al crear un *sprite* tendremos que pasar como argumento el número de imágenes que va a tener. En un principio no estará activo. Su imagen actual será la primera y se reservará memoria para todas las imágenes del *sprite*.

Tendremos métodos para asignar y consultar la posición del *sprite* en el eje de coordenadas, tanto para el eje X como para el Y.

```
public void setX(int x){  
    posX=x;  
}  
  
public void setY(int y){  
    posY=y;  
}  
  
int getX(){  
    return posX;  
}  
  
int getY(){  
    return posY;  
}
```

Podremos consultar la altura y anchura de nuestra imagen actual.

```
int getW() {
    return sprites[nframes].getWidth();
}

int getH() {
    return sprites[nframes].getHeight();
}
```

Podremos activar o desactivar nuestro *sprite* y consultar su estado (activado o desactivado).

```
public void on() {
    activo=true;
}

public void off() {
    activo=false;
}

public boolean estaActivo() {
    return activo;
}
```

También podremos cambiar el *frame* o imagen actual del *sprite*, consultar el número de imágenes del *sprite*, consultar la imagen actual y cargar las imágenes en el espacio reservado para tal caso en el constructor.

```
public void actualizarFrame(int frameo) {
    frame=frameo;
}

public int frames() {
    return nframes;
}

public int frameActual() {
    return frame;
}

public void añadirFrame(int frameo, String path) {
    try {
        sprites[frameo]=Image.createImage(path);
    } catch (IOException e) {
        System.err.println("No se puede cargar la imagen" + path + ":" + e.toString());
    }
}
```

Veamos los métodos que nos permiten detectar colisiones entre sprites:

```
boolean colision(Sprite sp){
    int w1,h1,w2,h2,x1,y1,x2,y2;
    w1=getW();
    h1=getH();
    w2=sp.getW();
    h2=sp.getH();
    x1=getX();
    y1=getY();
    x2=sp.getX();
    y2=sp.getY();

    if (((x1+w1)>x2) && ((y1+h1)>y2) && ((x2+w2)>x1) &&
        ((y2+h2)>y1)){
        return true;
    }else{
        return false;
    }
}
```

Este primer método comprueba la posible colisión entre el *sprite* que llama al método y otro pasado como parámetro. Simplemente vemos si el área que envuelve a una de las imágenes se solapa con la de la otra imagen (ver capítulo 4, apartado 4.1. Sprites). Este método es utilizado para las colisiones entre personaje y bichos, y para la de personaje y objetos.

El segundo método sólo lo utilizamos en el modo de juego difícil y comprueba la colisión entre personaje y un bloque gráfico (corazón de manzana).

```
boolean colisionBloque(Bloque bl){
    int i=0, j=1, w1 , h1 ,w2, h2, x1, y1, x2, y2;
    boolean colision=false;

    while ((!colision) &&
        (i<=bl.getBloquesRepetidos())){
        while ((!colision) && (j<=bl.getLongitud(i))){
            w1=getW();
            h1=getH();
            w2=bl.getW();
            h2=bl.getH();
            x1=getX();
            y1=getY();
            x2=bl.getCoordenadasX(i)+(j-1) *
                bl.getIncrX()*8;
            y2=bl.getCoordenadasY(i)/*(j-1) *
                bl.getIncrY()*8*/;
        }
        i++;
    }
}
```

```
if ((x1+w1)>x2) && ((y1+h1)>y2) &&
    ((x2+w2)>x1) && ((y2+h2)>y1)) {
    colision=true;
} else {
    colision=false;
}
j++;
}
j=1;
i++;
}
return colision;
}
```

Este método es similar al anterior, la única diferencia es que comprueba la colisión entre un *sprite* y un *bloque* (esta clase hereda de *Sprite* y la veremos en el próximo apartado).

El último método de esta clase es el que nos permite pintar el *sprite* por pantalla.

```
public void dibujar(Graphics g){
    g.drawImage(sprites[frame],
        posX,posY,Graphics.TOP|Graphics.LEFT);
}
```

Este método pinta la imagen actual en la posición asignada anteriormente (ver capítulo 3, apartado 3.3.4. Primitivas gráficas. Imágenes).

8.3. La clase Bloque

Esta clase representa el concepto, anteriormente explicado, de bloque de decorado (ver capítulo 8, apartado 8.1. Pantallas. Bloques gráficos y bloques de decorados). Un *bloque* hereda de la clase *Sprite*, por lo tanto tendrá además de los atributos de todo *sprite* unos atributos propios.

```
private int [] coordenadasX;
private int [] coordenadasY;
private int [] longitud;
private int bloquesRepetidos;
private int longH;
private int longV;
private int incrX;
private int incrY;
```

Las *coordenadasX* y *coordenadasY* son las coordenadas del comienzo del bloque de decorado (es un *array* porque en una misma pantalla puede haber varios bloques

de decorados que consten del mismo bloque gráfico, con lo que se almacenarán las coordenadas de todos los bloques de decorados de un mismo bloque gráfico de una misma pantalla). El atributo `longitud` es la longitud del bloque de decorado, o lo que es lo mismo el número de bloques gráficos que forman dicho bloque de decorado (es un *array* por el mismo motivo explicado para las coordenadas).

Con `bloquesRepetidos` obtenemos los diferentes bloques de decorados del mismo bloque gráfico almacenado en un mismo objeto de la clase `Bloque`. En `longH` y `longV` almacenamos la anchura y altura, en baja resolución, del bloque gráfico del que consta el bloque de decorado almacenado. Y por último en los atributos `incrX` e `incrY`, almacenamos los incrementos a sumar a las coordenadas para pintar el siguiente bloque gráfico del bloque de decorado (también en baja resolución, ver Figura 8.3.).

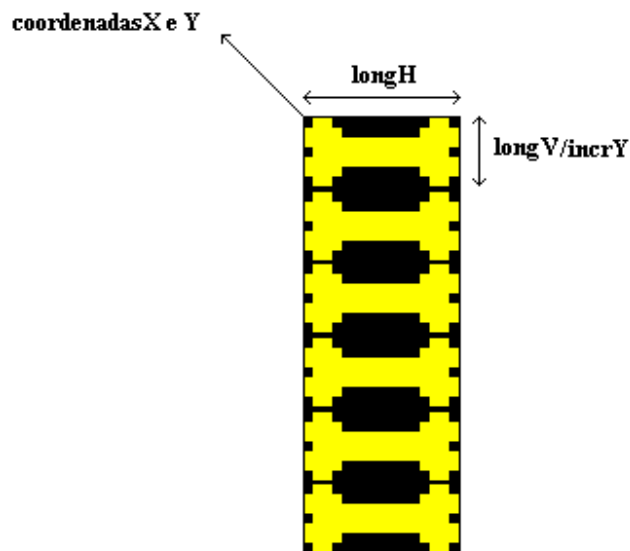


Figura 8.3. Ejemplo gráfico de la clase `Bloque`.

El atributo `incrX` sería 0 ya que los bloques gráficos se repiten en la dirección del eje Y, mientras que la longitud del bloque sería 6 (6 bloques gráficos repetidos).

Veamos el constructor de esta clase:

```
public Bloque(int nFrames,int longH,int longV,int incrX,int
incrY) {
    super(nFrames);
    this.longH=longH;
    this.longV=longV;
    this.incrX=incrX;
    this.incrY=incrY;
    coordenadasX=new int[40];
    coordenadasY=new int[40];
}
```

```
longitud=new int[40];  
bloquesRepetidos=0;  
}
```

Lo primero que hacemos es llamar a la clase `Sprite`, de la que hereda `Bloque`. Luego asignamos las dimensiones del bloque gráfico del que se compone el bloque y los incrementos X e Y. Reservamos memoria para 40 bloques de decorados compuestos del mismo bloque gráfico e indicamos con `bloquesRepetidos=0` que no hay ningún bloque almacenado.

Todos los métodos de esta clase, que no están heredados de la clase `Sprite`, sirven para asignar y consultar los distintos atributos.

```
public void setIncrX(int incrX){  
    this.incrX=incrX;  
}  
public int getIncrX(){  
    return incrX;  
}  
public void setIncrY(int incrY){  
    this.incrY=incrY;  
}  
public int getIncrY(){  
    return incrY;  
}  
public void setLongitud(int indice,int longitud){  
    this.longitud[indice]=longitud;  
}  
public int getLongitud(int indice){  
    return longitud[indice];  
}  
public void setBloquesRepetidos(int bloquesRepetidos){  
    this.bloquesRepetidos=bloquesRepetidos;  
}  
public int getBloquesRepetidos(){  
    return bloquesRepetidos;  
}  
public void setCoordenadasX(int indice,int coordenadaX){  
    coordenadasX[indice]=coordenadaX;  
}  
public int getCoordenadasX(int indice){  
    return coordenadasX[indice];  
}  
public void setCoordenadasY(int indice,int coordenadaY){  
    coordenadasY[indice]=coordenadaY;  
}
```

```
public int getCoordenadasY(int indice){
    return coordenadasY[indice];
}
```

8.4. La clase Bloque en el Canvas

En este apartado vamos a explicar todo lo relacionado con la clase `Bloque` dentro de nuestra clase principal, la clase `SSCanvas` (hereda de `Canvas`). Englobaremos tanto tablas como procedimientos usados relacionados con la clase `Bloque`.

8.4.1. Tabla de bloques de decorados

La tabla que vamos a utilizar para almacenar los distintos bloques de decorados de una misma pantalla, `tablaBloquesDecorados`, consta de 20 posiciones, instancias de la clase `Bloque`. Estas 20 posiciones son los distintos bloques de decorados que vamos a utilizar para la generación de las 36 pantallas. Los datos de los bloques que vamos a utilizar en cada pantalla los encontramos en unos *arrays* de enteros que explicaremos en el apartado 8.4.3.

```
private Bloque[] tablaBloquesDecorados=new Bloque[20];
```

8.4.2. Inicialización de la tabla de bloques de decorados

Para la inicialización de `tablaBloquesDecorados` lo único que vamos a hacer será crear una instancia de cada posición de la tabla y cargar la imagen correspondiente a cada bloque. Los valores almacenados en este procedimiento, `inicializarBloquesDecorados()`, no serán modificados en la ejecución del programa.

```
void inicializarBloquesDecorados() {
    int i;

    tablaBloquesDecorados[0]=new Bloque(1,2,2,2,0);
    tablaBloquesDecorados[0].añadirFrame(1,"/bloque1.png");

    tablaBloquesDecorados[1]=new Bloque(1,2,2,0,2);
    tablaBloquesDecorados[1].añadirFrame(1,"/bloque1.png");

    tablaBloquesDecorados[2]=new Bloque(1,3,2,3,0);
    tablaBloquesDecorados[2].añadirFrame(1,"/bloque2.png");

    tablaBloquesDecorados[3]=new Bloque(1,2,3,0,3);
    tablaBloquesDecorados[3].añadirFrame(1,"/bloque3.png");
}
```



```
tablaBloquesDecorados[4]=new Bloque(1,1,1,1,0);
tablaBloquesDecorados[4].añadirFrame(1,"/bloque4.png");

tablaBloquesDecorados[5]=new Bloque(1,1,1,0,1);
tablaBloquesDecorados[5].añadirFrame(1,"/bloque4.png");

tablaBloquesDecorados[6]=new Bloque(1,2,1,2,0);
tablaBloquesDecorados[6].añadirFrame(1,"/bloque13a.png");

tablaBloquesDecorados[7]=new Bloque(1,2,1,0,1);
tablaBloquesDecorados[7].añadirFrame(1,"/bloque13b.png");

tablaBloquesDecorados[8]=new Bloque(1,2,1,-1,1);
tablaBloquesDecorados[8].añadirFrame(1,"/bloque12.png");

tablaBloquesDecorados[9]=new Bloque(1,2,1,-1,-1);
tablaBloquesDecorados[9].añadirFrame(1,"/bloque12.png");

tablaBloquesDecorados[10]=new Bloque(1,2,1,0,1);
tablaBloquesDecorados[10].añadirFrame(1,"/bloque11.png");

tablaBloquesDecorados[11]=new Bloque(1,2,1,2,0);
tablaBloquesDecorados[11].añadirFrame(1,"/bloque10.png");

tablaBloquesDecorados[12]=new Bloque(1,2,2,2,0);
tablaBloquesDecorados[12].añadirFrame(1,"/bloque5.png");

tablaBloquesDecorados[13]=new Bloque(1,2,2,2,0);
tablaBloquesDecorados[13].añadirFrame(1,"/bloque6.png");

tablaBloquesDecorados[14]=new Bloque(1,2,2,2,0);
tablaBloquesDecorados[14].añadirFrame(1,"/bloque7.png");

tablaBloquesDecorados[15]=new Bloque(1,2,1,2,0);
tablaBloquesDecorados[15].añadirFrame(1,"/bloque8.png");

tablaBloquesDecorados[16]=new Bloque(1,2,2,2,0);
tablaBloquesDecorados[16].añadirFrame(1,"/bloque14.png");

tablaBloquesDecorados[17]=new Bloque(1,1,1,1,0);
tablaBloquesDecorados[17].añadirFrame(1,"/bloque9.png");

tablaBloquesDecorados[18]=new Bloque(1,1,1,1,1);
tablaBloquesDecorados[18].añadirFrame(1,"/bloque4.png");

tablaBloquesDecorados[19]=new Bloque(1,1,1,-1,1);
```

```

tablaBloquesDecorados[19].añadirFrame(1, "/bloque4.png");

for (i=0; i<20; i++) {
    tablaBloquesDecorados[i].off();
}
}

```

Además de inicializar, desactivamos los 20 bloques ya que todavía no hemos cargado ninguna pantalla (ver Figura 8.4. para una representación gráfica de `tablaBloquesDecorados`).





















código	bloque	longH	longV	IncrX	IncrY	bloque	longH	longV	IncrX	IncrY	código
0		2	2	2	0		2	1	0	1	10
1		2	2	0	2		2	1	2	0	11
2		3	2	3	0		2	2	2	0	12
3		2	3	0	3		2	2	2	0	13
4		1	1	1	0		2	2	2	0	14
5		1	1	0	1		2	1	2	0	15
6		2	1	2	0		2	2	2	0	16
7		2	1	0	1		1	1	1	0	17
8		2	1	-1	1		1	1	1	1	18
9		2	1	-1	-1		1	1	-1	1	19

Figura 8.4. Representación gráfica de `tablaBloquesDecorados`.

8.4.3. Generación de pantallas

Para la generación de las pantallas vamos a valernos de unos *arrays* de enteros por cada una de éstas (ver Apéndice A). Por cada dos enteros de estos *arrays* obtenemos un bloque de decorado y su posición en la pantalla. El siguiente procedimiento almacena todos los bloques de decorados a utilizar en una pantalla, incluida sus coordenadas, que posteriormente pintaremos por pantalla con el procedimiento que explicaremos en el siguiente apartado.

```
void generacionPantallas(int pantalla[]){
    int i=0,j,resto,cociente;
    Bloque bloque;

    if (!generadoPantalla){
        inicializarMapaPantalla();
    }

    while ((!generadoPantalla)&&(i<pantalla.length)){
        cociente=pantalla[i]/256;
        resto=pantalla[i]%256;
        int y=cociente*8;
        int x=resto*8;
        cociente=pantalla[i+1]/256;
        resto=pantalla[i+1]%256;
        int codigo=cociente%32;
        int longitud=resto%128;
        rellenarMapaPantalla(y/8,x/8,longitud,codigo);
        bloque=tablaBloquesDecorados[codigo];

        if (!bloque.estaActivo()){
            bloque.on();
            bloque.setCoordenadasX
                (bloque.getBloquesRepetidos(),x);

            bloque.setCoordenadasY
                (bloque.getBloquesRepetidos(),y);

            bloque.setLongitud
                (bloque.getBloquesRepetidos(),longitud);
        }else{
            bloque.setBloquesRepetidos
                (bloque.getBloquesRepetidos()+1);

            bloque.setCoordenadasX
                (bloque.getBloquesRepetidos(),x);

            bloque.setCoordenadasY
                (bloque.getBloquesRepetidos(),y);

            bloque.setLongitud
                (bloque.getBloquesRepetidos(),longitud);
        }

        bloque.setCoordenadasX
            (bloque.getBloquesRepetidos(),
            x-posicionRelativa);
    }
}
```

```
        i+=2;
    }
```

Esta primera parte de `generacionPantallas(...)` se encarga de almacenar los datos de los bloques de decorados la primera vez que entramos en una pantalla. Esto lo hacemos con una variable global llamada `generadoPantalla`. Una vez almacenado todos los datos ya no se entrará en el bucle `while`. Los datos recogidos son: el código del bloque, sus coordenadas iniciales y la longitud de éste (los demás datos están almacenados en `tablaBloquesDecorados`). Los procedimientos referidos al mapa de pantalla serán explicados en el apartado 8.5. y la variable global `posiciónRelativa` está relacionada con el *scroll* de pantalla que explicaremos más adelante. Veamos la segunda parte de este procedimiento que se entrará sólo si la pantalla ya ha sido generada con anterioridad.

```
if (hayScroll()){
    if (generadoPantalla){
        for (i=0;i<20;i++){
            bloque=tablaBloquesDecorados[i];

            if (bloque.estaActivo()){
                for (j=0;
                    j<=bloque.getBloquesRepetidos();j++){

                    bloque.setCoordenadasX(j,+indiceScroll);
                    bloque.getCoordenadasX(j)

                }
            }
        }
    }else{
        generadoPantalla=true;
    }
}
}
```

Una vez se ha generado la pantalla, tenemos todos los datos de los bloques de decorados de ésta, sólo nos hace falta comprobar si hay *scroll* y si es así tendremos que mover todos los bloques los píxeles necesarios, dado por `indiceScroll`, en el eje X (el *scroll* sólo se da en éste eje). Esta última variable, `indiceScroll`, será explicada con todo lo relacionado al *scroll*.

8.4.4. Pintando la pantalla

Una vez almacenados todos los datos referentes a los bloques de una pantalla, toca pintarla.

```
public void pintarPantalla(Graphics g){

    int i,j,k,incrementoX=0,incrementoY=0;
    Bloque bloque;

    for (i=19;i>=0;i--){
        bloque=tablaBloquesDecorados[i];

        if (bloque.estaActivo()) {
            for (j=0;j<=bloque.getBloquesRepetidos();j++){
                for (k=1;k<=bloque.getLongitud(j);k++){

                    bloque.setX(bloque.getCoordenadasX(j)
                        +incrementoX);

                    bloque.setY(bloque.getCoordenadasY(j)
                        +incrementoY);

                    bloque.dibujar(g);

                    incrementoX+=bloque.getIncrX()*8;
                    incrementoY+=bloque.getIncrY()*8;
                }
                incrementoX=0;
                incrementoY=0;
            }
        }
    }
}
```

Lo único que hacemos para pintar la pantalla es ver en la tabla de bloques de decorados qué bloques están activos, los que forman la pantalla actual, luego pintamos el primer bloque gráfico de estos y a continuación todos los demás, tantos como la longitud del bloque de decorado, sumando a las coordenadas iniciales los incrementos de cada bloque.

8.5. El mapa de pantalla

Para saber en todo momento si nuestro personaje colisiona con un bloque, se encarama a una escalera de huesos o cae por un hueco de la pantalla, vamos a utilizar lo que llamaremos mapa de pantalla. Este mapa constará de 32 columnas y 20 filas (el tamaño de nuestras pantallas en baja resolución), y en cada posición almacenaremos un *byte*, el cual nos indica:

- Con un 0, que no hay nada en esa posición.
- Con un 1, que hay algo sólido, como una piedra o caramelo, y no lo podemos

atravesar.

- Con un 2, hay una escalera de huesos que podremos atravesar, subir o bajar por ella.
- Con un 3, hay un corazón de manzana que nos restará una vida (sólo en el modo difícil).

En un principio nuestro mapa de pantalla estará inicializado con todas sus posiciones a 0. Esto lo haremos con el siguiente procedimiento:

```
void inicializarMapaPantalla(){  
  
    int i,j;  
    for (i=0;i<32;i++){  
        for (j=0;j<20;j++){  
            mapaPantalla[i][j]=0;  
        }  
    }  
}
```

Sólo inicializamos el mapa de pantalla al entrar en una pantalla nueva. A la misma vez que almacenamos los bloques de la pantalla actual, rellenamos el mapa de pantalla (ver `generacionPantallas(...)` en el apartado 8.4.3) con el siguiente procedimiento:

```
void rellenarMapaPantalla(int y, int x, int longitud, int
codigo){
    int i,j,k;
    byte tipoBloque;
    Bloque bloque=tablaBloquesDecorados[codigo];

    switch (codigo){
        case 10:tipoBloque=2;
        break;

        case 16:if(dificil){
            tipoBloque=3;
        }else{
            tipoBloque=1;
        }
        break;

        default:tipoBloque=1;
    }

    for (i=1;i<=longitud;i++){
        for (j=0;j<bloque.getLongV();j++){
            for (k=0;k<bloque.getLongH();k++){
                try{
                    mapaPantalla[x+k][y+j]=tipoBloque;
```

```
        } catch (ArrayIndexOutOfBoundsException  
            aioobe) {  
            System.out.println(aioobe.toString());  
        }  
    }  
    }  
    x+=bloque.getIncrX();  
    y+=bloque.getIncrY();  
}  
}
```

La forma de rellenar nuestro mapa de pantalla es similar al de pintar los bloques por pantalla (ver apartado 8.4.4). La única diferencia es que para pintar utilizábamos las coordenadas en alta resolución (píxeles) y ahora lo hacemos en baja resolución.

CAPÍTULO 9: Los bichos

En este capítulo vamos a explicar todo lo relacionado con nuestros bichos. Primero vamos a ver los tipos de bichos que pueden haber en una pantalla, a continuación de esto veremos la clase `Bicho` y por último todo lo relacionado con ésta en nuestra clase principal `SSCanvas`.

9.1. Tipos de bichos

Consideraremos tres tipos de bichos (ver Figura 9.1.), los tres tipos se moverán haciendo un recorrido de ida y vuelta siempre en la misma dirección.

1. **Estáticos.** Sólo tendrán un *frame*.
2. **Animados.** Tendrán varios *frames*, por lo que al alternar entre ellos dará la sensación de animación.
3. **Direccionales.** Además de ser animados, estos bichos cambiarán de *frames* dependiendo del sentido en el que vayan.



Figura 9.1. Tipos de bichos.

9.2. La clase Bicho

Esta clase, al igual que la clase `Bloque`, hereda de `Sprite` y tiene unos atributos propios que vamos a ver a continuación:

```
private int estado;  
private int incrementoX;  
private int incrementoY;  
private int velocidad;  
private int codigo;  
private int longitud;  
private int pasos;  
private int tipo;
```

El atributo `estado` puede ser 1, 2, 3 ó 4. Si estamos en el estado 1 nuestro bicho se moverá en la dirección del eje X con incrementos positivos, y con incrementos negativos si está en el estado 2. Si nos encontramos en el estado 3, nuestro bicho se moverá en diagonal con incrementos X positivos e incrementos Y negativos y al contrario si nos encontramos en el estado 4. Los incrementos X e Y (`incrementoX` e `incrementoY`)

son los píxeles que se moverá el bicho en cada paso de animación externa. La *velocidad* puede ser 1, 2 ó 3. Contra mayor sea esta más lento será nuestro bicho. El *codigo* es simplemente el índice de la tabla de bichos que explicaremos en el siguiente apartado. La *longitud* es la distancia en pasos que tiene que recorrer el bicho antes de cambiar de estado, cada vez que se mueve éste es un paso. El atributo *pasos* lleva el número de vueltas del *game loop*, con esto sabemos cuando debemos mover un bicho y cuando no. Y *tipo* es el tipo de bicho y puede valer 1, 2 ó 3.

El constructor de esta clase simplemente hace una llamada a la clase de la que hereda, *Sprite*.

```
public Bicho (int nFrames){  
    super(nFrames);  
}
```

En esta clase también nos encontramos con los pertinentes métodos para asignar y consultar los distintos atributos:

```
public void setEstado(int estado){  
    this.estado=estado;  
}  
public int getEstado(){  
    return estado;  
}  
  
public void setTipo(int tipo){  
    this.tipo=tipo;  
}  
public int getTipo(){  
    return tipo;  
}  
  
public void setIncrementoX(int incrementoX){  
    this.incrementoX=incrementoX;  
}  
public int getIncrementoX(){  
    return incrementoX;  
}  
  
public void setIncrementoY(int incrementoY){  
    this.incrementoY=incrementoY;  
}  
public int getIncrementoY(){  
    return incrementoY;  
}
```

```
public void setVelocidad(int velocidad){
    this.velocidad=velocidad;
}
public int getVelocidad(){
    return velocidad;
}

public void setCodigo(int codigo){
    this.codigo=codigo;
}
public int getCodigo(){
    return codigo;
}

public void setLongitud(int longitud){
    this.longitud=longitud;
}
public int getLongitud(){
    return longitud;
}

public void setPasos(int pasos){
    this.pasos=pasos;
}
public int getPasos(){
    return pasos;
}
```

El método más importante de esta clase es el método `mover()`. Mediante este método nuestros bichos se desplazarán por la pantalla, se moverán (siempre y cuando sean animados), y cambiarán de sentido (cambio de estado).

```
public void mover(){
    int deltaX=0,deltaY=0;
    if ((estado==1) && (pasos%(velocidad*longitud)!=0)){
        deltaX=incrementoX;
        deltaY=incrementoY;
    }else if (estado==1){
        estado=2;
        deltaX=0;
        deltaY=0;
    }else if ((estado==2) && (pasos%(velocidad*longitud)!=0)){
        deltaX=-incrementoX;
        deltaY=-incrementoY;
    }else if (estado==2){
        estado=1;
        deltaX=0;
    }
```

```

deltaY=0;

}else if ((estado==3) && (pasos%(velocidad*longitud)!=0)){
    deltaX=incrementoX;
    deltaY=-incrementoY;
}else if (estado==3){

    deltaX=0;
    deltaY=0;
    estado=4;
}else if ((estado==4) && (pasos%(velocidad*longitud)!=0)){
    deltaX=-incrementoX;
    deltaY=incrementoY;
}else{
    deltaX=0;
    deltaY=0;
    estado=3;
}

```

Esta primera parte se encarga de la animación externa de los bichos (el desplazamiento por la pantalla), además permite que los bichos cambien de sentido una vez hayan recorrido su camino de ida. Para saber si un bicho ha terminado su recorrido comprobamos si pasos (vueltas del *game loop*) es múltiplo de velocidad multiplicada por longitud, si lo es, cambiaremos de estado y no moveremos al bicho en ese ciclo.

Veamos la segunda parte del método:

```

if ((pasos/velocidad)%4==0){
    if (tipo==2){
        if (frameActual()<frames()){
            actualizarFrame(frameActual()+1);
        }else{
            actualizarFrame(1);
        }
    }else if (tipo==3){
        if ((deltaX==0) && ((estado==1)|| (estado==3))){
            if (frameActual()==(frames()/2)){
                actualizarFrame(frameActual()+1);
            }else{
                actualizarFrame(frameActual()+
                    frames()/2+1);
            }
        }
    }
}

```

```

        }else if ((deltaX==0) && ((estado==2) || (estado==4))) {
            if (frameActual()==frames()) {
                actualizarFrame(1);
            }else if (frameActual()==frames()/2+1) {
                actualizarFrame(frameActual()-1);
            }else{
                actualizarFrame(frameActual()-frames()/2-1);
            }
        }else if ((estado==1) || (estado==3)) {
            if (frameActual()<frames()) {
                actualizarFrame(frameActual()+1);
            }else{
                actualizarFrame(frames()/2+1);
            }
        }else if ((estado==2) || (estado==4)) {
            if (frameActual()<frames()/2) {
                actualizarFrame(frameActual()+1);
            }else{
                actualizarFrame(1);
            }
        }
    }
    setX(getX()+deltaX);
    setY(getY()+deltaY);
}
}

```

Esta segunda parte del método se encarga del movimiento interno (animación de los bichos). Dependiendo del tipo de bicho cambiaremos de *frame* de una manera distinta (como los de tipo 1 no tienen animación no los incluimos). Sólo cambiaremos de *frame* cuando se cumpla la condición $((\text{pasos}/\text{velocidad})\%4==0)$. Esto es para que la animación sea suave. Si por ejemplo cambiásemos de *frame* por cada ciclo del *game loop* (cada vez que *pasos* es aumentado en uno) tendríamos una animación muy brusca.

9.3. La clase Bicho en el Canvas

Como para la clase anterior (clase *Bloque*), vamos a explicar todo lo relacionado con ésta dentro de nuestra clase principal (*SSCanvas*). Explicaremos tanto la tabla utilizada para almacenar todos los bichos, como los procedimientos utilizados para inicializar y generar los bichos.

9.3.1. Tabla de bichos

La tabla que vamos a utilizar para almacenar todos los bichos consta de 24 posiciones. Cada posición es una instancia de la clase *Bicho* en la que almacenaremos un

bicho diferente.

```
private Bicho[] tablaBichos=new Bicho[24];
```

9.3.2. Inicialización de la tabla de bichos

Para inicializar nuestra tabla de bichos, lo primero que vamos a hacer es crear una instancia de `Bicho` por cada posición de la tabla. Una vez hecho esto añadiremos todos los *frames* que componen a cada bicho. Y por último asignaremos el tipo de cada bicho. También colocaremos como desactivados a todos los bichos ya que todavía no tenemos que pintar ninguno por pantalla. Los datos almacenados durante la inicialización de la tabla no serán modificados durante la ejecución del programa, sólo serán ampliados con otros atributos. Veamos dicho procedimiento:

```
void inicializarBichos() {
    int i;

    tablaBichos[0]=new Bicho(2);
    tablaBichos[0].añadirFrame(1,"/bicho0a.png");
    tablaBichos[0].añadirFrame(2,"/bicho0b.png");
    tablaBichos[0].setTipo(2);

    tablaBichos[1]=new Bicho(1);
    tablaBichos[1].añadirFrame(1,"/bicho1.png");

    tablaBichos[2]=new Bicho(1);
    tablaBichos[2].añadirFrame(1,"/bicho2.png");

    tablaBichos[3]=new Bicho(1);
    tablaBichos[3].añadirFrame(1,"/bicho3.png");

    tablaBichos[4]=new Bicho(1);
    tablaBichos[4].añadirFrame(1,"/bicho4.png");

    tablaBichos[5]=new Bicho(2);
    tablaBichos[5].añadirFrame(1,"/bicho5a.png");
    tablaBichos[5].añadirFrame(2,"/bicho5b.png");
    tablaBichos[5].setTipo(2);
```

```
tablaBichos[6]=new Bicho(8);
tablaBichos[6].añadirFrame(1,"/bicho6a.png");
tablaBichos[6].añadirFrame(2,"/bicho6b.png");
tablaBichos[6].añadirFrame(3,"/bicho6c.png");
tablaBichos[6].añadirFrame(4,"/bicho6b.png");
tablaBichos[6].añadirFrame(5,"/bicho6a2.png");
tablaBichos[6].añadirFrame(6,"/bicho6b2.png");
tablaBichos[6].añadirFrame(7,"/bicho6c2.png");
tablaBichos[6].añadirFrame(8,"/bicho6b2.png");
tablaBichos[6].setTipo(3);

tablaBichos[7]=new Bicho(2);
tablaBichos[7].añadirFrame(1,"/bicho7a.png");
tablaBichos[7].añadirFrame(2,"/bicho7b.png");
tablaBichos[7].setTipo(2);

tablaBichos[8]=new Bicho(4);
tablaBichos[8].añadirFrame(1,"/bicho8b.png");
tablaBichos[8].añadirFrame(2,"/bicho8a.png");
tablaBichos[8].añadirFrame(3,"/bicho8c.png");
tablaBichos[8].añadirFrame(4,"/bicho8a.png");
tablaBichos[8].setTipo(2);

tablaBichos[9]=new Bicho(4);
tablaBichos[9].añadirFrame(1,"/bicho9a.png");
tablaBichos[9].añadirFrame(2,"/bicho9b.png");
tablaBichos[9].añadirFrame(3,"/bicho9a2.png");
tablaBichos[9].añadirFrame(4,"/bicho9b2.png");
tablaBichos[9].setTipo(3);

tablaBichos[10]=new Bicho(2);
tablaBichos[10].añadirFrame(1,"/bicho10a.png");
tablaBichos[10].añadirFrame(2,"/bicho10b.png");
tablaBichos[10].setTipo(2);

tablaBichos[11]=new Bicho(4);
tablaBichos[11].añadirFrame(1,"/bicho11a.png");
tablaBichos[11].añadirFrame(2,"/bicho11b.png");
tablaBichos[11].añadirFrame(3,"/bicho11a2.png");
tablaBichos[11].añadirFrame(4,"/bicho11b2.png");
tablaBichos[11].setTipo(3);

tablaBichos[12]=new Bicho(1);
tablaBichos[12].añadirFrame(1,"/bicho12.png");

tablaBichos[13]=new Bicho(1);
tablaBichos[13].añadirFrame(1,"/bicho13.png");
```

```
tablaBichos[14]=new Bicho(1);
tablaBichos[14].añadirFrame(1,"/bicho14.png");

tablaBichos[15]=new Bicho(1);
tablaBichos[15].añadirFrame(1,"/bicho15.png");

tablaBichos[16]=new Bicho(2);
tablaBichos[16].añadirFrame(1,"/bicho16a.png");
tablaBichos[16].añadirFrame(2,"/bicho16b.png");
tablaBichos[16].setTipo(2);

tablaBichos[17]=new Bicho(2);
tablaBichos[17].añadirFrame(1,"/bicho17a.png");
tablaBichos[17].añadirFrame(2,"/bicho17b.png");
tablaBichos[17].setTipo(2);

tablaBichos[18]=new Bicho(1);
tablaBichos[18].añadirFrame(1,"/bicho18.png");

tablaBichos[19]=new Bicho(2);
tablaBichos[19].añadirFrame(1,"/bicho19a.png");
tablaBichos[19].añadirFrame(2,"/bicho19b.png");
tablaBichos[19].setTipo(2);

tablaBichos[20]=new Bicho(1);
tablaBichos[20].añadirFrame(1,"/bicho20.png");

tablaBichos[21]=new Bicho(1);
tablaBichos[21].añadirFrame(1,"/bicho21.png");

tablaBichos[22]=new Bicho(2);
tablaBichos[22].añadirFrame(1,"/bicho22a.png");
tablaBichos[22].añadirFrame(2,"/bicho22b.png");
tablaBichos[22].setTipo(2);

tablaBichos[23]=new Bicho(2);
tablaBichos[23].añadirFrame(1,"/bicho23a.png");
tablaBichos[23].añadirFrame(2,"/bicho23b.png");
tablaBichos[23].setTipo(2);

for (i=0;i<=23;i++){
    tablaBichos[i].off();
}
}
```


En la figura 9.2. mostramos esta tabla gráficamente.

























codigo	frames	código	frames
0		12	
1		13	
2		14	
3		15	
4		16	
5		17	
6		18	
7		19	
8		20	
9		21	
10		22	
11		23	

Figura 9.2. Representación gráfica de tablaBichos.

9.3.3. Generación de bichos

Para la generación de bichos vamos a valernos de unos *arrays* de enteros *cortos* (ver Apéndice A). En cada uno de estos *arrays* guardamos la información de todos los bichos de una pantalla (como máximo 4 bichos por pantalla). Cada 6 enteros *cortos* obtenemos los atributos necesarios para posicionar, mover, animar y pintar a uno de nuestros bichos. Veamos el código de este procedimiento:

```
void generacionBichos(short bichoActual[]){

    int i=0;
    Bicho bicho;

    while ((!generadoBicho) && (i<bichoActual.length)){
        int x=bichoActual[i];
        int y=bichoActual[i+1];
        int velocidad=bichoActual[i+2]/64;
        int resto=bichoActual[i+2]%64;
```

```
        int incrementoY=resto%8;
        int incrementoX=resto/8;
        int codigo=bichoActual[i+3]%32;
        int longitud=bichoActual[i+4];
        int estado=bichoActual[i+5];

        bicho=tablaBichos[codigo];

        bicho.on();
        bicho.setVelocidad(velocidad);
        bicho.setIncrementoX(incrementoX);
        bicho.setIncrementoY(incrementoY);
        bicho.setCodigo(codigo);
        bicho.setLongitud(longitud);
        bicho.setEstado(estado);
        bicho.setX(x-posicionRelativa);
        bicho.setY(y);

        i+=6;
    }
```

En esta primera parte del procedimiento recogemos los atributos de todos los bichos de una pantalla y se los asignamos a los correspondientes en `tablaBichos`. Esto sólo lo hacemos la primera vez que entramos en una pantalla, mediante una variable global llamada `generadoBicho` (igual que con los bloques). Además colocamos los bichos generados como activos para poder localizarlos más adelante (para saber que bichos debemos pintar en cada momento).

```
if (generadoBicho){
    for (i=0;i<=23;i++){
        bicho=tablaBichos[i];
        if (bicho.estaActivo()){
            bicho.setPasos(ciclo);
            if ((ciclo%bicho.getVelocidad())==0){
                bicho.mover();
            }
            if (hayScroll()){
                bicho.setX(bicho.getX()+indiceScroll);
            }
        }
    }
}else{
    generadoBicho=true;
}
}
```

En la segunda parte del procedimiento, que sólo entraremos una vez hayan sido generado los bichos, comprobaremos si a los bichos generados con anterioridad les toca moverse. Si es así llamaremos al método `mover()` de la clase `Bicho` (mirar apartado 9.2). Además si hay *scroll*, deberemos desplazar la posición de todos los bichos `indiceScroll` píxeles en el eje X (esta variable será explicada, con todo lo relacionado al *scroll*, más adelante).

9.3.4. Pintando los bichos

Una vez inicializada la tabla de los bichos y generados los bichos de la pantalla actual, toca pintarlos por pantalla. Veamos el procedimiento:

```
public void pintarBicho(Graphics g){
    int i=0;
    Bicho bicho;

    for (i=0;i<=23;i++){
        bicho=tablaBichos[i];
        if (bicho.estaActivo()){
            bicho.setX(bicho.getX());
            bicho.setY(bicho.getY());
            bicho.dibujar(g);
        }
    }
}
```

Lo único que hacemos es recorrer la tabla de bichos, posicionamos los bichos de la pantalla actual (los que están activos) y a continuación los pintamos por pantalla.

CAPÍTULO 10: Objetos

En este capítulo vamos a ver todo lo relacionado con los objetos de nuestro juego. Primero vamos a ver los tipos de objetos a repartir en 30 de las 36 pantallas, luego veremos la clase que da forma a nuestros objetos y por último veremos nuestra clase `Objeto` dentro de nuestro *canvas*, tanto tablas utilizadas como procedimientos para la inicialización, generación y pintado de objetos.

10.1. Tipos de objetos

Los objetos se dividirán en tres tipos:

1. Platos o comidas. Contaremos con 10 distintos y no se podrán repetir. Tendremos que cogerlos en orden si no perderemos una vida. Si recogemos los 7 primeros nos teletransportaremos de pantalla para recoger los 3 últimos platos, inaccesibles de otro modo. Cuando consigamos recoger los 10 platos el juego terminará.

2. Galletas de la suerte. Repartiremos 10 de estos objetos. Estos objetos nos darán una cantidad de puntos entre 100 y 800 ó bien una vida extra.

3. Vida extra. Repartiremos 5 vidas extras por las 36 pantallas. Si tenemos más de 10 vidas no subirá al marcador.

El reparto de estos objetos será aleatorio en cada partida.

10.2. La clase Objeto

Esta clase, al igual que las clases `Bicho` y `Bloque`, heredan de la clase `Sprite`. Veamos los atributos propios de esta clase:

```
private int pantalla;  
private int tipo;
```

Con el atributo `pantalla` indicamos en la pantalla que se encuentra el objeto, mientras que con `tipo` hacemos lo propio con el tipo de objeto.

Con el constructor de esta clase, además de la pertinente llamada a la clase `Sprite`, ponemos el atributo `pantalla` como 0, indicando con esto que no ha sido asignado a ninguna pantalla (la numeración de pantallas es de 1 a 36).

```
public Objeto(int nframes){  
    super(nframes);  
    pantalla=0;  
}
```

Todos los métodos utilizados por esta clase están destinados para la asignación y consulta de sus atributos.

```
public void setPantalla(int pantalla){
    this.pantalla=pantalla;
}

public int getPantalla(){
    return pantalla;
}

public void setTipo(int tipo){
    this.tipo=tipo;
}

public int getTipo(){
    return tipo;
}
```

10.3. La clase Objeto en el Canvas

Como con las dos clase anteriores, veremos todo lo relacionado con los objetos en nuestra clase principal (SSCanvas). Tanto las tablas utilizadas, como los procedimientos de inicialización, generación y pintado de los objetos.

10.3.1. Tablas de objetos

10.3.1.1. Pantallas

Con esta tabla almacenamos las coordenadas en las que colocaremos los objetos en las distintas pantallas.

```
private int [] pantallas=new int[37];
```

10.3.1.2. PantallasAsignadas

En esta tabla, local a la rutina `inicializarObjeto()`, indicaremos con un 1 que la pantalla correspondiente ya tiene un objeto asignado, mientras que con un 0 indicaremos lo contrario.

```
int []pantallasAsignadas=new int[37];
```

10.3.1.3. TablaPlatos

Con esta tabla almacenamos los 10 platos a repartir por las 36 pantalla

```
private Objeto[] tablaPlatos=new Objeto[10];
```

10.3.1.4. TablaGalletas

Con esta almacenaremos 15 objetos del tipo 2 (galleta de la suerte).

```
private Objeto[] tablaGalletas=new Objeto[15];
```

10.3.1.5. TablaVidasExtras

En esta almacenaremos 5 instancias de la clase `Objeto` del tipo 3 (vida extra).

```
private Objeto[] tablaVidasExtras=new Objeto[5];
```

10.3.2. Inicialización de las tablas de objetos

Para cada tipo de objeto (1, 2 ó 3) tendremos una tabla distinta donde almacenar los 10 platos, las 15 monedas de la suerte y las 5 vidas extras. Lo primero que vamos a hacer en esta rutina será inicializar `pantallasAsignadas` con todas sus posiciones a 0 (no asignadas).

```
void inicializarObjetos(){
    int i,pantalla,pantalla2,pantallaAux;
    Random rnd=new Random();
    int []pantallasAsignadas=new int[37];

    for (i=1;i<=36;i++){
        pantallasAsignadas[i]=0;
    }
}
```

A continuación inicializaremos la tabla `pantallas` con las coordenadas en la cual se colocará el objeto. En cada posición tenemos tanto la coordenada X como la Y concatenadas en un único entero.

```
pantallas[1]=4184;  
pantallas[2]=59440;  
pantallas[3]=59528;  
pantallas[4]=55352;  
pantallas[5]=4112;  
pantallas[6]=57776;  
pantallas[7]=57360;  
pantallas[8]=47144;  
pantallas[9]=57392;  
pantallas[10]=53328;  
pantallas[11]=59416;  
pantallas[12]=4112;  
pantallas[13]=4176;  
pantallas[14]=55312;  
pantallas[15]=59496;  
pantallas[16]=41096;  
pantallas[17]=57352;  
pantallas[18]=2056;  
pantallas[19]=22656;  
pantallas[20]=57472;  
pantallas[21]=32904;  
pantallas[22]=51296;  
pantallas[23]=57384;  
pantallas[24]=4232;  
pantallas[25]=4224;  
pantallas[26]=4112;  
pantallas[27]=47240;  
pantallas[28]=4120;  
pantallas[29]=4144;  
pantallas[30]=51280;  
pantallas[31]=4112;  
pantallas[32]=17504;  
pantallas[33]=59480;  
pantallas[34]=59520;  
pantallas[35]=59400;  
pantallas[36]=47240;
```

Ahora toca el turno a la inicialización de las tres tablas que albergan los tres tipos de objetos diferentes: `tablaPlatos`, `tablaGalletas` y `tablaVidasExtras`. Como esta inicialización es similar para las tres tablas, sólo explicaremos aquí la de `tablaPlatos`.


```
tablaPlatos[0]=new Objeto(1);
tablaPlatos[0].añadirFrame(1,"/objeto0.png");

tablaPlatos[1]=new Objeto(1);
tablaPlatos[1].añadirFrame(1,"/objeto1.png");

tablaPlatos[2]=new Objeto(1);
tablaPlatos[2].añadirFrame(1,"/objeto2.png");

tablaPlatos[3]=new Objeto(1);
tablaPlatos[3].añadirFrame(1,"/objeto3.png");

tablaPlatos[4]=new Objeto(1);
tablaPlatos[4].añadirFrame(1,"/objeto4.png");

tablaPlatos[5]=new Objeto(1);
tablaPlatos[5].añadirFrame(1,"/objeto5.png");

tablaPlatos[6]=new Objeto(1);
tablaPlatos[6].añadirFrame(1,"/objeto6.png");

tablaPlatos[7]=new Objeto(1);
tablaPlatos[7].añadirFrame(1,"/objeto7.png");
tablaPlatos[7].setPantalla(35);

tablaPlatos[8]=new Objeto(1);
tablaPlatos[8].añadirFrame(1,"/objeto8.png");
tablaPlatos[8].setPantalla(22);

tablaPlatos[9]=new Objeto(1);
tablaPlatos[9].añadirFrame(1,"/objeto9.png");
tablaPlatos[9].setPantalla(29);
```

Con esto creamos una instancia de la clase objeto por cada plato, añadimos la imagen a utilizar para cada plato (ver Figura 8.1.) y a los tres últimos platos (7,8,9) le asignamos una pantalla fija. A continuación veremos la asignación de pantallas para los siete objetos restantes (0-6).

```
for (i=0;i<=6;i++){
    tablaPlatos[i].on();
    tablaPlatos[i].setTipo(1);
    pantalla=Math.abs(rnd.nextInt())/134217728)+1;
    pantalla2=Math.abs(rnd.nextInt())/536870912)+33;
```

```
if (pantallasAsignadas[pantalla]==0) {
    pantallasAsignadas[pantalla]=1;
    tablaPlatos[i].setPantalla (pantalla);

}else if (pantallasAsignadas[pantalla2]==0) {
    pantallasAsignadas[pantalla2]=1;
    tablaPlatos[i].setPantalla (pantalla2);
}else{
    pantallaAux=pantalla+1;

    while ((pantallaAux<=36) &&
        (pantallasAsignadas[pantallaAux]!=0))
        pantallaAux++;

    if (pantallaAux<=36) {
        pantallasAsignadas[pantallaAux]=1;
        tablaPlatos[i].setPantalla (pantallaAux);
    }else{
        pantallaAux=pantalla-1;

        while ((pantallaAux>=0) &&
            (pantallasAsignadas[pantallaAux]!=0)) {
            pantallaAux--;
        }

        pantallasAsignadas[pantallaAux]=1;
        tablaPlatos[i].setPantalla (pantallaAux);
    }
}
}
```

Para repartir los objetos por pantallas aleatorias, creamos un número aleatorio entre 1-32 y otro entre 33-36. Comprobamos si la primera pantalla sacada al azar (1-32) está ocupada, si es así haremos lo propio con la segunda pantalla (33-36) y si esta también lo está buscaremos la primera pantalla a partir del primer número creado (1-32) que esté libre (primero buscaremos en las pantallas posteriores y si no encontramos ninguno libre buscaremos en los anteriores a éste). Esto lo hacemos así por la imposibilidad, del compilador utilizado, de buscar un número aleatorio en un rango de números que no sea potencia de dos.

Veamos todos los gráficos de los objetos en la figura 10.1. (el índice junto a estos representan el orden en el cual deben de ser recogidos en el juego, excepto en los dos últimos que es irrelevante en el orden que los cojamos).













codigo	objeto
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	

Figura 10.1. Gráficos de los objetos a recoger en Cómeme.

10.3.3. Generación de objetos

Al contrario que en la generación de bloques y bichos, aquí no vamos a utilizar ninguna estructura de datos auxiliar donde almacenamos los datos de estos, sino que nos basta con saber que tipo de objeto es y en que pantalla está localizado (todo esto se hace en la inicialización de objetos). Veamos el código de este procedimiento:

```
void generacionObjetos() {
    int i=0;
    boolean encontrado=false;

    if (!generadoObjeto) {
        while ((i<10) && (!encontrado)) {

            if (tablaPlatos[i].getPantalla()==pantallaBicho) {
                encontrado=true;
            }
        }
    }
}
```

```
        if (encontrado) {
            objeto=tablaPlatos[i];
            objeto.setX(pantallas[pantallaBicho]/256);
            objeto.setX(objeto.getX()-posicionRelativa);
            objeto.setY(pantallas[pantallaBicho]%256);
            generadoObjeto=true;
        }else{
            i++;
        }
    }
}
```

Al igual que en la generación de bloques y bichos, si es la primera vez que entramos en este procedimiento, habrá que dar al objeto actual las coordenadas en la que se ubicará éste en su correspondiente pantalla (el resto de información se la dimos en la inicialización). En esta primera parte del código buscamos si el objeto de la pantalla actual es un plato, si lo es le asignamos las coordenadas almacenadas en la tabla pantallas y si no buscamos en las respectivas tablas de galletas y vidas extras.

```
i=0;
while ((i<15) && (!encontrado)){

    if (tablaGalletas[i].getPantalla()==pantallaBicho){
        encontrado=true;
    }

    if (encontrado){
        objeto=tablaGalletas[i];
        objeto.setX(pantallas[pantallaBicho]/256);
        objeto.setX(objeto.getX()-posicionRelativa);
        objeto.setY(pantallas[pantallaBicho]%256);
        generadoObjeto=true;
    }else{
        i++;
    }
}
```

```
i=0;
while ((i<5) && (!encontrado)){

    if (tablaVidasExtras[i].getPantalla()==pantallaBicho){
        encontrado=true;
    }

    if (encontrado){
        objeto=tablaVidasExtras[i];
        objeto.setX(pantallas[pantallaBicho]/256);
        objeto.setX(objeto.getX()-posicionRelativa);
        objeto.setY(pantallas[pantallaBicho]%256);
        generadoObjeto=true;
    }else{
        i++;
    }
}
```

En esta segunda parte del procedimiento buscamos, igual que en la primera, si el objeto es una galleta o una vida extra y asignamos las coordenadas correspondientes de la misma forma que hemos explicado para la primera parte del procedimiento.

```
}else if (hayScroll()){
    objeto.setX(objeto.getX()+indiceScroll);
}
}
```

En la tercera y última parte entraremos una vez hayamos asignado las coordenadas al objeto. Aquí entramos sólo si hay scroll, para mover el objeto los píxeles necesarios en el eje X.

10.3.4. Pintando los objetos

Al igual que con los bloques y bichos, una vez los objetos han sido inicializados y colocados en sus correspondientes coordenadas, toca pintarlos por pantalla.

```
public void pintarObjeto(Graphics g){
    if (generadoObjeto){
        if (objeto.estaActivo()){
            objeto.setX(objeto.getX());
            objeto.setX(objeto.getX());
            objeto.setY(objeto.getY());
            objeto.dibujar(g);
        }
    }
}
```

Para pintar el objeto, sólo pintamos uno por pantalla al contrario que con los bloque y bichos, comprobamos que esté activo, y si lo está le asignamos sus coordenadas correspondientes y lo dibujamos por pantalla.

CAPÍTULO 11: El personaje (JaimeNu)

En este capítulo vamos a ver todo lo referente a la animación de nuestro personaje. La rutina que utilizamos para ello es, con toda seguridad, la más complicada de todo el juego. Con esta rutina haremos que nuestro personaje ande, salte, se agarre, suba y baje escaleras e interaccione con el escenario. Lo primero que vamos a ver es la clase que da forma a nuestro personaje.

11.1. La clase JaimeNu

Al igual que todas las clase explicadas hasta ahora, la clase JaimeNu hereda de Sprite. Además de los atributos heredados de esta, la clase JaimeNu cuenta con una serie de atributos propios:

```
private int estado;  
private int direccion;  
private int direccionAnterior;
```

El atributo `estado` nos indica con un 0 que nuestro personaje está parado, con un 1 que está andando, con un 3 que el personaje está subido y parado en unas escaleras y con un 4 está subiendo o bajando unas escaleras. El atributo `direccion` nos indica con un 1 que nuestro personaje está mirando hacia la derecha y con un 2 mira hacia la izquierda. Con esto controlamos el *frame* o imagen a pintar en todo momento. El atributo `direccionAnterior` puede tener los mismos valores que `direccion`.

Veamos ahora el constructor de la clase:

```
public JaimeNu(int nFrames) {  
    super(nFrames);  
    direccion=1;  
    direccionAnterior=1;  
    estado=0;  
}
```

En un principio nuestro personaje estará parado y mirando hacia la derecha. Esto lo indicamos con `estado=0` y `direccion=1` respectivamente. El atributo `direccionAnterior` lo utilizamos para poder detectar en todo momento los cambios de sentido de nuestro personaje (si en un ciclo de nuestro *game loop*, `direccion` y `direccionAnterior` son diferentes, quiere decir que nuestro personaje ha cambiado de sentido).

Los siguientes métodos nos permiten tanto asignar como consultar los atributos de la clase JaimeNu:

```
public void setEstado(int estado){
    this.estado=estado;
}
public int getEstado(){
    return estado;
}
public void setDireccion(int direccion){
    this.direccion=direccion;
}
public int getDireccion(){
    return direccion;
}
public void setDireccionAnterior(int direccionAnterior){
    this.direccionAnterior=direccionAnterior;
}
public int getDireccionAnterior(){
    return direccionAnterior;
}
```

Para que nuestro personaje cambie de *frame*, dando la impresión de animación, tenemos que redefinir el método `dibujar(...)` de la clase padre `Sprite`. Esta quedaría de la siguiente forma:

```
public void dibujar(javax.microedition.lcdui.Graphics g){

    if (estado==1){
        if ((direccion==1) && (direccionAnterior==1)){
            if (frameActual()<4){
                actualizarFrame(frameActual()+1);
            }else{
                actualizarFrame(1);
            }
        }else if ((direccion==1) && (direccionAnterior==2)){
            actualizarFrame(1);
            direccionAnterior=1;
        }else if ((direccion==2) && (direccionAnterior==2)){
            if (frameActual()<8){
                actualizarFrame(frameActual()+1);
            }else{
                actualizarFrame(5);
            }
        }else if ((direccion==2) && (direccionAnterior==1)){
            actualizarFrame(5);
            direccionAnterior=2;
        }
    }
}
```



```
if (estado==4) {
    if (frameActual()==9) {
        actualizarFrame(10);
    }else{
        actualizarFrame(9);
    }
}
super.dibujar(g);
```

Si nuestro personaje está andando, `estado==1`, cambiaremos de *frame* de cuatro formas diferentes. Si nuestro personaje va hacia la derecha y su dirección anterior es también de derecha, pasaremos al siguiente *frame* de derechas, sin embargo si su dirección anterior es de izquierda (acaba de cambiar de sentido) tendremos que pasar al primer *frame* de izquierdas. Igual para el sentido contrario.

Si nuestro personaje está subiendo o bajando unas escaleras, `estado==4`, los *frames* a dibujar serán distintos. En este caso alternaremos sólo entre dos *frames*, con lo que después de dibujar uno seguirá el otro.

11.2. La clase JaimeNu en el Canvas

Veamos ahora todo lo relacionado con la clase JaimeNu dentro de nuestra clase principal SSCanvas. Primero vamos a ver la inicialización de nuestro personaje, luego veremos todas las rutinas auxiliares utilizadas en `actualizarPersonaje()` y por último la explicación de ésta última rutina, una de las más importantes del juego.

11.2.1. Inicialización del personaje

Al contrario que en las inicializaciones anteriores, aquí no nos valdremos de una tabla, ya que sólo necesitamos una instancia de la clase JaimeNu. Otra diferencia con las inicializaciones anteriores es que la carga de las imágenes (ver Figura 11.1.) no la realizamos aquí, sino un poco antes, en el constructor de la clase principal SSCanvas. Veamos como cargamos los *frames* de nuestro personaje:

```
jaimeNu.añadirFrame(1, "/personaje1a.png");
jaimeNu.añadirFrame(2, "/personaje2a.png");
jaimeNu.añadirFrame(3, "/personaje1a.png");
jaimeNu.añadirFrame(4, "/personaje3a.png");
jaimeNu.añadirFrame(5, "/personaje1b.png");
jaimeNu.añadirFrame(6, "/personaje2b.png");
jaimeNu.añadirFrame(7, "/personaje1b.png");
jaimeNu.añadirFrame(8, "/personaje3b.png");
jaimeNu.añadirFrame(9, "/personaje4a.png");
jaimeNu.añadirFrame(10, "/personaje4b.png");
```











nº frame	frame
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Figura 11.1. Frames del personaje.

Además de cargar todas la imágenes del personaje, asignaremos como *frame* actual de éste, el *frame* número 1 (ver Figura 11.1.) y lo posicionaremos en pantalla. Esto lo hacemos con las tres primeras líneas de la rutina `inicializarPersonaje()` mostrada a continuación:

```
void inicializarPersonaje() {
    jaimeNu.setX(32);
    jaimeNu.setY(126);
    jaimeNu.actualizarFrame(1);
    xAnterior=jaimeNu.getX();
    yAnterior=jaimeNu.getY();
    direccionAnterior=1;
    estadoAnterior=0;
    frameAnterior=1;
    enEscaleraAnterior=false;
    saltandoAnterior=false;
    contadorSaltoAnterior=0;
    scrollAnterior=36;
    posicionRelativaAnterior=0;
}
```

Todas las líneas que siguen a las tres primeras, están destinadas para restaurar al personaje en caso de perder una vida en la primera pantalla. Esto lo veremos con detenimiento más adelante en la rutina `restaurarPantalla()`.

11.2.2. Actualización del personaje

Antes de empezar a explicar la rutina `actualizarPersonaje()`, vamos a ver una serie de métodos que nos ayudarán a la hora de abordar la explicación de éste. Ahora es cuando necesitamos de la estructura explicada en el apartado 8.5. (`mapaPantalla`). Si hacemos memoria, esta matriz de *bytes*, de 20 filas y 32 columnas, almacenaba en cada casilla un 0 si no había nada en esa posición (en baja resolución), un 1 si había algo sólido, un 2 si había una escalera de huesos y un 3 si nos encontrábamos con un corazón de manzana (sólo en el modo de dificultad difícil).

Pues bien veamos la primera de estas rutinas:

```
byte derecha() {
    byte derecha=1, derecha1=1, derecha2=1, derecha3=1;
    int x=(jaimeNu.getX()+posicionRelativa+jaimeNu.getH())/8;
    int y1=jaimeNu.getY()/8;
    int y2=(jaimeNu.getY()+4+jaimeNu.getH()/2)/8;
    int y3=(jaimeNu.getY()+jaimeNu.getH()/2)/8;

    try{
        if ((x>0) && (x<32)){
            derecha1=mapaPantalla[x][y1];
            derecha2=mapaPantalla[x][y2];
            derecha3=mapaPantalla[x][y3];

            if ((derecha1==1) || (derecha2==1) ||
                (derecha3==1)){
                derecha=1;
            }else if ((derecha1==3) || (derecha2==3) ||
                (derecha3==3)){
                derecha=3;
            }else{
                derecha=0;
            }
        }else{
            derecha=0;
        }
    }catch(ArrayIndexOutOfBoundsException aioobe){
        System.out.println(aioobe.toString());
    }
    return derecha;
}
```

Esta rutina nos dice lo que hay a la derecha de nuestro personaje. No comprobamos si hay un bloque del tipo 2 (escalera de hueso), ya que lo trataremos como si no hubiera nada (tipo 0).

La siguiente rutina es similar a la anterior, la única diferencia es que comprueba lo que tenemos a la izquierda de nuestro personaje:

```
byte izquierda() {
    byte izquierda=1, izquierda1=1, izquierda2=1, izquierda3=1;
    int x=(jaimeNu.getX()+posicionRelativa-2)/8;
    int y1=(jaimeNu.getY()+4)/8;
    int y2=(jaimeNu.getY()+4+jaimeNu.getH()/2)/8;
    int y3=(jaimeNu.getY()+jaimeNu.getH()/2)/8;

    try{
        if (x<32){
            izquierda1=mapaPantalla[x][y1];
            izquierda2=mapaPantalla[x][y2];
            izquierda3=mapaPantalla[x][y3];

            if ((izquierda1==1) || (izquierda2==1) ||
                (izquierda3==1)){
                izquierda=1;
            }else if ((izquierda1==3) ||
                (izquierda2==3) || (izquierda3==3)){
                izquierda=3;
            }else{
                izquierda=0;
            }
        }else{
            izquierda=0;
        }
    }catch (ArrayIndexOutOfBoundsException aioobe){
        System.out.println(aioobe.toString());
    }
    return izquierda;
}
```

La siguiente es similar a las dos anteriores, la única diferencia es que consulta lo que tenemos debajo de nuestro personaje y si tiene en cuenta los bloques de tipo 2 (escaleras de hueso):

```
byte suelo() {
    int x,x1,x2,y;
    byte suelo=1;
    x=(jaimeNu.getX()+posicionRelativa+jaimeNu.getW()/2)/8;
    x1=(jaimeNu.getX()+posicionRelativa)/8;
    y=(jaimeNu.getY()+jaimeNu.getH())/8;
    x2=(jaimeNu.getX()+posicionRelativa+jaimeNu.getW())/8;
```

```
try{
    if ((y<20)){
        if (mapaPantalla[x][y]==1){
            suelo=1;

        }else if (mapaPantalla[x][y]==3){
            suelo=3;

        }else if (((mapaPantalla[x1][y]==0) &&
        (mapaPantalla[x2][y]==0)) ||
        ((mapaPantalla[x1][y]==0) &&
        (mapaPantalla[x2][y]==2)) ||
        ((mapaPantalla[x1][y]==2) &&
        (mapaPantalla[x2][y]==0))){
            suelo=0;
        }else if ((mapaPantalla[x1][y]==2) &&
        (mapaPantalla[x2][y]==2)){
            suelo=2;
        }
    }else{
        suelo=0;
    }
}catch(ArrayIndexOutOfBoundsException aioobe){
    System.out.println(aioobe.toString());
}
return suelo;
}
```

Esta rutina es prácticamente la misma que la anterior, pues consulta en el mapa de pantalla que tenemos sobre la cabeza de nuestro personaje:

```
byte techo(){
    int x,x1,x2,y;
    byte techo=1;
    x=(jaimeNu.getX()+posicionRelativa+jaimeNu.getWidth()/2)/8;
    x1=(jaimeNu.getX()+posicionRelativa)/8;
    y=(jaimeNu.getY()-2)/8;
    x2=(jaimeNu.getX()+posicionRelativa+jaimeNu.getWidth())/8;
```

```
try{
    if ((y>=0) && (x2<32)){
        if (mapaPantalla[x][y]==1){
            techo=1;

        }else if (mapaPantalla[x][y]==3){
            techo=3;

        }else if (((mapaPantalla[x1][y]==0) &&
        (mapaPantalla[x2][y]==0)) ||
        ((mapaPantalla[x1][y]==0) &&
        (mapaPantalla[x2][y]==2)) ||
        ((mapaPantalla[x1][y]==2) &&
        (mapaPantalla[x2][y]==0))) {
            techo=0;

        }else if ((mapaPantalla[x1][y]==2) &&
        (mapaPantalla[x2][y]==2)) {
            techo=2;
        }
    }else{
        techo=0;
    }
}catch (ArrayIndexOutOfBoundsException aioobe){
    System.out.println("comeme"+aioobe.toString());
}
return techo;
}
```

Las dos siguientes rutinas que vamos a ver, las utilizaremos para salir de unas escaleras de hueso sin tener que saltar, esto sólo será posible si hay una plataforma junto a las escaleras. Veamos la primera de ellas:

```
boolean haySueloAIzquierda(){
    int x=(jaimeNu.getX()+posicionRelativa-2)/8;
    int y1=(jaimeNu.getY()+8)/8;
    int y2=(jaimeNu.getY()-4+jaimeNu.getH())/8;
    int y3=(jaimeNu.getY()+jaimeNu.getH())/8;
    boolean hay=false;
```

```

        try{
            if ((y1<19) && (x>0)){
                if ((mapaPantalla[x][y1]==0) &&
                    (mapaPantalla[x][y2]==0) &&
                    (mapaPantalla[x][y3]==1)){
                    hay=true;
                }
            }
        }catch(ArrayIndexOutOfBoundsException aioobe){
            System.out.println(aioobe.toString());
        }
        return hay;
    }

```

Para saber si podemos salir de una escalera de huesos hasta una plataforma sin necesidad de saltar, consultamos en el mapa de pantalla lo que hay, en este caso a nuestra izquierda. Si lo que tenemos a la izquierda y bajo nuestros pies es un bloque de tipo 1 (bloque sólido) y a nuestra izquierda no hay nada, podremos pasar a la plataforma situada a la izquierda/abajo de nuestro personaje. La siguiente rutina es igual que la anterior, la única diferencia es que la comprobación de si hay plataforma junto a la escalera es a la derecha.

```

boolean haySueloADerecha(){

    intx=(jaimeNu.getX()+posicionRelativa+jaimeNu.getH()+2)/8;
    int y1=(jaimeNu.getY()+8)/8;
    int y2=(jaimeNu.getY()-4+jaimeNu.getH())/8;
    int y3=(jaimeNu.getY()+jaimeNu.getH())/8;
    boolean hay=false;

    try{
        if ((y1<19) && (x>0)){
            if ((mapaPantalla[x][y1]==0) &&
                (mapaPantalla[x][y2]==0) &&
                (mapaPantalla[x][y3]==1)){
                hay=true;
            }
        }
    }catch(ArrayIndexOutOfBoundsException aioobe){
        System.out.println(aioobe.toString());
    }
    return hay;
}

```

Al pasar de una pantalla a otra tenemos que borrar todos los datos relacionados con la pantalla actual y cargar los de la siguiente. De esto se encarga la rutina `borrarPantalla()`:

```
void borrarPantalla() {
    int i;
    generadoPantalla=false;
    generadoBicho=false;
    generadoObjeto=false;
    ciclo=0;
    ciclo2=0;
    direccionAnterior=jaimeNu.getDireccion();
    estadoAnterior=jaimeNu.getEstado();
    frameAnterior=jaimeNu.frameActual();
    enEscaleraAnterior=enEscalera;
    contadorSaltoAnterior=contadorSalto;
    saltandoAnterior=saltando;
    tipoSaltoAnterior=tipoSalto;

    if (pantallasVisitadas[pantallaBicho]==0) {
        puntuacion+=100;
        pantallasVisitadas[pantallaBicho]=1;
    }

    for (i=0;i<20;i++){
        if (tablaBloquesDecorados[i].estaActivo()){
            tablaBloquesDecorados[i].off();
            tablaBloquesDecorados[i].setBloquesRepetidos(0);
        }
    }

    for (i=0;i<24;i++){
        if (tablaBichos[i].estaActivo()){
            tablaBichos[i].off();
        }
    }
}
```

Esta rutina además de restablecer los valores de todas las variables implicadas en la generación, tanto de bloques como de bichos, y poner todos estos como inactivos, consulta en una *array* de 36 posiciones, `pantallasVisitadas[]`, si hemos pasado con anterioridad por esta pantalla, si no es así sumaremos 100 puntos al global de nuestra puntuación. Este array estará inicializado con un 0 en todas sus posiciones, mediante la rutina `inicializarPantallasVisitadas()`, indicando con esto que ninguna pantalla ha sido visitada.

Una vez vistas las 7 rutinas auxiliares, vamos con la explicación de la rutina del movimiento del personaje. Esta rutina básicamente se subdivide en dos grandes partes. En la primera de ellas entraremos si y sólo si nuestro personaje sigue dentro de la pantalla actual. Si es así moveremos al personaje según proceda. En la segunda parte, entraremos si nuestro personaje tiene algún píxel fuera de la pantalla, con lo que tendríamos que cambiar

de pantalla borrando ésta y restaurando una nueva.

```
void actualizarPersonaje() {
    if ((jaimeNu.getX()+posicionRelativa>=0) &&
        (jaimeNu.getX()+posicionRelativa<=256-
         jaimeNu.getH()+1) && (jaimeNu.getY()>=0) &&
        (jaimeNu.getY()<=getHeight()-2-jaimeNu.getH()*2)) {

        //código de la primera parte
        //personaje sigue dentro de pantalla
        if (saltando){
            //salto
        }else{
            //interacción del personaje con el entorno
        }
    }else{
        //código de la segunda parte
        //personaje sale de la pantalla actual
    }
}
```

Empecemos por la primera parte (nuestro personaje sigue dentro de la pantalla). Esta parte se basa en el movimiento del personaje en sí y la podemos subdividir en otras dos partes: una de ellas será el movimiento del salto y la otra todo lo demás, como por ejemplo andar, caer de una plataforma o encaramarse a unas escaleras de hueso. Seguiremos por el salto de nuestro personaje.

En nuestro tipo de salto una vez empezado éste no podremos abortarlo, a menos que nos encaramemos a una escalera de huesos, y ni siquiera cambiar la trayectoria de nuestro personaje. Para ello contaremos con una variable global denominada `contadorSalto`, que llevará la cuenta de los incrementos de nuestro salto, y otra, también global, denominada `saltando` que nos indica si en un momento dado estamos saltando o no. Veamos los diferentes valores que puede tener la variable global `contadorSalto` (ver tabla 11.1.) y la acción realizada en cada uno de ellos:

```
switch (contadorSalto){
    case 0:if (!enEscalera){
        if (deltaX==0){
            jaimeNu.setEstado(1);
            tipoSalto=0;

        }else if (deltaX>0){
            tipoSalto=1;

        }else if (deltaX<0){
            tipoSalto=2;
        }
    }
```

```
        }else{
            if ((tipoSalto==0) || (tipoSalto==1)){
                jaimeNu.setDireccion(1);
                jaimeNu.actualizarFrame(1);
            }else if (tipoSalto==2){
                jaimeNu.setDireccion(2);
                jaimeNu.actualizarFrame(5);
            }
            jaimeNu.setEstado(1);
        }

        deltaX=0;
        deltaY=-2;
        contadorSalto++;

        break;
```

Podemos iniciar el salto (`contador==0`) tanto si estamos en el suelo como si estamos encaramados a una escalera de huesos. Veamos las dos formas:

Si comenzamos el salto desde una plataforma, tendremos que ver hacia donde es el salto. Para saber que tipo de salto vamos a realizar tendremos que consultar el valor de una variable global denominada `deltaX`. Esta variable, que actualizamos cada vez que pulsamos los controles a derecha o a izquierda, será positiva si vamos hacia la derecha, negativa si vamos hacia la izquierda y cero si estamos parado. Una vez consultada esta variable estamos en disposición de saber que tipo de salto vamos a realizar. Esto lo controlaremos con otra variable global, `tipoSalto`, que valdrá 1 si el salto es hacia la derecha (`deltaX>0`), 2 si el salto es hacia la izquierda (`deltaX<0`) y 0 si no es ninguno de los dos anteriores (`deltaX==0`).

Si comenzamos a saltar desde una escalera de huesos, lo único que tenemos que hacer es consultar que tipo de salto estamos haciendo, al contrario que para el salto desde una plataforma. El tipo de salto se especifica en la pulsación de las teclas que explicaremos más adelante. Una vez sabemos que tipo de salto vamos a realizar lo único que tenemos que hacer es asignar la nueva dirección, el nuevo estado y el nuevo *frame* (este cambio no lo hace el método dibujar de la clase `JaimeNu`, ya que pasa desde las escaleras, estado 3 ó estado 4, a estar saltando, estado 1).

Al final del paso 0 de nuestro salto, aumentaremos el contador de salto en uno y asignaremos los incrementos a sumar al personaje.

Veamos el trozo de código para los valores 1, 3, 5, 6, 7, 8, 9, 11, 13 y 16:

```
case 1:
case 3:
case 5:
case 6:
case 7:
case 8:
case 9:
case 11:
case 13:
case 16: if (tipoSalto==1){
        deltaX=2;
    }else if (tipoSalto==2){
        deltaX=-2;
    }
    deltaY=-2;
    contadorSalto++;
    break;
```

Para estos valores del contador de salto, asignamos los incrementos `deltaX=2`, `deltaY=2` para salto de tipo 1, y `deltaX=-2`, `deltaY=2` para salto de tipo 2.

Sigamos con los valores 2 y 4:

```
case 2:
case 4: deltaY=-2;
        deltaX=0;
        contadorSalto++;
        break;
```

Asignamos los incrementos `deltaX=0` y `deltaY=-2` independientemente del tipo de salto.

Veamos los valores 10, 12, 14, 15, 17:

```
case 10:
case 12:
case 14:
case 15:
case 17: if (tipoSalto==1){
        deltaX=2;
    }else if (tipoSalto==2){
        deltaX=-2;
    }
    deltaY=0;
    contadorSalto++;
    break;
```

Para los saltos de tipo 1 asignamos $\text{deltaX}=2$ y $\text{deltaY}=0$, mientras que para los de tipo 2 los incrementos serán $\text{deltaX}=-2$ y $\text{deltaY}=0$.

Sean los valores 19, 22, 24, 26, 27, 28, 29, 30, 32, 34 del contador de salto:

```
case 19:
case 22:
case 24:
case 26:
case 27:
case 28:
case 29:
case 30:
case 32:
case 34: if (tipoSalto==1){
           deltaX=2;
        }else if (tipoSalto==2){
           deltaX=-2;
        }
        deltaY=2;
        contadorSalto++;
        break;
```

Sería igual que el anterior pero con los incrementos de $\text{deltaY}=2$. Veamos los valores 31 y 33:

```
case 31:
case 33: deltaX=0;
           deltaY=2;
           contadorSalto++;
           break;
```

Independientemente del tipo de salto, los incrementos, para estos dos valores del contador de salto, son 0 para el eje X y 2 para el Y.

Sean los valores 18, 20, 21, 23, 25:

```
case 18:
case 20:
case 21:
case 23:
```

```

case 25: if (tipoSalto==1){
        deltaX=2;
    }else if (tipoSalto==2){
        deltaX=-2;
    }
    deltaY=0;
    contadorSalto++;
    break;

```

Incremento de 2 en el eje X para los saltos de tipo 1 e incremento de -2 en el mismo eje para los saltos de tipo 2. No habrá incremento en el eje Y.

Y el último valor que puede tomar el contador de salto:

```

case 35: deltaX=0;
        deltaY=2;
        jaimeNu.setEstado(0);
        saltando=false;
        contadorSalto=0;
    }

```

En el último paso del salto, siempre y cuando no nos encaramos a unas escaleras de hueso, asignaremos los incrementos correspondientes, 0 para el eje X y 2 para el eje Y, colocaremos a nuestro personaje como *parado*, estado 0, haremos terminar el salto y pondremos el contador de salto a 0 (ver tabla 11.1.).

contadorSalto	tipoSalto=0	tipoSalto=1	tipoSalto=2
0	y=-2	x=0, y=-2	x=0, y=-2
1	y=-2	x=2, y=-2	x=-2, y=-2
2	y=-2	x=0, y=-2	x=0, y=-2
3	y=-2	x=2, y=-2	x=-2, y=-2
4	y=-2	x=0, y=-2	x=0, y=-2
5	y=-2	x=2, y=-2	x=-2, y=-2
6	y=-2	x=2, y=-2	x=-2, y=-2
7	y=-2	x=2, y=-2	x=-2, y=-2
8	y=-2	x=2, y=-2	x=-2, y=-2
9	y=-2	x=2, y=-2	x=-2, y=-2
10	y=0	x=2, y=0	x=-2, y=0
11	y=-2	x=2, y=-2	x=-2, y=-2
12	y=0	x=2, y=0	x=-2, y=0
13	y=-2	x=2, y=-2	x=-2, y=-2

14	y=0	x=2, y=0	x=-2, y=0
15	y=0	x=2, y=0	x=-2, y=0
16	y=-2	x=2, y=-2	x=-2, y=-2
17	y=0	x=2, y=0	x=-2, y=0

Tabla 11.1. Trayectoria de la primera mitad del salto.

El resto de la tabla sería simétrica pero con los índices y cambiados de signo. Por ejemplo para el valor 18 del contador de salto tendríamos los mismos incrementos que para el valor 17, ya que $y=0$. Para el valor 19 tendríamos los mismos incrementos que para el valor 16 pero con $y=2$, y así sucesivamente hasta 35 que tendría los mismos incrementos que para el valor 0 pero con $y=2$.

Los incrementos almacenados con anterioridad sólo se darán siempre y cuando no nos encontremos ningún obstáculo en el salto. Para eso antes de sumar los incrementos a la posición actual de nuestro personaje tenemos que hacer una serie de comprobaciones:

```

if ((tipoSalto==1) && (derecha()==1)){
    deltaX=0;
}
if ((tipoSalto==2) && (izquierda()==1)){
    deltaX=0;
}

```

Si estamos saltando y nos encontramos con un bloque del tipo 1 (sólido), tendremos que reasignar los incrementos en el eje X, ya que no podremos continuar en esta dirección.

Veamos otras restricciones:

```

if ((suelo()==1) && (techo()!=1) && (contadorSalto>1)){
    deltaX=0;

    if ((jaimeNu.getY()%8!=0) && (jaimeNu.getY()%8<0)){
        deltaY=-jaimeNu.getY()%8;
    }else{
        deltaY=0;
    }
    jaimeNu.setEstado(0);
    saltando=false;
    contadorSalto=0;
}else if ((techo()==1) && (suelo()==1)){
    deltaY=0;
}else if ((techo()==1) && (deltaY<0)){
    deltaY=0;
}

```

Si estamos saltando y tenemos suelo bajo nuestros pies, no tenemos techo sobre nuestra cabeza y ya hemos realizado los dos primeros pasos de nuestro salto (`contadorSalto=0` y `contador=1`), paramos el salto (`deltaX=0`, `estado=0`, `saltando=false`, `contadorSalto=0`). Para el incremento del eje Y hacemos un pequeño ajuste. Para que nuestro personaje siempre interactúe con el escenario de una manera correcta. Comprobamos si la posición de nuestro personaje en el eje Y es múltiplo de 8 y si no lo es hacemos que lo sea restando los píxeles que le queden. Con esto conseguiremos que nuestro personaje siempre esté justo encima de una plataforma y no solapado con ella.

En otro caso, si tenemos suelo sobre nuestros pies y techo sobre nuestra cabeza, nuestro personaje no subirá más en el salto, `deltaY=0`, aunque se deslizará en el eje X. Si sólo tenemos techo sobre nuestra cabeza y estamos subiendo, `deltaY<0`, no podremos continuar saltando, `deltaY=0`.

El siguiente trozo de código está relacionado con el *scroll*, así que lo dejaremos para la explicación de éste.

Sigamos con las restricciones:

```
if (!saltando){
    deltaX=0;
    deltaY=0;
    indiceScroll=0;
}

if (techo()==2){
    flagEscalera=true;
}else{
    flagEscalera=false;
}
```

Si hemos terminado el salto sin que el contador haya llegado a su fin, por ejemplo nos hemos subido a una plataforma, tendremos que inicializar de nuevo las variables `deltaX` y `deltaY`. Y para terminar con el salto, si nos encontramos en plena trayectoria de éste y tenemos en nuestra cabeza una escalera de huesos, asignaremos una variable global llamada `flagEscalera` a `true`, para indicar que nos podemos encaramar a ella.

Una vez terminado con el salto, vamos a seguir con la segunda subparte de la primera parte de la rutina `actualizarPersonaje()`, o lo que es lo mismo todo lo relacionado con la interacción de nuestro personaje y el escenario, siempre y cuando no nos encontremos saltando. Lo primero que vamos a hacer es comprobar que hay debajo de nuestro personaje, y dependiendo de lo que haya actuar en consecuencia.

```
switch (suelo()) {
    case 3:
    case 0: if (!enEscalera) {
        jaimeNu.setEstado(0);
        jaimeNu.setY(jaimeNu.getY()+2);
        enAire=true;
        deltaX=0;
        indiceScroll=0;
    } else if (deltaY>=0) {
        jaimeNu.setEstado(3);
    } else {
        jaimeNu.setEstado(4);
        jaimeNu.setY(jaimeNu.getY()+deltaY);
    }

    break;
```

Podemos distinguir dos modos al encontrarnos que no tenemos nada, o un corazón de manzana en el nivel de dificultad difícil, bajo los pies de nuestro personaje. Uno que no estamos en una escalera de huesos y el otro en el que sí estamos encaramados a una de estas escaleras. En el primer modo caeremos al vacío. Para que nuestro personaje parezca que cae de una plataforma, colocaremos a éste como parado, estado 0, incrementaremos, la posición del personaje, en el eje Y 2 píxeles, definiremos una nueva variable global que nos indica que estamos en el aire, `enAire=true`, y asignaremos a `deltaX` el valor 0 para que nuestro personaje no se mueva en el eje X. En el segundo modo estaremos subido en una escalera de huesos. Si no tenemos nada bajo nuestros pies, cuando estamos subido en unas de estas escaleras, significa que estamos en la parte inferior de ésta con lo que no podremos bajar más.

```
case 1: if (enAire) {
    enAire=false;
}

if ((enEscalera) && (deltaY>0)) {
    jaimeNu.setEstado(0);

    if (jaimeNu.getDireccion()==1) {
        jaimeNu.actualizarFrame(1)
    } else {
        jaimeNu.actualizarFrame(5);
    }

    enEscalera=false;
    deltaY=0;
    jaimeNu.setY(jaimeNu.getY()+deltaY);
}
```



```
if ((techo()==2) && (!enEscalera)){
    flagEscalera=true;
}else if (techo()!=2){
    flagEscalera=false;
}else if (enEscalera){
    jaimeNu.setY(jaimeNu.getY()+deltaY);
}
```

Si lo que tenemos bajo nuestro personaje es un bloque sólido (tipo 1), lo primero que vamos a hacer es ver si con anterioridad estábamos en el aire, si es así asignaremos a la variable global `enAire` el valor `false`. Por otro lado si estamos subidos en una escalera de huesos y tocamos suelo podremos bajar a éste con el correspondiente cambio de *frame* (pasa de estar en una escalera a estar en una plataforma). Una vez en una plataforma, si tenemos en nuestra cabeza una escalera de huesos, indicaremos con la variable global `flagEscalera`, comentada con anterioridad, que podemos engancharnos a una de estas escaleras.

Una vez en el suelo y moviéndonos por él, tendremos que comprobar las colisiones de nuestro personaje con los distintos bloques tanto a derechas como a izquierdas:

```
if (jaimeNu.getDireccion()==1){
    switch (derecha()){
        case 3:
        case 0:if (!hayScroll()){
            if (jaimeNu.getDireccionAnterior()==1){
                jaimeNu.setX(jaimeNu.getX()+deltaX);
            }else{
                indiceScroll=0;
            }
        }else if(jaimeNu.getDireccionAnterior()==2){
            indiceScroll=0;
            jaimeNu.setX(jaimeNu.getX()+deltaX);
            jaimeNu.setDireccionAnterior(1);
            jaimeNu.actualizarFrame(1);
        }else{
            indiceScroll=-deltaX;
        }
        jaimeNu.setY(jaimeNu.getY()+deltaY);
        break;

        case 1: indiceScroll=0;
            break;
    }
}
```

Si vamos hacia la derecha, y nos encontramos con el camino libre o con una escaleras de hueso o con un corazón de manzana, esto último sólo se puede dar en el nivel

de dificultad difícil, básicamente aumentaremos la posición de nuestro personaje en `deltaX` si no hemos cambiado de sentido, en caso contrario, que hayamos cambiado de sentido, el personaje se quedará donde está cambiando sólo su imagen hacia izquierda. Esto es si no hay *scroll*, si lo hubiera habría que modificar unas variables pertenecientes a éste que veremos más adelante cuando lo expliquemos. También ajustaremos la coordenada Y, si es necesario, sumando a la posición actual de nuestro personaje `deltaY`.

Si lo que nos encontramos es un bloque sólido (tipo 1), no modificaremos la posición de nuestro personaje, con lo que este se quedará en el mismo sitio y con el mismo estado, andando, dándonos la sensación de que no puede atravesar el bloque.

A izquierdas será similar a lo explicado para derechas, la única diferencia es que el `deltaX` será negativo y que cambiaremos al *frame* 5 en lugar de al *frame* 1 si cambiamos de sentido.

```
}else if (jaimeNu.getDireccion()==2) {
    switch (izquierda()) {
        case 3:
            case 0: if (!hayScroll()) {
                if (jaimeNu.getDireccionAnterior()==2) {
                    jaimeNu.setX(jaimeNu.getX()+deltaX)
                } else {
                    indiceScroll=0;
                }
            } else if (jaimeNu.getDireccionAnterior()==1) {
                indiceScroll=0;
                jaimeNu.setX(jaimeNu.getX()+deltaX);
                jaimeNu.setDireccionAnterior(2);
                jaimeNu.actualizarFrame(5);
            } else {
                indiceScroll=-deltaX;
            }
            jaimeNu.setY(jaimeNu.getY()+deltaY);
            break;
        case 1: indiceScroll=0;
            break;
    }
}
```

Por último vamos a ver como interacciona nuestro personaje cuando nos encontramos con una escalera de huesos bajos sus pies.

```

        case 2:if (!enEscalera){
            enAire=true;
            jaimeNu.setEstado(0);
            jaimeNu.setY(jaimeNu.getY()+2);
            deltaX=0;
            indiceScroll=0;
        }else if ((deltaX==2) || (deltaX== -2)){
            if (!hayScroll()){
                jaimeNu.setX(jaimeNu.getX()+deltaX);
            }else if (jaimeNu.getDireccionAnterior() !=
                jaimeNu.getDireccion()){
                indiceScroll=0;
                jaimeNu.setX(jaimeNu.getX()+deltaX);
            }else{
                indiceScroll=-deltaX;
            }
            enEscalera=false;
        }else if ((techo()!=2) && (deltaY<=0)){
            jaimeNu.setEstado(3);

        }else{
            jaimeNu.setY(jaimeNu.getY()+deltaY);
        }

        if (techo()==2){
            flagEscalera=true;
        }else{
            flagEscalera=false;
        }
        break;
    }
}

```

Si nos encontramos unas escaleras de huesos debajo de los pies de nuestro personaje, y éste no está subido en ella, caeremos por esta igual que si no hubiera nada (igual que para tipo 0). Si estamos subido en la escalera, y `deltaX==2` ó `deltaX== -2` podremos salir de las escaleras. Si estamos subidos en unas escaleras y la variable `deltaX` tiene asignado uno de estos dos valores, es porque hemos pulsado la tecla derecha o izquierda cuando hay una plataforma junto a las escaleras (ver rutinas `haySueloADerechas()`, `haySueloAIzquierda()` al comienzo de este apartado).

Por otro lado si estamos subido en una de estas escaleras y lo que tenemos encima de nuestro personaje no es escalera, no podremos subir más. Como consecuencia de esto, no moveremos al personaje (estado 3).

Una vez hemos comprobado todos los bloques que nos rodea y actuado en consecuencia, queda comprobar si estamos saltando y hemos pulsado la tecla *arriba*. Si es

así, y la variable `flagEscalera` es `true`, nuestro personaje subirá por ellas abortando el salto.

```
if ((flagEscalera) && (arribaPulsado)){
    if ((saltando) && (!enEscalera)){
        jaimeNu.setEstado(4);
        jaimeNu.actualizarFrame(9);
        saltando=false;
        contadorSalto=0;
        deltaX=0;
        indiceScroll=0;
        enEscalera=true;
        enAire=false;
        deltaY=-4;

    }else if (!enEscalera){
        jaimeNu.setEstado(4);
        jaimeNu.actualizarFrame(9);
        enEscalera=true;
        enAire=false;
        deltaY=-4;
    }
}
```

La variable `arribaPulsado`, es una variable global que nos indica si estamos pulsando la tecla *arriba*, ya que los métodos utilizados sólo recogen la acción al pulsar y soltar la tecla (ver capítulo 4, apartado 4.2.). Con esto terminamos la primera parte de la rutina `actualizarPersonaje()`.

Vamos a comenzar la segunda parte de nuestra rutina, en la cual veremos cuando nuestro personaje alcanza uno de los bordes de la pantalla para salir de ésta y entrar en otra. Podemos salir de una pantalla por su parte derecha o por su parte izquierda o por la parte de abajo o por la de arriba. Dependiendo por la parte de la pantalla que salgamos, si se puede, tendremos que cargar una nueva pantalla y borrar la anterior.

```
}else if (jaimeNu.getX()+posicionRelativa>256-
    jaimeNu.getH()+1){
    pantallaBicho+=1;
    scroll=0;
    scrollAnterior=0;
    posicionRelativa=0;
    posicionRelativaAnterior=0;
    xAnterior=0;
    yAnterior=jaimeNu.getY();
    jaimeNu.setX(0);
    borrarPantalla();
```

Con este trozo de código controlamos que nuestro personaje salga por la parte derecha de la pantalla, asignamos a la variable global `pantallaBicho` (esta variable lleva la pantalla actual a cargar) su valor más uno, modificamos la posición en el eje X del personaje, la del eje Y no variará, guardamos todas las variables, como `scroll` y `posicionRelativa` en sus respectivas variables con sufijo `Anterior` (como `scrollAnterior` o `posicionRelativaAnterior`), por si perdemos una vida en esta pantalla y tenemos que restaurarla (con la rutina `restaurarPantalla()`), y por último borramos la pantalla anterior con la rutina `borrarPantalla()`.

Veamos el trozo de código por si salimos por la parte izquierda de la pantalla actual:

```
}else if (jaimeNu.getX()+posicionRelativa<0){
    pantallaBicho-=1;
    scroll=252;
    scrollAnterior=252;
    posicionRelativa=76;
    posicionRelativaAnterior=76;
    xAnterior=165;
    yAnterior=jaimeNu.getY();
    jaimeNu.setX(165);
    borrarPantalla();
```

La única diferencia con la salida por la derecha son los valores que toman las variables, ya que ahora nuestro personaje estará en la parte derecha de la nueva pantalla y no en la izquierda como en la anterior.

Veamos cuando salimos por la parte de abajo de una pantalla:

```
}else if (jaimeNu.getY()>(getHeight()-2-
    jaimeNu.getH()*2)){
    pantallaBicho+=6;
    scrollAnterior=scroll;
    posicionRelativaAnterior=posicionRelativa;
    xAnterior=jaimeNu.getX();
    yAnterior=0;
    jaimeNu.setY(0);
    borrarPantalla();
```

Ahora en lugar de modificar la posición del personaje en el eje X, lo hacemos en el eje Y, ya que de estar en la parte inferior de la pantalla actual pasamos a la parte superior de la pantalla actual más 6.

Y por último veamos el código para pasar a la pantalla de arriba que es similar al anterior.

```
    }else if (jaimeNu.getY()<getHeight()){
        pantallaBicho-=6;
        posicionRelativaAnterior=posicionRelativa;
        scrollAnterior=scroll;
        xAnterior=jaimeNu.getX();
        yAnterior=getHeight()-(jaimeNu.getH()*2+2);
        jaimeNu.setY(getHeight()-(jaimeNu.getH()*2+2));
        borrarPantalla();
    }
}
```

11.2.3. Pintando al personaje

Al igual que con los bloques, bichos y objetos, una vez posicionado nuestro personaje toca pintarlo. Además de pintar a nuestro personaje, actualizamos las variables globales `scroll` y `posicionRelativa` que harán posible que tengamos un *scroll* en nuestro juego (ver capítulo 13, apartado 13.1).

```
public void pintarPersonaje(Graphics g){

    jaimeNu.setX(jaimeNu.getX());

    if (((jaimeNu.getX()>0) &&
        (jaimeNu.getX()+posicionRelativa<240)) &&
        (jaimeNu.getY()%8!=0) && (suelo()==1)){

        jaimeNu.setY(jaimeNu.getY()-jaimeNu.getH());
    }

    jaimeNu.setY(jaimeNu.getH());
    jaimeNu.dibujar(g);

    if (!hayScroll()){
        scroll=jaimeNu.getX()+posicionRelativa;
    }else{
        if (jaimeNu.getDireccion()!=
            jaimeNu.getDireccionAnterior()){

            indiceScroll=0;
        }
        posicionRelativa+=-indiceScroll;
        scroll=jaimeNu.getX()+posicionRelativa;
    }
}
```

Antes de posicionar la coordenada Y en la que se encuentra nuestro personaje, hacemos un arreglo para que este pise de forma *real* el suelo y no se introduzca en él.

CAPÍTULO 12: Pié de pantalla

Ya hemos dibujado los bloques que forman nuestra pantalla, los objetos, los bichos y a nuestro personaje, lo único que nos queda es el marcador de puntuación y de vidas restantes situado en el pie de la pantalla. Este marcador además contendrá el nombre de la pantalla actual.

12.1. Inicialización del marcador

Tanto el cuadro gráfico de diálogo, que da forma al marcador, como unos puntos que utilizaremos a modo de contador de vidas, son instancias de la clase `Sprite` y son inicializadas y activadas en el constructor de la clase principal, `SSCanvas`.

```
dialogo.añadirFrame(1, "/dialogo.png");  
vida.añadirFrame(1, "/vidas.png");  
  
vida.on();  
dialogo.on();
```

Además sobre el cuadro de diálogos tendremos el nombre de la pantalla durante 100 vueltas del *game loop*, momento en el que cambiaremos la información del nombre de la pantalla por el número de vidas y la puntuación. Al pasar otros 100 ciclos volveremos a la información del nombre de pantalla y así sucesivamente. Esto es así porque toda la información no cabe en el cuadro de diálogo a la vez (ver Figura 12.1.).

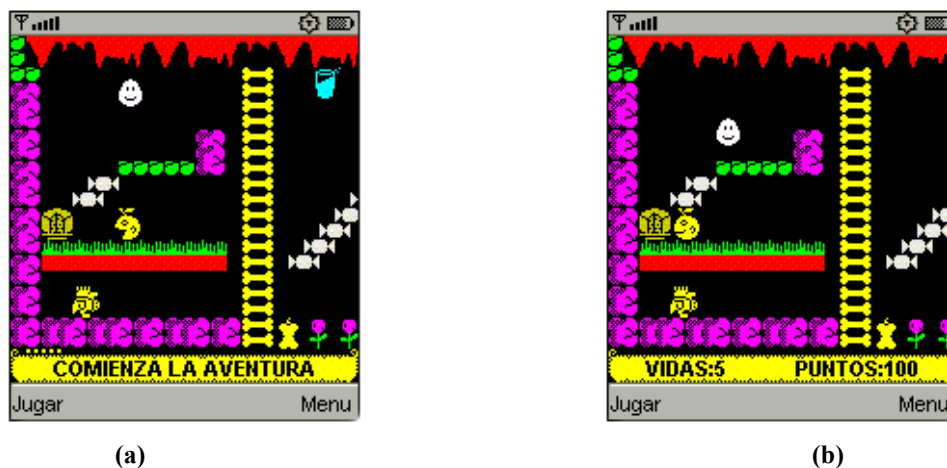


Figura 12.1. Pié de pantalla.

En la figura 12.1. (a) mostramos el nombre de la pantalla actual y un indicador de vidas gráfico, mientras que en la figura 12.1. (b) mostramos la puntuación y el indicador de vidas.

12.2. Pintando el marcador

Para pintar nuestro marcador lo único que tenemos que hacer es crear una fuente para el texto a escribir (ver capítulo 3, apartado 3.3.3), posicionar el cuadro de diálogo y dibujarlo, y por último pintar el nombre de la pantalla y el indicador de vidas gráfico o el indicador de vidas y puntos.

```
public void pintarMarcador(Graphics g){
    int i;
    String texto="? ? ? ? ? ? ? ?";
    Font fuente=Font.getFont
        (Font.FACE_PROPORTIONAL,Font.STYLE_BOLD,
         Font.SIZE_MEDIUM);
    dialogo.setX(0);
    dialogo.setY(getHeight()-dialogo.getH());
    dialogo.dibujar(g);
    g.setFont(fuente);
}
```

Con este trozo de la rutina creamos la fuente a utilizar y pintamos el cuadro de diálogo.

```
if (ciclo2<=100){
    switch(pantallaBicho){
        case 1:texto=texto1;
            . break;
            .
            .
        case 36:texto=texto36;
    }
    g.drawString(texto,getWidth()/2,getHeight(),
    Graphics.BOTTOM| Graphics.HCENTER);
    vida.setX(8);
    vida.setY(160);

    for (i=1;i<=vidas;i++){
        vida.dibujar(g);
        vida.setX(vida.getX()+4);
    }
}
```

Con esto dibujamos tanto el nombre de la pantalla como el indicador gráfico de vidas. Por último pintaríamos, si toca, el indicador de vidas y puntos:

```
}else{  
    g.drawString("VIDAS:"+vidas+"PUNTOS:"+puntuacion,  
        getWidth()/2,getHeight(),  
        Graphics.BOTTOM|Graphics.HCENTER);  
}  
}
```


CAPÍTULO 13: Scroll, teclas y otros

Han sido muchas las veces que hemos hablado sobre el *scroll* y sus variables en los capítulos anteriores. En este capítulo vamos a explicar todo lo relacionado con nuestro *scroll* y la pulsación de teclas, estrechamente relacionado con éste, y por último veremos las rutinas que se encargan tanto de las colisiones, entre personaje-bicho, personaje-objeto y personaje-corazón de manzana (esta última sólo en el nivel de dificultad difícil), como de restaurar la pantalla.

13.1. Scroll

A la hora de abordar este proyecto utilizamos como tamaño de pantalla la del emulador suministrado por Sun, esta pantalla tiene por dimensiones 180 píxeles de ancho por 177 píxeles de altura. Con tales dimensiones tenemos el problema de que nuestros escenarios tienen una anchura superior a la del emulador, concretamente 256 píxeles. Para solucionar esto nos valemos de un *scroll* horizontal.

La idea de nuestro *scroll* es la siguiente: nuestro personaje caminará por la pantalla hasta alcanzar, aproximadamente, la parte central de ésta. Una vez aquí desplazaremos los bloques de nuestra pantalla, en el eje X, para dar la sensación de movimiento. Si hay *scroll* y nuestro personaje se encuentra andando, éste no se desplazará, ya que con el desplazamiento de los bloques de pantalla nos dará la sensación de que el que se está moviendo es el personaje (movimiento relativo), cuando en realidad es la pantalla. Para la creación del *scroll* vamos a valernos de una serie de variables globales:

- `scroll`, es la suma de la posición real y relativa de nuestro personaje.
- `posicionRelativa`, lleva el desplazamiento de los bloques de decorados cuando hay *scroll*.
- `indiceScroll`, almacena los píxeles que se moverán los bloques de decorado en el ciclo actual, siempre y cuando haya *scroll*.

En todo momento sabremos si hay *scroll*, consultando la rutina `hayScroll()`:

```
boolean hayScroll() {  
    return ((scroll>=90) && (scroll<=166));  
}
```

Un ejemplo gráfico de lo explicado lo encontramos en la figura 13.1.

13.2. Interfaz con el jugador (teclas)

En este apartado veremos las acciones asociadas cada vez que pulsamos o soltamos una de las teclas válidas del juego. Éste apartado está estrechamente ligado tanto al *scroll* del escenario como a la actualización del personaje (ver capítulo 11, apartado 11.2.2.).

Empezaremos por la pulsación de teclas:

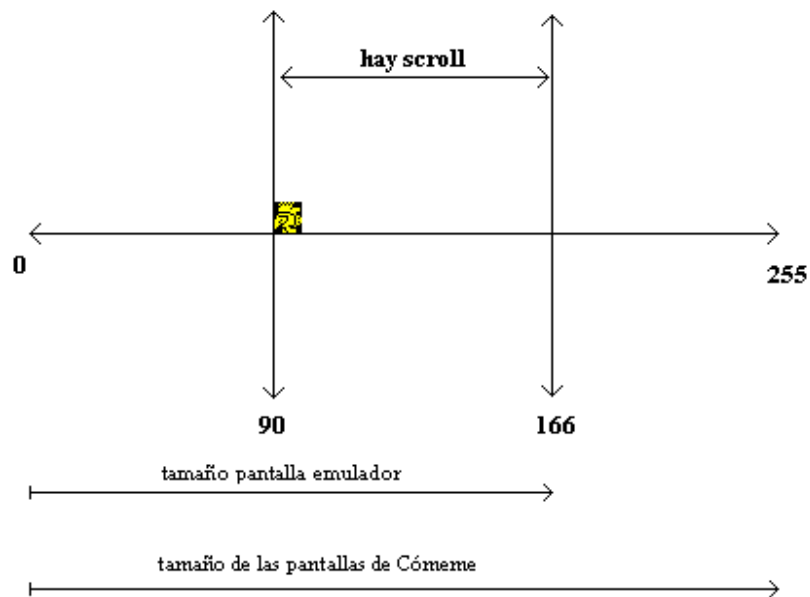


Figura 13.1. Gráfico sobre scroll.

```
public void keyPressed(int keyCode){
    int accion=getGameAction(keyCode);
    switch (accion){
        case LEFT: if ((!enAire) && (!saltando)){
                    if (!enEscalera){
                        deltaX=-2;
                        indiceScroll=2;
                        jaimeNu.setDireccionAnterior(
                            jaimeNu.getDireccion());
                        jaimeNu.setDireccion(2);
                        jaimeNu.setEstado(1);
                    }else if (haySueloAIzquierda()){
                        deltaX=-2;
                        indiceScroll=2;
                        jaimeNu.setDireccionAnterior(
                            jaimeNu.getDireccion());
                        jaimeNu.setDireccion(2);
                        jaimeNu.setEstado(1);
                        jaimeNu.actualizarFrame(5);
                    }else{
                        tipoSalto=2;
                    }
                }
        break;
    }
```

Cuando pulsamos la tecla asignada para el movimiento a izquierdas, siempre y cuando no estemos cayendo o saltando y no nos encontremos encaramados a ninguna escaleras de huesos, asignaremos a `deltaX` el valor `-2` (para el movimiento del personaje), a `indiceScroll` el valor `2` (igual que el anterior pero para el escenario), colocaremos como dirección anterior de nuestro personaje la dirección actual, a `direccion` le asignaremos el valor `2` (a izquierdas) y por último colocaremos a nuestro personaje en el estado `1` (en movimiento). Si nos encontramos subidos a unas escaleras, tendremos que comprobar si hay una plataforma a nuestra izquierda. En caso afirmativo haremos lo mismo que antes, con la única diferencia de que forzamos el cambio de *frame* (pasamos de estar en unas escaleras a estar en una plataforma) y en caso negativo asignaremos a la variable global `tipoSalto` el valor `2` (salto a izquierdas. Ver capítulo 11, apartado 11.2.2.).

Veamos la acción a realizar cuando pulsamos la tecla asignada para movimiento a derechas:

```
case RIGHT:if ((!enAire) && (!saltando)){
    if (!enEscalera){
        deltaX=2;
        indiceScroll=-2;
        jaimeNu.setDireccionAnterior(
            jaimeNu.getDireccion());
        jaimeNu.setDireccion(1);
        jaimeNu.setEstado(1);
    }else if (haySueloADerecha()){
        deltaX=2;
        indiceScroll=-2;
        jaimeNu.setDireccionAnterior(
            jaimeNu.getDireccion());
        jaimeNu.setDireccion(1);
        jaimeNu.setEstado(1);
        jaimeNu.actualizarFrame(1);
    }else{
        tipoSalto=1;
    }
}
break;
```

Básicamente es el mismo código que para el movimiento a izquierdas, la única diferencia son los valores tomados por algunas variables. Por ejemplo, el movimiento del personaje será a derechas (incremento positivo), la dirección será `1` (derecha) y el tipo de salto será `1` (salto a derechas).

Por último vamos a ver las acciones asociadas a las pulsaciones de las teclas *arriba*, *abajo* y *salto*:

```

case UP: if (enEscalera){
            jaimeNu.setEstado(4);
            deltaY=-4;
        }
        arribaPulsado=true;
        break;

case DOWN: if (enEscalera){
            jaimeNu.setEstado(4);
            deltaY=4;
        }
        break;
case FIRE: if ((!enAire) && (!saltando)){
            saltando=true;
        }
    }
}

```

Si pulsamos la tecla *arriba* y estamos subido a unas escaleras subiremos por ésta, incrementos en el eje Y negativo, mientras que si pulsamos la tecla *abajo* bajaremos, incrementos en el eje Y positivo. Si pulsamos la tecla *salto*, y no nos encontramos cayendo de una plataforma ni saltando, pondremos la variable global saltando a `true` (ver capítulo 11, apartado 11.2.2.).

Al soltar cualquiera de las teclas de juego también realizaremos una serie de acciones que nos permitirán el correcto funcionamiento del movimiento de nuestro personaje:

```

public void keyReleased(int keyCode){
    int accion=getGameAction(keyCode);

    switch (accion){
        case LEFT:if ((!enAire) && (!saltando)){
                    if ((!enEscalera) || ((enEscalera) &&
                        (haySueloAIzquierda()))){

                        deltaX=0;
                        indiceScroll=0;
                        jaimeNu.setEstado(0);
                        enEscalera=false;
                    }
                    tipoSalto=0;
                }
            break;
    }
}

```

Al soltar la tecla *izquierda* comprobaremos si nos podemos mover, no estamos en el aire o no estamos subidos a unas escaleras o si lo estamos y tenemos una plataforma a

nuestra izquierda, y si es así detendremos a nuestro personaje, $\text{deltaX}=0$, o a los bloques de decorados, $\text{indiceScroll}=0$, dependiendo si hay *scroll* o no.

```
case RIGHT: if ((!enAire) && (!saltando)){
    if ((!enEscalera) || ((enEscalera)
        && (haySueloADerecha()))){
        deltaX=0;
        indiceScroll=0;
        jaimeNu.setEstado(0);
        enEscalera=false;
    }
    tipoSalto=0;
}
break;
```

Igual que cuando soltamos la tecla izquierda, la única diferencia es que cuando estamos subidos a unas escaleras de huesos, comprobamos si tenemos una plataforma a nuestra derecha y no a nuestra izquierda como para el trozo anterior.

```
case UP: arribaPulsado=false;
case DOWN: if (enEscalera){
    jaimeNu.setEstado(3);
    deltaY=0;
}

if (!saltando){
    tipoSalto=0;
}
break;
}
}
```

Si dejamos de pulsar tanto la tecla *arriba* como la tecla *abajo*, detendremos a nuestro personaje en la escalera, estado 3 y $\text{deltaY}=0$.

13.3. Colisiones entre objetos

Vimos en el capítulo 11, apartado 11.2.2., como nuestro personaje interaccionaba con el escenario, pero lo que no vimos era como lo hacía con los bichos y objetos. Veamos como tratamos las colisiones entre personaje-bicho:

```
void colisiones() {
    int i;
    Bicho bicho;
```

```
for (i=0;i<=23;i++){
    bicho=tablaBichos[i];

    if ((bicho.estaActivo()) &&
        (jaimeNu.colision(bicho))){
        try{
            Thread.sleep(500);
        }catch (InterruptedException ie){
            System.out.println(ie.toString());
        }
        restaurarPantalla();
    }
}
```

Al colisionar con alguno de los bichos activos en nuestra pantalla actual, detendremos la ejecución del programa durante 500 milisegundos para luego restaurar la pantalla (ver apartado 13.4.). Veamos el código que trata las colisiones con objetos de tipo 1 (platos):

```
if ((objeto.estaActivo()) && (jaimeNu.colision(objeto))){
    switch (objeto.getTipo()){
        case 1:if (tablaPlatos[contadorComidas].getPantalla()
            == objeto.getPantalla()){

                objeto.off();
                contadorComidas++;
                puntuacion+=1000;
            }else{
                restaurarPantalla();
            }

            if (contadorComidas==10){
                playing=false;
            }else if (contadorComidas==7){
                pantallaBicho=23;
                scroll=16;
                scrollAnterior=16;
                posicionRelativa=0;
                posicionRelativaAnterior=0;
                xAnterior=16;
                yAnterior=jaimeNu.getY();
                jaimeNu.setX(16);
                borrarPantalla();
            }
            break;
    }
}
```

Si colisionamos con el objeto actual, almacenado en la variable global `objeto`, y

es el objeto que toca coger, quitaremos éste de la pantalla, aumentaremos el contador de comidas recogidas y sumaremos 1000 a la puntuación actual. En otro caso si no es el objeto que toca, restauramos la pantalla (perdemos una vida). Si ya hemos cogido 7 platos nos teletransportaremos a las pantallas donde se encuentran los 3 últimos platos, inaccesibles de otra forma, mientras que si son 10 los platos recogidos el juego terminará.

Veamos la colisión con los objeto de tipo 2 (galletas de la suerte) y tipo 3 (vidas extras).

```

        case 2: objeto.off();
                Random rnd=new Random();
                int puntos=Math.abs(rnd.nextInt())/536870912)+1;
                puntuacion+=puntos*100;
                break;

        case 3: objeto.off();
                if (vidas<10){
                        vidas++;
                }
        }

        try{
                Thread.sleep(500);
        }catch (InterruptedException ie){
                System.out.println(ie.toString());
        }
}

```

Las galletas nos darán una puntuación que oscila entre los 100 y 800 puntos, mientras que los objetos de tipo 3 nos dará una vida de más, siempre y cuando no sobrepasemos las 9. Tras la recogida de cualquiera de los tres objetos anteriores pararemos la ejecución de nuestro programa durante 500 milisegundos. Y por último veamos las colisiones de nuestro personaje con los corazones de manzana (sólo en el nivel de dificultad difícil).

```

if (difícil){
        if ((tablaBloquesDecorados[16].estaActivo()) &&
            (jaimeNu.colisionBloque(tablaBloquesDecorados[16]))){
                try{
                        Thread.sleep(500);
                }catch (InterruptedException ie){
                        System.out.println(ie.toString());
                }
                restaurarPantalla();
        }
}
}

```

Al colisionar con un corazón de manzana perderemos una vida con la consiguiente restauración de la pantalla.

13.4. Restauración de pantalla

Al perder una vida en nuestro juego, nuestro personaje será colocado en la misma pantalla y de la misma forma en la que entró. Para que esto sea posible tendremos que guardar los valores de todas las variables necesarias, en el momento de cambiar de pantalla, para la posible restauración. Para esto habrá que borrar la pantalla actual y crearla con los mismos parámetros con los que nuestro personaje entró en ella, por ejemplo el *scroll* (ver capítulo 11, apartado 11.2.2.). Veamos el código que realiza lo explicado.

```
void restaurarPantalla() {
    jaimeNu.setX(xAnterior);
    jaimeNu.setY(yAnterior);
    deltaX=0;
    deltaY=0;
    indiceScroll=0;
    scroll=scrollAnterior;
    posicionRelativa=posicionRelativaAnterior;
    jaimeNu.setDireccionAnterior(direccionAnterior);
    jaimeNu.setDireccion(direccionAnterior);

    if ((estadoAnterior==1) && (!saltandoAnterior)){
        estadoAnterior=0;
    }else if (estadoAnterior==4){
        estadoAnterior=3;
    }

    jaimeNu.setEstado(estadoAnterior);
    jaimeNu.actualizarFrame(frameAnterior);
    saltando=saltandoAnterior;
    tipoSalto=tipoSaltoAnterior;
    contadorSalto=contadorSaltoAnterior;
    enEscalera=enEscaleraAnterior;
    generadoBicho=false;
    inicializarBichos();
    ciclo=0;
    vidas--;

    if (vidas<=0){
        playing=false;
    }
    borrarPantalla();
}
```

CAPÍTULO 14: El menú

Hasta ahora hemos visto todo lo relacionado con la clase `SSCanvas` y las clases principales usadas en ella (las que heredan de `Sprite`). Pero el *canvas* sólo se mostrará por pantalla cuando elijamos la opción de nuestro menú *jugar*. Veamos como es este menú y las opciones que contiene.

14.1. La clase `Comeme`

Esta es la clase que hereda de `MIDlet`, y por tanto la que nos permite que nuestro proyecto se ejecute en un dispositivo móvil. Veamos sus atributos:

```
Public class Comeme extendí MIDlet implements CommandListener{
    private Command exitCommand, playCommand, endCommand;
    private Display display;
    private SSCanvas screen;
    private Dificultad dificultad;
    private Menu menu;
    private Resultado resultado;
    private NuevoRecord nuevoRecord;
    private Alerta alerta;
    private Musica musica;
    private String sonido="No";
    private String nivel="Normal";
```

Tenemos tres comandos que utilizaremos en nuestro *canvas*: una instancia de la clase `Display`, que es la que nos permite mostrar toda la información por pantalla, y una instancia de todas las clases que vamos a mostrar por pantalla: `screen` (es el *canvas*, ya explicado), `menu` (será nuestro menú y lo explicaremos en el apartado 14.2.), `dificultad` (es una de las opciones de menú, ver apartado 14.3.), `resultado` (otra de las opciones del menú, ver apartado 14.4.), `nuevoRecord` nos permite introducir un nuevo récord, `musica` (otra de las opciones del menú, ver apartado 14.5.) y `alerta` que nos mostrará por pantalla un mensaje de error siempre y cuando la paleta de colores sea inferior a 256 colores (ver apartado 14.6.). Por último los dos `String` son los valores por defecto de las opciones del menú música y dificultad respectivamente.

El constructor de esta clase es el siguiente:

```

public Comeme() {
    display=Display.getDisplay(this);
    exitCommand=new Command("Salir",Command.SCREEN,2);
    playCommand=new Command("Jugar",Command.CANCEL,2);
    endCommand=new Command("Fin",Command.SCREEN,2);
    screen=new SSCanvas(this);
    dificultad=new Dificultad(this);
    menu=new Menu(this);
    resultado=new Resultado(this);
    nuevoRecord=new NuevoRecord(this);
    alerta=new Alerta(this);
    musica=new Musica(this);
    screen.addCommand(playCommand);
    screen.addCommand(exitCommand);
    screen.addCommand(endCommand);
    screen.setCommandListener(this);
}

```

Primero asignamos como pantalla la clase `Comeme`, creamos los tres comandos explicados anteriormente y una instancia de cada una de las clases que mostraremos cuando elijamos su correspondiente opción en el menú, y por último añadimos los comandos creados y asignamos a la clase `Comeme` para que atienda los eventos de comandos.

Los métodos heredados de `MIDlet` a implementar son los siguientes:

```

public void startApp() throws MIDletStateChangeException{
    if (display.numColors() >= 256) {
        display.setCurrent(menu);
    } else {
        display.setCurrent(alerta);
    }
}

public void pauseApp() {
}

public void destroyApp(boolean unconditional) {
    screen=null; dificultad=null; menu=null; resultado=null;
    nuevoRecord=null; alerta=null; musica=null;

    if (sonido.compareTo("SI")==0) {
        EfectosDeSonido.getInstancia().cerrarRecurso();
    }
}

```

Nada más ser lanzado el *midlet* se ejecuta el método `startApp()`, siempre y cuando el dispositivo contenga una paleta de colores de al menos 256 colores, el cual

muestra por pantalla el menú. Con el método `pauseApp()` no haremos nada, y para destruir el *midlet* asignaremos a todas las instancias, creadas en el constructor, el valor `null`, y por último liberaremos los recursos ocupados por nuestra música, siempre y cuando haya sido elegida.

A continuación vamos a ver el método que trata los comandos añadidos en el constructor:

```
public void commandAction(Command c, Displayable s) {
    if (c==exitCommand) {
        if (screen.isPlaying()) {
            screen.quitGame();
        } else {
            destroyApp(false);
            notifyDestroyed();
        }
    } else if (c==endCommand) {
        destroyApp(false);
        notifyDestroyed();
    } else if (c==playCommand && !screen.isPlaying()) {
        new Thread(screen).start();
    }
}
```

Si pulsamos el comando `exitCommand` y estamos jugando, el juego se reiniciará a la pantalla de menú, y si estamos en esta pantalla y pulsamos este mismo comando, se cerrará la aplicación completa. Si en lugar de pulsar este comando pulsamos `endCommand`, la aplicación será cerrada inmediatamente. Y por último si pulsamos el comando `playCommand` y no estamos jugando, lanzaremos nuestro *canvas* en un *thread* aparte del *thread* principal.

También contamos con un método que cerrará la aplicación cuando sea invocada:

```
public void salir() {
    destroyApp(false);
    notifyDestroyed();
}
```

Veamos los métodos dedicados a la comprobación de récord (utilizando estos a su vez los métodos de la clases `Resultado` y `NuevoRecord` que veremos en el apartado 14.4.).

```
public boolean esRecord(int punt) {
    boolean record = resultado.esRecord(punt);
    return record;
}
```

```
public void setRecord(int punt){
    nuevoRecord.setPuntuacion(punt);
    display.setCurrent(nuevoRecord);
}

public void insertarRecord(String nombre, int puntuacion){
    resultado.insertarRegistro(nombre,puntuacion);
}
```

El primer método, `esRecord(...)`, comprueba si la puntuación pasada como parámetro es un nuevo récord. El segundo, `setRecord(...)`, asigna como nuevo récord la puntuación pasada como argumento y nos permite introducir tres letras para reconocer a éste. Y por último, el método `insertarRecord(...)`, introduce el nuevo récord y el *string* que reconoce a éste en nuestra pequeña base de datos.

Y por último, contamos con los métodos típicos para asignar y consultar los distintos atributos de la clase `Comeme`:

```
public void setScreen(){
    display.setCurrent(screen);
}

public void setMenu(){
    display.setCurrent(menu);
}

public void setDificultad(){
    display.setCurrent(dificultad);
}

public void setMusica(){
    display.setCurrent(musica);
}

public void setSonido(String s){
    sonido=s;
}

public String getSonido(){
    return sonido;
}
```



```
public void setNivel(String n){
    nivel=n;
}

public String getNivel(){
    return nivel;
}

public void setResultado(){
    display.setCurrent(resultado);
}

public void cargarResultado(){
    resultado.verResultado();
}
```

14.2. La clase Menu

Una vez lancemos nuestra aplicación, lo primero que veremos por pantalla será un menú con cuatro opciones. Este menú es representado por una lista (ver capítulo 3, apartado 3.2.2.) de la siguiente forma:

```
public class Menu extends List implements CommandListener{
    private Comeme midlet;
    private Command salir;
```

Tenemos como atributos nuestro *midlet* y un comando que nos servirá para salir de la aplicación.

```
public Menu(Comeme m){
    super("Menu",List.IMPLICIT);
    this.append("Jugar",null);
    this.append("Dificultad",null);
    this.append("Resultados",null);
    this.append("Música",null);
    this.midlet = m;
    salir = new Command("Salir",Command.EXIT,1);
    this.addCommand(salir);
    this.setCommandListener(this);
}
```

En el constructor creamos la lista, llamando a la clase padre `List`, y añadimos las cuatro opciones de las que se va a componer nuestro menú. Además de las cuatro opciones del menú, añadiremos el comando *salir* y asignaremos a esta clase como manejadora de los eventos de comandos.

```
public void commandAction(Command c, Displayable d){
    if (c == List.SELECT_COMMAND){
        switch(this.getSelectedIndex()){
            case 0: midlet.setScreen();
                    break;
            case 1: midlet.setDificultad();
                    break;
            case 2: midlet.cargarResultado();
                    midlet.setResultado();
                    break;
            case 3: midlet.setMusica();
                    break;
        }
    }else{
        midlet.salir();
    }
}
```

Según sea la opción elegida por el usuario, mostraremos una pantalla diferente. Por ejemplo si elegimos *jugar* comenzará nuestro juego, si elegimos *dificultad* mostraremos otra lista con una serie de opciones, y así para todas las opciones disponibles. Por último si pulsamos el comando *salir*, se cerrará la aplicación.

14.3. La clase Dificultad

La clase `Dificultad` es prácticamente igual que la anterior, la única diferencia son las distintas opciones a mostrar por pantalla y dos comandos en lugar de uno.

```
public class Dificultad extends List implements
CommandListener{
    private Comeme midlet;
    private Command atras, aceptar;
```

Ahora mostraremos dos comandos por pantalla, uno para volver al menú anterior y otro para confirmar los cambios hechos en nuestras opciones de dificultad.

```
public Dificultad(Comeme m){
    super("Dificultad",List.EXCLUSIVE);
    this.append("Normal",null);
    this.append("Difícil",null);
    this.setSelectedIndex(0,true);
    midlet = m;
    atras = new Command("Atras",Command.BACK,1);
    aceptar = new Command("Aceptar",Command.OK,1);
```

```
this.addCommand(aceptar);  
this.addCommand(atras);  
this.setCommandListener(this);  
}
```

Este constructor es prácticamente el mismo que para la clase `Menu`, la única diferencia es que esta es una lista explícita en lugar de ser implícita (ver capítulo 3, apartado 3.2.2.) como lo era la lista de `Menu`. Por tratarse de una lista explícita colocaremos una de nuestras opciones como predeterminada, en nuestro caso el nivel de dificultad predefinido será el nivel normal.

Para poder elegir el nivel de dificultad difícil, tendremos que entrar en la opción *dificultad* de nuestro menú principal, elegir nivel de dificultad difícil y aceptar los cambios.

```
public void commandAction(Command c, Displayable d){  
    if (c==aceptar){  
        midlet.setNivel(getString(this.getSelectedIndex()));  
        System.out.println("Nivel  
        "+getString(this.getSelectedIndex()));  
        midlet.setMenu();  
    }else if (c==atras){  
        midlet.setMenu();  
    }  
}  
}
```

Como ya hemos dicho antes, el comando `aceptar` nos sirve para confirmar la elección de dificultad, mientras que con el comando `atras` volvemos al menú principal. Además cada vez que confirmemos un nivel de dificultad, tanto el normal como el difícil, mostraremos por la pantalla de comandos el nivel elegido y pasaremos a nuestro menú principal.

14.4. La clase `Resultado` y la clase `NuevoRecord`

Con la clase `Resultado` mostraremos por pantalla las tres mejores puntuaciones (tres para cada nivel de dificultad), tanto cuando elegimos la opción *Resultados* de nuestro menú principal, como cuando conseguimos un nuevo récord. Una vez hayamos terminado una partida, si hemos conseguido un nuevo récord, se nos mostrará por pantalla un pequeño recuadro donde podemos escribir el texto que queremos asociar a esta nueva puntuación, esto lo haremos con nuestra clase `NuevoRecord`.

14.4.1. La clase `NuevoRecord`

Esta clase heredará de la clase `Form` (ver capítulo 3, apartado 3.2.4.) y contendrá

los siguientes atributos:

```
public class NuevoRecord extends Form implements
CommandListener{
    private TextField txtNombre;
    private Command aceptar;
    private Comeme midlet;
    private int puntuacion;
```

Nos encontramos con un campo para escribir el texto que asociaremos al nuevo récord, `txtNombre`, un comando para confirmar lo escrito en el campo anterior, `aceptar`, la puntuación a asociar al campo anterior, y por último una variable del tipo `Comeme`, la clase que desencadena toda nuestra aplicación.

Veamos el constructor:

```
public NuevoRecord(Comeme m){
    super("Nuevo Record");
    midlet = m;
    puntuacion = 0;
    txtNombre = new TextField("Nombre", "", 3, TextField.ANY);
    aceptar = new Command("Aceptar", Command.OK, 1);
    this.append(txtNombre);
    this.addCommand(aceptar);
    this.setCommandListener(this);
}
```

Creamos nuestro contenedor e introducimos tanto el campo para introducir texto como el comando `aceptar`. Además la puntuación que asociaremos, en un principio, al texto escrito anteriormente será 0. También asignaremos a esta clase como manejadora de sus propios comandos.

Esta clase contiene dos métodos, uno para asignar la puntuación del récord, ya que en el constructor la asignábamos como 0, y la otra que controla los eventos de comandos (en este caso del comando `aceptar`). En este último método guardaremos tanto puntuación como nombre asociado a ésta en nuestro *rms*, y lo mostraremos por pantalla.

```
public void setPuntuacion(int punt){
    puntuacion = punt;
}
public void commandAction(Command c, Displayable d){
    midlet.insertarRecord(txtNombre.getString(), puntuacion);
    midlet.cargarResultado();
    midlet.setResultado();
}
}
```

14.4.2. La clase Resultado

En el apartado anterior hemos visto como tras escribir el nombre de un nuevo récord, mostramos por pantalla las tres puntuaciones más altas hasta ahora conseguidas. Esto lo hacíamos mediante métodos de nuestro *midlet* que, a su vez, se ayudan de otros métodos de la clase *Resultado* y que ahora explicaremos con más detenimiento.

Esta clase hereda de *List*, veamos sus atributos:

```
public class Resultado extends List implements
CommandListener{
    private Comeme midlet;
    private Command aceptar;
    private RecordStore rs;
```

Los dos primeros son los mismos que para la clase *NuevoRecord*, mientras que el tercero de los atributos, *rs*, es nuestra pequeña base de datos donde almacenaremos nuestros récord (ver capítulo 5).

Su constructor:

```
public Resultado(Comeme m){
    super("Records",List.IMPLICIT);
    initPuntuacion("ClasificacionNormal");
    initPuntuacion("ClasificacionDifícil");
    midlet = m;
    aceptar = new Command("Aceptar",Command.OK,1);
    this.addCommand(aceptar);
    this.setCommandListener(this);
}
```

Primero creamos nuestra lista, no introducimos ninguna opción de ésta en el constructor, luego inicializamos dos base de datos para cada uno de los modos de dificultad, con el método *initPuntuacion(...)* que veremos a continuación, y por último añadimos el comando *aceptar* y asignamos a esta clase como manejador de los eventos de comandos. Veamos los dos métodos que utilizamos en el constructor:

```
public void commandAction(Command c, Displayable d){
    midlet.setMenu();
}
```

Tanto si elegimos una de las opciones de nuestra lista como si pulsamos el comando *aceptar*, volveremos a nuestro menú principal.

```
private void initPuntuacion(String nombre){
    try{
        rs = RecordStore.openRecordStore(nombre,true);
        if (rs.getNumRecords() == 0){
            for (int i=0;i<3;i++){
                byte[] registro;
                ByteArrayOutputStream baos = new
                ByteArrayOutputStream();
                DataOutputStream dos = new
                DataOutputStream(baos);
                dos.writeUTF("AAA");
                dos.writeInt(0);
                dos.flush();
                registro = baos.toByteArray();
                rs.addRecord(registro,0,registro.length);
                baos.close();
                dos.close();
                registro = null;
            }
        }
        rs.closeRecordStore();
    }catch (Exception e){
        System.out.println(e);
    }
}
```

Para inicializar nuestra pequeña base de datos, comprobamos que ésta está vacía, y si es así introducimos tres puntuaciones de 0 puntos cada una y el nombre AAA. Esto lo hacemos mediante *streams*, ya que los registros de nuestra base de datos son *arrays* de *bytes* (ver capítulo 5).

Otros métodos de esta clase son:

```
public boolean esRecord(int punt){
    try{
        rs = RecordStore.openRecordStore(
            "Clasificacion"+midlet.getNivel(),true);
        byte[] reg = rs.getRecord(3);
        int p3 = getPuntuacion(reg);
        rs.closeRecordStore();
        reg = null;
        return ((p3 >= punt)?false:true);
    }
```

```
    }catch (Exception e){
        System.out.println(e);
        return false;
    }
}
```

Con este método comprobamos si una puntuación, pasada como parámetro, es récord o no. Para eso nos valemos del método `getPuntuacion(...)`, que nos dará la puntuación de un registro determinado. En este caso consultaremos la tercera mejor puntuación, y comprobaremos si la puntuación pasada como parámetro es mayor que ésta. En caso afirmativo la nueva puntuación será un récord y en caso contrario no lo será.

Una vez que hemos comprobado que una cierta puntuación es récord, tenemos que introducirla en nuestra pequeña base de datos:

```
public void insertarRegistro(String nombre, int punt){
    try{
        ByteArrayOutputStream baos = new
        ByteArrayOutputStream();
        DataOutputStream dos = new DataOutputStream(baos);
        dos.writeUTF(nombre);
        dos.writeInt(punt);
        dos.flush();
        byte[] nreg = baos.toByteArray();
        baos.close();
        dos.close();
        rs = RecordStore.openRecordStore
        ("Clasificacion"+midlet.getNivel(),true);
        byte[] reg3 = rs.getRecord(3);

        if (getPuntuacion(reg3) < punt){
            byte[] reg2 = rs.getRecord(2);
            byte[] reg1 = rs.getRecord(1);
            if (getPuntuacion(reg2) < punt){
                if (getPuntuacion(reg1) < punt){
                    rs.setRecord(1,nreg,0,nreg.length);
                    rs.setRecord(2,reg1,0,reg1.length);
                    rs.setRecord(3,reg2,0,reg2.length);
                }else{
                    rs.setRecord(2,nreg,0,nreg.length);
                    rs.setRecord(3,reg2,0,reg2.length);
                }
            }
        }
    }
}
```

```
                }else{
                    rs.setRecord(3,nreg,0,nreg.length);
                }
            }
            rs.closeRecordStore();
        }catch (Exception e){
            System.out.println(e);
        }
    }
}
```

Para pasar nuestro nombre y puntuación, previo paso por `esRecord(...)`, a ser un *array* de *bytes*, utilizamos *streams*. Una vez hecho este paso, sólo nos queda saber en qué posición de nuestra base de datos introduciremos el nuevo récord. Primero comprobamos si la nueva puntuación es mayor que nuestro tercer récord, que debería de serlo si hemos pasado antes por el método `esRecord(...)`, si es así, haremos lo propio con nuestro récord número 2 y 1 si es necesario. Tras estas comprobaciones reordenaremos las puntuaciones.

Para estos dos últimos métodos nos hemos valido del método `getPuntuacion(...)`, veámoslo con más detalle:

```
private int getPuntuacion(byte[] registro){
    try{
        ByteArrayInputStream bais = new
        ByteArrayInputStream(registro);
        DataInputStream dis = new
        DataInputStream(bais);
        String nombre = dis.readUTF();
        int puntuacion = dis.readInt();
        bais.close();
        dis.close();
        return puntuacion;
    }catch (Exception e){
        System.out.println(e);
        return -1;
    }
}
```

Lo único que hacemos es sacar la puntuación de uno de los registros mediante la utilización de *streams*.

Ya hemos visto como creamos e introducimos las puntuaciones, ahora veremos como la mostramos por pantalla:


```
public void verResultado(){
    this.setTitle("Records Nivel "+midlet.getNivel());
    try{
        for (int i=this.size();i>0;i--){
            this.delete(i-1);
        }
        this.append("Puntuacion Jugador",null);
        rs = RecordStore.openRecordStore
            ("Clasificacion"+midlet.getNivel(),true);
        for (int i=1;i<=rs.getNumRecords();i++){
            byte[] registro = rs.getRecord(i);
            ByteArrayInputStream bais = new
                ByteArrayInputStream(registro);
            DataInputStream dis =new DataInputStream(bais);
            String nombre = dis.readUTF();
            int puntuacion = dis.readInt();
            this.append(" "+puntuacion+" "+nombre,null);
            bais.close();
            dis.close();
            registro = null;
        }
        rs.closeRecordStore();
    }catch (Exception e){
        System.out.println(e);
    }
}
```

Recorremos nuestra base de datos, extraemos el texto y la puntuación asociada a éste y lo mostramos por pantalla como opciones de la lista que creamos en el constructor.

14.5. La clase Musica

La clase *Musica* es prácticamente igual que la clase *Dificultad*, siendo también una de las opciones de nuestro menú principal, la cual nos permite la configuración de la música. La única diferencia son las opciones a elegir en la lista ,en este caso elegimos música *SI* o música *NO* en lugar del nivel de dificultad.

```
public class Musica extends List implements CommandListener{
    private Comeme midlet;
    private Command atras, aceptar;
```

```
public Musica(Comeme m){
    super("Música",List.EXCLUSIVE);
    this.append("SI",null);
    this.append("NO",null);
    this.setSelectedIndex(0,true);
    midlet = m;
    atras = new Command("Atras",Command.BACK,1);
    aceptar = new Command("Aceptar",Command.OK,1);
    this.addCommand(aceptar);
    this.addCommand(atras);
    this.setCommandListener(this);
}

public void commandAction(Command c, Displayable d){
    if (c==aceptar){
        midlet.setSonido(getString(
            this.getSelectedIndex()));
        System.out.println("Música
            "+getString(this.getSelectedIndex()));
        midlet.setMenu();
    }else if (c==atras){
        midlet.setMenu();
    }
}
}
```

14.6. La clase Alerta

Esta clase hereda de la clase `Alert`, y la utilizamos para mostrar un mensaje de error por pantalla si intentamos ejecutar nuestra aplicación en un dispositivo con una paleta de colores inferior a 256. En esta clase lo único que hacemos es crear un *alert* y añadirle una imagen y una leyenda explicativa (ver capítulo 3, apartado 3.2.1). Veamos dicha clase:

```
public class Alerta extends Alert implements
    CommandListener{
    private Comeme midlet;
    private Command salir;
    private Image imagen;
```

```
public Alerta(Comeme m) {
    super("Alerta");
    try{
        imagen=Image.createImage("/bicho10a.png");
    }catch(IOException ioe){
        System.err.println("No se puede cargar la
            imagen bicho10a.png" + ":" + ioe.toString());
    }

    setTimeout(FOREVER);
    setString(" N° de colores inferior a 256");
    setImage(imagen);
    setType(AlertType.ERROR);
    this.midlet = m;
    salir = new Command("Salir",Command.EXIT,1);
    this.addCommand(salir);
    this.setCommandListener(this);
}

public void commandAction(Command c, Displayable d){
    midlet.salir();
}
}
```


CAPÍTULO 15: La música

En este capítulo vamos a ver la clase que da forma a nuestra música y como la utilizamos en nuestra clase principal `SSCanvas`.

15.1. La clase `EfectosDeSonido`

Como atributos de esta clase tenemos un atributo estático de la clase `EfectosDeSonido` y un *player* que será el que dé las órdenes a nuestra composición (tocar la música, parar la música, seguir la música,...).

```
private static EfectosDeSonido instancia;  
private Player p;
```

El constructor de esta clase lo dejaremos vacío:

```
private EfectosDeSonido() {  
}
```

Veamos ahora todos los métodos de la clase `EfectosDeSonido`:

El primero de ellos nos crea una instancia de la clase `EfectosDeSonido` si ésta no ha sido creada con anterioridad.

```
static EfectosDeSonido getInstancia() {  
    if (instancia == null) {  
        instancia = new EfectosDeSonido();  
    }  
    return instancia;  
}
```

Los tres métodos siguientes nos permiten cerrar y dejar libre los recursos utilizados por el *player*, pararlo y reanudarlo tras una parada.

```
public void cerrarRecurso() {  
    p.close();  
}  
public void pararSecuencia() {  
    try {  
        p.stop();  
    } catch (MediaException me) {  
    }  
}
```

```
public void seguirSecuencia() {
    try{
        p.start();
    }catch (MediaException me){
    }
}
```

Y el último método es que toca la melodía creada (ver capítulo 6, apartado 6.2.).

```
public void tocarSecuencia() {
    byte tempo=30;
    byte d=8;
    byte C4=ToneControl.C4;;
    byte D4=(byte) (C4+2);
    byte E4=(byte) (C4+4);
    byte F4=(byte) (C4+5);
    byte G4=(byte) (C4+7);
    byte silencio=ToneControl.SILENCE;

    byte[] secuencia={
        ToneControl.VERSION,1,
        ToneControl.TEMPO, tempo,

        //comienzo del bloque 0
        ToneControl.BLOCK_START,0,
        //notas del bloque 0
        C4,d, F4,d, C4,d,F4,d, F4,d, C4,d, F4,d,
        //fin del bloque 0
        ToneControl.BLOCK_END,0,

        //inicio del bloque 1
        ToneControl.BLOCK_START, 1,
        //notas del bloqe 1
        C4,d, E4,d,E4,d,C4,d,E4,d,E4,d,C4,d, E4,d,
        //fin del bloque 1
        ToneControl.BLOCK_END,1,

        //reproducir bloque 0
        ToneControl.PLAY_BLOCK, 0,
        //reproducir bloque 1
        ToneControl.PLAY_BLOCK, 1,
        //reproducir bloque 0
        ToneControl.PLAY_BLOCK, 0,
    };
};
```

```
try{
    p = Manager.createPlayer
        (Manager.TONE_DEVICE_LOCATOR);
    p.realize();
    ToneControl
    c=(ToneControl)p.getControl("ToneControl");
    c.setSequence(sequencia);
    p.setLoopCount(-1);
    p.start();
}catch (IOException ioe){
}catch (MediaException me){
}
}
```

15.2. La clase EfectosDeSonido en el Canvas

Lo único que introduciremos en el *canvas* de nuestra clase EfectosDeSonido, será darle comienzo a la música sí y sólo si el usuario ha elegido la opción de música activa, en otro caso no haremos nada.

```
if (midlet.getSonido().compareTo("SI")==0){
    EfectosDeSonido.getInstancia().tocarSecuencia();
    musica=true;
}else{
    musica=false;
}
```

También detendremos la música cuando nuestro personaje colisione tanto con un objeto como con un enemigo. Tras la pausa que generamos al colisionar, continuaremos con la música.

```
if (musica){
    EfectosDeSonido.getInstancia().pararSecuencia();
}
//tratamiento de la colisión
if (musica){
    EfectosDeSonido.getInstancia().seguirSecuencia();
}
```

Y por último tras terminar una partida pararemos la música para liberar los recursos más tarde en la clase Comeme.

CAPÍTULO 16: Conclusiones

Este capítulo lo vamos a dedicar a explicar qué hemos hecho en general en nuestro proyecto, problemas encontrados, ... Y además comentaremos las herramientas utilizadas para la realización de nuestro proyecto y algunas de las cosas que nos hubiera gustado introducir y que, por falta de tiempo o por desconocimiento en un principio, no hemos podido incluir.

16.1. Conclusiones finales

Nuestro objetivo en este proyecto, era crear una aplicación J2ME. Y una de las aplicaciones más demandadas por los usuarios de móviles, es hoy en día, la de los videojuegos. Decidimos copiar un juego de Spectrum dada las similitudes en restricciones de estos microordenadores y los dispositivos móviles más abundantes hoy en día.

La elección de este tipo de género, *plataformas*, es por la simple razón de que es uno de los géneros más famosos de los videojuegos, y seguramente el más demandado en los años 80s (Phantomas) y 90s (Super Mario Bros y Sonic the Hedgehog).

Los principales problemas que hemos encontrado a la hora de afrontar nuestro proyecto han sido los siguientes:

1. El primer problema que nos encontramos al abordar nuestro proyecto fue la imposibilidad de introducir un fichero con los datos de todas las pantallas y bichos, y posteriormente leerlos desde nuestra aplicación. Así que creamos una programa auxiliar (con el lenguaje de programación C) que pasara estos datos a estructuras del lenguaje Java e introducir estas directamente en nuestra aplicación (ver apéndice A).

2. El siguiente problema que nos encontramos fue la generación de los gráficos de nuestro juego, ya que al *escanearlos* de la revista MicroHobby [3-12] estos se mostraban muy borrosos y distorsionados. Por lo que tuvimos que crearlos a mano con varios programas de dibujos y un editor de iconos.

3. Uno de los mayores problemas encontrados fue la del tamaño de las pantallas de nuestro juego. Este tamaño es superior a la mayoría de las pantallas de los dispositivos actuales, con lo que no podíamos introducirla entera. Como tamaño de pantalla elegimos la del emulador que viene con la versión del wireless toolkit utilizada (180*177 píxeles). Aún así nuestras pantallas no entraban enteras, así que tuvimos que crear un *scroll* en el eje X para la correcta visualización de todo el escenario. Este *scroll* [2] complicó bastante la realización de nuestro proyecto, pero es la única forma de que nuestra aplicación se pueda ejecutar en la mayor variedad de dispositivos.

4. Por último comentaremos un pequeño problema encontrado, el cual es debido a la versión del compilador utilizado. A la hora de organizar los objetos, de forma aleatoria, por todo el mapa de nuestro juego utilizamos el método `nextInt()` de la clase `Random`, el

cual, en esta versión del compilador, no nos permite pasarle un parámetro para acotar, a nuestro antojo, el valor aleatorio.

Como conclusión personal decir que uno de los principales motivos que me inclinaron a realizar esta carrera fue la de crear algún día un videojuego, aunque sea pequeño como éste, y aunque mi futuro profesional seguramente estará muy lejos de la creación de videojuegos, quiero agradecer esta pequeña oportunidad que me han brindado en este proyecto de fin de carrera.

16.2. Herramientas utilizadas

La herramienta principal que hemos utilizado ha sido J2ME Wireless Toolkit V.2.0_01 para poder compilar y ejecutar nuestra aplicación en un PC o empaquetar ésta y llevarla a un dispositivo móvil.

Para la compilación y ejecución de nuestros programas auxiliares, `auxComeme.c` y `auxComeme2.c`, hemos utilizado el entorno BloodShed Dev-C++ V.4.

Para la creación de todos nuestros gráficos hemos utilizado el editor de iconos IconCoolEditor V.4.4 y además otros auxiliares como Microsoft Paint e InfaView V.3.95.

Y por último nos hemos valido del emulador ZX Spectrum V1.03.97.1213, que es gratuito, para intentar que nuestra aplicación se pareciese lo más posible al juego original.

16.3. Líneas futuras

Debido a que nuestro proyecto debe de ser terminado en un plazo de tiempo prefijado, si no estaríamos incluyendo mejoras prácticamente cada día, hay cosas que no hemos podido incluir en esta versión pero que pueden servir como mejoras para el futuro. He aquí algunas líneas futuras:

1. Una de las cosas del juego original que no hemos introducido son los distintos colores utilizados para un mismo bloque de decorado, (utilizamos siempre el mismo para todas las pantallas). Esto sería muy interesante hacerlo, ya que en un principio habría que cambiar la paleta de colores de nuestros PNG.

2. Aunque hemos incluido música en nuestro juego, ésta deja mucho que desear. Sería muy apropiado incluir varias músicas dependiendo de la pantalla en la que nos encontremos y distintos sonidos para los bichos y nuestro personaje.

3. Para la realización de esta aplicación hemos utilizado una clase `Sprite` propia que podía ser modificada a nuestro gusto, pero J2ME nos provee de una clase `Sprite` y otra `LayerManager` ya definidas en la clase `GameCanvas`, la cual hereda de `Canvas` (ésta última es la que hemos utilizado nosotros). Esto nos permitiría crear *sprites* auténticos y no simples imágenes pegadas en la pantalla, además de rotarlos a nuestro antojo, etc. También con la clase `LayerManager` podemos crear los fondos de una forma simple y

hacer que nuestros *sprites* interaccionen de una forma más sencilla con los escenarios de nuestro juego. Además esta clase nos provee de otros métodos para recoger el estado de las teclas en el momento que nosotros queramos, al contrario de los que hemos utilizado nosotros, que cada vez que hay un evento de teclado tenemos que tratarlo en ese mismo instante. Esto hace que con los métodos de la clase `Canvas` no podamos tratar los eventos de varias teclas a la vez y a veces perdamos cierta información [14].

APÉNDICES

APÉNDICE A: Listados de pantallas y bichos

Para la creación de escenarios y bichos dijimos, en sus respectivos capítulos, que utilizábamos unos *arrays* de enteros de donde obteníamos todos los datos necesarios sobre estos, ¿pero de dónde sacamos estos datos?. Estos datos los sacamos de unos listados hexadecimales [3-12]. En un principio estos listados los íbamos a incluir como unos ficheros de datos en el propio programa y posteriormente leerlos desde éste, pero debido a la imposibilidad de hacer esto en J2ME decidimos hacer un programa auxiliar que nos pasara estos listados hexadecimales a estructuras del lenguaje (en este caso *arrays* de enteros).

Además de éste programa auxiliar, utilizamos otro para verificar que los datos de los listados, previamente *escaneados*, eran los correctos, pues tras este *escaneo* resultaban una gran cantidad de errores en los valores hexadecimales. Para la verificación de los valores de los listados sumamos los dígitos hexadecimales de una línea completa y comprobamos que esta suma coincide con la original (en la revista MicroHobby [3-12]).

Ambos programas, `auxComeme.c` y `auxComeme2.c`, están hechos en el lenguaje de programación C como sus extensiones indican. No veremos aquí la explicación de estos códigos debido a la simplicidad de estos, los típicos de lectura y escritura en ficheros, pero lo que sí veremos será un ejemplo de entrada y salida para el primero de ellos, `auxComeme.c`, con la primera pantalla de nuestro juego.

Listado para la primera pantalla (entrada para <code>auxComeme.c</code>)
00 00 85 83 02 01 A5 81 00 01
4C 81 00 03 4F 81 00 05 4D 81
00 07 4E 81 00 09 4C 81 00 0B
4D 81 00 0D 4E 81 00 0F 4F 81
00 11 4C 81 00 13 4D 81 00 15
4E 81 00 17 4C 81 00 19 4F 81
00 1B 4C 82 02 0F CA 12 00 1F
65 82 08 18 28 87 03 00 63 81
06 00 60 81 08 00 63 82 09 05
88 82 0E 00 61 83 12 02 82 82
0D 02 8B 86 0E 02 91 6C 12 08
80 81 12 0A 82 81 12 0D 80 81
12 11 D0 81 12 13 66 85 13 13
87 01 13 15 87 01 13 17 87 01
13 19 87 01 13 1B 87 01 12 1D
82 81 07 1A A2 02 02 1E A0 81
04 1E A3 81 08 07 64 85 06 0C
C3 81 FF 88 10 50 B1 2C 38 14
42 E7 0C 68 58 70 C6 24 FF

Estructura para la primera pantalla (salida para auxComeme.c)

```

int pantalla1[]={0, 0, 133, 131
                , 2, 1, 165, 129
                , 0, 1, 76, 129
                , 0, 3, 79, 129
                , 0, 5, 77, 129
                , 0, 7, 78, 129
                , 0, 9, 76, 129
                , 0, 11, 77, 129
                , 0, 13, 78, 129
                , 0, 15, 79, 129
                , 0, 17, 76, 129
                , 0, 19, 77, 129
                , 0, 21, 78, 129
                , 0, 23, 76, 129
                , 0, 25, 79, 129
                , 0, 27, 76, 130
                , 2, 15, 202, 18
                , 0, 31, 101, 130
                , 8, 24, 40, 135
                , 3, 0, 99, 129
                , 6, 0, 96, 129
                , 8, 0, 99, 130
                , 9, 5, 136, 130
                , 14, 0, 97, 131
                , 18, 2, 130, 130
                , 13, 2, 139, 134
                , 14, 2, 145, 108
                , 18, 8, 128, 129
                , 18, 10, 130, 129
                , 18, 13, 128, 129
                , 18, 17, 208, 129
                , 18, 19, 102, 133
                , 19, 19, 135, 1
                , 19, 21, 135, 1
                , 19, 23, 135, 1
                , 19, 25, 135, 1
                , 19, 27, 135, 1
                , 18, 29, 130, 129
                , 7, 26, 162, 2
                , 2, 30, 160, 129
                , 4, 30, 163, 129
                , 8, 7, 100, 133
                , 6, 12, 195, 129
                };
int bicho1[]={136, 16, 80, 177, 44, 56, 20, 66, 231, 12,
              104, 88, 112, 198, 36};

```


APÉNDICE B: Manual de usuario

En este apéndice vamos a ver un pequeño manual de nuestra aplicación, aún siendo ésta muy sencilla e intuitiva. Primero vamos a ver como movernos por nuestro menú, luego veremos como hacer todos los movimientos de nuestro personaje JaimeNu [3-12], y por último veremos el papel que desempeñan los diversos objetos de nuestro juego.

B.1. Paseo por el menú

Aunque en el capítulo 14 ya dábamos una explicación de este menú, ahora vamos a ver un *paso a paso* ilustrado:

1. Buscamos el nombre de nuestra aplicación, si ésta la tenemos en un dispositivo la encontraremos junto con todas nuestras aplicaciones (Figura B.1.), pulsamos el comando *Launch*.



Figura B.1. La aplicación Comeme.

2. Lo primero que mostramos en nuestra aplicación es el menú principal con las siguientes opciones (Figura B.2.): *Jugar*, *Dificultad*, *Resultados* y *Música*. Una vez aquí, podemos elegir cualquier opción de éste o pulsar el comando *Salir* para terminar la aplicación y volver al apartado anterior.

3. Si la opción elegida de nuestro menú es la de *Dificultad*, mostraremos los dos niveles posibles de nuestro juego, *Normal* o *Difícil* (Figura B.3.). Podemos elegir uno de los dos y pulsar el comando *Aceptar* para establecer dicho nivel en la próxima partida a jugar, o pulsar el comando *Atras* y volver al menú anterior (Figura B.2.).



Figura B.2. El menú principal.

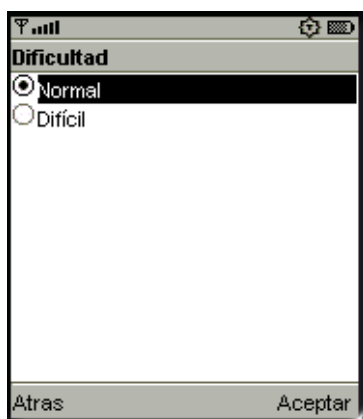


Figura B.3. Modos de dificultad.

4. Si la opción elegida es *Resultados*, mostraremos los *records* actuales (Figura B.4.). Si pulsamos el comando Aceptar o alguno de nuestros *records*, volveremos al menú anterior (Figura B.3.).



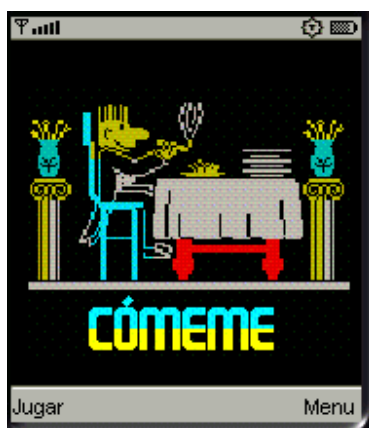
Figura B.4. Pantalla de records.

5. Sin embargo si la opción elegida es *Música*, mostraremos, al igual que para la opción *Dificultad*, otro menú con las opciones *SI* y *No* (Figura B.5.). Los comandos son iguales que los del apartado 3.

6. Por último si elegimos la opción *Jugar*, entramos en la pantalla de título de nuestro juego. Aquí nos encontramos con dos comandos (Figura B.6.(a)): *Jugar*, con el que comenzaremos el juego y *Menu* el cual desplegará dos opciones (Figura B.6.(b)): *Salir* vuelve un paso hacia atrás en nuestra aplicación, y *Fin*, que terminará ésta.



Figura B.5. Elección de música.



(a)



(b)

Figura B.6. Pantalla de título.



Figura B.7. Pantalla de nuevo récord.

7. Si hemos acabado una partida y nuestra puntuación es una de las tres más altas conseguidas hasta ahora, se nos pedirá que introduzcamos un texto, de tres caracteres como máximo, que estará asociado a nuestra puntuación (Figura B.7.).

B.2. Los movimientos de JaimeNu

Para acabar nuestro manual de usuario mostraremos como el usuario puede realizar todos los movimientos de nuestro héroe JaimeNu.

1. Andar hacia la derecha. Este movimiento lo realizaremos pulsando la tecla direccional *derecha*. Este movimiento sólo lo podremos hacer si nos encontramos en una plataforma o encaramados en unas escaleras de huesos (Figura B.8.), siempre y cuando haya una plataforma a nuestra derecha.



Figura B.8. Ejemplo de movimiento a derechas.

2. Andar hacia la izquierda. Igual que para derechas.

3. Podemos saltar (Figura B.9.(a)) tanto a derechas como a izquierdas como hacia

arriba (sin dirección). Una vez saltado podremos encaramarnos a una escaleras de huesos, siempre y cuando nos encontremos una en la trayectoria de nuestro salto (Figura B.9.(b)). Esta acción la realizaremos pulsando la tecla direccional *arriba*.

4. Una vez encaramados a unas escaleras de hueso, podremos subir y bajar por ellas con las correspondientes teclas de dirección *arriba* y *abajo*.

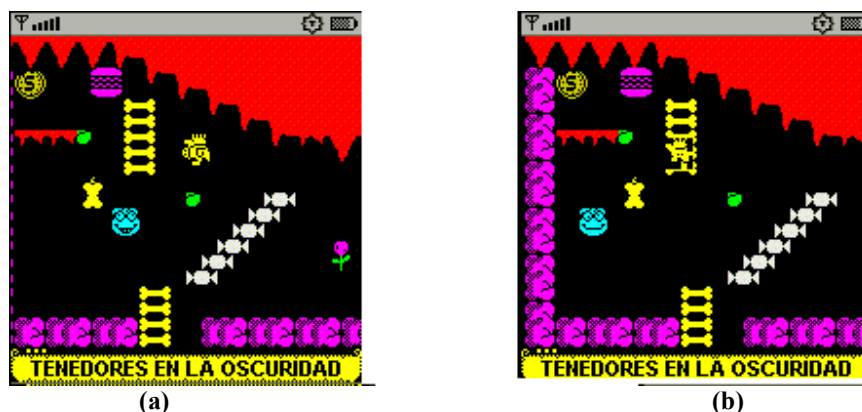


Figura B.9. Ejemplo de salto continuado de agarre a escaleras.

B.3. Los objetos de Cómeme

1. Nuestro personaje puede coger 10 comidas diferentes, y éstas ha de recogerlas siguiendo un orden (ver capítulo 10, apartado 10.3.2, Figura 10.1). Cada comida nos recompensará con 1000 puntos, y si conseguimos recoger 7 de ellas nos teletransportaremos a unas pantallas, inaccesibles de otro modo, en las que encontraremos las tres últimas comidas. Por último si recogemos las 10 terminará el juego.

2. Además de las comidas, nuestro personaje puede recoger galletas de la suerte, las cuales nos otorgarán una puntuación que oscila entre los 100 y los 800 puntos.

3. Y el último objeto que podemos recoger es el que nos da una vida de más, siempre y cuando tengamos menos de 10.

APÉNDICE C: Contenidos del CD y manual de instalación

Este apéndice vamos a dedicarlo a la explicación de los directorios que forman el CD-ROM que acompaña a esta memoria y a un pequeño manual de instalación para la ejecución de nuestra aplicación [0].

C.1. Contenidos del CD

La forma de nuestro directorio es la de la figura C.1.:

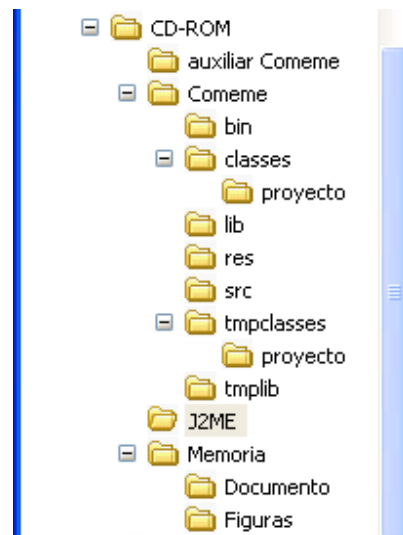


Figura C.1. Directorio del CD adjunto.

A continuación vamos a detallar los directorios más importantes:

- auxiliar Comeme: Contiene los ficheros `auxComeme.c`, `auxComeme2.c` y sus respectivos ejecutables `auxCom.exe` y `auxCom2.exe`. Además contiene el listado de pantallas original (listado hexadecimal) [3-12] y el listado de pantallas obtenido tras ejecutar `auxCom.exe`.
- Comeme: Esta carpeta contiene en el directorio `bin` tres ficheros, de los cuales dos son de suma importancia. Estos dos ficheros son los que nos permite ejecutar nuestra aplicación en un dispositivo móvil y son: `Comeme.jad` y `Comeme.jar`.

En `res` tenemos todos los gráficos de nuestro juego, incluida la pantalla de presentación.

En `src` tenemos todos los `<xxx>.java` (`<xxx>` son los distintos nombres de los ficheros) que forman nuestra aplicación.

Y por último tanto en `classes` como en `tmpclasses` tenemos otro directorio llamado `proyecto`, el cual contiene todos los ficheros con terminación `.class`.

- **Memoria:** Esta carpeta almacena todo lo relacionado con nuestra memoria. En el directorio `Figuras` almacenamos todas las figuras utilizadas en ésta, mientras que en `Documento` tenemos nuestra memoria en formato PDF.
- **J2ME:** Esta carpeta contiene la versión `j2sdk-1_5_0-beta` y `j2me_wireless_toolkit-2_0_01`, para el sistema operativo windows, utilizadas para la creación de nuestra aplicación.

C.2. Manual de instalación

En este apartado del apéndice C vamos a ver, en primer lugar, como instalar las herramientas necesarias para crear, cargar, compilar o ejecutar un proyecto en nuestro PC, y en segundo lugar, todo lo necesario para pasar nuestra aplicación a un dispositivo móvil.

C.2.1. Instalación de herramientas y ejecución en un PC

Para poder crear, compilar y ejecutar aplicaciones J2ME en nuestro ordenador, lo primero que debemos hacer es instalar el entorno de programación de J2SE (JDK). La versión que hemos utilizado para el proyecto está incluida en el CD adjunto, pero además podemos bajarnos la última versión de la URL <http://java.sun.com/j2se/downloads.html>. Una vez instalado J2SE, debemos de instalar J2ME, cuya versión utilizada en este proyecto se encuentra también en el CD adjunto o bajárnosla de la URL <http://java.sun.com/j2me/downloads.html>.

Una vez instalado todo lo anterior estamos en posición de ejecutar nuestra aplicación. Abrimos el wireless toolkit instalado anteriormente (ver Figura C.2.) y pinchamos en Ktoolbar.

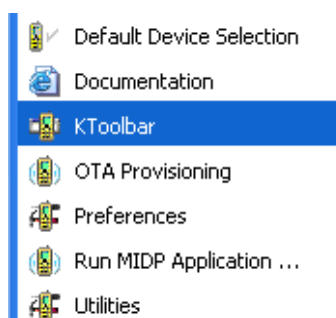


Figura C.2. La aplicación KToolbar.

Una vez hecho esto veremos aparecer la ventana del entorno (ver Figura C.3.).

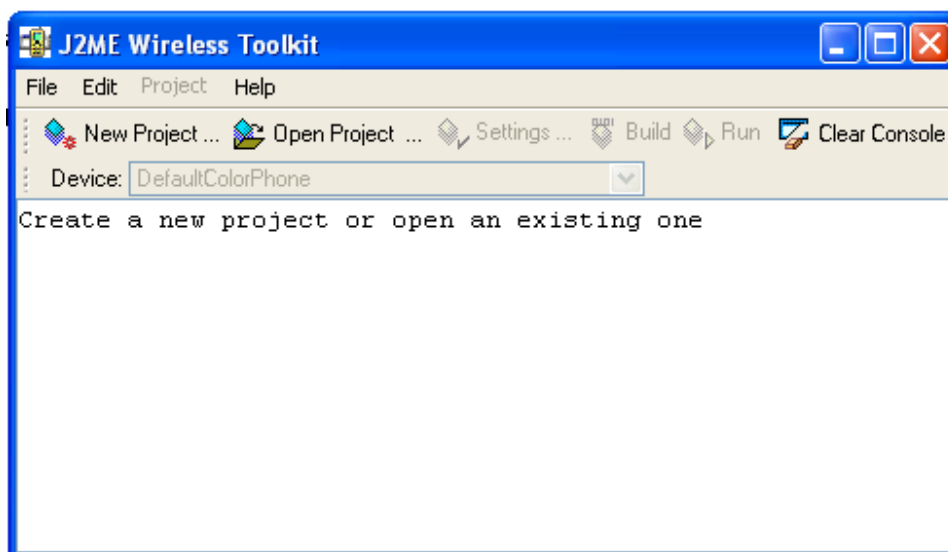


Figura C.3. Visualización del entorno de programación.

Para abrir nuestra aplicación tendremos que ubicar la carpeta Comeme junto a todas las aplicaciones J2ME, por ejemplo C : / WTK20 / apps, y pinchar en la opción *Open Project* de nuestro entorno de programación (ver Figura C.4.).

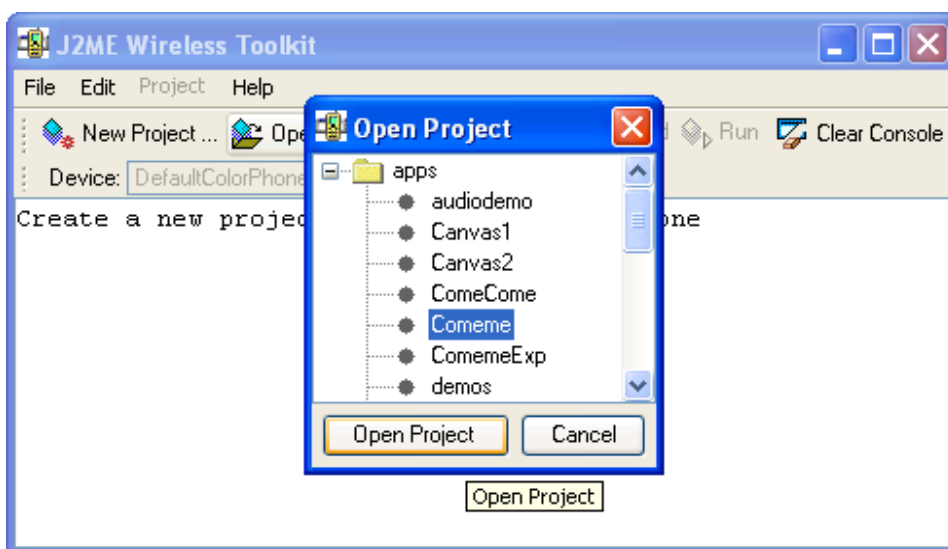


Figura C.4. Abriendo nuestra aplicación.

Tras abrir la aplicación podemos compilarla o ejecutarla pinchando en los correspondientes botones para tal uso, *Build* para compilar y *Run* para ejecutar (ver Figura C.5.).

Una vez aquí podemos seguir como explicamos en el apéndice B.

C.2.2. Transferencia de la aplicación a un dispositivo móvil

Una vez comprobado que nuestra aplicación funciona correctamente en el emulador, es hora de pasarlo a un dispositivo móvil. Lo primero que tenemos que hacer es empaquetar el programa y dejarlo listo para descargarlo en nuestro dispositivo. Así que nos vamos de nuevo a nuestro entorno KTollbar, desplegamos el menú *project* y seleccionamos *create package* del submenú *package* (ver Figura C.6.). KTollbar nos informa de que se han creado los archivos *Comeme.jar* y *Comeme.jad* (ver Figura C.7) dentro del directorio *bin* del wireless toolkit. Una vez tengamos estos dos archivos, sólo nos queda transferirlos a nuestro dispositivo mediante infrarrojos, bluetooth o el cable de transferencia de nuestro móvil. Otra forma de hacerlo, es subir estos dos archivos a un servidor *wap* o un espacio *web* y descargarlo desde nuestro móvil. Para esto es necesario que nuestro dispositivo cuente con navegador *wap* o *web* y soporte GPRS.

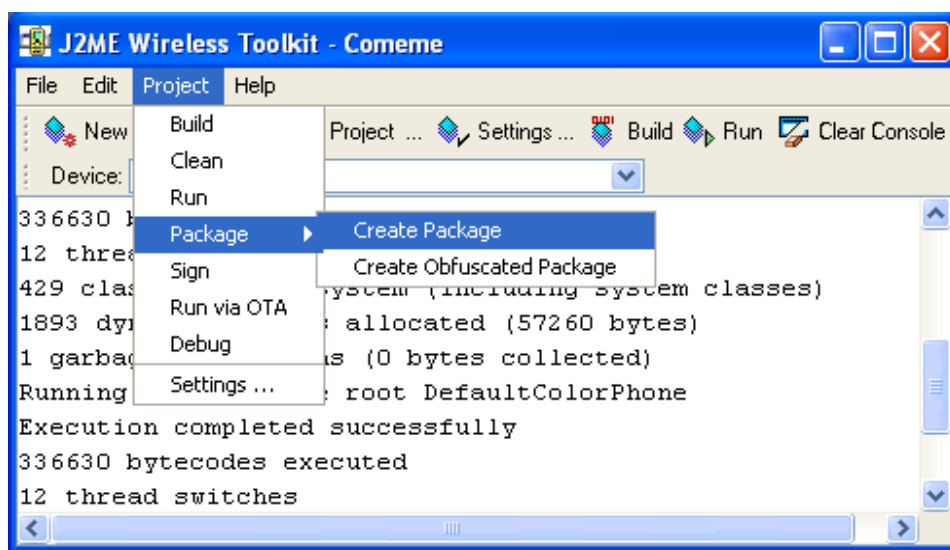


Figura C.6. Generación de *Comeme.jar* y *Comeme.jad*.



Figura C.5. Ejecución de nuestra aplicación.

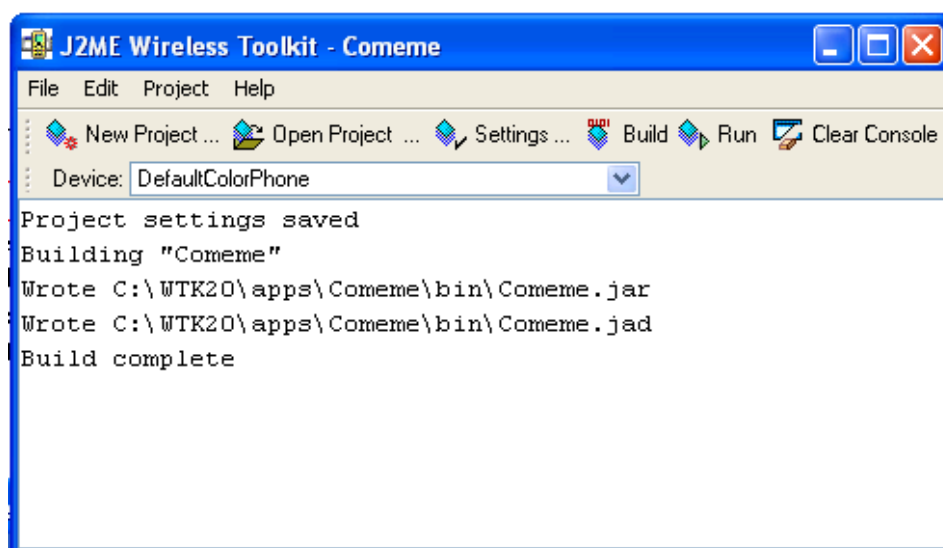


Figura C.7. Confirmación del empaquetado de Comeme.

Apéndice D. Mapa de Cómeme

En este apéndice vamos a ver el mapa completo del juego Cómeme separados en dos figuras (Figura D.1. y Figura D.2.). Esto es debido al gran tamaño que ocupa éste. La figura D.1. mostrará la parte izquierda del mapa completo, mientras que la figura D.2. mostrará la parte derecha.

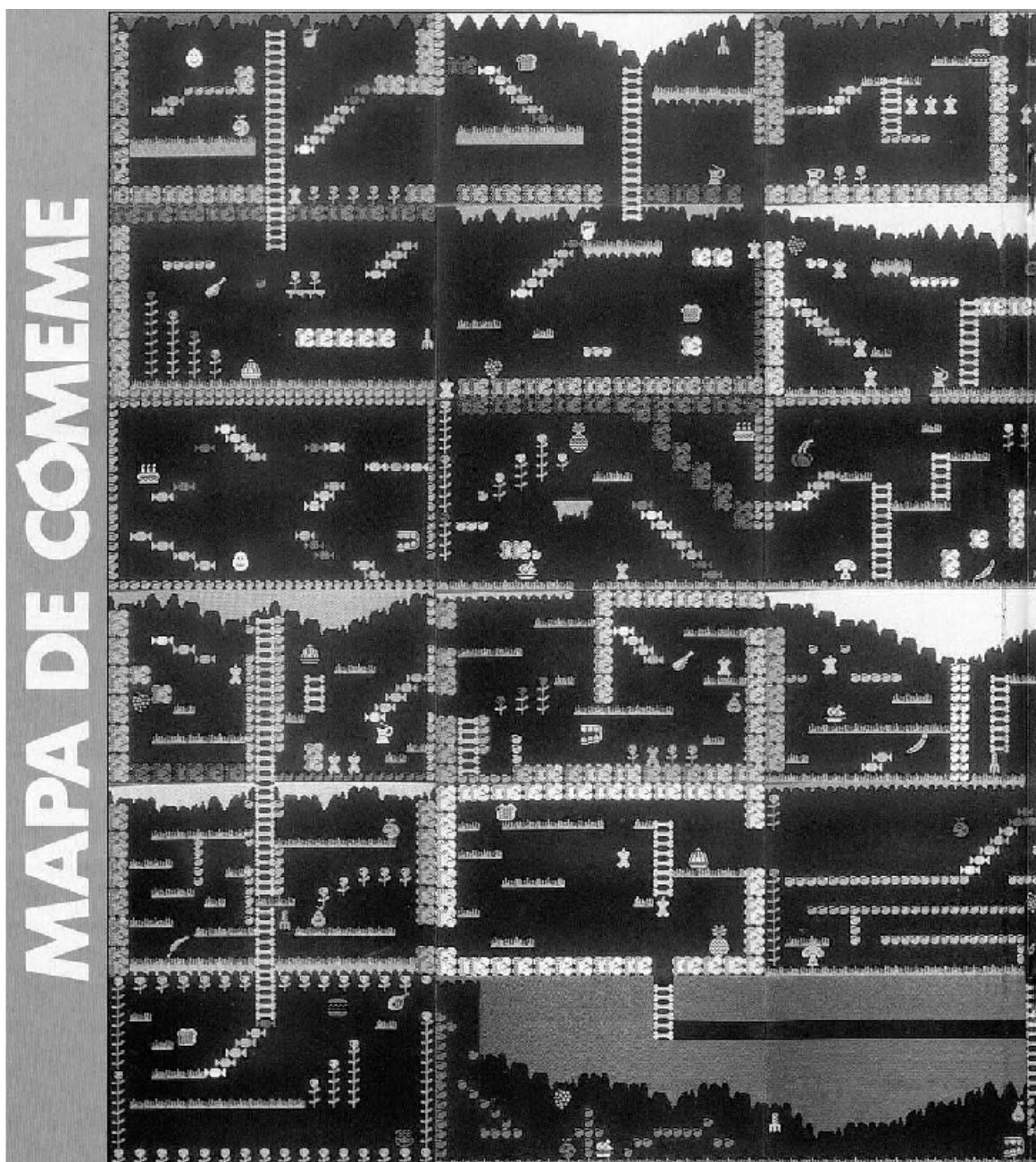


Figura D.1. Parte izquierda del mapa de Cómeme.

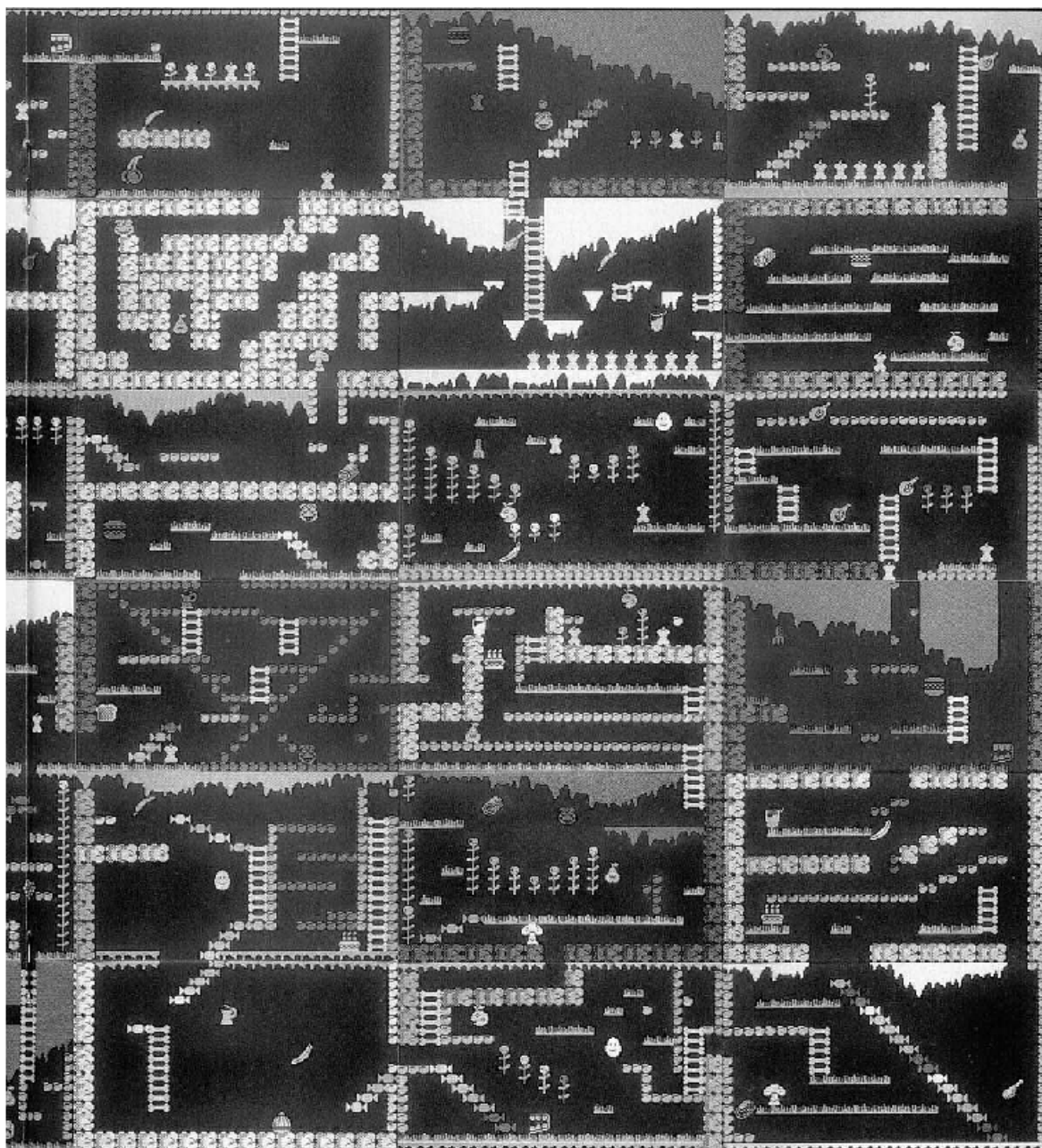


Figura D.2. Parte derecha del mapa de Cómeme.

Bibliografía y Referencias

[0] Alberto García Serrano, *"Programación de juegos para móviles con J2ME"*, editado por el propio autor (ISBN 84-609-1927-7), 2003. Disponible en la web: <http://www.agterrano.com/libros/Programacion de juegos para moviles con J2ME.pdf>

[1] Sergio Gálvez Rojas y Lucas Ortega Díaz, *"Java a Tope, edición electrónica"*, Universidad de Málaga, 2004.

[2] Diana Gruber, *"Action Arcade Adventure Set"*, The Coriolis Group, 1998.

[3] Pablo Ariza, *"Aprende a programar tu propio juego (I)"*, MicroHobby, año III, nº 97, pp.16-20, 1986.

[4] Pablo Ariza, *"Aprende a programar tu propio juego (II)"*, MicroHobby, año III, nº 98, pp.26-30, 1986.

[5] Pablo Ariza, *"Aprende a programar tu propio juego (III)"*, MicroHobby, año III, nº 99, pp.18-21, 1986.

[6] Pablo Ariza, *"Aprende a programar tu propio juego (I V)"*, MicroHobby, año III, nº 100, pp.22-27, 1986.

[7] Pablo Ariza, *"Aprende a programar tu propio juego (V)"*, MicroHobby, año III, nº 101, pp.26-30, 1986.

[8] Pablo Ariza, *"Aprende a programar tu propio juego (VI)"*, MicroHobby, año III, nº 102, pp.30-33, 1986.

[9] Pablo Ariza, *"Aprende a programar tu propio juego (VII)"*, MicroHobby, año III, nº 103, pp.28-32, 1986.

[10] Pablo Ariza, *"Aprende a programar tu propio juego (VIII)"*, MicroHobby, año III, nº 104, pp.34-37, 1986.

[11] Pablo Ariza, *"Aprende a programar tu propio juego (IX)"*, MicroHobby, año III, nº 105, pp.26-30, 1986.

[12] Pablo Ariza, *"Aprende a programar tu propio juego (X)"*, MicroHobby, año III, nº 106, pp.26-30, 1986.

[13] Apuntes de Laboratorio de Tecnología de Objetos, E.T.S.I. Informática, Universidad de Málaga.

[14] <http://www.javaworld.com/javaworld/jw-08-2004/jw-0809-j2me.html>, *"Using*

the MIDP 2.0. Game API by Carol Hamer“.

[15] <http://www.forum.nokia.com/main/1,6566,040,00.html?fsrParam=2-3-/main.html&fileID=4473> , juego sheepdog.

[16] Documentación de J2ME Wireless Toolkit 2.0_01, disponible en la web:
<http://java.sun.com/j2me/download.html>.