Desarrollo de juegos con J2ME

Manuel-Jesús Prieto Martín [1]

[1] Telefónica Investigación y Desarrollo, Spain
mjprieto@telefonica.es

Resumen: Este capítulo hacer una pequeña introducción al mundo del entretenimiento móvil y muestra como J2ME ha sido un elemento clave en él. Debido a que las tecnologías y los terminales avanzan y mejoran a gran velocidad, los servicios también deben evolucionar de forma rápida, lo que provoca un gran dinamismo en el mercado y un gran abanico de posibilidades para las empresas. A continuación, vamos a ver algunos de los marcos de trabajo en los que han aparecido servicios de entretenimiento para el mundo móvil. Todos los analistas coinciden en que este mercado será un gran negocio para todos los actores involucrados. Por un lado, tendremos a los operadores de telefonía que aumentarán sus beneficios, ya que serán ellos los que proveerán estos servicios y por lo tanto, generarán negocio en un primer momento con la provisión de los juegos y posteriormente con el uso de los mismos. Por otra parte, tenemos a las empresas de desarrollo de servicios para móviles. Entre estos servicios, tenemos los juegos (sea cómo sean estos y sea cual sea la tecnología en la que se basan) y dichas empresas de desarrollo se beneficiarán del aumento de la demanda de este tipo de servicios por parte de las operadoras. En este capítulo se presenta como J2ME está contribuyendo al desarrollo de este campo.

Palabras clave: J2ME; Juegos

Abstract. This chapter makes a small introduction to the world of mobile entertainment and shows how J2ME has been a key element in it. Because technologies and terminals are advancing and improving at high speed, services must also evolve rapidly, resulting in a very dynamic market and a wide range of possibilities for businesses. Below we will look at some of the frameworks in which entertainment services have appeared for the mobile world. All analysts agree that this market will be a big business for all actors involved. On the one hand, we will have the telephone operators that will increase their profits, since they will be the ones that will provide these services and therefore, they will generate business in a first moment with the provision of the games and later with the use of the same ones. On the other hand, we have the mobile services development companies. Among these services, we have games (no matter how they are and no matter what technology they are based on) and these development companies will benefit from the increased demand for this type of services by operators. In this chapter it is presented how J2ME is contributing to the development of this field.

Keywords: J2ME; Games

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

1 INTRODUCCIÓN A LOS JUEGOS EN MÓVILES

Dentro del mundo de los negocios basados en tecnologías móviles, uno de los campos que más expectativas está generando y que según los analistas crecerá exponencialmente en los próximos años es el mundo del entretenimiento a través del móvil. En este documento, vamos a hacer una pequeña introducción al mundo del entretenimiento móvil y vamos a situar la tecnología J2ME dentro de este campo, de tal manera que nos sirva de punto de comienzo para nuestra incursión en el entretenimiento móvil [1-5].

Si dejamos a un lado algunos casos aislados como Japón y Corea del Sur, el mercado del entretenimiento móvil, basado en terminales y tecnologías móviles, está aún en su comienzo, pero está empezando a ser una realidad indudable y palpable, sin duda alguna. Todos los analistas coinciden en que este mercado será un gran negocio para todos los actores involucrados. Por un lado, tendremos a los operadores de telefonía que aumentarán sus beneficios, ya que serán ellos los que proveerán estos servicios y por lo tanto, generarán negocio en un primer momento con la provisión de los juegos y posteriormente con el uso de los mismos. Por otra parte, tenemos a las empresas de desarrollo de servicios para móviles. Entre estos servicios, tenemos los juegos (sea cómo sean estos y sea cual sea la tecnología en la que se basan) y dichas empresas de desarrollo se beneficiarán del aumento de la demanda de este tipo de servicios por parte de las operadoras. Su negocio básico tiene dos vertientes, por una parte, pueden dedicarse a crear servicios para las operadoras y que estas sean las distribuidoras, o por otra parte, pueden crear un servicio y explotarlo junto con el operador, de tal manera que ambos se benefician de los resultados de dicho servicio. Este último modelo es el más común dentro del mercado japonés, en el que las empresas desarrollan un servicio y lo explotan junto con la operadora, repartiéndose los beneficios.

A pesar de estar en la fase inicial, las previsiones son muy alentadoras. Hace unos años, se hablaba de la gran cantidad de terminales móviles que habría en el futuro. Hoy, esas previsiones se han cumplido y la masa crítica de terminales móviles a la que se pueden orientar el mercado y ofrecer servicios es enorme. Además, los terminales se usan cada vez más para operaciones diferentes a la simple comunicación por voz. El más claro ejemplo de esto son los mensajes SMS que hoy por hoy son una verdadera fiebre para muchos usuarios. Si unimos estas cuestiones al hecho de que cada vez los terminales son mucho más potentes y atractivos, disponen de mejores pantallas y poseen sistemas de sonido cada vez más sofisticados, tendremos un número mayor de factores a favor de la explosión definitiva de los servicios móviles y entre ellos, de los servicios de entretenimiento móviles. Actualmente todas las operadoras están tomando posiciones y cada vez son más los servicios disponibles. Debido a que las tecnologías y los terminales avanzan y mejoran a gran velocidad, los servicios también deben evolucionar de forma rápida, lo que provoca un gran dinamismo en el mercado y un gran abanico de posibilidades para las empresas.

A continuación, vamos a ver algunos de los marcos de trabajo en los que han aparecido servicios de entretenimiento para el mundo móvil.

ISBN: 978-84-9012-861-9

Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

1.1 ENTRETENIMIENTO BASADO EN SMS Y MMS

SMS son las siglas de *Short Message Service*. La mensajería SMS se basa en el envío de mensajes de texto entre los terminales móviles. Como podemos comprender, la comunicación a través de mensajes de texto cortos es bastante limitada, pero este servicio ha sido un verdadero éxito en el mundo de las telecomunicaciones. En la actualidad hay en el mercado un amplio muestrario de servicios móviles basados en SMS, entre ellos, muchos servicios de entretenimiento.

Los juegos basados en SMS, como podemos comprender, deben ser muy sencillos ya que lo único que tenemos son mensajes de texto y además de un tamaño limitado. Algunos juegos típicos en este tipo de entorno son los juegos de pregunta-respuesta, o los juegos de aventuras basados en texto.
Otro tipo de entretenimiento basado en SMS y que no son exactamente juegos, son los sistemas de *chat* o similares que se utilizan este tipo de tecnología e incluso la integran con tecnologías no móviles, de tal forma que un usuario con un terminal móvil, puede *chatear* con un usuario que está frente a su PC.

Los servicios SMS pueden ser mejorados usando otras tecnologías, como la localización. La localización permite conocer de manera aproximada la posición del terminal móvil. De esta manera, las capacidades de los juegos y servicios de entretenimiento, pueden ser aumentadas y los resultados serán más atractivos para el usuario.

La evolución de los SMS han sido los MMS (si pasamos por alto los EMS). MMS es el acrónimo de *Multimedia Messaging Service*. Como su nombre indica, son mensajes en los que aparte de texto, se pueden incluir elementos multimedia, como son imágenes, sonidos e incluso video. Esta tecnología abre nuevas posibilidades en el mundo de los juegos basados en SMS ya que el resultado puede ser mucho más rico y por lo tanto el número de juegos y servicios que podemos crear es mayor [6-10].

1.2 ENTRETENIMIENTO BASADO EN WAP

Los juegos y servicios de entretenimiento basados en sistemas de navegación, trabajan sobre WAP (*Wireless Application Protocol*) y utilizan lenguajes de marcado como WML (*Wireless Markup Language*) o xHTML. De forma poco formal, podemos decir que WAP se puede considerar como la tecnología que nos permite navegar por Internet desde terminales móviles.

El tipo de servicios que nos permite ofrecer esta tecnología no son muy completos, pero

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

también hay un pequeño nicho de oportunidades en este campo. Los juegos que podemos desarrollados para este tipo de tecnología son similares a los presentados en el caso de los juegos basados en mensajería.

1.3 ENTRETENIMIENTO BASADO EN APLICACIONES

Existen varias tecnologías que nos permiten desarrollar aplicaciones que se ejecutan en el terminal. Entre estas aplicaciones, podemos incluir los juegos, y por lo tanto crear juegos propiamente dichos, que aprovechen todas las capacidades del terminal. Estas aplicaciones se ejecutan dentro del terminal y se instalan en el mismo a través de una conexión directa o a través de un proceso de descarga.

Sin duda alguna esta es la opción de las existentes que más posibilidades presenta y será la más demandada por parte del usuario ya que es la que aporta más a la experiencia del usuario y la que aprovecha al máximo todas las capacidades del terminal.

Para realizar las aplicaciones, tendremos varias tecnologías disponibles. Tenemos, entre otras, los teléfonos con sistema operativo Symbian, los teléfonos con sistemas Windows de Microsoft, también denominados *SmartPhones*, tenemos J2ME que será la tecnología en la que se basa este documento, la tecnología BREW...

Las dos primeras opciones, los sistemas Symbian y los *Smartphone* tienen la ventaja de que las aplicaciones se desarrollan como aplicaciones nativas del sistema, por lo que tienen la capacidad de aprovechar totalmente todos los elementos del terminal y por lo tanto, se pueden realizar aplicaciones, juegos en nuestro caso, más completas y ambiciosas.

Otra tecnología disponible para el desarrollo de aplicaciones y que actualmente es una de las más extendidas e implantadas en el mercado, es J2ME. A estas alturas ya tenemos un conocimiento claro de lo que es J2ME y deberíamos conocer su potencia y saber también sus carencias. Entre otras cosas, no tenemos acceso a todas las capacidades del terminal (agenda, APIs de bajo nivel...) y debemos tener en cuenta que los *midlets* J2ME funcionan sobre una máquina virtual, por lo que siempre serán más lentos que una aplicación nativa.

En lo comentado en el párrafo anterior, tenemos también la gran ventaja de J2ME y es su portabilidad, como es común en Java. Es decir, cuando creamos un *midlet* este puede instalarse y ejecutarse en terminales diferentes, de diferentes marcas, capacidades y con diferentes prestaciones. Como ya hemos comentado anteriormente y como veremos más adelante, para conseguir un resultado óptimo, habrá que ajustar el *midlet* a cada terminal. Esto se puede hacer en tiempo de ejecución o en tiempo de desarrollo, pero incluso con este esfuerzo extra, la reutilización de código es importante y por lo tanto el valor de J2ME como lenguaje de desarrollo para crear aplicaciones para un gran número de terminales muy diferentes entre si permanece intacto [11-15].

ISBN: 978-84-9012-861-9

Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Por otra parte, se está trabajando y se están definiendo constantemente ampliaciones de la especificación básica de J2ME para la provisión de APIs que aumenten las posibilidades funcionales de J2ME y así se aprovechen al máximo las capacidades y las prestaciones de los terminales. Por ejemplo, tenemos APIs adicionales (creadas o en proceso de definición) para la gestión de la tecnología *BlueTooth*, para la utilización de las capacidades de mensajería de los terminales o para la gestión de ficheros.

2 JUEGOS ESTÁTICOS

2.1 INTRODUCCIÓN

En primer lugar, como primera aproximación a la programación de videojuegos con J2ME, vamos a ver cómo desarrollar lo que denominaremos juegos estáticos. Este tipo de juegos, son juegos en los que no hay un movimiento continuo o masivo de elementos dentro de las pantallas del juego. Dentro de este grupo de juegos, podemos incluir los juegos de sobremesa, en los que se nos presenta una pantalla, realizamos un movimiento y se nos presenta una nueva pantalla, siendo todas estas pantallas netamente estáticas con respecto a los elementos gráficos que la componen.

El objetivo de este capítulo es comenzar con la gestión avanzada de gráficos y de la interacción con el usuario, para afianzar una serie de conceptos e ideas que serán básicos en el desarrollo de juego más complicados y completos. Para ilustrar todo esto, vamos a realizar un juego de este tipo. En concreto, el juego a desarrollar en el conocido buscaminas, que al ser muy popular, nos evita el plantearnos las normas o reglas del juego y al ser tan sencillo, nos permite centrarnos en los puntos del desarrollo que realmente nos interesan para nuestro objetivo.

Toda la funcionalidad contenida en este ejemplo y por lo tanto todo lo que se presenta en este capítulo está basada en la versión 1.04 de MIDP, por lo que no usa ninguna de las características orientadas al desarrollo de juegos que se incluyen dentro de la versión 2.0 de la especificación MIDP.

2.2 INTRODUCCIÓN AL OBJETO GRAPHICS

Antes de comenzar con el desarrollo del juego propiamente dicho, vamos a ver una introducción al objeto *Graphics*, de tal forma que el proceso de creación de las pantallas posteriormente sea más sencillo de comprender.

El objeto *Graphics*, perteneciente al paquete *javax.microedition.lcdui*, es el componente esencial de cualquier operación gráfica que se realice dentro de los midlets. El objeto *Graphics*, proporciona funciones para dibujar texto, imágenes, líneas, rectángulos y arcos de curva.

Como veremos en el código de ejemplo dentro de este mismo capítulo, las operaciones de dibujo se pueden realizar tanto sobre la pantalla del terminal *directamente* como sobre una imagen, que se compondrá totalmente independiente de la pantalla y que luego será mostrada en la pantalla de una sola vez. Esta solución evita que se produzcan parpadeos en las animaciones

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

y en los procesos de repintado de la pantalla. Para dibujar sobre una imagen externa a la pantalla, crearemos primero un objeto *Image* con las dimensiones deseadas (típicamente el tamaño de la pantalla o lo que es lo mismo, del *Canvas* final en el que vamos a dibujar). Una vez que tenemos la imagen, llamamos al método *getGraphics*, que nos retornará un objeto *Graphics*. Este objeto se puede usar tantas veces como queramos y definitivamente,

dentro del método *paint* del *Canvas* dibujaremos la imagen compuesta, lo que provoca que se dibuje esta imagen definitivamente sobre la pantalla.

El sistema de coordenadas utilizado para las operaciones de dibujo tiene su punto de origen en la esquina superior-izquierda del rectágulo de dibujo, es decir, de la pantalla o de la imagen en el caso de estar dibujando sobre una imagen externa.

Para controlar mejor los efectos sobre la pantalla de las operaciones de dibujo, podemos definir un rectángulo de *clipping* o recorte, de tal manera que sólo dentro de dicho rectángulo tendrán efecto las operaciones de dibujo. Se puede definir un rectángulo de *clipping* de tamaño cero, de tal forma que las operaciones de dibujo no tienen ningún efecto sobre la pantalla [16].

Los puntos de anclaje o *anchor points* se utilizan para posicionar el texto y las imágenes a la hora de dibujarlos. Estos puntos de anclaje se utilizan para reducir la necesidad de computación necesaria para situar el texto o la imagen. Por ejemplo, para centrar un texto, la aplicación necesita determinar la anchura del mismo a través de los métodos *stringWidth* o *charWidth* y realizar los cálculos necesarios para situar el texto en la posición correcta. Para definir el punto de anclaje debemos especificar un valor para el parámetro horizontal y otro valor para el parámetros vertical. Los valores que puede tomar el parámetro horizontal son las constantes LEFT, HCENTER y RIGHT y los valores que puede tomar el parámetro vertical son las constantes TOP, BASELINE y BOTTOM. Para combinar ambos parámetros se utiliza el operado OR:

```
g.drawString("Texto de prueba", 10, 10,

Graphics.TOP | Graphics.LEFT);
```

La posición del texto con respecto al punto especificado en el método de dibujo es determinada por el punto de anclaje.

2.3 COMENZANDO: PANTALLAS DE PRESENTACIÓN E INSTRUCCIONES

Para comenzar vamos a ver cómo se construye la pantalla de presentación del juego. Esta pantalla está presente en la mayoría de los juegos y en nuestro caso nos permitirá realizar nuestra primera aproximación al objeto *Graphics* y al funcionamiento de los objetos tipo *Canvas*.

Lo primera cuestión a plantearnos a la hora de desarrollar un juego, una vez que ya tenemos claras las especificaciones del juego en cuestión, es el diseño del software a desarrollar. Este punto es

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

esencial ya que nos facilitará enormemente el desarrollo y hará que el software creado sea de mayor calidad, al favorecer la portabilidad y adaptabilidad del software. Estas características son muy importantes en cualquier software, pero en el caso de los midlets son aún más importantes, ya que es un imperativo en el proceso de desarrollo de software J2ME si queremos que nuestro midlet se adapte a las capacidades de los diferentes terminales y que la implementación de esta adaptación no suponga un esfuerzo enorme.

El objetivo que nos propondremos al crear un juego, será que este se adapte de la mejor forma posible a las características de los terminales. Para conseguir este ambicioso objetivo, tendremos que hacer el diseño de las pantallas con sumo cuidado y adaptar nuestras operaciones de dibujo siempre que sea posible, a las características del terminal.

Lamentablemente, para conseguir un resultado óptimo y totalmente satisfactorio, deberemos crear en la mayoría de los casos una versión diferente del midlet para cada terminal. Cada una de estas versiones, tendrá una serie de imágenes adaptadas a las características del terminal. Bien, si partimos de este supuesto, tendremos que poner especial cuidado en el diseño del software, para que se puedan sustituir ciertas partes del código, las correspondientes a las pantallas que debamos adaptar, de forma sencilla y el impacto de esta acción se mínimo.Como primera aproximación al mundo de los gráficos orientados a juegos en J2ME, vamos a hacer una pantalla de presentación, sin muchas pretensiones pero que nos servirá para ver algunas de las técnicas básicas de trabajo con J2ME en general y con la programación de juegos en particular. La pantalla de presentación que vamos a hacer será una pantalla sencilla con una imagen. La primera versión de la pantalla de presentación será la siguiente:

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
import javax.microedition.lcdui.*;

import java.io.IOException;

public  class  Presentacion  extends  Canvas{  private  Buscabombas

    buscabombas = null; private Menu menu = null;

    private boolean salir = false;

    private Image logo; private Image tex-

    toImg;


    Font f1 = Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD,Font.SIZE_MEDIUM);

    int ancho; int alto;

    int   anchoTit2;   int   an-

    choTit1; int ximg;

    int yimg;


    public Presentacion(Buscabombas buscabombas) { this.buscabombas = buscabom-

        bas;

        menu = new Menu(Recursos.getString(
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                Recursos.ID_NOMBRE),

                List.IMPLICIT,buscabombas);

        try{
            logo = Image.createImage("/LogoBuscabombas.png");
        }catch(IOException ioe){ ioe.printStack-

            Trace();

        }
        ancho = this.getWidth(); alto =

        this.getHeight();

        ximg = (int)((ancho-logo.getWidth())/2);
        yimg = (int)((alto-logo.getHeight())/2);

    }
    protected void keyPressed(int keyCode){ buscabom-

        bas.mostrar(menu);

    }
    protected void paint(Graphics g) {

        // Borramos la pantalla, pintándola de blanco
          g.setColor(0xFFFFFF);

          g.fillRect(0,0,ancho,alto);

          g.drawImage(logo,ximg,yimg, Graphics.LEFT |

                    Graphics.TOP);

      }
}
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

En esta primera clase ya presenta ciertas cuestiones que debemos tener siempre presentes. Lo primero que debemos resaltar es el hecho de que todas las inicializaciones de las variables y todos los cálculos que se pueda, se deben realizar en el constructor de la clase. Los terminales móviles tienen una capacidad de proceso reducida y por lo tanto, cualquier operación tiene un cierto impacto en el rendimiento del midlet. Si todas estas operaciones las hacemos antes de que la pantalla sea mostrada, el funcionamiento posterior será mucho más ágil y este tiempo extra necesario para la creación de la clase es transparente en cierta medida para el usuario y por lo tanto, la experiencia de este es el uso del midlet es mucho mejor [17-20].

Para realizar nuestra pantalla de presentación, los cálculos a realizar son tan sencillos como recoger el tamaño del *Canvas* de la pantalla y calcular el punto en el que se tiene que dibujar la imagen, para que esta parezca centrada dentro de la pantalla.

Si durante la visualización de la pantalla de presentación el usuario pulsa alguna tecla, se pasa directamente a la pantalla principal de opciones del juego que explicaremos más adelante.

El método *paint* del juego lo único que hace es dibujar la imagen en la pantalla de terminal, después de limpiar esta. El resultado dentro del emulador es el siguiente:

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

2.4 PANTALLAS BÁSICAS BASADAS EN CANVAS

A continuación, vamos a ver algunas pantallas del juego, que sin ser la pantalla principal del mismo, tendrán ciertas características y funciones que son importantes y que conviene remarcar. Son pantallas basadas en *Canvas*, con alguna animación simple. En el caso concreto que nos ocupa, su función no es más que servir de pantalla de configuración o de puntuación, pero implementan ciertas características que pueden ser muy útiles en el desarrollo de los juegos.

Para comenzar, vamos a ver la pantalla de configuración.

La pantalla de configuración del nivel de dificultad del juego, está contenida dentro de la clase Nivel. Esta pantalla en un objeto Canvas, que presenta una lista con el número de bombas que hay ocultas en el juego, y por lo tanto, cuanto mayor sea el número de bombas, mayor será la dificultad del juego. Para hacer esta pantalla, y como primera aproximación a la creación de listas de opciones más atractivas que las que proporcionan los terminales J2ME dentro de sus implementación, tendremos una lista de elementos, pero en este caso, será la clase la que controle totalmente el proceso de pintado de la lista. De momento, nos conformaremos con que la lista muestre en negrita el elemento seleccionado [21]. A partir de esta implementación y viendo el sencillo funcionamiento de esta pantalla, podemos utilizar nuestra imaginación para hacer listas tan complicadas y atractivas como queramos. Para confeccionar esta lista, debemos tener claro que hay que controlar los siguientes elementos:

- Proceso de dibujo de los elementos de la lista
- Proceso de control y dibujo del elemento seleccionado
- Control de la transición de un elemento selecciona a otro
- Gestión de las acciones del usuario

A parte de estos elementos, podríamos incluir un hilo o *thread* que realice ciertas operaciones sobre el listado, aumentando así su vistosidad. Típicamente, podríamos hacer alguna animación sobre los elementos de la lista, especialmente sobre el elemento seleccionado en cada momento.

El siguiente listado muestra el código fuente de la clase Nivel:

```
import javax.microedition.lcdui.*;

public class Nivel extends Canvas implements CommandListener {
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
private Buscabombas buscabombas = null; private Opciones opcio-
nes = null; private Command salir = null;
private String msg = "";
private Font f1 = Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_BOLD, Font.SIZE_MEDIUM);
private Font f2 = Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_PLAIN, Font.SIZE_MEDIUM);
int ancho;
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
int alto;
private int selected = 0; private String[] items = null;;

private int[] niveles = null;


public Nivel(Buscabombas buscabombas, Opciones opciones) {

    this.opciones = opciones; this.buscabombas = buscabombas;


    ancho = this.getWidth(); alto = this.getHeight();

    niveles = Recursos.getNiveles(); items = new

    String[niveles.length]; for (int i=0; i<niveles.length; i++){

        items[i] = niveles[i] + " bombas";

    }

    selected = buscabombas.getNivel();

    setCommandListener(this);

    salir = new Command(Recursos.getString( Recursos.ID_SALIR), Com-

                    mand.EXIT, 1);

    addCommand(salir);

}

public void commandAction(Command command, Displayable displayable)

                    {

    buscabombas.mostrar(opciones);
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
}

public void paint (Graphics g){ g.setColor(255, 255, 255);
    g.fillRect(0, 0, ancho, alto); g.setColor (0, 0, 0);

    int y = 3; g.setFont(f2);
    for (int i=0; i<items.length; i++){ if (i == selected){
            // Pinto el elemento seleccionado g.setFont(f1); g.draw-
            String(items[i], 2, y,
                    Graphics.TOP | Graphics.LEFT); g.setFont(f2);
        } else { g.drawString(items[i], 2, y,
                    Graphics.TOP | Graphics.LEFT);
        }
        y += f2.getHeight()+2;
    }
}
public void keyPressed (int tecla){ tecla = getGameAction(tecla);

    // Hacia abajo
    if (tecla == Canvas.RIGHT || tecla == Canvas.DOWN){ if (selected < items.length-1){
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
            selected++;
            repaint(); this.serviceRepaints(); return;
        }
    }
    // Hacia arriba
    if (tecla == Canvas.LEFT || tecla == Canvas.UP){ if (selected > 0) {
            selected--; repaint();
            this.serviceRepaints(); return;
        }
    }

    // Gestión del click
    if (tecla == Canvas.FIRE){ buscabombas.setNi-
        vel(selected); buscabombas.mostrar(opciones);
    }
}

            }
```

Dentro del método paint de este listado, vemos como debemos dibujar todos los elementos y comprobar cuál es el elemento seleccionado para dibujarlo con alguna característica especial. En este caso, se cambia la fuente de texto con la que se escribe dicho elemento, pero podríamos ponerle una imagen a la izquierda indicando que es el elemento seleccionado o hacer que dicho elemento realice una determinada animación, que lo diferencie del resto.

Otro punto a destacar es la gestión de la pulsación de las teclas por parte del usuario (método *keypressed*). En este método, se mueve la selección dentro de la lista para abajo o para arriba en

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

función de la tecla que se pulse, y si se pulsa la tecla de fuego (*fire*), se toma como selección definitiva el elemento selecciona en ese momento y se va directamente a la pantalla de opciones principales de nuestro juego. En este caso en concreto, se modificará el nivel de dificultad del juego a través de la modificación del número de bombas escondidas [22-25].

2.5 PANTALLA CON UNA ANIMACIÓN MUY SIMPLE

A continuación vamos a ver una pantalla con una animación muy sencilla, que nos servirá para tener un primer acercamiento al desarrollo de animaciones y al uso de hilos o *threads*. La pantalla es cuestión, dentro de nuestro juego será la pantalla correspondiente a la opción de *Acerca de*, es decir, la pantalla que presente al desarrollador del juego. En nuestro caso, esta pantalla mostrará una imagen, el nombre del juego y el nombre del desarrollador. El código de esta pantalla es el siguiente:

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241

Ediciones Universidad de Salamanca - cc by nc dc

```
import javax.microedition.lcdui.*;

import java.io.IOException;

public class AcercaDe extends Canvas implements Runnable{



    private Buscabombas buscabombas = null; private Menu menu =

    null;

    private boolean salir = false;

    private Thread thread; private Image logo;



    Font f1 = Font.getFont(Font.FACE_SYSTEM,

                Font.STYLE_BOLD,Font.SIZE_MEDIUM);

    Font f2 = Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD,Font.SIZE_SMALL);

    int ancho; int alto;

    String tit1 = "BUSCABOMBAS";

    String tit2 = "Manuel J. Prieto (2003)"; int anchoTit2;
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
int anchoTit1;

int  xf1;  int  yf1;

int xf2; int yf2;



int ximg; int yimg;



int x;

int incremento=1;

/** Constructor */

public AcercaDe(Buscabombas buscabombas, Menu menu) { this.buscabombas = buscabombas;

    this.menu = menu;

    try{

        logo = Image.createImage("/bomba.png");

    }catch(IOException ioe){ ioe.printSta-

        ckTrace();

    }

    ancho = this.getWidth();
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
    alto = this.getHeight();

    anchoTit2 = f2.stringWidth(tit2); anchoTit1 =

    f1.stringWidth(tit1); xf1 = (int)((ancho-anchoTit1)/2); yf1 =

    (int)alto/2;

    xf2 = (int)((ancho-anchoTit2)/2);

    yf2 = yf1+f1.getHeight()+5;

    ximg = (int)((ancho-logo.getWidth())/2); yimg = (int)((yf1-

    logo.getHeight())/2);



    x=xf1;

    thread = new Thread(this); thread.start();

}

public void run(){ while (!salir){

        this.repaint(); this.serviceRepaints(); x+=incremento;

        if (x==xf1+anchoTit1) incremento = -1; if (x==xf1) incremento=1;
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        try{
            thread.sleep(50);
        }catch(InterruptedException ie){ ie.printStackTrace();
        }
    }
    buscabombas.mostrar(menu);
}

protected void keyPressed(int keyCode){ salir = true;
}
protected void paint(Graphics g) {

    // Borramos la pantalla, pintándola de blancho g.setColor(0xFFFFFF); g.fi-
    llRect(0,0,ancho,alto);

    g.drawImage(logo,ximg,yimg, Graphics.LEFT | Graphics.TOP);

    g.setColor(0x0000FF);
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        // Título principal de la pantalla de presentación

        g.setFont(f1); g.drawString(tit1,xf1,yf1,

                Graphics.LEFT | Graphics.TOP);

        // Título principal de la pantalla de presentación

        // Si el texto es muy grande, se disminuye el

        // tamaño de la fuente y el texto if (anchoTit2 > ancho){

          tit2 = "Manuel J. Prieto"; anchoTit2 = f2.stringWidth(tit2);

          xf2 = (int)((ancho-anchoTit2)/2);

        }

        g.setFont(f2); g.drawString(tit2,xf2,yf2,

                Graphics.LEFT | Graphics.TOP);

        g.setColor(0xFFFFFF); g.fillRect(x,yf1,3,f1.getHeight());


        // Hacemos un recuadro g.setColor(0x0000FF); g.drawRect(1,1,ancho-

        3,alto-2);


      }

  }
```

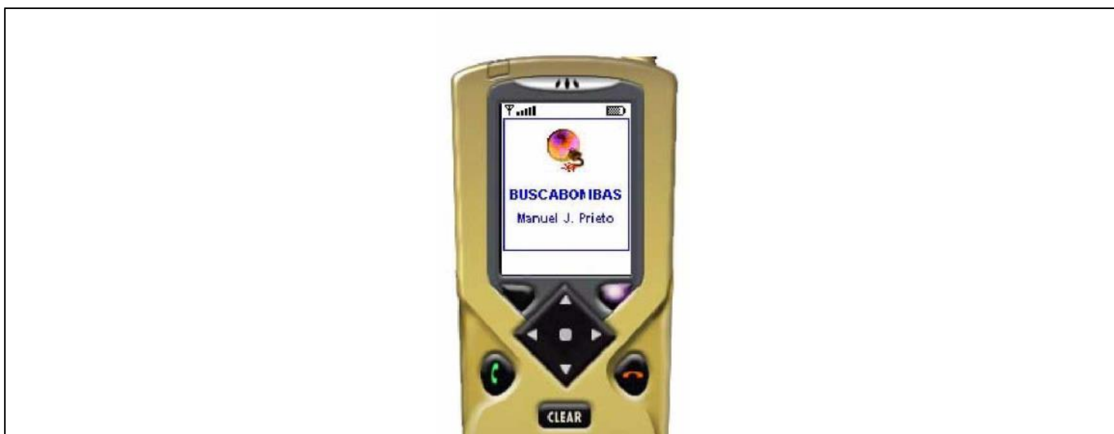

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

El resultado de este código se muestra en la siguientes figuras, en la que se muestra la pantalla de presentación del juego. Como vemos en la figura de la izquierda, el texto *Manuel J. Prieto (2003)* es demasiado grande para mostrarlo en la pantalla y es recortado. Este hecho está controlado dentro del método *paint*, tal como se comenta dentro del propio código. Sin embargo, como se puede ver en la imagen de la derecha, las dimensiones de la pantalla del terminal nos permiten mostrar el texto completo, sin recortar. Esto es un ejemplo claro de lo que hablábamos al comienzo del capítulo, en el que se comentaba que debíamos de tener siempre presente la adaptación de nuestros midlets a las características de cada terminal. En este caso, la adaptación es muy sencilla y se puede hacer en tiempo de ejecución dicha adaptación [26-30], pero en la mayoría de las situaciones, las adaptaciones serán más complicadas o imposibles, y tendremos que realizar varias *versiones* del midlet, adaptadas a diferentes tipos de terminales. Por ejemplo, si tenemos imágenes como la que hemos visto en la pantalla de presentación, tendremos que tener varias versiones con diferentes tamaños, para los diferentes grupos de terminales. Una solución dinámica podría ser la de incluir en la distribución del midlet, versiones de la imagen de diferentes tamaños, y en tiempo de ejecución recoger el tamaño del *Canvas* y cargar la imagen correspondiente. Esta solución presenta un gran problema que la hace inviable en la mayoría de los casos y es que el tamaño del fichero *jar* a distribuir sería demasiado grande. Sería demasiado grande desde varios puntos de vista:

Cuanto mayor sea el tamaño del fichero *jar*, mayor será el tiempo de descarga del mismo, con todo lo que esto implica

Muchos terminales tienen límites de tamaño para los *midlets*, de tal forma, que si se intenta descargar un *midlet* con un tamaño superior al límite, obtendremos un error y el *midlet* no se instalará

El espacio de almacenamiento disponible en los terminales para los *midlets* es limitado y por lo tanto, cuanto mayor sea el *midlet*, menos espacio dejará libre y más posibilidades hay de que no se pueda instalar por falta de espacio de almacenamiento



ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

En las figuras anteriores vemos una zona blanca sobre la M del título en el caso de la figura de la izquierda y vemos una zona del mismo color del fondo sobre la primera A del título en el caso de la figura de la derecha. Bien, esta franja del color del fondo de la pantalla será el único elemento animado de la pantalla de presentación y lo que hará será moverse de izquierda a derecha y viceversa sobre el texto del título del juego. Es una animación muy simple, pero servirá como primer acercamiento y punto de partida para empezar al modificar el código que se proporciona y crear animaciones más complejas y vistosas.

En el código del canvas correspondiente al *canvas* de la pantalla de presentación que se ha presentado anteriormente, vemos que se inicializan todas las variables posibles dentro del constructor de la clase. Como ya hemos comentado, esto se hace porque todos estos valores son usados en el método *paint* cada vez que se le llama, y se pueden calcular desde el primer momento, por lo que si demoramos su cálculo y lo hacemos dentro del método *paint*, haremos los mismos cálculos, con los mismos resultados cada vez que se invoque al método *paint*, mientras que en este caso estas operaciones se realizan sólo una vez. Debemos tener siempre en la cabeza que el objetivo último de los juegos desarrollados en J2ME es la ejecución de los mismos en un dispositivo móvil, con unos recursos muy limitados en cuanto a memoria y capacidad de proceso se refiere. Por esta razón debemos optimizar el código todo lo que podamos y siempre que podamos. En el ejemplo que nos ocupa, al realizar una animación, estamos llamando constantemente al método *paint* por lo que la mejora de rendimiento de nuestra aplicación al inicializar todos los valores en el constructor será lo suficientemente rentable como para optar por esta solución.

Otro punto a destacar del ejemplo anterior es la implementación de la interfaz *Runnable* y el uso de *Threads* o hilos de ejecución para realizar la animación del título. Como vemos, el método central de todo el proceso será el método *run* que será el método que se ejecute y en el que debemos crear el bucle que actualice y gestione constantemente todo el proceso correspondiente con el *canvas*, especialemente con la animación. Dentro del método *run* cabe destacar la invocación del método *repaint*, que indica al sistema que debe repintar el *canvas* (la pantalla de presentación en este caso) y también cabe destacar el método *serviceRepaints* que obliga al sistema a repintar (a cumplir con lo ordenado en el método *repaint*) de forma inmediata, dando prioridad total

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

a esta acción. El método *serviceRepaints* es muy importante si queremos conseguir que las animaciones en nuestros juegos funcionen correctamente y el resultado sea agradable para el usuario y profesional. Para finalizar el proceso que se realiza constantemente dentro del método *run* tenemos la orden de dormir o para el *thread*. Si no hacemos esta pequeña pausa, la animación irá demasiado rápida y no conseguiremos el efecto deseado. El código anterior está preparado para ser ejecutado dentro del emulador por lo que la pausa puede ser demasiado larga para la capacidad de proceso de un terminal real. Aún así, esta pausa no será de la misma cantidad de tiempo en todos los terminales, ya que no todos los terminales tienen la misma capacidad de proceso. En un futuro ya veremos cómo solucionar este tipo de situaciones y como adaptarnos dinámicamente a la capacidad de proceso del terminal.

Cuando se pulsa una tecla, el método *keyPressed* establece la variable que controla el bucle constante del método *run* a *true* de tal forma que se finaliza el proceso. Es decir, el proceso continuo que tiene lugar dentro de nuestra clase de presentación es parado al pulsar una tecla

del terminal. Dentro del método *run*, una vez que finaliza el proceso principal, se avisa al *midlet* (clase principal de la aplicación) de este hecho para que muestre el menú principal de la aplicación.

Por último, el método *paint* de la pantalla de presentación, lo único que hace es dibujar cada elemento en su lugar correspondiente dentro de la pantalla cada vez que es invocado. La posición de elemento animado es lo único variable a lo largo de la ejecución. El método *paint* se podría mejorar desde el punto de vista del rendimiento haciendo que el cálculo que se hace para comprobar si el texto *Manuel J. Prieto (2003)* se puede mostrar o debe ser recortado, se podría hacer un única vez dentro dentro del constructor.

2.6 LÓGICA Y PANTALLA PRINCIPAL DEL JUEGO

Por fin vamos a ver la última clase del juego, que tendrá la lógica del mismo y además presentará las técnicas básicas del desarrollo de los juegos que hemos denominado estáticos.

Hemos denominada a la clase Juego y por supuesto, extiende a la clase *Canvas*. Desde el punto de vista del desarrollo de los juegos en general, los puntos que debemos destacar en esta clase comienzan en el constructor. Como ya hemos comentado, hacemos todas las inicializaciones posibles dentro del constructor, para que una vez que se comienza a jugar de verdad, el funcionamiento sea más dinámico, ya que ahorramos gran cantidad de tiempo, al tener todo lo posible precalculado. En este caso, realizamos ciertas operaciones en función del tamaño de la pantalla del terminal, de tal forma que la pantalla principal del juego se adapta de forma totalmente automática a las características del terminal sobre el que se está ejecutando.

Dentro del constructor, creamos la imagen de la que obtendremos el objeto *Graphics* que nos servirá para dibujar en cualquier punto del código. El código en el que obtenemos este objeto es el siguiente:

ISBN: 978-84-9012-861-9

Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
img = Image.createImage(anchoCanvas,altoCanvas);
off = img.getGraphics();
```

A partir de este momento, cualquier operación gráfica se realiza sobre el objeto *off* y al final, en el método *main* lo único que tendremos que hacer es dibujar la imagen *img* que hemos ido componiendo en los diferentes métodos de la clase.

En nuestro caso, optamos por este tipo de funcionamiento del proceso de dibujo, para que los diferentes métodos de la clase puedan dibujar. Es decir, es consecuencia de una decisión simplemente de diseño del software. Sin embargo, esta técnica es muy utilizada dentro del campo del desarrollo de juegos, para implementar lo que se denomina *buffer* doble o *double buffering*. Esta técnica evita que al dibujar directamente sobre la pantalla se produzcan parpadeos. Si dibujamos todo en una imagen, sin mostrar nada por la pantalla y una vez compuesta la imagen se muestra esta por pantalla, el resultado es mucho más satisfactorio. Hay terminales móviles que ya implementan de forma interna esta técnica y que por lo tanto no necesitan que se use esta técnica para evitar los parpadeos de pantalla. La clase *Canvas* nos provee un método para comprobar si el terminal ya implementa la técnica del *double buffering* en la implementación de *Graphics*. Una implementación típica del proceso de dibujo, en base a este hecho es la siguiente:

```
protected void paint(Graphics g) {

    if (this.isDoubleBuffered()){

            // Dibujamos directamente sobre g
    } else {

            // Creo la imagen, obtengo el objeto Graphics

            // Dibujos fuera de la pantalla

            // Dibujo la pantalla compuesta

}
```

Otro punto a destacar dentro la clase Juego, es la gestión de las acciones del usuario, es decir, cómo responde el midlet a las pulsaciones de las teclas del usuario. Evidentemente, dentro de un juego es básico que las acciones del usuario sean bien gestionadas y que este reciba una respuesta ágil y rápida a sus acciones. Cuando el usuario pulsa una tecla, este evento se recoge dentro del método *keyPressed*, que recibe como parámetro el código de la tecla pulsada. Este código se debe comparar con las constantes que provee la clase *Canvas*, para comprobar qué tecla se ha pulsado.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Para recuperar el valor de la tecla pulsada, se debe utilizar el método *getGameAction*.

Una vez que hemos realizado las operaciones oportunas, en función de la tecla pulsada, repercutimos los cambios en la pantalla del terminal. Para hacer esto, solicitamos el repintado de la pantalla con el método *repaint* y para que este repintado se haga de forma inmediata, se invoca al método *serviceRepaints*.

Desde el punto de vista de la lógica de este juego en particular, tenemos las siguientes estructuras de datos básicas:

- campo – Será una matriz de dos dimensiones que simboliza el contenido de todas las celdas del juego. Si el valor de una celda es –1, indicará que posee una bomba y si el valor de distinto de –1, indicará el número de bombas que posee la celda alrededor de si misma
- estado – Será una matriz de dos dimensiones que mapea las celdas del juego y su contenido indica si la celda en cuestión ha sido destapada por el jugador o si permanece cubierta
- x – Contiene la posición en el eje de abscisas de cada una de las celdas. Esta información es calculada al inicio y se usa en el dibujo de las celdas a lo largo de todo el juego
- y – Contiene la posición en el eje de ordenadas de cada una de las celdas. Esta información es calculada al inicio y se usa en el dibujo de las celdas a lo largo de todo el juego

Al comenzar el juego, se crear el número de bombas adecuado y se calculan todos los valores de las celdas dentro de la variable *campo*.

Cuando el usuario mueve el cursor, se repinta la celda que estaba seleccionada hasta ese momento tal y como debe ser repitanda y se pinta la nueva celda seleccionada, para denotar su nuevo estado.

Al hacer *click* sobre una celda, en función del contenido de la misma, pueden ocurrir varias cosas:

- Si la celda en cuestión no oculta una bomba, se muestra su contenido. Si se destapa la última celda que no oculta una bomba, el juego finalizará, se mostrará un mensaje de felicitación y se almacenará la puntuación alcanzada
- Si la celda en cuestión oculta una bomba, el juego finalizará y se mostrará el contenido de todas las celdas y se almacenará la puntuación alcanzada

La puntuación que se almacena, será el número de celdas destapadas, y el número de bombas ocultas. Lo que se almacena no es un número, es una cadena de texto con la información recién descrita.

El siguiente listado muestra la clase Juego:

ISBN: 978-84-9012-861-9

Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
import javax.microedition.lcdui.*;

import java.util.Random;
import javax.microedition.rms.RecordStore; import javax.microedi-

tion.rms.RecordEnumeration;


public class Juego extends Canvas implements CommandListener {

    private Buscabombas buscabombas = null; private Menu menu =

    null;

    private boolean enPausa = false; private Command

    salir = null; private Command pausa = null; int mx =

    0;

    int my = 0;
    int lado = 0; int n = 0;


    // Contenido de cada una de las celdas int campo[][];

    int x[];

    int y[];
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
// Estado de cada una de las celdas (mostrada o no)

int estado[][];



int ladoCampo; Font f = null;

Font f2 = null;



int altoCanvas; int anchoCanvas;



int selecx = 0; int selecy = 0;

int antx = 0; int anty = 0; int

felx = 0; int fely = 0;


Image img; Graphics off;


private final static int BOMBA = -1; private final static int TAPADO = 0;

private final static int DESTAPADO = 1;
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
private boolean fin = false;

private int destapados=0; private int aDestapar = 0;

public Juego(Buscabombas buscabombas, Menu menu) { this.buscabombas = buscabom-
    bas;
    this.menu = menu;
    salir = new Command(Recursos.getString( Recursos.ID_SALIR), Com-
            mand.EXIT, 1);
    pausa = new Command(Recursos.getString( Recursos.ID_PAUSA),Command.SCREEN,1);
    this.addCommand(salir);
    this.addCommand(pausa); this.setCommandListener(this);

    altoCanvas = this.getHeight(); anchoCanvas =
    this.getWidth();

    img = Image.createImage(anchoCanvas,altoCanvas); off = img.getGraphics();

    f = Font.getFont(Font.FACE_MONOSPACE, Font.STYLE_PLAIN,Font.SIZE_SMALL);
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
f2 = Font.getFont(Font.FACE_MONOSPACE,
        Font.STYLE_BOLD,Font.SIZE_LARGE);

felx = (int)((anchoCanvas-f2.stringWidth( Recursos.getString( Recur-
        sos.ID_FELICITACION)))/2);
fely = (int)((anchoCanvas-f2.getHeight())/2);

  // Tamaño del lado de las celdas lado = 12;

  // Cálculo del número de celdas que podemos tener if (altoCanvas > anchoCanvas){
      n = (int)anchoCanvas/lado;
  }else{
      n = (int)altoCanvas/lado;
  }
  // Ancho del campo de celdas ladoCampo = n *
  lado;
  // Celdas a destapar para finalizar el juego aDestapar = (n*n)-Recursos.getNiveles()
        [buscabombas.getNivel()];
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241

Ediciones Universidad de Salamanca - cc by nc dc

```
mx = (int)(anchoCanvas-ladoCampo)/2;

my = (int)(altoCanvas-ladoCampo)/2;

x = new int[n]; y = new int[n];

campo = new int[n][n]; estado = new int[n][n];



for (int i=0; i<n; i++){ for (int j=0; j<n; j++){

        estado[i][j]=Juego.TAPADO; campo[i][j]=0;

  }

}

Random r = new Random(System.currentTimeMillis()); int j=0;

// Calculamos el contenido de las celdas

while (j < Recursos.getNiveles() [buscabombas.getNivel()]){

        int a = Math.abs(r.nextInt() % n); int b = Math.abs(r.nextInt()

        % n); if (campo[a][b] != Juego.BOMBA){

            campo[a][b] = Juego.BOMBA;
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                sumaBomba(a-1,b-1);

                sumaBomba(a-1,b); sumaBomba(a-1,b+1); suma-

                Bomba(a,b-1); sumaBomba(a,b); sumaBomba(a,b+1);

                sumaBomba(a+1,b-1); sumaBomba(a+1,b); suma-

                Bomba(a+1,b+1);

                j++;

            }

        }

        for (int i=0; i<n; i++){ x[i] = mx+(i*lado);

            y[i] = my+(i*lado);

        }

        pintaCampo();

    }

    protected void paint(Graphics g) {
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241

Ediciones Universidad de Salamanca - cc by nc dc

```
    // Dibujamos la imagen que hemos ido componiendo
    // en la pantalla g.drawImage(img,0,0,Graphics.LEFT|Graphics.TOP);
}
protected void keyPressed(int keyCode){ keysManager(keyCode);
        repaint();
        this.serviceRepaints();
}
private void keysManager(int keyCode){ if (destapados == aDesta-
    par){
        // Se ha finalizado la partida, al descubrir
        // todas las celdas sin bomba oculta menu.inicializar(buscabombas);
        buscabombas.mostrar(menu);
        return;
    }
    keyCode = getGameAction(keyCode); antx = selecx;
    anty = selecy;
    if (keyCode == Canvas.LEFT){
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        if (selecx > 0){
            selecx--;
        } else {
            if (selecy > 0){ selecx = n-1; selecy--;
            }
        }
        pintaCelda(antx,anty,0); pintaCelda(selecx,selecy,1);
    }
    if (keyCode == Canvas.RIGHT){ if (selecx == n-1){
            if (selecy < n-2){ selecx = 0; selecy++;
            }
        }else{
            selecx++;
        }
        pintaCelda(antx,anty,0); pintaCelda(selecx,selecy,1);
    }
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
if (keyCode == Canvas.UP){ if (selecy > 0){

        selecy--;

    }

    pintaCelda(antx,anty,0); pintaCelda(selecx,selecy,1);

}

if (keyCode == Canvas.DOWN){ if (selecy != n-1){

        selecy++;

    }

    pintaCelda(antx,anty,0); pintaCelda(selecx,selecy,1);

}

if (keyCode == Canvas.FIRE){

    if (campo[selecx][selecy] == Juego.BOMBA){

        // Al pinchar en una bomba, se acaba la partida finPartida();

    }else{

        if (estado[selecx][selecy] != Juego.DESTAPADO) destapados++;

    }
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
estado[selecx][selecy]=Juego.DESTAPADO;

if (campo[selecx][selecy] == 0){

    // Al pinchar en una celda "en blanco"

    // compruebo el entorno

    // para descubrir todas las celdas adyacentes

    // también en blanco. compruebaEntorno(selecx, selecy);

}

if (destapados == aDestapar){ finPartida();

    // Dibujo un mensaje de felicitación off.setColor(0xFFFFFF); off.fi-

    llRect(0,fely-5,anchoCanvas, 10+f2.getHeight()); off.setCo-

    lor(0x0000FF); off.setFont(f2); off.setColor(0x000099);

    // Dibujo el texto dos veces, desplazando un

    // poco una de ellas, para crear un

    // efecto de sombra off.drawString(Recursos.getString( Re-

    cursos.ID_FELICITACION),  felx+1, fely+1,
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                Graphics.LEFT | Graphics.TOP);

               off.drawString(Recursos.getString( Recursos.ID_FELICITACION),

                felx, fely, Graphics.LEFT | Graphics.TOP);

          }

     }

}
private void sumaBomba(int x, int y){

     if (x >= 0 && x < n && y >= 0 && y < n && campo[x][y] != Juego.BOMBA) {

                  campo[x][y]++;

     }

}
private void finPartida(){

     for (int i = 0; i < n; i++) { for (int j = 0; j < n; j++) {

               estado[i][j] = Juego.DESTAPADO; pintaCelda(i, j, 0);

          }

     }

     // Guardo la puntuación
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
    try{
        RecordStore rs = RecordStore.openRecordStore( "buscabombas", true);
        int i = 0;
        // Si hay más de 10 puntuaciones almacenadas
      .// se borra la más antigua de ellas if (rs.getNumRecords() > 10) {
            RecordEnumeration re = rs.enumerateRecords(
              null,null,false);
            if (re.hasNextElement()){ rs.deleteRecord(re.nextRecordId());
            }
        }
        String a = destapados + "(" + Recursos.getNiveles() [busca-
              bombas.getNivel()] + ")";
        rs.addRecord(a.getBytes(),0,a.length());
    }catch (Exception ex){ ex.printStackTrace();
    }
}
public void commandAction(Command cmd, Displayable dis) {
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
    if (cmd == salir) {
        menu.inicializar(buscabombas); buscabombas.mostrar(menu);
    }
    if (cmd == pausa){
        // Ponemos el juego en modo de pausa enPausa = true; menu.iniciali-
        zar(buscabombas); buscabombas.mostrar(menu);
    }
}
private void pintaCampo(){ off.setColor(0xFFFFFF); off.fillRect(0,0,anchoCan-
    vas,altoCanvas); off.setColor(0x000000);
    off.setFont(f);

    off.drawRect(mx,my,ladoCampo,ladoCampo);


    int myl = my+ladoCampo; int mxl = mx+lado-
    Campo; for (int i=0; i<n; i++){
        off.drawLine(x[i], my, x[i], myl);
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        off.drawLine(mx, y[i], mxl, y[i]);

    }

    int l2 = lado-4;

    for (int i=0; i<n; i++){ for (int j=0; j<n; j++){

            if (estado[i][j] == Juego.TAPADO){ off.setColor(0x888888); off.fillRect(x[i]+2,y[j]+2,l2,l2);

            }else{

                off.setColor(0x000000); off.drawString(""+campo[i][j],x[i]+2,y[j]-1,

             Graphics.LEFT|Graphics.TOP);

            }

        }

    }

    off.setColor(0x000000); off.fillRect(x[selecx],y[selecy],lado,lado);

}

private void pintaCelda(int i, int j, int selec){ int l2 = lado-4;

    off.setColor(0xFFFFFF);

    off.fillRect(x[i]+1, y[j]+1, lado-1, lado-1);
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
if (selec == 0){
    if (estado[i][j] == Juego.TAPADO) { off.setColor(0x888888);  off.fill-

        Rect(x[i] + 2, y[j] + 2, l2, l2);

    }

    else {

        if (campo[i][j] != 0){

            if (campo[i][j] != Juego.BOMBA){ off.setColor(0x000000);

                off.drawString("" + campo[i][j],

                    x[i] + 2, y[j] - 1,

                    Graphics.LEFT | Graphics.TOP);

            }else{

                // Dibujo la bomba off.setColor(0x000000); off.fi-

                llArc(x[i]+2,y[j]+3,

                    lado-3,lado-3,0,360); off.drawLine(x[i]+6,y[j]+2,x[i]+6,y[j]+1);

                off.drawLine(x[i]+6,y[j]+1,x[i]+8,y[j]+1);

            }

        }else{

            off.setColor(0xFFFFFF); off.fillRect(x[i]+1,y[j]+1,lado-1,lado-1);

        }
```

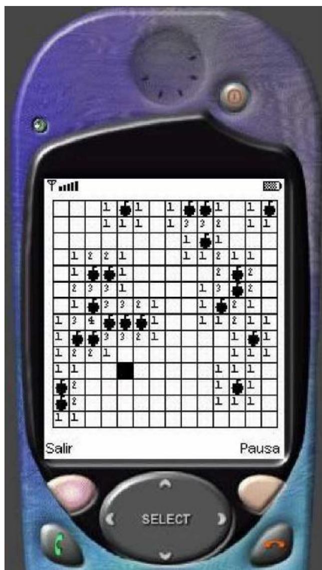

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        }
      } else { off.setColor(0x000000);
        off.fillRect(x[i],y[j],lado,lado);
      }
  }
  public void continuar() { enPausa = false;
  }
  public boolean enPausa() { return enPausa;
  }
  private void compruebaEntorno (int i, int j){ int l2 = lado-4;
      for (int a = i - 1; a <= i + 1; a++) {
        for (int b = j - 1; b <= j + 1; b++) {
          // La celda actual no se procesa if (a != i || b != j){
            // Si estoy fuera de los límites del campo,
            // no hago nada
            if (a < n && a >= 0 && b < n && b >= 0 &&
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                estado[a][b] == Juego.TAPADO &&
                campo[a][b] == 0) { estado[a][b] =
              Juego.DESTAPADO; off.setColor(0x888888);
              off.fillRect(x[a] + 2, y[b] + 2, l2, l2); pintaCelda(a,b,0);
              compruebaEntorno(a, b);
            }
          }
          }
        }
      }
  }
```

A continuación tenemos algunas pantallas del juego en diferentes terminales, para mostrar el resultado final de juego.



ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

3 JUEGOS DINÁMICOS

3.1 INTRODUCCIÓN

En este capítulo vamos a estudiar las técnicas básicas para el desarrollo de lo que denominaremos juegos dinámicos. Al contrario de los juegos estáticos vistos anteriormente, los juegos dinámicos se caracterizan por una presencia masiva de elementos móviles. Es decir son juegos en los que tendremos una serie de elementos (denominados *sprites* de forma general, sean móviles o no.) que se moverán por la pantalla del juego a lo largo de la ejecución del mismo. El movimiento de los *sprites*, conlleva consigo una serie de cuestiones a tener en cuenta como es la detección de colisiones entre *sprites* y la gestión de dichas colisiones [31-35].

A lo largo del capítulo veremos el desarrollo de un juego J2ME correspondiente a lo que podríamos denominar como un simulador de un minigolf. El desarrollo de este juego nos va a permitir explicar cómo se realizan los procedimientos básicos en el desarrollo de juegos J2ME relativamente complejos, de tal forma que podamos hacer, entre otras cosas y sin recurrir a las facilidades de MIDP 2.0 las siguientes operaciones:

- Gestión mejorada de la carga y distribución de imágenes (*sprites*)
- Gestión de la transparencia de los *sprites*
- Detección de colisiones entre *sprites* y gestión de las mismas
- Uso de *threads* o hilos de ejecución para hacer animaciones

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

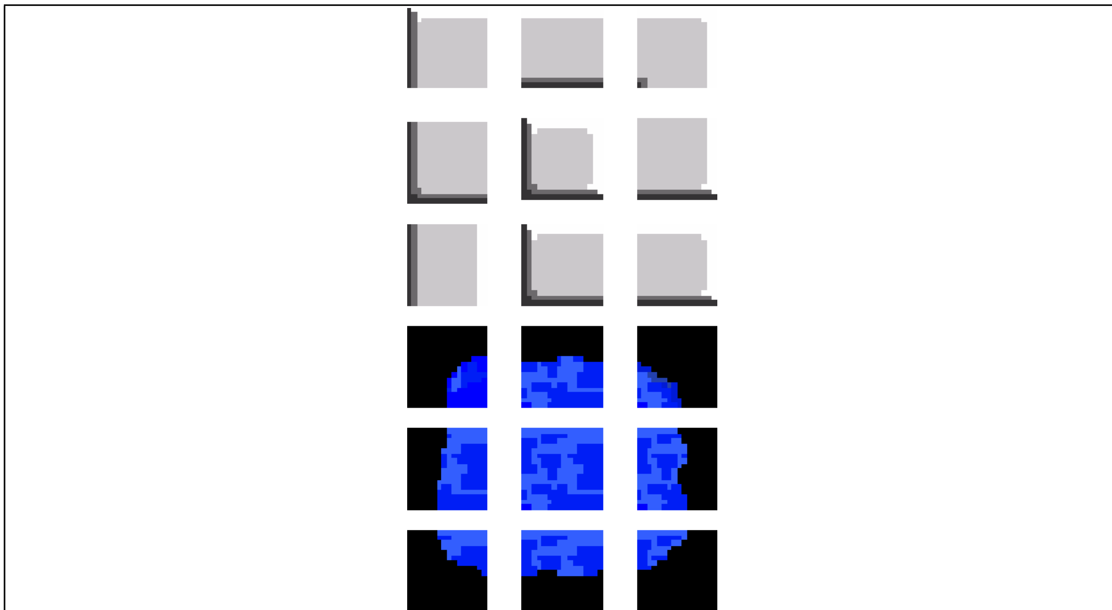
•

3.2 ALMACENAMIENTO Y DISTRIBUCIÓN DE LOS SPRITES

Como sabemos, en el desarrollo de *midlets* J2ME es básico controlar el tamaño final del fichero *jar* que debemos construir, ya que este fichero se ha de descargar posteriormente por red desde un terminal y además, algunos terminales presentan restricciones en cuanto al tamaño de los ficheros *jar* que son capaces de manejar. Uno de los elementos que se incluyen en este fichero y que suelen aumentar su tamaño preocupantemente, son las imágenes. Si bien el título de este juego hace referencia a *sprites*, en realidad vamos a ver cómo tratar las imágenes que componen la parte esencial de dichos *sprites [36]*.

En un primer momento lo que tendremos será una serie de imágenes con el formato *png*. Todas estas imágenes, estarán almacenadas en ficheros separados, y cada uno de estos ficheros contendrá la cabecera característica de los ficheros gráficos con formato *png*. Bien, la solución propuesta para ahorrar espacio, es la inclusión de todas las imágenes dentro de un único fichero. De esta forma tendremos una única cabecera y por lo tanto el fichero *jar* resultante será menor.

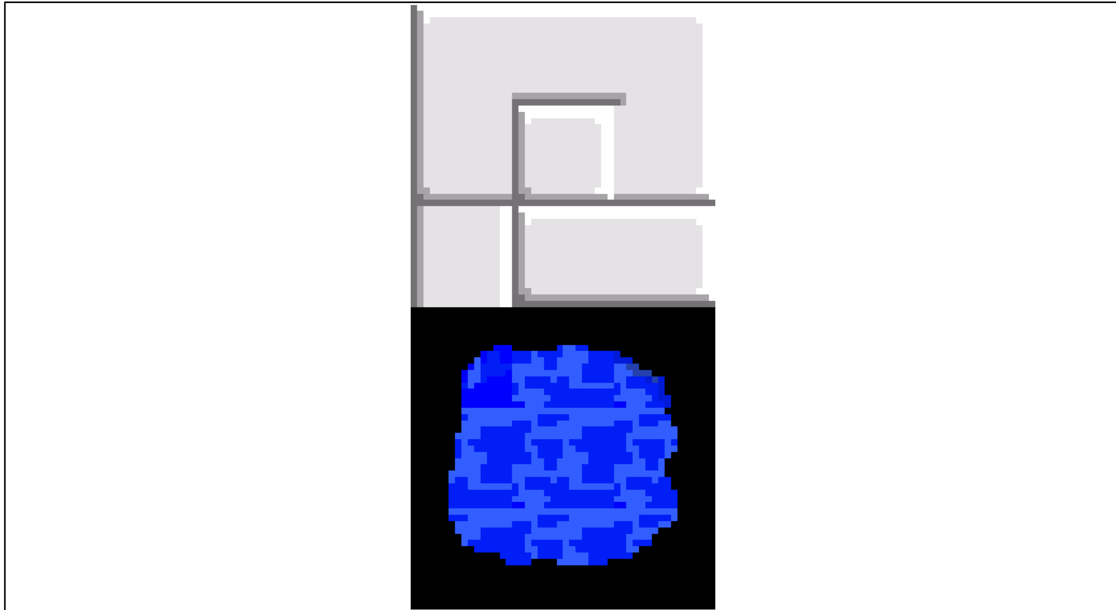
En nuestro caso, tendremos una serie de imágenes que corresponden a los elementos simples con los que compondremos el campo de minigolf. La siguiente figura muestra los *sprites* que se usan en el juego para componer los elementos que componen el campo de juego. En concreto tendremos bloques que nos permiten componer obstáculos en el campo y tendremos una serie de *sprites* para situar zonas de agua dentro del campo.



ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Como vemos, tenemos 18 imágenes independientes que en total suponen una carga de almacenamiento de 6240 bytes (algo más de 6 Kb).

El fichero con todas estas imágenes unidas, ocupa 1264 bytes (algo más de 1 Kb), por lo que el ahorro de espacio es de 5 Kb, lo que supone un ahorro de espacio del 80%. Como vemos, esta técnica es rentable en lo que a espacio se refiere y por lo tanto, nuestro objetivo ha sido cumplido. Un punto a tener en cuenta es que los *sprites* (en cualquier juego y de forma muy especial en los juegos móviles) se configuran a partir de imágenes de un tamaño relativamente pequeño, y por lo tanto la cabecera correspondiente al formato de imagen del fichero tiene un impacto importante sobre el tamaño final del fichero [37]. La imagen que contiene todas las imágenes es la siguiente:



Por supuesto, el uso de esta solución conlleva un cierto trabajo extra, tanto en tiempo de desarrollo (tendremos que unir todas las imágenes en una) como en tiempo de ejecución, ya que tendremos que dividir la imagen completa en sus componentes dentro del código del *midlet*. Si hacemos esta operación durante el periodo de inicialización del mismo, el usuario no se percatará de dicho tiempo extra y por lo tanto la solución será perfecta [38].

3.2.1 EL PROCESO DE SEPARACIÓN DE LAS IMÁGENES

Para separar las imágenes, dentro de la solución propuesta en este caso, se ha creado una clase denominada *SpritesManager,* que se encargará de cargar la imagen que contiene el conjunto de los *sprites* y extraer de la misma todas las imágenes que contiene. Estas imágenes se usarán

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

más adelante en el proceso de creación de objetos de la clase *Sprite*, de tal forma que estos objetos contengan tanto la imagen del *sprite*, así como otra información adicional, útil para la gestión de los *sprites*.

Bien, el código de la clase *SpritesManager* es el siguiente:

```
import java.io.IOException;
import javax.microedition.lcdui.Image; import javax.microedi-

tion.lcdui.Graphics;

public class SpritesManager {
    private Image[] imagenes = new Image[18];

    public SpritesManager() {

      Image img = null; try{

          img = Image.createImage("/Sprites.png");

      }catch (IOException ex){ ex.printStack-

          Trace();

      }

      int ind = 0;

      for (int i = 0; i < 3; i++) { for (int j = 0; j < 6; j++) {

              imagenes[ind] = Image.createImage(16, 16); Graphics g =

              imagenes[ind].getGraphics(); g.drawImage(img, i*-16, j*-16,

              Graphics.TOP | Graphics.LEFT);

              ind++;

          }

      }
   }
   public Image getImage (int i){ return imagenes[i];
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241

Ediciones Universidad de Salamanca - cc by nc dc

```
    }
  }
```

En nuestro caso concreto, todas las imágenes tienen las mismas dimensiones, cuestión que simplifica bastante el proceso. De hecho, esta restricción la presenta también la solución que incluye la especificación 2.0 de MIDP para hacer exactamente lo mismo, es decir, almacenar los *sprites* en un único fichero y separar estas imágenes en tiempo de ejecución [39-43].

Siguiente dentro del ámbito de nuestro caso concreto, nuestras imágenes tienen todas 16 *pixels* de ancho por 16 *pixels* de alto. Como veíamos anteriormente, tenemos un total de 18 imágenes y ese por lo tanto la clase *SpritesManager* deberá ser capaz de manejar exactamente ese número de imágenes. Una vez carga la imagen principal desde el fichero con todas las imágenes destinadas a los *sprites* incluidas dentro de ella, dibujaremos sobre cado uno de los objetos *Image* que contiene la clase *SpritesManager,* que posteriormente será un *sprite*, sólo aquella parte de la imagen completa que corresponde al *sprite* en cuestión. Para conseguir esto, lo único que hacemos es situar el punto de dibujo de la imagen completa en el lugar adecuado.

Una vez que tenemos separadas todas las imágenes ya podemos recuperarlas a través del método *getImage*, que veremos cómo se utiliza más tarde para construir los *sprites*.

3.3 LA CLASE SPRITE

La clase *Sprite* es básicamente una abstracción de una imagen, que además nos permite controlar la situación o posición en la que debe dibujarse el *Sprite* y comprobar las colisiones con otros *Sprites*. Además, dentro de la clase *Sprite* veremos cómo pintar imágenes con transparencias. Esta funcionalidad no la proporcionan las primitivas de dibujo del objeto *Graphics* dentro de la especificación 1.0 de MIDP y es de suma importancia en el desarrollo de juegos. Como vemos, la clase *Sprite* contiene algunas de las características imprescindibles para el desarrollo de juegos. Más adelante, veremos cómo utilizar esta clase dentro de nuestro juego para obtener resultados satisfactorios [44].

Para empezar, vamos a ver cómo gestiona la clase *Sprite* la imagen que debe mostrar y la posición en la que debe dibujar la misma. Los parámetros que contiene la clase *Sprite* son:

```
private int x = 0;

private int y = 0; private Image img; pri-

vate int imgId; private int mascara[][];
```

Las dos primeras variables corresponden a la posición de la esquina superior izquierda de la imagen base del *sprite* en la pantalla. Debemos tener en cuenta, que es posible que con la transparencia, lo que realmente se vea en la pantalla del terminal no comience en este punto. Este punto definirá el marco que envuelve toda la imagen. Evidentemente, la variable *img* almacena la imagen que contiene y gestiona el *sprite*. La variable *imgId* simplemente nos permite

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

identificar la imagen asociada al *sprite*, lo que nos permitirá controlar ciertas situaciones. Es decir, sabiendo que tenemos un conjunto limitado de imágenes, en nuestro caso 18, que pueden componer un *sprite*, podemos asignar un número a cada una de las imágenes y así poder hacer alguna gestión alternativa. En el caso de que varios *sprites*

alberguen la misma imagen, todos tendrán el mismo identificador. En el caso concreto de nuestro minigolf, usamos este valor para detectar si el *sprite* contiene una imagen correspondiente a una zona de agua, o una imagen correspondiente a un obstáculo. Por último, la variable máscara define una serie de rectángulos, que serán las zonas de dibujo de la imagen que nos permite realizar y gestionar las transparencias. Esta variable la veremos en detalle un poco más adelante [45-50].

Para que la clase *Sprite* sea más flexible, se le han proporcionado tres posibles constructores que son similares, pero que difieren en la forma de obtener la imagen del *sprite*. Los constructores son los siguientes:

```
public Sprite(Image img, int mascara[][],

          int x, int y, int imgId) public Sprite(int mascara[][],

int x,

          int y, int imgId)

public Sprite(String fichero, int mascara[][], int x, int y, int imgId)
```

El primer constructor que se muestra, tiene como parámetros un objeto de tipo *Image*, una matriz de dos dimensiones que será la máscara de transparencia, el punto que define la posición inicial del *sprite* (a través de sus valores x e y) y por último, un parámetro que especifica el tipo de *sprite*, tal y cómo explicamos antes. El segundo de los constructores, no tiene imagen. En principio esto puede parecer un poco absurdo, pero puede ser útil para definir zonas, en principio invisibles, pero que nos van a permitir controlar las colisiones de otros *sprites* contra estas zonas invisibles.

El último constructor es capaz de obtener la imagen del *sprite* a partir del fichero cuyo nombre recibe como primer parámetro.

3.3.1 TRANSPARENCIAS EN SPRITES

El uso de transparencia en los *sprites* es algo básico si queremos que un juego de los que hemos denominado dinámicos sea realmente profesional y vistoso. En el caso de nuestro

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

campo de golf, hemos visto anteriormente que tenemos una serie de *sprites* que definen zonas de agua, en los que el fondo de la imagen es negro y debemos tener en cuenta que nuestro campo de golf será verde. Las siguientes imágenes muestran la diferencia entre el uso de un *sprite* con transparencia y un *sprite* sin transparencia.

Como se puede comprobar, la solución realizada sin transparencia es poco útil y nuestros *sprites* serían bastante pobres sin transparencia. En nuestro caso, usando la transparencia, la zona de agua nos permite ver el campo de golf.

Bien, una vez que vemos que las transparencias en los *sprites* son imprescindibles para desarrollar un juego y que la especificación 1.0 de MIDP no nos proporciona una implementación de la gestión de imágenes transparentes, tendremos que desarrollar nuestra propia solución. Tenemos varias opciones para implementar transparencias:

Dibujar los *sprites* a partir de las primitivas que nos proporciona la clase *Graphics*, como por ejemplo *drawLine*. Esta solución presenta dos problemas graves, que la hacen viable sólo en casos muy concretos. Por una parte es una solución con un rendimiento muy bajo en tiempo de ejecución y por supuesto, realizar *sprites* de calidad (imágenes complejas) a partir de primitivas básicas es un trabajo ímprobo y complicado.

Dibujar la imagen compleja a partir de imágenes menores que componen la primera. Es decir, partiendo de la imagen real del *sprite* que queremos usar, la dividimos en tantas imágenes como sea necesario para que al pintar todas estas imágenes por separado, consigamos la imagen original con transparencia. Esta solución también es en cierta medida compleja y cuando las imágenes tienen muchas zonas redondeadas, la solución puede llevarnos a manejar un gran número de imágenes.

Por último, podemos implementar una solución en la que definamos una serie de zonas de

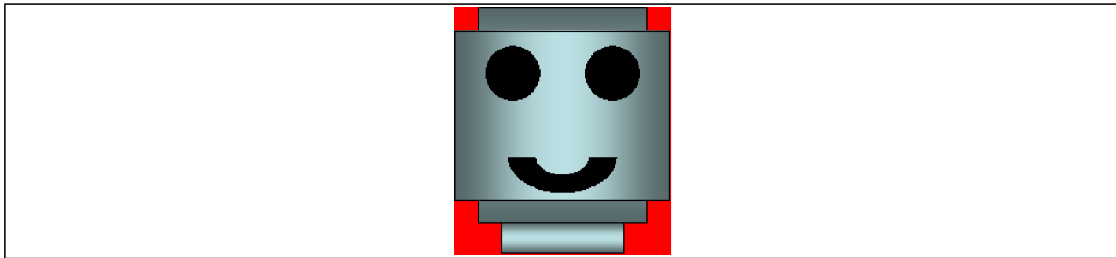
ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

dibujo rectangulares (también se podrían definir circulares) y utilizar el método *setClip* de la clase *Graphics* para que al dibujar la imagen, dicha acción sólo tenga efecto sobre la pantalla en las zonas definidas. Esta solución es un buen compromiso entre dificultad de implementación, necesidad de recursos en tiempo de ejecución y el resultado obtenido y por lo tanto, se ha optado por ella para implementar nuestro juego de ejemplo.

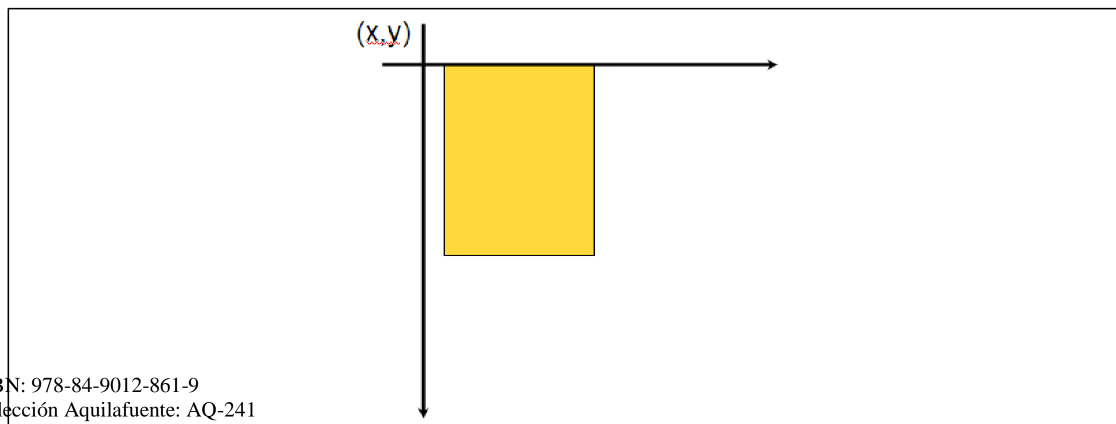
Ahora que ya tenemos una idea general de cómo implementar la transparencia de los *sprites*, vamos a desarrollar esta técnica en detalle. Debemos destacar que el hecho de que las imágenes para dispositivos móviles sean típicamente pequeñas y *pixeladas*, favorece en gran medida esta solución.

Ya hemos hablado de la variable máscara, que será una matriz de dos dimensiones de enteros. Esta variable lo que contiene es la definición de rectángulos, que compondrán aquellas zonas de la imagen que deben ser vistas. Con el siguiente ejemplo gráfico vamos a ver claramente este concepto.

Supongamos que queremos dibujar este simpático robot y la zona que en la imagen es de color rojo, debe ser transparente y debemos ver a través de ella.



La imagen anterior será la imagen *png* que tenemos almacenada dentro de la variable *img* en el objeto *Sprite* en cuestión. Para empezar, definiremos las zonas no transparentes de la imagen, es decir, lo que hemos denominado máscara. Para definir uno de estos rectángulos, lo que hacemos es configurar el punto en el que se sitúa la esquina superior-izquierda del rectángulo, y definimos su ancho y su alto. Por ejemplo, la siguiente imagen muestra un rectángulo que define una zona no transparente de la imagen.

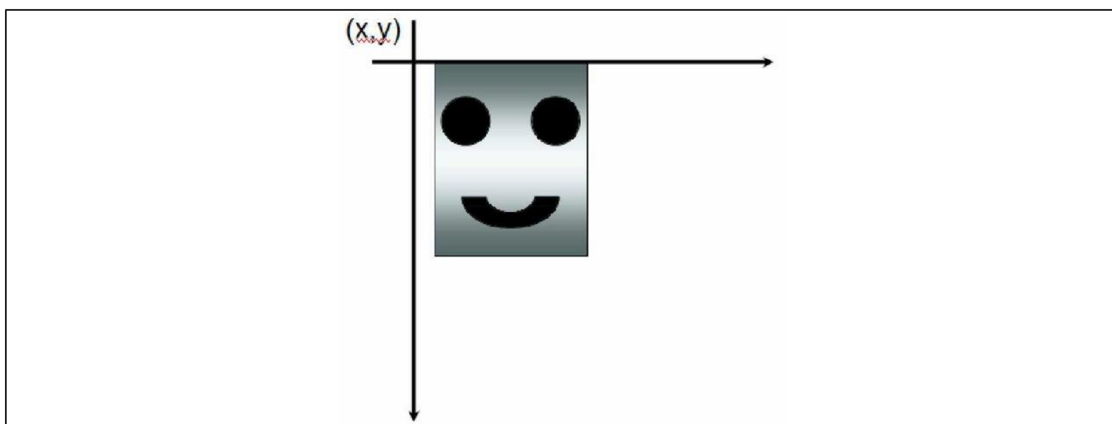


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

El rectángulo amarillo define una zona no transparente de la imagen, por lo que tendremos una primera versión de la variable máscara como:

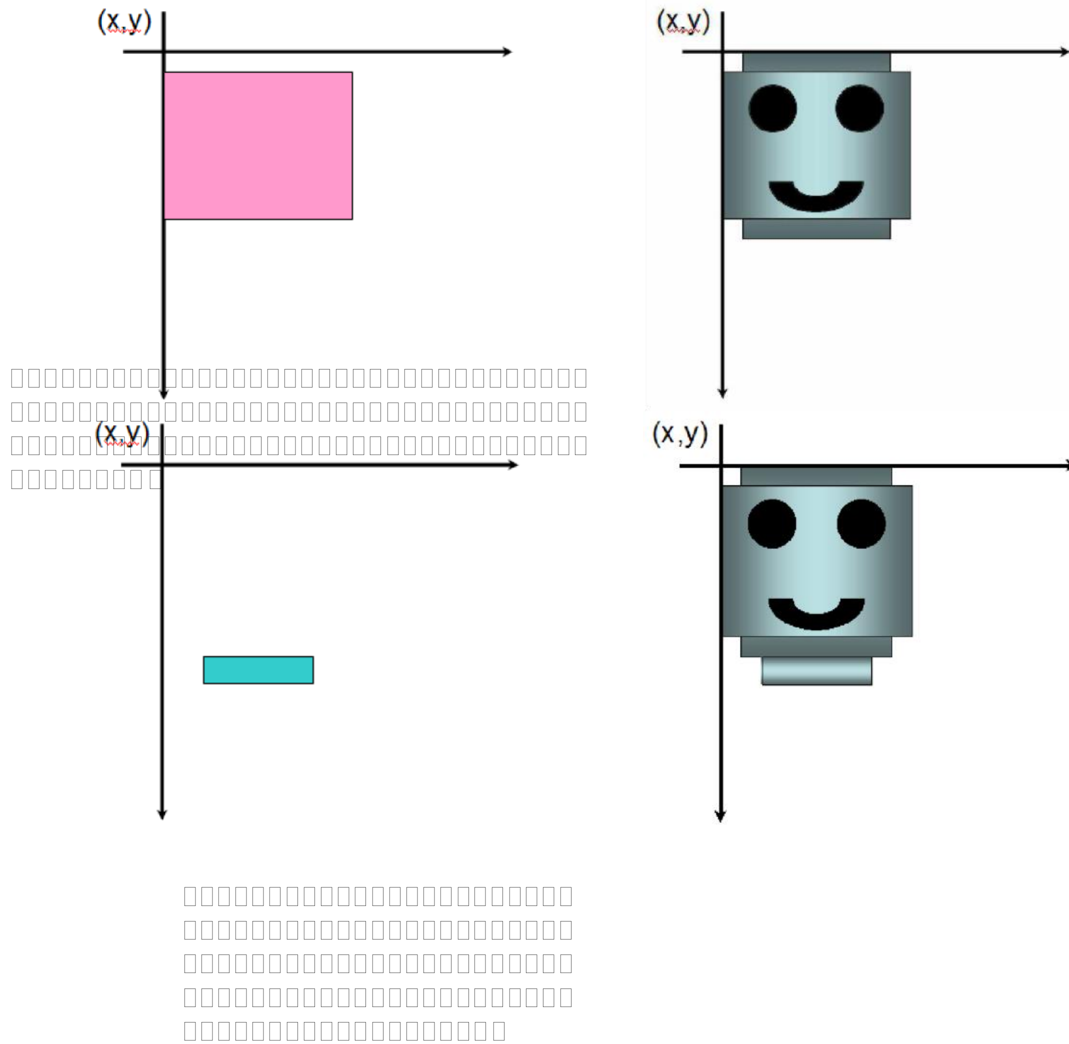
```
mascara = {{x+1,y,5,8}};
```

En el código anterior, 5 es el ancho del rectángulo amarillo y 8 su altura. Si ahora definimos esta zona como aquella única parte de la pantalla en la que tendrán efecto las operaciones de dibujo y dibujamos la imagen en el punto *(x,y)*, el resultado será el siguiente:



ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Si definimos alguna zona más y repetimos la operación, es decir, establecemos un rectángulo como zona visible a través de *setClip* y volvemos dibujar la imagen. El resultado sería el siguiente.



De esta forma, tendremos definitivamente en la pantalla toda la imagen del *sprite* y será transparente en aquellas zonas no presentes en algunos de los rectángulos de la máscara. El código final para definir la máscara sería algo como lo siguiente.

```
mascara = {{x+1,y,5,8}{x,y+1,8,5}{x+2, y+8,4,1}};
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Por supuesto, estos valores son aproximados, pero nos da una idea de cómo sería el funcionamiento. Ahora que ya tenemos claro cómo es el proceso, vamos a ver el código real que debemos escribir para que esto funcione.

Cuando una *sprite* no tiene zonas transparentes, es decir, las imágenes se deben dibujar enteras, en nuestra implementación, el valor de la máscara será *null*. Típicamente no tendremos transparencias cuando nuestro *sprite* sea cuadrado o rectangular, como lo son los obstáculos de nuestro campo de golf [51].

En nuestra clase *Sprite* hemos definido un método *paint* que recibe como parámetro un objeto *Graphics* y que dibuja la imagen del *sprite* en cuestión sobre dicho objeto *Graphics*. Este método será invocado cada vez que se quiera dibujar el *sprite*. El código del método *paint* es el siguiente.

```
public void paint (Graphics g){

    int ancho = g.getClipWidth(); int alto =

    g.getClipHeight();
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
if (mascara == null){
    g.drawImage(img, x, y, Graphics.LEFT |
        Graphics.TOP);
  }else{
    for (int i=0; i<mascara.length;i++){ g.setClip(x+mas-
      cara[i][0],y+mascara[i][1], mascara[i][2],mascara[i][3]);
      g.drawImage(img, x, y,
        Graphics.LEFT | Graphics.TOP);
    }
    g.setClip(0,0,ancho,alto);
}
    }
```

Como podemos comprobar, si la máscara tiene valor *null* se dibuja la imagen tal cual en el punto definido como posición del *sprite* y no hacemos nada más. Por el contrario, si tenemos zonas definidas en la variable máscara, para cada una de estas zonas (que es un rectángulo), definimos una zona de recorte con *setClip* igual a dicha zona y dibujamos la imagen en su posición. Al final del proceso, reestablezco la zona de recorte o de *clipping* a toda la pantalla, que es similar a no definir ninguna zona de recorte. De esta manera, tendremos implementado un sistema de transparencia de *sprites*.

3.3.2 DETECCIÓN DE COLISIONES ENTRE SPRITES

Otro punto esencial dentro de la programación de juegos es la detección de colisiones entre *sprites*. Es decir, debemos detectar cuando dos *sprites* chocan o colisionan en la pantalla debido a su movimiento. También es posible que sólo uno de los *sprites* esté en movimiento y choque contra otro *sprite* estático. La acción a realizar cuando dos *sprites* chocan dependerá de

ISBN: 978-84-9012-861-9

Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

cada juego en concreto. Por ejemplo, si un *sprite* que simula un disparo choca contra un personaje del juego, quizás debamos eliminar del juego el *sprite* del personaje [52]. En nuestro caso concreto, el simulador de minigolf tendrá colisiones entre la pelota y los obstáculos, las zonas de agua y el hoyo. En cada caso, deberemos saber contra qué elemento exactamente estamos colisionando, ya que el resultado de la colisión será distinto. Si la bola choca con un obstáculo, rebotará en el mismo y cambiará la trayectoria de su movimiento. Por el contrario si la bola choca con una zona de agua, la bola se pierde y vuelve a la posición que tenía antes de ser golpeada. Por último, si la colisión es entre el *sprite* que hace las funciones de bola y el que hace las funciones de hoyo, habremos conseguido nuestro objetivo y pasaremos al siguiente hoyo.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

La funcionalidad correspondiente a la detección de colisiones también recae sobre la clase *Sprite*. En concreto, la clase *sprite* tiene los métodos necesarios para realizar todo el proceso. Estos métodos son:

```
public int checkCollision(Sprite sprite);

public int[][] getMascaraColision();
```

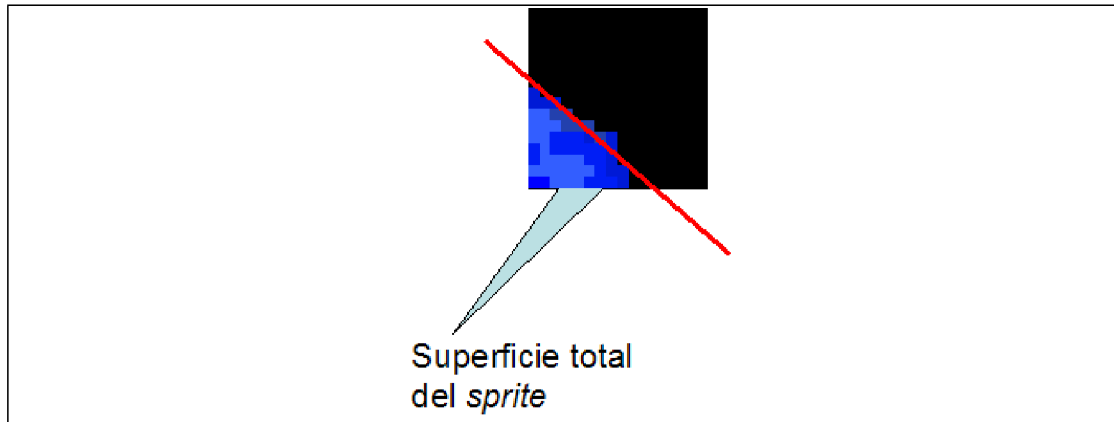
El primer método recibe como parámetro otro *sprite* y comprueba si el *sprite* en cuestión al que pertenece el método está colisionando con el *sprite* que se recibe como parámetro. El método devuelve un valor entero que en nuestro caso indica contra qué zona del *sprite* se ha chocado. Esto es especialmente útil cuando la colisión es contra un objeto cuadrado o rectangular como ocurre en alguno de nuestros casos. Otra opción sería que este método retornará simplemente *true* o *false* para indicar si hay o no colisión. En nuestra implementación, si el valor de retorno es –1, podemos asumir que no hay colisión entre los dos *sprites [53]*.

El segundo método, el método *getMascaraColision*, nos retorna una matriz bidimensional de enteros, que representa las zonas del *sprite* que corresponden con las zonas de colisión. El concepto es similar al usado para las transparencias. De hecho, en nuestra implementación, la zona no transparente del *sprite* es exactamente la zona de colisión del mismo.

Se definen zonas de colisión, por la misma razón por la que tenemos zonas transparentes y zonas no transparentes en los *sprites*. En nuestros *sprites* para las zonas de agua, existen zonas circulares. Debemos realizar la funcionalidad de tal forma que si la bola en su movimiento entra dentro de lo que es el *sprite* de agua pero no toca la zona de agua propiamente dicha del *sprite* (zona no transparente), no debemos detectar ninguna colisión. Si no lo hiciéramos de esta forma, el usuario obtendría un funcionamiento no esperado del sistema, ya que no refleja lo que ocurre en el mundo real. La siguiente imagen aclara lo que sería una zona de colisión dentro de un *sprite*.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241

Ediciones Universidad de Salamanca - cc by nc dc



Dentro de la imagen anterior, sólo deberíamos de detectar una colisión cuando esta ocurra con la zona a la izquierda de la línea roja. A la derecha de dicha línea, el *sprite* es transparente y por lo tanto, esa zona del *sprite* no existe para el usuario [54].

Como decíamos anteriormente, se definen zonas de colisión dentro del *sprite* para saber cuál es la zona inexistente para el usuario. Las zonas de colisión se definen mediante zonas rectangulares, de tal forma que la suma de las superficies de todos estos rectángulos es la superficie de colisión del *sprite*. En nuestro caso concreto y en la mayoría de los casos, la zona de colisión será la misma zona que la zona no transparente del *sprite*. Este hecho nos permite que la variable *mascara* de la clase *Sprite* de la que hemos hablado anteriormente, nos sirva para ambos propósitos, tanto para realizar las transparencias como para detectar las colisiones.

En este punto, es sencillo llegar a la conclusión de que podemos comprobar si dos *sprites* colisionan, comprobando si sus zonas de colisión se solapan. Para realizar esto, habrá que comprobar si alguno de los rectángulos que definen la zona de colisión de un *sprite*, tiene alguna superposición con alguno de los rectángulos de colisión del otro *sprite*.

El siguiente código muestra el contenido del método *getMascaraColision*.

```
public int[][] getMascaraColision(){

    int masc[][];

    if (mascara != null){

        masc = new int[mascara.length][4];
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        for (int i = 0; i < masc.length; i++) { masc[i][0] = mas-

            cara[i][0] + this.getX();

            masc[i][1] = mascara[i][1] + this.getY();
            masc[i][2] = mascara[i][2];
            masc[i][3] = mascara[i][3];
        }
    }else{
        masc = new int[1][4]; masc[0][0] = x;

        masc[0][1] = y;
        masc[0][2] = img.getWidth(); masc[0][3] =

        img.getHeight();

    }

    return masc;

}
```

Como podemos comprobar en el código anterior y como era de esperar, si la máscara de colisión es *null* (debemos recordar que esto indicaba que el *sprite* no tenía zonas transparentes) toda la superficie del *sprite* es objeto de colisión. Por lo tanto, si la variable *mascara* es *null*, se retornará un rectángulo igual a toda la superficie del *sprite*. Por el contrario,

si la máscara de colisión no es *null*, se retornarán aquellas zonas de la pantalla que son sensibles a colisiones debido al *sprite*. Este hecho es importante, ya que la zona de colisión real se calcula en cada momento, en función de la posición del *sprite*. Esto se hace gracias a las siguientes líneas de código del método *getMascaraColision*.

```
masc[i][0] = mascara[i][0] + this.getX();

masc[i][1] = mascara[i][1] + this.getY();
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

El método *checkColision* es, como ya hemos dicho, el que realiza realmente la detección de las colisiones entre *sprites*. El siguiente código corresponde a dicho método.

```
public int checkCollision(Sprite sprite){

    int sprt1[][] = this.getMascaraColision(); int sprt2[][] =

    sprite.getMascaraColision();


    for (int i=0; i<sprt1.length; i++){ for (int j=0;

        j<sprt2.length; j++){

            boolean colision = false;
            if (sprt1[i][0]+sprt1[i][2] == sprt2[j][0] &&
                    ((sprt1[i][1] >= sprt2[j][1] &&
                    sprt1[i][1] <= sprt2[j][1] + sprt2[j][3])
                        ||
                (sprt1[i][1]+sprt1[i][3] >= sprt2[j][1] && sprt1[i][1]+sprt1[i][3] <=

                            sprt2[j][1] + sprt2[j][3]))){
                //colision contra la parte izquierda del

                //sprite colision = true;

                return CalculadorTrayectoria.DERECHA;
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
}
if (sprt1[i][0] == sprt2[j][0] + sprt2[j][2] &&
        ((sprt1[i][1] >= sprt2[j][1] &&
        sprt1[i][1] <= sprt2[j][1] + sprt2[j][3])
              ||
      (sprt1[i][1]+sprt1[i][3] >= sprt2[j][1] && sprt1[i][1]+sprt1[i][3] <=
              sprt2[j][1] + sprt2[j][3]))){
    //colision contra la parte derecha del
    //sprite colision = true;
    return CalculadorTrayectoria.IZQUIERDA;
}
if (sprt1[i][1] == sprt2[j][1] + sprt2[j][3] &&
        ((sprt1[i][0] >= sprt2[j][0] &&
        sprt1[i][0] <= sprt2[j][0] + sprt2[j][2])
              ||
      (sprt1[i][0]+sprt1[i][2] >= sprt2[j][0] && sprt1[i][0]+sprt1[i][2] <=
              sprt2[j][0] + sprt2[j][2]))){
    //colision contra la parte abajo del
    //sprite
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                colision = true;

                return CalculadorTrayectoria.ARRIBA;

            }



            if (sprt1[i][1]+sprt1[i][3] == sprt2[j][1] &&

                    ((sprt1[i][0] >= sprt2[j][0] &&

                    sprt1[i][0] <= sprt2[j][0] + sprt2[j][2])

                        ||

                  (sprt1[i][0]+sprt1[i][2] >= sprt2[j][0] && sprt1[i][0]+sprt1[i][2] <=

                        sprt2[j][0] + sprt2[j][2]))){

                //colision contra la parte arriba del

                //sprite colision = true;

                return CalculadorTrayectoria.ABAJO;

            }

        }

    }

    // No hay colisión return -1;

}
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

En el código anterior aparecen una serie de constantes pertenecientes a la clase *CacularTrayectoria*, que veremos más adelante. El código anterior hace exactamente lo que hemos comentado, cada rectángulo perteneciente a la zona de colisión de un *sprite*, se compara con todos los rectángulos de la zona de colisión del otro *sprite* para detectar si coinciden en algún punto de su superficie [55].

En una versión sencilla de un detector de colisiones, bastaría con esto, pero en nuestro caso, además necesitamos saber con qué zona del *sprite* estamos colisionando (arriba, abajo, derecha o izquierda). Para hacer esto, detectamos con qué zona del rectángulo de colisión del *sprite* que se recibe como parámetro, está en contacto con nuestra zona de colisión. Por último, vemos que cuando no se detecta colisión alguna, se retorna el valor –1.

Hasta este punto, tenemos una descripción detallada de la clase *Sprite*, que contiene algunas características muy importantes. A continuación tenemos un listado completo del código de dicha clase, que será utilizada masivamente en el resto del código del juego.

```
import javax.microedition.lcdui.*;

import java.io.IOException;

public class Sprite { private int x = 0;

    private int y = 0; private Image

    img; private int imgId;


    private int mascara[][];

    public Sprite(Image img, int mascara[][],
                          int x, int y, int imgId) { this.mascara = mascara;

        this.x = x;

        this.y = y; this.img = img;

        this.imgId = imgId;

    }
```

ISBN: 978-84-9012-861-9

Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
    public Sprite(int mascara[][], int x,
                        int y, int imgId) { this.mas-
        cara = mascara; this.x = x;
        this.y = y; this.imgId = imgId;
    }

    public Sprite(String fichero, int mascara[][], int x, int y, int imgId) {
        this.mascara = mascara; this.x = x;
        this.y = y; this.imgId = imgId;
        try{
            this.img = Image.createImage(fichero);
        }catch (Exception ex){}
}

    public void moveTo (int x, int y){ this.x = x;
        this.y = y;
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
}
public void paint (Graphics g){ int ancho =

   g.getClipWidth();        int        alto        =

   g.getClipHeight(); if (mascara == null){

      g.drawImage(img, x, y, Graphics.LEFT |

          Graphics.TOP);

   }else{

      for (int i=0; i<mascara.length;i++){ g.setClip(x+mas-

         cara[i][0],y+mascara[i][1], mascara[i][2],mascara[i][3]);

         g.drawImage(img, x, y,

          Graphics.LEFT | Graphics.TOP);

      }

   g.setClip(0,0,ancho,alto);

   }

}

public int getX(){ return x;

}
public int getY(){ return y;
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241

Ediciones Universidad de Salamanca - cc by nc dc

```
}
public int[][] getMascaraColision(){ int masc[][];

    if (mascara != null){

        masc = new int[mascara.length][4];

        for (int i = 0; i < masc.length; i++) { masc[i][0] = mas-

            cara[i][0] + this.getX();

            masc[i][1] = mascara[i][1] + this.getY();

            masc[i][2] = mascara[i][2];

            masc[i][3] = mascara[i][3];
        }

    }else{

        masc = new int[1][4]; masc[0][0] = x;

        masc[0][1] = y;
        masc[0][2] = img.getWidth(); masc[0][3] =

        img.getHeight();

    }
    return masc;
}
public int checkCollision(Sprite sprite){ int sprt1[][] = this.get-

    MascaraColision();
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
int sprt2[][] = sprite.getMascaraColision();

for (int i=0; i<sprt1.length; i++){ for (int j=0;

  j<sprt2.length; j++){

    boolean colision = false;

    if (sprt1[i][0]+sprt1[i][2] == sprt2[j][0] &&

        ((sprt1[i][1] >= sprt2[j][1] &&

        sprt1[i][1] <= sprt2[j][1] + sprt2[j][3])

          ||

       (sprt1[i][1]+sprt1[i][3] >= sprt2[j][1] && sprt1[i][1]+sprt1[i][3] <=

          sprt2[j][1] + sprt2[j][3]))){

      //colision contra la parte izquierda del

      //sprite colision = true;

      return CalculadorTrayectoria.DERECHA;

    }

    if (sprt1[i][0] == sprt2[j][0] + sprt2[j][2] &&

        ((sprt1[i][1] >= sprt2[j][1] &&

        sprt1[i][1] <= sprt2[j][1] + sprt2[j][3])

          ||
       (sprt1[i][1]+sprt1[i][3] >= sprt2[j][1] && sprt1[i][1]+sprt1[i][3] <=
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                sprt2[j][1] + sprt2[j][3]))){
        //colision contra la parte derecha del
        //sprite"); colision = true;
        return CalculadorTrayectoria.IZQUIERDA;
    }
    if (sprt1[i][1] == sprt2[j][1] + sprt2[j][3] &&
            ((sprt1[i][0] >= sprt2[j][0] &&
            sprt1[i][0] <= sprt2[j][0] + sprt2[j][2])
                ||
          (sprt1[i][0]+sprt1[i][2] >= sprt2[j][0] && sprt1[i][0]+sprt1[i][2] <=
                sprt2[j][0] + sprt2[j][2]))){
        //colision contra la parte abajo del sprite colision = true;
                return CalculadorTrayectoria.ARRIBA;
    }
    if (sprt1[i][1]+sprt1[i][3] == sprt2[j][1] &&
            ((sprt1[i][0] >= sprt2[j][0] &&
            sprt1[i][0] <= sprt2[j][0] + sprt2[j][2])
                ||
                    (sprt1[i][0]+sprt1[i][2] >= sprt2[j][0] &&
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                    sprt1[i][0]+sprt1[i][2] <=
                        sprt2[j][0] + sprt2[j][2]))){
                  //colision contra la parte arriba del sprite colision = true;
                  return CalculadorTrayectoria.ABAJO;
                }
              }
            }
            // No hay colisión return -1;
          }

          public int getImgId(){ return
            imgId;
          }
        }
```

3.4 CONSTRUCCIÓN DEL CAMPO DE JUEGO

En este apartado, veremos cómo se genera el campo de juego a partir de una matriz de datos, que indica aquello que contiene en cada una de las celdas o partes en las que podemos descomponer el campo de juego. Como es habitual en los juegos 2D (en dos dimensiones), la pantalla se divide en celdas, de tal forma que se configure el campo a partir de la definición del contenido de cada celda.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

En el caso de nuestro juego de minigolf, trabajamos con un campo cuadrado de 11 celdas de ancho, por 11 celdas de alto. Nuestro juego está preparado para trabajar con un terminal cuya pantalla tiene 178 *pixels* de ancho y 180 *pixels* de alto. Por otra parte, los *sprites* que hemos definido para el juego son de forma cuadrada, con 16 *pixels* de lado. Con esta información es sencillo ver que si tenemos 11 celdas de 16 *pixels* de ancho, tenemos en total 176 *pixels* de lado en nuestro campo, que se acerca bastante a las dimensiones de la pantalla y que por lo tanto parece una dimensión razonable para nuestro campo de minigolf.

Para flexibilizar el proceso de carga de las pantallas por parte del juego y así hacer a este más potente, se ha creado una clase, que a partir de una matriz bidimesional que define los diferentes *sprites* que componen el campo de juego. La clase que se encarga de hacer esto, es

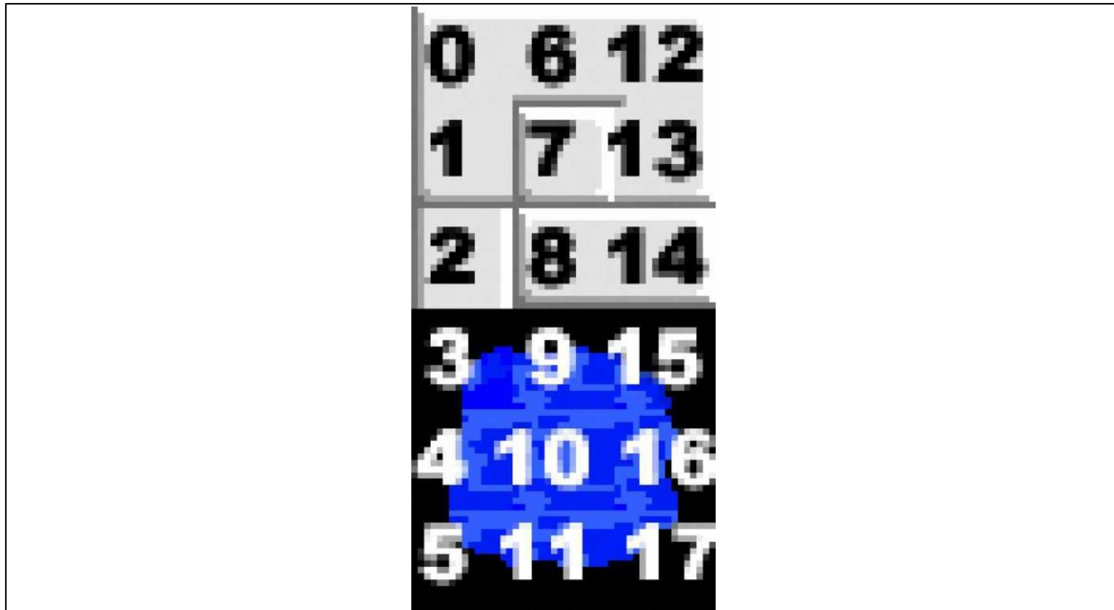
la clase *CargarCampo*. Esta clase posee un único método, que es estático y se denomina *getCampo*. Este método recibe como parámetro la matriz que define el campo y retorna una matriz con los *sprites* que componen el campo. Estos *sprites* serán los que tenemos que mantener controlados para comprobar las colisiones y dibujar el campo de juego.

Debemos tener en cuenta que evidentemente no todo el campo de juego está compuesto por *sprites* activos. Es decir, tendremos muchas zonas del campo de juego que podemos suponer vacías ya que no tienen ningún efecto sobre el juego. En nuestro caso, todas las zonas las zonas del campo que no correspondan a zonas de agua y no correspondan a obstáculos, podemos obviarlas, ya que no tendrán ningún efecto sobre nuestra bola. Para estas celdas sin efectos en el juego, que podríamos definir pasivas, no debemos comprobar las colisiones ni preocuparnos de su influencia en el juego, ya que esta no existe. Teniendo en cuenta esto, el método *getCampo* de la clase *CargarCampo*, sólo retorna en la matriz de *sprites*, aquellos *sprites* activos para el juego.

En la matriz que define el campo, los *sprites* pasivos se indican con el número –1, mientras que el resto de *sprites* se identifican con un número único. Este número se corresponde con el índice del *sprite* dentro de la clase *SpriteManager*. Este índice se asigna de manera automática durante el proceso de descomposición de la imagen que contiene todos los *sprites* necesarios para dibujar el campo. Es decir, el primer *sprite* que se extrae de la imagen contenedora tendrá el índice 0 dentro de la clase *SpriteManager*, el segundo tendrá el índice 1 y así sucesivamente. Según el código visto anteriormente para la clase *SpriteManager*, la extracción se realiza obteniendo primero todos los *sprites* de la primera columna (de arriba a abajo), después la segunda columna y así sucesivamente. Esto es importante porque determina el índice que definitivamente tendrá cada imagen.

El índice asignado a cada *sprite* dentro de la clase *SpriteManager*, será el mismo que se indicará dentro de la matriz que define el campo. La siguiente imagen muestra los índices asignados a cada *sprite* por la clase *SpriteManager* después de descomponer la imagen con todos los *sprites*.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc



A continuación vamos a ver un ejemplo del código fuente que define un campo de juego de nuestro minigolf, y la imagen que se generará en el terminal para dicho campo.

```
int tablaCampo[][]={

{-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
{-1, 0,12,-1,-1, 3, 9, 9,15,-1,-1},

{-1, 1,13,-1,-1, 4,10,10,16,-1,-1},

{-1,-1,-1,-1,-1, 5,11,11,17,-1,-1},

{-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},

{-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},

{-1, 0, 6, 6, 6,12,-1,-1,-1,-1,-1},

{-1, 2,-1,-1,-1, 2,-1,-1,-1,-1,-1},

{-1, 2,-1,-1,-1,-1,-1,-1,-1,-1,-1},

{-1, 2,-1,-1,-1, 2,-1,-1,-1,-1,-1},

{-1, 1, 6, 6, 6,13,-1,-1,-1,-1,-1}

};
```

La imagen correspondiente para el campo definido en el anterior código nos muestra el resultado del proceso de generación del campo de juego.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Ahora que hemos explicado el proceso de generación del campo, vamos a ver el código de la clase *CargarCampo* y que hace que todo esto sea posible.

```
public class CargarCampo {
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241

Ediciones Universidad de Salamanca - cc by nc dc

```
static int mask3[][] = {{8,10,8,7},{9,9,7,1},
    {10,8,6,1},{11,7,5,1},{13,6,3,1}};

static int mask9[][] = {{0,7,16,9},{7,8,5,1}};

static int mask15[][] = {{0,14,9,2},{0,11,8,3},

    {0,10,6,1},{0,9,4,1},{0,8,3,1},

    {0,7,1,1}};

static int mask16[][] = {{0,0,8,1},{0,1,9,2},

    {0,3,10,4},{0,7,9,1},{0,8,8,3},

    {0,11,9,2},{0,13,10,3}};

static int mask17[][] = {{0,0,10,2},{0,2,9,1},

    {0,3,7,1},{0,4,6,2},{0,6,4,1},{0,7,3,1}};

static int mask11[][] = {{0,0,16,8},{0,8,3,1},

    {7,8,8,1}};

static int mask5[][] = {{6,0,10,2},{7,2,9,2},

    {8,4,8,2},{10,6,6,1},{14,7,2,1},{15,8,1,1}};

static int mask4[][] = {{6,10,10,6},{7,4,9,6},

    {8,0,8,4}};

public static Sprite[] getCampo(int tablaCampo [][]){ int nSprites=0;

        Sprite campo[] = null;

 for (int i=0; i<tablaCampo.length; i++){

    for (int j=0; j<tablaCampo[i].length; j++){ if (tablaCampo[i][j] != -1){
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                nSprites++;
            }
        }
    }

    campo = new Sprite[nSprites];

    SpritesManager sprtMng = new SpritesManager(); int ind=0;

    for (int i=0; i<tablaCampo.length; i++){

        for (int j=0; j<tablaCampo[i].length; j++){ if (tablaCampo[i][j] != -1){

                int c = tablaCampo[i][j]; switch (c){

                    case 0:

                    case 1:

                    case 2:

                    case 6:

                    case 7:

                    case 8:

                    case 10:

                    case 12:

                    case 13:

                    case 14:

                        campo[ind++] = new Sprite( sprtMng.getImage(c), null,j*16, i*16,c);
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241

Ediciones Universidad de Salamanca - cc by nc dc

```
        break;
    case 3:
        campo[ind++] = new Sprite( sprtMng.getImage(c), mask3,j*16,i*16,c); break;
    case 4:
        campo[ind++] = new Sprite( sprtMng.getImage(c), mask4,j*16,i*16,c); break;
    case 5:
        campo[ind++] = new Sprite( sprtMng.getImage(c), mask5,j*16,i*16,c); break;
    case 9:
        campo[ind++] = new Sprite( sprtMng.getImage(c), mask9,j*16,i*16,c); break;
    case 11:
        campo[ind++] = new Sprite( sprtMng.getImage(c),mask11,j*16,i*16,c); break;
    case 15:
        campo[ind++] = new Sprite( sprtMng.getImage(c),mask15,j*16,i*16,c);
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                              break;
                         case 16:
                              campo[ind++] = new Sprite( sprtMng.getImage(c),mask16,j*16,i*16,c);
                              break;
                         case 17:
                              campo[ind++] = new Sprite( sprtMng.getImage(c),mask17,j*16,i*16,c);
                              break;
                    }
               }
          }
     }
     return campo;
   }
}
```

Al comienzo de la clase se definen una serie de variables estáticas que se corresponden con las matrices de colisión (o transparencia) de cada uno de los *sprites*. Estas variables son matrices de enteros, tal y como vimos en su día, y se llevan nombres de acuerdo al patrón *maskxx*, donde *xx* es el índice del *sprite*. Como se explicó anteriormente, algunos *sprites* no necesitan matriz de colisión ni de transparencia, por lo que la máscara que recibe el constructor del *sprite* tiene el valor *null*.

El método *getCampo*, lo primero que hace es descartar aquellas celdas cuyo índice del *sprite* es –1, ya que como hemos visto, son celdas sin importancia para el juego. Una vez hecho esto, para cada uno de las celdas a tener en cuenta, crea el *sprite* correspondiente y lo almacena en la matriz de *sprites* que retornará el método.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

3.5 OPERACIONES CON NÚMERO ENTEROS: FÍSICA DEL JUEGO

Los juegos J2ME son precisamente eso, juegos y por lo tanto suelen requerir de operaciones matemáticas más o menos complejas para simular en muchos casos ciertas leyes físicas. En Internet tenemos gran cantidad de información sobre estos temas, pensada para el desarrollo de juegos para ordenadores comunes. Toda estas información será útil y en muchos casos necesaria para el desarrollo de los juegos en J2ME, pero deberá adaptarse en mayor o menor alcance, para adaptarla a las capacidades de los entornos móviles. Las restricciones más importantes a tener en cuenta a la hora de implementar este tipo de código, es la pobre capacidad de proceso de los terminales móviles y que MIDP sólo presenta operaciones matemáticas con número enteros.

En nuestro caso, necesitamos conocer, entre otros puntos, cómo responden los objetos ante las colisiones, para simular el comportamiento de la bola de golf al rebotar contra los diferentes obstáculos. El cálculo de la trayectoria de la bola, concentra la mayor parte de las operaciones matemáticas presentes en el juego.

3.5.1 MANEJO DE SENOS Y COSENOS

Dentro de la especificación de MIDP no se contempla el cálculo de senos, cosenos o cualquier otra operación trigonométrica similar. En nuestro caso son necesarios los senos y los cosenos para el cálculo de la nueva trayectoria de la bola en caso de colisión y para la selección de la trayectoria inicial de la bola por parte del usuario.

Antes de golpear la bola, el usuario debe seleccionar la dirección inicial en la que se moverá esta. Para hacer esta operación, se le muestra al usuario una pequeña línea que parte de la bola y que se extiende en la dirección adecuada. Es decir, tendríamos una circunferencia virtual que tiene como punto central el centro de la bola (el *sprite* que representa a la bola realmente) y cada uno de los puntos de dicha circunferencia, definirá la dirección en la que se lanzará la bola. El cálculo de todos estos puntos que definen el círculo de selección de trayectoria, no se realizará en tiempo real, para optimizar el rendimiento del *midlet*. No debemos olvidar que cualquier operación, por simple que parezca, en el ámbito de los terminales móviles tiene un impacto, que debemos tener en cuenta, sobre el rendimiento de nuestro *midlet*.

Para optimizar este proceso se han precalculado todos los puntos que componen el círculo de selección de la trayectoria inicial. En concreto, se ha calculado, para cada posible punto de dicho círculo, la distancia vertical y horizontal desde el centro hasta dicho punto, de tal forma que sumando dichos valores al punto central de la bola, obtenemos el punto final de la línea que debemos dibujar para que el usuario seleccione la dirección de partida de la bola. Todos estos valores se han almacenado dentro de una variable estática que contiene 360 pares de valores. Cada uno de estos 360 elementos contiene la información referente a cada uno de los puntos del círculo, es decir, contendrá el desplazamiento vertical y el desplazamiento horizontal para el dibujado de cada punto. Estos valores están calculados de tal forma que el radio del círculo sea de 20 *pixels*.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Por otra parte, para calcular correctamente la trayectoria de la bola, debemos simular qué sucede cuando la bola choca con los límites del campo o con alguno de los obstáculos del campo. Cuando la bola colisiona con otro elemento dentro su trayectoria, la trayectoria cambia de dirección. Para calcular esta nueva dirección, debemos conocer el ángulo de entrada de la bola en la colisión, para averiguar el ángulo de salida y por lo tanto, conocer así la nueva trayectoria. Para hacer estos cálculos, necesitamos conocer los senos y los cosenos de los ángulos.

Como ya sabemos, las implementaciones estándar de J2ME no proporcionan métodos para calcular el seno o el coseno de un ángulo, por lo que tendremos que buscar una solución alternativa. La opción por la que se ha optado en nuestro caso es por precalcular los senos y cosenos de los diferentes ángulos. De esta manera, además de tener la funcionalidad que necesitamos, optimizamos el código en cierta medida.

Los valores de los senos y los cosenos están precalculados únicamente para los ángulos entre 0 y 90, ambos inclusive, ya que el valor para el resto de ángulos se puede calcular a partir de estos valores. Otro problema que nos encontramos en el manejo de los senos y los cosenos, es que estos elementos presentan valores entre 0 y 1 (ambos inclusive). Volvemos a enfrentarnos con un problema de manejo de número no enteros. Para solucionar este problema, se ha optado por tomar en el precálculo, en valor de seno multiplicado por 10.000. De esta forma, trabajaremos siempre con número enteros (con una precisión de 4 decimales) y los resultados serán los esperados.

Todas estas operaciones serán básicas en el cálculo de la trayectoria de la bola. El siguiente fragmento de código muestra cómo se recalcula la trayectoria de la bola después de una colisión contra un obstáculo.

```
if (pared == CalculadorTrayectoria.ARRIBA ||

        pared == CalculadorTrayectoria.ABAJO){ direccionY = -direccionY;

}

if (pared == CalculadorTrayectoria.DERECHA || pared == CalculadorTrayecto-

          ria.IZQUIERDA){

                direccionX = -direccionX;

  }

if (pared == ARRIBA){ angulo = -angulo;
```

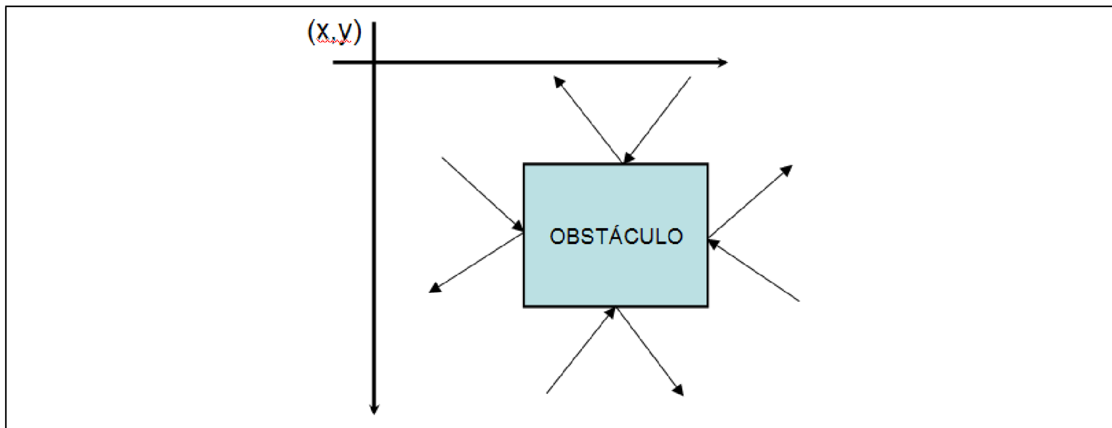
ISBN: 978-84-9012-861-9

Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        yActual++;;

}

if (pared == ABAJO){ angulo = -

        angulo; yActual--;;

}

if (pared == IZQUIERDA){ if (angulo <

     0){

                  angulo = -180 - angulo;

     }else{
          angulo = 180 - angulo;

     }

     xActual++;

}

if (pared == DERECHA){ if (angulo <

     0){

          angulo = -180 - angulo;

     }else{

          angulo = 180 - angulo;

     }

     xActual--;

}
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Para calcular correctamente el resultado, es necesario saber contra qué parte del bloque ha colisionado la bola. En nuestro caso, que los bloques son cuadrados, la colisión puede ser contra la parte de arriba o abajo, la izquierda o la derecha del obstáculo. Los dos primeros *if* del código anterior, comprueban si hay que cambiar la dirección de la bola. Básicamente, en una colisión con la parte inferior o superior de un bloque la dirección vertical de la bola variará, pero la horizontal permanecerá invariable. Por el contrario, en un choque con la parte izquierda o derecha de un obstáculo, la dirección horizontal variará pero la dirección vertical permanecerá inalterable. El siguiente dibujo ilustra este concepto.



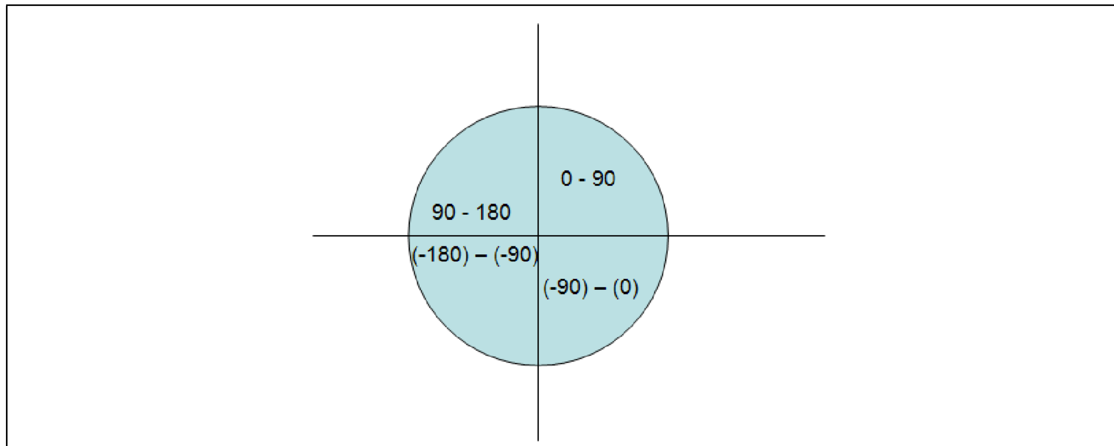
El elemento esencial para recalcular la trayectoria después de una colisión, es conocer el ángulo de entrada, para calcular el ángulo de salida y en función de este ángulo de salida, podremos saber hacia qué punto se dirige la bola como resultado de la colisión. Este cálculo se hace en la segunda parte del listado de código presentado. Para calcular el ángulo de salida lo

único que necesitamos es conocer el ángulo de entrada y la parte del obstáculo con el que ha colisionado la bola.

Para calcular hacia dónde se dirige la bola, una vez conocido el ángulo de salida de la colisión, tendremos que averiguar el seno y el coseno de dicho ángulo. La segunda parte del código presentado anteriormente, muestra cómo se calculan dichos valores, que como se ha mencionado anteriormente, se calculan a partir de los valore previamente conocidos del seno y el coseno de los ángulos entre 0 y 90.

En nuestro juego, usamos ángulos entre –180 y 180 grados. La siguiente imagen muestra el rango de ángulos con el que trabajamos y dónde se sitúa cada uno de los ángulos.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc



En principio, el semicírculo superior se puede considerar de la forma habitual, es decir, el extremos derecho de dicho semicírculo corresponde con el ángulo 0 y el extremo izquierdo corresponde con el ángulo 180. Una vez colocados los ángulos en este semicírculo, el ángulo que podríamos definir como reflejo en el semicírculo inferior, será el valor negativo del ángulo correspondiente en la parte superior. De este manera, el extremo izquierdo del semicírculo inferior tendrá corresponderá al ángulo –180 y este valor irá creciendo hasta el valor imaginario de –0.

Una vez sistema de denominación de ángulos que se ha usado en el desarrollo del juego, vamos a mostrar el fragmento de código que muestra cómo se calculan los senos y los cosenos para cualquier ángulo.aclarado el

```
if (angulo <= 90 && angulo > 0){

    coseno = Constantes.coseno[angulo]; seno = Constantes.seno[an-

    gulo];

} else if (angulo > 0) {

    coseno = -Constantes.coseno[180-angulo]; seno = Constantes.seno[180-
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        angulo];
    } else if (angulo > -90) {
        coseno = Constantes.coseno[-angulo];
        seno = -Constantes.seno[-angulo];
    } else{
        coseno = Constantes.coseno[180+angulo]; seno = -Constan-
        tes.seno[180+angulo];
    }
```

La clase C*onstantes* contiene las variables estáticas seno y coseno, que son una matriz de valores enteros, con el valor del seno y el coseno de los ángulos entre 0 y 90.

Ahora que conocemos el seno y el coseno del ángulo de salida de una colisión, vamos a calcular el punto exacto al que se dirigirá la bola. Por lo tanto, conocido el punto de colisión (punto actual de la bola) y el punto al que se dirige, conoceremos la nueva trayectoria de la bola, ya que dos puntos definen una recta. El siguiente código muestra cómo se calcula el punto al que se dirige la bola.

```
    if ((angulo >=0 && angulo <= 180) ||
            (angulo > -91)){
        y1 = yActual - (seno / 20); x1 = xActual + (co-
        seno / 20);
    } else {
        y1 = yActual - (seno / 20); x1 = xActual - (coseno
        / 20);
    }
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

El punto destino de la bola será el punto *(x1, y1)*. Vamos a explicar cómo realizamos este cálculo. A partir de un punto inicial (*x0, y0)*, y de un ángulo *a*, el punto destinatario será:

- x1 = x0 + (v*cos(a))
- y1 = y0 + (v*sen(a))

En las ecuaciones anteriores, *v* determinará la longitud de la recta, es decir, la distancia entre los puntos inicial y final. En nuestro caso, hemos optado por una valor de 500 para *v*, ya que 500 puntos en la recta que define la trayectoria, nos aseguran que antes de alcanzar el último punto de dicha recta, la bola colisionará con los bordes de la pantalla y por lo tanto se recalculará una nueva trayectoria. Como habíamos dicho anteriormente, nuestras constantes para el seno y el coseno tiene dicho valor multiplicado por 10000. Por lo tanto, para obtener el valor real, tendremos que dividir por 10000. Así, tendremos que el seno y el coseno se multiplica por 500 y se divide por 10000, es decir, se divide por 20 y se optimiza el cálculo, tal y como se puede comprobar en el último fragmento de código presentado.

3.5.2 ALGORITMO DE CÁLCULO DE LÍNEA

La trayectoria que describe la bola en su movimiento, es una línea recta. Como hemos comentado en el párrafo anterior, partiendo del punto inicial y el punto final de la trayectoria, debemos calcular todos los puntos intermedios. Para hacer este cálculo usaremos uno de los algoritmos de cálculo de líneas más conocido, el algoritmo de línea de Bresenham. Este algoritmo es especialmente adecuado para nuestro caso, ya que realiza todos los cálculos con números enteros. En realidad lo que haremos será usar este algoritmo como base para crear nuestro algoritmo de cálculo de trayectorias.
El código del algoritmo de Bresenham, para rectas con pendiente positiva, es el siguiente:

```
void bres_line (int x1, int y1, int x2, int y2){
    int dx; int

    dy; int x; int

    y;

    int x_end; int p;

    int c1; int

    c2;

    dx    =    Math.abs(x1-x2);    dy    =
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        Math.abs(y1-y2); p = 2*dy-dx;

        c1 = 2*dy;
        c2 = 2* (dy - dx);
        {Se determina el punto a usar como inicial y final} if (x1 > x2){x = x2; y = y2;

            x_end = x1;

        } else { x = x1;

            y = y1;

            x_end = x2;

        }

        dibujarPixel (x,y); while

        (x<x_end){

            x = x++; if

            (p<0){

                p = p + c1;

            }else{

                y++;

                p = p + c2;

            }

            dibujarPixel(x,y);

        }

    }
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Este código nos sirve como punto de partida, pero como podemos comprobar, este algoritmo calcula los puntos intermedios de la recta sin tener en cuenta el punto inicial y el punto final que se le indican como parámetro. Es decir, dados los puntos inicial y final, puede dibujar la línea del punto inicial al final o viceversa. En nuestro caso, los puntos se deben calcular en un determinado orden, ya que serán los puntos que recorrerá la bola en su trayectoria. Por lo tanto, a partir de este algoritmo, se calcularán los valores que nos permiten calcular el siguiente punto y se calculará este, siempre teniendo en cuenta cuál es el punto inicial y cuál es el punto final.

Dentro de la clase creada en nuestro juego para calcular la trayectoria completa de la bola, tendremos un método para calcular los parámetros que nos permiten ir averiguando los puntos por los que pasa la bola. Dicho método es muy similar al algoritmo de Bresenham, ya que parte de él. A continuación se muestra el listado del método en cuestión.

```
private void calcularParametrosRecta(int x0, int y0,

                  int x1, int y1){ int dx, dy, xend;

  dx = Math.abs(x1-x0); dy = Math.abs(y1-

  y0);


  if (dx >= dy){ pendiente = 0;  p =

    2 * dy - dx; incE = 2 * dy;

    incNE = 2 * (dy - dx);

    xend = x1;

    if (direccionY == 0) { if (y0 > y1) {

          direccionY = -1;

      }

      else {

              direccionY = 1;

      }

    }

    if (direccionX == 0) { if (x0 > x1) {

          direccionX = -1;
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        }

        else {
            direccionX = 1;
        }
    }
     }else{
    pendiente = 1;  p = 2 * dx - dy;

    incE = 2 * dx;

    incNE = 2 * (dx - dy); xend = y1;

    if (direccionY == 0) { if (y0 > y1) {

            direccionY = -1;
        }
        else {
            direccionY = 1;
        }
    }
    if (direccionX == 0) { if (x0 > x1) {

            direccionX = -1;

        }

else {
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                direccionX = 1;
            }
        }
    }
}
```

Las variables *p, incE* y *incNE*, presentes en el método son variables globales de la clase. En este método se establecen sus valores y posteriormente se usarán para calcular el siguiente punto de la trayectoria.

El código anterior es muy sencillo, conocido el algoritmo de Bresenham. En nuestro caso, se tienen en cuenta todas la rectas, no sólo aquellas de pendiente positiva. Además, también se calcular la dirección en la que se debe mover la bola, tanto en vertical como en horizontal. Es decir, básicamente lo que se calcula son unas variables (*direccionX* y *direccionY*), que nos permite saber si en nuestra trayectoria, dado un punto (*x,y)*, debemos sumar o restar valores a *x* o *y* para calcular el siguiente.

3.6 CÁLCULO COMPLETO DE LA TRAYECTORIA DE LA BOLA

En este punto del capítulo tenemos ya los conocimientos necesarios para comprender por completo el proceso de cálculo de la trayectoria de la bola. Dentro del juego, lo que se hace es calcular un número suficientemente grande de puntos de la trayectoria de la bola, de tal forma que nos aseguremos que para el valor máximo de la fuerza que el usuario puede seleccionar para golpear la bola, el número de puntos que tenemos precalculados es suficiente.

Como veremos más adelante, el proceso de golpeo de la bola tiene los siguientes pasos, desde el punto de vista del usuario:

- Seleccionar dirección de la bola
- Seleccionar fuerza con la que se golpea la bola
- Ver la trayectoria de bola y comprobar el resultado de los puntos anteriores

En el código implementado, para que el usuario no tenga que esperar (o tenga que esperar lo mínimo posible) por el cálculo de la trayectoria, esta se calcula en un *thread* independiente una vez que el usuario selecciona la dirección inicial de la bola. Es decir, es posible que calculemos

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

500 puntos de la trayectoria y que luego el usuario seleccione golpear la bola con poca fuerza y realmente se muestren sólo 50 de esos 500 puntos. Este es uno de los puntos que se pueden mejorar del juego y vuelvo a reiterar la propuesta de la introducción, le animo a ello.

Bien, vamos a ver el fragmento de código exacto que calcula la trayectoria, por supuesto, utilizando el código que ya hemos visto para detección de colisiones, cálculo de la nueva dirección de la bola después de una colisión...

El proceso es sencillo y se resume en los siguientes pasos:

- Se calcula el siguiente punto de la trayectoria, a partir de las variables calculadas anteriormente, tal y como se explicó en el apartado anterior
- Se comprueba si dicho punto sale fuera de los límites de la pantalla. Si es así, calculamos la nueva trayectoria, ya que la bola ha colisionado con un borde de la pantalla
- Comprobamos si la bola ha colisionado con algún otro *sprite*
- Si es un *sprite* correspondiente a una zona de agua, finaliza el cálculo de la trayectoria, indicando como punto final de la misma el punto inicial. Es decir, si la bola cae al agua, se vuelve a lanzar desde el punto inicial
- Si es un *sprite* correspondiente a un obstáculo, se recalcula la nueva trayectoria de la bola, tal y como hemos visto
- Si la colisión es contra el *sprite* correspondiente al hoyo, se finaliza el cálculo de la trayectoria y se indica como punto final el punto (1000,1000), de tal forma que si al dibujar la trayectoria encontramos que esta acaba en dicho punto, sabremos que se ha embocado.

Este proceso se repite hasta que se hayan calculado el total de puntos de la trayectoria o se finalice por los eventos antes descritos.

3.7 PANTALLA PRINCIPAL DEL JUEGO

A continuación vamos a ver cómo funciona la pantalla principal del juego. Como es de esperar, esta clase extiende la clase *Canvas* e implementa un hilo de ejecución (*thread*), de tal forma que sea capaz de ir dibujando las animaciones en la pantalla. Más adelante veremos esto en detalle. La clase principal del juego es la clase *PantallaGolf*.

Para comenzar vamos a ver el constructor de esta clase. En dicho constructor se inicializan algunas variables de la clase. Entre los procesos de inicialización, debemos destacar la creación del *sprite* para la bola y el *sprite* para el hoyo. También en el constructor se carga el campo de juego. Esto es así en este caso, porque estamos desarrollando una versión no completa del juego, de tal forma que sólo tendremos un campo de golf para jugar, cosa nada normal en lo que

ISBN: 978-84-9012-861-9

Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

sería un juego comercial. Para realizar un juego completo, tendríamos, seguramente, una clase con una serie de campos, y sabría qué campo proporcionar a la pantalla principal en cada momento para jugar. Esto es básicamente una versión del juego con varias pantallas. Una vez, animo al lector a hacerlo. Ya vimos en su momento cómo se crea el campo de minigolf.

Por último, dentro del constructor destacaremos la creación de una imagen del mismo tamaño que la pantalla, que se utilizará en aquellas situaciones en las que el fondo de la pantalla del juego no varía, de tal forma que en lugar de dibujar todos los elementos que componen la pantalla, tendremos una imagen con la pantalla completa y la dibujaremos con una única instrucción. Este hecho ocurre por ejemplo durante el proceso de selección de la fuerza del golpe. Durante este proceso se mostrará el campo de juego, y sobrepuesta sobre este una barra de de color que nos permite seleccionar la fuerza con la que se desea golpear la bola. La siguiente imagen ilustra este proceso y vemos que varía únicamente la barra roja y no el fondo de la pantalla.

3.7.1 CONTROL GENERAL DE LOS PROCESOS

La pantalla principal del juego deberá reaccionar de forma distinta ante distintas fases del juego. Así, en función del punto del juego en el que estemos, la pantalla principal deberá mostrar el círculo de selección de la trayectoria inicial de la bola, deberá mostrar el selector de fuerza (como se muestra en la imagen anterior) o se deberá dibujar el proceso de movimiento de la bola. Este hecho se repite para el control de las pulsaciones de las teclas por parte del usuario, según la fase del juego, la pulsación de una tecla tendrá consecuencias diferentes.

Para controlar en qué fase del juego nos encontramos, tenemos una variable global a toda la clase, denominada *tarea* y que indica en qué fase del juego nos encontramos. Los posibles

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

valores que puede tomar la variable *tarea* se definen en las siguientes constantes dentro de la clase:

- PINTAR_CAMPO – Determina que la operación a realizar es pintar el campo de juego. Esta tarea se realiza al comienzo del juego, para dibujar el campo completo de juego.

- PINTAR_CIRCULO – Indica que se está seleccionando la trayectoria inicial con la que comenzará a moverse la bola

- PINTAR_SELECTOR_FUERZA – Será la tarea de dibujar el selector de fuerza para que el usuario seleccione la fuerza con la que se golpeará la bola

- PINTAR_MOVIMIENTO_BOLA – Esta tarea, una vez calculada la trayectoria, muestra dicha trayectoria. Se pintarán tantos valores como permite la fuerza seleccionada para golpear la bola

- PINTAR_FINAL – Indica que hemos embocado la bola y que debemos mostrar la pantalla de felicitación. En nuestro caso se muestra el logo del juego

Teniendo en cuenta este método de control de las operaciones que se realizan en la pantalla principal, vamos a ver cómo funcionan los métodos de esta clase.

3.7.2 EL MÉTODO PAINT

Si el método *paint* es llamado durante la tarea de dibujado del campo, que será la primera operación que se realiza, lo que se hará es dibujar sobre la imagen de la que hemos hablado que se utilizará para almacenar el fondo todos los *sprites* que componen el campo, incluyendo el hoyo y la bola. Una vez que se ha dibujado el campo, se establece como tarea activa el dibujado del círculo de selección de la trayectoria inicial. En este momento ya tenemos almacenado el campo de juego dentro de la imagen antes mencionada.

Si la tarea activa es *PINTAR_CIRCULO*, se dibujará una línea desde el punto central de la bola, hasta el punto definido por el ángulo en cada momento. Este ángulo es modificado por el usuario a través de las pulsaciones de teclas, tal y como veremos más adelante. La siguiente imagen muestra el resultado del método *paint* durante esta parte del juego. La parte importante de esta pantalla es la que se ha remarcado con un círculo rojo.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Para pintar el selector de fuerza, se muestra un rectángulo relleno de color rojo en la pantalla que varía de longitud. La parte del rectángulo que se muestra de color indicará la fuerza con la que se golpeará la bola. La longitud de esta zona de color variará de forma automática, tal y como veremos más adelante cuando se estudie en detalle el método *run*. La siguiente imagen muestra el resultado del método *paint* durante el proceso de selección de fuerza.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Una vez seleccionada la fuerza para golpear la bola, se mostrará la trayectoria de la bola. Para realizar este proceso, el método *paint* simplemente muestra el campo del juego, a través de la imagen que hemos creado y configurado anteriormente y la posición de la bola en cada momento. Para hacer esto, se dibuja la imagen y se le ordena al *sprite* bola que se dibuje. El movimiento de la bola se hace en el método *run* tal y como veremos más adelante.

Por último, tenemos la pantalla final del juego, cuando se logra embocar la bola en el hoyo. En una versión comercial del juego, tendríamos una pantalla de puntación y el paso al siguiente nivel. En nuestro caso, lo que tendremos será la pantalla de presentación del juego. Con esto simplemente se pretende ilustrar el proceso de finalización del juego

En esta última pantalla será en la única en la que no se dibuja la posición de la bola dentro del campo. A continuación, se muestra el código del método *paint* de la clase *PantallGolf.*

```
public void paint (Graphics g){

    if (tarea == PantallaGolf.PINTAR_CAMPO){ gFondo.setColor(0,200,10); gFondo.fill-

        Rect(0,0,this.getWidth(),

                        this.getHeight());

        for (int i=0; i<campo.length; i++){ campo[i].paint(gFondo);
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
}

hoyo.paint(gFondo);

tarea = PantallaGolf.PINTAR_CIRCULO;
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        // Elemento del círculo a pintar = ángulo
        eltoCirculo=0; g.drawImage(fondo,0,0,
            Graphics.LEFT|Graphics.TOP);
    }
    if (tarea == PantallaGolf.PINTAR_CIRCULO){ g.drawImage(fondo, 0, 0,
                    Graphics.LEFT|Graphics.TOP); g.setColor(0,0,0);
        int[] c = pintarCirculo(); g.drawLine(bola.getX()+3, bola.getY()+3,
            bola.getX()+3+c[0], bola.getY()+3+c[1]);
    }
    if (tarea == PantallaGolf.PINTAR_SELECTOR_FUERZA){ g.drawImage(fondo, 0, 0,
            Graphics.LEFT | Graphics.TOP);
        g.setColor(0,0,0); g.drawRect(xFuerzaGolpe-1,yFuerzaGolpe-1,
                    maxFuerzaGolpe+1,21); g.setColor(200,0,0); g.fi-
        llRect(xFuerzaGolpe,yFuerzaGolpe,
                    fuerzaGolpe,20);
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        }

        if (tarea == PantallaGolf.PINTAR_MOVIMIENTO_BOLA){ g.drawImage(fondo, 0,

            0,

                Graphics.LEFT|Graphics.TOP);

        }

        if (tarea == PantallaGolf.PINTAR_FINAL){ g.drawImage(imagenFinal,0,0,

                Graphics.LEFT|Graphics.TOP);

        }else{

            bola.paint(g);

        }

    }
```

El método *pintarCirculo* que se utiliza en el código anterior, tiene como función retornar el punto concreto del círculo que nos servirá para trazar la línea correspondiente. Este punto se toma de la constante presentada anteriormente en función del valor del campo *elementoCirculo*, que define el ángulo del círculo seleccionado en cada momento.

3.7.3 GESTIÓN DE LOS COMANDOS DEL USUARIO

Como ya hemos comentado anteriormente y hemos visto en el apartado anterior, el juego reaccionará de forma diferente según la fase en la que se encuentre. Es decir, en función de la tarea en la que se encuentre el juego, deberemos responder de forma diferentes a las pulsaciones de las teclas por parte del usuario.

Como ya sabemos, dentro de un *canvas* tenemos en el método *keyPressed* como punto principal para la gestión de las pulsaciones de teclas. Este método es invocado cuando el usuario pulsa una tecla, pero tenemos otro método para controlar cuando el usuario suelta la tecla que ha pulsado. Este método es el método *keyReleased*. En nuestro caso, tenemos la mayoría de la funcionalidad dentro del método *keyPressed*, pero utilizaremos también el método *keyReleased* para implementar una funcionalidad que nos permite controlar pulsaciones prolongadas por parte del usuario.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

En este punto ya conocemos cómo funciona el juego, por lo que nos será sencillo comprender que únicamente tenemos que controlar pulsaciones de tecla durante el proceso de selección de la trayectoria inicial de la bola y durante la selección de la fuerza de golpeo.

Para el primer caso, el proceso de selección de la trayectoria inicial de movimiento de la bola, el juego responderá modificando el ángulo seleccionado y por lo tanto se moverá la línea que va desde el punto central de la bola hasta el punto del círculo determinado por el ángulo. Para hacer esto, el sistema responde a la pulsación de las teclas del cursor (arriba, abajo, derecha e izquierda). El número de puntos que podemos seleccionar es 360 y por lo tanto el proceso, si sólo respondemos a las pulsaciones de tecla, será muy tedioso. Por ejemplo, para pasar del punto inicial, que es el ángulo 0 y la línea es completamente horizontal, al punto determinado por el ángulo 90, es decir, una línea completamente vertical, el usuario tendría que pulsar 90 veces la tecla correspondiente. Para agilizar este proceso, controlamos las pulsaciones prolongadas de teclas.

Cuando el usuario pulsa una tecla, el ángulo empieza a variar de forma automática (controlado por el método *run* de la clase) hasta que el usuario deja de pulsar la tecla. Esto nos permite implementar una interfaz de usuario mucho más eficiente. El resultado es que mientras el usuario tenga pulsada una tecla, la línea que define la trayectoria inicial de la bola se moverá en el sentido adecuado hasta que el usuario deje de pulsar dicha tecla.

Para finalizar el proceso de selección de la trayectoria inicial, el usuario debe pulsar la tecla de fuego (*fire*). Cuando ocurre esto, se comienza a calcular la trayectoria que describirá la bola, tal y como se ha explicado anteriormente y se pasa a la tarea de selección de la fuerza con la que se golpeará la bola.

Durante el proceso de selección de la fuerza de golpeó, lo único que se debe hacer es esperar a que el usuario pulse una tecla, lo que indicará que desea golpear la bola con la fuerza que se está mostrando en ese momento. Como respuesta a esta pulsación, lo que se hará es recuperar la trayectoria calculada y pasar a la tarea en la que se dibuja la trayectoria de la bola. Como se explicó en su momento, si el cálculo de la trayectoria no ha finalizado cuando esta es requerida (cuando el usuario pulse una tecla) el método mediante el que se solicita la trayectoria devolverá *null* y por lo tanto deberemos a que esta esté disponible para comenzar la tarea de dibujo de la trayectoria en la pantalla.

A continuación se muestra el código de este método, que implementa todo lo que acabamos de exponer

ISBN: 978-84-9012-861-9

Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
public void keyReleased(int tecla){

    // Paramos el movimiento de la línea de selcción avanceEltoCirculo=0;

}

public void keyPressed(int tecla){ tecla = this.getGameAction(te-

    cla); switch(tecla){

        case PantallaGolf.LEFT:
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
case PantallaGolf.UP:

    if (tarea == PantallaGolf.PINTAR_CIRCULO){ avanceEltoCirculo = 1;

    }
    break;
case PantallaGolf.RIGHT: case PantallaG-

olf.DOWN:

    if (tarea == PantallaGolf.PINTAR_CIRCULO){
        avanceEltoCirculo = -1;
    }
    break;
case PantallaGolf.FIRE:
    if (tarea == PantallaGolf.PINTAR_CIRCULO){
        // Transformo el ángulo del rango 0-360
        // al rango (-180)-180, que es con el que
        // se trabaja para calcular la trayectoria if (eltoCirculo >= 180 &&

                eltoCirculo <= 270){
            eltoCirculo = -(90 + (270 - eltoCirculo));
        } else if (eltoCirculo > 270){ eltoCirculo = -(360 - eltoCirculo);

        }

        // Comienzo a calcular la trayectoria calc.setDatosIniciales(bola.getX(),
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                    bola.getY(), -eltoCirculo);
                calc.calcularTrayectoria();
                tarea = PantallaGolf.PINTAR_SELECTOR_FUERZA;
            } else {
                if (tarea ==
                    PantallaGolf.PINTAR_SELECTOR_FUERZA) {
                  // Mientras la trayectoria no esté lista
                  // esperamos
                  while ((trayectoria=calc.getTrayectoria())
                            == null){ try{
                      Thread.sleep(100);
                             }catch (InterruptedException iex){}
                  }
                  tarea=PantallaGolf.PINTAR_MOVIMIENTO_BOLA; indMovimiento=0;
                }
            }
        }
    }
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

3.8 EL MÉTODO RUN, EL CORAZON DEL JUEGO

Para finalizar, vamos a estudiar el método *run* de la clase *PantallaGolf*. Como ya hemos dicho, esta clase implementa la interfaz *Runnable*, de manera que el método *run*, como en cualquier-clase de este tipo, se ejecutará constantemente para realizar las operaciones adecuadas. En nuestro caso, en función de la fase del juego en la que nos encontremos, el método *run* deberá de hacer unas operaciones u otras.

La clase *PantallaGolf* es un *Canvas* y por lo tanto tiene la responsabilidad de dibujar en la pantalla del móvil lo que corresponda en cada momento. Es decir, lo que hará el método *run* responderá al siguiente esquema:

- Hacer los calculados adecuados y hacer las operaciones correspondientes
- Repintar la pantalla
- Esperar un determinado tiempo
- Volver al punto 1

Con la espera del punto 3, conseguimos asegurarnos de que se van a mostrar un número determinado de pantallas por segundo. Este factor se conoce como *fps (frames per second)*. Para conseguir que el número de *frames* por segundo sea constante, deberemos dormir siempre un tiempo, de tal forma que el tiempo total entre cada pantalla sea constante y por lo tanto el usuario tenga una sensación de dinamismo en el juego. Para conseguir que el tiempo entre diferentes pantallas sea constante, debemos calcular el tiempo que se tarda en hacer todas las operaciones y luego dormir el tiempo restante hasta conseguir que la suma del tiempo consumido, más el tiempo que duerme el proceso sea siempre el mismo.

A continuación vamos a ver qué debe hacer el método *run* en cada una de las fases del juego. Durante el proceso de selección de la trayectoria inicial de la bola, el método *run* deberá modificar el ángulo o lo que es lo mismo, modificar la línea en la dirección adecuada si alguna de las teclas de los cursores está pulsada, tal y como vimos en el punto anterior. El siguiente fragmento de código muestra este proceso.

```
if (tarea == PantallaGolf.PINTAR_CIRCULO){

    // avanceEltoCirculo será 1 o -1 eltoCirculo+=avanceElto-

    Circulo;

    if (eltoCirculo >= Constantes.circulo.length) eltoCirculo = 0;

    if (eltoCirculo < 0)
```

ISBN: 978-84-9012-861-9

Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        eltoCirculo = Constantes.circulo.length - 1;

    }
```

El proceso que realiza el método *run* durante la selección de la fuerza de golpeo de la bola es muy sencillo. Simplemente modifica el área de la zona coloreada que se muestra en la pantalla para seleccionar la fuerza. El siguiente fragmento de código muestra dicho proceso.

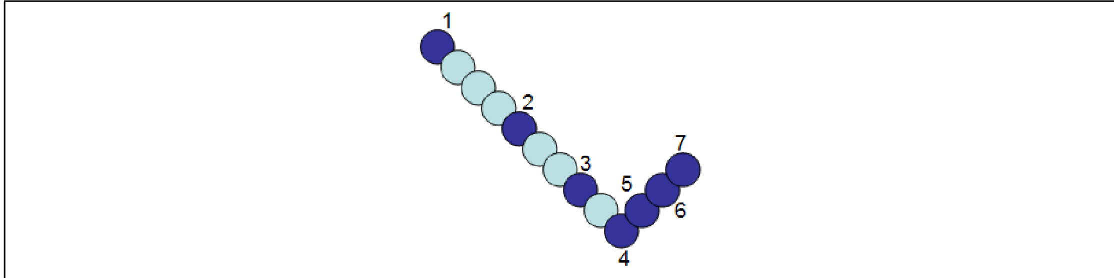
```
    if (tarea ==
```

```
            PantallaGolf.PINTAR_SELECTOR_FUERZA){

        fuerzaGolpe += incFuerzaGolpe;

        // Al llegar a los extremos, variamos la

        // dirección

        if (fuerzaGolpe == 100){ incFuerzaGolpe=-1;

        }

        if (fuerzaGolpe == 0){ incFuerzaGolpe=1;

        }

    }
```

Por último vamos a ver el código del método *run* que ejecuta para mostrar al usuario la trayectoria que describe la bola después de golpearse esta. Como sabemos, tenemos la trayectoria almacenada en una matriz bidimensional de números enteros. En el mundo real, al golpear la bola está empezará a moverse con una determinada fuerza que irá disminuyendo a medida que esta se mueve y va perdiendo la fuerza con la que se le impulsó. Para simular este hecho, nosotros

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

haremos lo siguiente. En función de la fuerza que tenga la bola en cada momento, se saltarán un determinado número de puntos de la trayectoria.



Por ejemplo, todos los puntos de la imagen anterior definen una trayectoria. A medida que la bola se mueve esta va perdiendo fuerza y por lo tanto cada vez se saltan menos puntos de la trayectoria. Siguiendo con el caso de la imagen anterior, se mostrarán únicamente por pantalla los puntos de la trayectoria de color azul oscuro. Es decir, se pinta el primer punto y se saltan tres puntos de la trayectoria. Se pinta el siguiente punto y luego se saltan dos puntos, ya que la bola va perdiendo fuerza. Este proceso se repite hasta que la bola pierde totalmente su fuerza o se llega al final de la trayectoria porque la bola ha caído al agua o porque ha embocado.

La máxima fuerza que se le puede asignar a la bola es un valor de 100. Cada vez que se muestra un punto, esta fuerza disminuye. Para calcular el número de puntos a saltar de la trayectoria y por lo tanto no hay que mostrar, se toma el número correspondiente a las decenas del valor que define la fuerza de movimiento de la bola en ese momento. Cuando el valor de dicha fuerza está por debajo 10, se avanzará siempre un punto sobre la trayectoria. Por

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

ejemplo, si el valor de la fuerza de la bola es 51, se saltarán 5 puntos de la trayectoria y se disminuirá el valor de la fuerza hasta 50. Para 50, se saltan otros 5 puntos y se disminuye la fuerza a 49, por lo que después de mostrar un punto, se saltaran otros 4.

De esta manera, determinamos el siguiente punto de la trayectoria a mostrar. Una vez que tenemos dicho punto, se mueve el *sprite* correspondiente a la bola hasta dicho punto y se actualiza la pantalla del terminal. Como se ha comentado, este proceso se repite hasta que la bola se queda sin fuerza o hasta que llegamos al final de la trayectoria. Cuando ocurre uno de estos dos hechos, se vuelve al principio del proceso y por lo tanto volvemos al proceso de selección de la trayectoria original de la bola, siempre que no se embocara la bola en el hoyo.

El siguiente fragmento de código se corresponde con el método *run* de la clase *PantallaGolf* e ilustra todo lo que acabamos de exponer.

```
public void run() {

    while (true) {

        long tiempoInicial = System.currentTimeMillis();

        if (tarea == PantallaGolf.PINTAR_CIRCULO) { eltoCirculo += avanceEltoCirculo;

            if (eltoCirculo >= Constantes.circulo.length) { eltoCirculo = 0;

            }

            if (eltoCirculo < 0) {

                eltoCirculo = Constantes.circulo.length - 1;

            }

        }

        if (tarea==PantallaGolf.PINTAR_SELECTOR_FUERZA){ fuerzaGolpe +=

            incFuerzaGolpe;

            if (fuerzaGolpe == 100) {

                incFuerzaGolpe = -1;
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
            }
            if (fuerzaGolpe == 0) { incFuerzaGolpe = 1;
            }
        }
        if (tarea==PantallaGolf.PINTAR_MOVIMIENTO_BOLA){ if (fuerzaGolpe > 0 &&
            indMovimiento < calc.getNumeroPuntos()) { int avance = (int) (fuerza-
                Golpe / 10); if (avance == 0) {
                    avance++;
                }
                indMovimiento += avance; fuerzaGolpe--;
                if (indMovimiento < calc.getNumeroPuntos()) { bola.moveTo(trayectoria[indMovimiento][0],
                                        trayectoria[indMovimiento][1]);
                }
                else {
                    bola.moveTo( trayectoria[calc.getNumeroPuntos() - 1][0],
            trayectoria[calc.getNumeroPuntos() - 1][1]);
                }
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                if (trayectoria[indMovimiento][0] == 1000 &&

                        trayectoria[indMovimiento][1] == 1000) { tarea = Pantalla-

                    Golf.PINTAR_FINAL;

                }

            }
            else {
                tarea = this.PINTAR_CIRCULO; eltoCirculo = 0;

                fuerzaGolpe = 0;
                incFuerzaGolpe = 1;
            }

        }
        this.repaint(); this.serviceRepaints();

        long tiempoFinal = System.currentTimeMillis(); try {

            long r = tiempoFinal - tiempoFinal + retardo;
            if (r > 0) { thread.sleep(r);

            }
        }
        catch (InterruptedException ie) { ie.printStackTrace();}




      }

  }
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

4 LOS APIS ORIENTADOS A JUEGOS EN MIDP 2.0

4.1 INTRODUCCIÓN

Una de las mejoras más importantes que presenta MIDP 2.0 es su API destinado al desarrollo de aplicaciones de entretenimiento. A través de este API, podemos desarrollar interfaces de usuario más rápidas, con mejor usabilidad y con un uso de los recursos más eficiente. El API de juegos también ayuda a reducir el tamaño del fichero *jar* final que se ha de distribuir y descargar al terminal. Con MIDP 1.0, los desarrolladores debían proveer sus propias rutinas de gestión de gráficos para realizar los procesos y esto provoca el aumento del tamaño de los *jar*, al tener que añadir estas clases en la distribución.

La idea principal del API de juegos y que supone la base para todo el sistema de gráficos, es que la pantalla del juego se compone de capas. Las pantallas a mostrar se componen a través de capas. Todas estas capas pueden ser manejadas por separado y el API se encarga de dibujar las distintas capas de forma adecuada. El área de trabajo puede ser mayor que el tamaño de la pantalla del dispositivo y el movimiento o *scroll* de esta área puede resultar costoso. El API de juegos proporciona una vista de una parte del área de trabajo que compone una pantalla del juego (no una pantalla en el dispositivo).

4.2 PAQUETEJAVAX.MICROEDITION.LCDUI.GAME

El API de juegos se encuentra en el paquete *javax.microedition.lcdui.game*. A parte de este paquete, tendremos otras APIs dentro de la especificación que nos serán muy útiles en el desarrollo de juegos, como el denominado *Mobile Media API* que nos permitirá trabajar con sonidos.

Como vemos, el paquete *javax.microedition.lcdui.game* está dentro de la parte *lcdui* del API, lo que nos indica que las clases hacen referencia al interfaz de usuario, en concreto, a la gestión de los elementos que se muestran en la pantalla del dispositivo y los procesos que permiten componer dichas pantallas. Dentro del paquete *game* tenemos cinco clases:

- *GameCanvas*
- *Layer*
- *LayerManager*
- *Sprite*
- *TiledLayer*

4.2.1 La clase GameCanvas

La clase *GameCanvas* es abstracta y proporciona la funcionalidad básica para la realización de la interfaz de usuario. Esta clase presenta dos beneficios sobre la versión básica de *Canvas*, de la cual hereda. Por una parte tiene un *buffer off-screen* implementado y permite recoger el estado de las teclas del dispositivo, es decir, comprobar si las teclas están pulsadas o están liberadas.

ISBN: 978-84-9012-861-9

Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Para cada instancia de *GameCanvas* tenemos un *buffer* dedicado, lo que nos permite tener una funcionalidad superior a la que proporciona *Canvas*, pero también debemos tener cuidado con este elemento, ya que el hecho de que cada instancia tenga un buffer dedicado, provoca que el uso masivo y sin consideraciones de objetos de este tipo, tenga un impacto importante sobre la memoria que consume nuestra aplicación. Es muy recomendable reutilizar los objetos de tipo *GameCanvas* para evitar que los requerimientos de memoria se disparen.

En un primer momento, al crear un objeto de tipo *GameCanvas*, todo el *buffer* sobre el que trabaja se llenará con *pixels* de color blanco. La clase *GameCanvas* provee una serie de constantes para averiguar qué tecla es pulsada por el usuario. El estado de las teclas, se puede recuperar a través del método *getKeyStates*, que retorna un entero en el que cada *bit* indica el estado de una de las teclas del terminal. Una ejemplo del uso de este método es el siguiente fragmento de código:

```
// Get the key state and store it

int keyState = getKeyStates();

if ((keyState & LEFT_PRESSED) != 0) { positionX--;

} else if ((keyState & RIGHT_PRESSED) != 0) {

    positionX++;

}
```

Otro método a destacar dentro de la clase *GameCanvas*, es el método *flushGraphics*. Este método, tiene dos versiones, una sin parámetros y otra con parámetros. Este método muestra el contenido del *buffer off-screen* en la pantalla del terminal. En la versión con parámetros, se especifica una parte del *buffer*, que será aquella zona del mismo que se mostrará por pantalla.

4.2.2 La clase Layer

La clase *Layer* es una clase abstracta que representa un elemento visual dentro del juego, es decir, en un juego típico de plataformas, cada uno de los dibujos que componen de las plataformas, los diferentes personajes del juego e incluso la imagen del fondo del mismo, serán objetos de tipo *Layer*. Las clases *Sprite* y *TiledLayer*, que veremos a continuación, heredan de la clase *Layer*.

Cada objeto de tipo *Layer* tiene una posición (determinada por unas coordenadas con respecto a la esquina superior-izquierda del objeto *Graphics* que recibe el método *paint*), un ancho y un alto y

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

puede ser establecida como visible o como invisible. Las subclases de *Layer* implementan el método *paint(Graphics)*, de tal forma que cualquiera de estos elementos puede ser mostrado por pantalla. Este método está definido como abstracto en la clase *Layer*.

Los métodos que contiene esta clase son básicamente los métodos de gestión de posición, las dimensiones y la visibilidad del objeto *Layer*.

4.2.3 La clase LayerManager

Los objetos de tipo *Layer* que tenemos en nuestro código son manejados por un gestor de capas, es decir, un objeto de tipo *LayerManager*. Esta clase se encarga de gestionar todo el proceso de pintado de las distintas capas (*Layers*) que componen cada pantalla del juego, dibujando en el orden adecuado los diferentes elementos. El gestor de capas conoce en todo momento las capas de las que debe encargarse. El índice de la capa dentro del gestor, indica su profundidad o posición en el eje z. Si el valor de este índice es 0, la capa será la más cercana al usuario y cuanto mayor sea el índice, mayor será la profundidad de la capa.

La clase *LayerManager* proporciona algunos métodos para el control de la visualización de la capas (*Layers*), es decir, para controlar qué es exactamente lo que se genera en la pantalla. Se puede controlar la región visible de la pantalla, lo que nos permite hacer movimientos laterales de la pantalla (*scroll*) de forma sencilla. Este efecto se denomina *ventana de visualización*, ya que la idea es exactamente esa, tenemos una ventana visible dentro de todo el gráfico que componen los diferentes objetos de tipo *Layer*.

El método *paint* de la clase *LayerManager* tiene dos parámetros de tipo entero, que indican la posición de la ventana de visualización de la que hablábamos en el párrafo anterior dentro de la pantalla del terminal. Las coordenadas de esta posición son relativas al objeto *Graphics* que recibe también como parámetro. Es decir, partiendo de una imagen compuesta de capas, de un tamaño superior al de la pantalla del terminal, definimos una región dentro de esta imagen que será la región que vamos a visualizar y además, indicamos también el punto dentro de la pantalla del terminal en la que se mostrará dicha región.

La clase *LayerManager* presenta una serie de métodos que nos permiten gestionar las diferentes capas que componen la escena, pudiendo añadir, eliminar e insertar en una determinada posición, las capas. A parte de estos métodos, cabe destacar el método *setViewWindow* que nos permitirá definir la ventana de visualización dentro la escena.

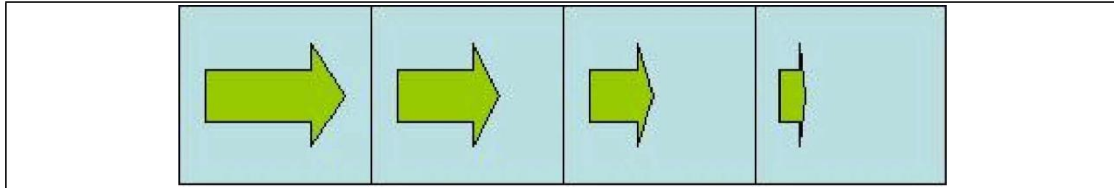
4.2.4 La clase Sprite

Los *sprites* se pueden definir como cada uno de los fotogramas de una animación para un personaje o elemento dentro de un juego. En MIDP 2.0, a partir de una única imagen se obtienen todos los *sprites* o fotogramas que componen un objeto animado. Esta imagen puede contener uno o más de estos fotogramas. En caso de que la imagen contenga más de un fotograma correspondiente a la animación de un objeto o elemento, estos fotogramas serán

ISBN: 978-84-9012-861-9

Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

todos iguales en cuanto al tamaño y por lo tanto se obtendrán dividiendo la imagen original en pedazos. Es decir, la imagen que contiene los fotogramas de puede concebir como la unión de dichos fotogramas.



En la imagen anterior, tenemos cuatro fotogramas de una animación en la que una flecha se encoge horizontalmente. Para extraer los fotogramas de esta imagen, simplemente dividimos la imagen en los cuadrados y todos deben tener las mismas dimensiones. El formato de la imagen puede variar y en lugar de tener la disposición de los fotogramas que tenemos en el ejemplo anterior, podemos agrupar las imágenes de forma vertical o en forma de tabla.

Los diferentes fotogramas son indexados a la hora de dividir la imagen. El fotograma más arriba y más a la izquierda será el correspondiente al índice 0 y el resto se irán numerando consecutivamente de izquierda a derecha y de arriba abajo. El orden de esta secuencia de fotogramas es el mismo orden en el que serán mostrados a la hora de realizar la animación. El programador puede modificar esta secuencia con métodos de la clase *Sprite*, de tal forma que usando los índices de los fotogramas, construya una secuencia propia de visualización. Si retomamos el ejemplo de la flecha que hemos anteriormente, tendríamos por defecto la secuencia 0,1,2,3. Esta secuencia de visualización de los fotogramas nos daría como resultado una animación en la que flecha encoge y desde el tamaño menor vuelve a pasar al más grande. Si queremos un resultado en el que la flecha crezca y encoja de forma lógica, deberíamos definir la secuencia 0,1,2,3,2,1,0.

La clase *Sprite* incluye lo que se denomina el *pixel* de referencia. El *pixel* de referencia es un punto del fotograma definido a través del método *defineReferencePixel*. Por defecto, este punto corresponde a las coordenadas (0,0), que será la esquina superior izquierda del fotograma. Hay que recordar que este punto es un punto del fotograma y no del sistema de coordenada global que define la pantalla del terminal. Este *pixel* se puede definir incluso fuera de los límites del fotograma. Con el método *setRefPixelPosition* podemos situar el *sprite* de tal forma que el *pixel* de referencia se sitúe en el punto indicado.

Otra de las funcionalidades útiles de la clase *Sprite* para manejar los fotogramas, es aquella que nos permite realizar transformaciones sobre el *sprite* original. El método para realizar estas transformaciones es el método *setTransform*, y las posibles transformaciones son:

- TRANS_NONE – Deja el *sprite* tal y como estaba originalmente

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

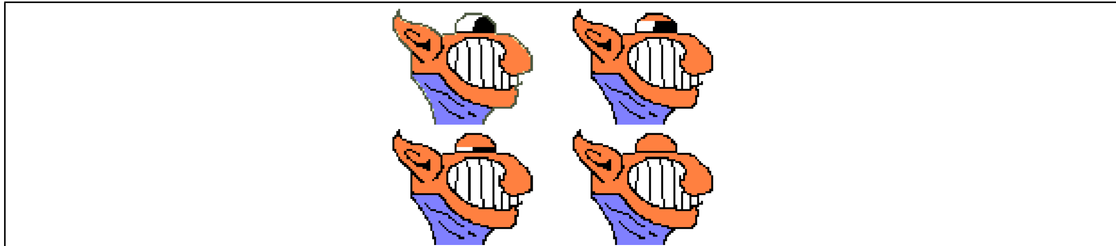
- TRANS_ROT180 – Gira la imagen 180 grados
- TRANS_MIRROR – Muestra el *sprite* como si se reflejara en un espejo
- TRANS_MIRROR_ROT180 – Refleja el *sprite* y lo gira 180 grados
- TRANS_ROT90 – Rota la imagen 90 grados
- TRANS_MIRROR_ROT90 – Refleja el *sprite* y lo gira 90 grados
- TRANS_ROT270 – Rota la imagen 270 grados
- TRANS_MIRROR_ROT270 – Refleja el *sprite* y lo gira 270 grados

Cuando aplicamos una transformación de estas a un *Sprite*, dicha transformación es aplicada sobre el fotograma original y siempre es aplicada con respecto al *pixel* de referencia, por lo tanto, el fotograma reposicionado, de tal forma que el *pixel* de referencia permanece inalterado con respecto al sistema de coordenadas global. Podemos considerar al *pixel* de referencia como el punto central de la transformación o el eje de giro de la transformación.

Para dibujar los *Sprites*, son los propios *Sprites* los que mantienen toda la información referente a la posición, el fotograma a visualizar... y simplemente tendremos que llamar al método *paint* de la clase *Sprite* para que el fotograma correspondiente del *sprite* sea dibujado.

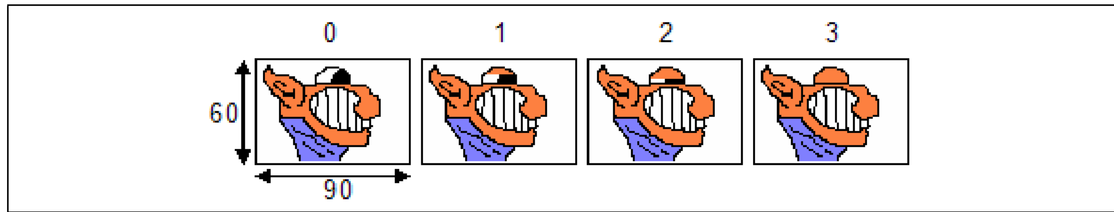
4.2.5 EJEMPLO DEL USO DE SPRITES

A continuación vamos a ver un pequeño ejemplo de cómo funciona la gestión de *sprites* en MIDP 2.0 y cómo se utilizan algunas de las funcionalidades que hemos visto hasta ahora. En primer lugar, partimos de una imagen *png* que contiene todos los fotogramas necesarios para implementar la animación de nuestro personaje. En nuestro caso, los fotogramas tienen unas dimensiones de 90x60 *pixels* y son cuatro fotogramas colocados en forma de tabla. El tamaño de la imagen contenedora será por tanto 180x120.



Como vemos, tenemos un personaje cuya animación en este caso consiste en el parpadeo de su único ojo. Al dividir esta imagen en fotogramas de 90x60, tendremos los siguientes fotogramas:

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc



Al dividir la imagen, como ya hemos indicado anteriormente, se asignará un número de secuencia para configurar la imagen, que irán del 0 al 3 en nuestro caso. Bien, si mantenemos esta secuencia inalterada tendremos que un efecto de parpadeo un poco raro, ya que pasamos del ojo totalmente cerrado al ojo totalmente abierto. Para solucionar esta situación, definiremos una nueva secuencia para los fotogramas, en la que el orden de visualización de los mismos sea 0,1,2,3,2,1,0.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

El siguiente código muestra estos detalles y algunos más que comentaremos a continuación dentro del código:

```
import javax.microedition.lcdui.*;

import javax.microedition.lcdui.game.*;

public class AnimaCiclope extends GameCanvas implements Runnable {

    // Imagen que contiene todos los sprites Image ciclopeimg = null;

    // Sprites que componen la animación Sprite sprt = null;

    // Gestor de capas

    LayerManager mng = new LayerManager(); private Thread

    thread;

    public AnimaCiclope() { super(false);

      // Cargamos la imagen contenedora try{

                    ciclopeimg = Image.createImage("/ciclope.png");

              }catch (Exception ex){ ex.printStackTrace();

              }

              // Extraemos los fotogramas de la imagen sprt = new Sprite(ciclo-

              peimg,90,60);
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                    // El punto de referencia es el centro del fotograma

      sprt.defineReferencePixel(45, 30);

      //  Definimos  al  secuencia  de  animación  int  seq[]  =  new  int[]

      {0,1,2,3,2,1,0}; sprt.setFrameSequence(seq);

      // Insertamos el sprite en gestor de capas mng.insert(sprt, 0);


    }

    protected void showNotify() { thread = new Thread(this);

      thread.start();

    }
    public void run() {

      // Recuperamos el buffer de la pantalla Graphics g = getGraphics();

Thread mythread = Thread.currentThread();

// Ponemos el pixel de referencia del sprite en el punto

// central de la pantalla. Así tenemos la animación en

// el centro de la patanlla del terminal sprt.setRefPixelPosition((int)(this.getWidth()/2),

(int)(this.getHeight()/2));
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
    while (mythread == thread) {

            // Borramos la pantalla del terminal g.setColor(0xFFFFFF); g.fillRect(0,0,this.getWidth(),this.get-

            Height());

            // Nos movemos al siguiente fotograma de la animación sprt.nextFrame();



            // Dibujamos el sprite sprt.paint(g);

            // Mostramos el buffer en la pantalla flushGraphics();

            try {

                mythread.sleep(100);

            } catch (java.lang.InterruptedException e) {

            }
    }

}
protected void keyPressed(int keyCode) {


    // Recuperamos la tecla que se ha pulsado

    // Modificamos la posición del sprite en función de la

    // tecla que se pulse

    int accion = getGameAction(keyCode);
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
switch (accion) {

        case Canvas.LEFT: sprt.setRefPixelPosition(

            sprt.getRefPixelX()-5,sprt.getRefPixelY()); break;

        case Canvas.RIGHT: sprt.setRefPixelPosition(

            sprt.getRefPixelX()+5,sprt.getRefPixelY());

            break;

        case Canvas.DOWN: sprt.setRefPixelPosition(

            sprt.getRefPixelX(),sprt.getRefPixelY()+5); break;

        case Canvas.UP: sprt.setRefPixelPosition(

            sprt.getRefPixelX(),sprt.getRefPixelY()-5); break;

        case Canvas.FIRE:

            // Cuando se pulsa la tecla de "fuego", rotamos

            // 90 grados la imagen sprt.setTransform(Sprite.TRANS_ROT90); break;

        default:

            return;
  }

  // Mostramos la nueva posición del pixel de referencia

  // dentro del sistema de coordenadas general System.out.println("("+sprt.getRefPixelX()+

              ","+sprt.getRefPixelY());

}

}
```

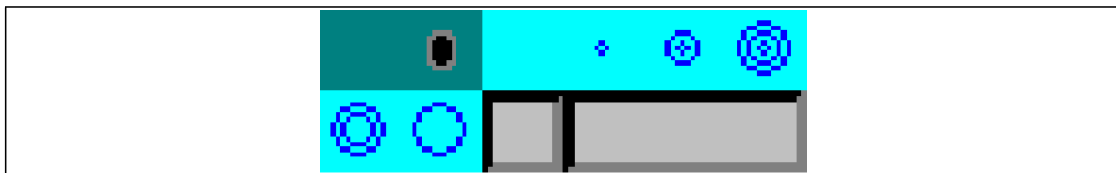

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

El código anterior nos permite comprobar el funcionamiento de algunos de los puntos de los que hemos hablado a lo largo de este capítulo. El anterior ejemplo muestra como extraer los fotogramas de la imagen y muestra como gestionar los diferentes elementos a través de un objeto *Layer-Manager*.

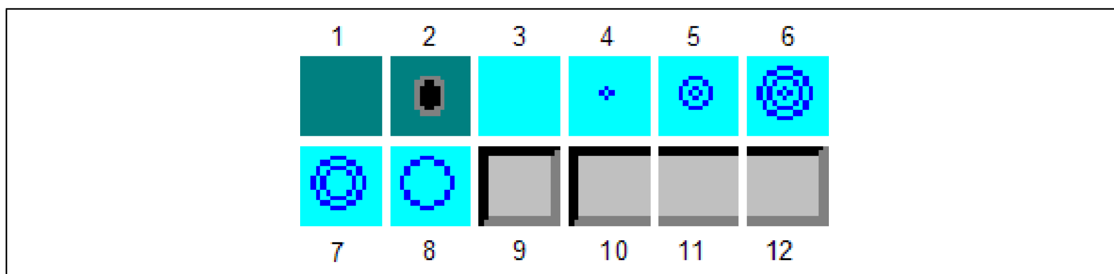
4.2.6 LA CLASE TILEDLAYER

Los objetos de tipo *TiledLayer*, representan un elemento visual compuesto por un grupo de celdas que se corresponden con una serie de imágenes. Esta clase nos permite crear dibujos a partir de pequeñas celdas, de tal forma que compondremos una imagen grande a partir de una o varias imágenes pequeñas. Se puede decir que el objetivo es crear un mosaico, es decir, a partir de pequeños elementos, repitiendo estos y colocándolos en las posiciones adecuadas conseguiremos una imagen mucho más compleja. Esta técnica es muy habitual dentro de los juegos de plataformas para conseguir escenarios complejos.

Los distintos elementos que componen el mosaico están contenidos en una única imagen, de forma similar a lo visto en el caso de los *Sprites*. Esta imagen es divida en pequeñas partes, de las mismas dimensiones cada una de las partes.

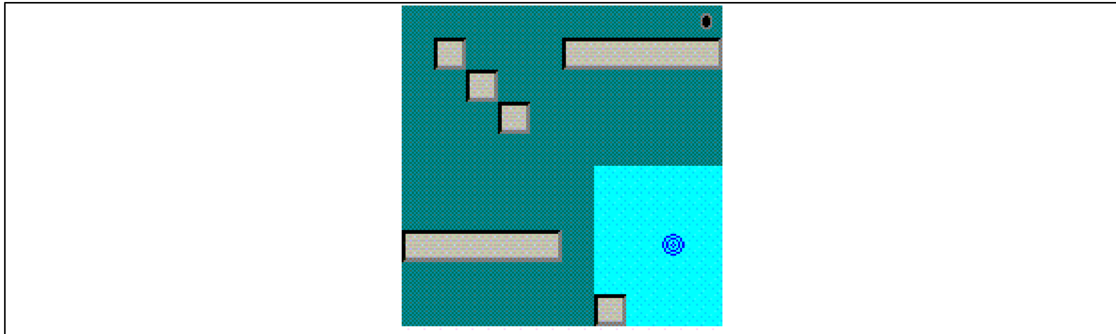


La imagen anterior contiene una serie de pequeñas imágenes que nos servirán para componer las pantallas de nuestro juego. En concreto, tiene 12 imágenes de 16 *pixels* de ancho por 16 *pixels* de alto. La siguiente figura muestra el resultado del proceso de separación de la imagen.



ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Al extraer los diferentes elementos de la imagen, se les asigna un índice automáticamente. Este índice comienza en 1. Con estas imágenes podemos componer una imagen como la siguiente, que podría corresponder a un juego de minigolf:



A la hora de componer la imagen final, también podemos usar lo que denominaremos *tile* (siguiendo con la metáfora del mosaico) o azulejo animado. Estos elementos no están presentes en la imagen original, pero se utilizan también para crear la pantalla. Los azulejos animados tendrán un determinado índice (siempre menor que 0) y son creados a través del método *createAnimatedTile*. Los azulejos virtuales son asociados dinámicamente en tiempo de ejecución con algún azulejo de los que extraídos de la imagen, y cambiando esta asociación tendremos una animación. La asociación entre el azulejo animado virtual y el azulejo estático se hace con el método *setAnimatedTile*. En la figura anterior, tenemos unos círculos dentro del lago. Esto es un elemento animado, que a través de las imágenes 4,5,6,7 y 8 extraídas de la imagen original, crea un efecto de ondas en el agua. En el ejemplo siguiente vemos el proceso de construcción de un escenario a partir de las imágenes contenidas en un fichero. En concreto, vamos a construir el escenario del minigolf que tenemos en la última figura.

```
import javax.microedition.lcdui.*;

import javax.microedition.lcdui.game.*;

public class Campo extends GameCanvas implements Runnable {

    // Imagen que contiene todos los azulejos o tiles Image campoimg = null;

    // Tiles o azulejos con los que vamos a trabajar TiledLayer tiles = null;

    // Gestor de capas

    LayerManager mng = new LayerManager(); Thread thread;
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
// Variable usada para realizar la animación de la onda en

// el agua. Indica el tile a dibujar en cada momento int gota;



public Campo() { super(false);

  // Cargamos la imagen contenedora try{

                          campoimg = Image.createImage("/campo.png");

              }catch (Exception ex){ ex.printStackTrace();

              }

              // Extraemos los tiles de la imagen y creamos una rejilla

              // de 10x10 que será sobre la que compongamos la imagen tiles = new TiledLayer(10,10,cam-

  poimg,16,16);

              // Creamos el tile animado para ponerlo sobre el lago

              // y simular el efecto de onda sobre el agua gota = tiles.createAnimatedTile(3);



              // Rellenamos todo el escenario con el azulejo verde for (int i=0;i<10;i++){

              for (int j=0;j<10;j++){ tiles.setCell(i,j,1);

              }
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
}

            // Situamos algunos azulejos sobre la rejilla para componer

            // el escenario ti-

les.setCell(9,0,2); tiles.setCell(9,1,12);

tiles.setCell(8,1,11); ti-

les.setCell(7,1,11); tiles.setCell(6,1,11);

tiles.setCell(5,1,10); tiles.setCell(1,1,9);

tiles.setCell(2,2,9); tiles.setCell(3,3,9);

tiles.setCell(0,7,10); ti-

les.setCell(1,7,11); tiles.setCell(2,7,11);

tiles.setCell(3,7,11); ti-

les.setCell(4,7,12);

            // Creamos el lago

for (int i=6;i<10;i++){

            for (int j=5;j<10;j++){ tiles.setCell(i,j,3);

            }

}

tiles.setCell(6,9,9);
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                // Añadimos el tile animado en la mitad del lago
    tiles.setCell(8,7,gota);

    // Insertamos la rejilla dentro del gestor de capas mng.insert(tiles, 0);

  }

  protected void showNotify() { thread = new Thread(this);
    thread.start();
  }

  public void run() {
    // Recuperamos el buffer de la pantalla Graphics g = getGraphics();
Thread mythread = Thread.currentThread(); int indiceGota = 3;

    while (mythread == thread) {
                // Borramos la pantalla del terminal g.setColor(0xFFFFFF); g.fill-
                Rect(0,0,this.getWidth(),this.getHeight());
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                // Calculamos el tile estático a mostrar

                // dentro del tile animado if (indiceGota == 3){

                        indiceGota=4;

                }else if (indiceGota == 4){ indiceGota=5;

                }else if (indiceGota == 5){ indiceGota=6;

                }else if (indiceGota == 6){ indiceGota=7;

                }else if (indiceGota == 7){ indiceGota=8;

                }else if (indiceGota == 8){

                        indiceGota=3;

                }

                // Dibujamos el tile animado, con el elemento adecuado tiles.setAnimatedTile(gota, indiceGota);



                tile.paint(g);

                // Mostramos el buffer en la pantalla flushGraphics();

                try {

                            mythread.sleep(500);
                  } catch (java.lang.InterruptedException e) {

                  }

        }

    }

}
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

5 APÉNDICE A – COMPILACIÓN CON ANT Y ANTENNA

Como sabemos, tenemos a nuestra disposición ciertas herramientas más o menos avanzadas que nos permiten realizar el proceso de compilación. Cuando hablamos de J2ME y de *midlets*, el proceso de compilación cubre más etapas que la propia compilación, ya que primero debemos realizar la compilación propiamente dicha, luego preverificar las clases y por último crear los ficheros *jar* y *jad*, para distribuir la aplicación. Una vez hecho todo este proceso manualmente para comprender los diferentes pasos y tener claro todo el proceso, seremos capaces de comprender mejor el funcionamiento de estas herramientas y solucionar posibles problemas que se nos presenten.

Ant es una herramienta de compilación avanzada, realizada en java, que permite automatizar los procesos de compilación, empaquetado y preparación para la distribución de las aplicaciones. *Ant* se basa en ficheros XML de configuración que permiten definir todas las tareas a realizar, así como las dependencias entre las distintas tareas. El fichero de configuración de *Ant*, tiene el nombre *build.xml*. Como todos los ficheros XML, este fichero contiene una arquitectura de etiquetas, que definen los valores de los distintos elementos del fichero.

El elemento *project* del fichero XML debe estar presente en el fichero al menos una vez y es el elemento más general para un determinado proyecto, que se compondrá de una serie de tareas. El elemento *project* tiene un atributo denominado *default* que indica la tarea (*target*) a realizar cuando no se especifica ninguna otra tarea en concreto. El atributo *basedir* del elemento *project* define el camino a partir del cual se deben tomar el resto de caminos (*path*) definidos en el fichero.

```
<project name="midlet1" default="run" basedir=".">

    <target name="compilar">

            <javac scrdir="${src}" destdir="${classes}"

                    bootclasspath="${classpath}">

            </javac>

    </target>
</project>
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

En el ejemplo anterior tenemos un proyecto con una única tarea, la tarea “compilar”. Esta situación no es común dentro de los ficheros *build.xml* de *Ant*, ya que lo habitual es tener una serie de tareas. En el caso de J2ME, esta situación es aún más anómala, ya que es inútil, porque como hemos dicho anteriormente, en J2ME necesitamos realizar varias tareas antes de tener la aplicación lista para su distribución. El ejemplo anterior, por lo tanto, se puede extender de la siguiente forma para que tenga cierta utilidad:

```
<project basedir="." default="empaquetar" name="MIDlet1">

    <property name="wtk_dir" value="c:/wtk104"/>

    <property name="clases" value="classes"/>

    <property name="fuentes" value="src"/>

    <property name="recursos" value="res"/>

    <property name="distribucion" value="dist"/>

    <property name="classpath" value="${wtk_dir}/lib/midpapi.zip"/>

    <property name="nombreDist" value="Midlet1"/>

    <target depends="init" name="compilar">

        <javac srcdir="${fuentes}" destdir="${clases}" bootclasspath="${classpath}" list-

                    files="true">

        </javac>

    </target>

    <target name="gestion_recursos">

        <echo message="Gestión de recursos" />
        <copy todir="${clases}">
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
            <fileset dir="${recursos}">
                <include name="**/*.png"/>
            </fileset>
        </copy>
    </target>

    <target name="preverificar" depends="compilar">
        <exec executable="${wtk_dir}/bin/preverify.exe">
            <arg line="-classpath ${classpath}"/>
            <arg line="-d ${clases}"/>
        </exec>
    </target>



    <target name="empaquetar" depends="preverificar,gestion_recursos">
        <jar destfile="${distribucion}/${nombreDist}.jar" manifest="MANIFEST.MF">
                <fileset dir="${clases}"/>
             </jar>
          <copy file="${nombreDist}.jad" tofile="${distribucion}/${nombreDist}.jad" />
    </target>

    <target name="ejecutar" depends="empaquetar">
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        <exec dir="${distribucion}"
                        executable="${wtk_dir}/bin/emulator">
            <arg line="-classpath ${nombreDist}.jar"/>
            <arg line="-Xdescriptor:${nombreDist}.jad"/>
        </exec>
    </target>


    <target name="init">
        <echo message="Limpiando...." />
        <delete dir="${clases}" />
        <delete dir="${distribucion}" />


        <mkdir dir="${clases}" />
        <mkdir dir="${distribucion}" />
    </target>


</project>
```

Con un fichero como este colocado en el directorio raiz del proyecto e invocando el comando *ant*, tendremos automatizado todo el proceso de compilación, empaquetamiento... del proyecto. Dentro del ejemplo podemos diferenciar tareas como compilar, en el que usamos ciertos elementos estándar de la herramienta (*javac*) de otras tareas como preverificar, que son propias de J2ME y deben ser indicadas más rudimentariamente, como tareas con ejecución de comandos.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

Si bien el uso de *Ant* supone una gran ayuda durante el proceso de desarrollo, la versión general que acabamos de ver, puede ser ampliada con nuevas tareas especificas para J2ME y que nos facilitará aún más la creación de *midlets*. El proyecto *Antenna* es desarrollado para extender *Ant* y solucionar este problema.

El fichero *build.xml* anterior, modificado para el uso de las tareas proporcionadas por el proyecto *Antenna*, será el siguiente:

```
<project basedir="." default="empaquetar" name="MIDlet1">

<!-- Información del midlet -->

  <property name="midlet" value="midlet1"/>

  <property name="nombreMidlet" value="Midlet1"/>

  <property name="claseMidlet" value="com.vitike.libroj2me.midlet1.MIDlet1"/>

  <property name="iconoMidlet" value=""/>

  <property name="perfil" value="1.0"/>

  <property name="configuracion" value="1.0"/>

  <property name="wtk_dir" value="c:/wtk104"/>

  <property name="clases" value="classes"/>

  <property name="fuentes" value="src"/>

  <property name="recursos" value="res"/>

  <property name="distribucion" value="dist"/>

  <property name="classpath" value="${wtk_dir}/lib/midpapi.zip"/>

  <!-- Configuración de las tareas de Antenna -->

  <property name="wtk.home" value="${wtk_dir}"/>

  <taskdef name="wtkjad" classname="de.pleumann.antenna.WtkJad"/>

  <taskdef name="wtkbuild"
                              classname="de.pleumann.antenna.WtkBuild"/>

  <taskdef name="wtkpackage"
                              classname="de.pleumann.antenna.WtkPackage"/>
```

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
<taskdef name="wtkmakeprc"

                    classname="de.pleumann.antenna.WtkMakePrc"/>

<taskdef name="wtkrun" classname="de.pleumann.antenna.WtkRun"/>

<taskdef name="wtkpreverify"

                    classname="de.pleumann.antenna.WtkPreverify"/>

<taskdef name="wtkobfuscate"

                    classname="de.pleumann.antenna.WtkObfuscate"/>



<target depends="init" name="compilar">

    <wtkbuild srcdir="${fuentes}" destdir="${clases}" preverify="true">

    </wtkbuild>

</target>

<target name="empaquetar" depends="compilar">

    <wtkjad jadfile="${distribucion}/${midlet}.jad" jarfile="${distribucion}/${midlet}.jar"

                    name="Ejemplos del libro de j2me" vendor="Manuel J. Prieto">



        <!-- Definición de atributos propios para el jad-->

        <attribute name="Propiedad" value="Valor de propiedad"/>



        <midlet name="${nombreMidlet}"
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
                        icon="${iconoMidlet}"

                        class="${claseMidlet}">

        </midlet>

      </wtkjad>

      <wtkpackage basedir="${clases}"

                              jarfile="${distribucion}/${midlet}.jar" jadfile="${distribucion}/${mid-

                              let}.jad" config="${configuracion}" profile="${perfil}">

              <fileset dir="${recursos}"/>

        </wtkpackage>



        <copy file="${midlet}.jad" tofile="${distribucion}/${midlet}.jad"/>

</target>


<target name="ejecutar" depends="empaquetar">

    <wtkrun jadfile="${distribucion}/${midlet}.jad" />

</target>



<target name="init">

    <echo message="Limpiando...." />

    <delete dir="${clases}" />
      <delete dir="${distribucion}" />
```


ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

```
        <mkdir dir="${clases}" />

        <mkdir dir="${distribucion}" />

    </target>

</project>
```

Se puede ver que las tareas nuevas, facilitan el proceso de construcción de los ficheros *jad* y *jar*. Un ejemplo de esta nueva situación, es el hecho de que el fichero *jad* es actualizado automáticamente con el tamaño del fichero *jar*, por lo que no debemos preocuparnos más por este valor. En el ejemplo anterior, vemos cómo añadir atributos propios al fichero *jad* (inserción del campo *propiedad* en el ejemplo anterior), que nos serán muy útiles como valores de configuración de los *midlets*.

6 APÉNDICE B – LUGARES EN INTERNET DE REFERENCIA

- wireless.java.sun.com – Página oficial de la compañía Sun Microsystems dedicada a J2ME y todo lo relacionado con dicha tecnología. Este es el lugar esencial para mantenerse informado sobre las nuevas especificaciones dentro de J2ME. También son de gran utilidad lo artículos que se publican en dicha página y los denominadas *Tech Tip*. Algunos de los artículos publicados en esta página son la base de conocimiento de algunos fragmentos de este documento. En este web también se pueden descargar las herramientas necesarias para el desarrollo de aplicaciones J2ME.
- www.nokia.com – La parte de la web oficial de la compañía Nokia dedicada a los creadores de aplicaciones, contiene una gran cantidad de información de una altísima calidad sobre la mayoría de las tecnologías móviles en el mercado. Por supuesto, hay una zona dedicada a J2ME. Destacable dentro de esta web son los foros de usuarios y los artículos y guías desarrolladas por Nokia y publicadas en dicha web. Algunos de estos artículos, como se ha mencionado anteriormente, son punto de referencia de este libro.
- www.j2me.org – Esta dirección contiene una serie de foros dedicados a tratar temas relacionados con J2ME. Uno de estos foros será especialmente interesante en nuestro caso, es el caso del foro dedicado al desarrollo de juegos con J2ME.
- www.microjava.com – Página web con un gran número de secciones dedicadas en exclusiva a J2ME. Cabe destacar la sección dedicada a los desarrolladores y la sección dedicada a las descargas, donde podemos encontrar desde aplicaciones completas hasta APIs concretas que usar en nuestros desarrollos.

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

References

1. Adriana Fernández-Fernández, Cristina Cervelló-Pastor, Leonardo Ochoa-Aday (2016). Energy-Aware Routing in Multiple Domains Software-Defined Networks. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 5, n. 3
2. Alberto Fernández-Isabel, Rubén Fuentes-Fernández (2015). Simulation of Road Traffic Applying Model-Driven Engineering. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 4, n. 2
3. Amir Hosein Keyhanipour, Behzad Moshiri (2013). Designing a Web Spam Classifier Based on Feature Fusion in the Layered Multi-Population Genetic Programming Framework. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 3
4. Ana Oliveira Alves, Bernardete Ribeiro (2015). Consensus-based Approach for Keyword Extraction from Urban Events Collections. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 4, n. 2
5. Ángel Martín del Rey, F. K. Batista, A. Queiruga Dios (2017). Malware propagation in Wireless Sensor Networks: global models vs Individual-based models. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 6, n. 3
6. Anna Závodská, Veronika Šramová, Anne-Maria AHO (2012). Knowledge in Value Creation Process for Increasing Competitive Advantage. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 1, n. 3
7. Antonio Pinto, Ricardo Costa (2016). Hash-chain-based authentication for IoT. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 5, n. 4
8. Casado-Vara, R., & Corchado, J. (2019). Distributed e-health wide-world accounting ledger via blockchain. Journal of Intelligent & Fuzzy Systems, 36(3), 2381-2386.
9. Casado-Vara, R., Chamoso, P., De la Prieta, F., Prieto J., & Corchado J.M. (2019). Non-linear adaptive closed-loop control system for improved efficiency in IoT-blockchain management. Information Fusion.
10. Casado-Vara, R., de la Prieta, F., Prieto, J., & Corchado, J. M. (2018, November). Blockchain framework for IoT data quality via edge computing. In Proceedings of the 1st Workshop on Blockchain-enabled Networked Sensor Systems (pp. 19-24). ACM.
11. Casado-Vara, R., Novais, P., Gil, A. B., Prieto, J., & Corchado, J. M. (2019). Distributed continuous-time fault estimation control for multiple devices in IoT networks. IEEE Access.
12. Casado-Vara, R., Vale, Z., Prieto, J., & Corchado, J. (2018). Fault-tolerant temperature control algorithm for IoT networks in smart buildings. Energies, 11(12), 3430.
13. Casado-Vara, R., Prieto-Castrillo, F., & Corchado, J. M. (2018). A game theory approach for cooperative control to improve data quality and false data detection in WSN. International Journal of Robust and Nonlinear Control, 28(16), 5087-5102.
14. Céline Ehrwein Nihan (2013). Healthier? More Efficient? Fairer? An Overview of the Main Ethical Issues Raised by the Use of Ubicomp in the Workplace. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 1
15. Chamoso, P., González-Briones, A., Rivas, A., De La Prieta, F., & Corchado J.M. (2019). Social computing in currency exchange. Knowledge and Information Systems.
16. Chamoso, P., González-Briones, A., Rivas, A., De La Prieta, F., & Corchado, J. M. (2019). Social computing in currency exchange. Knowledge and Information Systems, 1-21.
17. Chamoso, P., González-Briones, A., Rodríguez, S., & Corchado, J. M. (2018). Tendencies of technologies and platforms in smart cities: A state-of-the-art review. Wireless Communications and Mobile Computing, 2018.
18. Chamoso, P., Rodríguez, S., de la Prieta, F., & Bajo, J. (2018). Classification of retinal vessels using a collaborative agent-based architecture. AI Communications, (Preprint), 1-18.
19. Corchado, J. A., Aiken, J., Corchado, E. S., Lefevre, N., & Smyth, T. (2004). Quantifying the Ocean's CO2 budget with a CoHeL-IBR system. In Advances in Case-Based Reasoning, Proceedings (Vol. 3155, pp. 533–546).
20. Corchado, J. M., & Fyfe, C. (1999). Unsupervised neural method for temperature forecasting. Artificial Intelligence in Engineering, 13(4), 351–357. https://doi.org/10.1016/S0954-1810(99)00007-2

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

21. Corchado, J. M., Borrajo, M. L., Pellicer, M. A., & Yáñez, J. C. (2004). Neuro-symbolic System for Business Internal Control. In Industrial Conference on Data Mining (pp. 1–10). https://doi.org/10.1007/978-3-540-30185-1_1
22. Corchado, J. M., Pavón, J., Corchado, E. S., & Castillo, L. F. (2004). Development of CBR-BDI agents: A tourist guide application. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 3155, pp. 547–559). https://doi.org/10.1007/978-3-540-28631-8
23. Corchado, J., Fyfe, C., & Lees, B. (1998). Unsupervised learning for financial forecasting. In Proceedings of the IEEE/IAFE/INFORMS 1998 Conference on Computational Intelligence for Financial Engineering (CIFEr) (Cat. No.98TH8367) (pp. 259–263). https://doi.org/10.1109/CIFER.1998.690316
24. Costa, Â., Novais, P., Corchado, J. M., & Neves, J. (2012). Increased performance and better patient attendance in an hospital with the use of smart agendas. Logic Journal of the IGPL, 20(4), 689–698. https://doi.org/10.1093/jigpal/jzr021
25. Ester Martinez-Martin, Maria Teresa Escrig, Angel P. Del POBIL (2013). A Qualitative Acceleration Model Based on Intervals. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 2
26. Felicitas Mokom, Ziad Kobti (2013). Interventions via Social Influence for Emergent Suboptimal Restraint Use. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 2
27. Fyfe, C., & Corchado, J. M. (2001). Automating the construction of CBR systems using kernel methods. International Journal of Intelligent Systems, 16(4), 571–586. https://doi.org/10.1002/int.1024
28. Gonzalez-Briones, A., Chamoso, P., De La Prieta, F., Demazeau, Y., & Corchado, J. M. (2018). Agreement Technologies for Energy Optimization at Home. Sensors (Basel), 18(5), 1633-1633. doi:10.3390/s18051633
29. González-Briones, A., Chamoso, P., Yoe, H., & Corchado, J. M. (2018). GreenVMAS: virtual organization-based platform for heating greenhouses using waste energy from power plants. Sensors, 18(3), 861.
30. Gonzalez-Briones, A., Prieto, J., De La Prieta, F., Herrera-Viedma, E., & Corchado, J. M. (2018). Energy Optimization Using a Case-Based Reasoning Strategy. Sensors (Basel), 18(3), 865-865. doi:10.3390/s18030865
31. Heli Koskimaki, Pekka Siirtola (2016). Accelerometer vs. Electromyogram in Activity Recognition. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 5, n. 3
32. K. S. Jasmine, Gavani Prathviraj S., P Ijantakar Rajashekar, K. A. Sumithra Devi (2013). Inference in Belief Network using Logic Sampling and Likelihood Weighing algorithms. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 3
33. Karel Macek, Jiri Rojicek, Georgios Kontes, Dimitrios V. Rovas (2013). Black-Box Optimization for Buildings and Its Enhancement by Advanced Communication Infrastructure. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 2
34. Laza, R., Pavn, R., & Corchado, J. M. (2004). A reasoning model for CBR_BDI agents using an adaptable fuzzy inference system. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 3040, pp. 96–106). Springer, Berlin, Heidelberg.
35. Li, T., Sun, S., Corchado, J. M., & Siyau, M. F. (2014). Random finite set-based Bayesian filters using magnitude-adaptive target birth intensity. In FUSION 2014 - 17th International Conference on Information Fusion. Retrieved from https://www.scopus.com/inward/record.uri?eid=2-s2.0-84910637788&partnerID=40&md5=bd8602d6146b014266cf07dc35a681e0
36. Marisol García-Valls (2016). Prototyping low-cost and flexible vehicle diagnostic systems. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 5, n. 4
37. Méndez, J. R., Fdez-Riverola, F., Díaz, F., Iglesias, E. L., & Corchado, J. M. (2006). A comparative performance study of feature selection methods for the anti-spam filtering domain. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 4065 LNAI, 106–120. Retrieved from https://www.scopus.com/inward/record.uri?eid=2-s2.0-33746435792&partnerID=40&md5=25345ac884f61c182680241828d448c5
38. Muñoz, M., Rodríguez, M., Rodríguez, M. E., & Rodríguez, S. (2012). Genetic evaluation of the class III dentofacial in rural and urban Spanish population by AI techniques. Advances in Intelligent and Soft Computing (Vol. 151 AISC). https://doi.org/10.1007/978-3-642-28765-7_49

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241
Ediciones Universidad de Salamanca - cc by nc dc

39. Nadia Alam, Munira Sultana, M.S. Alam, M. A. Al-Mamun, M. A. Hossain (2013). Optimal Intermittent Dose Schedules for Chemotherapy Using Genetic Algorithm. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 2
40. Naoufel Khayati, Wided Lejouad-Chaari (2013). A Distributed and Collaborative Intelligent System for Medical Diagnosis. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 2
41. Pablo Chamoso, Fernando De La Prieta (2015). Swarm-Based Smart City Platform: A Traffic Application. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 4, n. 2
42. Pérez, A., Chamoso, P., Parra, V., & Sánchez, A. J. (2014). Ground Vehicle Detection Through Aerial Images Taken by a UAV. In Information Fusion (FUSION), 2014 17th International Conference on.
43. Prieto, J., Mazuelas, S., Bahillo, A., Fernandez, P., Lorenzo, R. M., & Abril, E. J. (2012). Adaptive data fusion for wireless localization in harsh environments. IEEE Transactions on Signal Processing, 60(4), 1585–1596.
44. Prieto, J., Mazuelas, S., Bahillo, A., Fernández, P., Lorenzo, R. M., & Abril, E. J. (2013). Accurate and Robust Localization in Harsh Environments Based on V2I Communication. In Vehicular Technologies - Deployment and Applications. INTECH Open Access Publisher.
45. Rodriguez-Fernandez J., Pinto T., Silva F., Praça I., Vale Z., Corchado J.M. (2018) Reputation Computational Model to Support Electricity Market Players Energy Contracts Negotiation. In: Bajo J. et al. (eds) Highlights of Practical Applications of Agents, Multi-Agent Systems, and Complexity: The PAAMS Collection. PAAMS 2018. Communications in Computer and Information Science, vol 887. Springer, Cham
46. Rodríguez, S., De La Prieta, F., Tapia, D. I., & Corchado, J. M. (2010). Agents and computer vision for processing stereoscopic images. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 6077 LNAI). https://doi.org/10.1007/978-3-642-13803-4_12
47. Rodríguez, S., Gil, O., De La Prieta, F., Zato, C., Corchado, J. M., Vega, P., & Francisco, M. (2010). People detection and stereoscopic analysis using MAS. In INES 2010 - 14th International Conference on Intelligent Engineering Systems, Proceedings. https://doi.org/10.1109/INES.2010.5483855
48. Rodríguez, S., Tapia, D. I., Sanz, E., Zato, C., De La Prieta, F., & Gil, O. (2010). Cloud computing integrated into service-oriented multi-agent architecture. IFIP Advances in Information and Communication Technology (Vol. 322 AICT). https://doi.org/10.1007/978-3-642-14341-0_29
49. Saadi Bin Ahmad Kamaruddin, Nor Azura Md Ghanib, Choong-Yeun Liong, Abdul Aziz Jemain (2012). Firearm Classification using Neural Networks on Ring of Firing Pin Impression Images. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 1, n. 3
50. Sandrine Mouysset, Ronan Guivarch, Joseph Noailles, Daniel Ruiz (2013). Segmentation of cDNA Microarray Images using Parallel Spectral Clustering. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 1
51. Sebastián Romero, Habib Moussa Fardoun, Victor Manuel Ruiz Penichet, José Antonio Gallud (2013). Tweacher: New proposal for Online Social Networks Impact in Secondary Education. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 1
52. Sérgio Matos, Hugo Araújo, José Luís Oliveira (2013). Biomedical Literature Exploration through Latent Semantics. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 2
53. Sumit Goyal, Gyanendra Kumar Goyal (2013). Machine Learning ANN Models for Predicting Sensory Quality of Roasted Coffee Flavoured Sterilized Drink. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 3
54. Tiago Oliveira, José Neves, Paulo Novais (2012). Guideline Formalization and Knowledge Representation for Clinical Decision Support. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 1, n. 2
55. Tiancheng Li, Shudong Sun (2013). Online Adapting the Magnitude of Target Birth Intensity in the PHD Filter. ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal (ISSN: 2255-2863), Salamanca, v. 2, n. 4

ISBN: 978-84-9012-861-9
Colección Aquilafuente: AQ-241

Ediciones Universidad de Salamanca - cc by nc dc