Juegos

Índice

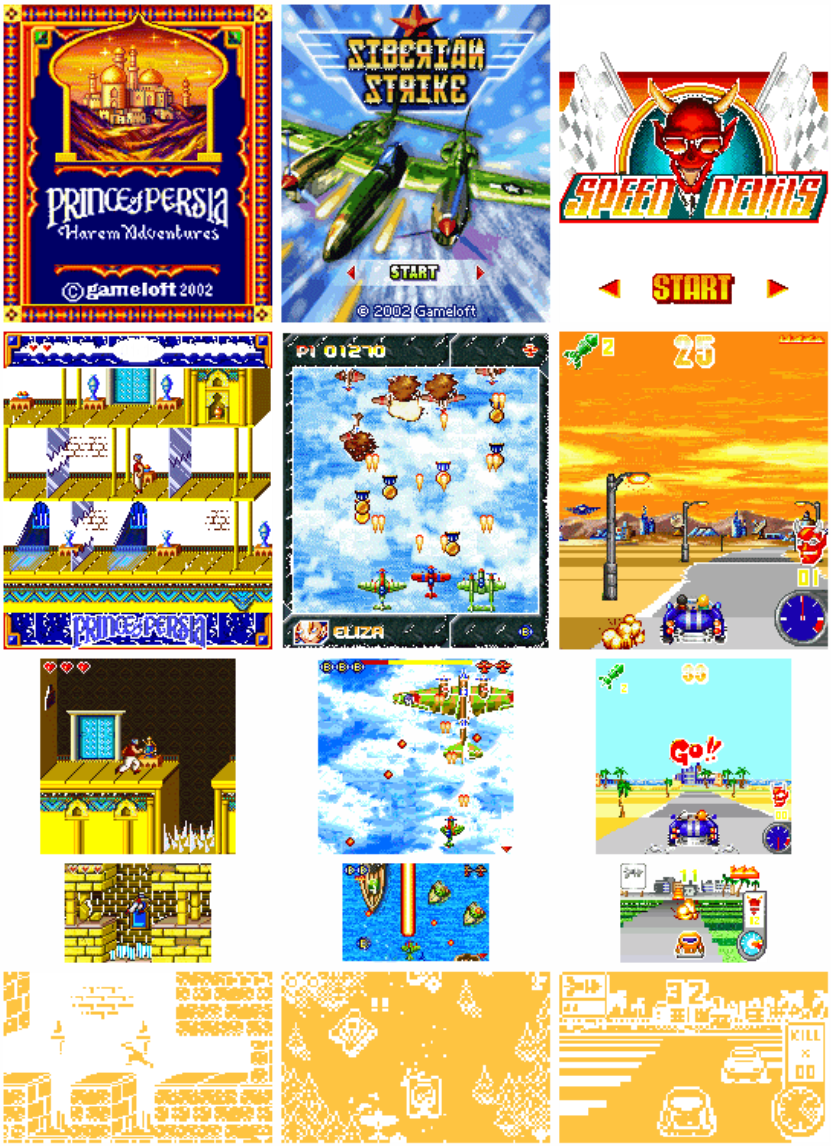
Sin duda una de las aplicaciones que más famosas se han hecho con el surgimiento de los teléfonos móviles MIDP son los juegos Java. Con estos teléfonos los usuarios pueden descargar estos juegos de Internet, normalmente previo pago de una tarifa, e instalarlos en el teléfono. De esta forma podrán añadir fácilmente al móvil cualquier juego realizado en Java, sin limitarse así a tener únicamente el típico juego de "la serpiente" y similares que vienen preinstalados en determinados teléfonos.

Vamos a ver en esta sección los conceptos básicos de la programación de videojuegos y las APIs de MIDP dedicadas a esta tarea. Comenzaremos viendo una introducción a los tipos de videojuegos que normalmente encontraremos en los teléfonos móviles.

1. Tipos de juegos

Podemos distinguir diferentes tipos de juegos:

- **Juegos de mesa**: Podemos encontrar juegos de mesa como por ejemplo juegos de cartas, ajedrez, reversi, etc. Estos juegos suelen ser muy sencillos, y la velocidad con la que se ejecuten no es crítica como ocurre en el caso de los juegos de acción.
- **Puzzles**: Son juegos de inteligencia, normalmente con una mecánica bastante sencilla. Lo fundamental en estos juegos es superar las pruebas propuestas, sin necesitar unos gráficos complejos ni tener ninguna componente de acción.
- **Juegos de acción 2D**: Consideremos juegos de acción 2D aquellos juegos en los que debemos manejar a un personaje u objeto. Podemos encontrar multitud de géneros dentro de este tipo de juegos: *shoot'em ups*, simuladores, plataformas, lucha, etc.
- **Juegos de acción 3D**: Con la evolución de los procesadores gráficos los juegos de acción han ido pasando de desarrollarse con gráficos 2D a desarrollarse con gráficos en 3D. Podemos encontrar prácticamente los mismos géneros que en el caso de los juegos 2D. Este tipo de juegos todavía es demasiado complejo para los modelos de móviles actuales, por lo que no lo tendremos en consideración.



Capturas de juegos para diferentes modelos de móviles

Los primeros juegos que podíamos encontrar en los móviles eran normalmente juegos muy sencillos tipo puzzle o de mesa, o en todo caso juegos de acción muy simples similares a los primeros videojuegos aparecidos antes de los 80. En los móviles con soporte para Java podremos tener juegos más complejos, como los que se podían ver en los ordenadores y consolas de 8 bits, y estos juegos irán mejorando conforme evolucionen los teléfonos móviles.

Además tenemos las ventajas de que existe una gran comunidad de programadores en Java, a los que no les costará aprender a desarrollar este tipo de juegos para móviles, por lo que el número de juegos disponible crecerá rápidamente. El poder descargar y añadir

estos juegos al móvil de forma sencilla, como cualquier otra aplicación Java, hará estos juegos especialmente atractivos para los usuarios, ya que de esta forma podrán estar disponiendo continuamente de nuevos juegos en su móvil.

Los juegos que se ejecutan en un móvil tendrán distintas características que los juegos para ordenador o videoconsolas, debido a las peculiaridades de estos dispositivos.

- **Escasa memoria**. No podremos crear demasiados objetos. Además el tamaño del JAR con todos los datos del juegos también suele estar limitado, muchas veces deberemos hacer un juego en 64KB. Esto nos obligará a rescatar viejas técnicas de programación de videojuegos de los tiempos de los 8 bits a mediados/finales de los 80.
- **CPU lenta**. La CPU de los móviles es bastante lenta comparada con la de los ordenadores de sobremesa y las videoconsolas. Es importante que los juegos vayan de forma fluida, por lo que antes de distribuir nuestra aplicación deberemos probarla en móviles reales para asegurarnos de que funcione bien, ya que muchas veces los emuladores funcionarán a velocidades distintas.
- **Pantalla reducida**. Deberemos tener esto en cuenta en los juegos, y hacer que todos los objetos se vean correctamente. Podemos utilizar *zoom* en determinadas zonas para poder visualizar mejor los objetos de la escena.
- **Almacenamiento limitado**. En el móvil el espacio para guardar datos sobre la partida también está bastante limitado. Será interesante permitir guardar la partida, para que el usuario puede continuar más adelante donde se quedó. Esto es especialmente importante en los móviles, ya que muchas veces se utilizan estos juegos mientras el usuario viaja en autobús, o está esperando, de forma que puede tener que finalizar la partida en cualquier momento. Deberemos hacer esto utilizando la mínima cantidad de espacio posible.
- **Poco ancho de banda**. Si desarrollamos juegos en red deberemos tener en cuenta la baja velocidad y la latencia de la red. Deberemos minimizar el tráfico que circula por la red.
- **Teclado pequeño**. El teclado de los móviles es muy pequeño y si hacemos que deban utilizarse muchas teclas puede resultar incómodo y complicado de manejar. Deberemos intentar proporcionar un manejo cómodo, haciendo que el control sea lo más sencillo posible, con un número reducido de posibles acciones.
- **Posibles interrupciones**. En el móvil es muy probable que se produzca una interrupción involuntaria de la partida, por ejemplo cuando recibimos una llamada entrante. Deberemos permitir que esto ocurra. Además también es conveniente que el usuario pueda pausar la partida fácilmente.

Ante todo, estos videojuegos deben ser atractivos para los jugadores, ya que su única finalidad es entretener.

2. Desarrollo de juegos para móviles

Hemos visto que los juegos son aplicaciones que deben resultar atractivas para los usuarios. Por lo tanto deben tener gráficos personalizados e innovadores. La API de bajo nivel de LCDUI nos ofrece el control suficiente sobre lo que dibujamos en pantalla para poder crear cualquier interfaz gráfica que queramos que tengan nuestros juegos. Esto, junto al control a bajo nivel que nos ofrece sobre los eventos de entrada, hace que esta API sea suficiente para desarrollar videojuegos para móviles.

Además, en MIDP 2.0 se incluyen una serie de clases adicionales en el paquete `javax.microedition.lcdui.game` que basándose en la API a bajo nivel de LCDUI nos ofrecerán facilidades para el desarrollo de juegos. En estas clases tendremos implementados los objetos genéricos que solemos encontrar en todos los juegos. Si queremos desarrollar juegos con MIDP 1.0, deberemos implementar nosotros manualmente todos estos objetos.

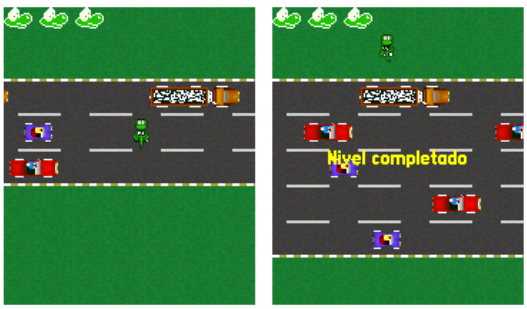
Los juegos se instalarán en el móvil como cualquier otra aplicación Java. En el caso concreto de los teléfonos Nokia, podemos establecer en el fichero JAD un atributo propio de esta marca con el que especificamos el tipo de aplicación:

```
Nokia-MIDlet-Category: Game
```

Especificando que se trata de un juego, cuando lo instalemos en el teléfono los guardará directamente en la carpeta *juegos*, y no en la carpeta *aplicaciones* como lo haría por defecto.

Para ilustrar las explicaciones, utilizaremos como ejemplo un clon del clásico *Frogger*. El mecanismo de este juego es muy sencillo:

- Tenemos una carretera con varios carriles por los que circulan coches.
- Por cada carril los coches circulan a distinta velocidad y con distinta separación entre ellos.
- Debemos cruzar la carretera pasando entre los coches sin ser atropellados.
- Cuando consigamos llegar al otro lado de la carretera habremos completado el nivel.
- Si un coche nos atropella, perderemos una vida.



Ejemplo de juego

2.1. Aplicación conducida por los datos

Cuando desarrollamos juegos, será conveniente llevar a la capa de datos todo lo que podamos, dejando la parte del código lo más simple y genérica posible.

Por ejemplo, podemos crear ficheros de datos donde se especifiquen las características de cada nivel del juego, el tipo y el comportamiento de los enemigos, los textos, etc.

Normalmente los juegos consisten en una serie de niveles. Cada vez que superemos un nivel, entraremos en uno nuevo en el que se habrá incrementado la dificultad, pero la mecánica del juego en esencia será la misma. Por esta razón es conveniente que el código del programa se encargue de implementar esta mecánica genérica, que conoceremos como motor del juego, y lea de ficheros de datos todas las características de cada nivel concreto.

De esta forma, si queremos añadir o modificar niveles del juego, cambiar la inteligencia artificial de los enemigos, añadir nuevos tipos de enemigos, o cualquier otro cambio de este tipo, no tendremos que modificar el código fuente, simplemente bastará con cambiar los ficheros de datos.

Es recomendable también centralizar la carga y la gestión de los recursos en una única clase. De esta forma quedará más claro qué recursos carga la aplicación, ya que no tendremos la carga de recursos dispersa por todo el código de las clases del juego. En este mismo fichero podemos tener los textos que se muestren en pantalla, lo que nos facilitará realizar traducciones del juego, ya qué sólo tendremos que modificar este fichero.

Por ejemplo, podemos tener un clase `Resources` donde se centralice la carga de los recursos como la siguiente:

```
public class Resources {

  // Identificadores de las imagenes
  public static final int IMG_TIT_TITULO = 0;
  public static final int IMG_SPR_CROC = 1;
  public static final int IMG_BG_STAGE_1 = 2;
  public static final int IMG_CAR_TYPE_1 = 3;
  public static final int IMG_CAR_TYPE_2 = 4;
  public static final int IMG_CAR_TYPE_3 = 5;
  public static final int IMG_FACE_LIVES = 6;

  // Imagenes y datos del juego
  public static Image[] img;
  public static Image splashImage;
  public static StageData[] stageData;

  // MIDlet principal del juego
  public static MIDletJuego midlet;

  // Nombres de los ficheros de las imagenes
  private static String[] imgNames =
  {
    "/title.png",
    "/sprite.png",
    "/stage01.png",
    "/car01.png",
```

```
    "/car02.png",
    "/car03.png",
    "/face.png"
  };
  private static String splashImageFile = "/logojava.png";
  private static String stageFile = "/stages.dat";

  // Obtiene una imagen
  public static Image getImage(int imgIndex) {
    return img[imgIndex];
  }

  // Obtiene los datos de una fase
  public static StageData getStage(int stageIndex) {
    return stageData[stageIndex];
  }

  // Carga los recursos
  public static void init() throws IOException {
    // Carga las imagenes
    loadCommonImages();

    // Carga datos de niveles
    InputStream in =
           stageFile.getClass().getResourceAsStream(stageFile);
    stageData = loadStageData(in);
  }

  // Inicializa los recursos para la pantalla splash
  public static void initSplash(MIDletJuego pMidlet)
                                       throws IOException {
    midlet = pMidlet;
    splashImage = Image.createImage(splashImageFile);
  }

  // Carga imagenes
  private static void loadCommonImages() throws IOException {
    int nImg = imgNames.length;

    img = new Image[nImg];

    for (int i = 0; i < nImg; i++) {
      img[i] = Image.createImage(imgNames[i]);
    }
  }

  // Carga los datos de los niveles
  public static StageData[] loadStageData(InputStream in)
    throws IOException {

    StageData[] stages = null;

    DataInputStream dis = new DataInputStream(in);

    int stageNum = dis.readInt();
    stages = new StageData[stageNum];
    for (int i = 0; i < stageNum; i++) {
      stages[i] = StageData.deserialize(dis);
    }
```

```
    return stages;
  }
}
```

Cuando se inicie el juego, llamaremos al método `init` de esta clase para que se inicialicen los recursos, en este momento se cargarán todas las imágenes y los datos que necesitemos. Una vez inicializado, podremos utilizar el método `getImage` para obtener las imágenes que vayamos a utilizar desde cualquier parte del código.

Como la carga de recursos suele ser lenta, normalmente mostraremos una pantalla de presentación, llamada *Splash Screen*, mientras éstos se cargan. En esta pantalla se suele mostrar el logo de nuestra compañía e información sobre el *copyright* de la aplicación.

Por esta razón existe en la clase `Resources` anterior un método `initSplash`, que hace una inicialización previa de los recursos que necesita esta pantalla *splash*, que normalmente será el logo que se vaya a mostrar. De esta forma, cuando carguemos la aplicación, se cargarán estos recursos básicos para la pantalla de *splash*, se mostrará la pantalla *splash*, y mientras ésta se muestra se cargarán el resto de recursos.

También debemos destacar que proporcionamos en la inicialización el MIDlet principal de nuestra aplicación. De esta forma, a través de la clase de recursos desde cualquier lugar de la aplicación tendremos acceso a este MIDlet, que será necesario para poder cambiar de pantalla o salir de la aplicación.

Por último, como hemos comentado, es recomendable llevar a la capa de datos toda la información posible. Esta clase `Resources` tiene un método `loadStageData` que la función que realiza es cargar desde un fichero la información sobre las fases del juego.

En el ejemplo de nuestro clon de *Frogger*, para cada fase consideraremos los siguientes datos:

- Título de la fase
- Carriles de la carretera. Para cada carril tendremos los siguientes datos:
  - Velocidad de los coches que circulan por él.
  - Separación que hay entre los coches del carril.
  - Tipo de coche que circula por el carril.

Toda esta información será conveniente tenerla almacenada en un fichero, de forma que simplemente editando el fichero, podremos editar las fases del juego. Por ejemplo, podemos utilizar un objeto `DataOutputStream` para codificar esta información de forma binaria con la siguiente estructura:

```
<int> Numero de fases
Para cada fase
  <UTF> Titulo
  <byte> Número de carriles
  Para cada carril
    <byte> Velocidad
    <short> Separación
```

```
    <byte> Tipo de coche
```

Para leer en nuestro juego este tipo de información, podemos crear los métodos de deserialización oportunos en las clases que encapsulen estos datos.

En el método `loadStageData` de la clase `Resources` se lee el primer valor con el número de fases, y para cada fase deserializaremos la información utilizando el objeto que encapsula los datos de las fases:

```
public class StageData {

  public String title;
  public TrackData [] tracks;

  public static StageData deserialize(DataInputStream dis)
                                      throws IOException    {
    StageData data = new StageData();

    data.title = dis.readUTF();
    byte nTracks = dis.readByte();

    data.tracks = new TrackData[nTracks];
    for(byte i=0;i<nTracks;i++) {
      data.tracks[i] = TrackData.deserialize(dis);
    }

    return data;
  }
}
```

De la misma forma, utilizaremos la clase que encapsula los datos de los carriles para deserializar la información de los mismos:

```
public class TrackData {

  public byte velocity;
  public short distance;
  public byte carType;

  public static TrackData deserialize(DataInputStream dis)
                                          throws IOException {
    TrackData data = new TrackData();

    data.velocity = dis.readByte();
    data.distance = dis.readShort();
    data.carType = dis.readByte();

    return data;
  }
}
```

2.2. Portabilidad

Al utilizar la API de bajo nivel para desarrollar el juego, la portabilidad de la aplicación se reducirá, ya que la implementación será dependiente de parámetros como el tamaño de

la pantalla, el número de colores, el teclado del dispositivo, etc.

Para poder adaptar nuestra aplicación fácilmente a distintos tipos de dispositivos, conviene definir toda esta información como constantes centralizadas en una única clase, de forma que simplemente cambiando la información sobre las dimensiones de los distintos elementos del juego podremos adaptarlo a distintos tamaños de pantalla.

Por ejemplo, podríamos definir una clase como la siguiente:

```
public class CommonData {

  // Numero de vidas total
  public static final int NUM_LIVES = 3;

  // Dimensiones de la pantalla
  public final static int SCREEN_WIDTH = 176;
  public final static int SCREEN_HEIGHT = 208;

  // Dimensiones del sprite
  public final static int SPRITE_WIDTH = 16;
  public final static int SPRITE_HEIGHT = 22;

  // Posicion inicial y velocidad del personaje
  public final static int SPRITE_STEP = 2;
  public final static int SPRITE_INI_X = 80;
  public final static int SPRITE_INI_Y = 180;
  public final static int SPRITE_END_Y = 20;

  // Datos del texto de la pantalla de titulo
  public static final int GAME_START_X = 88;
  public static final int GAME_START_Y = 150;
  public static final String GAME_START_TEXT =
      "PULSA START PARA COMENZAR";
  public static final int GAME_START_COLOR = 0x0FFFF00;
  public static final Font GAME_START_FONT =
      Font.getFont(Font.FACE_SYSTEM, Font.STYLE_BOLD,
                   Font.SIZE_SMALL);

  ...
}
```

En ella definimos el tamaño de la pantalla, el tamaño de nuestro personaje, las coordenadas (x,y) donde ubicar distintos elementos del juego, etc. De esta forma, para hacer una nueva versión sólo tendremos que editar este fichero y dar a cada elemento el tamaño y las coordenadas adecuadas.

2.3. Optimización

Los juegos deben funcionar de manera fluida y dar una respuesta rápida al usuario para que estos resulten jugables y atractivos. Por lo tanto, será conveniente optimizar el código todo lo posible, sobretodo en el caso de los dispositivos móviles en el que trabajamos con poca memoria y CPUs lentas.

No debemos cometer el error de intentar escribir un código optimizado desde el principio. Es mejor comenzar con una implementación clara, y una vez funcione esta implementación, intentar optimizar todo lo que sea posible.

Para optimizar el código deberemos detectar primero en qué lugar se está invirtiendo más tiempo. Por ejemplo, si lo que está ralentizando el juego es el volcado de los gráficos, deberemos optimizar esta parte, mientras que si es la lógica interna del juego la que requiere un tiempo alto, deberemos fijarnos en cómo optimizar este código.

El dibujado de los gráficos suele ser bastante costoso. Para optimizar este proceso deberemos intentar dibujar sólo aquello que sea necesario, es decir, lo que haya cambiado de un fotograma al siguiente. Muchas veces en la pantalla se está moviendo sólo un personaje pequeño, sería una pérdida de tiempo redibujar toda la pantalla cuando podemos repintar únicamente la zona en la que se ha movido este personaje. Esto lo podemos hacer con una variante del método `repaint` que nos permite redibujar sólo el área de pantalla indicada.

Por otro lado, dentro del código del juego deberemos utilizar las técnicas de optimización que conocemos, propias del lenguaje Java. Es importante intentar crear el mínimo número de objetos posibles, reutilizando objetos siempre que podamos. Esto evita el tiempo que requiere la instanciación de objetos y su posterior eliminación por parte del colector de basura.

También deberemos tener en cuenta que la memoria del móvil es muy limitada, por lo que deberemos permitir que se desechen todos los objetos que ya no necesitamos. Para que un objeto pueda ser eliminado por el colector de basura deberemos poner todas las referencias que tengamos a dicho objeto a `null`, para que el colector de basura sepa que ya nadie va a poder usar ese objeto por lo que puede eliminarlo.

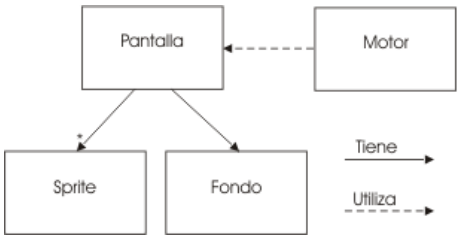
3. Componentes de un videojuego

Cuando diseñemos un juego deberemos identificar las distintas entidades que encontraremos en él. Normalmente en los juegos de acción en 2D tendremos una pantalla del juego, que tendrá un fondo y una serie de personajes u objetos que se mueven en este escenario. Estos objetos que se mueven en el escenario se conocen como *sprites*. Además, tendremos un motor que se encargará de conducir la lógica interna del juego. Podemos abstraer los siguientes componentes:

- **Sprites**: Objetos o personajes que pueden moverse por la pantalla y/o con los que podemos interactuar.
- **Fondo**: Escenario de fondo, normalmente estático, sobre el que se desarrolla el juego. Muchas veces tendremos un escenario más grande que la pantalla, por lo que tendrá *scroll* para que la pantalla se desplace a la posición donde se encuentra nuestro personaje.
- **Pantalla**: En la pantalla se muestra la escena del juego. Aquí es donde se deberá

dibujar todo el contenido, tanto el fondo como los distintos *sprites* que aparezcan en la escena y otros datos que se quieran mostrar.

- **Motor del juego**: Es el código que implementará la lógica del juego. En él se leerá la entrada del usuario, actualizará la posición de cada elemento en la escena, comprobando las posibles interacciones entre ellos, y dibujará todo este contenido en la pantalla.

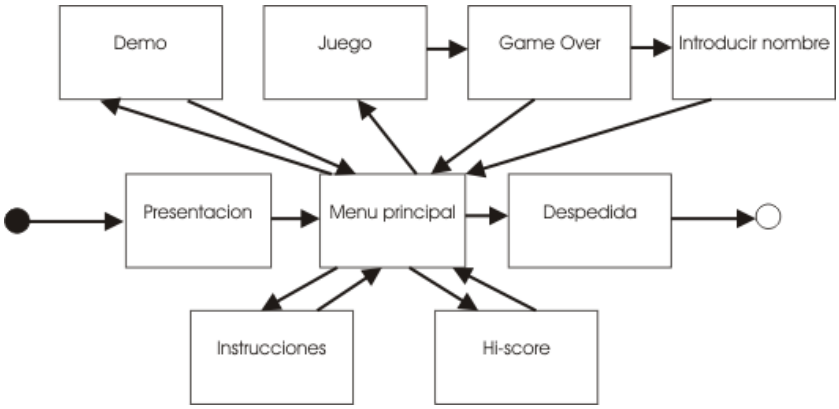


Componentes de un juego

A continuación veremos con más detalle cada uno de estos componentes, viendo como ejemplo las clases que MIDP 2.0 nos proporciona para crear cada uno de ellos.

Esto es lo que encontraremos en la pantalla de juego, mientras dure la partida. Sin embargo los juegos normalmente constarán de varias pantallas. Las más usuales son las siguientes:

- **Pantalla de presentación (*Splash screen*)**. Pantalla que se muestra cuando cargamos el juego, con el logo de la compañía que lo ha desarrollado y los créditos. Aparece durante un tiempo breve (se puede aprovechar para cargar los recursos necesarios en este tiempo), y pasa automáticamente a la pantalla de título.
- **Título y menú**. Normalmente tendremos una pantalla de título principal del juego donde tendremos el menú con las distintas opciones que tenemos. Normalmente podremos comenzar una nueva partida, reanudar una partida anterior, ver las puntuaciones más altas, ver las instrucciones, o bien salir del juego.
- **Hi-score**. Pantalla de puntuaciones más altas obtenidas. Se mostrará el *ranking* de puntuaciones, donde aparecerá el nombre o iniciales de los jugadores junto a su puntuación obtenida.
- **Instrucciones**. Nos mostrará un texto con las instrucciones del juego.
- **Juego**. Será la pantalla donde se desarrolle el juego, que tendrá los componentes que hemos visto anteriormente.



Mapa de pantallas típico de un juego

Hemos de decidir qué API utilizar para desarrollar cada pantalla. La pantalla de juego claramente debe realizarse utilizando la API de bajo nivel. Sin embargo con esta API hemos visto que no podemos introducir texto de una forma sencilla. Si utilizamos la API de alto nivel para la pantalla de puntuaciones más altas se nos facilitará bastante la tarea. El inconveniente es que siempre tendrá un aspecto más atractivo y propio del juego si utilizamos la API de bajo nivel, aunque nos cueste más programarla.

Es importante poder salir en todo momento del juego, cuando el usuario quiera de terminar de jugar. Durante la partida se deberá permitir volver al menú principal, o incluso salir directamente del juego. Para las acciones de salir y volver al menú se deben utilizas las teclas asociadas a las esquinas de la pantalla (*soft-keys*). No es recomendable utilizar estas teclas para acciones del juego.

4. Sprites

Los *sprites* hemos dicho que son todos aquellos objetos que aparecen en la escena que se mueven y/o podemos interactuar con ellos de alguna forma.

4.1. Animación

Estos objetos pueden estar animados. Para ello deberemos definir los distintos fotogramas (o *frames*) de la animación. Podemos definir varias animaciones para cada *sprite*, según las acciones que pueda hacer. Por ejemplo, si tenemos un personaje podemos tener una animación para andar hacia la derecha y otra para andar hacia la izquierda.

El *sprite* tendrá un determinado tamaño (ancho y alto), y cada fotograma será una imagen de este tamaño. Para no tener un número demasiado elevado de imágenes lo que haremos será juntar todos los fotogramas del *sprite* en una misma imagen, dispuestos como un mosaico.

Mosaico con los frames de un sprite

Cambiando el fotograma que se muestra del *sprite* en cada momento podremos animarlo. En MIDP 2.0 se proporciona la clase `Sprite` que nos permite manejar este tipo de mosaicos para definir los fotogramas del *sprite* y animarlo. Podemos crear el *sprite* de la siguiente forma:

```
Sprite personaje = new Sprite(imagen, ancho_fotograma,
                              alto_fotograma);
```

Proporcionamos la imagen donde tenemos el mosaico de fotogramas, y definimos las dimensiones de cada fotograma. De esta forma esta clase se encargará de separar los fotogramas que hay dentro de esta imagen.

Cada fotograma tendrá un índice que se empezará a numerar a partir de cero. La ordenación de los *frames* en la imagen se realiza por filas y de izquierda a derecha, por lo que el *frame* de la esquina superior izquierda será el *frame* `0`. Podemos establecer el *frame* a mostrar actualmente con:

```
personaje.setFrame(indice);
```

Podremos definir determinadas secuencias de *frames* para crear animaciones.

4.2. Desplazamiento

Además el *sprite* se podrá desplazar por la pantalla, por lo que deberemos tener algún método para moverlo. El *sprite* tendrá una cierta localización, dada en coordenadas *(x,y)* de la pantalla, y podremos o bien establecer una nuevas coordenadas o desplazar el *sprite* respecto a su posición actual:

```
personaje.setPosition(x, y);
personaje.move(dx, dy);
```

Con el primer método damos la posición absoluta donde queremos posicionar el *sprite*. En el segundo caso indicamos un desplazamiento, para desplazarlo desde su posición actual. Normalmente utilizaremos el primer método para posicionarlo por primera vez en su posición inicial al inicio de la partida, y el segundo para moverlo durante el transcurso de la misma.

4.3. Colisiones

Otro aspecto de los *sprites* es la interacción entre ellos. Nos interesará saber cuando somos tocados por un enemigo o una bala para disminuir la vida, o cuando alcanzamos nosotros a nuestro enemigo. Para ello deberemos detectar las colisiones entre *sprites*. La colisión con *sprites* de formas complejas puede resultar costosa de calcular. Por ello se

suele realizar el cálculo de colisiones con una forma aproximada de los *sprites* con la que esta operación resulte más sencilla. Para ello solemos utilizar el *bounding box*, es decir, un rectángulo que englobe el *sprite*. La intersección de rectángulos es una operación muy sencilla, podremos comprobarlo simplemente con una serie de comparaciones menor que < y mayor que >.

La clase `Sprite` nos permite realizar esta comprobación de colisiones utilizando un *bounding box*. Podemos comprobar si nuestro personaje está tocando a un enemigo con:

```
personaje.collidesWith(enemigo, false);
```

Con el segundo parámetro a `true` le podemos decir que compruebe la colisión a nivel de *pixels*, es decir, que en lugar de usar el *bounding box* compruebe *pixel* a *pixel* si ambos *sprites* colisionan. Esto será bastante más costoso. Si necesitamos hacer la comprobación a este nivel, podemos comprobar primero si colisionan sus *bounding boxes* para descartar así de forma eficiente bastantes casos, y en caso de que los *bounding boxes* si que intersecten, hacer la comprobación a nivel de *pixels* para comprobar si realmente colisionan o no. Normalmente las implementaciones harán esto internamente cuando comprobemos la colisión a nivel de *pixels*.

4.4. Ejemplo

Vamos a ver como ejemplo la creación del sprite de nuestro personaje para el clon del *Frogger*. Para cada sprite podemos crear una subclase de `Sprite` que se especialice en el tipo de sprite concreto del que se trate.

```
public class CrocSprite extends Sprite {

  public final static int MOVE_UP = 0;
  public final static int MOVE_DOWN = 1;
  public final static int MOVE_LEFT = 2;
  public final static int MOVE_RIGHT = 3;
  public final static int MOVE_STAY = 4;

  int lastMove;

  public CrocSprite() {
    super(Resources.getImage(Resources.IMG_SPR_CROC),
        CommonData.SPRITE_WIDTH, CommonData.SPRITE_HEIGHT);
    this.defineCollisionRectangle(CommonData.SPRITE_CROP_X,
        CommonData.SPRITE_CROP_Y, CommonData.SPRITE_CROP_WIDTH,
        CommonData.SPRITE_CROP_HEIGHT);
    reset();
  }

  public void reset() {
    // Inicializa frame y movimiento actual
    lastMove = MOVE_STAY;
    this.setFrameSequence(null);
    this.setFrame(CommonData.SPRITE_STAY_UP);

    // Inicializa posición
```

```
  this.setPosition(CommonData.SPRITE_INI_X,
                   CommonData.SPRITE_INI_Y);
}

public void stay() {
  switch(lastMove) {
    case MOVE_UP:
      this.setFrameSequence(null);
      this.setFrame(CommonData.SPRITE_STAY_UP);
      break;
    case MOVE_DOWN:
      this.setFrameSequence(null);
      this.setFrame(CommonData.SPRITE_STAY_DOWN);
      break;
    case MOVE_LEFT:
      this.setFrameSequence(null);
      this.setFrame(CommonData.SPRITE_STAY_LEFT);
      break;
    case MOVE_RIGHT:
      this.setFrameSequence(null);
      this.setFrame(CommonData.SPRITE_STAY_RIGHT);
      break;
  }
  lastMove = MOVE_STAY;
}

public void die() {
  this.setFrameSequence(null);
  this.setFrame(CommonData.SPRITE_STAY_DIED);
  lastMove = MOVE_STAY;
}

public void moveUp() {
  if(lastMove != MOVE_UP) {
    this.setFrameSequence(CommonData.SPRITE_MOVE_UP);
  }
  lastMove = MOVE_UP;
  this.move(0,-CommonData.SPRITE_STEP);
  this.nextFrame();
}

public void moveDown() {
  if(lastMove != MOVE_DOWN) {
    this.setFrameSequence(CommonData.SPRITE_MOVE_DOWN);
  }
  lastMove = MOVE_DOWN;
  this.move(0,CommonData.SPRITE_STEP);
  this.nextFrame();
}

public void moveLeft() {
  if(lastMove != MOVE_LEFT) {
    this.setFrameSequence(CommonData.SPRITE_MOVE_LEFT);
  }
  lastMove = MOVE_LEFT;
  this.move(-CommonData.SPRITE_STEP,0);
  this.nextFrame();
}
```

```
  public void moveRight() {
    if(lastMove != MOVE_RIGHT) {
      this.setFrameSequence(CommonData.SPRITE_MOVE_RIGHT);
    }
    lastMove = MOVE_RIGHT;
    this.move(CommonData.SPRITE_STEP,0);
    this.nextFrame();
  }
}
```

En este ejemplo vemos como para los movimientos que requieran animación (como por ejemplo andar en las distintas direcciones) podemos indicar una secuencia de frames que componen esta animación con `setFrameSequence`, y mediante el método `nextFrame` ir pasando al siguiente.

Si queremos volver a disponer del conjunto de frames completo deberemos llamar al método `setFrameSequence` pasándole como parámetro `null`.

Otro problema que nos puede surgir es por ejemplo que no queramos que las colisiones se calculen con el rectángulo completo de nuestro sprite, sino sólo con un subrectángulo de éste. Por ejemplo, si en nuestro juego tenemos a un cocodrilo que debe cruzar la calle, no querremos que cuando un coche pise su cola se considere que lo han atropellado, el jugador se sentirá frustrado si cada vez que un coche roza su cola pierde una vida.

La parte inferior de la imagen del cocodrilo sólo contiene la cola, de forma que por defecto, como el rectángulo de colisión es del tamaño de la imagen, cuando haya un coche que colisione con esta parte inferior se considerará que nos han atropellado. Podemos establecer un nuevo rectángulo para el cálculo de colisiones con el método `defineCollisionRectangle`, de forma que ya no consideremos toda la imagen, sino sólo el área de la misma en la que está el cuerpo del cocodrilo.

5. Fondo

En los juegos normalmente tendremos un fondo sobre el que se mueven los personajes. Muchas veces los escenarios del juego son muy extensos y no caben enteros en la pantalla. De esta forma lo que se hace es ver sólo la parte del escenario donde está nuestro personaje, y conforme nos movamos se irá desplazando esta zona visible para enfocar en todo momento el lugar donde está nuestro personaje. Esto es lo que se conoce como *scroll*.

El tener un fondo con *scroll* será más costoso computacionalmente, ya que siempre que nos desplacemos se deberá redibujar toda la pantalla, debido a que se está moviendo todo el fondo. Además para poder dibujar este fondo deberemos tener una imagen con el dibujo del fondo para poder volcarlo en pantalla. Si tenemos un escenario extenso, sería totalmente prohibitivo hacer una imagen que contenga todo el fondo. Esta imagen podría llegar a ocupar el tamaño máximo permitido para los ficheros JAR es muchos móviles.

Para evitar este problema lo que haremos normalmente en este tipo de juegos es construir

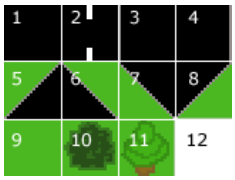
el fondo como un mosaico. Nos crearemos una imagen con los elementos básicos que vamos a necesitar para nuestro fondo, y construiremos el fondo como un mosaico en el que se utilizan estos elementos.

Mosaico de elementos del fondo

Al igual que hacíamos con los *sprites*, tomaremos estos distintos elementos de una misma imagen. A cada elemento se le asociará un índice con el que lo referenciaremos posteriormente. El fondo lo crearemos como un mosaico del tamaño que queramos, en el que cada celda contendrá el índice del elemento que se quiere mostrar en ella.

La clase `TiledLayer` de MIDP 2.0 nos permite realizar esto. Deberemos especificar el número de filas y columnas que va a tener el mosaico, y después la imagen que contiene los elementos y el ancho y alto de cada elemento.

```
TiledLayer fondo = new TiledLayer(columnas, filas, imagen,
                                  ancho, alto);
```



Índices de los elementos del mosaico

De esta forma el fondo tendrá un tamaño en *pixels* de `(columnas*ancho)x(filas*alto)`. Ahora podemos establecer el elemento que contendrá cada celda del mosaico con el método:

```
fondo.setCell(columna, fila, indice);
```

Como `indice` especificaremos el índice del elemento en la imagen que queremos incluir en esa posición del mosaico, `0` si queremos dejar esa posición vacía con el color del fondo, o un número negativo para crear celdas con animación.

Ejemplo de fondo construido con los elementos anteriores

A continuación mostramos el código que podríamos utilizar para generar la carretera de nuestro juego:

```
public class Background extends TiledLayer {

  public Background() {
    super(CommonData.BG_H_TILES, CommonData.BG_V_TILES,
          Resources.getImage(Resources.IMG_BG_STAGE_1),
          CommonData.BG_TILE_WIDTH, CommonData.BG_TILE_HEIGHT);
  }

  public void reset(StageData stage) {
    TrackData [] tracks = stage.tracks;
    int nTracks = tracks.length;

    int row;

    // Filas superiores de cesped
    for(row=0;row<CommonData.BG_V_TOP_TILES;row++) {
      for(int col=0;col<CommonData.BG_H_TILES; col++) {
        this.setCell(col, row, CommonData.BG_TILE_GRASS);
      }
    }

    // Margen superior de la carretera
    for(int col=0;col<CommonData.BG_H_TILES; col++) {
      this.setCell(col, row, CommonData.BG_TILE_TOP);
    }
    row++;

    // Parte interna de la carreteta
    for(;row<CommonData.BG_V_TOP_TILES + nTracks - 1;row++) {
      for(int col=0;col<CommonData.BG_H_TILES; col++) {
        this.setCell(col, row, CommonData.BG_TILE_CENTER);
      }
    }

    // Margen inferior de la carretera
    for(int col=0;col<CommonData.BG_H_TILES; col++) {
      this.setCell(col, row, CommonData.BG_TILE_BOTTOM);
    }
    row++;

    // Hierba en la zona inferior
    for(;row<CommonData.BG_V_TILES;row++) {
      for(int col=0;col<CommonData.BG_H_TILES; col++) {
        this.setCell(col, row, CommonData.BG_TILE_GRASS);
      }
    }
  }
}
```

6. Pantalla

En la pantalla se dibujarán todos los elementos anteriores para construir la escena del

juego. De esta manera tendremos el fondo, nuestro personaje, los enemigos y otros objetos que aparezcan durante el juego, además de marcadores con el número de vidas, puntuación, etc.

La pantalla la vamos a dibujar por capas. Cada *sprite* y cada fondo que incluyamos será una capa, de esta forma poniendo una capa sobre otra construiremos la escena. Tanto la clase `Sprite` como la clase `TiledLayer` heredan de `Layer`, que es la clase que define de forma genérica las capas, por lo que podrán comportarse como tales. Todas las capas podrán moverse o cambiar de posición, para mover de esta forma su contenido en la pantalla.

Construiremos el contenido de la pantalla superponiendo una capa sobre otra. Tenemos la clase `LayerManager` en MIDP 2.0 que nos permitirá construir esta superposición de capas. Este objeto contendrá todo lo que se vaya a mostrar en pantalla, encargándose él internamente de gestionar y dibujar esta superposición de capas. Podemos crear este objeto con:

```
LayerManager escena = new LayerManager();
```

Ahora deberemos añadir por orden las capas que queramos que se muestren. El orden en el que las añadamos indicará el orden *z*, es decir, la profundidad de esta capa en la escena. La primera capa será la más cercana al punto de vista del usuario, mientras que la última será la más lejana. Por lo tanto, las primeras capas que añadamos quedarán por delante de las siguientes capas. Para añadir capas utilizaremos:

```
escena.append(personaje);
escena.append(enemigo);
escena.append(fondo);
```

También podremos insertar capas en una determinada posición de la lista, o eliminar capas de la lista con los métodos `insert` y `remove` respectivamente.

El área dibujada del `LayerManager` puede ser bastante extensa, ya que abarcará por lo menos toda la extensión del fondo que hayamos puesto. Deberemos indicar qué porción de esta escena se va a mostrar en la pantalla, especificando la ventana de vista. Moviendo esta ventana podremos implementar *scroll*. Podemos establecer la posición y tamaño de esta ventana con:

```
escena.setViewWindow(x, y, ancho, alto);
```

La posición de esta ventana de vista es referente al sistema de coordenadas de la escena (de la clase `LayerManager`).

Debemos tener en cuenta al especificar el tamaño del visor que diferentes modelos de móviles tendrán pantallas de diferente tamaño. Podemos hacer varias cosas para que el juego sea lo más portable posible. Podríamos crear un visor del tamaño mínimo de pantalla que vayamos a considerar, y en el caso de que la pantalla sea de mayor tamaño mostrar este visor centrado. Otra posibilidad es establecer el tamaño de la ventana de vista según la pantalla del móvil, haciendo que en los móviles con pantalla más grande se

vea un mayor trozo del escenario.

Una vez hemos establecido esta ventana de vista, podemos dibujarla en el contexto gráfico g con:

```
escena.paint(g, x, y);
```

Donde daremos las coordenadas donde dibujaremos esta vista, dentro del espacio de coordenadas del contexto gráfico indicado.

7. Motor del juego

Hemos visto que los juegos suelen constar de varias pantallas. En la mayoría de los casos tenemos una pantalla de título, la pantalla en la que se desarrolla la partida, la pantalla de *game over*, una pantalla de demo en la que vemos el juego funcionar automáticamente como ejemplo, etc.

Normalmente lo primero que veremos será el título, de aquí podremos pasar al juego o al modo demo. Si transcurre un determinado tiempo sin que el usuario pulse ninguna tecla pasará a demo, mientras que si el usuario pulsa la tecla *start* comienza el juego. La demo finalizará pasado un tiempo determinado, tras lo cual volverá al título. El juego finalizará cuando el jugador pierda todas sus vidas, pasando a la pantalla de *game over*, y de ahí volverá al título.

7.1. Ciclo del juego

Vamos a centrarnos en cómo desarrollar la pantalla en la que se desarrolla la partida. Aquí tendremos lo que se conoce como ciclo del juego (o *game loop*). Se trata de un bucle infinito en el que tendremos el código fuente que implementa el funcionamiento del juego. Dentro de este bucle se efectúan las siguientes tareas básicas:

- **Leer la entrada**: Lee la entrada del usuario para conocer si el usuario ha pulsado alguna tecla desde la última iteración.
- **Actualizar escena**: Actualiza las posiciones de los *sprites* y su fotograma actual, en caso de que estén siendo animados, la posición del fondo si se haya producido *scroll*, y cualquier otro elemento del juego que deba cambiar. Para hacer esta actualización se pueden tomar diferentes criterios. Podemos mover el personaje según la entrada del usuario, la de los enemigos según su inteligencia artificial, o según las interacciones producidas entre ellos y cualquier otro objeto (por ejemplo al ser alcanzados por un disparo, colisionando el *sprite* del disparo con el del enemigo), etc.
- **Redibujar**: Tras actualizar todos los elementos del juego, deberemos redibujar la pantalla para mostrar la escena tal como ha quedado en el instante actual.
- **Dormir**: Normalmente tras cada iteración dormiremos un determinado número de milisegundos para controlar la velocidad a la que se desarrolla el juego. De esta forma podemos establecer a cuantos fotogramas por segundo (*fps*) queremos que funcione el

juego, siempre que la CPU sea capaz de funcionar a esta velocidad.

La clase `GameCanvas` es un tipo especial de *canvas* para el desarrollo de juegos presente en MIDP 2.0. Esta clase nos ofrecerá los métodos necesarios para realizar este ciclo del juego, ofreciéndonos un acceso a la entrada del usuario y un método de *render* adecuados para este tipo de aplicaciones. Para hacer la pantalla del juego crearemos una subclase de `GameCanvas` en la que introduciremos el ciclo del juego.

La forma en la que dibujaremos los gráficos utilizando esta clase será distinta a la que hemos visto para la clase `Canvas`. En el caso del `Canvas` utilizamos *render* pasivo, es decir, nosotros definimos en el método `paint` la forma en la que se dibuja y es el sistema el que llama a este método. Ahora nos interesa poder dibujar en cada iteración los cambios que se hayan producido en la escena directamente desde el bucle del ciclo del juego. Es decir, seremos nosotros los que decidamos el momento en el que dibujar, utilizaremos *render* activo. Para ello en cualquier momento podremos obtener un contexto gráfico asociado al `GameCanvas` desde dentro de este mismo objeto con:

```
Graphics g = getGraphics();
```

Este contexto gráfico estará asociado a un *backbuffer* del *canvas* de juegos. De esta forma durante el ciclo del juego podremos dibujar el contenido de la pantalla en este *backbuffer*. Cuando queramos que este *backbuffer* se vuelque a pantalla, llamaremos al método:

```
flushGraphics();
```

Existe también una versión de este método en la que especificamos la región que queremos que se vuelque, de forma que sólo tenga que volcar la parte de la pantalla que haya cambiado.

Vamos a ver ahora el esqueleto de un ciclo del juego básico que podemos realizar:

```
Graphics g = getGraphics();

while(true) {
    leeEntrada();
    actualizaEscena();
    dibujaGraficos(g);

    flushGraphics();
}
```

Será conveniente dormir un determinado tiempo tras cada iteración para controlar así la velocidad del juego. Vamos a considerar que `CICLO` es el tiempo que debe durar cada iteración del juego. Lo que podremos hacer es obtener el tiempo que ha durado realmente la iteración, y dormir el tiempo restante hasta completar el tiempo de ciclo. Esto podemos hacerlo de la siguiente forma:

```
Graphics g = getGraphics();
long t1, t2, td;

while(true) {
```

```
    t1 = System.currentTimeMillis();

    leeEntrada();
    actualizaEscena();
    dibujaGraficos();

    flushGraphics();

    t2 = System.currentTimeMillis();
    td = t2 - t1;
    td = td<CICLO?td:CICLO;

    try {
        Thread.sleep(CICLO - td);
    } catch(InterruptedException e) { }
}
```

Todo este bucle donde se desarrolla el ciclo del juego lo ejecutaremos como un hilo independiente. Un buen momento para poner en marcha este hilo será el momento en el que se muestre el *canvas*:

```
public class Juego extends GameCanvas implements Runnable {
    public void showNotify() {
        Thread t = new Thread(this);
        t.start();
    }

    public void run() {
        // Ciclo del juego
        ...
    }
}
```

7.2. Máquina de estados

Hemos visto cómo implementar el ciclo del juego en el que en cada momento se lea la entrada del usuario, se actualice la escena, y se vuelquen los gráficos a la pantalla.

Sin embargo, este ciclo no siempre deberá comportarse de la misma forma. El juego podrá pasar por distintos estados, y en cada uno de ellos deberán realizarse una tareas e ignorarse otras.

Podemos modelar esto como una máquina de estados, en la que en cada momento, según el estado actual, se realicen unas funciones u otras, y cuando suceda un determinado evento, se pasará a otro estado.

Por ejemplo, en un juego podremos encontrar comúnmente los siguientes estados durante el desarrollo del mismo.

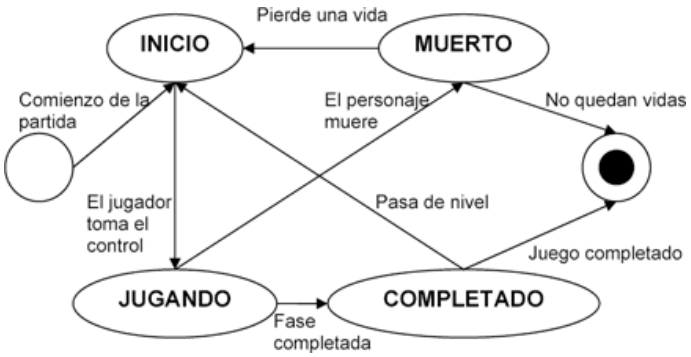


Diagrama de estados del juego

- *INICIO*: Al comienzo de la fase, se muestra durante un instante la fase en la que se va a jugar y el título de la misma para que el jugador se vaya preparando. En este momento todavía no se deja al usuario manejar al personaje. Pasado un determinado tiempo, pasará a estado *JUGANDO*.
- *JUGANDO*: En este estado el jugador tiene el control sobre el personaje. Si el jugador muere, pasará a estado *MUERTO*, mientras que si consigue completar el nivel pasará a estado *COMPLETADO*.
- *MUERTO*: Se muestra la muerte del personaje. Aquí el usuario ya no tendrá el control sobre el mismo. Una vez pasado un tiempo, nos restará una vida y pasará a estado de *INICIO*, o si no nos quedan vidas finalizará el juego.
- *COMPLETADO*: Se muestra un letrero en el que se indique se el jugador ha conseguido completar este nivel y posiblemente un resumen de lo conseguido en él, como por ejemplo la puntuación, el tiempo que ha sobrado, etc. Una vez pasado un tiempo, pasará a estado *INICIO* para la siguiente fase del juego, o bien si estábamos en la última fase finalizará el juego.

8. Entrada de usuario en juegos

8.1. Acciones de juegos en MIDP 1.0

La clase `Canvas` en MIDP 1.0 ofrece facilidades para leer la entrada de usuario para juegos. Hemos visto en el tema anterior que asocia a los códigos de las teclas lo que se conoce como acciones de juegos, lo cual nos facilitará el desarrollo de aplicaciones que se controlan mediante estas acciones (arriba, abajo, izquierda, derecha, fuego), que principalmente son juegos.

Cuando se lee una tecla en cualquier evento de pulsación de teclas (`keyPressed`, `keyRepeated` o `keyReleased`), se nos proporciona como parámetro el código de dicha tecla. Podemos comprobar a qué acción de juego corresponde dicha tecla de forma independiente a la plataforma con el método `getGameAction` como se muestra en el siguiente ejemplo:

```
public void keyPressed(int keyCode) {

    int action = getGameAction(keyCode);

    if (action == LEFT) {
        moverIzquierda();
    } else if (action == RIGHT) {
        moverDerecha();
    } else if (action == FIRE) {
        disparar();
    }
}
```

También podemos obtener la tecla principal asociada a una acción de juego con el método `getKeyCode`, pero dado que una acción puede estar asociada a varias teclas, si usamos este método no estaremos considerando las teclas secundarias asociadas a esa misma acción. A pesar de que en algunos ejemplos podamos ver código como el que se muestra a continuación, esto no debe hacerse:

```
public class MiCanvas extends Canvas {

    // NO HACER ESTO!

    int izq, der, fuego;

    public MiCanvas() {
        izq = getKeyCode(LEFT);
        der = getKeyCode(RIGHT);
        fuego = getKeyCode(FIRE);
    }

    public void keyPressed(int keyCode) {
        if (keyCode == izq) {
            moverIzquierda();
        } else if (keyCode = der) {
            moverDerecha();
        } else if (keyCode = fuego) {
            disparar();
        }
    }
}
```

Hemos de tener en cuenta que muchos modelos de móviles no nos permiten mantener pulsadas más de una tecla al mismo tiempo. Hay otros que, aunque esto se permita, el *joystick* no puede mantener posiciones diagonales, sólo se puede pulsar una de las cuatro direcciones básicas al mismo tiempo.

Esto provoca que si no tenemos suficiente con estas cuatro direcciones básicas y necesitamos realizar movimientos en diagonal, tendremos que definir nosotros manualmente los códigos de las teclas para cada una de estas acciones. Esto reducirá portabilidad a la aplicación, será el precio que tendremos que pagar para poder establecer una serie de teclas para movimiento diagonal.

Si definimos nosotros las acciones directamente a partir de los códigos de teclas a bajo

nivel deberemos intentar respetar en la medida de lo posible el comportamiento que suele tener cada tecla en los móviles. Por ejemplo, las teclas asociadas a las esquinas de la pantalla (*soft keys*) deben utilizarse para terminar el juego y volver al menú principal o bien salir directamente del juego.

8.2. Acceso al teclado con MIDP 2.0

Hasta ahora hemos visto las facilidades que nos ofrece MIDP para leer la entrada del usuario en los juegos desde su versión 1.0. Sin embargo, MIDP 2.0 incluye facilidades adicionales en la clase `GameCanvas`.

Podremos leer el estado de las teclas en cualquier momento, en lugar de tener que definir *callbacks* para ser notificados de las pulsaciones de las teclas. Esto nos facilitará la escritura del código, pudiendo obtener directamente esta información desde el ciclo del juego.

Podemos obtener el estado de las teclas con el método `getKeyStates` como se muestra a continuación:

```
int keyState = getKeyStates();
```

Esto nos devolverá un número entero en el que cada uno de sus *bits* representa una tecla. Si el *bit* vale `0` la tecla está sin pulsar, y si vale `1` la tecla estará pulsada. Tenemos definidos los *bits* asociados a cada tecla como constantes, que podremos utilizar como máscaras *booleanas* para extraer el estado de cada tecla a partir del número entero de estado obtenido:

`GameCanvas.LEFT_PRESSED`	Movimiento a la izquierda
`GameCanvas.RIGHT_PRESSED`	Movimiento a la derecha
`GameCanvas.UP_PRESSED`	Movimiento hacia arriba
`GameCanvas.DOWN_PRESSED`	Movimiento hacia abajo
`GameCanvas.FIRE_PRESSED`	Fuego

Por ejemplo, para saber si están pulsadas las teclas izquierda o derecha haremos la siguiente comprobación:

```
if ((keyState & LEFT_PRESSED) != 0) {
    moverIzquierda();
}

if ((keyState & RIGHT_PRESSED) != 0) {
    moverDerecha();
}
```

Vamos a ver ahora un ejemplo de ciclo de juego sencillo completo:

```
public class CanvasJuego extends GameCanvas implements Runnable {
```

```
    private final static int CICLO = 50;

    public CanvasJuego() {
        super(true);
    }

    public void showNotify() {
        Thread t = new Thread(this);
        t.start();
    }

    public void run() {
        Graphics g = getGraphics();
        long t1, t2, td;

        // Carga sprites
        CrocSprite personaje = new CrocSprite();
        personaje.reset();

        // Crea fondo
        Background fondo = new Background(Resources.getStage(0));
        fondo.reset();

        // Crea escena
        LayerManager escena = new LayerManager();
        escena.append(personaje);
        escena.append(fondo);

        while(true) {
            t1 = System.currentTimeMillis();

            // Lee entrada del teclado
            int keyState = getKeyStates();

            if ((keyState & LEFT_PRESSED) != 0) {
                personaje.moveLeft();
            } else if ((keyState & RIGHT_PRESSED) != 0) {
                personaje.moveRight();
            } else if ((keyState & UP_PRESSED) != 0) {
                personaje.moveUp();
            } else if ((keyState & DOWN_PRESSED) != 0) {
                personaje.moveDown();
            }

            escena.paint(g);

            flushGraphics();

            t2 = System.currentTimeMillis();
            td = t2 - t1;
            td = td<CICLO?td:CICLO;

            try {
                Thread.sleep(CICLO - td);
            } catch(InterruptedException e) {}
        }
    }
}
```

9. Ejemplo de motor de juego

Una vez hemos visto todos los elementos necesarios para desarrollar nuestro juego, vamos a ver un ejemplo de motor de juego completo que utilizaremos para nuestro clon de *Frogger*.

La siguiente clase será la encargada de realizar el ciclo básico del juego. En ella implementaremos la posibilidad de realizar pausas en el juego, lo cual ya hemos comentado que es muy importante en juegos para móviles:

```
public class GameEngine extends GameCanvas implements Runnable {

  // Milisegundos que transcurren entre dos frames consecutivos
  public final static int CICLO = 50;

  // Codigos de las teclas soft
  public final static int LEFT_SOFTKEY = -6;
  public final static int RIGHT_SOFTKEY = -7;

  // Escena actual
  Scene escena;

  // Estado de pausa
  boolean isPaused;

  // Hilo del juego
  Thread t;

  // Fuente pausa
  Font font;

  public GameEngine(Scene escena) {
    super(false);
    this.setFullScreenMode(true);

    // Establece la escena actual
    this.escena = escena;

    // Inicializa fuente para el texto de la pausa
    font = Font.getFont(Font.FACE_SYSTEM,
                        Font.STYLE_BOLD, Font.SIZE_MEDIUM);
  }

  // Cambia la escena
  public void setScene(Scene escena) {
    this.escena = escena;
  }

  public void showNotify() {
    start();
  }

  public void hideNotify() {
    stop();
  }
```

```
public void keyPressed(int keyCode) {

  if(isPaused) {
    if(keyCode == LEFT_SOFTKEY) {
      start();
    } else if(keyCode == RIGHT_SOFTKEY) {
      Resources.midlet.exitGame();
    }
  } else {
    if(keyCode == LEFT_SOFTKEY || keyCode == RIGHT_SOFTKEY) {
      stop();
    }
  }
}

// Pausa el juego
public synchronized void stop() {
  t = null;
  isPaused = true;

  Graphics g = getGraphics();
  render(g);
  flushGraphics();
}

// Reanuda el juego
public synchronized void start() {
  isPaused = false;

  t = new Thread(this);
  t.start();
}

// Ciclo del juego
public void run() {

  // Obtiene contexto gráfico
  Graphics g = getGraphics();

  while (t == Thread.currentThread()) {

    long t_ini, t_dif;
    t_ini = System.currentTimeMillis();

    // Lee las teclas
    int keyState = this.getKeyStates();

    // Actualiza la escena
    escena.tick(keyState);

    // Vuelca los graficos
    render(g);
    flushGraphics();

    t_dif = System.currentTimeMillis() - t_ini;

    // Duerme hasta el siguiente frame
    if (t_dif < CICLO) {
```

```
        try {
          Thread.sleep(CICLO - t_dif);
        } catch (InterruptedException e) { }
      }
    }
  }

  public void render(Graphics g) {

    escena.render(g);

    if(isPaused) {
      g.setColor(0x000000);
      for(int i=0;i<CommonData.SCREEN_HEIGHT;i+=2) {
        g.drawLine(0,i,CommonData.SCREEN_WIDTH,i);
      }

      g.setColor(0x0FFFF00);
      g.setFont(font);
      g.drawString("Reanudar", 0, CommonData.SCREEN_HEIGHT,
                   Graphics.LEFT | Graphics.BOTTOM);
      g.drawString("Salir", CommonData.SCREEN_WIDTH,
                   CommonData.SCREEN_HEIGHT,
                   Graphics.RIGHT | Graphics.BOTTOM);
      g.drawString("PAUSADO", CommonData.SCREEN_WIDTH/2,
                   CommonData.SCREEN_HEIGHT/2,
                   Graphics.HCENTER | Graphics.BOTTOM);
    }
  }
}
```

Con los método `stop` y `start` definidos en esta clase podremos pausar y reanudar el juego. En el estado pausado se detiene el hilo del ciclo del juego y en la pantalla se muestra el mensaje de que el juego ha sido pausado.

Podemos ver también que este ciclo de juego es genérico, deberemos proporcionar una clase que implemente la interfaz `Scene` en la que se especificará el comportamiento y el aspecto de la escena. Esta interfaz tiene los siguientes métodos:

```
public interface Scene
{
  public void tick(int k);
  public void render(Graphics g);
}
```

Con `tick` nos referimos a cada actualización instantánea de la escena. En cada ciclo se invocará una vez este método `tick` para que la escena se actualice según la entrada del usuario y las interacciones entre los distintos objetos que haya en ella. Por otro lado, en render definiremos la forma de dibujar la escena.

La escena para nuestro clon de *Frogger* será la siguiente:

```
public class GameScene implements Scene {

  public static int E_INICIO = 0;
  public static int E_JUGANDO = 1;
```

```
  public static int E_MUERTO = 2;
  public static int E_CONSEGUIDO = 3;

  int numLives;
  int state;
  int timer;
  int stage;

  CrocSprite croc;
  Background bg;
  Tracks tracks;
  Image face;

  Vector cars;

  public GameScene() {

    croc = new CrocSprite();
    bg = new Background();
    tracks = new Tracks();
    face = Resources.getImage(Resources.IMG_FACE_LIVES);

    cars = new Vector();

    reset();
  }

  // Reiniciado de la partida
  public void reset() {
    numLives = CommonData.NUM_LIVES;
    stage = 0;

    reset(stage);
  }

  // Reiniciado de una fase
  public void reset(int nStage) {
    StageData stage = Resources.stageData[nStage];
    croc.reset();
    bg.reset(stage);
    tracks.reset(stage);

    cars.removeAllElements();

    this.setState(E_INICIO);
  }

  // Cambia el estado actual del juego
  public void setState(int state) {
    this.state = state;

    if(state==E_INICIO) {
      timer = 50;
    } else if(state==E_MUERTO) {
      timer = 30;
      croc.die();
    } else if(state==E_CONSEGUIDO) {
      timer = 50;
    }
```

```
}

public void tick(int keyState) {

  // Decrementa el tiempo de espera
  if(state==E_INICIO || state==E_MUERTO || state==E_CONSEGUIDO) {
    timer--;
  }

  // Comienza el juego
  if(state==E_INICIO && timer <= 0) {
    this.setState(E_JUGANDO);
  }

  // Reinicia el nivel o termina el juego
  if(state==E_MUERTO && timer <= 0) {
    numLives--;
    if(numLives<0) {
      Resources.midlet.showTitle();
    } else {
      this.reset(stage);
      return;
    }
  }

  // Pasa de fase
  if(state==E_CONSEGUIDO && timer <= 0) {
    stage++;
    if(stage >= Resources.stageData.length) {
      stage = 0;
    }

    this.reset(stage);
    return;
  }

  // Permite controlar el personaje si el estado es JUGANDO
  if(state==E_JUGANDO) {

    // Control del sprite
    if( (keyState & GameCanvas.UP_PRESSED)!=0
              && croc.getY() > 0 ) {
      croc.moveUp();
    } else if( (keyState & GameCanvas.DOWN_PRESSED)!=0
              && croc.getY() < CommonData.SCREEN_HEIGHT
                               - CommonData.SPRITE_HEIGHT) {
      croc.moveDown();
    } else if( (keyState & GameCanvas.LEFT_PRESSED)!=0
              && croc.getX() > 0) {
      croc.moveLeft();
    } else if( (keyState & GameCanvas.RIGHT_PRESSED)!=0
              && croc.getX() < CommonData.SCREEN_WIDTH
                               - CommonData.SPRITE_WIDTH) {
      croc.moveRight();
    } else {
      croc.stay();
    }
  }
```

```
    // Crea nuevos coches en los carriles si ha llegado el momento
    tracks.checkTracks(cars);

    // Actualiza coches y comprueba colisiones
    Enumeration enum = cars.elements();
    while(enum.hasMoreElements()) {
      CarSprite car = (CarSprite)enum.nextElement();
      car.tick();

      if(state == E_JUGANDO && car.collidesWith(croc, false)) {
        this.setState(E_MUERTO);
      }
    }

    // Ha conseguido cruzar la carretera
    if(state == E_JUGANDO && croc.getY()<CommonData.SPRITE_END_Y) {
      this.setState(E_CONSEGUIDO);
    }

    return;
  }

  public void render(Graphics g) {

    bg.paint(g);
    croc.paint(g);

    Enumeration enum = cars.elements();
    while(enum.hasMoreElements()) {
      CarSprite car = (CarSprite)enum.nextElement();
      car.paint(g);
    }

    for(int i=0;i<this.numLives;i++) {
      g.drawImage(face, i*CommonData.FACE_WIDTH, 0,
          Graphics.TOP | Graphics.LEFT);
    }

    if(state==E_INICIO) {
      g.setFont(CommonData.STAGE_TITLE_FONT);
      g.setColor(CommonData.STAGE_TITLE_COLOR);
      g.drawString(Resources.stageData[stage].title,
          CommonData.STAGE_TITLE_X, CommonData.STAGE_TITLE_Y,
          Graphics.HCENTER | Graphics.TOP);
    }

    if(state==E_CONSEGUIDO) {
      g.setFont(CommonData.STAGE_TITLE_FONT);
      g.setColor(CommonData.STAGE_TITLE_COLOR);
      g.drawString(CommonData.STAGE_COMPLETED_TEXT,
          CommonData.STAGE_TITLE_X, CommonData.STAGE_TITLE_Y,
          Graphics.HCENTER | Graphics.TOP);
    }
  }
}
```

Los métodos `reset` nos permiten reiniciar el juego. Si no proporcionamos ningún parámetro, se reiniciará el juego desde el principio, nos servirá para empezar una nueva

partida. Si proporcionamos como parámetro un número de fase, comenzaremos a jugar desde esa fase con las vidas que nos queden actualmente. Esta segunda forma nos servirá para empezar de nuevo una fase cuando nos hayan matado, o cuando hayamos pasado al siguiente nivel.

De esta forma no tendremos que instanciar nuevos objetos, sino que simplemente llamaremos a este método para reiniciar los valores del objeto actual.

Por otro lado podemos observar en este ejemplo cómo se ha modelado la máquina de estados para el juego. Tenemos los estados `E_INICIO`, `E_JUGANDO`, `E_MUERTO` y `E_CONSEGUIDO`. El estado `E_JUGANDO` durará hasta que nos maten o hayamos pasado a la siguiente fase. Los demás estados durarán un tiempo fijo durante el cual se mostrará algún gráfico o mensaje en pantalla, y una vez pasado este tiempo se pasará a otro estado.

Podemos ver que en `tick` es donde se implementa toda la lógica del juego. Dentro de este método podemos ver las distintas comprobaciones y acciones que se realizan, que variarán según el estado actual.

En `render` podemos ver también algunos elementos que se dibujarán sólo en determinados estados.