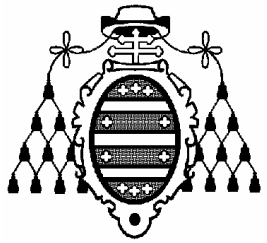
# UNIVERSIDAD DE OVIEDO



**ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA EN INFORMÁTICA DE OVIEDO**

**INFORMÁTICA MÓVIL**

# Práctica 3:
# Gestión avanzada de la interfaz de usuario

(Java Micro Edition)

**CURSO 2007-2008**

# TABLA DE CONTENIDOS

# 1. OBJETIVOS

Esta sesión práctica pretende completar los conceptos de implementación de interfaces de usuario en JavaME mediante el uso de la API de bajo nivel. Puesto que el tiempo para ello es limitado, y las posibilidades de este API son muchas, en el último apartado de este guión se comentan ciertos trabajos voluntarios para que el alumno profundice fuera del horario de clase con todas las posibilidades de este API.

# 2. GESTIÓN DE LA PANTALLA A BAJO NIVEL

El API de gestión de la interfaz de usuario a bajo nivel en JavaME, al contrario que el API de alto nivel, permite el control de la pantalla de un MID a nivel de pixels. Esto se consigue utilizando la clase *Canvas*, incluida en el paquete *javax.microedition.lcdui*, tal y como se comentó en las clases de teoría.

Un objeto *Canvas* gestiona el dibujo de los pixels de la pantalla mediante el método *paint()*. Este método incluirá las llamadas pertinentes a los métodos de las clases relacionadas con el dibujo de la pantalla a bajo nivel. El método *paint()* no es llamado nunca de forma explícita sino que se trata de un método callback: cuando el AMS (el Gestor de Aplicaciones) determina que es necesario dibujar el contenido de la pantalla, éste invoca el método *paint()* de la clase *Canvas* (esto tiene lugar cuando una instancia de esta clase es mostrada en la pantalla mediante *Display.setCurrent()*).

El único parámetro que recibe el método *paint()* es un objeto de la clase *Graphics*. Esta clase es la que proporciona todos los métodos que dibujarán el contenido de la pantalla. Algunos de estos métodos son:

- *drawLine():* para dibujar líneas.
- *drawRect()*: para dibujar un rectángulo.
- *fillRect()*: para dibujar un área rectángular.
- *drawArc()*: para dibujar un arco (y, por extensión, una elipse o una circunferencia)
- *drawString()*: para dibujar cadenas de texto.

Las aplicaciones pueden también repintar explícitamente la pantalla mediante el uso del método *Canvas.repaint()*, que notificará al AMS la necesidad de pintar la pantalla, invocando éste entonces al método *Canvas.paint()*. Hay que tener en cuenta que esta invocación no tiene porqué ser efectuada de forma instantánea; es posible que exista cierto retardo debido a que el sistema se encuentre procesando un evento que hubiese llegado anteriormente. Podría ocurrir que el sistema recogiese varias peticiones de repintado consecutivas antes de llamar una sola vez a *Canvas.paint()*. Existen formas de tratar adecuadamente estos retardos para, por ejemplo, efectuar animaciones (uso de *Display.callSerially()* para notificar la necesidad de repintar, efectuar la notificación desde un hilo de ejecución aparte o utilizar el método *serviceRepaints()* para un repintado inmediato).

La clase *Canvas* proporciona también algunos métodos de entrada de tipo callback (son llamados cuando el usuario presiona o libera una tecla, toca la pantalla con un dispositivo puntero, etc.).

## 2.1. EJEMPLO SENCILLO DE DIBUJO EN PANTALLA

El siguiente ejemplo intenta servir como ejemplo inicial de uso de las clases *Canvas* y

*Graphics*. Pretende dibujar una cara sonriente en pantalla, adecuándola a las dimensiones de la misma.

Primero, crearemos una suite de MIDlets en la que crear los distintos ejemplos de uso de las capacidades gráficas de JavaME. Podemos darle a la suite el nombre *PruebasGraficos*, con lo que inicialmente se crearía un MIDlet vacío *PruebasGraficosMIDlet*. Antes de crear nuestra primera aplicación de ejemplo, será necesario añadir a la suite una nueva clase (llamada *MiCanvas*), que heredará de *Canvas* e implementará el método *paint()*.

En resumen, habrá que implementar un MIDLet, que simplemente creará y presentará un objeto *MiCanvas*:

```
Display pantalla = Display.getDisplay(this);
Canvas miPrimerCanvas = new MiCanvas();
pantalla.setCurrent(miPrimerCanvas);
```

La clase MIDP *MiCanvas*, que heredará de *Canvas*, ha de implementar el método *paint()*. Dentro de *paint()* se invocarán a los métodos de la clase *Graphics* que crearán una figura como la siguiente:


**Ilustración 1: Figura a representar mediante el API de bajo nivel con *MiCanvas.paint()***

Para crear esta figura debes hacer las siguientes acciones:

- Poner todo el fondo de la pantalla en color amarillo (creando un área rectángular a partir de las coordenadas (0,0) y con ancho *MiCanvas.getWidth()* y alto *MiCanvas.getHeight()*).

- Crear un área circular de color verde inscrita en un cuadrado con origen en (20,20) y lado de 150 pixels. Crear el círculo que delimita ese área con línea discontínua (utilizar para ello, *Graphics.setStrokeStyle()*).

- Dibujar el contorno de los ojos en negro, relleno de blanco y con otro círculo interior en negro. El cuadrado en el que se inscribe el ojo izquierdo tiene origen en (30,50) y 60 pixels de lado. El ojo derecho es idéntico, pero con origen en (100,50). Los círculos negros interiores están inscritos en cuadrados con origen en (50,70) y (120,30), respectivamente, y 20 pixels de lado.

- Para la nariz, se trazarán dos líneas: la primera desde (95, 80) hasta (80, 120); la segunda va desde (95,80) hasta (110,120). El arco inferior de la nariz está inscrito en un rectángulo con origen en (80, 110), 30 pixels de ancho y 20 pixels de alto.

- La boca es un arco inscrito en el rectángulo con origen en (70, 120), ancho = 50 y altura = 40.

### *2.1.1. Uso de Commands en un Canvas*

Un *Canvas* puede tener asociados varios *Commands*, de la misma forma que un *Form*. Prueba a añadir un *Command* para salir de la aplicación.

### *2.1.2. Ocultando un Canvas*

Es evidente que puede ser necesario llevar un *Canvas* al fondo (llevando al frente otro *Displayable* mediante *Display.setCurrent()*) para ir conmutando las distintas pantallas o diálogos que compongan nuestras aplicaciones. Cuando esto ocurre, el *Canvas* llevado al fondo recibe un evento, que desencadenará la ejecución del método *hideNotify()*. En el instante en que el *Canvas* sea llevado al frente de nuevo, se ejecutará el método *showNotify()* como respuesta al evento correspondiente.

Crea un *Form* en la aplicación que simplemente mostrará un texto a través de un *StringItem* (por ejemplo, “Hola, mundo”). Inicialmente, la aplicación seguirá mostrando el *Canvas* con la cara de nuestro amigo. Sin embargo, se añadirá al *Canvas* un nuevo *Command* (del tipo *ITEM*) para hacer que se lleve al frente el *Form*. En el *Form*, otro *Command* permitirá volver a mostrar el *Canvas*. La Ilustración 2 muestra el aspecto y comportamiento de la aplicación.


**Ilustración 2: Conmutación entre distintos objetos Displayable**

A continuación, añadiremos a la clase *MiCanvas* los métodos *showNotify()* y *hideNotify()*, de forma que cada uno de ellos muestre un mensaje por la salida estándar (mediante *System.out.println()*). Observa como, al pulsar los *Command*s "Mostrar Form" y "Mostrar Canvas", los mensajes aparecen en la pestaña "*PruebasGraficosMIDlet – IO*" del IDE.

## 2.2. CLIPPING

El siguiente aspecto a practicar es el uso de la técnica de clipping, que será ilustrada mediante un ejemplo muy sencillo:

- Prueba a fijar, al final del método *MiCanvas.paint()*, el rectángulo objeto del clipping: un rectángulo centrado en la pantalla y cuyo lado sea de la mitad de pixels que el mínimo entre el alto y el ancho de la pantalla. El código para hacer esto sería:

```
int m = Math.min(this.getWidth(), this.getHeight());
graphics.setClip (m / 4, m / 4, m / 2, m / 2);
```

- A continuación, dibuja una línea desde (0,0) hasta (m, m).

- Observa cómo sólo aparece la parte de la línea que está dentro del área de clipping

Esta técnica es muy útil para redibujar las partes de la pantalla que sea necesario repintar y no toda la pantalla entera. Por ejemplo, es posible dibujar un diálogo emergente con el API de bajo nivel (una especie de *MessageBox*). Dicho diálogo estará inscrito en un rectángulo y, cuando el diálogo desaparezca, habrá que repintar lo que había "debajo" de él. En lugar de repintar toda la pantalla, bastaría con fijar un rectángulo de clipping con el mismo origen,

ancho y altura que el diálogo eliminado y repintar la pantalla. De este modo, sólo se repintaría la parte de la pantalla que el diálogo borró.

## 2.3. SISTEMA DE COORDENADAS

El sistema de coordenadas por defecto de la pantalla tiene su origen, como hemos estado viendo y utilizando, en la esquina superior izquierda de la pantalla. El píxel ubicado en esa posición es considerado como origen de coordenadas y sus coordenadas son (0,0). El primer elemento de la dupla indica el desplazamiento en pixels en sentido horizontal alejándose del origen de coordenadas. El segundo elemento indica, de forma análoga, el desplazamiento en pixels y en sentido vertical de un punto de la pantalla respecto del origen de coordenadas.

En ciertas aplicaciones, puede ser más útil usar otro punto de la pantalla como origen de coordenadas (por ejemplo, el centro de la pantalla, cuyas coordenadas son (*getWidth()*/2, *getHeight()*/2) para cualquier formato de pantalla de un MID. Prueba a trasladar a esa posición el origen del sistema de coordenadas y modifica la aplicación para que muestre lo mismo que venía mostrando. Ten en cuenta que los pixels por encima del punto central de la pantalla tendrán una componente Y negativa y que los pixels a la izquierda de dicho punto tendrán una componente X negativa.

## 2.4. IMPRESIÓN DE TEXTO

Crea un nuevo MIDlet *ImprimeTexto* en el MIDlet *PruebasGraficos* y juega con la impresión de textos, utilizando como referencia las diapositivas de clase de teoría. Puedes utilizar la clase *Font* para variar el estilo y tamaño de la fuente.

## 2.5. INTERACCIÓN DEL USUARIO CON UN MIDLET

Tal y como se ha probado en el apartado 2.1.1, un objeto *Canvas* (como el resto de *Displayable* vistos en el API de alto nivel) puede atender a la interacción del usuario mediante *Command*s. Además, *Canvas* permite atender tanto entradas directas de teclado como eventos de puntero.

La gestión de este tipo de entrada de usuario (y también de los *Command*s) se implementa mediante un modelo de eventos. Todos los eventos, los anteriores y otros como el evento de repintado o el de notificación de mostrar/ocultar un *Canvas*, son tratados en serie. Esto implica que el AMS no llamará a ninguno de los métodos encargados de responder ante cada uno de los tipos de eventos, hasta que el anterior método de manejo de evento haya retornado. Por este motivo, todos estos métodos deberían retornar tan pronto como sea possible, o crear un nuevo hilo encargado de ejecutar la tarea a realizar.

### *2.5.1. Entrada directa de teclado*

Como se comentó en teoría, `Canvas` proporciona tres métodos callback: *keyPressed()*, *keyReleased()*, y *keyRepeated()*. Estos métodos son ejecutados cuando una tecla del móvil es pulsada, se mantiene pulsada por un periodo largo de tiempo y cuando la tecla es "soltada", respectivamente. Todos ellos proporcionan un parámetro entero: el código de carácter Unicode asociado a la tecla en cuestión. MIDP define una serie de constantes simbólicas asociadas a las teclas estándar de un teclado, que pueden ser comprobadas en las diapositivas de teoría.

Además, algunas de las teclas tienen un significado adicional en los juegos (constantes *UP*, *DOWN*, *LEFT*, …; comentadas también en la teoría de la asignatura). Este mapeo de teclas de juego es dependiente del dispositivo, pero los métodos *getGameAction()* y *getKeyCode()* permiten traducir códigos de juego a códigos Unicode y viceversa.

Se puede incluso obtener el String asociado a una tecla de juego:

```
System.out.println("Pulse "+getKeyName (getKeyCode (FIRE)));
```

ofrecerá como salida

```
Pulse FIRE
```

Como ejemplo, crea un nuevo MIDlet (*MoverPunto*) en el que se muestre inicialmente un *Canvas* que muestre un pequeño círculo en el centro de la pantalla. Para implementar el *Canvas*, habrá también que crear una nueva clase MIDP que herede de Canvas (*MuevePunto*).

MuevePunto debe contener dos miembros públicos *int keyCode* e *int tipoEvento*. También implementará los métodos *keyPressed()*, *keyReleased()* y *keyRepeated()*. Todos ellos son métodos públicos que retornan *void* y reciben como parámetro un entero con el código Unicode asociado a la tecla correspondiente. Se pretende que se almacene en el miembro privado *keyCode* el código de tecla pulsada y, en *tipoEvento*, el tipo de evento (0 para tecla pulsada, 1 para tecla soltada y 2 para tecla repetida). A continuación, se debería repintar la pantalla. El ejemplo para *keyPressed()* es mostrado en el siguiente listado:

```
public void keyPressed (int keyCode) {
  eventType = 0;
  this.keyCode = keyCode;
  repaint ();
}
```

Además, el método *paint()* deberá mostrar el punto en las coordenadas correctas:

- Inicialmente, en el punto central de la pantalla.
- Cuando se pulse una tecla, se desplazará un píxel una posición arriba, abajo, a izquierda, o a derecha, según la tecla pulsada.

### *2.5.2. Detección de eventos de puntero*

Tan solo los dispositivos MID de gama más alta ofrecen un lapiz puntero, y por ello se dejará de lado la práctica de los eventos de gestión de eventos. Queda como trabajo adicional voluntario la práctica de este tipo de entrada de datos, mediante el uso de los métodos *pointerPressed()*, *pointerDragged()*, y *pointerReleased()*.

## 2.6. ANIMACIÓN

La animación de gráficos suele acarrear dos problemas: la sincronización de la pantalla con los cálculos de los nuevos "fotogramas" de la animación, por una parte; y el parpadeo de la pantalla, por otro.

### *2.6.1. Sincronización y renderización*

Tal y como se propuso en el epígrafe 2.5.1, la forma de hacer animaciones hasta el momento, consiste en calcular el contenido de la pantalla ante una entrada de usuario y llamar posteriormente a *repaint()* para pintar el nuevo "fotograma". Esta estrategia no tiene en cuenta que, cuando solicitemos repintar, se podría estar procesando aún la petición de repintado anterior.

Una posibilidad para solucionar el problema puede ser utilizar *Canvas.serviceRepaints()*, que se bloquea hasta que todas las actualizaciones pendientes de la pantalla finalicen. Esta posibilidad puede ser útil en nuestras aplicaciones monohilo, pero si se diseña la aplicación de forma que *paint()* sea llamado desde otro hilo habrá que tener cuidado con el uso de cerrojos u otros mecanismos similares de sincronización, puesto que pueden producirse bloqueos (por ejemplo, *serviceRepaints()* podría apropiarse de un cerrojo necesario para el hilo que ejecuta *paint()*).

Otra alternativa, sería utilizar *callSerially()* al final de *paint()*, pero también exige un planteamiento multihilo, por lo que excede los objetivos de esta asignatura.

Como trabajo voluntario, intenta sincronizar la aplicación encargada de mover el punto por la pantalla mediante el uso de *serviceRepaints()*. Puedes también utilizar la hora del sistema para que, una vez modificadas las nuevas coordenadas del punto, sólo se repinte si pasó 1/10 de segundo desde el final de la anterior ejecución de *paint()*. De esta forma, podrás conseguir fijar un "frame rate" de 10 cuadros ("fotogramas" o "frames") por segundo.

### *2.6.2. Parpadeo*

En algunos dispositivos, este tipo de implementaciones pueden mostrar parpadeos en la pantalla, debido a que toda la pantalla es dibujada completamente y con mucha frecuencia para crear nuevos "fotogramas" que proporcionen la sensación de movimiento.

Para aliviar este problema, desde los primeros tiempos de la industria del videojuego se ideó la técnica de "double buffering". Esta técnica se basa en la idea de utilizar un área de memoria para calcular los valores de los pixels del nuevo "fotograma" a generar (en los manejadores de eventos de entrada de usuario, por ejemplo) y volcar luego (en el método *paint()*) ese área de una sola vez por completo a la pantalla.

El método *Canvas.isDoubleBuffered()* permite determinar si la pantalla del dispositivo soporta de forma automática el doble buffer. Si este método devuelve *true*, no tiene sentido preocuparse pues el propio MID implementa automáticamente la técnica de doble buffer (las acciones de dibujo se hacen sobre un área de memoria auxiliar del MID y, al salir de *paint()*, se vuelca automáticamente el contenido de esa área a la pantalla). Si devuelve *false*, debería crearse un objeto *Image* del tamaño de la pantalla:

```
Image bufferAux = null;

if (isDoubleBuffered () == false)
   bufferAux = Image.createImage (getWidth (), getHeight ());
```

En el método *paint()*, se crearía un objeto *Graphics* en el ámbito de la función, de forma que si *bufferAux* es null se le asigne el *Graphics* pasado como parámetro a *paint()*, y en caso contrario, se cree el objeto a partir del *Image* creado anteriormente:

```
public void paint (Graphics g) {
  Graphics grafAux;

  if bufferAux==null
      grafAux = g;
  else
     grafAux = bufferAux.getGraphics ();

/* …. Dibujar sobre grafAux …. */

  if (bufferAux != null)
    g.drawImage (graphAux, 0, 0, Graphics.TOP | Graphics.RIGHT);
}
```

Prueba a implementar *paint()* de esta forma y ejecuta en dos emuladores (con y sin doble buffer integrado) la aplicación.

## 3. TRABAJO VOLUNTARIO: AMPLIACIONES

Se proponen las siguientes ampliaciones en para profundizar en el uso del API de bajo nivel:

- Intenta crear la aplicación de Contactos con la API de bajo nivel o "combinando" ambas. Crea algún formulario propio con este API (por ejemplo, la pantalla de navegación) y utiliza la entrada directa de teclado en él. Puedes añadir[1] una foto del contacto en esa pantalla de navegación, mediante el uso de un objeto *Image*[2].

- Adapta las aplicaciones con API de bajo nivel de forma que funcionen en pantallas de color o con niveles de gris. Utiliza los métodos comentados en teoría (*Display.isColor()*, *Display.numColors()*, etc.).

[1] Para que las fotos estén disponibles para la aplicación, habrá que introducir las distintas fotos en el fichero *JAR*.
[2] Ten en cuenta que crear un Image con Image.create(int ancho, int alto) implica que se puede usar la imagen para dibujar pixels sobre ella, como se ha visto en 2.6.2. Si, en cambio, se crea a partir de un fichero gráfico (el único formato obligatorio en MIDP 1.0 es el formato PNG), no se puede modificar el valor de los pixels de la imagen. Por ello, se habla de **imágenes mutables y no mutables**.

- Investiga en la documentación del API de J2ME el uso de las clases y métodos disponibles en MIDP 2.0 para la creación de juegos: *GameCanvas*, *Layer*, *Sprite*, *TiledLayer*, *LayerManager*.