



Sesión 12: Gráficos avanzados



- Gráficos en LCDUI
- Contexto gráfico
- Animaciones
- Eventos de entrada



- Gráficos en LCDUI
- Contexto gráfico
- Animaciones
- Eventos de entrada

- Con la API de bajo nivel podremos crear componentes personalizados
 - Adecuado para juegos
 - Se reduce la portabilidad
- Utilizaremos el *displayable* **Canvas**
 - Consiste en una pantalla vacía
 - Deberemos especificar lo que se mostrará en él
 - Controlaremos la interacción con el usuario a bajo nivel
- Nos permitirá dibujar el contenido que queramos
 - Se hará de forma similar a J2SE
 - Utilizaremos un objeto **Graphics** para dibujar en pantalla

Creación de un canvas



- **Debemos crear una clase que herede de Canvas**

```
public class MiCanvas extends Canvas {  
    public void paint(Graphics g) {  
        // Dibujamos en la pantalla  
        // usando el objeto g proporcionado  
    }  
}
```

- **Render pasivo**
 - No controlamos el momento en el que se dibujan los gráficos
 - Sólo definimos la forma de dibujarlos en el método **paint**
 - El sistema invocará este método cuando necesite dibujar nuestro componente

Propiedades del canvas



- Según el dispositivo el canvas tendrá distinta resolución
- Podemos consultar la resolución con

```
int ancho = getWidth();  
int alto = getHeight();
```

- El canvas no suele ocupar toda la pantalla
 - Se reserva un área para el dispositivo
 - Cobertura, título de la pantalla, comandos, etc
- En MIDP 2.0 podemos utilizar la pantalla completa

```
setFullScreenMode(true);
```



- Gráficos en LCDUI
- Contexto gráfico
- Animaciones
- Eventos de entrada

Atributos del contexto



- El objeto **Graphics** representa el contexto gráfico
 - Nos permitirá dibujar contenido en la pantalla
- El contexto tiene asociado atributos
 - Color del lápiz

```
g.setColor(0x00FF99); // Color codificado en 0xRRGGBB
```

- Tipo del lápiz (sólido o punteado)

```
g.setStrokeStyle(Graphics.SOLID); // o Graphics.DOTTED
```

- Fuente de texto

```
g.setFont(fuente); // Utilizamos objetos de la clase Font
```

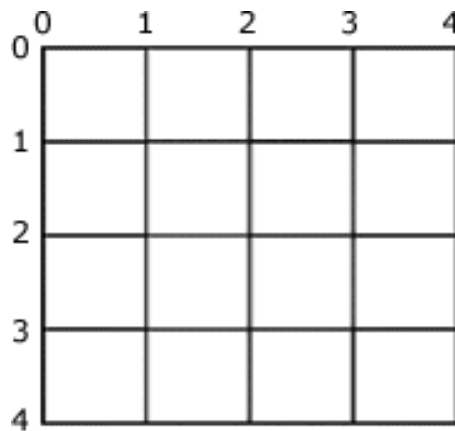
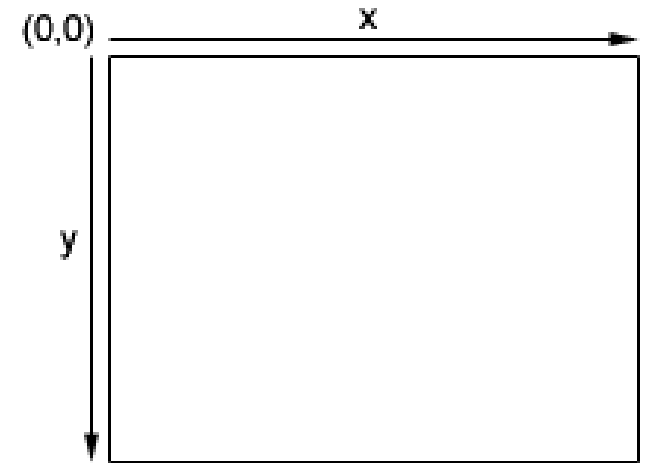
- Área de recorte

```
g.setClip(x, y, ancho, alto);
```

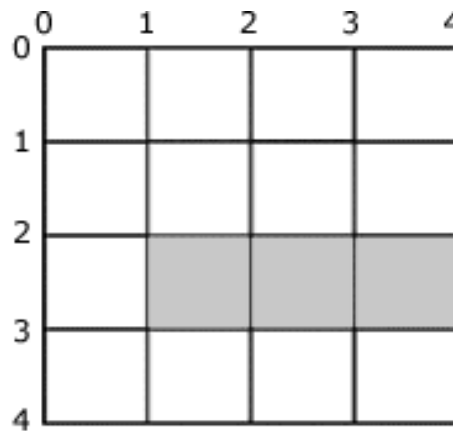
- Origen de coordenadas

```
g.translate(x, y);
```

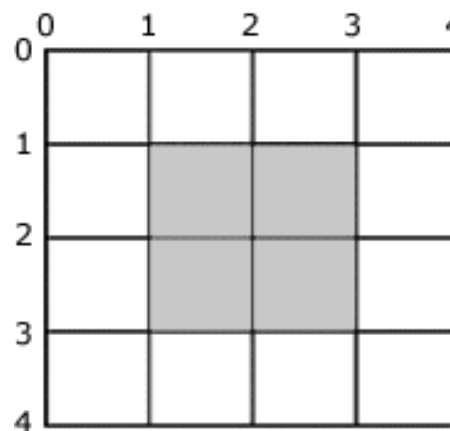

- La esquina superior izquierda tiene coordenadas (0,0)
 - Las X son positivas hacia la derecha
 - Las Y son positivas hacia abajo
- Las coordenadas corresponden a límites entre píxeles



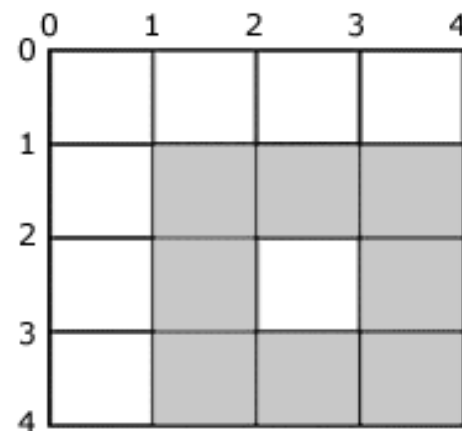
Matriz de píxeles vacía



Línea de (1,2) a (3,2)



Rectángulo relleno
de (1,1) a (3,3)



Rectángulo sin relleno
de (1,1) a (3,3)

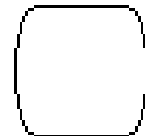
Dibujado de primitivas geométricas



■ Podemos dibujar distintas primitivas geométricas:

➤ Líneas

```
g.drawLine(x1, y1, x2, y2);
```



➤ Rectángulos

```
g.drawRect(x, y, ancho, alto);  
g.fillRect(x, y, ancho, alto);
```



➤ Rectángulos redondeados

```
g.drawRoundRect(x, y, ancho, alto, wArco, hArco);  
g.fillRoundRect(x, y, ancho, alto, wArco, hArco);
```

Rectángulo

Rectángulo redondeado



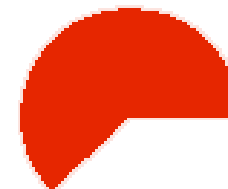
SOLID



DOTTED

➤ Arcos

```
g.drawArc(x, y, ancho, alto, iniArco, arco);  
g.fillArc(x, y, ancho, alto, iniArco, arco);
```



Arco

Líneas

Puntos anchor



- **Nos sirven para ubicar elementos en la pantalla**
 - Lo utilizaremos para texto e imágenes
- **Especificaremos**
 - Coordenadas de la pantalla (x,y)
 - Qué posición del elemento se ubicará en dichas coordenadas
- **Esta posición puede ser:**
 - Para la horizontal:

Graphics.LEFT

Graphics.HCENTER

Graphics.RIGHT

- Para la vertical

Graphics.TOP


Graphics.VCENTER

Graphics.BASELINE

Graphics.BOTTOM

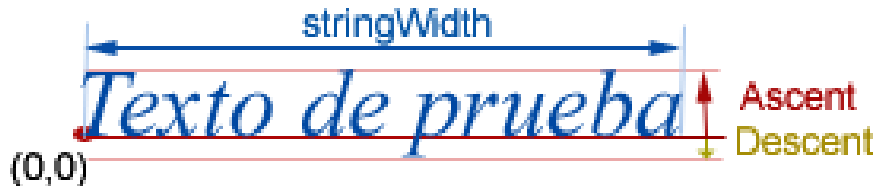
- Dibujamos texto con:

```
g.drawString(cadena, x, y, anchor);
```



```
g.drawString("Texto de prueba", 0, 0, Graphics.LEFT|Graphics.BASELINE);
```

- Podemos necesitar las medidas del texto en píxeles
 - La clase **Font** de la fuente utilizada nos proporciona esa información



- Podemos dibujar tanto imágenes mutables como inmutables
- Dibujaremos la imagen en pantalla con:

```
g.drawImage(img, x, y, anchor);
```

- Por ejemplo:

```
g.drawImage(img, 0, 0, Graphics.TOP|Graphics.LEFT);
```

- En el caso de las imágenes mutables, editaremos su contenido utilizando su contexto gráfico

```
Graphics offg = img_mut.getGraphics();
```

- Se utilizará igual que cuando se dibuja en pantalla
- En este caso los gráficos se dibujan en la imagen en memoria



- Gráficos en LCDUI
- Contexto gráfico
- Animaciones
- Eventos de entrada

- Para crear una animación tendremos que modificar el contenido de la pantalla con el tiempo
- Debemos solicitar al sistema que redibuje

```
repaint();
```

- Una vez hecho esto, cuando el sistema tenga tiempo redibujará la pantalla invocando nuestro método `paint`
- Si sólo hemos modificado un área, podemos solicitar el redibujado sólo de este área

```
repaint(x, y, ancho, alto);
```

Técnica del doble buffer



- **Para mostrar cada frame de la animación debemos**
 - **Borrar el frame anterior**
 - **Dibujar el nuevo frame**
- **Al hacer esto repetidas veces puede producirse un efecto de “parpadeo” en la pantalla**
- **Para evitarlo podemos utilizar la técnica del doble buffer**
 - **Dibujamos todo el contenido en una imagen mutable del mismo tamaño de la pantalla**
 - **Volcamos la imagen a la pantalla como una unidad**
- **Muchos dispositivos ya implementan esta técnica**
 - **Con `isDoubleBuffered()` sabremos si lo implementa el dispositivo**
 - **Si no lo implementa el dispositivo, deberíamos hacerlo nosotros**

- Creamos un hilo que cada cierto intervalo:
 - Modifique las propiedades de los objetos a dibujar
 - Por ejemplo su posición (x,y)
 - Llame a `repaint` para solicitar el redibujado de la pantalla

```
public void run() {  
    // El rectangulo comienza en (10,10)  
    x = 10; y = 10;  
    while(x < 100) {  
        x++;  
        repaint();  
        try {  
            Thread.sleep(100);  
        } catch(InterruptedException e) {}  
    }  
}
```

- Para poner en marcha el hilo podemos utilizar el evento **showNotify** del **Canvas** por ejemplo

➤ Este evento se produce cuando el **Canvas** se muestra

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

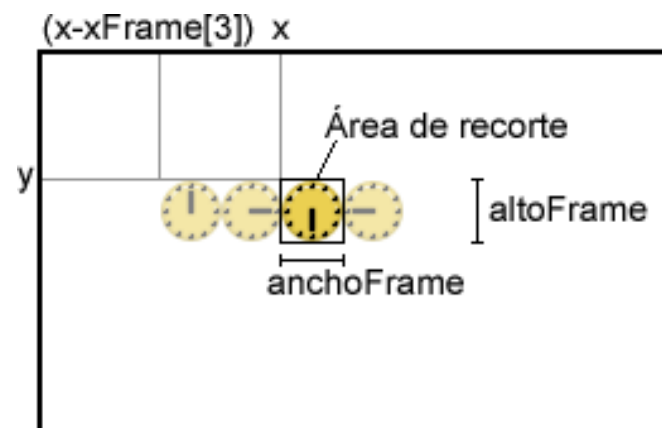
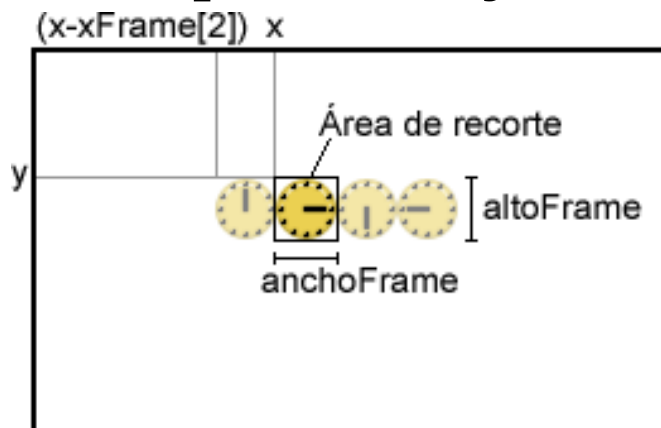
➤ Podemos utilizar **hideNotify** para detenerlo

- En los eventos deberemos devolver el control inmediatamente
 - Si necesitamos realizar una operación de larga duración, crearemos un hilo que la realice como en este caso
 - Si no devolviésemos el control, se bloquearía el hilo de eventos y la aplicación dejaría de responder
 - No actualizaría los gráficos, no leería la entrada del usuario, etc

- Si queremos mostrar una imagen animada necesitamos tener varios frames
 - Para evitar tener varias imágenes, podemos guardar todos los frames en una misma imagen



- Podemos utilizar un área de recorte para seleccionar el frame que se dibuja en cada momento





- Gráficos en LCDUI
- Contexto gráfico
- Animaciones
- Eventos de entrada

Eventos del teclado



- Con el canvas tenemos acceso a los eventos a bajo nivel
 - Sabremos cuando el usuario pulsa o suelta una tecla
- Para dar respuesta a estos eventos debemos sobrescribir los siguientes métodos del **Canvas**

```
public class MiCanvas extends Canvas {  
    ...  
    public void keyPressed(int cod) {  
        // Se ha presionado la tecla con código cod  
    }  
    public void keyRepeated(int cod) {  
        // Se mantiene pulsada la tecla con código cod  
    }  
    public void keyReleased(int cod) {  
        // Se ha soltado la tecla con código cod  
    }  
}
```

Códigos de las teclas



- **Tenemos definido como constante los códigos de las teclas estándar**
 - **Utilizar estos códigos mejora la portabilidad**

<code>Canvas.KEY_NUM0</code>	0
<code>Canvas.KEY_NUM1</code>	1
<code>Canvas.KEY_NUM2</code>	2
<code>Canvas.KEY_NUM3</code>	3
<code>Canvas.KEY_NUM4</code>	4
<code>Canvas.KEY_NUM5</code>	5
<code>Canvas.KEY_NUM6</code>	6
<code>Canvas.KEY_NUM7</code>	7
<code>Canvas.KEY_NUM8</code>	8
<code>Canvas.KEY_NUM9</code>	9
<code>Canvas.KEY_POUND</code>	#
<code>Canvas.KEY_STAR</code>	*

Acciones de juegos



- Cada tecla tiene asociada una acción de juego
- Las acciones de juego son:

`Canvas.LEFT`

`Canvas.RIGHT`

`Canvas.UP`

`Canvas.DOWN`

`Canvas.FIRE`

- Podemos consultar la acción de juego asociada a una tecla

```
int accion = getGameAction(cod);
```

- Estas acciones mejoran la portabilidad en juegos

- **En dispositivos con puntero podremos recibir estos eventos**

```
public class MiCanvas extends Canvas {  
    ...  
    public void pointerPressed(int x, int y) {  
        // Se ha pinchado con el puntero en (x,y)  
    }  
    public void pointerDragged(int x, int y) {  
        // Se ha arrastrado el puntero a (x,y)  
    }  
    public void pointerReleased(int x, int y) {  
        // Se ha soltado el puntero en (x,y)  
    }  
}
```