Gráficos avanzados

Índice

Copyright © 2010 Depto. CCIA All rights reserved.

1. Interfaz gráfica de bajo nivel

Hasta ahora hemos visto la creación de aplicaciones con una interfaz gráfica creada a partir de una serie de componentes de alto nivel definidos en la API LCDUI (alertas, campos de texto, listas, formularios).

En este punto veremos como dibujar nuestros propios gráficos directamente en pantalla. Para ello Java nos proporciona acceso a bajo nivel al contexto gráfico del área donde vayamos a dibujar, permitiéndonos a través de éste modificar los *pixels* de este área, dibujar una serie de figuras geométricas, así como volcar imágenes en ella.

También podremos acceder a la entrada del usuario a bajo nivel, conociendo en todo momento cuándo el usuario pulsa o suelta cualquier tecla del móvil.

Este acceso a bajo nivel será necesario en aplicaciones como juegos, donde debemos tener un control absoluto sobre la entrada y sobre lo que dibujamos en pantalla en cada momento. El tener este mayor control tiene el inconveniente de que las aplicaciones serán menos portables, ya que dibujaremos los gráficos pensando en una determinada resolución de pantalla y un determinado tipo de teclado, pero cuando la queramos llevar a otro dispositivo en el que estos componentes sean distintos deberemos hacer cambios en el código.

Por esta razón para las aplicaciones que utilizan esta API a bajo nivel, como los juegos Java para móviles, encontramos distintas versiones para cada modelo de dispositivo. Dada la heterogeneidad de estos dispositivos, resulta más sencillo rehacer la aplicación para cada modelo distinto que realizar una aplicación adaptable a las características de cada modelo.

Al programar las aplicaciones deberemos facilitar en la medida de lo posible futuros cambios para adaptarla a otros modelos, permitiendo reutilizar la máxima cantidad de código posible.

1.1. Gráficos en LCDUI

La API de gráficos a bajo nivel de LCDUI es muy parecida a la existente en AWT, por lo que el aprendizaje de esta API para programadores que conozcan la de AWT va a ser casi inmediato.

Las clases que implementan la API de bajo nivel en LCDUI son `Canvas` y `Graphics`. Estas clases reciben el mismo nombre que las de AWT, y se utilizan de una forma muy parecida. Tienen alguna diferencia en cuanto a su interfaz para adaptarse a las necesidades de los dispositivos móviles.

El `Canvas` es un tipo de elemento *displayable* correspondiente a una pantalla vacía en la que nosotros podremos dibujar a bajo nivel el contenido que queramos. Además este

Copyright © 2010 Depto. CCIA All rights reserved.

componente nos permitirá leer los eventos de entrada del usuario a bajo nivel.

Esta pantalla del móvil tiene un contexto gráfico asociado que nosotros podremos utilizar para dibujar en ella. Este objeto encapsula el *raster* de pantalla (la matriz de *pixels* de los que se compone la pantalla) y además tiene una serie de atributos con los que podremos modificar la forma en la que se dibuja en este *raster*. Este contexto gráfico se definirá en un objeto de la clase `Graphics`. Este objeto nos ofrece una serie de métodos que nos permiten dibujar distintos elementos en pantalla. Más adelante veremos con detalle los métodos más importantes.

Este objeto `Graphics` para dibujar en la pantalla del dispositivo nos lo deberá proporcionar el sistema en el momento en que se vaya a dibujar, no podremos obtenerlo nosotros por nuestra cuenta de ninguna otra forma. Esto es lo que se conoce como *render* pasivo, definimos la forma en la que se dibuja pero es el sistema el que decidirá cuándo hacerlo.

1.1.1. Creación de un Canvas

Para definir la forma en la que se va a dibujar nuestro componente deberemos extender la clase `Canvas` redefiniendo su método `paint`. Dentro de este método es donde definiremos cómo se realiza el dibujado de la pantalla. Esto lo haremos de la siguiente forma:

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

Con esto en la clase `MiCanvas` hemos creado una pantalla en la que nosotros controlamos lo que se dibuja. Este método `paint` nunca debemos invocarlo nosotros, será el sistema el que se encargue de invocarlo cuando necesite dibujar el contenido de la pantalla. En ese momento se proporcionará como parámetro el objeto correspondiente al contexto gráfico de la pantalla del dispositivo, que podremos utilizar para dibujar en ella. Dentro de este método es donde definiremos cómo dibujar en la pantalla, utilizando para ello el objeto de contexto gráfico `Graphics`.

Siempre deberemos dibujar utilizando el objeto `Graphics` dentro del método `paint`. Guardarnos este objeto y utilizarlo después de haberse terminado de ejecutar `paint` puede producir un comportamiento indeterminado, y por lo tanto no debe hacerse nunca.

1.1.2. Propiedades del Canvas

Las pantallas de los dispositivos pueden tener distintas resoluciones. Además normalmente el área donde podemos dibujar no ocupa toda la pantalla, ya que el móvil utiliza una franja superior para mostrar información como la cobertura o el título de la aplicación, y en la franja inferior para mostrar los comandos disponibles.

Copyright © 2010 Depto. CCIA All rights reserved.

Sin embargo, a partir de MIDP 2.0 aparece la posibilidad de utilizar modo a pantalla completa, de forma que controlaremos el contenido de toda la pantalla. Para activar el modo a pantalla completa utilizaremos el siguiente método:

```
setFullScreenMode(true); // Solo disponible a partir de MIDP 2.0
```

Es probable que nos interese conocer desde dentro de nuestra aplicación el tamaño real del área del `Canvas` en la que podemos dibujar. Para ello tenemos los métodos `getWidth` y `getHeight` de la clase `Canvas`, que nos devolverán el ancho y el alto del área de dibujo respectivamente.

Para obtener información sobre el número de colores soportados deberemos utilizar la clase `Display` tal como vimos anteriormente, ya que el número de colores es propio de todo el visor y no sólo del área de dibujo.

1.1.3. Mostrar el Canvas

Podemos mostrar este componente en la pantalla del dispositivo igual que mostramos cualquier otro *displayable*:

```
MiCanvas mc = new MiCanvas();
mi_display.setCurrent(mc);
```

Es posible que queramos hacer que cuando se muestre este *canvas* se realice alguna acción, como por ejemplo poner en marcha alguna animación que se muestre en la pantalla. De la misma forma, cuando el *canvas* se deje de ver deberemos detener la animación. Para hacer esto deberemos tener constancia del momento en el que el *canvas* se muestra y se oculta.

Podremos saber esto debido a que los métodos `showNotify` y `hideNotify` de la clase `Canvas` serán invocados son invocados por el sistema cuando dicho componente se muestra o se oculta respectivamente. Nosotros podremos en nuestra subclase de `Canvas` redefinir estos métodos, que por defecto están vacíos, para definir en ellos el código que se debe ejecutar al mostrarse u ocultarse nuestro componente. Por ejemplo, si queremos poner en marcha o detener una animación, podemos redefinir los métodos como se muestra a continuación:

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

    public void showNotify() {
        // El Canvas se muestra
        comenzarAnimacion();
    }

    public void hideNotify() {
        // El Canvas se oculta
```


Copyright © 2010 Depto. CCIA All rights reserved.

```
            detenerAnimacion();
        }
}
```

De esta forma podemos utilizar estos dos métodos como respuesta a los eventos de aparición y ocultación del *canvas*.

1.2. Contexto gráfico

El objeto `Graphics` nos permitirá acceder al contexto gráfico de un determinado componente, en nuestro caso el *canvas*, y a través de él dibujar en el *raster* de este componente. En el caso del contexto gráfico del *canvas* de LCDUI este *raster* corresponderá a la pantalla del dispositivo móvil. Vamos a ver ahora como dibujar utilizando dicho objeto `Graphics`. Este objeto nos permitirá dibujar distintas primitivas geométricas, texto e imágenes.

1.2.1. Atributos

El contexto gráfico tendrá asociados una serie de atributos que indicarán cómo se va a dibujar en cada momento, como por ejemplo el color o el tipo del lápiz que usamos para dibujar. El objeto `Graphics` proporciona una serie de métodos para consultar o modificar estos atributos. Podemos encontrar los siguientes atributos en el contexto gráfico de LCDUI:

- **Color del lápiz**: Indica el color que se utilizará para dibujar la primitivas geométricas y el texto. MIDP trabaja con color de 24 bits (*truecolor*), que codificaremos en modelo RGB. Dentro de estos 24 bits tendremos 8 bits para cada uno de los tres componentes: rojo (R), verde (G) y azul (B). No tenemos canal *alpha*, por lo que no soportará transparencia. No podemos contar con que todos los dispositivos soporten color de 24 bits. Lo que hará cada implementación concreta de MIDP será convertir los colores solicitados en las aplicaciones al color más cercano soportado por el dispositivo.

  Podemos trabajar con los colores de dos formas distintas: tratando los componentes R, G y B por separado, o de forma conjunta. En MIDP desaparece la clase `Color` que teníamos en AWT, por lo que deberemos asignar los colores proporcionando directamente los valores numéricos del color.

  Si preferimos tratar los componentes de forma separada, tenemos los siguientes métodos para obtener o establecer el color actual del lápiz:

```
g.setColor(rojo, verde, azul);
int rojo = g.getRedComponent();
int green = g.getGreenComponent();
int blue = g.getBlueComponent();
```

  Donde `g` es el objeto `Graphics` del contexto donde vamos a dibujar. Estos componentes rojo, verde y azul tomarán valores entre 0 y 255.

Copyright © 2010 Depto. CCIA All rights reserved.

Podemos tratar estos componentes de forma conjunta empaquetándolos en un único entero. En hexadecimal se codifica de la siguiente forma:

```
0x00RRGGBB
```

Podremos leer o establecer el color utilizando este formato empaquetado con los siguientes métodos:

```
g.setColor(rgb);
int rgb = g.getColor();
```

Tenemos también métodos para trabajar con valores en escala de grises. Estos métodos nos pueden resultar útiles cuando trabajemos con dispositivos monocromos.

```
int gris = g.getGrayScale();
g.setGrayScale(gris);
```

Con estos métodos podemos establecer como color actual distintos tonos en la escala de grises. El valor de gris se moverá en el intervalo de 0 a 255. Si utilizamos `getGrayScale` teniendo establecido un color fuera de la escala de grises, convertirá este color a escala de grises obteniendo su brillo.

- **Tipo del lápiz**: Además del color del lápiz, también podemos establecer su tipo. El tipo del lápiz indicará cómo se dibujan las líneas de las primitivas geométricas. Podemos encontrar dos estilos:

`Graphics.SOLID`	Línea sólida (se dibujan todos los *pixels*)
`Graphics.DOTTED`	Línea punteada (se salta algunos *pixels* sin dibujarlos)

Podemos establecer el tipo del lápiz o consultarlo con los siguientes métodos:

```
int tipo = g.getStrokeStyle();
g.setStrokeStyle(tipo);
```

- **Fuente**: Indica la fuente que se utilizará para dibujar texto. Utilizaremos la clase `Font` para especificar la fuente de texto que vamos a utilizar, al igual que en AWT. Podemos obtener o establecer la fuente con los siguientes métodos:

```
Font fuente = g.getFont();
g.setFont(fuente);
```

- **Área de recorte**: Podemos definir un rectángulo de recorte. Cuando definimos un área de recorte en el contexto gráfico, sólo se dibujarán en pantalla los *pixels* que caigan dentro de este área. Nunca se dibujarán los *pixels* que escribamos fuera de este espacio. Para establecer el área de recorte podemos usar el siguiente método:

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

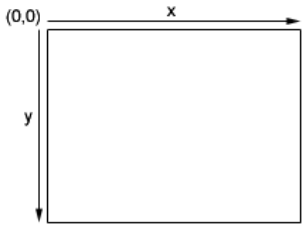
También tenemos disponible el siguiente método:

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


Copyright © 2010 Depto. CCIA All rights reserved.

Este método establece un recorte en el área de recorte anterior. Si ya existía un rectángulo de recorte, el nuevo rectángulo de recorte será la intersección de ambos. Si queremos eliminar el área de recorte anterior deberemos usar el método `setClip`.

- **Origen de coordenadas**: Indica el punto que se tomará como origen en el sistema de coordenadas del área de dibujo. Por defecto este sistema de coordenadas tendrá la coordenada (0,0) en su esquina superior izquierda, y las coordenadas serán positivas hacia la derecha (coordenada x) y hacia abajo (coordenada y), tal como se muestra a continuación:

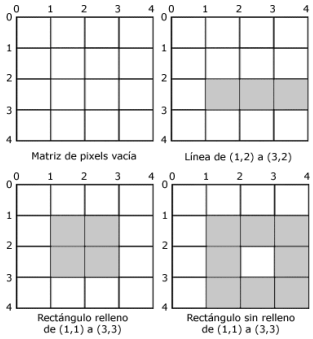


Sistema de coordenadas del área de dibujo

Podemos trasladar el origen de coordenadas utilizando el método `translate`. También tenemos métodos para obtener la traslación del origen de coordenadas.

```
int x = g.getTranslateX();
int y = g.getTranslateY();
g.translate(x, y);
```

Estas coordenadas no corresponden a *pixels*, sino a los límites de los *pixels*. De esta forma, el *píxel* de la esquina superior izquierda de la imagen se encontrará entre las coordenadas (0,0), (0,1), (1,0) y (1,1).



Coordenadas de los límites de los pixels

1.2.2. Dibujar primitivas geométricas

Una vez establecidos estos atributos en el contexto gráfico, podremos dibujar en él una serie de elementos utilizando una serie de métodos de `Graphics`. Vamos a ver en primer lugar cómo dibujar una serie de primitivas geométricas. Para ello tenemos una serie de métodos que comienzan por `draw_` para dibujar el contorno de una determinada figura, o

Copyright © 2010 Depto. CCIA All rights reserved.

`fill_` para dibujar dicha figura con relleno.

- **Líneas**: Dibuja una línea desde un punto *(x1,y1)* hasta *(x2,y2)*. Dibujaremos la línea con:

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

  En este caso no encontramos ningún método `fill` ya que las líneas no pueden tener relleno. Al dibujar una línea se dibujarán los *píxels* situados inmediatamente abajo y a la derecha de las coordenadas indicadas. Por ejemplo, si dibujamos con `drawLine(0, 0, 0, 0)` se dibujará el *pixel* de la esquina superior izquierda.

- **Rectángulos**: Podemos dibujar rectángulos especificando sus coordenadas y su altura y anchura. Podremos dibujar el rectángulo relleno o sólo el contorno:

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

  En el caso de `fillRect`, lo que hará será rellenar con el color actual los *pixels* situados entre las coordenadas limítrofes. En el caso de `drawRect`, la línea inferior y derecha se dibujarán justo debajo y a la derecha respectivamente de las coordenadas de dichos límites, es decir, se dibuja con un *pixel* más de ancho y de alto que en el caso relleno. Esto es poco intuitivo, pero se hace así para mantener la coherencia con el comportamiento de `drawLine`.

  Al menos, lo que siempre se nos asegura es que cuando utilizamos las mismas dimensiones no quede ningún hueco entre el dibujo del relleno y el del contorno.

  Podemos también dibujar rectángulos con las esquinas redondeadas, utilizando los métodos:

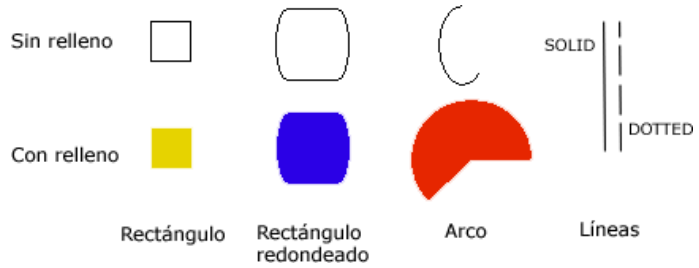
```
g.drawRoundRect(x, y, ancho, alto, ancho_arco, alto_arco);
g.fillRoundRect(x, y, ancho, alto, ancho_arco, alto_arco)
```

- **Arcos:** A diferencia de AWT, no tenemos un método para dibujar directamente elipses, sino que tenemos uno más genérico que nos permite dibujar arcos de cualquier tipo. Nos servirá tanto para dibujar elipses y círculos como para cualquier otro tipo de arco.

```
g.drawArc(x, y, ancho, alto, angulo_inicio, angulo_arco);
g.fillArc(x, y, ancho, alto, angulo_inicio, angulo_arco);
```

Los ángulos especificados deben estar en grados. Por ejemplo, si queremos dibujar un círculo o una elipse en `angulo_arco` pondremos un valor de 360 grados para que se cierre el arco. En el caso del círculo los valores de `ancho` y `alto` serán iguales, y en el caso de la elipse serán diferentes.

Copyright © 2010 Depto. CCIA All rights reserved.



Ejemplos de diferentes primitivas

Por ejemplo, el siguiente *canvas* aparecerá con un dibujo de un círculo rojo y un cuadrado verde:

```
public class MiCanvas extends Canvas {
        public void paint(Graphics g) {
                g.setColor(0x00FF0000);
                g.fillArc(10,10,50,50,0,360);
                g.setColor(0x0000FF00);
                g.fillRect(60,60,50,50);
        }
}
```

1.2.3. Puntos anchor

En MIDP se introduce una características no existente en AWT que son los puntos *anchor*. Estos puntos nos facilitarán el posicionamiento del texto y de las imágenes en la pantalla. Con los puntos *anchor*, además de dar una coordenada para posicionar estos elementos, diremos qué punto del elemento vamos a posicionar en dicha posición.

Para el posicionamiento horizontal tenemos las siguientes posibilidades:

`Graphics.LEFT`	En las coordenadas especificadas se posiciona la parte izquierda del texto o de la imagen.
`Graphics.HCENTER`	En las coordenadas especificadas se posiciona el centro del texto o de la imagen.
`Graphics.RIGHT`	En las coordenadas especificadas se posiciona la parte derecha del texto o de la imagen.

Para el posicionamiento vertical tenemos:

`Graphics.TOP`	En las coordenadas especificadas se posiciona la parte superior del texto o de la imagen.
`Graphics.VCENTER`	En las coordenadas especificadas se posiciona el centro de la imagen. No se aplica a texto.
`Graphics.BASELINE`	En las coordenadas especificadas se posiciona la línea de base del texto. No se aplica a imágenes.
`Graphics.BOTTOM`	En las coordenadas especificadas se posiciona

Copyright © 2010 Depto. CCIA All rights reserved.

	la parte inferior del texto o de la imagen.

1.2.4. Cadenas de texto

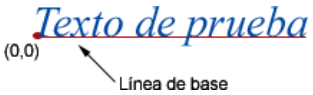
Podemos dibujar una cadena de texto utilizando el método `drawString`. Deberemos proporcionar la cadena de texto de dibujar y el punto *anchor* donde dibujarla.

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

Por ejemplo, si dibujamos la cadena con:

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

Este punto corresponderá al inicio de la cadena (lado izquierdo), en la línea de base del texto como se muestra a continuación:



Línea de base del texto

Con esto dibujaremos un texto en pantalla, pero es posible que nos interese conocer las coordenadas que limitan el texto, para saber exactamente el espacio que ocupa en el área de dibujo. En AWT podíamos usar para esto un objeto `FontMetrics`, pero este objeto no existe en MIDP. En MIDP la información sobre las métricas de la fuente está encapsulada en la misma clase `Font` por lo que será más sencillo acceder a esta información. Podemos obtener esta información utilizando los siguientes métodos de la clase `Font`:

- **`stringWidth(cadena)`**: Nos devuelve el ancho que tendrá la cadena *cadena* en *pixels*.
- **`getHeight()`**: Nos devuelve la altura de la fuente, es decir, la distancia entre las líneas de base de dos líneas consecutivas de texto. Llamamos ascenso (*ascent*) a la altura típica que suelen subir los caracteres desde la línea de base, y descenso (*descent*) a lo que suelen bajar desde esta línea. La altura será la suma del ascenso y el descenso de la fuente, más un margen para evitar que se junten los caracteres de las dos líneas. Es la distancia existente entre el punto superior (`TOP`) y el punto inferior (`BOTTOM`) de la cadena de texto.
- **`getBaselinePosition()`**: Nos devuelve el ascenso de la fuente, es decir, la altura típica desde la línea de base hasta la parte superior de la fuente.

Con estas medidas podremos conocer exactamente los límites de una cadena de texto, tal como se muestra a continuación:

Métricas del textu

Copyright © 2010 Depto. CCIA All rights reserved.

1.2.5. Imágenes

Hemos visto como crear imágenes y como utilizarlas en componentes de alto nivel. Estas mismas imágenes encapsuladas en la clase `Image`, podrán ser mostradas también en cualquier posición de nuestro área de dibujo.

Para ello utilizaremos el método:

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

En este caso podremos dibujar tanto imágenes mutables como inmutables.

Vimos que las imágenes mutables son aquellas cuyo contenido puede ser modificado. Vamos a ver ahora como hacer esta modificación. Las imágenes mutables, al igual que el *canvas*, tienen un contexto gráfico asociado. En el caso de las imágenes, este contexto gráfico representa el contenido de la imagen que es un *raster* en memoria, pero podremos dibujar en él igual que lo hacíamos en el *canvas*. Esto es así debido a que dibujaremos también mediante un objeto de la clase `Graphics`. Podemos obtener este objeto de contexto gráfico en cualquier momento invocando el método `getGraphics` de la imagen:

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

Si queremos modificar una imagen que hemos cargado de un fichero o de la red, y que por lo tanto es inmutable, podemos crear una copia mutable de la imagen para poder modificarla. Para hacer esto lo primero que deberemos hacer es crear la imagen mutable con el mismo tamaño que la inmutable que queremos copiar. Una vez creada podremos obtener su contexto gráfico, y dibujar en él la imagen inmutable, con lo que habremos hecho la copia de la imagen inmutable a una imagen mutable, que podrá ser modificada más adelante.

```
Image img_mut = Image.createImage(img.getWidth(), img.getHeight());
Graphics offg = img_mut.getGraphics();
offg.drawImage(img, 0, 0, Graphics.TOP|Graphics.LEFT);
```

1.3. Animación

Hasta ahora hemos visto como dibujar gráficos en pantalla, pero lo único que hacemos es definir un método que se encargue de dibujar el contenido del componente, y ese método será invocado cuando el sistema necesite dibujar la ventana.

Sin embargo puede interesarnos cambiar dinámicamente los gráficos de la pantalla para realizar una animación. Para ello deberemos indicar el momento en el que queremos que se redibujen los gráficos.

1.3.1. Redibujado del área

Para forzar que se redibuje el área de la pantalla deberemos llamar al método `repaint` del

Copyright © 2010 Depto. CCIA All rights reserved.

canvas. Con eso estamos solicitando al sistema que se repinte el contenido, pero no lo repinta en el mismo momento en el que se llama. El sistema introducirá esta solicitud en la cola de eventos pendientes y cuando tenga tiempo repintará su contenido.

```
MiCanvas mc = new MiCanvas();
...
mc.repaint();
```

En MIDP podemos forzar a que se realicen todos los repintados pendientes llamando al método `serviceRepaints`. La llamada a este método nos bloqueará hasta que se hayan realizado todos los repintados pendientes. Por esta razón deberemos tener cuidado de no causar un interbloqueo invocando a este método.

```
mc.serviceRepaints();
```

Para repintar el contenido de la pantalla el sistema llamará al método `paint`, en MIDP no existe el método `update` de AWT. Por lo tanto, deberemos definir dentro de `paint` qué se va a dibujar en la pantalla en cada instante, de forma que el contenido de la pantalla varíe con el tiempo y eso produzca el efecto de la animación.

Podemos optimizar el redibujado repintando únicamente el área de la pantalla que haya cambiado. Para ello en MIDP tenemos una variante del método `repaint` que nos permitirá hacer esto.

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

Utilizando este método, la próxima vez que se redibuje se invocará `paint` pero se proporcionará un objeto de contexto gráfico con un área de recorte establecida, correspondiente a la zona de la pantalla que hemos solicitado que se redibuje.

Al dibujar cada *frame* de la animación deberemos borrar el contenido del *frame* anterior para evitar que quede el rastro, o al menos borrar la zona de la pantalla donde haya cambios.

Imaginemos que estamos moviendo un rectángulo por pantalla. El rectángulo irá cambiando de posición, y en cada momento lo dibujaremos en la posición en la que se encuentre. Pero si no borramos el contenido de la pantalla en el instante anterior, el rectángulo aparecerá en todos los lugares donde ha estado en instantes anteriores produciendo este efecto indeseable de dejar rastro. Por ello será necesario borrar el contenido anterior de la pantalla.

Sin embargo, el borrar la pantalla y volver a dibujar en cada *frame* muchas veces puede producir un efecto de parpadeo de los gráficos. Si además en el proceso de dibujado se deben dibujar varios componentes, y vamos dibujando uno detrás de otro directamente en la pantalla, en cada *frame* veremos como se va construyendo poco a poco la escena, cosa que también es un efecto poco deseable.

Para evitar que esto ocurra y conseguir unas animaciones limpias utilizaremos la técnica del *doble buffer*.

Copyright © 2010 Depto. CCIA All rights reserved.

1.3.2. Técnica del doble buffer

La técnica del *doble buffer* consiste en dibujar todos los elementos que queremos mostrar en una imagen en memoria, que denominaremos *backbuffer*, y una vez se ha dibujado todo volcarlo a pantalla como una unidad. De esta forma, mientras se va dibujando la imagen, como no se hace directamente en pantalla no veremos efectos de parpadeo al borrar el contenido anterior, ni veremos como se va creando la imagen, en pantalla se volcará la imagen como una unidad cuando esté completa.

Para utilizar esta técnica lo primero que deberemos hacer es crearnos el *backbuffer*. Para implementarlo en Java utilizaremos una imagen (objeto `Image`) con lo que tendremos un *raster* en memoria sobre el que dibujar el contenido que queramos mostrar. Deberemos crear una imagen del mismo tamaño de la pantalla en la que vamos a dibujar.

Crearemos para ello una imagen mutable en blanco, como hemos visto anteriormente, con las dimensiones del *canvas* donde vayamos a volcarla:

```
Image backbuffer = Image.createImage(getWidth(), getHeight());
```

Obtenemos su contexto gráfico para poder dibujar en su *raster* en memoria:

```
Graphics offScreen = backbuffer.getGraphics();
```

Una vez obtenido este contexto gráfico, dibujaremos todo lo que queremos mostrar en él, en lugar de hacerlo en pantalla. Una vez hemos dibujado todo el contenido en este contexto gráfico, deberemos volcar la imagen a pantalla (al contexto gráfico del *canvas*) para que ésta se haga visible:

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

La imagen conviene crearla una única vez, ya que la animación puede redibujar frecuentemente, y si cada vez que lo hacemos creamos un nuevo objeto imagen estaremos malgastando memoria inútilmente. Es buena práctica de programación en Java instanciar nuevos objetos las mínimas veces posibles, intentando reutilizar los que ya tenemos.

Podemos ver como quedaría nuestra clase ahora:

```
public MiCanvas extends Canvas {

  // Backbuffer
  Image backbuffer = null;

  // Ancho y alto del backbuffer
  int width, height;

  // Coordenadas del rectangulo dibujado
  int x, y;

  public void paint(Graphics g) {
    // Solo creamos la imagen la primera vez
    // o si el componente ha cambiado de tamaño
```


Copyright © 2010 Depto. CCIA All rights reserved.

```
    if( backbuffer == null ||
        width != getWidth() ||
        height != getHeight() ) {
      width = getWidth();
      height = getHeight();
      backbuffer = Image.createImage(width, height);
    }

    Graphics offScreen = backbuffer.getGraphics();

    // Vaciamos el área de dibujo

    offScreen.clearRect(0,0,getWidth(), getHeight());

    // Dibujamos el contenido en offScreen
    offScreen.setColor(0x00FF0000);
    offScreen.fillRect(x,y,50,50);

    // Volcamos el back buffer a pantalla
    g.drawImage(backbuffer,0,0,Graphics.TOP|Graphics.LEFT);
  }
}
```

En ese ejemplo se dibuja un rectángulo rojo en la posición *(x,y)* de la pantalla que podrá ser variable, tal como veremos a continuación añadiendo a este ejemplo métodos para realizar la animación.

Algunas implementaciones de MIDP ya realizan internamente el doble *buffer*, por lo que en esos casos no será necesario que lo hagamos nosotros. Es más, convendrá que no lo hagamos para no malgastar innecesariamente el tiempo. Podemos saber si implementa el doble *buffer* o no llamando al método `isDoubleBuffered` del `Canvas`.

Podemos modificar el ejemplo anterior para en caso de realizar el doble *buffer* la implementación de MIDP, no hacerla nosotros:

```
public MiCanvas extends Canvas {
    ...
    public void paint(Graphics gScreen) {

        boolean doblebuffer = isDoubleBuffered();

        // Solo creamos el backbuffer si no hay doble buffer
        if( !doblebuffer ) {
            if ( backbuffer == null ||
                 width != getWidth() ||
                 height != getHeight() )
            {
                width = getWidth();
                height = getHeight();
                backbuffer = Image.createImage(width, height);
            }
        }

        // g sera la pantalla o nuestro backbuffer segun si
        // el doble buffer está ya implementado o no
```


Copyright © 2010 Depto. CCIA All rights reserved.

```
        Graphics g = null;

        if(doblebuffer) {
            g = gScreen;
        } else {
            g = backbuffer.getGraphics();
        }

        // Vaciamos el área de dibujo

        g.clearRect(0,0,getWidth(), getHeight());

        // Dibujamos el contenido en g
        g.setColor(0x00FF0000);
        g.fillRect(x,y,50,50);

        // Volcamos si no hay doble buffer implementado
        if(!doblebuffer) {
            gScreen.drawImage(backbuffer,0,0,
                              Graphics.TOP|Graphics.LEFT);
        }
    }
}
```

1.3.3. Código para la animación

Si queremos hacer una animación tendremos que ir cambiando ciertas propiedades de los objetos de la imagen (por ejemplo su posición) y solicitar que se redibuje tras cada cambio. Esta tarea deberá realizarla un hilo que se ejecute en segundo plano. El bucle para la animación podría ser el siguiente:

```
public class MiCanvas extends Canvas {
        ...
        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) {}
                }
        }
}
```

Con este código de ejemplo veremos una animación en la que el rectángulo que dibujamos partirá de la posición (10,10) y cada 100ms se moverá un *pixel* hacia la derecha, hasta llegar a la coordenada (100,10).

Si queremos que la animación se ponga en marcha nada más mostrarse la pantalla del canvas, podremos hacer que este hilo comience a ejecutarse en el método `showNotify`

Copyright © 2010 Depto. CCIA All rights reserved.

como hemos visto anteriormente.

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

Para implementar estas animaciones podemos utilizar un hilo que duerma un determinado período tras cada iteración, como en el ejemplo anterior, o bien utilizar temporizadores que realicen tareas cada cierto periodo de tiempo. Los temporizadores nos pueden facilitar bastante la tarea de realizar animaciones, ya que simplemente deberemos crear una tarea que actualice los objetos de la escena en cada iteración, y será el temporizador el que se encargue de ejecutar cíclicamente dicha tarea.

1.3.4. Hilo de eventos

Hemos visto que existen una serie de métodos que se invocan cuando se produce algún determinado evento, y nosotros podemos redefinir estos métodos para indicar cómo dar respuesta a estos eventos. Estos métodos que definimos para que sean invocados cuando se produce un evento son denominados *callbacks*. Tenemos los siguientes *callbacks*:

- **`showNotify`** y **`hideNotify`**, para los eventos de aparición y ocultación del canvas.
- **`paint`** para el evento de dibujado.
- **`commandAction`** para el evento de ejecución de un comando.
- **`keyPressed`**, **`keyRepeated`**, **`keyReleased`**, **`pointerPressed`**, **`pointerDragged`** y **`pointerReleased`** para los eventos de teclado y de puntero, que veremos más adelante.

Estos eventos son ejecutados por el sistema de forma secuencial, desde un mismo hilo de eventos. Por lo tanto, estos *callbacks* deberán devolver el control cuanto antes, de forma que bloqueen al hilo de eventos el mínimo tiempo posible.

Si dentro de uno de estos *callbacks* tenemos que realizar una tarea que requiera tiempo, deberemos crear un hilo que realice la tarea en segundo plano, para que el hilo de eventos siga ejecutándose mientras tanto.

En algunas ocasiones puede interesarnos ejecutar alguna tarea de forma secuencial dentro de este hilo de eventos. Por ejemplo esto será útil si queremos ejecutar el código de nuestra animación sin que interfiera con el método `paint`. Podemos hacer esto con el método `callSerially` del objeto `Display`. Deberemos proporcionar un objeto `Runnable` para ejecutar su método `run` en serie dentro del hilo de eventos. La tarea que definamos dentro de este `run` deberá terminar pronto, al igual que ocurre con el código definido en los *callbacks*, para no bloquear el hilo de eventos.

Podemos utilizar `callSerially` para ejecutar el código de la animación de la siguiente forma:

Copyright © 2010 Depto. CCIA All rights reserved.

```
public class MiCanvas extends Canvas implements Runnable {
    ...
    public void anima() {
        // Inicia la animación
        repaint();
        mi_display.callSerially(this);
    }

    public void run() {
        // Actualiza la animación
        ...
        repaint();
        mi_display.callSerially(this);
    }
}
```

La llamada a `callSerially` nos devuelve el control inmediatamente, no espera a que el método `run` sea ejecutado.

1.3.5. Optimización de imágenes

Si tenemos varias imágenes correspondientes a varios *frames* de una animación, podemos optimizar nuestra aplicación guardando todas estas imágenes como una única imagen. Las guardaremos en forma de mosaico dentro de un mismo fichero de tipo imagen, y en cada momento deberemos mostrar por pantalla sólo una de las imágenes dentro de este mosaico.

Imagen con los frames de una animación

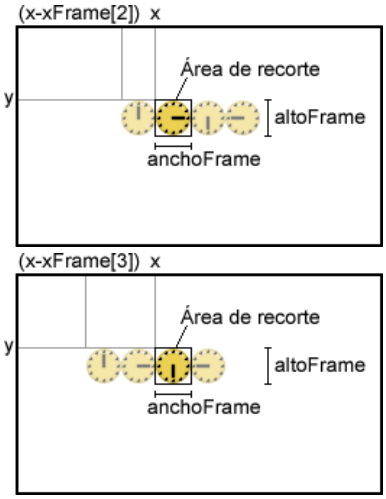
De esta forma estamos reduciendo el número de ficheros que incluimos en el JAR de la aplicación, por lo que por una lado reduciremos el espacio de este fichero, y por otro lado tendremos que abrir sólo un fichero, y no varios.

Para mostrar sólo una de las imágenes del mosaico, lo que podemos hacer es establecer un área de recorte del tamaño de un elemento del mosaico (*frame*) en la posición donde queramos dibujar esta imagen. Una vez hecho esto, ajustaremos las coordenadas donde dibujar la imagen de forma que dentro del área de recorte caiga el elemento del mosaico que queremos mostrar en este momento. De esta forma, sólo será dibujado este elemento, ignorándose el resto.

Podemos ver esto ilustrado en la Figura 18. En ella podemos ver a la izquierda cómo mostrar el segundo frame del reloj, y a la derecha cómo mostrar el tercer frame. Queremos dibujar el reloj en la posición `(x,y)` de la imagen. Cada frame de este reloj tiene un tamaño `anchoFrame x altoFrame`. Por lo tanto, el área de recorte será un rectángulo cuya esquina superior izquierda estará en las coordenadas `(x,y)` y tendra una altura y una anchura de `altoFrame` y `anchoFrame` respectivamente, para de esta manera poder dibujar en esa región de la pantalla cada *frame* de nuestra imagen.

Copyright © 2010 Depto. CCIA All rights reserved.

Para dibujar cada uno de los *frames* deberemos desplazar la imagen de forma que el *frame* que queramos mostrar caiga justo bajo el área de recorte establecida. De esta forma al dibujar la imagen, se volcará a la pantalla sólo el *frame* deseado, y el resto, al estar fuera del área de recorte, no se mostrará. En la figura podemos ver los *frames* que quedan fuera del área de recorte representados con un color más claro, al volcar la imagen estos *frames* no se dibujarán.



A continuación se muestra el código fuente del ejemplo anterior, con el que podremos dibujar cada *frame* de la imagen.

```
// Guardamos el área de recorte anterior
int clipX = g.getClipX();
int clipY = g.getClipY();
int clipW = g.getClipWidth();
int clipH = g.getClipHeight();

// Establecemos nuevo área de recorte
g.clipRect(x, y, anchoFrame, altoFrame);

// Dibujamos la imagen con el desplazamiento adecuado
g.drawImage(imagen,
            x - xFrame[frameActual],
            y - yFrame[frameActual],
            Graphics.TOP | Graphics.LEFT);

// Reestablecemos el área de recorte anterior
g.setClip(clipX, clipY, clipW, clipH);
```

1.4. Eventos de entrada

La clase `Canvas` nos permite acceder a los eventos de entrada del usuario a bajo nivel. De esta forma podremos saber cuando el usuario pulsa o suelta cualquier tecla del dispositivo. Cuando ocurra un evento en el teclado se invocará uno de los siguientes métodos de la clase `Canvas`:

Copyright © 2010 Depto. CCIA All rights reserved.

`keyPressed(int cod)`	Se ha presionado la tecla con código `cod`
`keyRepeated(int cod)`	Se mantiene presionada la tecla con código `cod`
`keyReleased(int cod)`	Se ha soltado la tecla con código `cod`

Estos dispositivos, además de generar eventos cuando presionamos o soltamos una tecla, son capaces de generar eventos de repetición. Estos eventos se producirán cada cierto período de tiempo mientras mantengamos pulsada una tecla.

Al realizar aplicaciones para móviles debemos tener en cuenta que en la mayoría de estos dispositivos no se puede presionar más de una tecla al mismo tiempo. Hasta que no hayamos soltado la tecla que estemos pulsando, no se podrán recibir eventos de pulsación de ninguna otra tecla.

Para dar respuesta a estos eventos del teclado deberemos redefinir estos métodos en nuestra subclase de `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
    }
}
```

1.4.1. Códigos del teclado

Cada tecla del teclado del dispositivo tiene asociado un código identificativo que será el parámetro que se le proporcione a estos métodos al presionarse o soltarse. Tenemos una serie de constantes en la clase `Canvas` que representan los códigos de las teclas estándar:

`Canvas.KEY_NUM0`	0
`Canvas.KEY_NUM1`	1
`Canvas.KEY_NUM2`	2
`Canvas.KEY_NUM3`	3
`Canvas.KEY_NUM4`	4
`Canvas.KEY_NUM5`	5
`Canvas.KEY_NUM6`	6
`Canvas.KEY_NUM7`	7

Copyright © 2010 Depto. CCIA All rights reserved.

`Canvas.KEY_NUM8`	8
`Canvas.KEY_NUM9`	9
`Canvas.KEY_POUND`	#
`Canvas.KEY_STAR`	*

Los teclados, además de estas teclas estándar, normalmente tendrán otras teclas, cada una con su propio código numérico. Es recomendable utilizar únicamente estas teclas definidas como constantes para asegurar la portabilidad de la aplicación, ya que si utilizamos cualquier otro código de tecla no podremos asegurar que esté disponible en todos los modelos de teléfonos.

Los códigos de tecla corresponden al código Unicode del carácter correspondiente a dicha tecla. Si la tecla no corresponde a ningún carácter Unicode entonces su código será negativo. De esta forma podremos obtener fácilmente el carácter correspondiente a cada tecla. Sin embargo, esto no será suficiente para realizar entrada de texto, ya que hay caracteres que corresponden a múltiples pulsaciones de una misma tecla, y a bajo nivel sólo tenemos constancia de que una misma tecla se ha pulsado varias veces, pero no sabemos a qué carácter corresponde ese número de pulsaciones. Si necesitamos que el usuario escriba texto, lo más sencillo será utilizar uno de los componentes de alto nivel.

Podemos obtener el nombre de la tecla correspondiente a un código dado con el método `getKeyName` de la clase `Canvas`.

1.4.2. Acciones de juegos

Tenemos también definidas lo que se conoce como acciones de juegos (*game actions*) con las que representaremos las teclas que se utilizan normalmente para controlar los juegos, a modo de *joystick*. Las acciones de juegos principales son:

`Canvas.LEFT`	Movimiento a la izquierda
`Canvas.RIGHT`	Movimiento a la derecha
`Canvas.UP`	Movimiento hacia arriba
`Canvas.DOWN`	Movimiento hacia abajo
`Canvas.FIRE`	Fuego

Una misma acción puede estar asociada a varias teclas del teléfono, de forma que el usuario pueda elegir la que le resulte más cómoda. Las teclas asociadas a cada acción de juego serán dependientes de la implementación, cada modelo de teléfono puede asociar a las teclas las acciones de juego que considere más apropiadas según la distribución del teclado, para que el manejo sea cómodo. Por lo tanto, el utilizar estas acciones hará la aplicación más portable, ya que no tendremos que adaptar los controles del juego para cada modelo de móvil.

Copyright © 2010 Depto. CCIA All rights reserved.

Para conocer la acción de juego asociada a un código de tecla dado utilizaremos el siguiente método:

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

De esta forma podremos realizar de una forma sencilla y portable aplicaciones que deban controlarse utilizando este tipo de acciones.

Podemos hacer la transformación inversa con:

```
int codigo = getKeyCode(accion);
```

Hemos de resaltar que una acción de código puede estar asociada a más de una tecla, pero con este método sólo podremos obtener la tecla principal que realiza dicha acción.

1.4.3. Punteros

Algunos dispositivos tienen punteros como dispositivos de entrada. Esto es común en los PDAs, pero no en los teléfonos móviles. Los *callbacks* que deberemos redefinir para dar respuesta a los eventos del puntero son los siguientes:

`pointerPressed(int x, int y)`	Se ha pinchado con el puntero en *(x,y)*
`pointerDragged(int x, int y)`	Se ha arrastrado el puntero a *(x,y)*
`pointerReleased(int x, int y)`	Se ha soltado el puntero en *(x,y)*

En todos estos métodos se proporcionarán las coordenadas *(x,y)* donde se ha producido el evento del puntero.

1.5. APIs propietarias

Existen APIs propietarias de diferentes vendedores, que añaden funcionalidades no soportadas por la especificación de MIDP. Los desarrolladores de estas APIs propietarias no deben incluir en ellas nada que pueda hacerse con MIDP. Estas APIs deben ser únicamente para permitir acceder a funcionalidades que MIDP no ofrece.

Es recomendable no utilizar estas APIs propietarias siempre que sea posible, para hacer aplicaciones que se ajusten al estándar de MIDP. Lo que podemos hacer es desarrollar aplicaciones que cumplan con el estándar MIDP, y en el caso que detecten que hay disponible una determinada API propietaria la utilicen para obtener alguna mejora. A continuación veremos como detectar en tiempo de ejecución si tenemos disponible una determinada API.

Vamos a ver la API Nokia UI, disponible en gran parte de los modelos de teléfonos Nokia, que incorpora nuevas funcionalidades para la programación de la interfaz de usuario no disponibles en MIDP 1.0. Esta API está contenida en el paquete `com.nokia.mid`.

Copyright © 2010 Depto. CCIA All rights reserved.

1.5.1. Gráficos

En cuanto a los gráficos, tenemos disponibles una serie de mejoras respecto a MIDP.

Añade soporte para crear un *canvas* a pantalla completa. Para crear este *canvas* utilizaremos la clase `FullCanvas` de la misma forma que utilizábamos `Canvas`.

Define una extensión de la clase `Graphics`, en la clase `DirectGraphics`. Para obtener este contexto gráfico extendido utilizaremos el siguiente método:

```
DirectGraphics dg = DirectUtils.getDirectGraphics(g);
```

Siendo `g` el objeto de contexto gráfico `Graphics` en el que queremos dibujar. Este nuevo contexto gráfico añade:

- Soporte para nuevos tipos de primitivas geométricas (triángulos y polígonos).
- Soporte para transparencia, incorporando un canal *alpha* al color. Ahora tenemos colores de 32 bits, cuya forma empaquetada se codifica como `0xAARRGGBB`.
- Acceso directo a los *pixels* del *raster* de pantalla. Podremos dibujar *pixels* en pantalla proporcionando directamente el *array* de *pixels* a dibujar, o bien obtener los *pixels* de la pantalla en forma de un *array* de *pixels*. Cada *pixel* de este *array* será un valor `int`, `short` o `byte` que codificará el color de dicho *pixel*.
- Permite transformar las imágenes a dibujar. Podremos hacer rotaciones de 90, 180 o 270 grados y transformaciones de espejo con las imágenes al mostrarlas.

1.5.2. Sonido

Una limitación de MIDP 1.0 es que no soporta sonido. Por ello para incluir sonido en las aplicaciones de dispositivos que sólo soporten MIDP 1.0 como API estándar deberemos recurrir a APIs propietarias para tener estas funcionalidades. La API Nokia UI nos permitirá solucionar esta carencia.

Nos permitirá reproducir sonidos como tonos o ficheros de onda (WAV). Los tipos de formatos soportados serán dependientes de cada dispositivo.

1.5.3. Control del dispositivo

Además de las características anteriores, esta API nos permitirá utilizar funciones propias de los dispositivos. En la clase `DeviceControl` tendremos métodos para controlar la vibración del móvil y el parpadeo de las luces de la pantalla.

1.5.4. Detección de la API propietaria

Si utilizamos una API propietaria reduciremos la portabilidad de la aplicación. Por ejemplo, si usamos la API Nokia UI la aplicación sólo funcionará en algunos dispositivos de Nokia. Hay una forma de utilizar estas APIs propietarias sin afectar a la portabilidad

Copyright © 2010 Depto. CCIA All rights reserved.

de la aplicación. Podemos detectar en tiempo de ejecución si la API propietaria está disponible de la siguiente forma:

```
boolean hayNokiaUI = false;

try {
    Class.forName("com.nokia.mid.sound.Sound");
    hayNokiaUI = true;
} catch(ClassNotFoundException e) {}
```

De esta forma, si la API propietaria está disponible podremos utilizarla para incorporar más funcionalidades a la aplicación. En caso contrario, no deberemos ejecutar nunca ninguna instrucción que acceda a esta API propietaria.

2. Gráficos 3D

Podemos crear aplicaciones que muestren gráficos en 3D utilizando la API opcional Mobile 3D Graphics for J2ME (JSR-184).

Esta API nos permitirá añadir contenido 3D a las aplicaciones de dos modos distintos:

- **Modo inmediato**: Este modo nos servirá para crear los gráficos 3D a bajo nivel, creando directamente los polígonos que va a tener nuestro mundo 3D. Este modo nos permitirá tener un control absoluto sobre los polígonos que se dibujan, pero el tener que definir manualmente estos polígonos hace que este modo no sea adecuado cuando tengamos que mostrar objetos complejos (por ejemplo personajes de un juego). Este modo resultará útil por ejemplo para generar gráficos 3D para representar datos.
- **Modo *retained***: En este modo contaremos con una serie de objetos ya creados que podremos añadir a la escena. Estos objetos que utilizamos los podremos cargar de

Copyright © 2010 Depto. CCIA All rights reserved.

ficheros en los que tendremos almacenados los distintos objetos 3D que vayamos a mostrar. La escena se representará como un grafo, del que colgarán todos los objetos que queramos mostrar en ella. Este modo será útil para aplicaciones como juegos, en las que tenemos que mostrar objetos complejos, como por ejemplo los personajes del juego.

Más adelante estudiaremos con más detalle cómo mostrar gráficos 3D utilizando cada uno de estos modos.

2.1. Renderización 3D

Para renderizar gráficos 3D en el móvil utilizaremos un objeto de tipo `Graphics3D`. Para obtener este objeto utilizaremos el siguiente método estático:

```
Graphics3D g3d;
...
g3d = Graphics3D.getInstance();
```

Podremos declarar este objeto como campo global de la clase y obtenerlo una única vez durante la inicialización de la misma.

Para poder utilizar este objeto, deberemos decirle en qué contexto gráfico queremos que vuelque el contenido 3D que genere. Normalmente estableceremos como objetivo donde volcar el contexto gráfico asociado a la pantalla del móvil.

Esto lo haremos en el método `paint`, como se muestra a continuación:

```
protected void paint(Graphics g) {
  try {
    g3d.bindTarget(g);
    ...
    g3d.render(escena);
  } finally {
    g3d.releaseTarget();
  }
}
```

Con `bindTarget` establecemos en qué contexto gráfico vamos a volver los gráficos 3D. Una vez hecho esto podremos renderizar la escena que hayamos definido utilizando el método `render` del objeto `Graphics3D`. Más adelante veremos cómo definir esta escena utilizando tanto el modo inmediato como el modo *retained*. Por último, con `releaseTarget` liberamos el contexto gráfico que estabamos utilizando para volcar los gráficos.

2.2. Transformaciones geométricas

Cuando queramos establecer la posición de un objeto 3D en el mundo, deberemos transformar sus coordenadas para que éstas estén en la posición deseada. Para representar estas transformaciones geométricas en el espacio tenemos la clase `Transform`.

Copyright © 2010 Depto. CCIA All rights reserved.

Utilizaremos objetos `Transform` para indicar la posición y el tamaño que van a tener los objetos en el mundo. En la transformación podemos establecer traslaciones del objeto, rotaciones, y cambios de escala.

Normalmente asociaremos a cada objeto que tengamos en el mundo 3D una transformación, donde se indicará la posición de dicho objeto. Además, esto nos servirá para hacer animaciones, ya que simplemente cambiando los valores de la transformación en el tiempo podremos hacer que el objeto cambie de posición o tamaño.

Utilizando el constructor de `Transform` podemos crear una transformación identidad:

```
Trasform trans = new Transform();
```

Con esta transformación, el objeto al que se aplique permanecerá en sus coordenadas iniciales. Podremos modificar esta transformación para cambiar la posición o el tamaño del objeto mediante:

- **Traslación**: Especificamos la traslación en (x,y,z) que queremos hacer con el objeto.

```
trans.postTranslate(tx, ty, tz);
```

- **Rotación**: Especificamos un eje de rotación del objeto, dado con el vector (x,y,z) que representa el eje, un un ángulo de rotación alrededor de dicho eje dado en grados.

```
trans.postRotate(angulo, vx, vy, vz);
```

- **Escala**: Especificamos el factor de escala a aplicar en cada una de las tres coordenadas. Con factor 1.0 mantendrá su tamaño actual, y reduciendo o aumentando este valor conseguiremos reducir o agrandar el objeto en dicha coordenada.

```
trans.postScale(sx, sy, sz);
```

Podemos hacer que la transformación vuelva a ser la identidad para volver a la posición original del objeto:

```
trans.setIdentity();
```

2.3. Cámara

Para poder renderizar una escena es imprescindible establecer previamente el punto de vista desde el que la queremos visualizar. Para hacer esto tendremos que definir una cámara, encapsulada en el objeto `Camera`.

Crearemos la cámara utilizando el constructor vacío de esta clase:

```
Camera cam;
...
cam = new Camera();
```

Con esto tendremos la cámara, que siempre estará apuntando hacia la coordenada Z negativa (0, 0, -1). Si queremos mover o rotar la cámara, deberemos hacerlo mediante un

Copyright © 2010 Depto. CCIA All rights reserved.

objeto `Transform` que aplicaremos a la cámara:

```
Transform tCam;
...
tCam = new Transform();
tCam.postTranslate(0.0f, 0.0f, 2.0f);
```

De la cámara deberemos especificar el tipo de proyección que queremos utilizar y una serie de parámetros. Podemos establecer dos tipos de proyecciones:

- **Paralela**: En este tipo de proyecciones, el centro de proyección está en el infinito. Las líneas que son paralelas en el espacio, se mantienen paralelas en la imagen proyectada. Estableceremos este tipo de proyección con:

```
cam.setParallel(campoDeVision, relacionAspecto,
                planoFrontal, planoPosterior);
```

- **Perspectiva**: El centro de proyección es un punto finito. En este caso, las líneas que son paralelas en el espacio, en la imagen proyectada convergen en algún punto. Estableceremos este tipo de proyección con:

```
cam.setPerspective(campoDeVision, relacionAspecto,
                planoFrontal, planoPosterior);
```

En el parámetro `campoDeVision` indicaremos el campo visual de la cámara en grados. Este valor normalmente será 45º o 60º. Valores superiores a 60º distorsionan la imagen. En `relacionAspecto` indicaremos la proporción entre el ancho y el alto de la pantalla donde se van a mostrar los gráficos. Además deberemos establecer dos planos de recorte: uno cercano (`planoFrontal`) y uno lejano (`planoPosterior`), de forma que sólo se verán los objetos que estén entre estos dos planos. Todo aquello más cercano al plano frontal, y más lejano al plano posterior será recortado.

Por ejemplo, podemos utilizar una proyección perspectiva como la siguiente:

```
cam.setPerspective(60.0f, (float)getWidth()/(float)getHeight(),
                   1.0f, 1000.0f);
```

Una vez definida la cámara, deberemos establecerla en el objeto `Graphics3D` de la siguiente forma:

```
 g3d.setCamera(cam, tCam);
```

Vemos que al establecer la cámara, además de el objeto `Camera`, deberemos especificar la transformación que se va a aplicar sobra la misma. Si queremos hacer movimientos de cámara no tendremos más que modificar la transformación que utilizamos para posicionarla en la escena.

Si vamos a dejar la cámara fija, podemos establecer la cámara en el objeto `Graphics3D` una única vez durante la inicialización. Sin embargo, si queremos mover la cámara, deberemos establecerla dentro del método `paint`, para que cada vez que se redibuje se sitúe la cámara en su posición actual.

Copyright © 2010 Depto. CCIA All rights reserved.

2.4. Luces

Deberemos también definir la iluminación del mundo. Para crear luces utilizaremos el objeto `Light`. Podemos crear una luz utilizando su constructor vacío:

```
Light luz;
...
luz = new Light();
```

La luz tendrá un color y una intensidad, que podremos establecer utilizando los siguientes métodos:

```
luz.setColor(0x0ffffff);
luz.setIntensity(1.0f);
```

Como color podremos especificar cualquier color RGB, y como intensidad deberemos introducir un valor dentro del rango [0.0, 1.0]. Estos atributos son comunes para cualquier tipo de luces.

Además, podemos utilizar diferentes tipos de luces con distintas propiedades. Estos tipos son:

- **Ambiente** (`Light.AMBIENT`): Esta luz afectará a todos los objetos por igual en todas las direcciones. Es la luz mínima del ambiente, que incide en todas las partes de los objetos. Simula la reflexión de la luz sobre los objetos que produce dicha luz que está "en todas partes".
- **Direccional** (`Light.DIRECTIONAL`): Esta luz incide en todos los objetos en una misma dirección. Con está luz podemos modelar una fuente de luz muy lejana (en el infinito). Por ejemplo, nos servirá para modelar la iluminación el sol, que incide en todos los objetos en la misma dirección en cada momento. La luz estará apuntando en la dirección del eje de las Z negativo (0, 0, -1).
- **Omnidireccional** (`Light.OMNI`): Se trata de una fuente de luz que ilumina en todas las direcciones. Por ejemplo con esta luz podemos modela una bombilla, que es un punto que emana luz en todas las direcciones.
- **Foco** (`Light.SPOT`): Este tipo produce un cono de luz. El foco ilumina en una determinada dirección, produciendo un cono de luz. El foco estará orientado en la dirección del eje de las Z negativo (0, 0, -1).

Para establecer el tipo de luz que queremos utilizar tenemos el método `setMode`:

```
luz.setMode(Light.DIRECTIONAL);
```

Dentro de los tipos de luces anteriores, podemos distinguir dos grupos según la posición de la fuente. Tenemos luces sin posición (ambiente y direccional) y luces con posición (omnidireccional y foco). En el caso de las luces con posición, podemos utilizar el método `setAttenuation` para dar una atenuación a la luz en función a la distancia de la fuente. Con esto haremos que los objetos que estén más lejos de la fuente de luz estén menos iluminados que los que estén más cerca. Esto no se puede hacer en el caso de las

Copyright © 2010 Depto. CCIA All rights reserved.

fuentes de luz sin posición, ya que en ese caso la distancia de cualquier objeto a la fuente es infinito o no existe.

Otra clasificación que podemos hacer es entre fuentes de luz sin dirección (ambiente y omnidireccional) y con dirección (direccional y foco). Hemos visto que las fuentes con dirección siempre están apuntando en la dirección del eje de las Z negativo. Si queremos modificar este dirección podemos aplicar una rotación a la luz mediante un objeto `Transform`. De la misma forma, en las fuentes con posición podremos aplicar una traslación para modificar la posición de la fuente de luz con este objeto.

```
Transform tLuz;
...
tLuz = new Transform();
```

Para añadir las luces al mundo utilizaremos el método `addLight` sobre el objeto de contexto gráfico 3D, en el que especificaremos la luz y la transformación que se va a realizar:

```
g3d.addLight(luz, tLuz);
```

Si como transformación especificamos `null`, se aplicará la identidad. Por ejemplo en el caso de una luz ambiente no tiene sentido aplicar ninguna transformación a la luz, ya que no tiene ni posición ni dirección.

```
g3d.addLight(luzAmbiente, null);
```

Al igual que ocurría en el caso de la cámara, si las luces van a permanecer fijas sólo hace falta que las añadamos una vez durante la inicialización. Sin embargo, si se van a mover, tendremos que establecer la posición de la luz cada vez que se renderiza para que se apliquen los cambios. En este caso si añadimos las luces dentro del método `paint` deberemos llevar cuidado, ya que el método `addLight` añade luces sobre las que ya tiene definidas el mundo. Para evitar que las luces se vayan acumulando, antes de volver a añadirlas tendremos que eliminar las que teníamos en la iteración anterior con `resetLights`. Una vez hecho esto podremos añadir las demás luces:

```
g3d.resetLights();
g3d.addLight(luz, tLuz);
g3d.addLight(luzAmbiente, null);
```

2.5. Fondo

También deberemos establecer un fondo para el visor antes de renderizar en él los gráficos 3D. Como fondo podemos poner un color sólido o una imagen. El fondo lo representaremos mediante un objeto de la clase `Background`:

```
Background fondo;
...
fondo = new Background();
```

Con el constructor vacío construiremos el fondo por defecto que consiste en un fondo

Copyright © 2010 Depto. CCIA All rights reserved.

negro. Podemos cambiar el color de fondo con el método `setColor`, o establecer una imagen de fondo con `setImage`.

Cada vez que vayamos a renderizar los gráficos en `paint` es importante que vaciemos todo el área de dibujo con el color (o la imagen) de fondo. Para ello utilizaremos el método `clear`:

```
g3d.clear(fondo);
```

Si no hiciésemos esto, es posible que tampoco se visualizasen los gráficos 3D que se rendericen posteriormente. Si no queremos crear ningún fondo y utilizar el fondo negro por defecto, podremos llamar a este método proporcionando `null` como parámetro:

```
g3d.clear(null);
```

Vamos a recapitular todo lo visto hasta ahora, y a ver cómo quedará el método `paint` añadiendo luces, cámara y fondo:

```
protected void paint(Graphics g) {
  try {
    g3d.bindTarget(g);

    g3d.setCamera(cam, tCam);

    g3d.resetLights();
    g3d.addLight(luz, tLuz);
    g3d.addLight(luzAmbiente, null);

    g3d.clear(fondo);

    g3d.render(vb, tsa, ap, tCubo);
  } finally {
    g3d.releaseTarget();
  }
}
```

2.6. Modo inmediato

Para crear gráficos utilizando este modo deberemos definir la lista de vértices (x, y, z) de nuestro gráfico, y una lista de índices en la que definimos las caras (polígonos) que se van a mostrar. Cada cara se construirá a partir de un conjunto de vértices. Los índices con los que se define cada cara en la lista de caras, harán referencia a los índices de los vértices que la componen en la lista de vértices.

A continuación se muestra un ejemplo de cómo crear un cubo utilizando modo inmediato:

```
// Lista de vertices
short [] vertexValues = {
   0, 0, 0, // 0
   0, 0, 1, // 1
   0, 1, 0, // 2
   0, 1, 1, // 3
   1, 0, 0, // 4
```


Copyright © 2010 Depto. CCIA All rights reserved.

```
   1, 0, 1, // 5
   1, 1, 0, // 6
   1, 1, 1  // 7
};

// Lista de caras
int [] faceIndices = {
   0, 1, 2, 3,
   7, 5, 6, 4,
   4, 5, 0, 1,
   3, 7, 2, 6,
   0, 2, 4, 6,
   1, 5, 3, 7
};
```

En la lista de vértices podemos ver que tenemos definidas las coordenadas de las 8 esquinas del cubo. En la lista de caras se definen las 6 caras del cubo. Cada cara se define a partir de los 4 vértices que la forman. De estos vértices especificaremos su índice en la lista de vértices.

En M3G representaremos las caras y vértices de nuestros objetos mediante los objetos `VertexBuffer` e `IndexBuffer` respectivamente. Estos objetos los crearemos a partir de los arrays de vértices y caras definidos anteriormente.

Para crear el objeto `VertexBuffer` es necesario que le pasemos los arrays de datos necesarios en forma de objetos `VertexArray`. Para construir el objeto `VertexArray` como parámetros deberemos proporcionar:

```
VertexArray va = new VertexArray(numVertices,
                                 numComponentes, bytesComponente);
```

Debemos decir en `numVertices` el número de vértices que hemos definido, en `numComponentes` el número de componentes de cada vértice (al tratarse de coordenadas 3D en este caso será siempre 3: x, y ,z) y el número de bytes por componente. Es decir, si se trata de una array de tipo `byte` tendrá un byte por componente, mientras que si es de tipo `short` tendrá 2 bytes por componente.

Una vez creado el array, deberemos llenarlo de datos. Para ello utilizaremos el siguiente método:

```
va.set(verticeInicial, numVertices, listaVertices);
```

Proporcionaremos un array de vértices (`listaVertices`) como el visto en el ejemplo anterior para insertar dichos vértices en el objeto `VertexArray`. Además diremos, dentro de este array de vértices, cual es el vértice inicial a partir del cual queremos empezar a insertar (`verticeInicial`) y el número de vértices, a partir de dicho vértice, que vamos a insertar (`numVertices`).

En el caso del array de vértices de nuestro ejemplo, utilizaremos los siguientes datos:

```
VertexArray va = new VertexArray(8, 3, 2);
va.set(0, 8, vertexValues);
```


Copyright © 2010 Depto. CCIA All rights reserved.

Con esto tendremos construido un objeto `VertexArray` en el que tendremos el array de posiciones de los vértices.

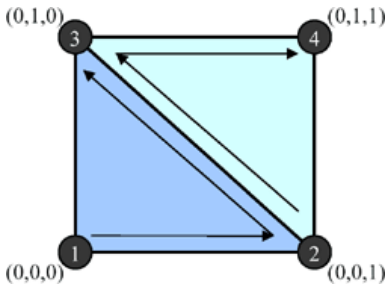
Ahora podremos crear un objeto `VertexBuffer` y utilizar el objeto anterior para establecer su lista de coordenadas de los vértices.

```
VertexBuffer vb;
...
vb = new VertexBuffer();
vb.setPositions(va, 1.0f, null);
```

Al método `setPositions` le podemos proporcionar como segundo y tercer parámetro un factor de escala y una traslación respectivamente, para transformar las coordenadas de los puntos que hemos proporcionado. Si no queremos aplicar ninguna transformación geométrica a estos puntos, simplemente proporcionaremos como escala `1.0f` y como traslación `null`.

Una vez creado este objeto, necesitaremos el objeto de índices en el que se definen las caras. Este objeto será de tipo `IndexBuffer`, sin embargo esta clase es abstracta, por lo que deberemos utilizar una de sus subclases. La única subclase disponible por el momento es `TriangleStripArray` que representa las caras a partir de tiras (strips) de triángulos.

Para definir cada tira de triángulos podremos especificar 3 o más vértices. Si indicamos más de 3 vértices se irán tomando como triángulos cada trio adyacente de vértices. Por ejemplo, si utilizamos la tira `{ 1,2,3,4 }`, se crearán los triángulos `{ 1,2,3 }` y `{ 2,3,4 }`.



Tira de triángulos para una cara del cubo

El orden en el que indiquemos los vértices de cada polígono será importante. Según el sentido de giro en el que se defina su cara frontal y cara trasera será una u otra. La cara frontal será aquella para la que el sentido de giro de los vértices vaya en el sentido de las agujas del reloj.

Cuando creemos este objeto deberemos proporcionar, a parte de la lista de índices, un array con el tamaño de cada strip.

Por ejemplo, en el caso del ejemplo del cubo, como cada strip se compone de 4 índices, el array de tamaños tendrá el valor 4 para cada uno de los 6 strips que teníamos definidos:

Copyright © 2010 Depto. CCIA All rights reserved.

```
int [] stripSizes = {
   4, 4, 4, 4, 4, 4
};
```

Con esta información ya podremos crear el objeto `TriangleStripArray`:

```
TriangleStripArray tsa;
...
tsa = new TriangleStripArray(faceIndices, stripSizes);
```

Además de la información de vértices y caras, deberemos darle una apariencia al objeto creado. Para ello podremos utilizar el objeto `Appearance`. Podemos crear un objeto con la apariencia por defecto:

```
Appaerance ap;
...
ap = new Appaerance();
```

Sobre este objeto podremos cambiar el material o la textura de nuestro objeto 3D.

Para renderizar el gráfico 3D que hemos construido, utilizaremos la siguiente variante del método `render`:

```
g3d.render(VertexBuffer vertices, IndexBuffer indices,
           Appaerance apariencia, Transform transformacion);
```

En ella especificaremos los distintos componentes de nuestra figura 3D, conocida como *submesh*: lista de vértices (`VertexBuffer`), lista de caras (`IndexBuffer`) y apariencia (`Appaerance`). Además especificaremos también un objeto `Transform` con la transformación geométrica que se le aplicará a nuestro objeto al añadirlo a la escena 3D. Podemos crear una transformación identidad utilizando el constructor vacío de esta clase:

```
Transform tCubo;
...
tCubo = new Transform();
```

Aplicando esta transformación se dibujará el cubo en sus coordenadas originales. Si posteriormente queremos moverlo en el espacio (trasladarlo o rotarlo) o cambiar su tamaño, podremos hacerlo simplemente modificando esta transformación.

Para renderizar nuestro ejemplo del cubo utilizaremos el método `render` como se muestra a continuación:

```
g3d.render(vb, tsa, ap, tCubo);
```

Con lo que hemos visto hasta ahora, si mostramos el cubo que hemos creado aparecerá con el siguiente aspecto:

Copyright © 2010 Depto. CCIA All rights reserved.

Esta es la apariencia (`Appaerance`) definida por defecto. En ella no hay ningún material ni textura definidos para el objeto. Al no haber ningún material establecido (el material es `null`), la iluminación no afecta al objeto.

Vamos ahora a establecer un material para el objeto. Creamos un material (objeto `Material`):

```
Material mat = new Material();
```

Esto nos crea un material blanco por defecto. Podremos modificar en este objeto el color de ambiente, el color difuso, el color especular, y el color emitido del material.

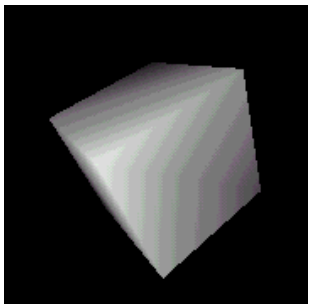
Cuando hayamos establecido un material para el objeto, la iluminación afectará sobre él. En este caso será necesario definir las normales de los vértices del objeto. Si estas normales no estuviesen definidas, se producirá un error al renderizar.

Para asignar las normales a los vértices del objeto utilizaremos un objeto `VertexArray` que añadiremos al `VertexBuffer` de nuestro objeto utilizando el método `setNormals`.

```
byte [] normalValues = {
   -1, -1, -1,
   -1, -1, 1,
   -1, 1, -1,
   -1, 1, 1,
   1, -1, -1,
   1, -1, 1,
   1, 1, -1,
   1, 1, 1
};

...

VertexArray na = new VertexArray(8, 3, 1);
na.set(0, 8, normalValues);

...

vb.setNormals(na);
```

Con esto, al visualizar nuestro cubo con una luz direccional apuntando desde nuestro punto de vista hacia el cubo, se mostrará con el siguiente aspecto:

Copyright © 2010 Depto. CCIA All rights reserved.



Vamos a ver ahora como añadir textura al objeto. Para ello lo primero que debemos hacer es añadir coordenadas de textura a cada vértice. Para añadir estas coordenadas utilizaremos también un objeto `VertexArray` que añadiremos al `VertexBuffer` mediante el método `setTexCoords`.

```
short [] tex = {
   0,0, 0,0, 0,1, 0,1,
   1,0, 1,0, 1,1, 1,1
};

VertexArray ta = new VertexArray(8, 2, 2);
ta.set(0, 8, tex);

vb.setTexCoords(0, ta, 1.0f, null);
```

El primer parámetro que toma `setTexCoords` es el índice de la unidad de textura a la que se van a asignar esas coordenadas. Como segundo parámetro se proporcionan las coordenadas en forma de `VertexArray`. El tercer y cuarto parámetro nos permiten realizar escalados y traslaciones de estas coordenadas respectivamente.

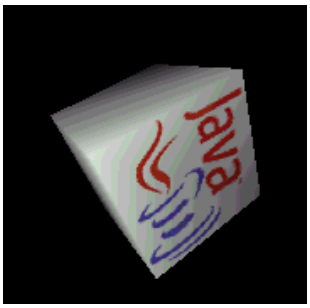
Una vez hecho esto podemos crear la textura a partir de una imagen y añadirla a la apariencia:

```
try {
  Image img = Image.createImage("/texture.png");
  Image2D img2d = new Image2D(Image2D.RGB, img);

  Texture2D tex2d = new Texture2D(img2d);
  ap.setTexture(0, tex2d);
} catch (IOException e) { }
```

La textura se establece en el objeto `Appaerance` mediante el método `setTexture`. A este método le proporcionamos como parámetros el índice de la unidad de textura, y la textura que vamos a establecer en dicha unidad de textura. De esta forma se podrán definir varias unidades de textura, teniendo cada una de ellas unas coordenadas y una imagen.

Con esto el aspecto del cubo será el siguiente:

Copyright © 2010 Depto. CCIA All rights reserved.



Podemos añadir animación al objeto modificando su transformación a lo largo del tiempo. Para ello podemos crear un hilo como el siguiente:

```
public void run() {
  while(true) {
    tCubo.postTranslate(0.5f, 0.5f, 0.5f);
    tCubo.postRotate(1.0f, 1.0f, 1.0f, 1.0f);
    tCubo.postTranslate(-0.5f, -0.5f, -0.5f);
    repaint();
    try {
      Thread.sleep(25);
    } catch (InterruptedException e) { }
  }
}
```

A continuación mostramos el código completo del canvas de este ejemplo:

```
package es.ua.j2ee.m3d;

import java.io.IOException;

import javax.microedition.lcdui.*;
import javax.microedition.m3g.*;

public class Visor3DInmediate extends Canvas implements Runnable {

  MIDlet3D owner;

  Graphics3D g3d;
  Transform tCam;
  Transform tLuz;
  Transform tCubo;
  VertexBuffer vb;
  IndexBuffer tsa;
  Appearance ap;
  Camera cam;
  Light luz;
  Light luzAmbiente;
  Background fondo;

  short [] vertexValues = {
    0, 0, 0, // 0
    0, 0, 1, // 1
    0, 1, 0, // 2
    0, 1, 1, // 3
    1, 0, 0, // 4
```


Copyright © 2010 Depto. CCIA All rights reserved.

```
    1, 0, 1, // 5
    1, 1, 0, // 6
    1, 1, 1 // 7
  };

  byte [] normalValues = {
    -1, -1, -1,
    -1, -1, 1,
    -1, 1, -1,
    -1, 1, 1,
    1, -1, -1,
    1, -1, 1,
    1, 1, -1,
    1, 1, 1
  };

  int [] faceIndices = {
    0, 1, 2, 3,
    7, 5, 6, 4,
    4, 5, 0, 1,
    3, 7, 2, 6,
    0, 2, 4, 6,
    1, 5, 3, 7
  };

  int [] stripSizes = {
    4, 4, 4, 4, 4, 4
  };

  short [] tex = {
    0,0, 0,0, 0,1, 0,1,
    1,0, 1,0, 1,1, 1,1
  };

  public Visor3DInmediate(MIDlet3D owner) {
    this.owner = owner;
    init3D();
  }

  public void init3D() {

    g3d = Graphics3D.getInstance();
    tCubo = new Transform();
    tCam = new Transform();
    tLuz = new Transform();

    cam = new Camera();
    cam.setPerspective(60.0f, (float)getWidth()/(float)getHeight(),
                       1.0f, 10.0f);
    tCam.postTranslate(0.0f, 0.0f, 2.0f);

    luz = new Light();
    luz.setColor(0x0ffffff);
    luz.setIntensity(1.0f);
    luz.setMode(Light.DIRECTIONAL);
    tLuz.postTranslate(0.0f, 0.0f, 5.0f);

    luzAmbiente = new Light();
    luzAmbiente.setColor(0x0ffffff);
```


Copyright © 2010 Depto. CCIA All rights reserved.

```
    luzAmbiente.setIntensity(0.5f);
    luzAmbiente.setMode(Light.AMBIENT);

    fondo = new Background();

    VertexArray va = new VertexArray(8, 3, 2);
    va.set(0, 8, vertexValues);

    VertexArray na = new VertexArray(8, 3, 1);
    na.set(0, 8, normalValues);

    VertexArray ta = new VertexArray(8, 2, 2);
    ta.set(0, 8, tex);

    vb = new VertexBuffer();
    vb.setPositions(va, 1.0f, null);
    vb.setNormals(na);
    vb.setTexCoords(0, ta, 1.0f, null);

    tsa = new TriangleStripArray(faceIndices, stripSizes);

    ap = new Appearance();
    Material mat = new Material();
    ap.setMaterial(mat);

    try {
      Image img = Image.createImage("/texture.png");
      Image2D img2d = new Image2D(Image2D.RGB, img);

      Texture2D tex2d = new Texture2D(img2d);
      ap.setTexture(0, tex2d);
    } catch (IOException e) { }

    tCubo.postTranslate(-0.5f, -0.5f, -0.5f);
  }

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

  public void run() {
    while(true) {
      tCubo.postTranslate(0.5f, 0.5f, 0.5f);
      tCubo.postRotate(1.0f, 1.0f, 1.0f, 1.0f);
      tCubo.postTranslate(-0.5f, -0.5f, -0.5f);
      repaint();
      try {
        Thread.sleep(25);
      } catch (InterruptedException e) { }
    }
  }

  protected void paint(Graphics g) {
    try {
      g3d.bindTarget(g);

      g3d.setCamera(cam, tCam);
      g3d.resetLights();
```


Copyright © 2010 Depto. CCIA All rights reserved.

```
      g3d.addLight(luz, tLuz);
      g3d.addLight(luzAmbiente, null);
      g3d.clear(fondo);

      g3d.render(vb, tsa, ap, tCubo);
    } finally {
      g3d.releaseTarget();
    }
  }
}
```

2.7. Modo retained

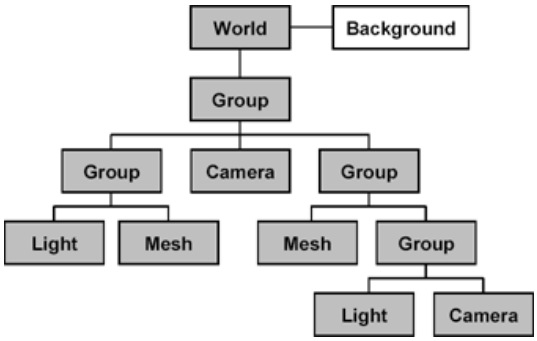
Utilizando este modo trabajaremos con un grafo en el que tendremos los distintos elementos de la escena 3D: objetos 3D, luces, cámaras, etc. Este grafo estará compuesto por objetos derivados de `Node` (nodos). Tenemos los siguientes tipos de nodos disponibles:

- **`World`**: El nodo raíz de este grafo es del tipo `World`, que representa nuestro mundo 3D completo. De él colgaremos todos los objetos del mundo, las cámaras y las luces. Además, el nodo `World` tendrá asociado el fondo (`Background`) de la escena.
- **`Group`**: Grupo de nodos. Nos permite crear grupos de nodos dentro del grafo. De él podremos colgar varios nodos. El tener los nodos agrupados de esta forma nos permitirá, aplicando transformaciones geométricas sobre el grupo, mover todos estos nodos como un único bloque. Podemos crear un nuevo grupo creando un nuevo objeto `Group`, y añadir nodos hijos mediante su método `addChild`.
- **`Camera`**: Define una cámara (punto de vista) en la escena. Las cámaras definen la posición del espectador en la escena. Podemos tener varias cámaras en el mundo, pero en un momento dado sólo puede haber una cámara activa, que será la que se utilice para renderizar. Se crean como hemos visto en apartados anteriores. El objeto correspondiente al mundo (`World`) tiene un método `setActiveCamera` con el que podremos establecer cuál será la cámara activa en cada momento.
- **`Light`**: Define las luces de la escena. Se crean como hemos visto en apartados anteriores.
- **`Mesh`**: Define un objeto 3D. Este objeto se puede crear a partir de sus vértices y caras como vimos en el apartado anterior.

```
Mesh obj = new Mesh(vb, tsa, ap);
```

- **`Sprite3D`**: Representa una imagen 2D posicionada dentro de nuestro mundo 3D y alineada con la pantalla.

Copyright © 2010 Depto. CCIA All rights reserved.



Ejemplo de grafo de la escena

En este modo normalmente cargaremos estos componentes de la escena 3D de un fichero con formato M3G. Para crear mundo en este formato podremos utilizar herramientas como Swerve Studio. De este modo podremos modelar objetos 3D complejos utilizando una herramienta adecuada, y posteriormente importarlos en nuestra aplicación.

Para cargar objetos 3D de un fichero M3G utilizaremos un objeto `Loader` de la siguiente forma:

```
World mundo;
...
try {
  Object3D [] objs = Loader.load("/mundo.m3g");
  mundo = (World)objs[0];
} catch (IOException e) {
  System.out.println("Error al cargar modelo 3D: " +
                     e.getMessage());
}
```

Con `load` cargaremos del fichero M3G especificado los objetos 3D que contenga. La clase `Object3D` es la superclase de todos los tipos de objetos que podemos utilizar para definir nuestra escena. De esta forma permitimos que esta función nos devuelva cualquier tipo de objeto, según lo que haya almacenado en el fichero. Estos objetos pueden ser nodos como los vistos anteriormente, animaciones, imágenes, etc.

Por ejemplo, si tenemos un fichero M3G que contiene un mundo 3D completo (objeto `World`), cogeremos el primer objeto devuelto y haremos una conversión cast al tipo adecuado (`World`).

Como en este objeto se define la escena completa, no hará falta añadir nada más, podemos renderizarlo directamente con:

```
protected void paint(Graphics g) {
  try {
    g3d.bindTarget(g);
    g3d.render(mundo);
  } finally {
    g3d.releaseTarget();
  }
}
```


Copyright © 2010 Depto. CCIA All rights reserved.

En el mundo 3D del fichero, a parte de los modelos 3D de los objetos, luces y cámaras, podemos almacenar animaciones predefinidas sobre elementos del mundo. Para utilizar esta animación llamaremos al método `animate` sobre el mundo que queremos animar:

```
protected void paint(Graphics g) {
  try {
    g3d.bindTarget(g);
    mundo.animate(tiempo);
    g3d.render(mundo);
  } finally {
    g3d.releaseTarget();
  }
}
```

Le deberemos proporcionar un valor de tiempo, que deberemos ir incrementando cada vez que se pinta. Podemos crear un hilo que cada cierto periodo incremente este valor y llame a `repaint` para volver a renderizar la escena.

A continuación vemos el código completo el ejemplo de mundo que utiliza modo *retained*:

```
package es.ua.j2ee.m3d;

import java.io.IOException;

import javax.microedition.lcdui.*;
import javax.microedition.m3g.*;

public class Visor3DRetained extends Canvas implements Runnable {

  MIDlet3D owner;

  Graphics3D g3d;
  World mundo;

  int tiempo = 0;

  public Visor3DRetained(MIDlet3D owner) {
    this.owner = owner;
    init3D();
  }

  public void init3D() {
    g3d = Graphics3D.getInstance();

    try {
      Object3D [] objs = Loader.load("/mundo.m3g");
      mundo = (World)objs[0];
    } catch (IOException e) {
      System.out.println("Error al cargar modelo 3D: " +
                         e.getMessage());
    }
  }

  protected void showNotify() {
    Thread t = new Thread(this);
```


Copyright © 2010 Depto. CCIA All rights reserved.

```
    t.start();
  }

  public void run() {
    while(true) {
      tiempo+=10;
      repaint();
      try {
        Thread.sleep(25);
      } catch (InterruptedException e) { }
    }
  }

  protected void paint(Graphics g) {
    try {
      g3d.bindTarget(g);

      mundo.animate(tiempo);
      g3d.render(mundo);
    } finally {
      g3d.releaseTarget();
    }
  }
}
```


Copyright © 2010 Depto. CCIA All rights reserved.

Copyright © 2010 Depto. CCIA All rights reserved.