

# PRÁCTICA 5

## Gráficos 2D

Parte 1

El objetivo de esta práctica es adentrarnos en las principales características de la programación de gráficos usando la tecnología Java2D; concretamente, abordaremos los distintos *Shape* que ofrece la tecnología y la ventaja de tener objetos asociados a las formas. Una vez practicados estos fundamentos, ampliaremos la práctica 4 para añadirle nuevas funcionalidades.

Para alcanzar el objetivo anterior, iremos siguiendo el tutorial “[2D Graphics](#)” de Java.

### ■ Formas geométricas: Shape

En este primer bloque probaremos los distintos *Shape* que ofrece el Java2D. Para ello, crearemos una aplicación sencilla e iremos siguiendo el tutorial en su capítulo “[Working with Geometry](#)”. Concretamente, iremos realizando lo siguiente:

- Usando NetBeans, y siguiendo la misma dinámica que en prácticas anteriores, crear una aplicación que incorpore una ventana principal (*JFrame*) y sobrecargar el método *paint* para usar *Graphics2D*<sup>1</sup>:

```
public void paint(Graphics g) {  
    super.paint(g);  
    Graphics2D g2d=(Graphics2D)g;  
  
    //Código usando g2d  
}
```

- Probar todos los *Shape* definidos en Java2D. Para ello, nos guiaremos por los ejemplos que podemos encontrar en el capítulo “[Working with Geometry](#)” del tutorial. El código lo iremos incorporando en el método *paint*, incluyendo, para cada forma que probemos<sup>2</sup>:
  1. La creación de la forma (*new*) usando el constructor correspondiente
  2. El dibujo de la forma creada usando las sentencias *draw* y *fill* de la clase *Graphics2D*.

Por ejemplo, para el caso de la línea<sup>3</sup>:

```
public void paint(Graphics g) {  
    super.paint(g);  
    Graphics2D g2d=(Graphics2D)g;  
  
    // Línea  
    Point2D p1=new Point2D.Float(70,70);  
    Point2D p2=new Point2D.Float(200,200);  
    Line2D linea = new Line2D.Float(p1,p2);  
    g2d.draw(linea);  
}
```

<sup>1</sup> Otra opción es incorporar una clase *Lienzo* (que herede de *JPanel*), en la línea que se hizo en la práctica 4, y trabajar sobre ese lienzo en lugar de sobre la ventana principal. Esto evitará parpadeos en la renderización.

<sup>2</sup> Se recomienda crear un método *pruebaShape* (*Graphics2D g2d*) en el que incluir todo el código de prueba; este método será llamado desde el método *paint*.

<sup>3</sup> Como es sabido, en el método *paint* no se deben crear los objetos de formas *Shape*; no obstante, y para esta prueba, haremos la excepción. Más adelante se hará correctamente (con eventos)

Dentro del capítulo “Working with Geometry” del tutorial, la sección “[Drawing Geometric Primitives](#)” incluye ejemplos correspondientes a los *Shape*:

- [Line2D](#)
- [Rectangle2D](#)
- [RoundRectangle2D](#)
- [Ellipse2D](#)
- [Arc2D](#)
- [QuadCurve2D](#)
- [CubicCurve2D](#)

Analizar los ejemplos del tutorial e incorporar las formas anteriores en nuestra aplicación.

- En la sección “[Drawing Arbitrary Shapes](#)” del mismo capítulo, se explica el uso de la forma “Trazo libre” ([GeneralPath](#)). Analizar los ejemplos del tutorial e incorporarlos a nuestra aplicación.
- Por último, y para concluir con la prueba de los *Shape* existentes, trabajaremos la forma [Area](#) que permite definir nuevas figuras mediante la composición de otras. Para ello, seguiremos el capítulo “[Advanced topics in Java 2D](#)” del tutorial, en su apartado “[Constructing Complex Shapes from Geometry Primitives](#)”. Así, y siguiendo el ejemplo del tutorial, incorporaremos en el método el código que permita dibujar una forma “pera” diseñada mediante objeto *Area*.

## ■ Ampliando la práctica 4

En los ejemplos anteriores hemos usado coordenadas fijas para probar las distintas formas. El objetivo era conocer los objetos *Shape* que nos ofrece Java2D, de ahí que hayamos optado por ejemplos sencillos que no implicasen interacción con el usuario. No obstante, en la práctica 4 ya trabajamos de forma interactiva con tres formas: líneas, rectángulos y elipses; el objetivo ahora es mejorar la funcionalidad de esa práctica aprovechando que tenemos objetos asociados a las formas dibujadas.

### ■ Mantener las formas: vector de *Shape*

En el ejercicio desarrollado en la práctica 4 sólo se mostraba la última figura dibujada; el objetivo de este bloque es conseguir que las figuras que vayamos dibujando se mantengan en el lienzo. Para ello, en la clase *Lienzo* definida en la práctica 4:

1. Creamos un vector de formas:

```
| private List<Shape> vShape = new ArrayList();
```

2. En el método *paint* dibujamos el contenido del vector<sup>4</sup>

```
public void paint(Graphics g {  
    super.paint(g);  
    Graphics2D g2d = (Graphics2D) g;  
    for(Shape s:vShape)  
        g2d.draw(s);  
}
```

<sup>4</sup> Asumimos que la forma que se está dibujando en un momento dado estará almacenada en la última posición del vector

3. Cada nueva figura se introducirá en el vector:

```

private void formMousePressed(java.awt.event.MouseEvent evt) {
    .
    .
    . //Código realizado en la práctica 4 para crear la figura
    . //de dibujo y asignarla a la variable 'forma'
    .
    vShape.add(forma);
}

```

## ■ Moviendo las figuras...

El objetivo de este bloque es mover las figuras ya pintadas; en este ejemplo no centraremos en el caso del rectángulo, asumiendo que son objetos de la clase *Rectangle*<sup>5</sup>. Para ello, incluiremos en la ventana principal una casilla de verificación (*JCheckBox*) a la que llamaremos “mover”, de forma que, si está seleccionada, indicará que la interacción del usuario será para mover los rectángulos (en caso contrario, dicha interacción implicará la incorporación de nuevas figuras). Concretamente, el usuario podrá situarse sobre cualquiera de los rectángulos dibujados y moverlos mediante la secuencia *pressed*→*dragged*→*released*.

Algunas consideraciones:

- La clase *Shape* ofrece el método *contains(Point2D)* que comprueba si un punto está dentro de la correspondiente forma<sup>6</sup>. Este método puede usarse para, dado un vector de figuras, localizar aquella(s) forma(s) que esté(n) situada(s) en la posición donde se ha hecho un clic (o *pressed*). Por ejemplo, dado un vector *vShape* de objetos *Shape*, el siguiente código seleccionaría la primera figura situada bajo del punto *p* (null si no hubiese ninguna):

```

private Shape getFiguraSeleccionada(Point2D p) {
    for(Shape s:vShape)
        if(s.contains(p)) return s;
    return null;
}

```

---

<sup>5</sup> En java existe la clase abstracta *Rectangle2D* de la cual heredan las subclases *Rectangle*, *Rectangle2D.Float* y *Rectangle2D.Double*. La diferencia entre estas tres clases es el tipo (y con ello la precisión) usado para las coordenadas que definen al rectángulo (entero, flotante o doble, respectivamente). El conjunto de métodos es diferente en las tres clases (excepto los heredados), siendo la clase *Rectangle* la que ofrece mayor riqueza semántica.

<sup>6</sup> En el caso de líneas (objetos *Line2D*), el método *contains* siempre devuelve *false* por no haber área asociada. Por este motivo, si se quisiese seleccionar una línea, habría que usar un método que comprobase si un punto está cerca de la línea (en lugar de contenido en ella); dicho método no existe en la clase *Line2D*, por lo que es necesaria su implementación. Una posible codificación sería:

```

public boolean isNear(Point2D p) {
    // Caso p1=p2 (punto)
    if(this.getP1().equals(this.getP2())) return this.getP1().distance(p)<=2.0;
    // Caso p1!=p2
    return this.ptLineDist(p)<=2.0;
}

```

asumiendo que dicho método pertenece a una clase propia que hereda de *Line2D*. Una vez definido este método, podría optarse por sobrecargar el método *contains* de la siguiente forma:

```

@Override
public boolean contains(Point2D p) {
    return isNear(p);
}

```

asumiendo que está contenido si está cerca

- Para mover un rectángulo, la clase `Rectangle` ofrece el método `setLocation(Point)`, que sitúa su coordenada superior izquierda en el punto pasado como argumento (manteniendo el ancho y alto)<sup>7</sup>.
- En los manejadores de eventos asociados al ratón, habrá que considerar si se está o no en “modo mover” (es decir, desplazando formas). Por ejemplo, en el caso del `pressed`:

```
private void formMousePressed(java.awt.event.MouseEvent evt) {
    if(mover) {
        forma = getFiguraSeleccionada(evt.getPoint());
    }
    else {
        .
        .
        //Código realizado en la práctica 4
        .
        vShape.add(forma);
    }
}
```

y en el del dragged:

```
private void formMouseDragged(java.awt.event.MouseEvent evt) {
    if(mover) {
        //Código para el caso del rectángulo
        if (forma!=null && forma instanceof Rectangle)
            ((Rectangle) forma).setLocation(evt.getPoint());
    }
    else {
        .
        .
        //Código realizado en la práctica 4
        .
        .
    }
    this.repaint();
}
```

Siguiendo el esquema anterior, incluir en `Lienzo` el código necesario para mover también las líneas y las elipses<sup>8</sup>.

---

<sup>7</sup> Salvo la clase `Rectangle`, el resto de clases `Shape` no tienen el método `setLocation` (por ejemplo, las clases `Line2D`, `Rectangle2D` o `Ellipse2D` no lo incluye); en estos casos, debemos de implementarlo si queremos mover la forma. Por ejemplo, una posible codificación para el caso de la línea podría ser:

```
public void setLocation(Point2D pos) {
    double dx=pos.getX()-this.getX1();
    double dy=pos.getY()-this.getY1();
    Point2D newp2 = new Point2D.Double(this.getX2()+dx, this.getY2()+dy);
    this.setLine(pos,newp2);
}
```

asumiendo que dicho método pertenece a una clase propia que hereda de `Line2D`. En el caso de `Rectangle2D` o `Ellipse2D` habría que usar el método `setFrame` (pasándole como parámetros x e y la nueva localización, y como alto y ancho los valores actuales del rectángulo/elipse).

<sup>8</sup> Para el caso de línea será necesario crear una clase propia, heredando de `Line2D.Float`, donde definir el método `setLocation` (véase nota de página anterior); esto implicará que cuando se cree la línea (en el `pressed`), habrá que hacer el `new` usando la nueva clase. Para el caso de la elipse, se recomienda seguir la misma estrategia (y crear una clase propia que defina un método `setLocation`).

## ■ Incorporando ventanas internas

Por último, vamos a convertir nuestra aplicación en un entorno multiventana, esto es, vamos a introducir un escritorio y trabajar con ventanas internas. Para ello<sup>9</sup>, aplicaremos lo ya visto en la práctica 3:

- Anadir un escritorio en el centro de la ventana principal (donde se tuviese el lienzo en la práctica 4, que ahora se elimina de esta ventana)<sup>10</sup>. Para ello, incorporar un *JDesktopPane* usando el NetBeans (área “contenedores swing”).
- Crear una clase propia *VentanaInterna* que herede de *JInternalFrame* (para ello, usar NetBeans). Activar las propiedades “closable”, “iconifiable”, “maximizable” y “resizable”.
- Añadir un *Lienzo* (clase creada en la práctica 4) al centro de la ventana interna definida en el punto anterior (usar el “arrastrar y soltar” que ofrece NetBeans, de la misma forma que se hizo en la práctica 4). Llamarle a la variable *lienzo* (nótese que en este caso se tratar de una variable definida en la clase *VentanaInterna*).
- Para incorporar una ventana interna, hay que (i) crear el objeto, (ii) añadirlo al escritorio y (iii) mandar el mensaje para visualizarla. Así, por ejemplo, en el manejador del evento asociado a la opción “Nuevo”, tendríamos:

```
private void menuNuevoActionPerformed(ActionEvent evt) {  
    VentanaInterna vi = new VentanaInterna();  
    escritorio.add(vi);  
    vi.setVisible(true);  
}
```

- Si en un momento dado queremos acceder a la ventana que esté activa en el escritorio (p.e., desde un método gestor de eventos), el siguiente código devuelve la ventana activa (*null* si no hay ninguna):

```
| VentanaInterna vi = (VentanaInterna)escritorio.getSelectedFrame();
```

A través de *vi* podemos acceder, por ejemplo, al lienzo de la ventana activa y a sus métodos y variables.

En particular, se aconseja definir en la clase *VentanaInterna* un método *getLienzo()* que devuelva el lienzo asociado a dicha ventana:

```
public Lienzo getLienzo() {  
    return lienzo;  
}
```

Siguiendo la misma filosofía, se aconseja definir un método en la *VentanaPrincipal* que devuelva el lienzo de la ventana interna activa (*null* si no hubiera ninguno):

```
public Lienzo getLienzoSeleccionado() {  
    VentanaInterna vi = (VentanaInterna)escritorio.getSelectedFrame();  
    return vi!=null ? vi.getLienzo() : null;  
}
```

Así, donde en la práctica 4 usábamos la variable *lienzo*, ahora lo sustituiremos por una llamada a *getLienzoSeleccionado()*, junto con la comprobación de que no sea *null*.

<sup>9</sup> Si tienes dudas, puedes consultar el vídeo tutorial que hay en PRADO (módulo de la práctica 7).

<sup>10</sup> Al quitar el lienzo de la zona central para colocar el escritorio, perderemos la referencia (variable) al lienzo y, con ello, se marcará como error aquellas líneas de código que usen dicha referencia. En los siguientes pasos se solventará este problema.