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.

Para alcanzar el objetivo anterior, iremos siguiendo el tutorial **"***2D Graphics***"** de Java (http://docs.oracle.com/javase/tutorial/2d/index.html).

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`:

```
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 lo 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[1]:

  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:

```
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);
}
```

[1] 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`.

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`.

Incorporando eventos…

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. En esta parte de la práctica, incorporaremos la gestión de eventos en alguno de los ejemplos anteriores. Concretamente, incluiremos la interacción con el usuario para el dibujo de un rectángulo.

En este caso, a diferencia de prácticas anteriores (en las que sólo disponíamos de la clase `Graphics)`, ya trabajamos con las posibilidades que ofrece `Graphics2D`; en particular, contamos con clases para las distintas formas, por lo que en este ejemplo se usará un objeto `Rectangle`[2] al que mandaremos mensajes para actualizar sus valores[3,4]:

```
1 | Rectangle rectangulo;

2 | public void paint(Graphics g) {
  |   super.paint(g);
  |   Graphics2D g2d=(Graphics2D)g;
  |   if(rectangulo!=null) g2d.draw(rectangulo);
  | }
```

[2] 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.

[3] El ejemplo incluye los métodos incorporados por NetBeans cuando se gestionan los correspondientes eventos y que son llamados desde la clase manejadora (no se incluye dicha clase ni sus métodos en el código).

[4] Si en lugar de un rectángulo estuviéramos pintando una línea, en el método `formMousePressed` crearíamos un objeto línea (asignado a una variable línea de tipo `Line2D`) mediante la sentencia `linea = new Line2D.Float(p,p)` y en el método `formMouseDragged` iríamos actualizándola mediante la sentencia `linea.setLine(p,evt.getPoint())`.

③

```
private void formMousePressed(java.awt.event.MouseEvent evt) {
  p = evt.getPoint();
  rectangulo = new Rectangle(p);
}

private void formMouseDragged(java.awt.event.MouseEvent evt) {
  rectangulo.setFrameFromDiagonal(p, evt.getPoint());
  this.repaint();
}

private void formMouseReleased(java.awt.event.MouseEvent evt){
  this. formMouseDragged(evt);
}
```

Mantener las formas: vector de *Shape*

En el ejemplo anterior sólo se mostraba el último rectángulo pintado; el objetivo de este bloque es conseguir que los rectángulos que vayamos dibujando se mantengan en el lienzo. Para ello:

1. Creamos un vector de formas:

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

2. En el método `paint` dibujamos el contenido del vector[5]

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

3. Cada nuevo rectángulo se introducirá en el vector:

```
private void formMousePressed(java.awt.event.MouseEvent evt) {
  p = evt.getPoint();
  rectangulo = new Rectangle(p);
  vShape.add(rectangulo);
}
```

Moviendo las figuras…

El objetivo de este bloque es mover los rectángulos que ya estén dibujados. Para ello, incluiremos en la ventana principal una casilla de verificación (`JCheckBox`), de forma que, si está seleccionada, indicará que la interacción del usuario será para mover las líneas (en caso contrario, dicha interacción implicará la incorporación de nuevas líneas). Concretamente, el usuario podrá situarse sobre cualquiera de las líneas dibujadas y moverlas 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[6]. Este método puede usarse para,

[5] Asumimos que la forma que se está dibujando en un momento dado estará almacenada en la última posición del vector

dado un vector de figuras, localizar aquella(s) forma(s) que está(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 getSelectedShape(Point2D p){
  for(Shape s:vShape)
    if(s.contains(p)) return s;
  return null;
}
```

- 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)[7].

- En los manejadores de eventos asociados al ratón, habrá que considerar si se está o no en "modo edición" (es decir, desplazando formas). Por ejemplo:

```
private void formMouseDragged(java.awt.event.MouseEvent evt) {
  if(editar){
    if(rectangulo!=null) rectangulo.setLocation(evt.getPoint());
  }
  else rectangulo.setFrameFromDiagonal(p, evt.getPoint());
  this.repaint();
```

De igual forma, habrá que considerar si se está o no en edición en los métodos asociados al *pressed* y al *released*.

---

[6] 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){
   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

[7] No todos los `Shape` tienen el método `setLocation` (por ejemplo, la clase `Line2D` 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.