LIBRERÍA GEOMÉTRICA en Java

Sistemas Informáticos
2001−2002

Pablo Suau Pérez

Índice de contenidos

1.INTRODUCCIÓN

La librería geométrica JavaRG surge, a partir de las prácticas de la asignatura *Razonamiento Geométrico* de la *Universidad de Alicante*, como una necesidad de disponer de un conjunto de clases que nos permitan implementar algoritmos geométricos de forma sencilla teniendo a nuestra disposición elementos como puntos, segmentos, rectas, etc... De esta forma tan solo debemos preocuparnos del diseño del algoritmo geométrico en sí mismo, creando los objetos geométricos que necesitemos y llamando a las funciones adecuadas, sin necesidad de conocer detalles de más bajo nivel.

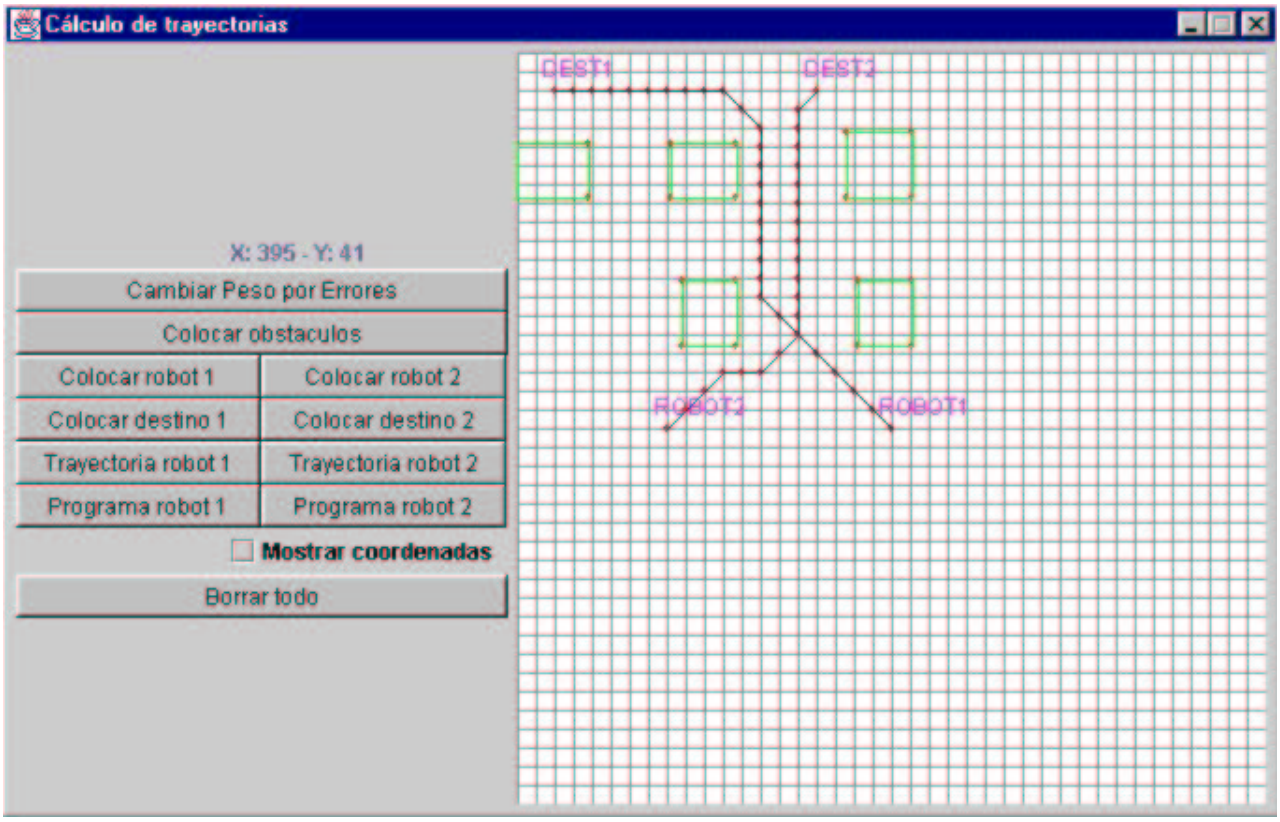
A la hora de decidir la estructura de la librería se tomó como base la librería matemática **CGAL**, con la clara diferencia del lenguaje de programación empleado; C++ en el caso de CGAL, y Java en el caso de JavaRG. Se ha intentado respetar tanto la estructura de la citada librería, así como las clases presentes y sus métodos, añadiéndose nuevas características conforme se ha creido oportuno. Por supuesto la cantidad de tiempo disponible para la realización de *JavaRG* no permitie la implementación de todas y cada una de las partes de *CGAL*, así el trabajo se centró en las capas más bajas de la librería que proporcionaban las funcionalidades básicas.

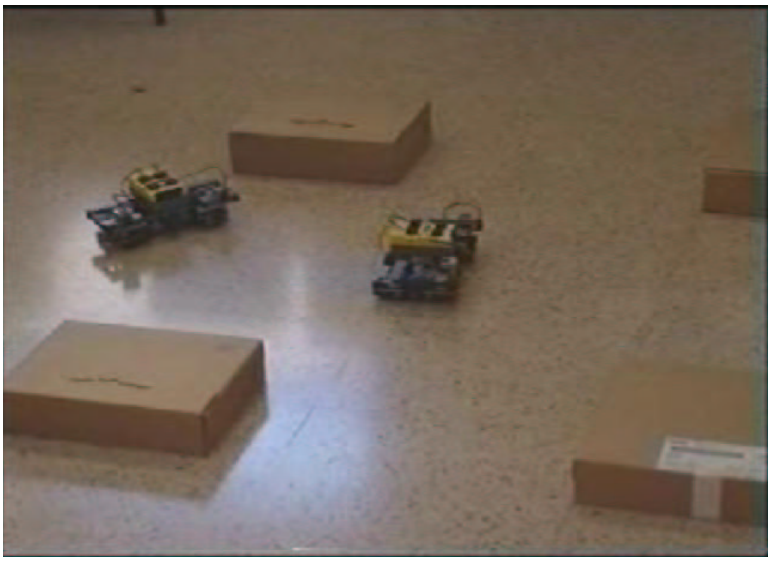
La librería viene acompañada de una aplicación que nos permite observar de forma gráfica, mediante componentes *Swing* de Java, el funcionamiento de sus métodos principales. Podremos dibujar objetos geométricos, y realizar operaciones con ellos, así como con otros objetos ya dibujados. Uno de los beneficios de haber incluido esta aplicación es que se han incorporado a la librería una serie de métodos de dibujo de los objetos, de tal forma que estos además podrán ser empleados en nuestras aplicaciones gráficas (característica de la que *CGAL* no dispone).

Además de las características explicadas, y debido a que se intentó en todo momento respetar la estructura de la librería CGAL original, la modularidad de *JavaRG* permite su futura ampliación añadiendo funcionalidades como algoritmos geométricos basados en los objetos geométricos de la librería, por ejemplo. Así pues, y aunque la interfaz que acompaña a la librería pueda considerarse como una aplicación independiente y finalizada, un futuro trabajo podría consistir precisamente en añadir funcionalidades a la librería para acercarnos más al objetivo de conseguir un producto similar a *CGAL* empleando el lenguaje Java.

Por último comentar que, además de haber sido utilizada como material didactico en la asignatura *Razonamiento Geométrico* durante el curso 2001−2002, la librería geométrica ha sido tomada como base de otros dos proyectos, basados en los robots de *Lego Mindstroms:*

- *Cálculo de trayectorias con evitación de colisiones*: el objetivo de este proyecto es el de diseñar una aplicación de cálculo de trayectorias de dos robots, de tal forma que se eviten las colisiones entre los mismos. La forma de evitar colisiones era mediante el envío de mensajes entre los robots cuando éstos entraban en zonas conflictivas de su trayectoria, donde se podría producir una colisión. Tras calcular las trayectorias mediante la aplicación ésta producía como salida un programa para cada robot en Java, de tal forma que una vez descargado en los robots (tras modificar su firmware introduciéndoles un sistema operativo basado en una máquina virtual de Java llamado *lejos*) éstos realizaran de forma real la trayectoria y la comunicación. La librería *JavaRG* fue utilizada tanto para la creación de la parte gráfica de la aplicación (dibujo de obstáculos, posición de robots y trayectorias) como para el cálculo de los puntos de colisión. Para ello, las trayectorias son tratadas como listas de segmentos, de tal forma que los puntos de colisión eran las intersecciones entre segmentos de las trayectorias de los distintos robots.





- *Mindstorms VDK (Virtual Development Kit)*: conjunto de aplicaciones que tienen como propósito ofrecer un entorno de desarrollo y prueba de robots Mindstorms de una forma virtual. Para ello ofrece herramientas de edición de programas, de generación de entornos, y finalmente de simulación y emulación. Concretamente la librería es utilizada en la simulación de los sensores del robot, así como en la simulación del propio robot, y en cálculos con sistemas de coordenadas.

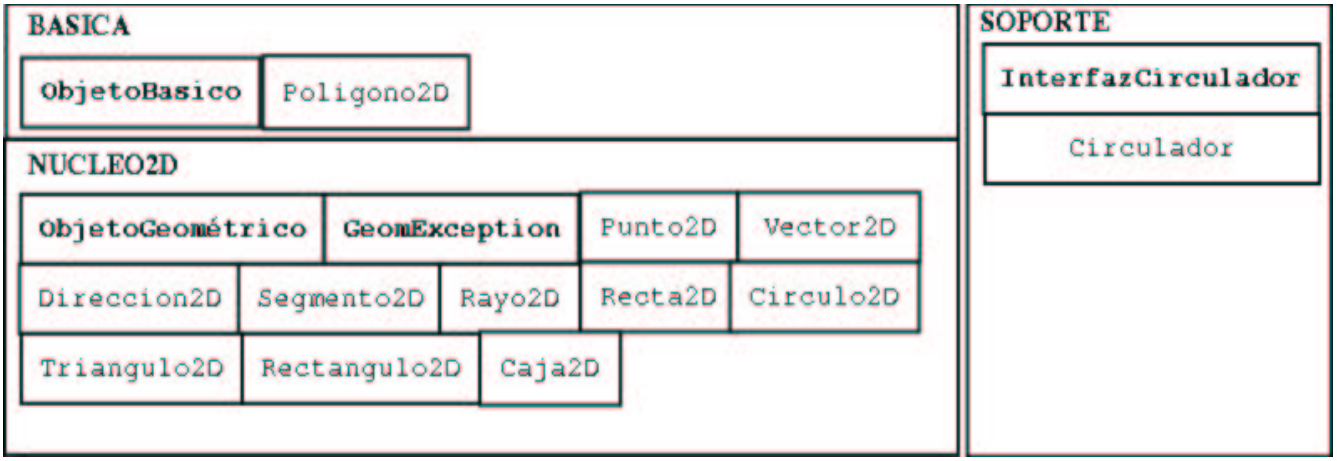
2.Tutorial de uso de la librería JavaRG

El siguiente texto trata de ser una breve introducción a los conceptos básicos de la librería, así como de sus clases y métodos principales, con el objetivo de que el programador pueda incorporarlos a sus futuras aplicaciones de geometría computacional. Para ello primero se comenzará explicando como está estructurada la librería (los paquetes y clases que contiene), se mostrarán ejemplos de dificultad incremental de programas que hagan uso de la librería, y por último, se incluirá un ejemplo más desarrollado que haga uso de las facilidades de dibujo incluidas.
Para obtener una información más detellada y completa de las funcionalidades de la librería se aconseja consultar el API que acompaña a la misma, en el interior del directorio **javadoc**.
El código fuente de todos los ejemplos mostrados se puede encontrar en el directorio **ejemplos.**

2.1.Estructura de la librería

Tal como se comenta en la introducción, la librería JavaRG está basada en la librería matemática **CGAL** (http://www.cgal.org). Esta librería está compuesta por tres capas, más una librería de apoyo (support library), que se encuentra aparte. Estas tres capas son la *core library* con la funcionalidad básica no geométrica, el *geometric kernel* o núcleo geométrico con los objetos y operaciones geométricos básicos y el *basic library* con los algoritmos y las estructuras de datos:

La estructura de **JavaRG** es similar, como puede verse en la siguiente figura:



Cada uno de estos componentes en los que se divide la librería se implementa en forma de paquete, de tal forma que son tres los paquetes los que disponemos:

- **Nucleo2D**: consiste en los objetos geométricos simples como el punto, el vector, la dirección, la recta, el triángulo, etc, todos ellos dentro del dominio de las dos dimensiones. Cada objeto incluye operaciones para calcular intersecciones, realizar transformaciones afines, etc.
- **Basica**: este paquete contiene objetos más complejos. De momento solo incluye el polígono en 2D, pero este debería ser el lugar donde deberían almacenarse otros objetos que hagan uso de varios de los objetos del núcleo, así como estructuras de datos (por ejemplo, mapas planares, superficies poliédricas, etc.).
- **Soporte:** Contiene el circulador, un objeto relacionado con los objetos geométricos anteriormente indicados, pero que no es un objeto geométrico en sí mismo. En este paquete deberían incluirse todas aquellas clases que, como el circulador, nos proporcionen funcionalidades adicionales y que no sean objetos geométricos simples o complejos, o estructuras de datos formadas a partir de éstos.

Todos estos paquetes estarán incluidos dentro del paquete **JavaRG**. Fuera de este paquete encontramos la clase **Mat**, de gran importancia ya que contiene una serie de valores constantes y métodos auxiliares (como, por ejemplo, la determinación del valor absoluto) utilizados por la práctica totalidad de los componentes de la librería.
Por último destacar que dentro del paquete *JavaRG* también podemos encontrar un paquete especial llamado **Interfaz**. Dicho paquete contiene las clases relacionadas con la aplicación gráfica en la que se hace mención en el capítulo de *Manual del usuario de la interfaz*, y aunque relacionado de forma estrecha con la librería geométrica, **no** forma parte de ella.

2.2.Un primer ejemplo

Este primer ejemplo está destinado a tener un primer contacto con la librería, así como una visión de sus características principales.

Prueba.java

```
// Importamos la parte básica de la librería
import JavaRG.Mat;
import JavaRG.Nucleo2D.*;

public class Prueba {
    public Prueba() {
    }

    public static void main (String []args) throws GeomException {
        // Creamos el primer segmento
        float inicioX = 10, finX = 15;
        float inicioY = 11, finY = 6;
        Segmento2D segmento1 = new Segmento2D(inicioX, inicioY, finX, finY);
        segmento1.cambiarNombre("Segmento 1");
        System.out.println("Creado el primer segmento");
        System.out.println(segmento1);

        // Creamos el segundo segmento
        Punto2D inicio = new Punto2D(14,7);
        Punto2D fin = new Punto2D(17,4);
        Segmento2D segmento2 = new Segmento2D(inicio, fin);
        segmento2.cambiarNombre("Segmento 2");
        System.out.println("Creado el segundo segmento");
        System.out.println(segmento2);
```

```
            // Creamos el tercer segmento
            Segmento2D segmento3 = new Segmento2D(segmento1);
            segmento3 = segmento3.gira(Mat.grad2Rad(90));
            segmento3 = segmento3.trasladar(20, –5);
            segmento3.cambiarNombre("Segmento 3");
            System.out.println("Creado el tercer segmento");
            System.out.println(segmento3);

            // Calculamos intersecciones
                  System.out.println("Interseccion de '" + segmento1.obtenerNombre() + "' y '" +
segmento2.obtenerNombre() + "' : ");
            ObjetoGeometrico resultado1 = segmento1.interseccionSegmento(segmento2);
            System.out.println(resultado1);
                  System.out.println("Interseccion de '" + segmento1.obtenerNombre() + "' y '" +
segmento3.obtenerNombre() + "' : ");
            ObjetoGeometrico resultado2 = segmento1.interseccionSegmento(segmento3);
            System.out.println(resultado2);
                  System.out.println("Interseccion de '" + segmento2.obtenerNombre() + "' y '" +
segmento3.obtenerNombre() + "' : ");
            ObjetoGeometrico resultado3 = segmento2.interseccionSegmento(segmento3);
            System.out.println(resultado3);

            // Ejemplo de excepción geométrica
            try {
                  Segmento2D segmento4 = new Segmento2D(inicio, inicio);
            } catch (GeomException e) {
                  System.out.println(e);
            }
      }
}
```

Este ejemplo crear tres segmentos de diversas formas y calcula algunas intersecciones entre ellos, mostrando los resultados por pantalla. La salida que genera es la siguiente:

```
Creado el primer segmento
Segmento 1 – Segmento2D: ( – Punto2D: (x=10.0,y=11.0), – Punto2D: (x=15.0,y=6.0))
Creado el segundo segmento
Segmento 2 – Segmento2D: ( – Punto2D: (x=14.0,y=7.0), – Punto2D: (x=17.0,y=4.0))
Creado el tercer segmento
Segmento 3 – Segmento2D: ( – Punto2D: (x=9.0,y=5.0), – Punto2D: (x=14.0,y=10.0))
Interseccion de 'Segmento 1' y 'Segmento 2' :
 – Segmento2D: ( – Punto2D: (x=14.0,y=7.0), – Punto2D: (x=15.0,y=6.0))
Interseccion de 'Segmento 1' y 'Segmento 3' :
 – Punto2D: (x=12.5,y=8.5)
Interseccion de 'Segmento 2' y 'Segmento 3' :
null
JavaRG.Nucleo2D.GeomException: Segmento2D (Point2D, Point2D): igual punto de inicio y de final
```

Los tres segmentos generados son *Segmento 1*, con un extremo en (10, 11) y otro en (15, 6), *Segmento 2*, cuyos extremos son (14, 7) y (17, 4), y *Segmento 3*, con extremos en los puntos (9, 5) y (14, 10). El resultado de la intersección entre *Segmento 1* y *Segmento 2* será a su vez otro segmento con extremos en (14, 7) y (15, 6); la intersección entre *Segmento 1* y *Segmento 3* será el punto (12.5, 8.5), y finalmente se observa que los segmentos *Segmento 2* y *Segmento 3* no intersectan. Por último, observamos como se ha producido una excepción por intentar crear un nuevo segmento cuyos puntos de inicio y final son el mismo (un segmento degenerado). Pasamos a comentar el código.

Solo vamos a utilizar segmentos y puntos, por lo que el único paquete de la librería que deberemos incluir será **Nucleo2D**. Como también vamos a utilizar parte de las utilidades proporcionadas por la clase **Mat**, la importamos:

```
import JavaRG.Mat;
import JavaRG.Nucleo2D.*;
```

Al ser una sencilla prueba hemos creado una única clase, *Prueba*, que tendrá un único método relevante, el método *main*. En dicho método es donde se producirá la totalidad del proceso. El primer paso es crear los tres segmentos. La librería JavaRG es muy flexible en la medida que nos permite realizar ciertas operaciones de muchas formas distintas. El aspecto en el que esto se ve de forma más evidente es en la creación de objetos geométricos; por ejemplo, podremos crear segmentos a partir de las coordenadas cartesianas de sus dos extremos o sus coordenadas homogéneas, a partir de dos objetos de tipo Punto, a partir de otros segmentos, etc. En el ejemplo, el primer segmento es creado a partir de las coordenadas cartesianas de sus dos extremos:

```
float inicioX = 10, finX = 15;
float inicioY = 11, finY = 6;
Segmento2D segmento1 = new Segmento2D(inicioX, inicioY, finX, finY);
```

En JavaRG, cada objeto geométrico puede tener asociado un **nombre** que lo identifique, el cual es posible modificar con **cambiarNombre(String nombre)**, u obtener con **obtenerNombre()**:

```
segmento1.cambiarNombre("Segmento 1");
```

Este nombre puede ser cualquier cadena válida. Por último, antes de crear el siguiente segmento, mostramos por salida estándar información sobre el que acabamos de crear. Todos los objetos geométricos de JavaRG tienen sobrecargada la salida estándar de tal manera que al pasarlos como parámetro a *System.out.println()* se nos muestre información como sus coordenadas, si se trata de un punto, sus vértices, si se trata de un polígono, etc. , así como el nombre del objeto. En este caso se nos muestra el nombre del segmento, *Segmento1*, y entre paréntesis la información de los puntos que lo componen. Como ninguno de esos puntos tenía asignado ningún nombre, éste no aparecerá.

Para crear el segundo segmento se parte de dos objetos geométricos de tipo punto, que crearemos de la misma forma:

```
Punto2D inicio = new Punto2D(14,7);
Punto2D fin = new Punto2D(17,4);
Segmento2D segmento2 = new Segmento2D(inicio, fin);
```

La notación de *inicio* y *fin* de los dos puntos a partir de los cuales construimos el segmento ayuda a ver que los segmentos en JavaRG son segmentos *orientados*.
Con el tercer segmento complicamos un poco más la cosa. Comenzamos por crearlo como una copia del primer segmento:

```
Segmento2D segmento3 = new Segmento2D(segmento1);
```

y a continuación le aplicamos un par de transformaciones afines, un giro y una traslación. Los giros se hacen respecto al origen de coordenadas, es por ello que además debemos hacer una traslación para conseguir que *Segmento 3* intersecte con *Segmento 1*, a partir del cual se creó:

```
segmento3 = segmento3.gira(Mat.grad2Rad(90));
segmento3 = segmento3.trasladar(20, -5);
```

En estas líneas también se puede ver un ejemplo de uso de la clase **Mat**, que como se ha dicho contiene una serie de constantes y métodos útiles. Todos ellos están definidos como *static*, por lo que no es

necesario crear objetos de la clase Mat. En este caso lo hemos utilizado para convertir una cantidad en grados (90) a su correspondiente valor en radianes para pasarlo como parámetro a la función de giro.

A continuación calculamos las intersecciones. La intersección entre dos segmentos puede ser un punto, otro segmento, o ninguna intersección. Aquí es donde vemos otra característica importante de la librería. Todos los objetos geométricos del núcleo son subclases de **ObjetoGeométrico**. Una de las ventajas de esto se ve en el ejemplo, y es que la función de intersección nos devolverá el objeto geométrico resultante de la intersección, sea del tipo que sea (Segmento2D, Punto2D, o null). Esta clase *ObjetoGeométrico* es abstracta, pues contiene una serie de funciones abstractas que todas sus subclases deben implementar. Por ejemplo, podríamos calcular la distancia de un punto a cualquiera de los resultados devueltos de las intersecciones (excepto en el caso de que el resultado fuera null, lo cual tendríamos que controlar), pues al tener la clase *ObjetoGeométrico* el método abstracto *distancia()*, todas sus subclases (lo que incluye a Segmento2D y Punto2D) deben tener implementado este método. Otra ventaja más es que podríamos tener estructuras de datos como listas y colecciones conteniendo objetos geométricos de distinto tipo.

Lo último que nos muestra este ejemplo introductorio es la existencia de un nuevo tipo de excepción creada especialmente para la librería JavaRG; una expeción de tipo **GeomException** se lanzará para evitar ciertas situaciones, como crear direcciones a partir de vectores nulo, pasar como parámetro un número incorrecto de vértice, etc... En todos los casos el texto de la excepción nos muestra el método desde donde se lanzó, así como la razón. Concretamente en este ejemplo hemos empleado la excepción que se lanza cuando se crea un segmento cuyos dos extremos son iguales:

```
try {
        Segmento2D segmento4 = new Segmento2D(inicio, inicio);
} catch (GeomException e) {
        System.out.println(e);
}
```

A partir de este ejemplo y del API de la librería ya se puede comenzar a desarrollar aplicaciones geométricas sencillas. Las siguientes secciones de este tutorial penetrarán en aspectos más concretos del núcleo, y explicarán el resto de paquetes. La última parte pretende mostrar un ejemplo de aplicación gráfica que nos pueda servir como base para desarrollar nuestras aplicaciones geométricas basadas en Swing.

2.3.La clase Mat

Dentro del paquete JavaRG encontramos la clase *Mat*, cuyo cometido es almacenar una serie de constantes y métodos de uso común en la librería. Todos los métodos de Mat están definidos como *static*, de tal forma que no será necesario crear un objeto *Mat* para acceder a ellos (tal como se vio en el ejemplo anterior).
Quizás el valor más importante es el *epsilon de la aritmética en punto flotante*. Este valor es un indicativo de la precisión de las operaciones en punto flotante, y se usa para amortiguar los errores de redondeo. Se considera que cualquier valor inferior en valor absoluto a Mat.EPSILON es cero. Por ejemplo, el siguiente código:

```
float a, b;
a = 0.477;
b = java.lang.Math.log(3);
System.out.println(a – b == 0);
```

mostraría por pantalla el valor *false*, pues b no valdría exactamente 0.477. Si queremos despreciar los

decimales a partir de una determinada posición podemos usar Mat.EPSILON de la siguiente forma:

```
System.out.println(Mat.absoluto(a – b) <= Mat.EPSILON);
```

El resultado sería que se mostraria por la salida estándar el valor *true*.
Los métodos de *Mat* permiten obtener el valor absoluto de un número, convertir cantidades en radianes a grados y viceversa, aplicar el operando xor, y determinar si una matriz bidimensional es cuadrada.

2.4.Uso del paquete Nucleo2D

Este paquete contiene los objetos geométricos básicos, así como la clase abstracta **ObjetoGeometrico** a partir de la cual se crean todos ellos, y la excepción **GeomException**. En concreto, podremos crear los siguientes tipos de objetos:

- **Punto2D:** representa un punto en el plano. Aunque se puede manejar a partir de sus coordenadas cartesianas, internamente está implementado mediante coordenadas homogéneas.
- **Vector2D:** representa un vector de dos dimensiones. Geométricamente un vector es la diferencia entre dos puntos, y denota la dirección y la distancia desde el primer punto hasta el segundo. Está implementado a partir de desplazamientos en el eje x y en el eje y.
- **Direccion2D:** se trata de un vector de dos dimensiones del cual obviamos su longitud. Pueden ser vistos como vectores unitarios (aunque internamente no se produzca ninguna normalización) y también como ángulos. Su implementación, al igual que en el caso de los vectores, se basa en desplazamientos.
- **Segmento2D:** representa un segmento *orientado* en el plano. Está definido entre (e implementado a partir de) un punto de inicio y un punto final, ambos contenidos en el segmento.
- **Rayo2D:** representa un rayo en el espacio, siendo un rayo una línea que a partir de un punto de inicio se extiende infinitamente en una dirección. Un rayo estará definido por su ecuación paramétrica.
- **Recta2D:** representa una recta en el espacio, definida por su ecuación paramétrica (a partir de un punto base y un vector director).
- **Circulo2D:** representa un círculo orientado (su circunferencia asociada tiene un sentido horario o antihorario), que divide el espacio en una zona positiva y una zona negativa, quedando la positiva a la izquierda de la circunferencia.
- **Triangulo2D:** representa un triángulo orientado en el plano. Al contrario que con otro tipo de figuras geométricas, sí será posible la creación de triángulos degenerados (con sus tres vértices colineales), pues de lo contrario no sería posible detectar la colinealidad en otras figuras geométricas.
- **Rectangulo2D:** representa un rectángulo en el plano, con sus lados paralelos a los ejes x e y; estará definido por dos vértices, el inferior izquierdo y el superior derecho.
- **Caja2D:** representa una caja contenedora de objetos geométricos.

Todos estos objetos pueden crearse, tal como se vió en el ejemplo anterior, mediante el operador **new**, usando uno de los múltiples constructores que incorporan las clases. No hay que olvidar que cualquier objeto geométrico que creemos es también una instancia de *ObjetoGeométrico*. En el siguiente ejemplo vemos que podremos emplear métodos de *ObjetoGeométrico* (ya sean métodos implementados en esa misma clase o métodos abstractos implementados por cada una de sus subclases) sea cual sea el tipo de objeto geométrico que contiene. Sin embargo, para usar métodos específicos de una clase deberemos hacerlo a partir de una referencia de dicha clase:

EjemploObjeto.java

```
import JavaRG.Nucleo2D.*;

public class EjemploObjeto {
    public EjemploObjeto() {
    }

    public static void main (String args())
    {
        // Cualquier clase del paquete Nucleo puede ser referenciada como
        // ObjetoGeometrico
        ObjetoGeometrico punto = new Punto2D();
        ObjetoGeometrico segmento = new Segmento2D();

        // La función cambiarNombre está implementada en ObjetoGeometrico
        punto.cambiarNombre("EjemploPunto");
        segmento.cambiarNombre("EjemploSegmento");

        // La funcion toString() está sobrecargada por todas las subclases
        // de ObjetoGeometrico
        System.out.println(punto);
        System.out.println(segmento);

        // El método hx() de Punto2D solo puede ser llamada desde una referencia
        // de esa clase
        Punto2D copiaPunto = (Punto2D) punto;
        System.out.println(copiaPunto.hx());
    }
}
```

Esta característica de polimorfismo tiene la ventaja de que ciertos métodos (como los de intersecciones) pueden devolver un objeto geométrico u otro según el resultado, con tan solo devolver un *ObjetoGeométrico*. Algunos otros métodos admiten estos objetos como parámetro. Por ejemplo, cualquier objeto geométrico tiene implementado el método **equals()**, que acepta como parámetro un objeto geométrico. Cuando en java comparamos dos objetos con el operador == lo que en realidad estamos haciendo es comprobar si ambos objetos hacen referencia a la misma zona de memoria. Por lo tanto, podría darse el caso de que dos puntos exactamente iguales, al tratarse de objetos distintos, no fuieran considerados iguales por este operador. Aquí es donde entra en juego el método **equals**, permitiendo comparar el objeto sobre el cual se llama al método con un objeto geométrico de cualquier tipo que se le pase como parámetro, comparando únicamente los aspectos geométricos que nos interesan. Para ver de forma más clara esto, observemos el siguiente ejemplo:

Ejemplo.java

```
import JavaRG.Nucleo2D.*;

public class Equals {
    public Equals() {
    }

    public static void main(String args()) {
        Punto2D punto1 = new Punto2D(10, 10);
        Punto2D punto2 = new Punto2D(punto1);
        Segmento2D segmento = new Segmento2D();

        System.out.println(punto1 == punto2);
        System.out.println(punto1.equals(punto2));
        System.out.println(punto1.equals(segmento));
    }
}
```

Los objetos *punto1* y *punto2* son objetos distintos, en el sentido de que están almacenados en distintas zonas de memoria. Sin embargo, geométricamente hablando, ambos puntos son iguales, porque sus coordenadas son exactamente las mismas. Es por ello que el operador == nos dirá que no son iguales, mientras que equals sí. Igual que pasamos como parámetro a equals un punto, podemos pasar por ejemplo un segmento. En este caso la comparación es inmediata, ya que equals determina que se trata de objetos geométricos distintos (ademas de ser objetos distintos o instancias de distintas clases). La salida que produce este ejemplo es:

```
false
true
false
```

Otro aspecto tratado por la librería es el de las *transformaciones afines*. Cada objeto geométrico tendrá operaciones de *traslación*, *escalado* y *giro* (respecto al origen de coordenadas; si queremos realizar un giro respecto al centro del objeto, deberemos trasladarlo al origen, realizar el giro, y volver a trasladarlo a su posición original), así como un método especial, *transforma*, que nos permitirá aplicar cualquier combinación de transformaciones al objeto mediante una matriz bidimensional pasada como parámetro. Todas estas operaciones modificarán el objeto, pero además nos devolverán el propio objeto. Por ejemplo, en el siguiente ejemplo de transformaciones, el resultado obtenido tras cada una es mostrado por pantalla directamente:

EjemploTransformaciones.java

```
import JavaRG.Nucleo2D.*;
import JavaRG.Mat;

public class EjemploTransformaciones {
    public EjemploTransformaciones() {
    }

    public static void main (String []args) throws GeomException {
        Punto2D comienzo = new Punto2D(0,0);
        Punto2D fin = new Punto2D(3,3);
        Segmento2D segmento = new Segmento2D(comienzo, fin);
        System.out.println("Segmento inicial:");
        System.out.println(segmento);
        System.out.println("----------");

        // Realizamos una traslación
        System.out.println("Traslacion en el eje x tx=2 y en el eje y ty=-2");
        System.out.println(segmento.trasladar(2,-2));
        System.out.println("----------");

        // Giramos el segmento tomando como centro de giro el centro
        // de gravedad del propio segmento
        Punto2D centroSegmento = segmento.centroGravedad();
        segmento.trasladar(-centroSegmento.x(), -centroSegmento.y());
        segmento.gira(Mat.grad2Rad(90));
        System.out.println("Giro de 90 grados");
        System.out.println(segmento.trasladar(centroSegmento.x(), centroSegmento.y()));
        System.out.println("----------");

        // Aplicamos una transformacion de espejo creando para ello la
        // matriz de transformacion adecuada
        double [][]matrizTransformacion = new double[2][2];
        matrizTransformacion[0][0] = -1;
        matrizTransformacion[0][1] = 0;
        matrizTransformacion[1][0] = 0;
        matrizTransformacion[1][1] = -1;
```

```
            System.out.println("Transformacion de espejo");
            System.out.println(segmento.transforma(matrizTransformacion));
        }
    }
```

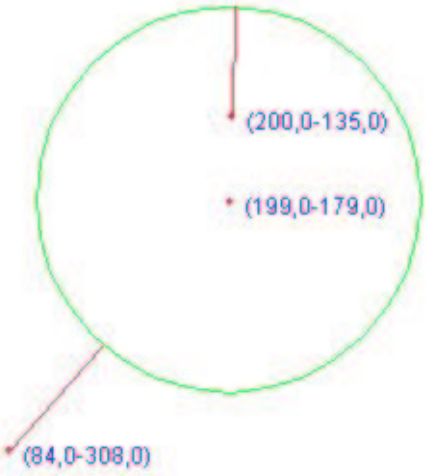
La primera transformación utilizada es una traslación, que en el caso del segmento se aplica directamente a sus extremos. Lo siguiente que hacemos es un giro respecto al propio segmento, para obtener un segmento perpendicular al segmento recien trasladado. Para ello hacemos uso del método *centroGravedad*, que nos devuelve el punto central del segmento. Al trasladar el centro del segmento al origen de coordenadas y hacer el giro respecto a dicho origen de coordenadas, es como si estuvieramos realizando el giro respecto al punto central del segmento. Finalmente deberemos devolver el segmento a su posición original. La última transformación que empleamos es una transformación especular respecto a los ejes *x* e *y*. Para ello debemos crear una matriz de transformación con valores −1 en la diagonal y ceros en el resto. La matriz que se pase como parámetro al método *transforma* deberá ser cuadrada, y de orden dos o tres, pues en caso contrario se producirá una excepción geométrica. Si es de orden dos, se aplicará a las coordenadas cartesianas de los puntos que forman el objeto geométrico, y si es de orden tres, a las coordenadas homogéneas.

Siguiendo con las características de JavaRG, un añadido interesante es el de las **distancias.** La librería CGAL solo incluye el concepto de distancia entre puntos. Sin embargo, JavaRG va más allá y define la operación de distancia de un punto a *cualquier* objeto geométrico. Un ejemplo de utilidad de esto se encuentra en las aplicaciones gráficas, en la selección de figuras: seleccionaremos aquel objeto geométrico más cercano al punto en el que hacemos click con el ratón. Las distancias son calculadas de la siguiente forma:

- **Segmento2D**: se obtiene la recta perpendicular al segmento que contenga al punto implicado en el cálculo de la distancia, obteniéndose la distancia de dicho punto al punto de corte de la recta con el segmento. Si no hay punto de corte, se obtendrá la menor distancia del punto a uno de los dos extremos del segmento.

- **Rayo2D**: si la recta perpendicular al rayo que pasa por el punto del que se desea conocer la distancia al rayo no intersecta con el mismo, la distancia será la que haya entre dicho punto y el comienzo del rayo; en caso contrario será la existente entre dicho punto y el punto de intersección de la recta con el rayo.
- **Recta2D**: la distancia de un punto a una recta se define como la distancia de dicho punto al punto de intersección de la recta con su perpendicular que contenga al primer punto.
- **Triangulo2D, Poligono2D, Rectangulo2D:** la mínima distancia a uno de sus lados, calculada de igual forma a como se obtiene con el segmento. Si el polígono solo contiene un vértice, la distancia será la existente entre el punto y el vértice del polígono.

- **Circulo2D:** se define como la distancia del punto a la circunferencia, y se calcula como el valor absoluto de la distancia del punto al centro del círculo menos su radio.



Podemos acceder a esta funcionalidad mediante el empleo del método *distancia(Punto2D)*, presente en todos los objetos geométricos (así como en el polígono), ya que dicho método se encuentra como método abstracto en la clase *ObjetoGeometrico.*

Por último destacar que todos los objetos geométricos del paquete *Nucleo2D***,** excepto *Caja2D*, *Vector2D* y *Direccion2D* incorporan una serie de métodos de dibujo que nos permitirán crear aplicaciones gráficas de forma sencilla. Para más información consultar la última sección del tutorial, donde además se podrá ver un ejemplo.

2.5.Uso del paquete Basica. Poligono2D

Este paquete es el encargado de recopilar todos aquellos objetos geométricos más complejos y estructuras de datos que hacen uso de los objetos geométricos del núcleo. De momento solo se dispone de la clase **Poligono2D**, así como de la clase abstracta **ObjetoBasico**. Todas las clases del paquete

Basica deberían ser subclases de esta clase, al igual que en el paquete *Nucleo2D* todos los objetos geométricos son subclases de *ObjetoGeometrico*.
La particularidad que ofrecen los objetos que dependen de esta clase abstracta es la implementación de la interfaz correspondiente a una estructura especial denominada *Circuladores* (consultar la siguiente sección). Obviamente, cualquier *ObjetoBasico* será también un *ObjetoGeometrico*, y como consecuencia, un polígono también lo será.

Centrándonos en la clase **Poligono2D**, hacemos un resumen de sus características:

- Un polígono perteneciente a esta clase se define como una cadena ordenada de vértices (objetos de la clase *Punto2D*) definida en un sentido horario o antihorario.
- Como cualquier clase perteneciente al paquete, implementa la interfaz de los circuladores, lo cual implica que podremos recorrer sus vértices de forma cíclica cómodamente.
- Se ha definido como un *polígono dinámico*, lo cual quiere decir que una vez creado el polígono, y al contrario que otros objetos como triángulos o rectángulos, será posible introducir nuevos vértices, e incluso modificar o eliminar los ya existentes de forma individual.

Podremos crear un polígono de tres formas distintas:

- Como copia de otro polígono mediante un constructor de copia, que acepta como parámetro un objeto del tipo **Poligono2D**.
- Utilizando el constructor por defecto, que no acepta ningún parámetro. Para ir construyendo el polígono deberemos ir insertando vértices una vez creado. El polígono será abierto hasta que introduzcamos el mismo vértice dos veces seguidas, momento en que se transformará en un polígono cerrado. Se considera entonces que además de existir una arista entre cada par de vértices existe también entre el primero y el último (que aunque se haya tenido que introducir dos veces, solo aparecerá una vez). A partir de ese momento no se podrán introducir más vértices.
- Utilizando un constructor especial, que permite crear polígonos al azar, tan solo especificando el número de lados que deseamos que tenga el polígono, así como la altura y la anchura del mismo.

El siguiente ejemplo ilustra estos tres casos:

CrearPoligonos.java

```
import JavaRG.Nucleo2D.*;
import JavaRG.Basica.*;

public class CrearPoligonos {
    public CrearPoligonos() {
    }

    public static void main (String []args) throws GeomException {
        // Creacion de un poligono al azar de 10 lados, con altura 100 y anchura 100)
        Poligono2D pol1 = new Poligono2D(10, 100, 100);
        pol1.cambiarNombre("Poligono 1");
        System.out.println(pol1);
        // Creacion de un poligono a partir de otro polígono
        Poligono2D pol2 = new Poligono2D(pol1);
        pol2.cambiarNombre("Poligono 2");
        System.out.println(pol2);
        // Creacion de un poligono a partir de sus vertices
        Poligono2D pol3 = new Poligono2D();
```

```
            pol3.cambiarNombre("Poligono 3");
            // Introducimos los vertices que conformaran el poligono
            pol3.insertar(new Punto2D(10.0, 0.5));
            pol3.insertar(new Punto2D(12.0, 8.0));
            pol3.insertar(new Punto2D(11.2, 3.7));
            // Hasta aquí el polígono pol3 es un polígono abierto (no completo)
            System.out.println(pol3);
            System.out.println(pol3.esCompleto());
            // Hacemos que el polígono sea un polígono cerrado (completo) insertando
            // de nuevo el ultimo vertice introducido
            pol3.insertar(new Punto2D(11.2, 3.7));
            System.out.println(pol3); // se observa que este ultimo vertice NO
                                      // aparece dos veces
            System.out.println(pol3.esCompleto());
        }

    }
```

Conforme introducimos vértices se comprueban las condiciones que hacen que un polígono sea simple (un polígono es simple si ningún par de lados del mismo intersecta y si ningún vértice es compartido por más de dos lados). Este efecto lo podemos ver en el siguiente ejemplo, en el que se comprueba que la condición de simplicidad del polígono cambia dinámicamente al insertar el cuarto vértice:

PoligonoSimple.java

```
    import JavaRG.Nucleo2D.*;
    import JavaRG.Basica.*;

    public class PoligonoSimple {

        Poligono2D poligono;

        public PoligonoSimple () {
            poligono = new Poligono2D();
        }

        public void escribeDatos() {
            if (poligono != null)
            {
                System.out.print("esSimple –> " + poligono.esSimple());
                System.out.println(" – esCompleto –> " + poligono.esCompleto());
            }
        }

        public void insertar(Punto2D punto) {
            poligono.insertar(punto);
        }

        public static void main (String []args) {
            PoligonoSimple polSim = new PoligonoSimple();

            polSim.insertar(new Punto2D(0,0));
            polSim.escribeDatos();
            polSim.insertar(new Punto2D(1,1));
            polSim.escribeDatos();
            polSim.insertar(new Punto2D(1,0));
            polSim.escribeDatos();
            polSim.insertar(new Punto2D(0,1));
            polSim.escribeDatos();
            polSim.insertar(new Punto2D(0,1)); // Cerramos el poligono
            polSim.escribeDatos();
        }
    }
```

El resultado de la ejecución será el siguiente:

```
esSimple –> true – esCompleto –> false
esSimple –> true – esCompleto –> false
esSimple –> true – esCompleto –> false
esSimple –> false – esCompleto –> false
esSimple –> false – esCompleto –> true
```

El polígono deja de ser simple cuando introducimos el vértice (0,1), ya que la arista definida entre este vértice y el vértice (1,0) intersectaría con la arista definida entre los vértices (0,0) y (1,1).

Al eliminar o modificar vértices (esto se hace de forma individual, tal como se ha comentado anteriormente) también se comprueban las condiciones que hacen que el polígono sea o no simple. Por ejemplo, en el siguiente código creamos un polígono de cuatro vértices que no es simple, pues intersectan la primera y la última arista del mismo. Al eliminar el último vértice, la última arista desaparece, y el polígono pasa a ser simple de nuevo:

PoligonoSimple2.java

```
import JavaRG.Nucleo2D.*;
import JavaRG.Basica.*;

public class PoligonoSimple2 {

    Poligono2D poligono;

    public PoligonoSimple2 () {
        poligono = new Poligono2D();
    }

    public void escribeDatos() {
        if (poligono != null)
        {
            System.out.print("esSimple –> " + poligono.esSimple());
            System.out.println(" – esCompleto –> " + poligono.esCompleto());
        }
    }

    public void insertar(Punto2D punto) {
        poligono.insertar(punto);
    }

    public void eliminar (int vertice) {
        try {
            poligono.eliminar(vertice);
        } catch (GeomException ge) {
            System.out.println(ge);
        }
    }

    public static void main (String []args) {
        PoligonoSimple2 polSim = new PoligonoSimple2();

        polSim.insertar(new Punto2D(0,0));
        polSim.escribeDatos();
        polSim.insertar(new Punto2D(1,1));
        polSim.escribeDatos();
        polSim.insertar(new Punto2D(1,0));
        polSim.escribeDatos();
        polSim.insertar(new Punto2D(0,1));
        polSim.escribeDatos();
```

```
            // Borramos el vértice recien introducido y comprobamos
            // si es simple de nuevo
            polSim.eliminar(3);
            polSim.escribeDatos();
        }
    }
```

Lo explicado anteriormente se puede comprobar observando la salida del programa:

```
esSimple -> true – esCompleto -> false
esSimple -> true – esCompleto -> false
esSimple -> true – esCompleto -> false
esSimple -> false – esCompleto -> false
esSimple -> true – esCompleto -> false
```

Hasta que no se introduce el cuarto vértice, el polígono es simple. Una vez este vértice es eliminado, no hay ninguna arista que interseccione, por lo que vuelve a ser simple. Al actualizar el polígono se vuelven a determinar las condiciones de simplicidad de forma transparente.

La otra característica peculiar de los polígonos, los circuladores, se explicará en el siguiente apartado.

2.6.Uso del paquete Soporte. Los Circuladores

La parte de soporte de la librería tan solo dispone actualmente de la implementación de una estructura iteradora denominada **Circulador**. Un iterador nos permite recorrer todas las posiciones de una determinada secuencia lineal (listas enlazadas, vectores, etc.). Sin embargo, en las estructuras geométricas son naturales las secuencias circulares, y aquí es donde entran en juego los circuladores, que son una implementación particular de los iteradores; se distinguen de éstos en que no tienen una posición final en la secuencia.
En JavaRG se han definido tres tipos de circuladores (como valores constantes dentro de la clase *InterfazCirculador*):

- cUNIDIRECCIONAL: solo permiten avanzar hacia una posición inmediatamente posterior o inmediatamente anterior.
- cBIDIRECCIONAL: permiten avanzar en ambas direcciones.
- cALEATORIO: circulador de acceso aleatorio que permite saltar a cualquier posición dentro de la secuencia circular.

Además de los tipos de circulador, hay dos conceptos a tener en cuenta:

- Alcanzable: un circulador **d** es alcanzable desde otro circulador **c** si **c** puede hacerse igual a **d** en un número finito de operaciones de avance simple. Al estar asociados los circuladores a estructuras circulares (en las que el final de la secuencia está conectado con el principio de la misma), esto siempre será posible si **c** y **d** pertenecen a la *misma estructura* (son vértices de un mismo polígono, o caras de un mismo mapa planar).
- Rango: Dados dos circuladores **c** y **d**, el rango **[c,d]** denota todos los circuladores obtenidos cuando comenzando desde **c** avanzamos posiciones simples hasta que **d** es alcanzado, sin incluir a **d**, para **d!=c**. El rango **[c,c]** denota, por lo tanto, todos los elementos de la secuencia circular.

Ambos conceptos están también implementados en JavaRG. Por lo tanto, un **Circulador** es un objeto iterador que se caracteriza por:

- un tipo de circulador (entre los tres indicados anteriormente)
- un objeto geométrico asociado, sobre la estructura del mismo se iterará
- una posición dentro de la secuencia del objeto

al crear un objeto de tipo Circulador deberemos proporcionar estos tres datos. Los objetos que se pueden asociar a un Circulador son los pertenecientes al paquete *Basica*, pues todos ellos derivan de *ObjetoBasico*, que implementa la interfaz *InterfazCirculador***.** La utilidad de esta interfaz es que un objeto geométrico nos pueda decir cuales son las posiciones siguiente y anterior a una dada, así como el número de posiciones de las que dispone. Por ejemplo, un objeto de la clase *Poligono2D* debería ser capaz de indicarnos cuantos vértices tiene (número de posiciones de la secuencia. Un polígono está definido como una secuencia circular de vértices, aunque éste no esté cerrado), y cual es el vértice posterior y anterior a uno dado.
La clase *InterfazCirculador* proporcionará los métodos básicos a partir de los cuales definimos todas las funcionalidades del *Circulador*.
Un detalle importante de los circuladores es que cada vez que cambiemos de posición en la secuencia, estaremos obteniendo un *circulador distinto*. Si disponemos de un circulador *c* asociado a un determinado objeto geométrico en la posición *i*, si avanzamos a la posición *j* obtendremos un circulador *d* distinto al anterior (aunque nada impide asignar *d* a *c*, igual que cuando aplicamos transformaciones a objetos geométricos simples podemos asignar el resultado de la operación a la misma referencia sobre la cual se hizo la llamada al método de transformación).
El siguiente ejemplo nos muestra como podemos recorrer todos los vértices de un polígono en ambos sentidos mediante un circulador bidireccional:

EjemploCirculador.java

```
import JavaRG.Nucleo2D.*;
import JavaRG.Basica.*;
import JavaRG.Soporte.*;

public class EjemploCirculador {
    public EjemploCirculador() {
    }

    public static void main (String ()args) throws GeomException {
        // Creamos el poligono para el cual deseamos recorrer
        // sus vertices
        Poligono2D poligono = new Poligono2D();
        poligono.insertar(new Punto2D(0,0));
        poligono.insertar(new Punto2D(1,0));
        poligono.insertar(new Punto2D(2,2));
        poligono.insertar(new Punto2D(1,3));
        poligono.insertar(new Punto2D(1,3)); // Cerramos el polígono

        // Creamos el circulador inicial, de tipo bidireccional,
        // asociado al primer vertice del polígono (posición 0)
        Circulador c = new Circulador(poligono, InterfazCirculador.cBIDIRECCIONAL, 0);

                System.out.println("Numero de circuladores alcanzables desde el primer vertice: " +
c.tamanyo());
        System.out.println("----");

        // Avanzamos por la secuencia en un sentido
        System.out.println("Avanzamos");
        System.out.println("----");
        for (int i=0; i<=c.tamanyo(); i++)
        {
            try {
```

```
                Punto2D vertice = poligono.vertice(c.posicion());
                System.out.println(vertice);
            } catch (GeomException ge) {
                // Precisamente debido al uso del circulador esta excepción
                // nunca se va a producir, pues nunca indicaremos un vértice
                // fuera del rango de vértices del polígono
                System.out.println(ge);
            }

            c = c.avanzar();
        }

        System.out.println("Retrocedemos");
        System.out.println("----");
        // y avanzamos en el otro sentido
        for (int i=0; i<=c.tamanyo(); i++)
        {
            try {
                Punto2D vertice = poligono.vertice(c.posicion());
                System.out.println(vertice);
            } catch (GeomException ge) {
                // Precisamente debido al uso del circulador esta excepción
                // nunca se va a producir, pues nunca indicaremos un vértice
                // fuera del rango de vértices del polígono
                System.out.println(ge);
            }

            try {
                c = c.retroceder();
            } catch (GeomException ge) {
                // No se va a producir nunca ya que el circulador no es de tipo
                // unidireccional
                System.out.println(ge);
            }
        }
    }
}
```

La salida sería la siguiente:

```
Numero de circuladores alcanzables desde el primer vertice: 3
----
Avanzamos
----
 - Punto2D: (x=0.0,y=0.0)
 - Punto2D: (x=1.0,y=0.0)
 - Punto2D: (x=2.0,y=2.0)
 - Punto2D: (x=1.0,y=3.0)
Retrocedemos
----
 - Punto2D: (x=0.0,y=0.0)
 - Punto2D: (x=1.0,y=3.0)
 - Punto2D: (x=2.0,y=2.0)
 - Punto2D: (x=1.0,y=0.0)
```

Tal como se puede observar, al utilizar el método *avanzar()* del circulador cuando nos encontramos en la última posición (el último vértice), obtendremos un nuevo circulador asociado a la primera posición del polígono (el primer vértice). De la misma forma, al usar *retroceder()* sobre un circulador asociado a la primera posición, obtendremos otro apuntando a la última. Aquí es donde se puede observar esa diferencia que se comentaba con el iterador, y es que no se considera un final en la secuncia de vértices, sino que se toma como una secuencia circular.

El circulador bidireccional nos ha permitido el uso de los métodos *avanzar* y *retroceder*, pero de haber utilizado *avanzarAleatorio* se hubiera producido una excepción geométrica, pues esta operación solo está reservada a circuladores de tipo aleatorio. Con respecto a los otros tipos, el circulador *unidireccional* solo permite el uso del método *avanzar*, produciéndose una excepción si intentamos utilizar los métodos *retroceder* y *avanzarAleatorio*, y un circulador de tipo *aleatorio* permite usar los tres métodos de avance: *avanzar, troceder* y *avanceAleatorio*. Este último método permite acceder a cualquier posición de la secuencia, pasando como parámetro un entero que indicará el número de posiciones a avanzar en un solo paso. Por ejemplo:

EjemploAleatorio.java

```
import JavaRG.Nucleo2D.*;
import JavaRG.Basica.*;
import JavaRG.Soporte.*;

public class EjemploAleatorio {
    public EjemploAleatorio() {
    }

    public static void main (String ()args) throws GeomException {
        // Creamos el poligono para el cual deseamos recorrer
        // sus vertices
        Poligono2D poligono = new Poligono2D();
        poligono.insertar(new Punto2D(0,0));
        poligono.insertar(new Punto2D(1,0));
        poligono.insertar(new Punto2D(2,2));
        poligono.insertar(new Punto2D(1,3));
        poligono.insertar(new Punto2D(1,3)); // Cerramos el polígono

        // Creamos el circulador inicial, de tipo bidireccional,
        // asociado al primer vertice del polígono (posición 0)
        Circulador c = new Circulador(poligono, InterfazCirculador.cALEATORIO, 0);


        // Avanzamos por la secuencia de tres en tres posiciones
        for (int i=0; i<=5; i++)
        {
            try {
                Punto2D vertice = poligono.vertice(c.posicion());
                System.out.println(vertice);
            } catch (GeomException ge) {
                // Precisamente debido al uso del circulador esta excepción
                // nunca se va a producir, pues nunca indicaremos un vértice
                // fuera del rango de vértices del polígono
                System.out.println(ge);
            }

            c = c.avanzarAleatorio(3);
        }

    }
```

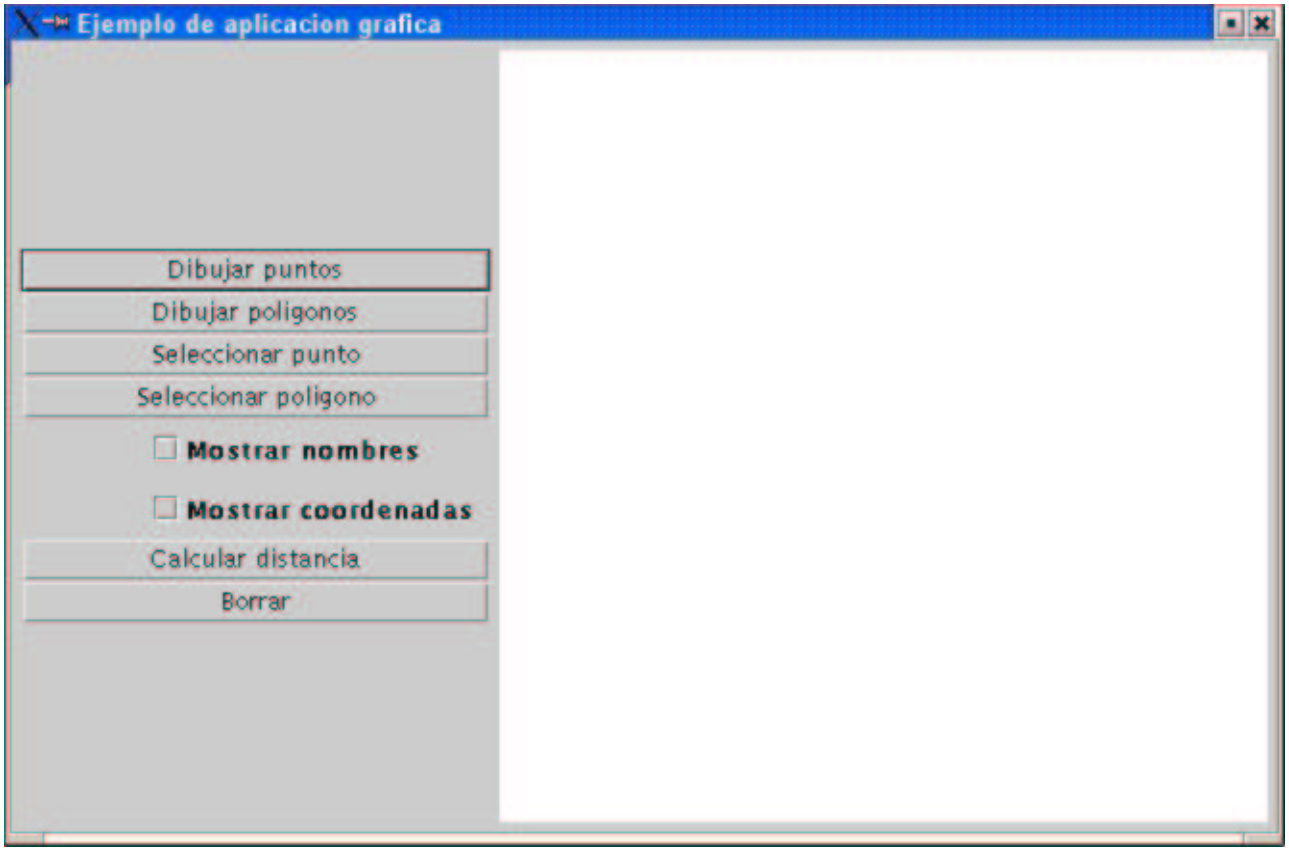
2.7.Características gráficas de JavaRG. Un ejemplo gráfico

Todos los objetos geométricos del paquete *Nucleo2D* (excepto *Caja2D*), así como el *Poligono2D*, incorporan métodos de dibujo de tal forma que podamos utilizarlos en nuestras aplicaciones gráficas. El presente apartado explicará como realizar paso por paso una aplicación Swing empezando desde cero, que pueda ser tomada como base para nuestras futuras aplicaciones.

En concreto el ejemplo nos permitirá determinar la distancia de puntos a polígonos. Podremos dibujar puntos individuales o polígonos a partir de sus vértices. Para determinar la distancia de un punto a un polígono, seleccionaremos el polígono, el punto, y seleccionaremos la opción adecuada. Los resultados se mostrarán además de forma gráfica, y se nos permitirá elegir entre ocultar o mostrar tanto el nombre de los polígonos y los puntos así como las coordenadas de los objetos.

2.7.1.Creando la interfaz gráfica

El aspecto del programa es el siguiente:



En la parte izquierda tenemos una serie de botones que nos permitirán seleccionar el tipo de acción a realizar como la información a mostrar. En la parte derecha tenemos la zona de dibujo, donde dibujaremos los objetos geométricos y mostraremos los resultados.
El código de la aplicación ha quedado dividido en dos partes. Una primera clase sería *EjemploCanvas*, que se corresponde con la zona de dibujo anteriormente citada. Esta clase contendrá la información de los objetos geométricos dibujados y una serie de métodos para añadirlos, dibujarlos, seleccionarlos, mostrar resultados, etc. El esqueleto de dicha clase, que iremos ampliando en siguientes secciones, sería el siguiente:

EjemploCanvas.java

```
import JavaRG.Nucleo2D.*;
import JavaRG.Interfaz.*;
import JavaRG.Basica.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class EjemploCanvas extends JComponent {

	private static int tamX = 400;
	private static int tamY = 400;

	public EjemploCanvas() {
		super();
		// Cambiamos el tamaño
		setPreferredSize(new Dimension(tamX, tamY));

	}

	public void paint(Graphics g) {
		// Cambiamos el color de fondo
		Graphics2D g2 = (Graphics2D) g;
		g.setColor(Color.white);
		g.fillRect(0, 0, getSize().width, getSize().height);
	}
}
```

La otra clase creada es *EjemploGrafico*, encargada de la interacción del usuario con los controles de la aplicación, y de llamar a los métodos adecuados de *EjemploCanvas* según lo que se desee realizar. El esqueleto de esta clase (que iremos completando) se muestra, comentado, a continuación:

EjemploGrafico.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;


public class EjemploGrafico extends JFrame {

	// Zona de dibujo
	EjemploCanvas canvas = new EjemploCanvas();

	// Constructor de la clase
	public EjemploGrafico() {
		// Le ponemos titulo a la ventana
		super("Ejemplo de aplicacion grafica");
		// Evitamos que el usuario cambie el tamaño
		setResizable(false);
		// Capturamos el evento de finalizacion de la aplicacion
		// al cerrar la ventana principal
		this.addWindowListener(new WindowAdapter() {
			public void windowClosing(WindowEvent we)
			{
				System.exit(0);
			}
		});
		// Creamos el contenido de la ventana
		crearContenido();
	}

	// Metodo para crear el contenido de la ventana
	private void crearContenido() {
		// Barra de botones
		Container botones = Box.createVerticalBox();
		Button dibujarPuntos = new Button("Dibujar puntos");
		Button dibujarPoligonos = new Button("Dibujar poligonos");
		Button seleccionarPunto = new Button("Seleccionar punto");
		Button seleccionarPoligono = new Button("Seleccionar poligono");
		JCheckBox mostrarNombres = new JCheckBox("Mostrar nombres");
		JCheckBox mostrarCoordenadas = new JCheckBox("Mostrar coordenadas");
		Button distancia = new Button("distancia");
```

```
            botones.add(Box.createVerticalGlue());
            botones.add(dibujarPuntos);
            botones.add(Box.createVerticalGlue());
            botones.add(dibujarPoligonos);
            botones.add(Box.createVerticalGlue());
            botones.add(seleccionarPunto);
            botones.add(Box.createVerticalGlue());
            botones.add(seleccionarPoligono);
            botones.add(Box.createVerticalGlue());
            botones.add(mostrarNombres);
            botones.add(Box.createVerticalGlue());
            botones.add(mostrarCoordenadas);
            botones.add(Box.createVerticalGlue());
            botones.add(distancia);

            // Panel general, que contendra los botones a la izquierda y la zona
            // de dibujo a la derecha
            JPanel contenedor = new JPanel();
            contenedor.setLayout(new FlowLayout());
            contenedor.add(botones);
            contenedor.add(canvas);

            // Asginamos el panel general a la ventana principal
            this.setContentPane(contenedor);
        }

        // Metodo principal
        static public void main(String []args) {
            EjemploGrafico f = new EjemploGrafico();
            // Adaptamos la ventana al tamaño de los componentes que la conforman
            f.pack();
            // La mostramos
            f.setVisible(true);
        }
    }
```

No explicaremos en detalle el funcionamiento de Swing en Java, pues queda fuera del alcance del presente documento.
Las siguientes secciones explicarán que código deberemos ir añadiendo a estas clases para dotar de funcionalidad al ejemplo y conseguir lo que deseamos. Aunque tan solo iremos indicando el código que se deberá añadir, al final del tutorial se mostrará el código completo de ambas clases.

2.7.2.Dibujo de puntos

Tal como se ha indicado anteriormente, *EjemploCanvas* contendrá los objetos geométricos y ralizará las acciones en si mismas, mientras que *EjemploGrafico* se limitará a indicar a *EjemploCanvas* (mediante llamadas a los métodos de *miCanvas*, variable de tipo *EjemploCanvas* contenida en *ejemploGrafico*) las acciones a realizar.

Introducimos para ello una variable en ***EjemploCanvas***, llamada **accion**, que determina que es lo que pasará cuando el usuario pinche con el ratón sobre la zona de dibujo. Los posibles valores de esta variable serán definidos dentro de la clase *EjemploCanvas* como valores static constantes:

```
    public static final int NADA = 0; // Ninguna accion
    public static final int D_PUNTO = 1; // Dibujar puntos
    public static final int S_PUNTO = 2; // Seleccionar puntos
    public static final int D_POLIGONO = 3; // Dibujar polígono
    public static final int S_POLIGONO = 4; // Seleccionar polígono

    // Accion actual
```

```
private int accion = NADA;
```

También definimos dentro de ***EjemploCanvas*** el método ***cambiarAccion()*** que permita cambiar el valor de esta variable:

```
// Modifica la accion a realizar cuando se pincha sobre la zona
// de dibujo
public void cambiarAccion(int nuevaAccion) {
    accion = nuevaAccion;
}
```

Con esto ya le podemos decir a los cuatro primeros botones de ***EjemploGrafico*** que cambien la acción a realizar cuando son pulsados, añadiendo el siguiente código después de la creación de los mismos y antes de que sean introducidos en el panel de botones:

```
// Modificamos la acción a realizar al pinchar sobre la zona
// de dibujo cuando pulsamos alguno de los botones de dibujo
// o selección
dibujarPuntos.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        canvas.cambiarAccion(EjemploCanvas.D_PUNTO);
    }
});
dibujarPoligonos.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        canvas.cambiarAccion(EjemploCanvas.D_POLIGONO);
    }
});
seleccionarPunto.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        canvas.cambiarAccion(EjemploCanvas.S_PUNTO);
    }
});
seleccionarPoligono.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        canvas.cambiarAccion(EjemploCanvas.S_POLIGONO);
    }
});
```

Ya que ***EjemploCanvas*** va a contener la infomación sobre los objetos geométricos dibujados y la forma de mostrarlos, debemos crear dentro de dicha clase las siguientes variables:

```
// Numero maximo de objetos geometricos de cada tipo
private static int max = 100;

// Objetos geometricos dibujados
private Punto2D  puntos()= new Punto2D(max);

// Numero de objetos geometricos dibujados
private int numPuntos = 0;

// Opciones de visualizacion
private boolean mostrarCoordenadas = false;
private boolean mostrarNombres = false;
```

Para implementar el dibujo de puntos, dentro de ***EjemploCanvas*** debemos realizar dos modificaciones, una que nos permita introducir los puntos, y otra que los muestre por pantalla. Para introducir los puntos, creamos una clase privada llamada ***TicRaton***, que se encargará del manejo del evento de pinchar con el ratón en el interior de la zona de dibujo. Esta clase se encontrará dentro del archivo

EjemploCanvas.java, en el interior de la propia clase *EjemploCanvas*. Aunque de momento solo vamos a dibujar puntos, dejamos preparado su esqueleto para ir añadiendo funcionalidades más adelante. Así pues, esta clase sería la siguiente:

```
// CLASE PRIVADA PARA EL MANEJO DE EVENTOS DEL RATON
private class TicRaton extends MouseAdapter {
    public void mousePressed (MouseEvent e) {
        int x = e.getX();
        int y = −e.getY();

        switch(accion) {
            case D_PUNTO:
                if (numPuntos < max) {
                    puntos(numPuntos) = new Punto2D(x, y);
                    String nombre = "PUNTO" + (numPuntos+1);
                    puntos(numPuntos).cambiarNombre(nombre);
                    numPuntos++;
                    repaint();
                }
            break;
            case S_PUNTO:
            break;
            case D_POLIGONO:
            break;
            case S_POLIGONO:
            break;
            default:
                // No se hace nada
            break;
        }
    }
}
```

Como se puede ver, lo primero que se hace es obtener las coordenadas en el interior de la zona de dibujo donde se pinchó con el ratón. En el caso de la coordenada y debemos tomarla de signo contrario, pues en la zona de dibujo esta coordenada es negativa, encontrándose el origen de coordenadas (0,0) en la esquina superior izquierda.

Tras esto, y según la acción seleccionada, haremos una cosa u otra. De momento solo seremos capaces de crear puntos. Para ello, creamos un nuevo objeto de la clase *Punto2D* que introducimos en la posición correspondiente del array de puntos, y cuyas coordenadas serán las del punto donde se pulsó con el ratón. También le cambiamos el nombre, por medio de la función *cambiarNombre* de la clase *ObjetoGeometrico*. Por último forzamos el redibujado de la zona de dibujo para que se muestre el punto recién creado.

Ahora debemos asociar esta clase a los eventos de ratón de *EjemploCanvas*, añadiendo la siguiente línea en el constructor de la clase:

```
addMouseListener(new TicRaton());
```

Finalmente añadimos las siguientes líneas al método *dibujar* de *EjemploCanvas* para que se muestren los puntos introducidos:

```
// Dibujo de puntos
for (int i = 0; i < numPuntos; i++)
{
    puntos(i).dibujar(g, mostrarCoordenadas);
```

```
            if (mostrarNombres) puntos(i).dibujarNombre(g);
        }
```

Simplemente recorremos el array de puntos hasta el total de puntos dibujados, llamando al método *dibujar* para cada uno de sus correspondientes objetos de la clase *Punto2D*. Este método acepta dos parámetros, el contexto gráfico (obtenido a su vez como parámetro de *paint*), y una coordenada booleana que indica si se deben mostrar por pantalla las coordenadas del punto dibujado. También mostramos para cada punto, en caso de que el usuario lo desee, los nombres de los puntos, por medio del método *dibujarNombre*.

Tras estos cambios ya podremos dibujar puntos, pero no tendremos la posibilidad de mostrar sus coordenadas o nombres, pues no se ha definido ninguna forma de cambiar el valor de las variables booleanas *mostrarCoordenadas* y *mostrarNombres*. El primer paso será crear los dos métodos dentro de ***EjemploCanvas*** que permitirán hacer este cambio:

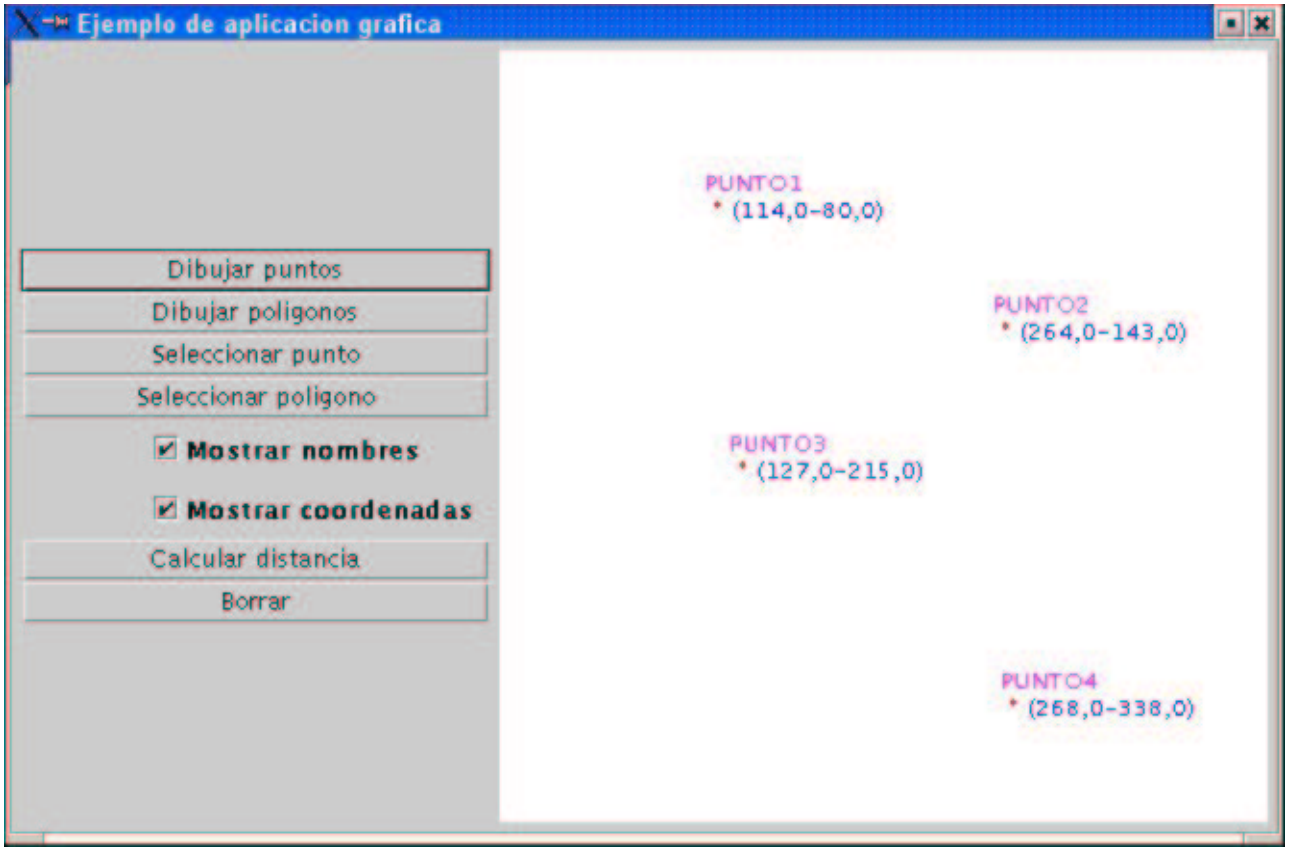
```
// Métodos para el cambio de las variables booleanas mostrarCoordenadas y
// mostrarNombres
public void cambiarCoordenadas(boolean valor) {mostrarCoordenadas = valor; repaint();}
public void cambiarNombres(boolean valor) {mostrarNombres = valor; repaint();}
```

Ahora solo hay que hacer que cuando se pulse sobre los checkboxes de la interfaz correspondientes a las opciones de *mostrar nombres* y *mostrar coordenadas* se llame a la función adecuada. Para ello, dentro de ***EjemploGrafico*** añadimos los siguiente, tras crear los checkboxes pero antes de introducirlos en el panel de botones:

```
mostrarNombres.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
                canvas.cambiarNombres(mostrarNombres.isSelected());
        }
});
mostrarCoordenadas.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
                canvas.cambiarCoordenadas(mostrarCoordenadas.isSelected());
        }
});
```

Al compilar el código anterior nos encontraremos con que *javac* nos dice que no se pueden utilizar las variables *mostrarNombres* y *mostrarCoordenadas* dentro de una clase interna, y que deben ser declaradas como finales. Pero no podemos declarar estas variables como finales, ya que en ese caso su valor no podría cambiar nunca. La solución es declararlas como variables globales de la clase, fuera de cualquier método de la misma.

Tras estos pasos habremos creado una aplicación que de momento nos permite dibujar puntos, y seleccionar si queremos que se muestren sus nombres y coordenadas.



2.7.3.Dibujo de polígonos

Realizamos los pasos necesarios para poder utilizar la opción de *dibujar polígonos.* Un polígono se dibujará a partir de sus vértices. Mientras pinchemos con el ratón en la zona de dibujo irán apareciendo los vértices y las aristas del polígono. Para terminar de dibujarlo, pincharemos dos veces en el mismo punto (con lo que el polígono quedará cerrado). Si durante el dibujo del polígono se pulsa cualquiera de los demás botones, el polígono que estuvieramos dibujando será borrado (a menos que ya hubieramos terminado de dibujarlo).

Lo primero es añadir en ***EjemploCanvas*** el array donde almacenaremos los polígonos introducidos, así como la variable que indicará cuántos (al igual que se hizo con el punto):

```
// Objetos geometricos dibujados
private Punto2D puntos() = new Punto2D(max);
private Poligono2D poligonos() = new Poligono2D(max);

// Numero de objetos geometricos dibujados
private int numPuntos = 0;
private int numPoligonos = 0;
```

También introducimos una variable donde almacenaremos de forma temporal el polígono que se está dibujando para que, cuando sea completado, se introduzca en el array de polígonos

```
// Polígono que se está dibujando
private Poligono2D poligonoTemporal = null;
```

Deberemos indicar en el método *paint* de ***EjemploCanvas*** que se dibujen los polígonos existentes, así

como su nombre en caso de que se haya seleccionado así (igual que se hace con los puntos), y el polígono temporal (en caso de que exista):

```
// Dibujo de poligonos
for (int i = 0; i < numPoligonos; i++)
{
	poligonos(i).dibujar(g, mostrarCoordenadas);
	if (mostrarNombres) poligonos(i).dibujarNombre(g);
}
if (poligonoTemporal != null) poligonoTemporal.dibujar(g, mostrarCoordenadas);
```

Para que la introducción de polígonos sea tal como se ha dicho, introducimois las siguientes líneas en el *case D_POLIGONO* del interior de la clase *TicRaton* que manejaba el evento de pulsado del botón del ratón sobre la zona de dibujo:

```
case D_POLIGONO:
	if (numPoligonos < max) {
		if (poligonoTemporal == null)
			poligonoTemporal = new Poligono2D();
		poligonoTemporal.insertar(new Punto2D(x, y));
		if (poligonoTemporal.esCompleto())
		{
			poligonos(numPoligonos) = new Poligono2D(poligonoTemporal);
			String nombre = "POLIG" + (numPoligonos+1);
			poligonos(numPoligonos).cambiarNombre(nombre);
			numPoligonos++;
			poligonoTemporal = null;
		}
		repaint();
	}
break;
```

El comportamiento es el siguiente: si no existe el polígono temporal, se crea. Los nuevos vértices que introduzcamos los insertaremos en dicho polígono temporal, de tal forma que aunque no se encuentre en el array de polígonos el polígono que estamos dibujando, se pueda mostrar en la zona de dibujo conforme lo creamos. El polígono temporal se incorporará al array de polígonos cuando lo cerremos, insertando dos veces el mismo vértice (momento en el cual el polígono temporal será completo). Almacenamos el nuevo polígono en la posición correspondiente, mediante el constructor de copia, y le cambiamos el nombre. Finalmente, hacemos que *poligonoTemporal* no referencie a ningún objeto, asignándole el valor *null*, de tal forma que si volvemos a pinchar con el ratón en la zona de dibujo, se generará un nuevo polígono temporal.
Con respecto al dibujo de polígonos tan solo queda que si se pulsa sobre cualquier otro botón que no sea *Dibujar polígonos* se elimine el polígono temporal. Primero creamos el método *eliminarPoligonoTemporal* dentro de ***EjemploCanvas***, que haga que la variable *poligonTemporal* deje de referenciar un objeto asignándole el valor null, y que fuerce el redibujado de la zona de dibujo para que dicho polígono temporal sea ’borrado’:

```
// Borra el polígono temporal
public void eliminarPoligonoTemporal() {
	poligonoTemporal = null;
	repaint();
}
```

El caso de los botones de selección y dibujo de polígonos y puntos lo podemos solucionar introduciendo una llamada a este método dentro de *cambiarAccion*, que quedaría de la siguiente forma:

```
public void cambiarAccion(int nuevaAccion) {
```

```
            accion = nuevaAccion;
            if (accion != D_POLIGONO) {
                poligonoTemporal = null;
                repaint();
            }
        }
```

Los botones de *Borrar* y *Calcular distancia* requieren que se introduzcan los *ActionListener* correspondientes (junto a los ya introducidos para los otros botones). Para el botón de cálculo de distancias:

```
distancia.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        canvas.eliminarPoligonoTemporal();
        canvas.cambiarAccion(EjemploCanvas.NADA);
    }
});
```

(este manejador será completado en la sección de dibujo de resultados); en el caso del botón de borrado aprovechamos para implementar su funcionalidad, de tal forma que al pulsarlo, lo que hacemos es llamar a un método *borrar* de *EjemploCanvas:*

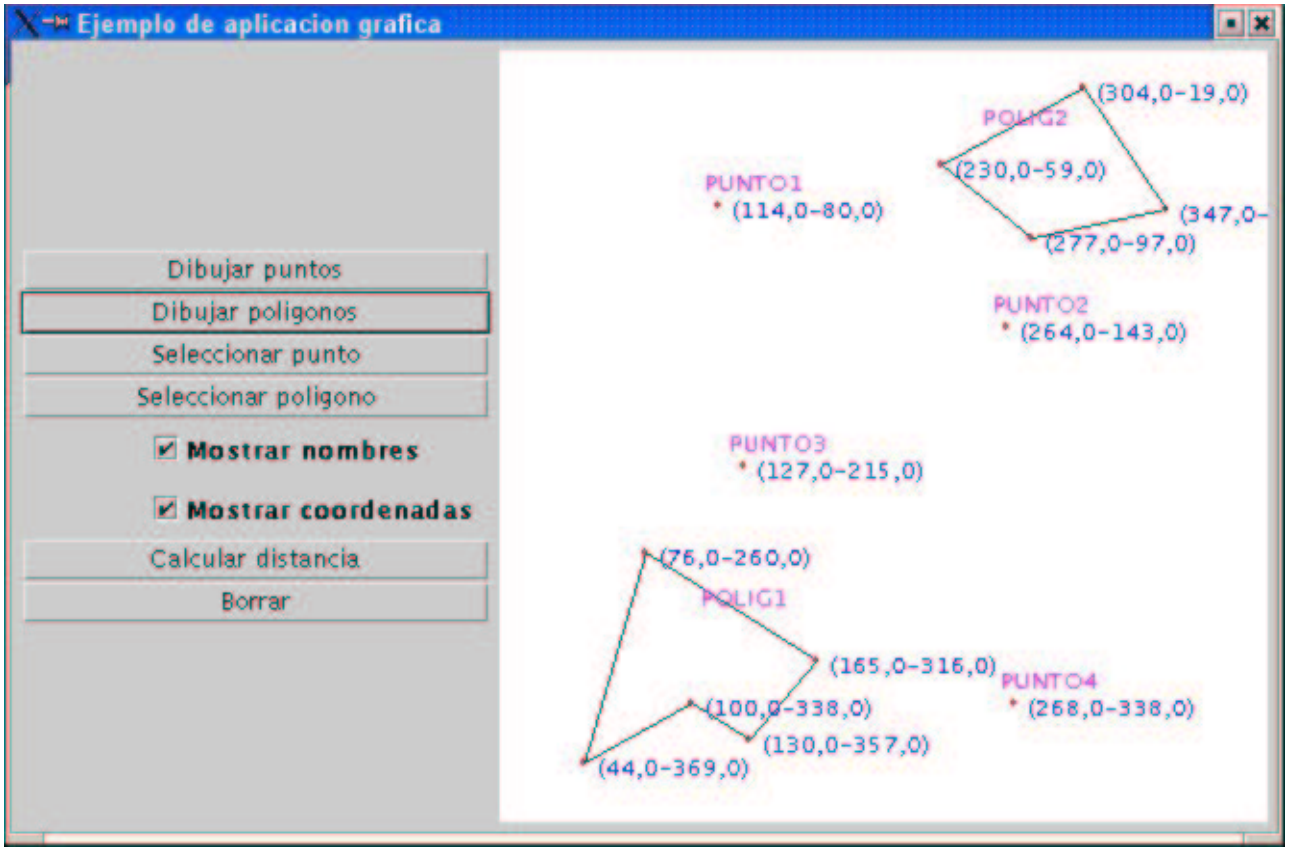
```
borrar.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        canvas.borrar();
        canvas.cambiarAccion(EjemploCanvas.NADA);
    }
});
```

El método *borrar* de ***EjemploCanvas*** tendría el siguiente aspecto:

```
// Borra los objetos geometricos dibujados
public void borrar() {
    numPuntos = 0;
    numPoligonos = 0;
    eliminarPoligonoTemporal();
}
```

Al igualar el número de puntos y de polígonos a cero, al redibujar no se mostrará ninguno. Además, los nuevos polígonos y puntos que se inserten lo harán reescribiendo los que ya existieran. No es necesario llamar a *repaint()* porque esto ya se hace en el interior de *eliminarPoligonoTemporal()*.

Ya podemos dibujar puntos y polígonos, así como limpiar la zona de dibujo. En el siguiente apartado veremos qué tenemos que hacer para que el usuario pueda seleccioar un punto y un polígono.



2.7.4.Selección de figuras

Cada objeto geométrico dispone de varios métodos de dibujo (todos ellos apoyándose en un método privado general para el dibujo, que recibe los parámetros adecuados según el método desde el que se le llame). Además del *dibujar* ya visto, disponemos de *dibujarSeleccionado*, que usaremos en esta sección para mostrar los objetos seleccionados por el usuario, *dibujarXor,* que puede ser utilizado en programas de dibujo para no tener que borrar todos los objetos mientras se modifica uno solo, y *dibujarResultado*, que muestra el objeto resaltado.
Ya que no lo vamos a usar en este ejemplo, comentaremos brevemente el método *dibujarXor.* Este método simplemente dibuja el objeto geométrico con todos sus elementos en blanco (incluyendo coordenadas). Esto permite actualizar un objeto que haya cambiado sin necesidad de borrar todos y volver a dibujarlos. Para ello, usamos *dibujarXor* con el objeto anterior y que ha sido modificado (con lo que se borra), y *dibujar* con el nuevo objeto. Esta técnica es empleada en programas de dibujo vectorial.

Para seleccionar figuras, debemos disponer de dos variables dentro de ***EjemploCanvas***, que almacenen el índice de la figura seleccionada (punto o polígono):

```
// Objetos seleccionados
private int puntoSeleccionado = -1;
private int poligonoSeleccionado = -1;
```

El valor inicial de estas variables (−1) indica que no hay ningún punto ni polígono seleccionado. En el momento en que se seleccione algún punto, se almacenará en *puntoSeleccionado* el índice de dicho punto en el array de puntos. El funcionamiento en el caso de los polígonos es identico. Mostraremos los objetos seleccionados añadiendo las siguientes líneas al final del método *paint* de ***EjemploCanvas***:

```
// Dibujo de objetos seleccionados
if (puntoSeleccionado != -1)
        puntos(puntoSeleccionado).dibujarSeleccionado(g, mostrarCoordenadas);
if (poligonoSeleccionado != -1)
        poligonos(poligonoSeleccionado).dibujarSeleccionado(g, mostrarCoordenadas);
```

Y para que no haya errores debemos introducir las siguientes líneas al comienzo del método *borrar* de ***EjemploCanvas***:

```
puntoSeleccionado = -1;
poligonoSeleccionado = -1;
```

Ya solo falta poder realizar la selección en sí misma, haciendo los cambios oportunos en el interior de la clase *TicRaton:*

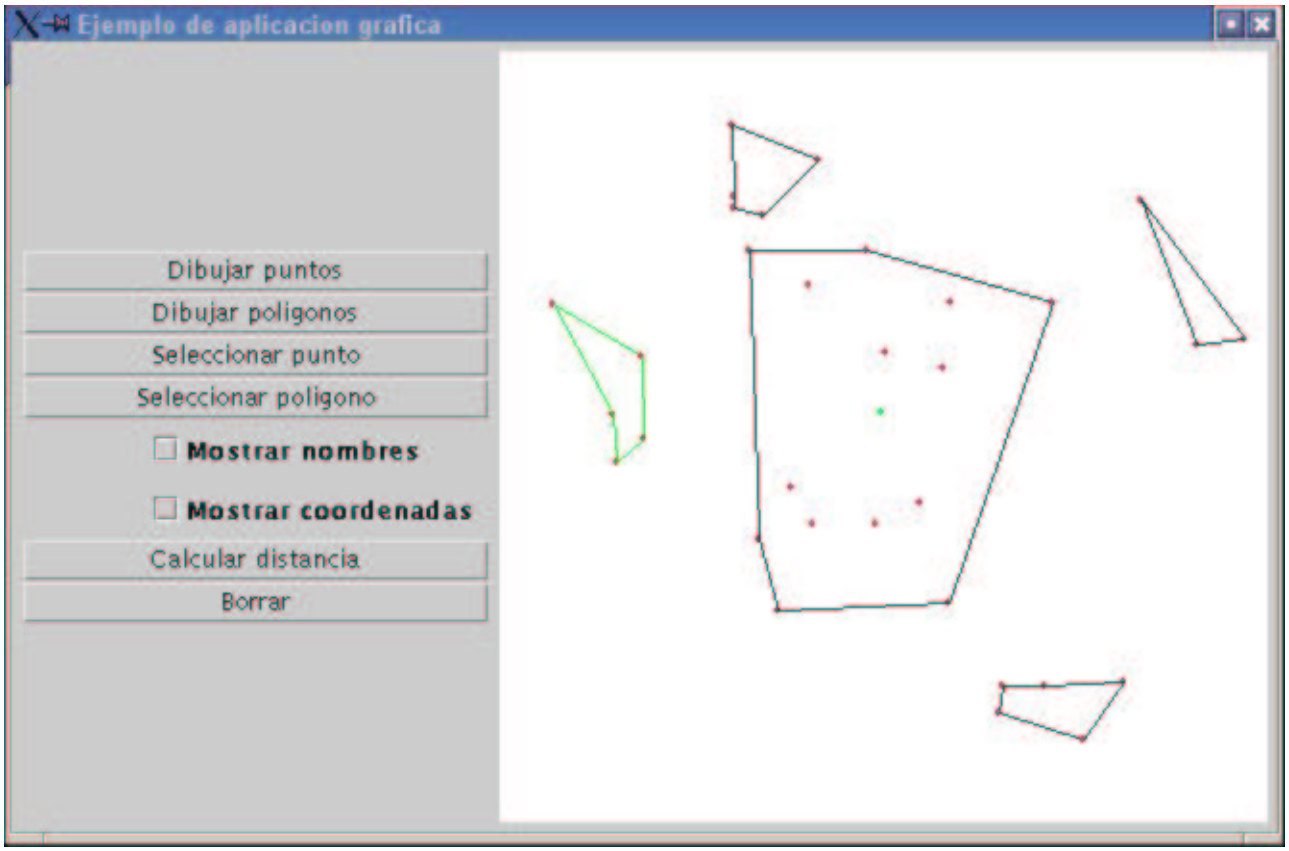
```
                        // Seleccion de puntos
                        case S_PUNTO:
                                double minimaDistancia = java.lang.Integer.MAX_VALUE;
                                Punto2D p = new Punto2D(x,y);
                                for (int i=0; i<numPuntos;i++)
                                {
                                        double distancia = puntos(i).distancia(p);
                                        // Determinamos el polígono más cercano al punto
                                        // en el que se pincho con el raton
                                        if (distancia < minimaDistancia) {
                                                minimaDistancia = distancia;
                                                puntoSeleccionado = i;
                                        }
                                }
                                // Solo seleccionamos una figura si su distancia al
                                // punto pinchado es menor de un umbral
                                if (minimaDistancia > umbral)
                                        puntoSeleccionado = -1;
                                repaint();
                        break;

.....

                        // Seleccion de poligonos
                        case S_POLIGONO:
                                double minDistancia = java.lang.Integer.MAX_VALUE;
                                Punto2D punto = new Punto2D(x,y);
                                for (int i=0; i<numPoligonos;i++)
                                {
                                        double distancia = poligonos(i).distancia(punto);
                                        // Determinamos el polígono más cercano al punto
                                        // en el que se pincho con el raton
                                        if (distancia < minDistancia) {
                                                minDistancia = distancia;
                                                poligonoSeleccionado = i;
                                        }
                                }
                                // Solo seleccionamos una figura si su distancia al
                                // punto pinchado es menor de un umbral
                                if (minDistancia > umbral)
                                        poligonoSeleccionado = -1;
                                repaint();
                        break;
                        default:
                                // No se hace nada
                        break;
```

Observemos por ejemplo el caso del polígono (para los puntos es exactamente el mismo comportamiento). Creamos un objeto de tipo *Punto2D* correspondiente al punto de la zona de dibujo donde se pinchó con el ratón, y que será pasado como parámetro a la función *distancia*. También inicializamos una variable llamada *minDistancia* con el máximo entero posible. Hecho esto recorremos todo el array de polígonos, determinando la distancia de cada uno al punto creado, y le daremos a la variable *poligonoSeleccionado*, que indica el índice del polígono que es seleccionado, el índice de aquel polígono cuya distancia al punto en el que pulsó con el ratón es mínima. Finalmente, determinamos si la distancia del polígono es menor que un determinado umbral, definido como constante en la clase *EjemploCanvas:*

```
public static final int umbral = 4;
```

Con esto conseguiremos dos cosas. Solo podremos selccionar objetos si están realmente cerca del punto donde se pinchó con el ratón en la zona de dibujo. Al asignar el valor −1 en caso contrario, se deseleccionará el objeto que estuviera seleccionado. El *repaint* es necesario tanto si se seleccionó un nuevo objeto como si no, ya que en caso de seleccionarlo deberemos mostrarlo destacado, y en caso de que no se seleccionara ninguno deberemos dibujar todos los polígonos de forma normal, por si se hubiera seleccionado uno antes (permitiendo realizar una 'deselección' de polígonos pulsando en cualquier punto de la zona de dibujo lo suficientemente alejado de cualquier polígono).

Nuestro pequeño ejemplo ya está casi completo. Podemos dibujar puntos y polígonos, y tener selccionados un punto y un polígono a la vez. También podemos borrar todos los objetos, mostrar sus nombres, y las coordenadas de los puntos y vértices. Tan solo queda que podamos determinar la distancia entre el punto y el polígono seleccionado, lo cual veremos como se hace en la siguiente sección.



2.7.5.Dibujo de resultados

La idea es que una vez el usuario haya seleccionado un punto y un polígono, pueda calcular la distancia del primero al segundo pulsando el botón *calcular distancia*, y que el resultado se muestre en forma gráfica como un segmento entre el punto seleccionado y el punto más cercano a éste dentro del polígono seleccionado.

Añadimos una nueva variable a ***EjemploCanvas*** que contenga el segmento resultante:

```
private Segmento2D resultado = null;
```

Añadimos de igual forma el código necesario al método *paint* de ***EjemploCanvas*** para que se dibuje dicho resultado en caso de existir. En este caso hacemos uso de *dibujarResultado*, que mostrará el segmento de forma distinta al resto de métodos de dibujo:

```
// Dibujo del resultado si existe
if (resultado != null)
{
      resultado.dibujarResultado(g);
      resultado.dibujarNombre(g);
}
```

El método *dibujarResultado* tan solo admite un parámetro, el contexto gráfico g. Nunca se dibujarán las coordenadas de los puntos de un resultado. También añadimos la siguiente línea tanto a *cambiarAccion* como a *borrar*, con el objetivo de que el resultado se elimine de pantalla si se pulsa cualquier botón que no sea el de *calcular distancia*:

```
resultado = null;
```

El último cambio que tenemos que hacer en ***EjemploCanvas*** es crear el propio método que calcule el resultado:

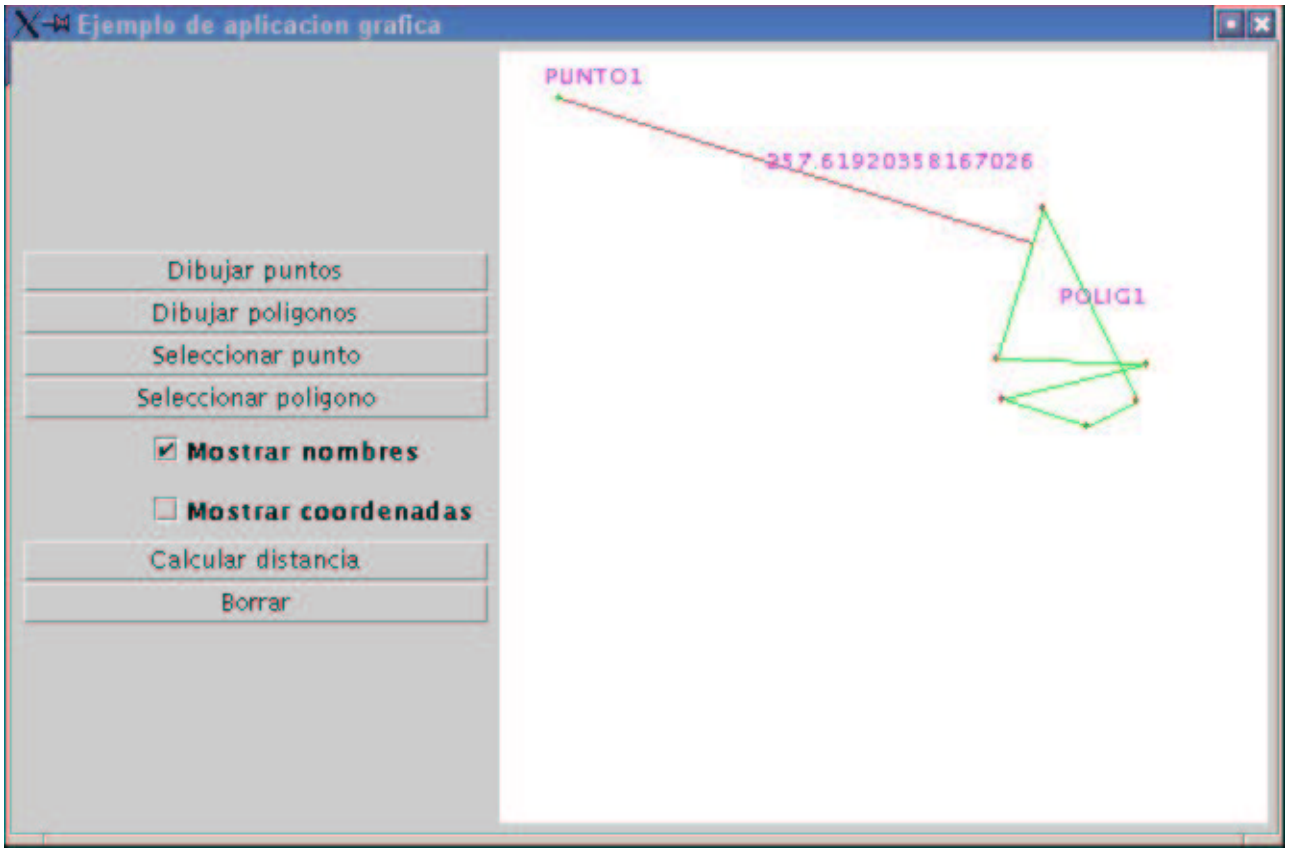
```
      // Calcula la distancia y crea el objeto geométrico 'resultado' para mostrar dicha
      // distancia gráficamente
      public void calcularDistancia() {
            // Primero comprobamos que haya tanto un punto como un polígono seleccionados
            if (poligonoSeleccionado == -1 || puntoSeleccionado == -1)
            {
                        JOptionPane.showMessageDialog(this, "Debe haber un punto y un poligono
seleccionados", "ERROR",JOptionPane.WARNING_MESSAGE);
            }
            else try {

                  // calculamos la distancia, así como el punto perteneciente al poligono
                  // seleccionado mas cercano al punto seleccionado
                  Punto2D puntoCercano = new Punto2D();
                    double distancia =
poligonos(poligonoSeleccionado).puntoMasCercano(puntos(puntoSeleccionado), puntoCercano);
                  // Creamos el segmento resultado entre los dos puntos anteriores, dandole
                  // como nombre la distancia entre ambos
                  resultado = new Segmento2D(puntos(puntoSeleccionado), puntoCercano);
                  resultado.cambiarNombre(String.valueOf(distancia));
                  // Mostramos el resultado por la salida estandar
                  System.out.println("Distancia de " + puntos(puntoSeleccionado).obtenerNombre() +
```

```
" a " + poligonos(poligonoSeleccionado).obtenerNombre() + " : " + distancia);
                    // y redibujamos la zona de dibujo para que se muestre el resultado
                    repaint();
                } catch (GeomException ge) {System.out.println(ge); }
        }
```

En caso de que no haya un polígono y un punto seleccionados (el valor de *poligonoSeleccionado* o *puntoSeleccionado* es −1) mostramos un mensaje de error. En caso contrario se realiza el proceso. Para el cálculo de la distancia se emplea la función *puntoMasCercano* en lugar de *distancia*. Mientras que la segunda tan solo devuelve el valor de la distancia, la primera permite además obtener el punto del objeto geométrico para el cual se llama el método que es más cercano al punto. Esta función devuelve la distancia, que almacenamos en una variable, y admite dos parámetros. El primero es el punto para el que se desea conocer su distancia al polígono. El segundo parámetro es un parámetro de salida. Debemos pasar un objeto de tipo *Punto2D* que ya haya sido inicializado (pues en caso contrario se lanzará una excepción), y tras ejecutarse la función, almacenará el punto del objeto geométrico (en este caso el polígono) más cercano al punto pasado como primer parámetro. Precisamente estos dos puntos son usados para construir el segmento *resultado*, al que se le asigna como nombre la distancia. Por último se muestra por salida estándar el resultado, y se redibuja la zona de dibujo para mostrar el segmento recien creado.

Solo queda un paso, y es incluir la siguiente línea al final del método *actionPerformed* del *ActionListener* asociado al botón d*istancia* de ***EjemploGrafico***:

```
canvas.calcularDistancia();
```



2.7.6.Código fuente completo

Por último mostramos el código de los dos archivos .java implicados en la creación de este ejemplo de forma íntegra:

EjemploGrafico.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.event.*;


public class EjemploGrafico extends JFrame {

    // Zona de dibujo
    EjemploCanvas canvas = new EjemploCanvas();

    // Variables de la interfaz que necesitan ser declaradas como globales
    JCheckBox mostrarNombres = new JCheckBox("Mostrar nombres");
    JCheckBox mostrarCoordenadas = new JCheckBox("Mostrar coordenadas");

    // Constructor de la clase
    public EjemploGrafico() {
        // Le ponemos titulo a la ventana
        super("Ejemplo de aplicacion grafica");
        // Evitamos que el usuario cambie el tamaño
        setResizable(false);
        // Capturamos el evento de finalizacion de la aplicacion
        // al cerrar la ventana principal
        this.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we)
            {
                System.exit(0);
            }
        });
        // Creamos el contenido de la ventana
        crearContenido();
    }

    // Metodo para crear el contenido de la ventana
    private void crearContenido() {
        // Barra de botones
        Container botones = Box.createVerticalBox();
        Button dibujarPuntos = new Button("Dibujar puntos");
        Button dibujarPoligonos = new Button("Dibujar poligonos");
        Button seleccionarPunto = new Button("Seleccionar punto");
        Button seleccionarPoligono = new Button("Seleccionar poligono");
        Button distancia = new Button("Calcular distancia");
        Button borrar = new Button("Borrar");

        // Modificamos la acción a realizar al pinchar sobre la zona
        // de dibujo cuando pulsamos alguno de los botones de dibujo
        // o selección
        dibujarPuntos.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                canvas.cambiarAccion(EjemploCanvas.D_PUNTO);
            }
        });
        dibujarPoligonos.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                canvas.cambiarAccion(EjemploCanvas.D_POLIGONO);
```

```
            }
        });
        seleccionarPunto.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                canvas.cambiarAccion(EjemploCanvas.S_PUNTO);
            }
        });
        seleccionarPoligono.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                canvas.cambiarAccion(EjemploCanvas.S_POLIGONO);
            }
        });
        mostrarNombres.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                canvas.cambiarNombres(mostrarNombres.isSelected());
            }
        });
        mostrarCoordenadas.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                canvas.cambiarCoordenadas(mostrarCoordenadas.isSelected());
            }
        });
        distancia.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                canvas.eliminarPoligonoTemporal();
                canvas.cambiarAccion(EjemploCanvas.NADA);
                canvas.calcularDistancia();
            }
        });
        borrar.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                canvas.borrar();
                canvas.cambiarAccion(EjemploCanvas.NADA);
            }
        });

        botones.add(Box.createVerticalGlue());
        botones.add(dibujarPuntos);
        botones.add(Box.createVerticalGlue());
        botones.add(dibujarPoligonos);
        botones.add(Box.createVerticalGlue());
        botones.add(seleccionarPunto);
        botones.add(Box.createVerticalGlue());
        botones.add(seleccionarPoligono);
        botones.add(Box.createVerticalGlue());
        botones.add(mostrarNombres);
        botones.add(Box.createVerticalGlue());
        botones.add(mostrarCoordenadas);
        botones.add(Box.createVerticalGlue());
        botones.add(distancia);
        botones.add(Box.createVerticalGlue());
        botones.add(borrar);

        // Panel general, que contendra los botones a la izquierda y la zona
        // de dibujo a la derecha
        JPanel contenedor = new JPanel();
        contenedor.setLayout(new FlowLayout());
        contenedor.add(botones);
        contenedor.add(canvas);

        // Asginamos el panel general a la ventana principal
        this.setContentPane(contenedor);
    }
```

```
        // Metodo principal
        static public void main(String ()args) {
            EjemploGrafico f = new EjemploGrafico();
            // Adaptamos la ventana al tamaño de los componentes que la conforman
            f.pack();
            // La mostramos
            f.setVisible(true);
        }
    }
```

EjemploCanvas.java

```
    import JavaRG.Nucleo2D.*;
    import JavaRG.Interfaz.*;
    import JavaRG.Basica.*;
    import java.awt.*;
    import java.awt.event.*;
    import javax.swing.*;

    public class EjemploCanvas extends JComponent {

        private static int tamX = 400;
        private static int tamY = 400;

        // Posibles acciones a realizar
        public static final int NADA = 0; // Ninguna accion
        public static final int D_PUNTO = 1; // Dibujar puntos
        public static final int S_PUNTO = 2; // Seleccionar puntos
        public static final int D_POLIGONO = 3; // Dibujar polígono
        public static final int S_POLIGONO = 4; // Seleccionar polígono

        // Maxima distancia de seleccion de una figura
        public static final int umbral = 4;

        // Accion actual
        private int accion = NADA;

        // Numero maximo de objetos geometricos de cada tipo
        private static int max = 100;

        // Objetos geometricos dibujados
        private Punto2D puntos() = new Punto2D(max);
        private Poligono2D poligonos() = new Poligono2D(max);

        // Numero de objetos geometricos dibujados
        private int numPuntos = 0;
        private int numPoligonos = 0;

        // Polígono que se está dibujando
        private Poligono2D poligonoTemporal = null;

        // Objetos seleccionados
        private int puntoSeleccionado = -1;
        private int poligonoSeleccionado = -1;

        // Opciones de visualizacion
        private boolean mostrarCoordenadas = false;
        private boolean mostrarNombres = false;

        // Resultado del cálculo de la distancia
        private Segmento2D resultado = null;

        public EjemploCanvas() {
            super();
            // Cambiamos el tamaño
```

```
        setPreferredSize(new Dimension(tamX, tamY));
        // Asociamos la zona de dibujo a la clase que maneja
        // los eventos del raton
        addMouseListener(new TicRaton());
}

public void paint(Graphics g) {
        // Cambiamos el color de fondo
        Graphics2D g2 = (Graphics2D) g;
        g.setColor(Color.white);
        g.fillRect(0, 0, getSize().width, getSize().height);

        // Dibujo de puntos
        for (int i = 0; i < numPuntos; i++)
        {
                puntos(i).dibujar(g, mostrarCoordenadas);
                if (mostrarNombres) puntos(i).dibujarNombre(g);
        }

        // Dibujo de poligonos
        for (int i = 0; i < numPoligonos; i++)
        {
                poligonos(i).dibujar(g, mostrarCoordenadas);
                if (mostrarNombres) poligonos(i).dibujarNombre(g);
        }
        if (poligonoTemporal != null) poligonoTemporal.dibujar(g, mostrarCoordenadas);

        // Dibujo de objetos seleccionados
        if (puntoSeleccionado != -1)
                puntos(puntoSeleccionado).dibujarSeleccionado(g, mostrarCoordenadas);
        if (poligonoSeleccionado != -1)
                poligonos(poligonoSeleccionado).dibujarSeleccionado(g, mostrarCoordenadas);
        // Dibujo del resultado si existe
        if (resultado != null)
        {
                resultado.dibujarResultado(g);
                resultado.dibujarNombre(g);
        }
}

// Modifica la accion a realizar cuando se pincha sobre la zona
// de dibujo
public void cambiarAccion(int nuevaAccion) {
        accion = nuevaAccion;
        if (accion != D_POLIGONO) {
                poligonoTemporal = null;
                repaint();
        }
        resultado = null;
}

// Métodos para el cambio de las variables booleanas mostrarCoordenadas y
// mostrarNombres
public void cambiarCoordenadas(boolean valor) {mostrarCoordenadas = valor; repaint();}
public void cambiarNombres(boolean valor) {mostrarNombres = valor; repaint();}

// Borra el polígono temporal
public void eliminarPoligonoTemporal() {
        poligonoTemporal = null;
        repaint();
}

// Borra los objetos geometricos dibujados
public void borrar() {
```

```
        puntoSeleccionado = -1;
        poligonoSeleccionado = -1;
        numPuntos = 0;
        numPoligonos = 0;
        resultado = null;
        eliminarPoligonoTemporal();
    }

    // CLASE PRIVADA PARA EL MANEJO DE EVENTOS DEL RATON
    private class TicRaton extends MouseAdapter {
        public void mousePressed (MouseEvent e) {
            int x = e.getX();
            int y = -e.getY();

            switch(accion) {
                // Dibujo de puntos
                case D_PUNTO:
                    if (numPuntos < max) {
                        puntos(numPuntos) = new Punto2D(x, y);
                        String nombre = "PUNTO" + (numPuntos+1);
                        puntos(numPuntos).cambiarNombre(nombre);
                        numPuntos++;
                        repaint();
                    }
                break;
                // Seleccion de puntos
                case S_PUNTO:
                    double minimaDistancia = java.lang.Integer.MAX_VALUE;
                    Punto2D p = new Punto2D(x,y);
                    for (int i=0; i<numPuntos;i++)
                    {
                        double distancia = puntos(i).distancia(p);
                        // Determinamos el polígono más cercano al punto
                        // en el que se pincho con el raton
                        if (distancia < minimaDistancia) {
                            minimaDistancia = distancia;
                            puntoSeleccionado = i;
                        }
                    }
                    // Solo seleccionamos una figura si su distancia al
                    // punto pinchado es menor de un umbral
                    if (minimaDistancia > umbral)
                        puntoSeleccionado = -1;
                    repaint();
                break;
                // Dibujo de poligonos
                case D_POLIGONO:
                    if (numPoligonos < max) {
                        if (poligonoTemporal == null)
                            poligonoTemporal = new Poligono2D();
                        poligonoTemporal.insertar(new Punto2D(x, y));
                        if (poligonoTemporal.esCompleto())
                        {
                            poligonos(numPoligonos) = new Poligono2D(poligonoTemporal);
                            String nombre = "POLIG" + (numPoligonos+1);
                            poligonos(numPoligonos).cambiarNombre(nombre);
                            numPoligonos++;
                            poligonoTemporal = null;
                        }
                        repaint();
                    }
                break;
                // Seleccion de poligonos
                case S_POLIGONO:
```

```
                        double minDistancia = java.lang.Integer.MAX_VALUE;
                        Punto2D punto = new Punto2D(x,y);
                        for (int i=0; i<numPoligonos;i++)
                        {
                            double distancia = poligonos(i).distancia(punto);
                            // Determinamos el polígono más cercano al punto
                            // en el que se pincho con el raton
                            if (distancia < minDistancia) {
                                minDistancia = distancia;
                                poligonoSeleccionado = i;
                            }
                        }
                        // Solo seleccionamos una figura si su distancia al
                        // punto pinchado es menor de un umbral
                        if (minDistancia > umbral)
                            poligonoSeleccionado = -1;
                        repaint();
                    break;
                    default:
                        // No se hace nada
                    break;
                }
            }
        }

        // Calcula la distancia y crea el objeto geométrico 'resultado' para mostrar dicha
        // distancia gráficamente
        public void calcularDistancia() {
            // Primero comprobamos que haya tanto un punto como un polígono seleccionados
            if (poligonoSeleccionado == -1 || puntoSeleccionado == -1)
            {
                    JOptionPane.showMessageDialog(this, "Debe haber un punto y un poligono
seleccionados", "ERROR",JOptionPane.WARNING_MESSAGE);
            }
            else try {

                // calculamos la distancia, así como el punto perteneciente al poligono
                // seleccionado mas cercano al punto seleccionado
                Punto2D puntoCercano = new Punto2D();
                                                                    double distancia =
poligonos(poligonoSeleccionado).puntoMasCercano(puntos(puntoSeleccionado), puntoCercano);
                // Creamos el segmento resultado entre los dos puntos anteriores, dandole
                // como nombre la distancia entre ambos
                resultado = new Segmento2D(puntos(puntoSeleccionado), puntoCercano);
                resultado.cambiarNombre(String.valueOf(distancia));
                // Mostramos el resultado por la salida estandar
                System.out.println("Distancia de " + puntos(puntoSeleccionado).obtenerNombre() +
" a " + poligonos(poligonoSeleccionado).obtenerNombre() + " : " + distancia);
                // y redibujamos la zona de dibujo para que se muestre el resultado
                repaint();
            } catch (GeomException ge) {System.out.println(ge); }
        }
}
```

3. Manual de usuario de la interfaz de la librería

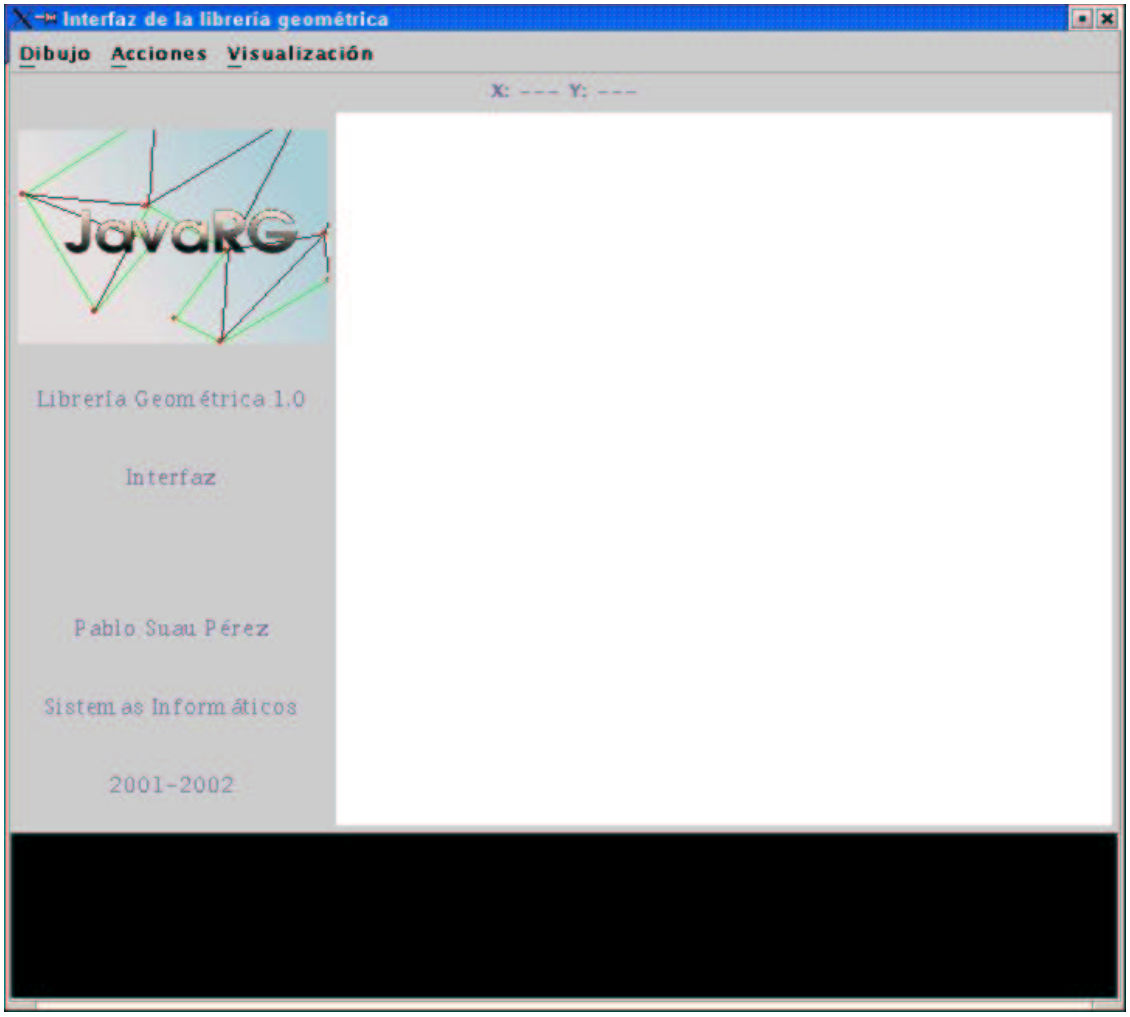
El presente apartado explica cómo usar la aplicación gráfica que acompaña a la librería, en el paquete **JavaRG/Interfaz**. Con esta aplicación podremos dibujar los objetos geométricos existentes en *JavaRG*, así como realizar operaciones entre ellos de forma gráfica.
Aunque esta interfaz no forma parte de la librería, se incluye por diversos motivos:

- Se pueden usar casi la totalidad de los métodos de la mayoría de los objetos geométricos (la interfaz no permite el dibujo de direcciones, vectores o cajas), por lo que como herramienta de desarrollo tiene la utilidad de permitir comprobar que estos métodos se encuentran bien implementados.
- Se puede tomar como base en futuros desarrollos gráficos que hagan uso de la librería geométrica, siendo un ejemplo mucho más complejo que el que se incluye en el *Tutorial de uso de la librería*.

Se incluye un fichero denominado **Interfaz.jar**, que contiene la aplicación así como todos los archivos necesarios para su funcionamiento. Para ponerla en funcionamiento, nos situamos en el directorio donde se encuentre este archivo y tecleamos:

java −jar Interfaz.jar

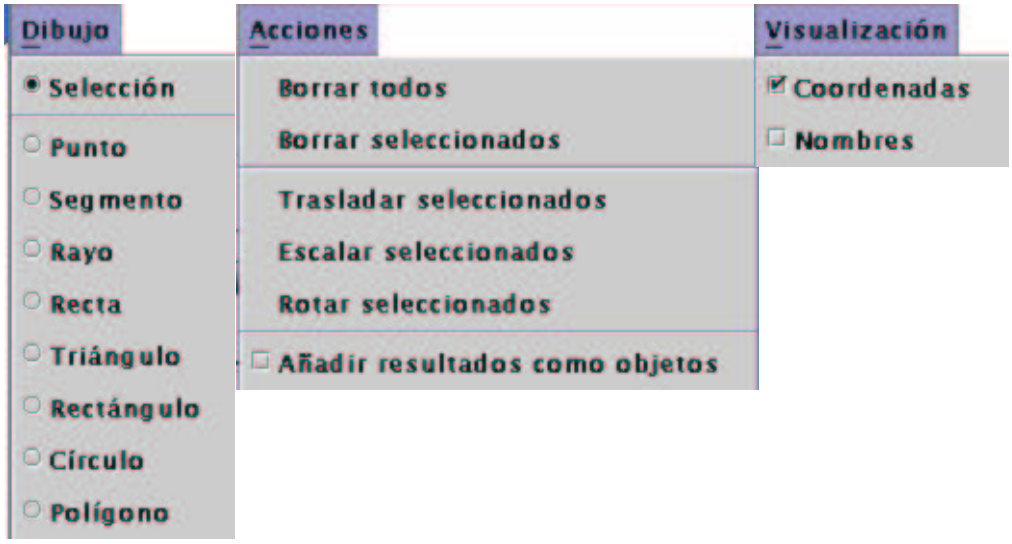
El aspecto inicial de la aplicación, en un entorno Linux, es el siguiente (puede varíar ligeramente en un entorno Windows):



En la parte superior disponemos de una barra de menús, cuyas distintas opciones serán explicadas más adelante. En la parte inferior, de color negro, disponemos de una zona de texto donde se nos mostrarán distintos mensajes y en donde aparecerán los resultados de las operaciones que vayamos realizando con los objetos geométricos. La parte central de la aplicación está dividida en dos partes. En la derecha, de color blanco, tenemos la zona de dibujo, donde podremos dibujar los objetos y se nos mostrarán los resultados de forma gráfica, siempre que esto sea posible. Sobre dicha zona de dibujo aparecerán dos coordenadas. Estas coordenadas representan el punto en el cual está situado el cursor en la zona de dibujo, y se actualizan cada vez que lo movemos el cursor dentro de dicha zona. A la izquierda de la zona de dibujo observamos un panel, que inicialmente nos muestra el nombre de la aplicación así como su autor. Cuando seleccionemos algún objeto geométrico dibujado anteriormente, será en esta zona donde aparecerán las distintas operaciones que podremos realizar con dicho objeto, en forma de botones.

3.1. Barra de menús

A continuación se indica la función de cada opción de los distintos menús de opciones situados en la barra superior de menús.



- **Menú DIBUJO**: menú en el que podremos seleccionar el tipo de objeto geométrico que estaremos dibujando al pinchar con el ratón sobre la zona de dibujo, y que también nos permitirá ponernos en modo *selección*, de tal forma que en lugar de dibujar podremos seleccionar objetos ya dibujados.
  - **Selección:** Permite seleccionar un objeto geométrico dibujado pulsando con el ratón sobre alguna de sus aristas o lados. Al seleccionar un objeto geométrico éste quedará resaltado con un color distinto al resto de objetos de mismo tipo,y en el panel de la izquierda aparecerán una serie de botones que nos permitirán realizar distintas operaciones sobre el objeto seleccionado. Se explicará más acerca de estos botones en la siguiente sección. Podemos **seleccionar más de un objeto geométrico**, incluso si son objetos de distinto tipo, pinchando con el ratón sobre los que deseemos seleccionar mientras mantenemos pulsada la tecla *mayúsculas*. Si se tiene más de un objeto seleccionado, los botones del panel izquierdo desaparecerán, siendo su única utilidad el borrar más de una figura a la vez. Para deseleccionar, simplemente pincharemos con el ratón sobre cualquiera parte de la zona de dibujo lo suficientemente alejada de cualquier objeto geométrico. Si solo hubiera un objeto seleccionado, los botones correspondientes a las acciones que se pueden realizar con el mismo (y que como se ha comentado aparecen en el panel de la izquierda) dejarán ver de nuevo el mismo texto que al iniciarse la aplicación.
  - **Punto**: Con esta opción podremos dibujar puntos. Cada vez que pinchemos con el ratón sobre la zona de dibujo un nuevo punto será creado y mostrado.
  - **Segmento**: Permite dibujar segmentos. Al pinchar con el ratón sobre la zona de dibujo indicaremos la situación del primer extremo del segmento, que será mostrado. Al realizar la misma operación por segunda vez, se creará el segmento. Una vez completado un segmento, podremos seguir creando más objetos de este tipo de la misma forma. El orden en que se dibujen los extremos del segmento indicará la orientación del mismo (en *JavaRG*, todos los segmentos son *segmentos orientados)*.
  - **Rayo**: Opción para el dibujo de rayos. El proceso es similar al del dibujo de un segmento. Al pinchar por primera vez con el ratón sobre la zona de dibujo indicaremos la situación del origen del rayo. Pinchando por segunda vez estaremos indicando la dirección en la que se extenderá el rayo (el vector director se obtiene como diferencia entre estos dos puntos). Tras dibujar un rayo de esta forma, podremos seguir dibujando los que queramos siguiendo el mismo proceso.
  - **Recta**: Permite dibujar rectas. El dibujo de rectas se realiza de la misma forma que el de segmentos. El primer punto indicará el punto incial y la diferencia entre el primero

y el segundo el vector director de la ecuación paramétrica de la recta.

- **Triángulo**: Permite dibujar triángulos. Cada vez que pinchemos con el ratón sobre la zona de dibujo estaremos definiendo la situación de un vértice del triángulo. Una vez hayamos dibujado tres vértices, se creará un triángulo, y podremos seguir creando objetos de este tipo de la misma forma.
- **Rectángulo**: Opción para dibujar rectángulos. Al pinchar por primera vez con el ratón sobre la zona de dibujo estaremos indicando la situación de la primera esquina (superior izquierda o inferior derecha). La segunda vez que lo hagamos definiremos la esquina contraria, de tal forma que el proceso es como si dibujáramos un segmento, solo que ese segmento se sitúa entre esquinas opuestas del rectángulo.
- **Círculo**: Permite dibujar una circunferencia. Pinchando por primera vez sobre la zona de dibujo indicamos la posición de su centro. Al mover el ratón veremos como el radio va aumentando o disminuyendo, hasta que pinchemos por segunda vez con el ratón sobre la zona de dibujo, momento en el cual se creará el nuevo círculo. Cada vez que creemos un objeto de este tipo podremos seguir dibujándo círculos siguiendo el mismo proceso.
- **Polígono**: Con esta opción podemos dibujar polígonos. Los polígonos que dibujemos van a ser siempre completos o cerrados. Cada vez que pinchemos con el ratón sobre la zona de dibujo añadiremos un vértice al polígono. Para completarlo pulsamos dos veces sobre el mismo punto (dicho punto solo se añadirá *una vez*). El sentido del polígono (horario o antihorario) vendrá determinado por el orden en que se dibujaron sus vértices.

- **Menú ACCIONES**: Opciones para el manejo de los objetos ya dibujados, y que afectan al dibujo de los mismos o su representación en la zona de dibujo (las operaciones puramente geométricas son las que aparecen en los botones de la parte izquierda cuando seleccionamos un objeto).
  - **Borrar todos**: Borra todos los objetos geométricos dibujados hasta el momento.
  - **Borrar seleccionados**: Borra los objetos geométricos seleccionados, si alguno lo está.
  - **Trasladar seleccionados**: Permite cambiar la posición de los objetos seleccionados. Aparecerá un cuadro de diálogo que nos pedirá la traslación a aplicar tanto en el eje x como en el eje y. Si queremos que alguna de las dos coordenadas quede sin modificarse, deberemos darle el valor 0.
  - **Escalar seleccionados**: Permite aumentar o disminuir el tamaño de los objetos seleccionados. Aparecerá un cuadro de diálogo pidiéndonos el factor de escala a aplicar en cada eje. En caso de no quere que se modifique la escala de alguna de las coordenadas, deberemos asignarle un valor de escala de 1. Valores mayores aumentan el tamaño y menores lo disminuyen.
  - **Rotar seleccionados**: Permite rotar cada objeto geométrico seleccionado respecto a su propio centro de gravedad o punto central. Se nos pedirá el ángulo, en grados, a rotar, el cual podrá tener valor positivo para realizar un giro antihorario o negativo para que sea antihorario.
  - **Añadir resultados como objetos**: Ciertas operaciones con los objetos geométricos producirán como resultado otro objeto geométrico. Por ejemplo, la interesección entre dos segmentos podría ser a su vez otro segmento o un punto. Si esta opción está seleccionada, el objeto resultante de cada operación con objetos geométricos será añadido a la zona de dibujo como si lo hubiera creado el usuario de la aplicación.
- **Menú VISUALIZACIÓN**: Contiene opciones para indicar la información sobre los objetos geométricos dibujados que nos interesa que se muestre.
  - **Coordenadas**: Si esta opción se encuentra seleccionada (por defecto sí que lo estará),

se mostrarán las coordenadas de los vértices de los objetos y de los puntos.

- **Nombres**: Si esta opción se encuentra seleccionada (por defecto no lo estará), se mostrará el nombre de todos los objetos dibujados. A cada objeto creado la aplicación le asigna un nombre de forma automática, consistente en una secuencia de letras identificando el tipo de objeto y un número, que comenzando en 1, se incrementa para cada objeto del mismo tipo dibujado (este valor **no** vuelve a ser 1 si se borran todos los objetos geométricos). El nombre podrá ser modificado más adelante, mediante los botones que aparecerán en el panel izquierdo cuando se seleccione un objeto geométrico. Hay que destacar que este nombre se lo proporciona por defecto a un objeto *la aplicación.* Si creamos un objeto geométrico utilizando el API de la librería geométrica en nuestras propias aplicaciones, se crea sin nombre.

3.2. Botones de operación

Al seleccionar un único objeto geométrico, en el panel de la parte izquierda aparecerán unos botones que nos permitirán seleccionar distintas operaciones que realizar con el objeto, así como obtener información del mismo. Aunque el tipo y cantidad de operaciones a realizar con cada objeto varía, hay una serie de aspectos en común para todos ellos:

- Ciertas operaciones no implican tan solo al propio objeto, como por ejemplo el cálculo de la distancia de un punto al objeto. Cuando esto sea así, aparecerá una ventana en la que podremos seleccionar el o los objetos con los que realizar la operación. Si no existe ningún objeto dibujado adecuado para realizar la operación seleccionada, se nos mostrará un mensaje de error. Por ejemplo, en el caso del cálculo de la distancia de un punto al objeto seleccionado, si no hay ningún punto creado, se nos mostrará un mensaje de error. En caso contrario se nos mostrará el nombre de todos los puntos dibujados. Si se seleccionaron varios puntos se nos mostrará el resultado para todos ellos, tanto en el cuadro de texto inferior, como en modo gráfico (en caso de que el resultado se pueda expresar de forma gráfica). En algunas situaciones el mensaje de error puede mostrarse por otras causas (porque sea necesario seleccionar más de un objeto geométrico para realizar una operación, etc.).
- Para algunos objetos los botones quedan separados en dos grupos, pudiendo alternar entre uno y otro mediante botones etiquetados como ’>>’ y ’<<’ en la parte inferior del panel.
- El resultado de las operaciones de distancia se expresará de forma gráfica mediante un segmento, que podrá ser añadido, si se selecciona la opción de menú adecuada, a los objetos definidos por el usuario.
- Si el resultado de una operación es un vector, tan solo se mostrarán sus coordenadas en el cuadro de texto. No se ha realizado ninguna representación gráfica de los vectores.
- Para todos los objetos geométricos se podrá seleccionar una operación especial, *Cambiar Nombre*, que como su propio nombre indica, permite modificar el nombre del objeto geométrico seleccionado.

Los siguientes apartados indican cual es la función de cada uno de los botones para cada tipo de objeto geométrico.

3.2.1. Punto

Comprobar igualdad: Comprueba si el punto seleccionado es igual a otro objeto geométrico. Un punto será igual a otro objeto geométrico si dicho objeto es a su vez un punto y las coordenadas de ambos son las mismas.
Comparar x: Compara la coordenada x del punto seleccionado con la coordenada x de otro punto,

indicando si es mayor, menor o igual.
Comprar y: Compara la coordenada y del punto seleccionado con la coordenada y de otro punto, indicando si es mayor, menor o igual.
Posición relativa: Para realizar esta operación deben existir al menos tres puntos. Se determina la posición de un punto (izquierda, derecha o colineal), tomando como referencia el punto seleccionado y un segundo punto que deberemos indicar.
Distancia a punto: Determina la distancia del punto seleccionado a otro punto.
Vector desde origen: Indica las componentes del vector definido entre el origen de coordenadas y el punto seleccionado.
Vector a origen: Indica las componentes del vector definido entre el punto seleccionado y el origen de coordenadas.
Vector diferencia: Indica las componentes del vector definido entre el punto seleccionado y otro punto.

3.2.2. Segmento

Comprobar igualdad: comprueba si el segmento seleccionado es igual a otro objeto. Para que un segmento orientado sea igual a otro objeto geométrico, dicho objeto deberá ser a su vez un segmento, y tener los mismos puntos de inicio y final.
Comienzo: Muestra el punto de inicio del segmento orientado (el que fue creado en primer lugar al dibujar el segmento).
Fin: Muestra el punto final del segmento orientado (el que fue creado en segundo lugar al dibujar el segmento).
Min x: Determina la mínima coordenada x de entre todos los puntos del segmento.
Max x: Determina la máxima coordenada x de entre todos los puntos del segmento.
Min y: Determina la mínima coordenada y de entre todos los puntos del segmento.
Max y: Determina la máxima coordenada y de entre todos los puntos del segmento.
Horizontal: Determina si el segmento es horizontal, lo cual sucederá cuando ambos extremos tengan la misma coordenada y.
Vertical: Determina si el segmento es vertical, lo cual sucederá cuando ambos extremos tengan la misma coordenada x.
Rayo: Construye un rayo a partir de los dos extremos del segmento, tomándose como origen del rayo el punto de comienzo del segmento.
Recta: Construye una recta a partir de los dos extremos del segmento.
Vértice: Permite seleccionar cualquiera de los extremos del segmento, mostrando sus coordenadas.
Distancia a punto: Obtiene la distancia de un punto al segmento.
Longitud: Calcula la longitud del segmento (que se trata de la distancia entre sus dos extremos).
Longitud al cuadrado: Calcula la longitud al cuadrado del segmento.
Punto a izquierda: Indica si un punto se encuentra a la izquierda del segmento seleccionado.
Punto a derecha: Indica si un punto se encuentra a la derecha del segmento seleccionado.
Contiene punto: Indica si un punto se encuentra contenido por el segmento seleccionado.
Contiene o colineal con punto: Indica si un punto se encuentra contenido por el segmento seleccionado o si dicho punto es colineal con los extremos del segmento.
Intersección propia con segmento: Obtiene el resultado de la intersección propia (los extremos no se incluyen como resultado de la intersección) entre el segmento seleccionado y otro segmento. Este resultado podrá ser un punto, otro segmento, o ningún objeto (en el caso de que no haya intersección).
Intersección con segmento Obtiene el resultado de la intersección entre el segmento seleccionado y otro segmento. Este resultado podrá ser un punto, otro segmento, o ningún objeto (en el caso de que no haya intersección).

3.2.3. Rayo

Comprobar igualdad: Comprueba si el rayo es igual a otro objeto geométrico. Esto sucederá si dicho objeto geométrico es a su vez un rayo, y ambos rayos tienen el mismo punto de origen y vector director.
Comienzo: muestra el origen del rayo, tanto en forma gráfica (como un punto), como por medio de sus coordenadas en el cuadro de texto inferior.
Director: Muestra las componentes del vector director del rayo seleccionado.
Horizontal: Indica si el rayo seleccionado es horizontal, lo cual sucederá cuando el vector director del mismo tenga componente nula en el eje y.
Vertical: Indica si el rayo seleccionado es vertical, lo cual sucederá cuando el vector director del mismo tenga compoenente nual en el eje x.
Recta asociada: Crea una recta a partir del rayo, tomando como referencia su punto inicial y su vector director.
Distancia a punto: Obtiene la distancia de un punto al rayo seleccionado.
Punto a izquierda: Determina si un punto se encuentra a la izquierda del rayo seleccionado.
Punto a derecha: Determina si un punto se encuentra a la derecha del rayo seleccionado.
Punto a partir del parámetro: Sustituye el valor de parámetro indicado (que debe ser mayor o igual que cero) en la ecuación paramétrica obtenida a partir del punto de comienzo y el vector director del rayo seleccionado, mostrando el punto resultante.
Parámetro a partir de punto: Realiza la operación inversa a la anterior. Si el punto se encuentra contenido en el rayo, indicará cuál debería ser el valor del parámetro dentro de la ecuación paramétrica para obtener como resultado dicho punto.
Contiene punto: Determina si un punto se encuentra contenido en el rayo seleccionado.
Contiene o colineal con punto: Determina si un punto se encuentra contenido en el rayo seleccionado, o si el primero es colineal con cualquier pareja de puntos del segundo.
Intersección con recta: Determina el resultado de la intersección de una recta con el rayo seleccionado. Dicho resultado puede ser otro rayo, un segmento, un punto, o ningún objeto en caso de que no haya intersección.
Intersección con rayo: Determina el resultado de la intersección del rayo seleccionado con otro rayo. El resultado de dicha intersección puede ser otro rayo, un segmento, un punto, o ningún objeto en caso de que no haya intersección.
Intersección con segmento: Determina el resultado de la intersección del rayo seleccionado con un segmento. El resultado de dicha intersección puede ser otro segmento, un punto, o ningún objeto en caso de que no haya intersección.

3.2.4. Recta

Comprobar igualdad: Comprueba si la recta es igual a otro objeto geométrico. Esto sucederá si dicho objeto geométrico es una recta y el punto base de una recta está contenida en la otra y ambas tienen la misma dirección.
Director: Muestra las componentes del vector director de la recta.
Opuesta: Obtiene la recta con vectori director opuesto.
Horizontal: Determina si la recta es horizontal. Esto será así si el vector director de la misma no tiene componente en el eje y.
Vertical: Determina si la recta es vertical. Esto será así si el vector director de la misma no tiene componente en el eje x.
Coeficiente a: La recta está construida a partir de su ecuación paramétrica. Esta operación y las dos

siguientes nos permiten obtener los coeficientes de la ecuación cartesiana. Esta opción en concreto nos permite obtener el primer coeficiente de la ecuación cartesiana de la recta. Hay que tener en cuenta que la ecuación paramétrica nos permite expresar rectas verticales, de pendiente infinita, imposibles de expresar con ecuaciones cartesianas.
Coeficiente b: Obtiene el segundo coeficiente de la ecuación cartesiana de la recta.
Coeficiente c: Obtiene el tercer coeficiente de la ecuación cartesiana de la recta.
X en y: Determina la coordenada x del punto perteneciente a la recta cuya coordenada y se especifique.
Y en x: Determina la coordenada y del punto perteneciente a la recta cuya coordenada x se especifique.
Distancia a punto: Calcula la distancia de un punto a la recta.
Punto a partir de parámetro: Sustituye el valor del parámetro indicado en la ecuación paramétrica de la recta, mostrando el punto resultante.
Parámetro a partir de punto: Si el punto seleccionado está contenido en la recta se nos indica el valor del parámetro que deberiamos utilizar en la ecuación paramétrica de la recta para obtener dicho punto.
Intersección con recta: Determina el resultado de la intersección de la recta seleccionada con otra recta. El resultado puede ser otra recta (si ambas rectas son coincidentes), un punto o ningún objeto, si no hay intersección.
Intersección con rayo: Determina el resultado de la intersección de la recta seleccionada con un rayo. El resultado puede ser un rayo (si el rayo está contenido en la recta), un punto, o ningún objeto si rayo y recta no intersectan.
Intersección con segmento: Determina el resultado de la intersección de la recta seleccionada con un segmento. El resultado puede ser el propio segmento, si éste está contenido en la recta, un punto, o ningún objeto si recta y segmento no intersectan.
Punto a izquierda: Indica si un punto se encuentra a la izquierda de la recta.
Punto a derecha: Indica si un punto se encuentra a la derecha de la recta.
Posición punto: Indica la posición de un punto respecto a la recta (izquierda, derecha o contenido).
Proyección ortogonal: Determina la proyección ortogonal de un punto sobre la recta. Dicha proyección es el punto de intersección entre la recta y una recta perpendicular a la misma que contenga al punto del que se desea conocer su proyección.
Perpendicular pasando por punto: Seleccionado un punto obtiene la recta perpendicular a la recta seleccionada que contenga dicho punto.

3.2.5. Triángulo

Comprobar igualdad: Comprueba si el triángulo es igual a otro objeto geométrico. Esto sucederá si dicho objeto es también un triángulo y además existe una permutación cíclica de los vértices de uno igual a los vértices del otro.
Comprobar degenerado: Comprueba si el triángulo es degenerado, lo cual sucede si sus tres vértices son colineales, y por tanto, su área es cero.
Opuesto: Obtiene el triángulo cuyos vértices se encuentran definidos en sentido contrario al del triángulo seleccionado.
Sentido: Determina si los vértices del triángulo seleccionado están definidos en sentido horario o antihorario.
Área sin signo: Calcula el área del triángulo, sin tener en cuenta el signo, el cual es dependiente de su sentido.
Vértice: Muestra las coordenadas del vértice que se le indique de entre los pertenecientes al triángulo seleccionado.
Distancia a punto: Determina la distancia de un punto al triángulo seleccionado.
Lados contienen punto: Comprueba si los lados del triángulo, definidos como segmentos entre los

vértices del mismo, contienen un determinado punto.
Contiene punto: Comprueba si el triángulo o alguno de sus lados contienen a un determinado punto.
Contiene punto estrictamente: Comprueba si el triángulo contiene un determinado punto, sin estar dicho punto contenido por alguno de sus lados.

3.2.6 Rectángulo

Comprobar igualdad: Comprueba si el rectángulo es igual a otro objeto geométrico. Esto sucederá si el otro objeto es también un rectángulo y ambos rectángulos tienen exactamente los mismos extremos.
Min X: Muestra la mínima coordenada x de entre todos los puntos pertenecientes al rectángulo.
Max X: Muestra la máxima coordenada x de entre todos los puntos pertenecientes al rectángulo.
Min Y: Muestra la mínima coordenada y de entre todos los puntos pertenecientes al rectángulo.
Max Y: Muestra la máxima coordenada y de entre todos los puntos pertenecientes al rectángulo.
Esquina inferior izquierda: Muestra el vértice situado en la parte inferior izquierda del rectángulo seleccionado.
Esquina superior derecha: Muestra el vértice situado en la parte superior derecha del rectángulo seleccionado. El por qué de la inclusión de estas dos opciones es debido a que internamente, en la librería *JavaRG*, un rectángulo está definido a partir de dos vértices opuestos, precisamente los situados en la esquina inferior izquierda y en la superior derecha.
Área: Calcula el área del rectángulo seleccionado.
Distancia a punto: Calcula la distancia de un punto al rectángulo seleccionado.
Vértice: Muestra las coordenadas del vértice que se le indique entre los pertenecientes al rectángulo seleccionado.
Borde contiene punto: Determina si alguno de los lados del rectángulo, definidos como segmentos entre cada par de sus vértices, contiene al punto indicado.
Contiene punto: Comprueba si el rectángulo o alguno de sus lados contienen a un determinado punto.
Contiene punto estrictamente: Comprueba si el rectángulo contiene un determinado punto, sin estar dicho punto contenido por alguno de sus lados.

3.2.7. Círculo

Centro: Muestra el centro del círculo.
Radio: Muestra el radio del círculo.
Comprobar igualdad: Comprueba si el círculo es igual a otro objeto geométrico. Esto será así si dicho objeto geométrico es también un círculo y ambos círculos tienen el mismo centro y radio.
Comprobar degenerado: Comprueba si el círculo seleccionado se trata de un círculo degenerado, lo cual sucede si su radio es cero.
Circunferencia contiene punto: Comprueba si un punto está contenido en la circunferencia del círculo.
Círculo contiene punto: Compruena si un punto está contenido por el círculo seleccionado o su circunferencia.
Contiene punto estrictamente: Comprueba si un pubto está contenido por el círculo seleccionado sin estar contenido por su circunferencia.
Distancia circunferencia a punto: Determina la distancia de un determinado punto a la circunferencia del círculo, tanto si el punto se encuentra contenido por el círculo como si no lo está.

3.2.8. Polígono

Comprobar igualdad: Comprueba si el polígono es igual a un objeto geométrico. Esto será así si

dicho objeto es a su vez un polígono y ambos polígonos tienen los mismos vértices, aunque no es necesario que estén definidos en el mismo orden y en el mismo sentido.

Min X: Determina la mínima coordenada x de entre todos los puntos pertenecientes al polígono.

Max X: Determina la máxima coordenada x de entre todos los puntos pertenecientes al polígono.

Min Y: Determina la mínima coordenada y de entre todos los puntos pertenecientes al polígono.

Max Y: Determina la máxima coordenada y de entre todos los puntos pertenecientes al polígono.

Sentido: Determina si los vértices del polígono están definidos en sentido horario o antihorario.

Cóncavo: Determina si el polígono seleccionado es un polígono cóncavo, es decir, si al menos uno de sus vértices es un vértice cóncavo.

Convexo: Determina si el polígono seleccionado es un polígono convexo, es decir, si todos sus vértices son vértices convexos.

Simple: Determina si el polígono seleccionado es un polígono simple, para lo cual cada vértice debe ser compartido tan solo por dos aristas y no debe haber ningún cruce entre aristas.

Área: Calcula el área del polígono con signo, dependiendo el signo de si el polígono fue definido en sentido horario o antihorario.

Distancia a punto: Determina la distancia de un punto a los lados del polígono seleccionado, tanto si el punto está contenido por el polígono como si no.

Vértice cóncavo: Determina si el vértice indicado es un vértice cóncavo. Para que esto sea así, si el polígono es antihorario, el punto Pi+1 deberá estar a la derecha del segmeno orientado (Pi−1, Pi). En el caso de sentido horario, debe estar a la izquierda.

Vértice convexo: Determina si el vértice indicado es un vértice convexo. Para que esto sea así, si el polígono es antihorario, el punto Pi+1 deberá estar a la izquierda del segmento orientado (Pi−1, Pi). En el caso de sentido horario, debe estar a la derecha.

Diagonal: Determina si el segmento definido entre los dos vértices indicados del polígono es una diagonal. Para ello, dicho segmento no debe interseccionar con ninguna arista del polígono.

Lados contienen punto: Determina si alguna de las aristas del polígono seleccionado contiene el punto indicado.

Contiene punto estrictamente: Determina si el punto indicado está contenido por el polígono seleccionado, pero por ninguna de sus aristas.

Contiene punto: Determina si el polígono seleccionado o alguna de sus aristas contienen al punto indicado.

Posiciones: Esta opción y las dos siguientes hacen uso de las facilidades proporcionadas por los *Circuladores*. Un circulador se define como iterador circular que permite recorrer todos los vértices de un polígono. Para obtener más información sobre los circuladores consultar el *Tutorial de uso de la librería geométrica*. Esta operación simplemente muestra el número de posiciones que tiene la estructura circular formada por el conjunto de vértices del polígono seleccionado. O dicho de una forma más sencilla, determina el número de vértices del polígono.

Posición anterior, Posición siguiente: La primera vez que utilizamos una de estas operaciones sobre un polígono seleccionado se nos mostrará su primer vértice. Las siguientes veces avanzaremos sucesivamente por los vértices del polígono (en un sentido u otro, según el botón pulsado; *Posición siguiente* permite avanzar por los vértices en el sentido en el que éstos fueron creados, y *Posición anterior* en sentido contario) de manera circular, de tal forma que el siguiente vértice al último sería el primero y el anterior al primero sería el último.

4. Detalles de implementación y conclusiones

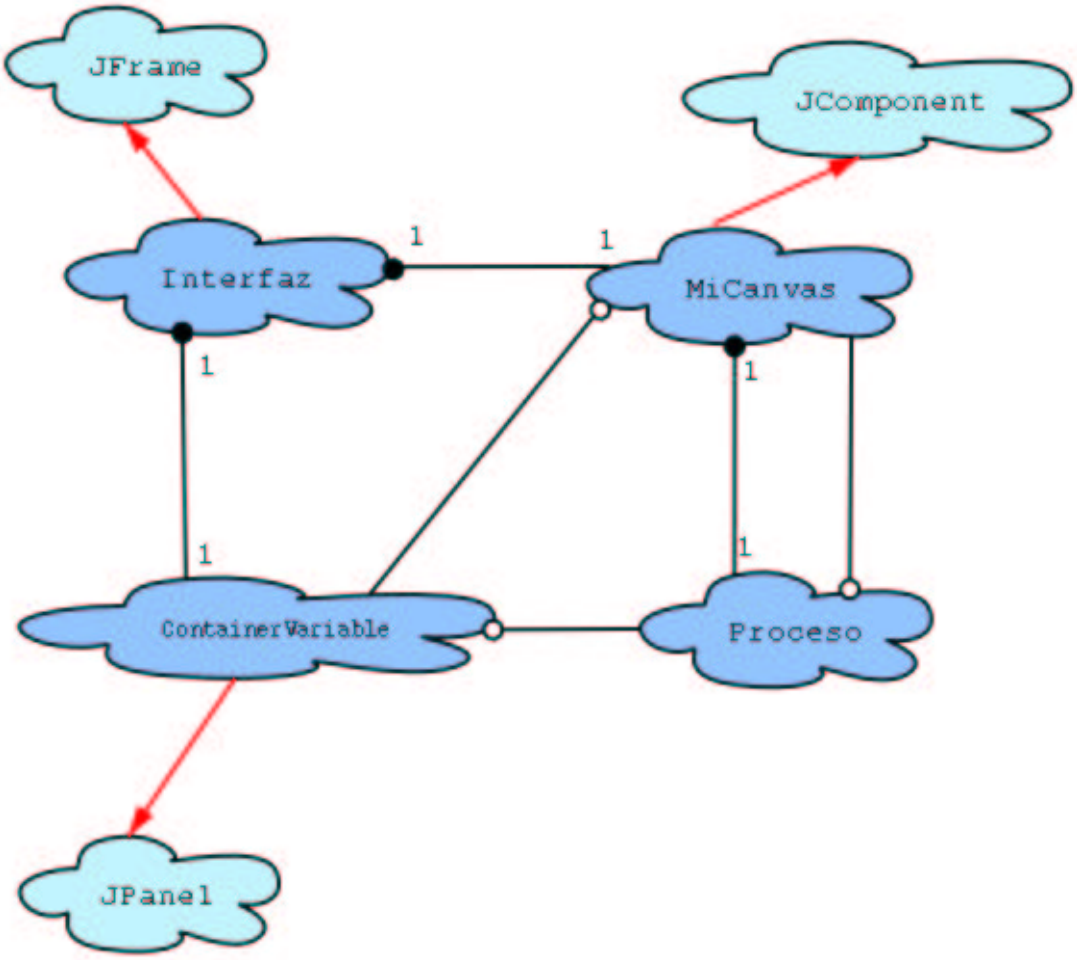
Al ser el proyecto de creación de la librería geométrica y de su interfaz más un trabajo de desarrollo que de investigación, pocas conclusiones sobre el proceso se pueden extraer, debiéndonos centrar más en los problemas surgidos durante su implementación así como las futuras mejoras que se podrían realizar.

De todas formas, el empleo de Java ha planteado una serie de ventajas con respecto a otros lenguajes, como el lenguaje C en el que está implementado CGAL. El empleo de paquetes de Java ha permitido crear una estructura de la librería mucho más clara, de tal forma que las clases no quedan divididas en distintas categorías solo formalmente, sino que además esta división queda explícita al tener paquetes distintos. Por otra parte, ciertas características de orientación a objetos también han supuesto una ventaja. La herencia nos ha permitido también estructurar la librería de forma más clara y cohesionada, ya que clases pertenecientes a un mismo paquete comparten ciertos métodos de su interfaz. Esto se traduce además en el uso de polimorfismo, lo que hace que, entre otras ventajas, algunos métodos puedan devolver como resultado objetos geométricos de distinto tipo, y que podamos acceder a métodos comunes a todos ellos (así, el resultado de una intersección entre dos rayos podría ser un rayo, un segmento, un punto, o null). La característica de los *interface* de Java ha sido determinante a la hora de la creación de los circuladores, pues éstos tan solo deben hacer uso de ciertos métodos proporcionados por los objetos que implementan dicha interfaz, sin preocuparse de como dichos objetos devuelven los resultados pedidos.

A continuación mostramos cómo se llevó a cabo la implementación, centrándonos en el diseño de la interfaz. Los dos siguientes apartados mostrarán cómo se podrían realizar estas mejoras y que aspectos de la librería son susceptibles de ser modificados para obtener una librería más adecuada. El último apartado de esta sección hace referencia a problemas surgidos durante el desarrollo.

4.1. Diseño de la interfaz

La interfaz ha sido implementada mediante el uso de cuatro clases distintas, entre las que existe una gran interacción. Las dos clases principales son **Interfaz** y **MiCanvas**, creadas siguiendo el mismo proceso que se describe en el ultimo punto del tutorial de uso de la librería geométrica, en el apartado de creación de interfaces gráficas.
A estas dos clases se le añaden dos más, **ContainerVariable**, encargada de la actualización de los botones del panel de la izquierda cuando se selecciona un objeto geométrico de un determinado tipo, y **Proceso**, que contiene los métodos llamados por los botones cuando son pulsados y que realizan relamente las llamadas a los métodos de la librería geométrica.
La relación entre las distintas clases se puede observar en el siguiente **diagrama de clases**, realizado empleando la *metodología de Booch:*



Una descripción más detallada de las clases antes indicadas sería la siguiente:

- **Interfaz**: es la clase que contiene el método *main*, y por lo tanto la encargada de poner en marcha la interfaz. Su funcionalidad básicamente es la del dibujado de la interfaz (no el dibujado de los objetos), creando la barra de menús, y controlando los eventos relacionados con las opciones de dicha barra de menús. Entre los elementos de la interfaz se incluyen un objeto de tipo *ContainerVariable* y otro objeto de tipo *MiCanvas*. Como además algunos métodos del *ContainerVariable* creado serán llamados desde el objeto de tipo *MiCanvas*, se debe pasar a éste último una referencia al ContainerVariable por medio de un método de *MiCanvas* llamado *asociarContainer*.
- **MiCanvas**: clase asociada a la zona de dibujo de la aplicación. Aunque realmente no es un objeto de tipo Canvas, se le da este nombre debido a su funcionalidad. Dicha funcionalidad no se limita a mostrar los objetos geométricos dibujados, el objeto u objetos seleccionados y los resultados, sino que también realiza otras acciones como almacenar los objetos creados en arrays, actualizar el panel de botones de la parte izquierda de la interfaz según el objeto seleccionado (de ahi la necesidad del uso del método *asociarContainer* indicado más arriba), etc. Contiene un objeto de tipo *Proceso*, que se asocia al canvas que lo creó mediante un argumento en el constructor de la clase proceso.
- **ContainerVariable**: básicamente se trata de un panel de botones, en el que se mostrarán las diversas operaciones a realizar con el objeto seleccionado según su tipo en forma de botones. Aunque el objeto del tipo *ContainerVariable* se crea en el interior de *Interfaz*, se asocia con

un determinado *MiCanvas*, de tal forma que la selección de un objeto geométrico en el interior de la zona de dibujo provocará la inmediata actualización del panel *ContainerVariable*, que quedará a su vez reflejada en el interfaz de la aplicación al pertener a un objeto de tipo *Interfaz*. El único propósito de un objeto de esta clase es crear los botones necesarios según el objeto seleccionado en su *MiCanvas* asociado, y sus respectivos manejadores de eventos. Cuando se pulse un botón de este panel se llamará al correspondiente método de la clase *Proceso* asociada al objeto *MiCanvas* que a su vez se relaciona con el *ContainerVariable.*

- **Proceso**: contiene métodos que se activan al pulsar los botones del panel correspondiente a la clase *ContainerVariable.* Estos métodos se corresponden con operaciones a realizar con el objeto geométrico seleccionado. Los pasos en todos ellos suelen ser comunes: primero se solicita al usuario (si es necesario) el o los otros objetos con los que realizar la operación (por ejemplo, si se trata del cálculo de intersecciones con segmentos se mostrarán todos los segmentos disponibles y se pedirá que se seleccione uno o varios), se llama al método o métodos correspondientes de *JavaRG*, y se muestra el resultado en la zona de dibujo implementada en el objeto *MiCanvas* asociado. Por lo tanto, es esta clase *Proceso* la que pone realmente en contacto la interfaz de la librería geométrica con la propia librería geométrica. Algunos de los métodos admiten parámetros, de tal forma que puedan ser llamados desde diversos manejadores de eventos de botones distintos de un objeto perteneciente a la clase *ContainerVariable*. Esto es así porque para esos botones el cambio de funcionalidad es muy pequeño, requiriéndose muy pocos cambios, y por lo tanto, se ahorran líneas de código dentro de *Proceso.*

Para más detalles acerca del código fuente de las clases citadas se recomienda consultar los anexos de la documentación.

4.2. Ampliación de la librería geométrica

La librería geométrica JavaRG surge como un intento de adaptación de CGAL al lenguaje Java; sin embargo, esta primera versión tan solo dispone de una pequeña funcionalidad comparada con CGAL. Por lo tanto, la mejora más evidente es la ampliación de sus funcionalidades, por medio de nuevas clases que representen nuevas estructuras de datos geométricas y algoritmos de geometría computacional. La parte de **Nucleo2D** se puede considerar completa, así que en caso de ampliar la librería, se debería prestar atención al resto de paquetes. A continuación se indica cómo se debería realizar este trabajo de ampliación, o en otras palabras, dónde debería ser colocada cada nueva clase creada:

- **Basica**: Aquí situaríamos todas aquellas nuevas estructuras geométricas que se basaran en objetos geométricos del núcleo, como el caso del polígono. Por ejemplo, recubrimientos convexos, triangulaciones de Delaunay (en estos dos casos hablaríamos de la *estructura geométrica*, no del proceso de obtención de los mismos), mapas planares, árboles, etc... Todos estos objetos deberían ser subclases de *ObjetoBasico*, y por lo tanto, tendrán que implementar la interfaz *InterfazCirculador*.
- **Soporte**: Este paquete debe contener clases auxiliares, así como clases que nos faciliten el manejo o visualización de los datos asociados a otros objetos de la librería (como el caso de los circuladores, que nos permiten ir avanzando por los vértices de un objeto de forma sencilla).

Aparte de añadir clases a los paquetes ya existentes, sería conveniente la creación de nuevos paquetes.

Dos nuevos paquetes a crear, que se encontrarían al mismo nivel que *Nucleo2D*, serían los paquetes *Nucleo3D* y *NucleodD*. Tal como indican sus nombres, contendrían objetos geométricos de 3 y d dimensiones, respectivamente. Sería conveniente, como en el caso de *Nucleo2D*, que todos los objetos geométricos de estos nuevos paquetes fueran subclases de una clase abstracta más general (por ejemplo *ObjetoGeometrico3D y ObjetoGeometricodD*) de tal forma que en estos núcleos también tuvieramos las ventajas asociadas al *Nucleo2D*, como funciones que puedan devolver más de un tipo de objeto geométrico según el resultado de las operaciones, la posibilidad de creación de contenedores de distintos tipos de objetos geométricos simultáneamente, etc.
Otro paquete a crear, que se situaría jerárquicamente por encima de *Basica*, sería aquel encargado de contener todas las clases relacionadas con distintos algoritmos y procesos de geométrica computacional, esto es, clases para realizar triangulaciones por diversos métodos, recubrimientos convexos, cálculo de intersección entre varios segmentos de forma eficiente, etc, con métodos que permitieran elegir el tipo de algoritmo a aplicar, calcular el tiempo de proceso, etc. Este paquete podría ser llamado, por ejemplo, *Algoritmos*.

4.3. Excepciones

Al utilizar la librería se hace evidente que existe una gran cantidad de excepciones geométricas que tratar. Se plantean dos problemas:

- Solo existe un tipo de excepción geométrica, lo que impide discriminar entre distintos tipos de excepciones y hacer un tratamiento más específico de las mismas.
- Ciertos métodos lanzan excepciones geométricas cuando se podrían haber obviado y haber hecho uso de las propias excepciones de Java. El ejemplo más claro lo tenémos en aquellos métodos que nos permiten acceder a los vértices de un objeto geométrico indicando el índice del vértice al que acceder. Si el índice indicado es incorrecto, porque sobrepasa el número de vértices del objeto en cuestión o tiene un valor negativo, se lanza una excepción geométrica; sin embargo, este tipo de excepciones las puede manejar Java perfectamente.

Por lo tanto, se podría mejorar el tema de las excepciones mediante dos vías: creación de diversos tipos de excepciones distintos, y eliminación de excepciones redundantes.

4.4. Problemas surgidos

Problemas de implementación aparte, los cuales fueron resueltos durante la misma, es necesario destacar que durante las pruebas se encontró un error en el algoritmo de cácluclo de intersección entre segmentos extraido de la bibliografía utilizada (en concreto, de las transparencias utilizadas en la asignatura de *Razonamiento Geométrico*). Dicho algoritmo no tenía el objetivo de calcular la intersección, sino que de averiguar si había o no intersección. Esto se realiza antes del cálculo de la intersección en sí misma, porque en caso de que no hubiera intersección no sería necesario realizar los cálculos asociados para averiguarla, de mayor complejidad que una simple comprobación booleana. El algoritmo utilizado originalmente para la **comprobación booleana de intersección propia de segmentos** a partir de las posiciones relativas de sus segmentos, y que se descubrió que era erróneo, era el siguiente:

```
Bool IntersectaProp (Punto a,b,c,d)
        Si (Colineal (a,b,c) o Colineal(a,b,d) o Colineal(c,d,a)
        o Colineal(c,d,b))
                devolver FALSE;
    Sino devolver (Xor(Izquierda(a,b,c), Izquierda(a,b,d)) &&
                   Xor(Izquierda(c,d,a), Izquierda(c,d,b));
```

donde:

- *Colineal* e *Izquierda* son funciones de posición relativa, devolviendo la primera true si los tres puntos pasados como parámetro son colineales y la segunda si el tercer punto se encuentra a la izquierda del segmento definido entre los dos primeros.
- El primer segmento tendrá como extremos los puntos a y b, y el segundo tendrá como extremos los puntos c y d.

El caso concreto en que esta función no funciona correctamente se da cuando la intersección de los dos segmentos tiene como resultado otro segmento, es decir, cuando existe más de un punto de intersección, como en el siguiente ejemplo:

La solución consiste en añadir una condición más, de tal forma que distingamos entre las intersecciones cuyo resultado sea un único punto (condición que ya existía) y aquellas cuyo resultado sea más de un punto. Para ello, cada uno de los dos extremos de un segmento debe ser colineal con los dos extremos del otro segmento, y además, estar uno de estos extremos del primer segmento contenido en el segundo. Antes de la modificación, para que la intersección fuera propia, se debía de evitar la colinealidad, tal como se muestra en el algoritmo mostrado anteriormente, pero como ha quedado demostrado gráficamente en el ejemplo anterior, la colinealidad no tiene por qué evitar que haya intersecciones propias. Por ello, sustituimos la comprobación inicial de colinealidad por una comprobación de igualdad de los extremos de los segmentos.

```
Bool IntersectaProp (Punto a,b,c,d)
        Si (Igual(a,c) o Igual(a,d) o Igual (b,c) o Igual(b,d))
                devolver FALSE;
        Sino Si (Colineal(a,c,d) && Colineal (b,c,d) &&
                (ContienePunto(a,b,c) || ContienePunto(a,b,d)))
                        devolver TRUE
                Sino devolver (Xor(Izquierda(a,b,c), Izquierda(a,b,d)) &&
                                Xor(Izquierda(c,d,a), Izquierda(c,d,b));
```

Otro problema que se observó durante el desarrollo, está vez más relacionada con Swing que con algoritmos geométricos, fue el del **redibujado de los botones** del panel de la izquierda. Como se puede observar al usar la interfaz, cada vez que se selecciona un objeto geométrico de un determinado tipo se muestran en dicho panel botones para realizar operaciones con dicho objeto, dependiendo el tipo de botones del tipo de objeto geométrico. Esto en la implementación implicaba eliminar botones del panel llamado *herramientas*, añadir los nuevos botones, y ejecutar un *herramientas.repaint()*. Sin embargo, el funcionamiento no era el correcto, pues el panel de botones no se actualizaba. El problema se solucionó ejecutando un *herramientas.revalidate()* antes del mencionado *herramientas.repaint()*, lo cual solo se consiguió por ensayo y error probando todos los métodos del API de Java, pues es algo que no se encontraba documentado en ningún lugar.

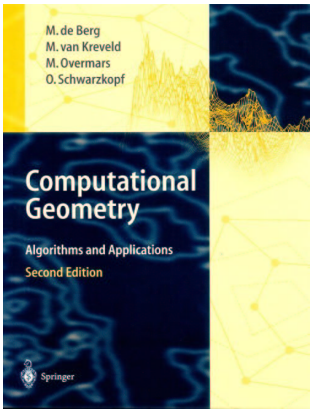
5. Bibliografía

A continuación se enumeran las fuentes de información utilizadas durante la realización del presente trabajo, tanto libros como direcciones de Internet.

5.1. Libros

- **Curso de Java**
  Patrick Niemeyer, Joshua Peck
  768 Páginas
  O'Reilly, Anaya

- **Computational Geomtry: Algorithms and Applications**
  Mark De Berg, Mark Van Kreveld, Mark Overmars, O. Schwarzkopf
  375 páginas
  Springer Verlag



5.2. URLs

- http://java.sun.com/docs/books/tutorial/
  The Java Tutorial: tutorial en el uso del lenguaje Java cubriendo los aspectos básicos del lenguaje, proporcionado por Sun.
- http://java.sun.com/j2se/1.3/docs/api/index.html
  API de la plataforma Java 2, para la versión 1.3 de jdk, con la descripción de todos los métodos pertenecientes a todas las clases de la librería de clases de la plataforma, en formato Javadoc.
- http://www.cgal.org/Manual/doc_html/index.html
  Documentación de la librería *CGAL*, incluyendo una guía de instalación, tutoriales, y descripción de los componentes de cada una de las partes de la librería (núcleo 2D y 3D, librería básica, etc.).
- http://www.dccia.ua.es/dccia/inf/asignaturas/RG/
  Página de la asignatura *Razonamiento Geométrico* de la *Universidad de Alicante*, desde donde se pueden obtener las transparencias utilizadas en la teoría de la asignatura y que sirvieron de base para la implementación de diversos métodos de *JavaRG*.