Practica de curso: Flying Balls

21707 - Programació II - Grupo 1

Curso 2018-2019

Palmer Pérez, Rubén

rpp776@id.uib.cat

43474448D

Dr. Miguel Mascaró Oliver

Video Explicativo

GitHub

Índice

1 Introducción

En está práctica se nos plantea diseñar una aplicación que **permita visualizar el movimiento de círculos** en una ventana. está simulación ha sido construida bajo el compilador NetBeans y el lenguaje de programación Java. El proyecto *Flying Balls* es una simulación que utiliza las librerías de **Swing** y **AWT** para renderizar todos los elementos dentro de la simualación y crear una **IGU** para poder interactuar con ella.
Para está práctica fue necesario implementar un programa principal que contenga la definición y comportamiento de la interfaz y la llamada a un método que pone en marcha la simulación. Además de eso, las siguientes clases:

- **Panel -** Panel gráfico que contiene la definición de una colección de círculos y los datos provenientes de la interfaz gráfica.
- **Ball -** Caracterizado por su tamaño, forma, color, posición, velocidad y aceleración.
- **Vector -** Caracterizado por dos coordenadas X e Y, contiene las principales operaciones que se pueden hacer con vectores.

2 Diseño

A continuación se describirá el diseño descendente que ha conducido a la solución propuestá, explicando las clases y métodos. Primero se describirá el diseño con una descripción poco detallada de las clases y métodos más relevantes, al final de este apartado se podrá encontrar una descripción mas precisa.

2.1 Acercamiento al problema

Se nos presentó el desafío de crear una simulación en Java utilizando sus librerías graficas, principalmente **Swing** y **AWT**. En la ventana de la aplicación hay diferenciadas **dos partes**:
Un **panel** donde se ven los círculos en movimiento y la **IGU** con los siguientes elementos:

- Un **campo de texto editable** que permite insertar el número de objetos que se quiere simular.
- Dos **cajas de verificación**:
  - **Follow Mouse -** En estádo activado, los objetos dentro de la simulación serán atraidos por la posición del ratón. En caso contrario, serán atraidos por el limite inferior de la ventana.
  - **With Walls -** En estádo activado, los objetos dentro de la simulación chocarán con las paredes. En caso contrario, las atravesarán y aparecerán en el lado opuesto.

Además de eso, se debían crear las clases mencionadas en la Introducción siendo:

- Un **Panel** donde ocurriria toda la simulación.
- Una clase **Bola**, que efectivamente es nuestro objeto a simular
- Una clase **Vector** que nos sirvira para calcular posición, velocidad y acceleración

Así pues, lo primero a hacer serían las cosas que no dependiesen de nada. En este caso, la única clase que cumple eso es la clase **Vector**, ya que debería comportarse igual en todos los programas.

2.1.1 Panel de simulación

En está clase no deberíamos tener muchos métodos ya que panel se centra en llamar a los métodos necesarios para efectuar una simulación. Para eso fue necesaria la implementación de los siguientes métodos principales:

- Un método **"actualizar"**, que actualice la posición de todos y cada uno de los objetos dentro que contiene este panel
- Un método **"pintar"**, que nos permita visualizar cada uno de los objetos del panel

Con estos dos métodos principales seria posible, dado un conjunto de objetos, visualizar en cada frame cada uno de estos objetos. Sin embargo no tenemos actualmente ninguna manera de crear un conjunto de estos objetos sin hacerlo a mano, por eso se implemento un método adicional:

- **"Popular":** Permite crear un conjunto de objetos. Este conjunto será el simulado dentro del panel

Con estos métodos, se puede crear una simulación. Sin embargo se preciso crear estos *"objetos"* que queremos simular.

2.1.2 Objeto a simular

Nos enfrentamos al siguiente problema, la clase que definirá lo que nosotros estemos simulando en el panel. Lo primero que me pregunté al crear un objeto a simular fueron cuales son las posibles acciones de este, que es lo que puede o no puede hacer. Fue inmediato, despues de está pregunta, cuales eran los métodos que debía crear para que este objeto fuese simulable:

- Un método **"mover"**, que permitiría al objeto cambiar su posición dependiendo de su velocidad.
- Un método **"rebotar"**, ya que existe la posibilidad que este objeto pueda rebotar en las paredes.
- Un método **"atravesar"**, ya que existe la posibilidad de que este objeto pueda atravesar las paredes.
- un método **"pintar"**, que permitiría poder visualizar el objeto en la posición en la que se encuentre.
- Un método **"color"**, ya que uno de los parámetros es color, y no era de mi agrado encontrar que todos y cada uno de los objetos fuesen del mismo color.

3 Clases y métodos

A continuación serán explicados de manera concisa cada una de las clases y sus métodos para una visión mas detallada.

3.1 Clase Main

Encargada de la administración de la ventana como tal. En ella se encuentra la posición de la **IGU** al igual que sus controladores.

3.1.1 initcomponents

En este método se encuentra todo el diseño de la ventana en el que incluye:

- Posición del panel.
- Posición de todos los elementos de la **IGU**.
- Controlador de la caja de texto.
- Controlador de la checkbox "With Walls".
- Controlador de la checkbox "Follow Mouse".

De estos elementos, nos centraremos en los tres últimos.

3.1.2 ballNumberActionPerformed

Función: Controlar la cantidad de objetos presentes en todo momento en la simulación

Para su funcion se aprovechó de la clase Integer y su método parseInt() y el error NumberFormatException para detectar si el usuario escribía un valor completamente numérico o no. Creando un Try-Catch podiamos solucionar esto. Si el usuario ponía un valor no numérico, surgiría la excepción y se cambiaría un valor booleano "numérico" a false. En caso contrario, podíamos obtener el número deseado. Una vez salimos del Try-Catch comprobaríamos si el valor era numérico mediante la variable booleana "numeric". En caso de ser true, popularíamos el ArrayList de objetos del Panel con la cantidad deseada por el usuario. Contrariamente, ignorariamos está "populacion" y como resultado no se haría nada.

3.1.3 setWallsActionPerformed

Función: Controlar la intereacción con los bordes.
Según el valor de la checkbox, se copiaría su estádo a una variable booleana correspondiente de la clase Panel, en este caso wallMode.

3.1.4 followMouseActionPerformed

Función: Controlar la dirección de la gravedad.
Según el valor de la checkbox, se copiaría su estádo a una variable booleana correspondiente de la clase Panel, en este caso mouseMode.

3.2 Clase Panel

está calse se dedica a gestionar la simulacíon como tal. En ella encontramos las siguientes variables:

- **public static final Integer Height -** Toma como valor la anchura del JPanel.
- **public static final Integer Width -** Toma como valor la altura del JPanel.
- **private static final Vector G -** Vector gravedad que permite una acceleración constante en el eje y.
- **private ArrayList<Ball> balls -** ArrayList que contiene todos los objetos a simular dentro del JPanel.
- Las siguientes variables permiten la comunicación de la **IGU** del JFrame con el JPanel.
  - **private Integer arrayLength -** Presenta la cantidad de objetos activos en la simulación.
  - **private Boolean mouseMode -** Permite cambiar la dirección del vector gravedad entre siguiendo al ratón o hacia abajo.
  - **private Boolean wallMode -** Permite cambiar la interacción con los bordes del JPanel del los objetos simulados entre rebotar o atravesarlas

A continuación se presentarán los métodos de la clase:

3.2.1 public void populate

Función: Crear un nuevo ArrayList con la cantidad de objetos especificado
Primero, mediante el método clear(), se eliminan todos los posibles objetos presentes en el ArrayList. Seguido de eso, se añadiran X cantidad de objetos "Ball" mediante un for que depende de la variable ***arrayLength***. Además de eso, se les proporciona una posición aleatoria mediante el método random() de la clase Math.

3.2.2 public void start

Función: Da comienzo a la simulación
Dentro de un while infinito, se llaman a los métodos **update** y **repaint**. Para que la visualización sea adecuada Después de cada iteración hay programado un retraso, que se consigue mediante el método Thread.sleep().

3.2.3 public void paintComponent

Función: Renderizar todos y cada uno de los objetos presentes en el **balls**
Mediante un for que depende del tamaño del **balls**, se llama al método **paint** de cada uno de sus objetos.

3.2.4 public void update

Función: Actualizar la posición de cada uno de los objetos dentro de **balls**
Mediante un if, se revisa la variable ***mouseMode*** para determinar cual será el vector gravedad.En ambos casos, se utilizara el método **setAcceleration** de la clase **Ball**. En el caso de que ***mouseMode*** sea true, se pondrá como parámetro del método **setAcceleration** el vector que va desde la bola a la posición del ratón. Para conseguir ese vector, se llama al método **MouseVector**. En el caso de que sea falso, se pone como parámetro la variable ***G***.
Una vez determinado el tipo de gravedad que los objetos sufrirán, se llamará al método **move** de la clase **Ball**, que calcula la nueva posición del objeto.
Seguidamente se comprueba la variable ***wallMode***, en el caso de que sea true se llamará al método **rebound** de la clase **Ball**, que permite la interacción "rebote" con los bordes. En caso contrario, se llamaría al método **tp** de la misma clase que activa la interacción de "atravesar" los bordes. Todos estos métodos están dentro de un for que depende del tamaño de ***balls*** para llamarlos en cada uno de sus objetos.

3.2.5 public Vector MouseVector (int i)

Función: Obtener el vector que va desde un objeto de ***balls*** al puntero del ratón
Primero se obtiene el punto en el que está el puntero del ratón con los métodos **getPointerInfo** y **getLocation**. Las coordenadas de este punto sin embargo, son respectivas a la pantalla en la que se encuentre la aplicacion de Java. Para arreglar este problema se utiliza el método **convertPoint-FromScreen** de la clase **SwingUtilities** para cambiar las coordenadas de ese punto y hacer que sean respecto a nuestra ventana. Mediante los métodos **getX** y **getY** obtenemos las coordenadas individuales del punto y creamos un nuevo vector ***Mouse*** que se definiría como el vector que va desde la esquina superior izquierda de nuestra ventana hasta el puntero del ratón.
Por último restámos el vector ***position*** del objeto cuya posición sea la ***i*** en ***balls*** y ***Mouse***. A este vector, se le aplica el método **Uni**, que lo convierte en unitario y multiplicamos sus componentes por el módulo del vector ***G*** para obtener un vector Gravedad de la misma intensidad. Este último vector es devuelto.

3.2.6 public void setArrayLength(Integer arrayLength)

Función: Setter de ***arrayLength***

3.2.7 public void setMouseMode(Boolean mouseMode)

Función: Setter de ***mouseMode***

3.2.8 public void setWallMode(Boolean wallMode)

Función: Setter de ***wallMode***

3.3 Clase Ball

está clase se dedica a definir y gestionar todos los métodos y variables de los objetos a simular. Dentro de ella se encuentran las siguientes variables:

- **private static final Vector Terminal -** Vector que determina la velocidad máxima que este objeto puede tener
- **private Vector position -** Vector posición del objeto.
- **private Vector velocity -** Vector velocidad del objeto.
- **private Vector acceleration -** Vector aceleración del objeto.
- **public final static Integer radius -** Tamaño del objeto, toma el valor 45 de base.
- **private final Color color -** Color del objeto.
- **private Ellipse2D shape -** Forma del objeto.

A continuación se presentarán los métodos de la clase:

3.3.1 public static Color rndColor

Función: Crear un color RGB aleatorio
Utilizando el método **nextInt** de la clase **Random**, creamos tres variables ***R***, ***G***, ***B*** y devolvemos un nuevo color creado con estás tres variables.

3.3.2 public void rebound()

Función: Cambiar los sentidos de las componentes de ***velocity*** para obtener el efecto de rebote tanto en el eje X como en el eje Y
En un if comprobamos si el objeto está dentro del JPanel, en caso de que no lo esté se multiplica la componente X o Y de ***velocity*** por -1. En el caso de querer rebotar en la parte inferior del JPanel, se multiplicaria la componente Y, pues es un rebote vertical.

3.3.3 public void tp()

Función: Cambiar ***position*** para obtener el efecto de atravesar paredes tanto en el eje X como en el eje Y

En un if comprobamos si el objeto está fuera del JPanel, en el caso de que lo esté primero se debe comprobar desde que pared está saliendo. Una vez se comprueba que está fuera del JPanel y se conoce la pared por la que ha salido, se cambia el vector ***posición*** para que aparezca en la pared contraria

3.3.4 public void paint(Graphics g)

Función: Renderizar el objeto

Primero definimos la forma que tiene el objeto. Seguidamente, definimos los parámetros para los bordes y despues de esto rellenamos la forma con el color ***color***

3.3.5 public void move()

Función: Actualiza ***velocity*** y ***position*** para obtener un movimiento natural. Además de esto comprueba, y en caso de superarla actualizarla, si las componentes de ***velocity*** superan las de ***Terminal***.

Se suma ***acceleration*** a ***velocity***. Debido a que existe una velocidad terminal por objeto, se revisa si la componente X o Y de ***velocity*** superan las respectivas componentes de ***Terminal***, en caso de que las superen, dicha componente se capa. De está manera obtenemos la implementación de una velocidad Terminal. Una vez hecho los calculos con la velocidad, se suma ***velocity*** a ***position***, de está manera conseguimos que, al renderizar, aparezca el efecto de movimiento.

3.3.6 public Vector getPosition()

Función: Getter de ***position***

3.3.7 public void setAcceleration(Vector acceleration)

Función: Setter de ***acceleration***

3.4 Clase Vector

está clase se dedica a gestionar y definir el concepto de vector. Dentro de ella se pueden encontrar las siguientes variables:

- **public Double[] vector -** Define un vector como un conjunto ordenado de valores. La cantidad de valores expresa la dimensión del vector.

A continuación se presentarán los métodos de la clase. Cabe decir que muchos de estos métodos tienen una versión estática que no van a ser mostrados aquí ya que tienen la misma funcionalidad. :

3.4.1 public Double module()

Función: Devuelve el módulo del vector.
Mediante un for que depende de la dimensión del vector, se van sumando cada una de las componentes multiplicadas por si misma a un valor auxiliar que es devuelto.

3.4.2 public void AddVector(Vector Vx)

Función: Permite sumar las componentes de dos vectores.
Mediante un for que depende de la dimensión del vector se van sumando las componentes de cada uno de los vectores.

3.4.3 public void AddVector(Vector Vx)

Función: Permite sumar las componentes de dos vectores.
Mediante un for que depende de la dimensión del vector se van sumando las componentes de cada uno de los vectores.

3.4.4 public void SubVector(Vector Vx)

Función: Permite restár las componentes de dos vectores.
Mediante un for que depende de la dimensión del vector se van restándo las componentes de cada uno de los vectores.

3.4.5 public void EMultVector(Double y)

Función: Permite multiplicar las componentes de un vector por un valor escalar $\boldsymbol{y}$.
Mediante un for que depende de la dimensión del vector se multiplica por el valor $\boldsymbol{y}$. cada una de las componentes del vector.

3.4.6 public void EDivVector(Double y)

Función: Permite dividir las componentes de un vector por un valor escalar $\boldsymbol{y}$.
Mediante un for que depende de la dimensión del vector se divide por el valor $\boldsymbol{y}$. cada una de las componentes del vector.

3.4.7 public Vector Uni()

Función: Devuelve el vector unitario de un vector.
Mediante un for que depende de la dimensión del vector se divide cada componente del vector por su módulo y se van depositando en un vector auxiliar $\boldsymbol{z}$. Este vector es devuelto al final del método.

3.4.8 public Double EscVector(Vector Vx)

Función: Devuelve el producto escalar entre el vector y un vector ***Vx*** pasado por parámetro.
Mediante un for que depende de la dimensión del vector se multiplican cada una de las componentes de ambos vectores y se van sumando a un valor auxiliar ***sum***. Este valor sum es devuelto al final del método.

3.4.9 public void limit(Double max)

Función: Si la magnitud es mayor que un valor máximo ***max***, normalizar el vector y hacer la multiplicación escalar por ***max***.
Mediante un if, se comprueba que el módulo del vector sea mayor que ***max***. En caso de que sea true, se llama al método **Uni** seguido del método **EMultVector(max)**

3.4.10 public Integer getDim()

Función: Getter de la dimensión de ***vector***

3.4.11 public String toString()

Función: Convertir un vector a String.
Mediante un for que depende de la dimensión del vector se van añadiendo a un String las componentes individuales

4 Conclusiones

Para realizar esta práctica ha sido necesario primero comprender el funcionamiento y comportamiento de las interfaces gráficas que nos proporciona Java ademas de la idea abstracta de una simulación. Entonces, entendiendo como debería operar y cómo funciona, he sido capaces de simular su funcionalidad utilizando solo las herramientasqeu me ha proporcionado Java y StackOverflow. Algunos de los puntos mas difíciles los encontre a la hora de diseñar la simulación, principalmente la abstracción del JPanel, entendiendo que es lo que debería estar ahi y que lo que no debería.

La implementación de gravedad hacia el ratón tambien fué un desafío, pues sabía lo que tenía que hacer pero surgian problemas a la hora de implementar. El principal problema que tuve fue la intensidad del vector, pues desconocía que el punto que se creaba era respecto a la pantalla y no a la vantana de la aplicación. Solucionarlo fue en parte suerte, pues yo trabajo tanto en un Pc de sobre mesa y portátil. Al pasar el proyecto entre ordenadores noté una leve diferencia y una pequeña busqueda en StackOverflow me solucionó el problema que tenía al igual de indicarme mejores maneras para afrontar el problema.

Para finalizar, debo decir que la programación es una de las dos tres pasiones que tengo en mi vida y no podria sentirme más identificado con una frase que encontré en Reddit, desesperado por soluciones para mi código:

“Give a man a program, frustrate him for a day. Teach a man to program, frustrate him for a lifetime.”