

PRACTICA 19: ASTEROIDES VIII

Parte I: Introduciendo movimiento en Asteroides

Paso 1. Comienza declarando las siguientes variables en la clase `VistaJuego`:

```
public class VistaJuego extends View {
    // NAVE
    private Grafico nave; // Gráfico de la nave
    private int giroNave; // Incremento de dirección
    private float aceleracionNave; // aumento de velocidad
    // Incremento estándar de giro y aceleración
    private static final int PASO_GIRO_NAVE = 5;
    private static final float PASO_ACCELERACION_NAVE = 0.5f;

    // ASTEROIDES
    private Vector<Grafico> Asteroides; // Vector con los Asteroides
    private int numAsteroides = 5; // Número inicial de asteroides
    private int numFragmentos = 3; // Fragmentos en que se divide

    // THREAD Y TIEMPO
    // Thread encargado de procesar el juego
    private ThreadJuego thread = new ThreadJuego();
    // Cada cuanto queremos procesar cambios (ms)
    private static int PERIODO_PROCESO = 50;
    // Cuando se realizó el último proceso
    private long ultimoProceso = 0;
}
```

Paso 2. La animación del juego la llevará a cabo con el método `actualizaFisica()` que será ejecutado a intervalos regulares definidos por la constante `PERIODO_PROCESO`. Esta constante ha sido inicializada a 50 ms. En `ultimoProceso` se almacena el instante en que se llamó por última vez a `actualizaFisica()`.

Paso 3. Copia el siguiente método dentro de la clase `VistaJuego`:

```
protected void actualizaFisica(){
    long ahora = System.currentTimeMillis();
    //No hagas nada si el periodo de proceso no se ha cumplido"
    if (ultimoProceso + PERIODO_PROCESO > ahora){
        return;
    }
    // Para una ejecución en tiempo real calculamos retardo
    double retardo = (ahora - ultimoProceso) / PERIODO_PROCESO;
    ultimoProceso = ahora; // Para la próxima vez
    // Actualizamos velocidad y dirección de la nave a partir de
    // giroNave y aceleracionNave (según la entrada del jugador)
    nave.setAngulo((int) (nave.getAngulo() + giroNave * retardo));
    double nIncX = nave.getIncX() + aceleracionNave *
        Math.cos(Math.toRadians(nave.getAngulo())) * retardo;
    double nIncY = nave.getIncY() + aceleracionNave *
        Math.sin(Math.toRadians(nave.getAngulo())) * retardo;
    // Actualizamos si el módulo de la velocidad no excede el máximo
    //if (Math.hypot(nIncX,nIncY) <= MAX_VELOCIDAD_NAVE){
    if (Math.hypot(nIncX,nIncY) <= Grafico.getMaxVelocidad()){ //MAX_VELOCIDAD_NAVE){
        nave.setIncX(nIncX);
        nave.setIncY(nIncY);
    }
    // Actualizamos posiciones X e Y
    nave.incrementaPos(retardo);
    for (Grafico asteroide : Asteroides) {
        asteroide.incrementaPos(retardo);
    }
}
```

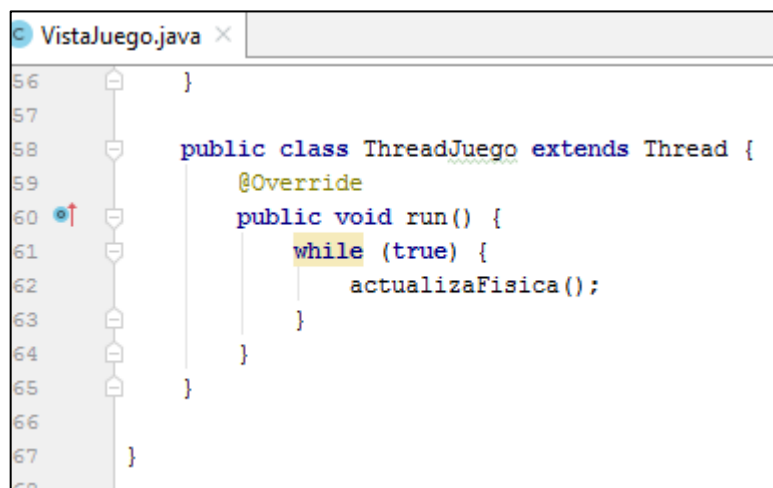
Como veremos a continuación este método será llamado de forma continua. Como queremos desplazar los gráficos cada **PERIODO_PROCESO** milisegundos, verificamos si ya ha pasado este tiempo desde la última vez que se ejecutó (**ultimoProceso**).

Como también es posible que el sistema esté ocupado y no nos haya podido llamar hasta un tiempo superior a **PERIODO_PROCESO**, vamos a calcular el factor de retardo en función del tiempo adicional que haya pasado. Si por ejemplo, desde la última llamada ha pasado dos veces **PERIODO_PROCESO**, la variable **retardo** ha de valer 2. Lo que significará que los gráficos han de desplazarse el doble que en circunstancias normales. De esta forma conseguiremos un desplazamiento continuo en tiempo real.

A continuación, se actualizan las variables que controlan la aceleración y cambios de dirección de la nave. Se consiguen por medio de las variables **aceleracionNave** y **giroNave**. En el siguiente capítulo modificaremos estas variables para que el jugador pueda pilotar la nave. A partir de estas variables se obtiene una nueva velocidad de la nave, descompuesta en sus componentes x e y. Si el módulo de estos componentes es mayor que la velocidad máxima permitida, no se actualizará la velocidad.

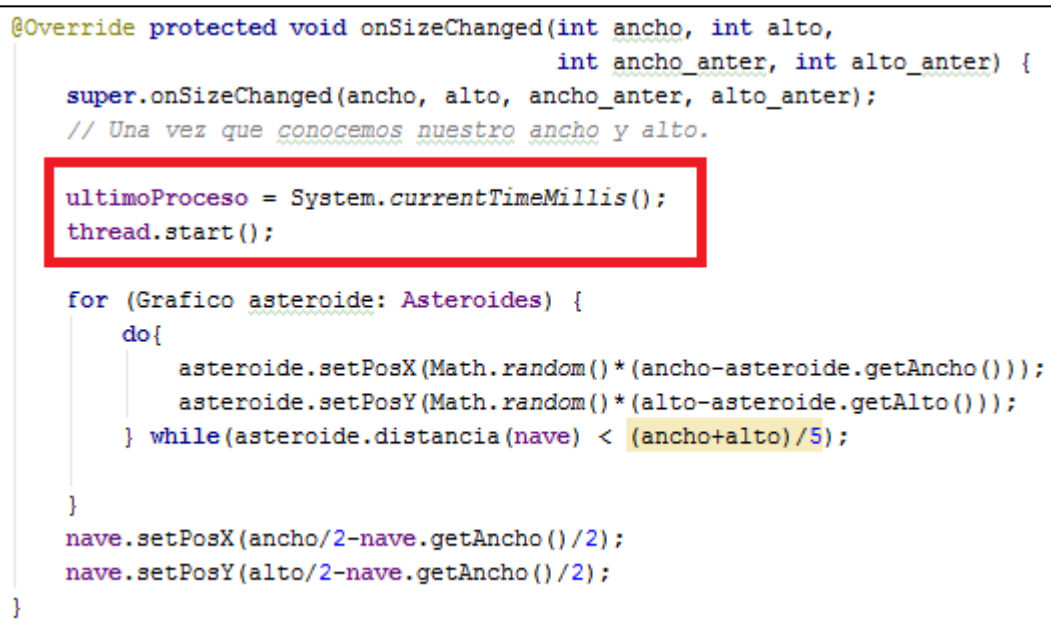
Finalmente se actualizan las posiciones de todos los gráficos (nave y asteroides) a partir de sus velocidades. Esto se consigue llamando al método **incrementaPos()** definido en la clase **Grafico**.

Paso 4. Ahora necesitamos que esta función sea llamada continuamente, para lo que utilizaremos un **Thread**. Crea la siguiente clase dentro de la clase **VistaJuego**:



```
56     }
57
58     public class ThreadJuego extends Thread {
59         @Override
60         public void run() {
61             while (true) {
62                 actualizaFisica();
63             }
64         }
65     }
66
67 }
```

Paso 5. Introduce estas líneas al final del método **onSizeChanged()**:



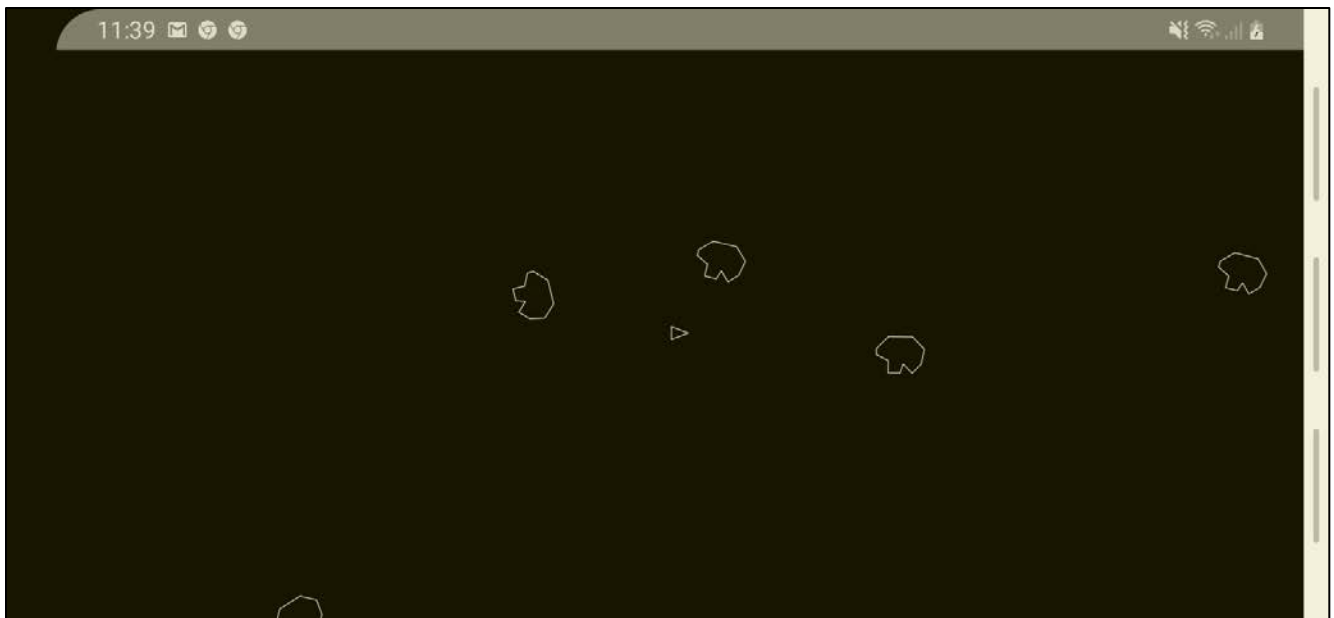
```
@Override protected void onSizeChanged(int ancho, int alto,
                                         int ancho_antes, int alto_antes) {
    super.onSizeChanged(ancho, alto, ancho_antes, alto_antes);
    // Una vez que conocemos nuestro ancho y alto.

    ultimoProceso = System.currentTimeMillis();
    thread.start();

    for (Grafico asteroide: Asteroides) {
        do{
            asteroide.setPosX(Math.random() * (ancho-asteroide.getAncho()));
            asteroide.setPosY(Math.random() * (alto-asteroide.getAlto()));
        } while(asteroide.distancia(nave) < (ancho+alto)/5);
    }
    nave.setPosX(ancho/2-nave.getAncho()/2);
    nave.setPosY(alto/2-nave.getAlto()/2);
}
```

Esto ocasionará que se llame al método **run()** del hilo de ejecución. Este método es un bucle infinito que continuamente llama al **actualizaFisica()**.

Paso 6. Ejecuta la aplicación y observa como el juego cobra vida.



El trabajo con hilos de ejecución es especialmente delicado. Como veremos en próximos capítulos, este código nos va a ocasionar varios quebraderos de cabeza. Un problema es que seguirá ejecutándose aunque nuestra aplicación esté en segundo plano. Veremos cómo detener el hilo de ejecución, cuando estudiemos el ciclo de vida de una actividad. (**NOTA:** Si ejecutas el programa en el terminal real y detectas que este funciona más lentamente, puede ser buena idea detener la aplicación). Un segundo problema, aparecerá cuando dos hilos de ejecución traten de acceder a la misma variable a la vez. También se resolverá más adelante.

Parte II: Introduciendo secciones críticas en Java (synchronized)

Cuando se realiza una aplicación que ejecuta varios hilos de ejecución hay que prestar un especial cuidado a que ambos hilos pueden acceder de forma simultánea a los datos. Cuando se limitan a leer las variables, no suele haber problemas. El problema aparece cuando un hilo esté modificando algún dato y justo en este instante se pasa a ejecutar un segundo hilo que ha de leer estos datos. Este segundo hilo va a encontrar unos datos a mitad de modificar, lo que posiblemente cause errores en su interpretación. El método más común para evitar que dos hilos accedan al mismo tiempo a un recurso es el de la exclusión mutua. En Java que se consigue utilizando la palabra reservada **synchronized**.

Paso 1. Introduce la palabra reservada **synchronized** delante del método `onDraw()` y `actualizaFisica()`. De esta forma se evita que cuando `actualizaFisica()` esté modificando alguno de los valores de la nave o los asteroides en método `onDraw()` acceda a estos valores.

```
@Override protected synchronized void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    nave.dibujaGrafico(canvas);

    for (Grafico asteroide: Asteroides) {
        asteroide.dibujaGrafico(canvas);
    }
}

protected synchronized void actualizaFisica(){
    long ahora = System.currentTimeMillis();
    //No hagas nada si el periodo de proceso no se ha cumplido"
    if (ultimoProceso + PERIODO_PROCESO > ahora){
        return;
    }
}
```

Nota sobre Java: La palabra clave **synchronized** permite definir una sección crítica en Java. Expliquemos en qué consiste: Cada vez que un hilo de ejecución (thread) entra en un método o bloque de instrucciones marcado con **synchronized** se pregunta al objeto si ya hay algún otro thread que haya entrado en la sección crítica de ese objeto. La sección crítica está formada por todos los bloques de instrucciones marcados con **synchronized**. Si nadie ha entrado en la sección crítica, se entrará normalmente. Si ya hay otro thread dentro, entonces el thread actual es suspendido y ha de esperar hasta que la sección crítica quede libre. Esto ocurrirá cuando el thread que está dentro de la sección crítica salga.

Dos matizaciones importantes: La primera es que la sección crítica se define a nivel de objeto no de clase. Es decir, cada objeto instanciado no influyen en las secciones críticas de otros objetos. En segundo lugar, solo se define una sección crítica por clase. Aunque se haya utilizado **synchronized** en varios métodos, realmente solo hay una sección crítica.

Parte III: Manejo de la nave con el teclado

Veamos cómo podemos utilizar un manejador de eventos de teclado para maniobrar la nave de Asteroides:

Paso 1. Abre el proyecto `Asteroides`.

Paso 2. Inserta este código en la clase `VistaJuego`.

```

@Override
public boolean onKeyDown(int codigoTecla, KeyEvent evento) {
    super.onKeyDown(codigoTecla, evento);
    // Suponemos que vamos a procesar la pulsación
    boolean procesada = true;
    switch (codigoTecla) {
        case KeyEvent.KEYCODE_DPAD_UP:
            aceleracionNave = +PASO_ACELERACION_NAVE;
            break;
        case KeyEvent.KEYCODE_DPAD_LEFT:
            giroNave = -PASO_GIRO_NAVE;
            break;
        case KeyEvent.KEYCODE_DPAD_RIGHT:
            giroNave = +PASO_GIRO_NAVE;
            break;
        case KeyEvent.KEYCODE_DPAD_CENTER:
        case KeyEvent.KEYCODE_ENTER:
            ActivaMisil();
            break;
        default: // Si estamos aquí, no hay pulsación que nos interese
            procesada = false;
            break;
    }
    return procesada;
}

```

Paso 3. Cada vez que se pulse una tecla se realizará una llamada al método `onKeyDown()` con los siguientes parámetros: El primero es un entero que nos identifica el código de la tecla pulsada. El segundo es de la clase `KeyEvent` nos permite obtener información adicional sobre el evento, como por ejemplo, cuando se produjo. Este método ha de devolver un valor booleano, verdadero, si consideramos que la pulsación ha sido procesada por nuestro código, y falso, si queremos que otro manejador de evento siguiente al nuestro reciba la pulsación.

Paso 4. Antes de ponerlo en marcha comenta la llamada a `ActivaMisil()`, dado que esta función aún no está implementada.

```

case KeyEvent.KEYCODE_ENTER:
    //ActivaMisil();
    break;

```

Paso 5. Verifica si funciona correctamente.

Desde el simulador no se puede probar. No funciona el teclado desde el simulador. Habría que probarlo en una tableta.

NOTA: Para poder recoger eventos de teclado desde una vista es necesario que esta tenga el foco y para que esto sea posible verifica que tiene la propiedad `focusable="true"`



Efectivamente ya lo tiene activado en juego.xml

Paso 6. El ejercicio anterior no funciona de forma satisfactoria. Cuando pulsamos una tecla para girar la nave se pone a girar pero ya no hay manera de pararla. El manejador de eventos `onKeyDown` solo se activa cuando se pulsa una tecla, pero no cuando se suelta.

Trata de escribir el manejador de eventos `onKeyUp` para que la nave atienda a las órdenes de forma correcta. Puedes partir del siguiente código:

```
public boolean onKeyUp(int codigoTecla, KeyEvent evento) {
    super.onKeyUp(codigoTecla, evento);
    // Suponemos que vamos a procesar la pulsación
    boolean procesada = true;
    switch (codigoTecla) {
        case KeyEvent.KEYCODE_DPAD_UP:
            aceleracionNave = 0;
            break;
        case KeyEvent.KEYCODE_DPAD_LEFT:
        case KeyEvent.KEYCODE_DPAD_RIGHT:
            giroNave = 0;
            break;
        default:
            // Si estamos aquí, no hay pulsación que nos interese
            procesada = false;
            break;
    }
    return procesada;
}
```

Parte IV: Manejo de la nave con la pantalla táctil

Veamos cómo podemos utilizar un manejador de eventos de la pantalla táctil para maniobrar la nave de Asteroides. El código que se muestra permite manejar la nave de la siguiente forma: un desplazamiento del dedo horizontalmente hace girar la nave, un desplazamiento vertical produce una aceleración y, si al soltar la pulsación no hay movimiento, se provoca un disparo.

Paso 1. Abre el proyecto Asteroides.

Paso 2. Inserta este código en la clase `VistaJuego`.

```
public class VistaJuego extends View {

    private float mX=0, mY=0;
    private boolean disparo=false;

    @Override
    public boolean onTouchEvent (MotionEvent event) {
        super.onTouchEvent(event);
        float x = event.getX();
        float y = event.getY();
        switch (event.getAction()) {
            case MotionEvent.ACTION_DOWN:
                disparo=true;
                break;
            case MotionEvent.ACTION_MOVE:
                float dx = Math.abs(x - mX);
                float dy = Math.abs(y - mY);
                if (dy<6 && dx>6){ //Desplazamiento horizontal produce giro
                    giroNave = Math.round((x - mX) / 2);
                    disparo = false;
                } else if (dx<6 && dy>6){ //Desplazamiento vertical produce aceleracion
                    aceleracionNave = Math.round((mY - y) / 25);
                    disparo = false;
                }
                break;
            case MotionEvent.ACTION_UP: //Pulsacion sin movimiento, provoca un disparo
                giroNave = 0;
                aceleracionNave = 0;
                if (disparo){
                    ActivaMisil();
                }
                break;
        }
        mX=x; mY=y;
        return true;
    }
}
```

Paso 3. Las variables globales `mX` y `mY` van a ser utilizadas para recordar las coordenadas del último evento. Comparándolas con las coordenadas actuales (`x`, `y`) podremos verificar si se trata de un desplazamiento horizontal o vertical. Por otra parte, la variable `disparo` es activada cada vez que comienza una pulsación (`ACTION_DOWN`). Si esta pulsación es continuada con un desplazamiento horizontal o vertical, `disparo` es desactivado. Si por el contrario, se levanta el dedo (`ACTION_UP`) sin haberse producido estos desplazamientos, `disparo` no estará desactivado y se llamará a `ActivaMisil()`.

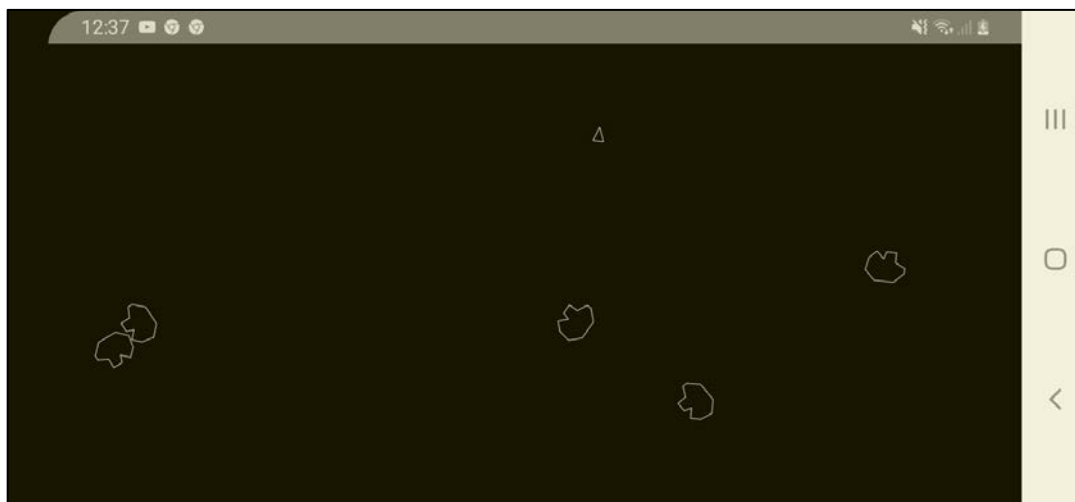
Paso 4. Antes de ponerlo en marcha comenta la llamada a `ActivaMisil()`, dado que esta función aún no está implementada.


```

case MotionEvent.ACTION_UP:    //Pulsacion sin movimiento, provoca un disparo
    giroNave = 0;
    aceleracionNave = 0;
    if (disparo){
        //ActivaMisil();
    }
    break;

```

Paso 5. Verifica si funciona correctamente.



Según la disposición de la pantalla anterior, los movimientos verticales con el dedo son los que hacen girar la nave y los movimientos horizontales son los que hacen que la nave coja velocidad.

Paso 6. Modifica los parámetros de ajuste (<6,>6, /2, /25), para que se adapten de forma adecuada a tu terminal.

Los parámetros <6 y >6 son para detección de movimiento horizontal o vertical.

/2 y /25 son escalados para dar más o menos giro o velocidad respectivamente.

En principio de adaptan correctamente a la forma del terminal

Paso 7. En el juego original podíamos acelerar pero no decelerar. Si queríamos detener la nave teníamos que dar un giro de 180 grados y acelerar lo justo. Modifica el código anterior para que no sea posible decelerar.

Efectivamente podemos ver que si hacemos un giro horizontal en la pantalla:

- De izquierda a derecha → la nave acelera (en la dirección que apunta)
- De derecha a izquierda → la nave decelera (en la dirección que apunta)

Para quitar la deceleración de manera que se ajuste más al juego original, sólo hay que poner en valor absoluto el valor conseguido de la variables aceleracionNave:

```

case MotionEvent.ACTION_MOVE:
    float dx = Math.abs(x - mX);
    float dy = Math.abs(y - mY);
    if (dy<6 && dx>6){          //Desplazamiento horizontal produce giro
        giroNave = Math.round((x - mX) / 2);
        disparo = false;
    } else if (dx<6 && dy>6){    //Desplazamiento vertical produce aceleracion
        aceleracionNave = Math.abs(Math.round((mY - y) / 25));
        disparo = false;
    }
    break;

```

Esto hace que se parezca más juego original, pero complica mucho más la navegación. Ahora para decelerar se debe girar la nave en sentido y acelerar.