UNIVERSIDAD NACIONAL
DE GENERAL SARMIENTO

PROGRAMACIÓN I

Trabajo Práctico Final

Integrantes

APELLIDO Y NOMBRE	LEGAJO	EMAIL
Pereira, Fabián	37.247.692/2015	fabianeze93@gmail.com
Sánchez, Matías Alejandro	38.391.082/2015	mattisanchez94@gmail.com
Tula, Ignacio Mariano	35.226.620/2014	itula@ungs.edu.ar

Versión 2

INTRODUCCIÓN:

Se desarrolló un videojuego inspirado en el antiguo juego “Donkey Kong”. Donde el jugador controla un personaje que debe desplazarse por el nivel hasta llegar a la posición del antagonista del juego, sin que ninguna de las dificultades y obstáculos móviles toquen al personaje.

Para ello se intentó que el apartado gráfico sea lo más fiel posible al juego original. Se han utilizado las herramientas propuestas por los métodos de la clase entorno. Principalmente las que permiten cargar una imagen (que puede ser estática o un gif animado).

Para la representación gráfica de objetos más sencillos se han utilizado las herramientas gráficas de dibujos (principalmente rectángulos y triángulos) que combinados entre ellos en posiciones y colores estratégicos permitieron simular vigas y escaleras.

De manera general, los métodos más importantes calculan las trayectorias y las posiciones de los objetos no controlados por el usuario.

Y en el caso particular del personaje, analizan diferentes escenarios posibles a la hora de permitirle al usuario o no ejecutar la acción requerida (no es posible saltar si se está cayendo por una cornisa, o posterior a un salto).

A su vez se detecta en cada instante de tiempo las posiciones extremas de cada objeto para conocer si hay colisiones que hagan perder al usuario.

DIFICULTADES:

Entre las dificultades que se hallaron fueron las acciones que realizaban acciones variadas a lo largo de un intervalo de tiempo.

Un ejemplo de ellas fué el salto, que en determinado momento se desplaza hacia arriba, hasta cierto punto donde cae. Pero la caída no debería estar limitada por una cantidad de tiempo sino que debería producirse hasta que en algún momento encuentre suelo.

Otra dificultad relacionada con el tiempo fué la producción del sonido que representan los pasos del personaje. El sonido no puede ejecutarse en cada tick, sino se produce un exceso de sonidos indistinguibles.

En todos los casos, fue necesaria la aplicación de una variable numérica que guardara el tick en el cual se produjo la última acción y prohibir a los métodos su ejecución hasta que no se dieran ciertas condiciones, entre ellas que hubiera una distancia temporal razonable.

Otra complejidad añadida fueron las colisiones del personaje con los barriles, donde los puntos de contacto pueden ser múltiples y provenir de una combinación de diferentes lados de los objetos estudiados.

Aquí la dificultad era matemática y eminentemente referida al resultados que arrojaban las diferentes inecuaciones de las distancias entre puntos “x” e “y” de cada extremo de cada objeto, pero a nivel informático realizable con condicionales encadenados o con varias condiciones.

Similar al problema anterior, lo fué la detección de cercanías del personaje con escaleras o los saltos de barriles, pero la complejidad nuevamente viene dada por la combinación de al menos dos puntos “x” por cada objeto y dos puntos “y” por cada objeto para conocer la verdadera cercanía y no obtener un falso positivo.

Un falso positivo de un salto de barril sería que uno de estos se encontrara en la misma posición “x” que el personaje pero en un piso inferior. Evidentemente, aquí no hay salto alguno.

MODIFICACIONES:

Por indicación de los docentes se realizaron las siguientes modificaciones al código fuente:

1. **Revisión completa de las variables de instancia de cada clase. No existe ninguna que no sea “`private`”.**

   En la anterior entrega existía algunas clases donde por error involuntario de este equipo, se omitió indicar que las variables sean privadas.

   De todas formas, ninguna clase accede a las variables de otra de forma directa, siempre que sea necesario existen métodos de obtener y cambiar (getter y setter escritos por el equipo).

2. **Se eliminó la variable `String estado` de la clase Personaje y se utilizó una sola en la clase Juego.**

   A su vez se eliminaron dos variables adicionales que existían en la clase juego: `boolean juegoPerdido y boolean juegoGanado = false;`

   `Ahora se utiliza una sola variable de tipo String para representar tres estados:`

   `String estadoDelJuego;`

   Los tres posibles estados pueden ser: "jugando", "ganado" o "perdido"

Este cambio facilitó que la función `tick()` de la clase juego, fuera fácilmente dividida (mediante condicionales) entre los tres estados, y el código sea más ordenado y explícito.

Es decir que la clase principal maneja el estado del juego. En la versión anterior, dependía de la clase jugador, y luego se informaba a la clase principal, siendo esto más confuso.

3. **Se cambió la variable `char ultima` de la clase Personaje por una variable `boolean miraDerecha`.**

   Este cambio, adicionado con la modificación de la lógica principal del estado y los movimientos del Personaje a la clase Juego, facilitaron que la animación del personaje mire hacia izquierda o derecha con menor código fuente.

   También facilitó la escritura de código, eliminando recurrentes condicionales que preguntaban por izquierda o derecha según el char. Ahora no se consulta con condicionales si mira a derecha o izquierda, la lógica del juego aplica el cambio directamente y se utiliza el valor guardado en la variable.

   Por otro lado, una nueva función `cambiarImagen(String s)` se encarga de cambiar la imagen y aplicar si debe mirar a izquierda o derecha con la ayuda de esta nueva variable.

4. **El Constructor de la clase Personaje no recibe un parámetro de tipo Viga.**

   Ahora el personaje se crea en coordenadas fijas.

```
 // Posición por defecto de spawn
this.posx = 50;
this.posy = 530;
```

5. **Se modificó la lógica principal del programa y ahora se ubica en la función `tick()` de la clase Juego.**

   Esto permitió descargar lógica de tres funciones distribuidas de Personaje que se encargaban de dibujar al mismo según diferentes condiciones. Si bien las funciones no se superponían, ahora el código es más legible y ordenado.

   Las funciones de personaje son para brindar cálculos auxiliares, para obtener datos de las variables de instancia o para setear datos nuevos, y una sola para realizar el dibujo en pantalla.

   Las posibles situaciones que puede realizar un personaje (desplazarse, saltar, caer, moverse por escalera) ahora son analizadas dentro de una sóla función y ya no distribuidas entre tres. Esa sola función es tick() de la clase Juego.

La función dibujar (de Personaje) se ejecuta una sola vez por cada tick y utiliza los parámetros guardados en sus variables de instancias. Las cuales fueron modificadas por los algoritmos nuevos implementados en tick().

6. **El personaje no puede ser controlado por el usuario una vez ganado el juego.**

   Se implementó con la función tick(), en los condicionales que analizan el estado del juego. Cuando el estado es "ganado" ya no se ejecutan los algoritmos que calculan los movimientos del personaje.

   Por otro lado, se mantuvo que el personaje cayera por completo cuando el estado es "perdido".

7. **Todos los randoms que utilizan las clases para tomar decisiones pseudoaleatorias, son variables locales de sus respectivos métodos.**

   La clase Donkey, utiliza randoms para decidir cuando va a lanzar su próximo barril. También se utiliza un random para decidir si el barril seguirá por la viga o caerá por la escalera al encontrarse alguna.

   La clase Escalera, utiliza randoms para decidir en qué posición con respecto al eje "x" va a construir la escalera en la ejecución actual. También para decidir si la escalera adicional que construye va a estar completa o no.

8. **Se eliminó un parámetro no utilizado en el método de instancia `estoyCercaDeEscalera(Escaleras[] escaleras, Viga[] suelos)` y en `pisando(Viga[] suelos)`**

   Se detectó que el método `pisando(Viga[] suelos)` requería un parámetro de tipo Entorno pero el mismo no era utilizado. A su vez el método `estoyCercaDeEscalera(Escaleras[] escaleras, Viga[] suelos)` utilizaba el primero, y por ende requería, el Entorno.

   Se solucionó este error.

9. **Explicación de porqué los objetos de la clase Barril tiene una variable de instancia llamada `saltado` de tipo `boolean`.**

   Se agregó dicha información a la documentación de la clase. También se explica en los comentarios del código fuente.

10. **Eliminación de la clase Mensaje e implementar lo que esta hacía en la clase Juego.**

    Las líneas de código fueron trasladadas a los apartados condicionales que ejecutan

acciones según el estado del juego.

IMPLEMENTACIÓN DE EXTRAS:

Se implementaron las siguientes funcionalidades que el equipo aspira a que sean considerados extras por los docentes:

1. **Sistema de puntuación.**
   Aplica 15 puntos cuando se detecta un correcto salto sobre un barril, pero no permite que un barril sea saltado varias veces. Aplica 100 puntos por ganar la simulación.

2. **Sonidos.**
   Existe un loop de audio en todo el juego. Tres variaciones de audio para los pasos del personaje. Sonido para el salto del personaje. Sonido para el correcto salto del personaje sobre un barril.

3. **Escaleras adicionales.**
   Por cada nivel de viga-suelo con su viga-techo existen 2 escaleras .

   La escalera obligatoria al final de la viga-suelo (las escaleras varían levemente su posición “x” en cada ejecución).

   Y una escalera adicional que también varía (poco más que levemente su posición “x”) y que además según un random puede estar completa o incompleta.

   Es decir que las escaleras varían sus posiciones y además se pueden obtener niveles del juego con escaleras todas completas (poco probable), todas incompletas (relativamente probable) o una combinación de ellas.

4. **Barriles pueden caer por las escaleras**
   Los barriles, cada vez que detectan que pasan por una escalera completa, deciden si continúan su rodamiento por la viga o bien pueden caer por la escalera. Se eligió un nivel relativamente bajo de probabilidades de que esto ocurra para que no sea imposible ascender por las escaleras.

5. **Combinación de apartado gráfico con estrategias gráficas nativas de la herramientas provistas.** (A consideración de los docentes)
   Las vigas son una combinación de un rectángulo sólido con triángulos negros (dando la sensación de estructura metálica) construidos mediante un bucle. No se colocaron manualmente los triángulos, los mismos se construyen mediante iteraciones calculando cantidad y posición de triángulos según el rectángulo (o la el objeto Viga a construir).

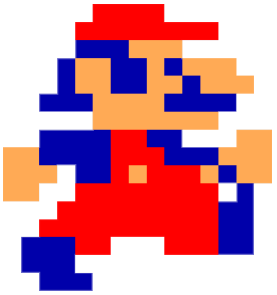
   El personaje está representado por animaciones gifs o imágenes estáticas png y la

lógica del programa es relativamente más compleja, no sólo cambia las posiciones “x” e “y”, sino que se encarga de mostrar la imagen adecuada (tipo y sentido) para que el gráfico sea representativo para el usuario.

Los barriles también son imágenes animadas para representar el rodamiento, y sus animaciones coinciden con el movimiento. La lógica implementada también genera cambios en la rotación, de la animación de la caída de un barril por una escalera para dar la sensación de que el mismo da tumbos en la caída.

Las escaleras fueron creadas emulando las vigas pero en vez de triángulos se utilizan rectángulos negros, y la construcción es vertical en vez de horizontal.
Al construir nuevas escaleras en esta segunda versión (las escaleras adicionales) no fue necesario cambiar la forma de la construcción, puesto que el bucle que aplica rectángulos negros funciona correctamente para las variables de instancia de dicha escalera (sea cuales sean, respetando el invariante de representación).

Clase: Personaje



Alto Original:	500px
Ancho Original:	500px
Escala utilizada:	0.090
Alto Utilizado:	40px
Ancho Utilizado:	40px
Distancia centro-suelo:	20px
Distancia centro-lateral:	15px

La clase Personaje permite generar un objeto que representará gráficamente al protagonista del juego.

VARIABLES

Las posiciones en coordenadas X e Y

```
private int posx;
private int posy;
```

Todas las imágenes que forman las animaciones del personaje

```
private Image mirandoIzquierda;
private Image mirandoDerecha;
private Image caminandoIzquierda;
private Image caminandoDerecha;
private Image saltandoIzquierda;
private Image saltandoDerecha;
private Image subiendo;
private Image subiendo_quieto;
```

Esta referencia realizará un aliasing (de las anteriores) a la imagen que debe ser mostrada.

```
private Image imagenMario;
```

Tick en el cual se ejecutó el último salto (o el salto actual)

```
private int tiempoSalto;
```

Indica si está saltando (ascendiendo) o no.

```
private boolean estaSaltando;
```

Indica si está cayendo (es decir que sus pies no están tocando viga alguna.

```
private boolean estaCayendo;
```

Indica si el personaje se encuentra lo suficientemente cerca de una escalera (para poder usarla)

```
private boolean estaCercaEscalera;
```

Indica si el personaje se encuentra dentro (usando) una escalera

```
private boolean estaEnEscalera;
```

Indica el índice que corresponde a la posición de la escalera que se está usando dentro del arreglo de escaleras

```
private int subidoAEscaleraNro;
```

Último archivo de sonido que se usó para caminar, hay 3 variantes.

```
private int sonando;
```

Tick en el cual se ejecutó el último sonido de caminar (ayuda a evitar que suenen sonidos en cada tick)

```
private int sonandoDesde;
```

Esta variable indica si el personaje está mirando a derecha o no (Vital para que se cargue la imagen correcta del personaje según los movimientos que indique el usuario)

```
private boolean miraDerecha;
```

MÉTODOS

- **Constructor** ()

  El constructor asigna:
  - La posición (50,530).
  - Las rutas URL donde se encuentran los archivos GIFs y PNGs
  - Mirando a derecha como verdadero
  - Y los estados “está cayendo”, “está saltando”, “está en una escalera”, y “está cerca de una escalera” como falsos.

- **tocando** (Debe recibir un barril como parámetro)

  Esta función se encarga de comparar las coordenadas de los extremos del personaje y los del barril recibido como parámetro. Luego informa si hay colisión o no.

- **hacerSonar** (Debe recibir el momento actual expresado como un número entero, donde cero es el inicio de ejecución)

  Esta función ejecuta el sonido de caminar pero evita que suene en cada tick donde se está caminando. Sino habría una bola de sonido indistinguible.

  La función decide sonar alguna de las 3 variantes de sonidos de pasos que hay. Y sólo hace sonar cuando la distancia entre el sonido anterior y el actual es de 40 ticks.

- **pisando**(Debe recibir como parámetro, el arreglo con las vigas utilizadas en la instancia Juego)

  Esta función devuelve el índice que ocupa la viga en el arreglo de suelos que el jugador está pisando[1]. Si no se encuentra pisando, entonces devuelve -1.

  Para saber si **no** está pisando la viga, el centro 'y' del personaje + 20 pixeles (para llegar al pie del personaje) **`obtenerPosPies()`** debe poseer un valor distinto para la coordenada 'y' donde comienza cada viga (la posy - 12px) **`(int)suelos[i].dondeEmpiezaElSuelo()`**.

  En el caso de que el personaje se encuentra pisando la viga. Queda por conocer si se encuentra dentro de todos los puntos 'x' que conforman el largo de la viga. Porque de lo contrario, no se encontraría pisándola.

  Por eso la función analiza que el extremo derecho de la viga, sea pisada por al menos el lateral izquierdo del personaje, y lo mismo de forma opuesta.

  Si no se cumple esta condición, el personaje está cayendo por estar fuera de la viga a pesar de estar a la altura de alguna de ellas.

- **estoyCercaDeEscalera** (Debe recibir como parámetro el arreglo con las vigas utilizadas en la instancia Juego; y el arreglo con las escaleras utilizadas en la instancia Juego)

  Esta función cambia el valor de **`estaCercaEscalera`** a true o false dependiendo si el personaje está cerca de una escalera como para poder subir o descender por ella.

  Sólo analiza la proximidad de una escalera, si la función pisando devuelve el índice de la viga pisada. No se analiza proximidad para valores -1 (en el aire) ni si se está cayendo.

  Las escaleras se analizan en dos casos separados. Las que comienzan en el piso actual del personaje y ascienden al superior, y las que terminan en el piso actual porque descienden al inferior. A su vez, dentro de cada caso, se analiza la existencia de una escalera obligatoria y completa, y la de una escalera adicional (sea completa o no).

  Para estar cerca de una escalera los puntos “x” extremos de la escalera deben contener al punto central “x” del personaje. Y la ubicación del punto “y” ocupada por los pies del personaje debe estar a una distancia cercana al extremo correspondiente de la escalera.

[1] Definamos pisando como ocupar el pixel igual o inmediatamente superior del último píxel superior ocupado por una viga. Dicho pìxel ocupado debe ser el primero inferior del personaje.

- **saltandoBarril** (Debe recibir una variable de tipo Barril)

Retorna verdadero cuando se realiza un salto exitoso sobre un barril. Se debe ejecutar en cada tick y se analiza cada barril.

El salto es exitoso cuando:

- La posición “x” del barril es igual a la posición “x” del personaje (con un ayuda de +/- 1 píxel a cada lado)
- Los pies del personaje están por encima de la parte superior del barril
- Pero no a tanta diferencia (tan alto no salta el personaje, sin esta condición los barriles en pisos inferiores serían considerados como saltados)
- Que el barril no haya sido previamente saltado.
- No estar dentro de una escalera.

- **dibujar** (Debe recibir el entorno como parámetro;una variable de tipo entera indicando la rotación de la figura del personaje)

  Dibuja al personaje. Se deben calcular las situaciones y cambiar las variables previamente con otros métodos.

  Utiliza siempre la image almacenada en

- **cambiarImagen** (Debe recibir una variable de tipo String)

  Dependiendo el String recibido, cambia la imagen a la que apunta `imagenMario`. También utiliza para decidir la imagen correcta el valor de `miraDerecha`.

- **obtenerPosPies**(No requiere parámetros adicionales)

  Devuelve la posición del extremo inferior del personaje.

- **obtenerPoscabeza** (No requiere parámetros adicionales)

  Devuelve la posición del extremo superior del personaje.

- **lateralDerecho** (No requiere parámetros adicionales)

  Devuelve la posición del extremo derecho del personaje.

- **lateralIzquierdo** (No requiere parámetros adicionales)

  Devuelve la posición del extremo izquierdo del personaje.

- **estaEnEscalera**(No requiere parámetros adicionales)

  Devuelve verdadero o falso según si está en escalera o no.

- **cambiarY** (Requiere un entero)

  Setea a la posicicón “Y” actual como la suma, al valor actual, del entero recibido.

- **cambiarX**(Requiere un entero)
  Setea a la posicicón “X” actual como la suma, al valor actual, del entero recibido.

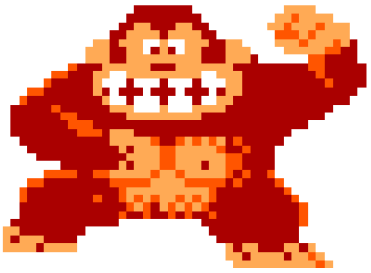
  El resto de los métodos son del tipo “**obtener<NombreVariable>**” (get) y “**cambiar<NombreVariable>**” (set).

```
/*
 * Getters
 */

public boolean obtenerEstaEnEscalera() {
	return this.estaEnEscalera;
}

public int obtenerMomentoDeSalto() {
	return this.tiempoSalto;
}

public boolean obtenerEstaCayendo() {
	return this.estaCayendo;
}

public boolean obtenerEstaCercaEscalera() {
	return this.estaCercaEscalera;
}
```

```java
public int obtenerSubidoAEscaleraNro() {
	return this.subidoAEscaleraNro;
}

public boolean obtenerMiraDerecha() {
	return this.miraDerecha;
}

public boolean obtenerEstaSaltando() {
	return this.estaSaltando;
}
```

```java
/*
 * Setters
 */

public void cambiarMomentoDeSalto(int i) {
	this.tiempoSalto = i;

}

public void cambiarEstaEnEscalera(boolean escalera) {
	this.estaEnEscalera = escalera;
}

public void cambiarMiraDerecha(boolean mira) {
	this.miraDerecha = mira;
}

public void cambiarEstaSaltando(boolean salta) {
	this.estaSaltando = salta;
}
```

Clase: Donkey



Alto Original:	350px
Ancho Original:	500px
Escala utilizada:	0.19
Alto Utilizado:	66px
Ancho Utilizado:	95px
Distancia centro-suelo:	33px

La clase Donkey permite generar un objeto que se encargará, de arrojar barriles según una elección aleatoria del tiempo. También, desde el apartado gráfico, genera una animación del antagonista del juego, que varía entre estar enojado de forma constante, y la de simular que arroja barriles en la creación de uno de estos.

VARIABLES

- **`int ultimoLanzamiento;`**
  Indica el momento (número de tick desde que comenzó la ejecución) en que se realizó el último lanzamiento de un barril.

- **`int lanzarRandom;`**
  Indica el momento (número de tick desde que comenzó la ejecución) en que se realizará el próximo lanzamiento de un barril.

- **`String violencia;`**
  Sirve para indicar el estado de donkey con respecto a si debe arrojar barriles o no.

MÉTODOS *

- **Constructor** (No requiere parámetros adicionales)

  El constructor solamente asigna como último lanzamiento el momento cero. Y el estado de violencia como “violento”.

- **Gorilear** (Debe recibir el entorno como parámetro y el momento actual expresado como un número entero, donde cero es el inicio de ejecución)

  Este método genera una constante animación. Es solamente un decorado.

  Si el último lanzamiento se realizó hace menos de 30 ticks debe mostrar la animación “tirar”. De lo contrario debe mostrar la simple animación llamada “gorilear”.

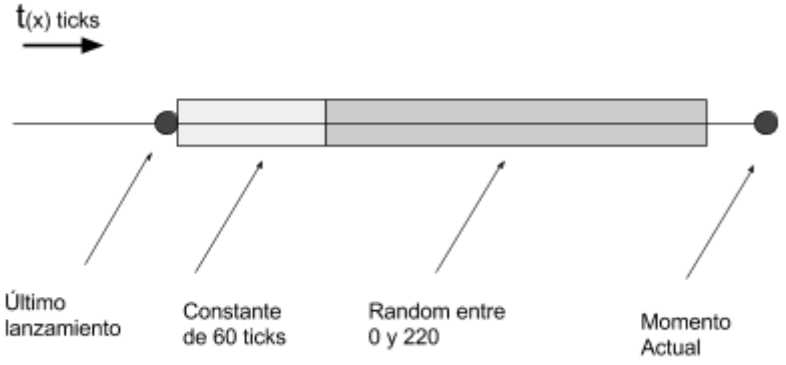
- **decidir**(Debe recibir el momento actual expresado como un número entero, donde cero es el inicio de ejecución)

  Crea y utiliza una variable local: `Random rnd = new Random();`

  Es el método que se encarga de ejecutar un algoritmo que decide de forma aleatoria en cual tick del futuro (momento del juego) se lanzará el siguiente barril. También es el método que indica que debe ser lanzado el barril en ese preciso instante, si el momento actual es igual al tick que fue planeado su lanzamiento. Lo anterior ocurre si la **`violencia`** está seteada en "violento".

  Retorna **`true`** si el momento actual es igual a **`lanzarRandom`**. Retorna **`false`** para los demás casos.
  Una vez lanzado el barril, **`lanzarRandom`** pasa a tener valor 0 (no hay lanzamiento a posterior planificado).

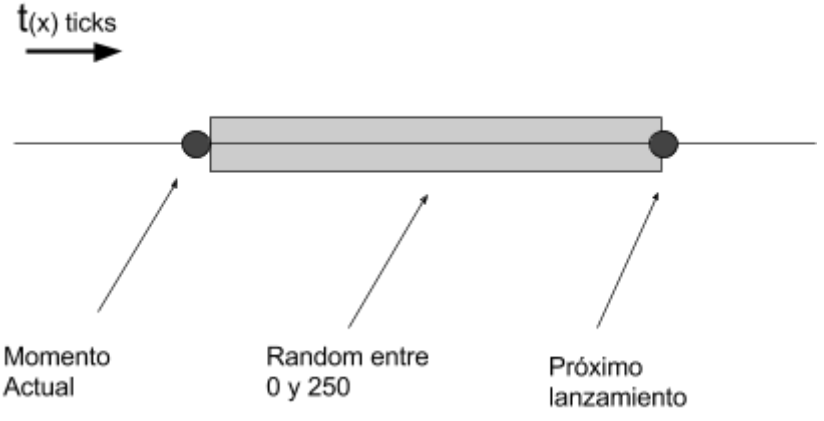
  Existe un algoritmo basado en dos randoms diferentes para decidir el próximo lanzamiento.



  Se decidirá una planificación de un lanzamiento futuro, si ocurre primero que el momento actual es mayor o igual al último lanzamiento más 60 ticks y un random de entre 0 y 220 ticks.

  Esto garantiza que no se dé un paso inicial hacia lanzar barriles cada menos de 60 ticks (aprox 2 segundos o menos).

  Cuando se cumple lo anterior. Se establece un random entre 0 y 250 hacia el futuro para planificar el próximo lanzamiento.



- **noMasViolencia**(No requiere parámetros adicionales)

  Este método cambia la variable **`violencia`** a “noviolento”. Lo que impide al método “decidir“ de arrojar barriles o planificar futuros lanzamientos. Esta función es llamada desde la clase principal cuando el juego termina, ya sea al perder o al ganar.

- **arribaOabajo**(No requiere parámetros adicionales)

  Crea y utiliza una variable local: Random rnd = new Random();

  Este método genera un random entre 0 y 60. Si el número elegido aleatoriamente es múltiplo de 3, entonces se decide arrojar el barril directamente hacia la viga inferior, de lo contrario se arroja por la misma viga donde está donkey.

  Este método se diseñó para agregarle dificultad al juego.

  Se utiliza un random y que el resultado sea múltiplo de 3 para que en general se respete que haya un aproximado de 33% de posibilidades de arrojar el barril por debajo y un 66% por la viga normal.

  Se retorna -3 o -1 con la intención de que el resultado de esta función le reste dichas unidades al valor length del arreglo de Vigas[].

ANOTACIONES

Para la correcta aparición gráfica de Donkey se recomienda colocar ciertos valores de **entorno.dibujarImagen**

```
viga = Viga (pos = 6)
y = distancia ( viga.y - viga.alto / 2)
```

La distancia entre el "y" de la viga (con pos = 6) menos la mitad de su alto

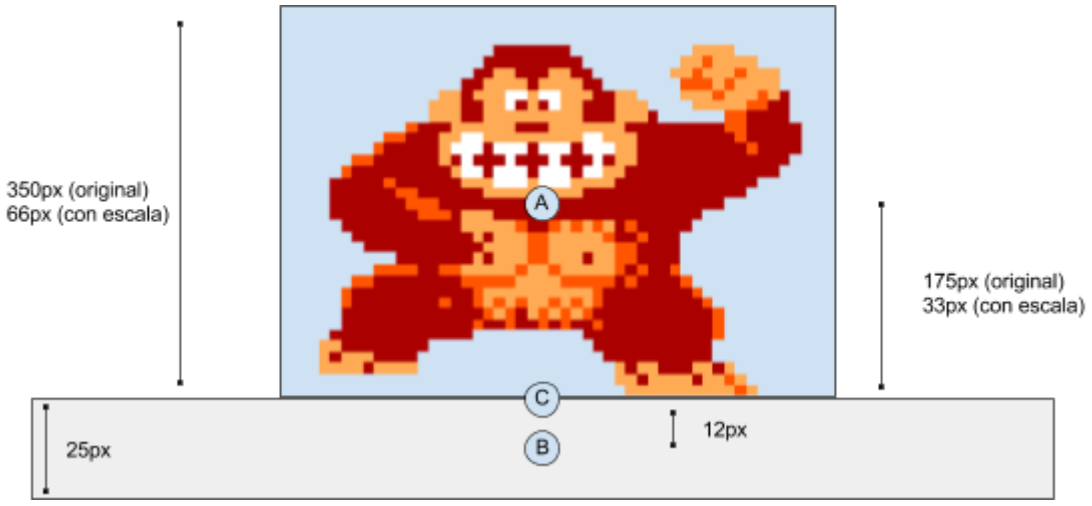
Para una viga en posición 6 con los siguientes valores:

```
x = 325;
y = 75;
largo = 700;
alto = 25;
```

Se recomienda ubicar a Donkey en:

```
x = 50
y = 30
escala = 0.19
```

Esto produce que el último píxel inferior del Donkey pise (o límite inmediata y superiormente con) el último píxel superior de la viga).



A = Punto centro de la imagen (x,y) Donkey.
B = Punto centro del rectángulo que dibuja la Viga.
C = Límite entre la viga y la imagen.

Clase: Viga

La clase Viga permite generar un objeto que se encargará de dibujar piso y techo del nivel del juego. Las vigas en este software son implementadas como rectángulos sólidos que tienen dentro de ellos parejas triángulos (uno normal y otro invertido) con el mismo color de fondo para asemejar una estructura metálica.

VARIABLES

```
private int pos;
```

Un indicador propio de posición (no el índice en el arreglo de Vigas). La posición 1 es la planta baja, la 2 el primer piso y así sucesivamente.

```
private double x;
private double y;
private double largo;
private double alto;
```

MÉTODOS

- **Constructor** (Requiere que se le pasé un parámetro de tipo numérico de posición)

  El constructor asigna valores a “x”, “y” , “largo” y “alto” según la posición solicitada para construir gráficamente los pisos del nivel.

  Este equipo de trabajo decidió que las vigas tuvieran un grosor (o alto) de 25px, un largo de 700px (excepto para la planta baja que ocupa todo el ancho).

- **dibujar** (Debe recibir el entorno como parámetro)

  Esta función debe ser llamada en cada tick por cada viga que exista. Por cada tick dibuja la viga.

  La construcción gráfica de la viga tiene un condimento especial. Para dar la sensación de que es una estructura metálica, se dibuja un rectángulo de color rojo de fondo, y sobre él de forma

estratégica, triángulos del mismo color que el fondo, en juegos de a dos. Cada uno invertido 90º con respecto al anterior.

- **getPosx** (No requiere parámetros adicionales)

  Devuelve la posición del centro x de la viga.

- **getPosy** (No requiere parámetros adicionales)

  Devuelve la posición del centro y de la viga.

- **getAncho** (No requiere parámetros adicionales)

  Devuelve el largo o ancho de la viga. Longitud en el eje “x”.

- **getPos** (No requiere parámetros adicionales)

  Devuelve la pos que se indicó al momento de su creación “x”.

- **dondeEmpiezaElSuelo** (No requiere parámetros adicionales)

  Devuelve la posición “y” del extremo inferior de la viga.

- **dondeTerminaElTecho** (No requiere parámetros adicionales)

  Devuelve la posición “y” del extremo superior de la viga.

- **extremoIzquierdo**(No requiere parámetros adicionales)

  Devuelve la posición “x” del extremo izquierdo de la viga.

- **extremoDerecho**(No requiere parámetros adicionales)

  Devuelve la posición “x” del extremo derecho de la viga.

Clase: Escalera

La clase Escalera permite generar un objeto que en el apartado gráfico simula una escalera y al existir una cercanía adecuada con el personaje, le permitirá a este último desplazarse por el eje “y” del juego.

VARIABLES

```
private double x;
private double y;
private double largo;
private double alto;
```

MÉTODOS *

- **Constructor** (Requiere que se le pasé un parámetro de tipo numérico de posición y el arreglo con las vigas utilizadas en la instancia Juego)

  La posición número no es el índice en el arreglo de Escaleras, sino la ubicación entre que vigas se encontrará la escalera y que tipo de escalera es (obligatoria o adicional).

  La posición 1 es la escalera obligatoria de planta baja hacia primer piso, la 2 del primer piso con el segundo**.** Pero la posición 5 representa la escalera adicional de planta baja hacia primer piso, la 6 del primer piso con el segundo.

  Para las escaleras obligatorias:

  El constructor asigna valores a “x”, “y” , “largo” y “alto” según la posición solicitada para construir gráficamente las escaleras que conectar un piso con otro.

  Se determinado un ancho fijo de 30px para cada escalera. Un alto lo suficiente para que cubra la distancia entre el suelo inferior y el suelo de la viga superior.

  El punto “y” se sitúa en el punto medio de dicha distancia.

El punto “x” se sitúa en un random que varía 50px en el extremo de la viga superior y con un margen de seguridad de otros 30 píxeles. Esto proporciona una mínima diferencia en la posición horizontal de las escaleras con respecto a cada ejecución del juego.

Para las escaleras adicionales:

El constructor asigna valores a “x”, “largo” según la posición solicitada para construir gráficamente las escaleras que conectar un piso con otro. Pero luego de realizar un random para decidir si la escalera es completa o está incompleta (no llega hasta el piso superior) se decide el “y” y el “alto” correcto.

- **dibujar** (Debe recibir el entorno como parámetro)

  Este método dibuja la escalera. Debe ser llamado en cada tick de la ejecución del juego.

  Dibuja el rectángulo con los atributos que posee la instancia creada, y a su vez realiza un cálculo para dibujar pequeños rectángulos inscritos dentro del principal, del mismo color que el fondo de la ventana, con la finalidad de simular una escalera.

- **extremoInferior**(No requiere parámetros adicionales)

  Devuelve la posición “y” del extremo inferior de la escalera.

- **extremoSuperior**(No requiere parámetros adicionales)

  Devuelve la posición “y” del extremo superior de la escalera.

- **lateralIzquierdo**(No requiere parámetros adicionales)

  Devuelve la posición “x” del extremo izquierdo de la escalera.

- **lateralDerecho**(No requiere parámetros adicionales)

  Devuelve la posición “x” del extremo derecho de la escalera.

Clase: Barril

Alto Original:	108px
Ancho Original:	108px
Escala utilizada:	0.157
Alto Utilizado:	17px
Ancho Utilizado:	17px
Distancia centro-suelo:	10px
Distancia centro-lateral:	10px

La clase Barril permite generar un objeto que se encargará de dibujar barriles que se desplazarán en el sentido correcto por las vigas del nivel, caerán cuando no haya suelo o lo decida hacer por una escalera. Su función es la de ser un obstáculo móvil en el desarrollo del juego que el personaje no debe tocar.

VARIABLES

```
private double posx;
private double posy;
private int diametro;
private double escala;


// Contiene  las animaciones de rotar hacia derecha o izquierda para dar sensacion de movimiento.
private Image spin_izquierda;
private Image spin_derecha;

// Indica hacia donde se movía por última vez el barril "izquierda" o "derecha"
private String ultima;

/*
 * Es necesaria una variable que indique que el barril fue saltado para que no
 * sea contado doble en las siguientes situaciones: En el proceso de salto, la
 * función que detecta el correcto salto puede dar varios positivos durante una
 * cantidad de ticks cercanos entre sí. Atrapando el primer tick donde se
 * detecta que el barril fue saltado, se evita que en los ticks siguientes donde
 * también es positivo, se cuenten los puntos innecesariamente. Evitar que el
 * jugador salte el barril en una viga y luego intente saltarlo en una viga
 * inferior para contador doble puntaje.
 */
private boolean saltado;
```

```
// Conocer si está cayendo por escalera permite detener el movimiento hacia
// izquierda o derecha. Ayuda a diferenciar una caída desde una viga con
// respecto a la de escalera.
private boolean cayendoPorEscalera;

// Como un barril tarda varios ticks en atravesar el ancho de una escalera.
// Necesitamos indicar en qué tick se tomó la decisión de caer o no por la
// misma. Para que en el tick siguiente no sobreescriba la decisión. Toma una decisión
// por escalera,y bloquea decidir de nuevo por un cierto tiempo.
private int ultimaEleccion;

// Para animar la caída por escalera se utiliza esta variable para intercambiar
// entre 10º y -10º.
private int anguloRotacion;

// Asiste en la elección del anguloRotacion
private boolean sentidoRotacionDerecha;
```

MÉTODOS

- **Constructor** (Requiere que se le pasé una variable de tipo Viga)

  Asigna el diámetro fijado y calcula la escala. Iguales para todos los barriles

  Calcula las posiciones “x” e “y” según en qué viga es lanzado el barril.

  Se asignan las rutas URL de los gifs que proveen la animación de rodamiento.

- **deboDestruirme** (Debe recibir el entorno como parámetro; y el arreglo con las vigas utilizadas en la instancia Juego)

  Esta función debe ser llamada en cada tick por cada barril que exista.

  Analiza si la posición del barril es en la planta baja y en el extremo izquierdo. Entonces retorna verdadero para indicar que este barril debe ser destruido para dejar paso a que donkey pueda crear otro.

- **dibujar** (Debe recibir el entorno como parámetro; el momento actual expresado como un número entero, donde cero es el inicio de ejecución; el arreglo con las vigas utilizadas en la instancia Juego; y el arreglo con las escaleras utilizadas en la instancia Juego)

  Esta función debe ser llamada en cada tick por cada barril que exista. Por cada tick se dibuja el barril.

  Esta función se encarga de mostrar el apartado gráfico del barril y calcular su trayectoria y movimiento.

  Calcula su desplazamiento en el eje “x” y analiza si dicho desplazamiento debe cambiar de sentido.

  También utiliza la función `caerPorEscalera` para saber si se encuentra sobre una escalera y si debe dejarse caer por ella.

  También se encarga de las caídas en el eje “y” cuando no existe suelo sobre el cual rodar o porque está cayendo por una escalera. Llama a las funciones `pisando` para determinar

  Una caída por escaleras sólo desplaza al barril en el eje “y” utilizando una imagen. La caída por final de viga desplaza al barril en el eje “y” y en el eje “x” utilizando otra imagen.

- **pisando** (Debe recibir como parámetro el arreglo con las vigas utilizadas en la instancia Juego)

  Esta función devuelve el índice que ocupa la viga en el arreglo de suelos que el barril está pisando[1]. Si no se encuentra pisando, entonces devuelve -1.

- **caerPorEscalera** (Debe recibir como parámetro el momento actual expresado como un número entero, donde cero es el inicio de ejecución; el arreglo con las vigas utilizadas en la instancia Juego; y el arreglo con las escaleras utilizadas en la instancia Juego)

  Esta función toma la decisión de decidir si el barril caerá por la siguiente escalera. Se encarga también de tomar una sola vez la decisión, puesto que se decide al estar cerca de una escalera.

  Como la cercanía hacia una escalera es verdadera en diferentes ticks para que la decisión se tome una sola vez, esta función se apoya en la veriable `ultimaEleccion`.

  Retorna true para que en ese preciso momento el barril caiga por la escalera en donde se encuentra. Retorna false si no hay escalera o sino se ha decido no caer.

[1] Definamos pisando como ocupar el pixel igual o inmediatamente superior del último píxel superior ocupado por una viga. Dicho pìxel ocupado debe ser el primero inferior del personaje.

- **pies**(No requiere parámetros adicionales)

  Devuelve la posición del extremo inferior del barril.

- **superior**(No requiere parámetros adicionales)

  Devuelve la posición del extremo superior del barril.

- **lateralDerecho** (No requiere parámetros adicionales)

  Devuelve la posición del extremo derecho del personaje.

- **lateralIzquierdo** (No requiere parámetros adicionales)

  Devuelve la posición del extremo izquierdo del barril.

- **centroX**(No requiere parámetros adicionales)

  Devuelve la posición del centro “x” del barril.

- **saltado**(No requiere parámetros adicionales)

  Cambia el valor de **`saltado`** a verdadero.

- **fueSaltado**(No requiere parámetros adicionales)

  Devuelve el valor de verdad sobre el atributo **`saltado`** que indica si fue saltado el barril.

Clase: Puntaje

La clase Puntaje permite generar un objeto que contará puntos por cada barril saltado y al llegar a cumplir el objetivo. También se encarga de mostrar dicha información en pantalla.

VARIABLES

```
private int puntos;
```

MÉTODOS *

- **Constructor** (No requiere parámetros adicionales)

  Asigna el puntaje en cero.

- **saltarbarril** (No requiere parámetros adicionales)

  Suma quince puntos al puntaje actual

- **ganar** (No requiere parámetros adicionales)

  Suma cien puntos al puntaje actual

- **dibujar** (Debe recibir el entorno como parámetro)

  Esta función muestra en pantalla la cantidad de puntos en el extremo superior derecho de la pantalla.

```java
package juego;

import java.awt.Color;

import entorno.Entorno;
import entorno.Herramientas;
import entorno.InterfaceJuego;

public class Juego extends InterfaceJuego {

    // El objeto Entorno que controla el tiempo y otros
    private Entorno entorno;

    // Puede ser "juganddo", "ganado" o "perdido"
    private String estadoDelJuego = "jugando";

        // Creación del arreglo de vigas
    static Viga suelos[] = new Viga[] {

            new Viga(1), new Viga(2), new Viga(3), new Viga(4), new Viga(5),
            new Viga(6)

    };

    // Creación del arreglo de escaleras
    static Escaleras escaleras[] = new Escaleras[] {
            new Escaleras(0, suelos), new Escaleras(1, suelos),
            new Escaleras(2, suelos), new Escaleras(3, suelos),
            new Escaleras(4, suelos), new Escaleras(5, suelos),
            new Escaleras(6, suelos), new Escaleras(7, suelos),
            new Escaleras(8, suelos), new Escaleras(9, suelos) };

    // Antagonista y Personaje principal
    private Donkey donkeyKong = new Donkey();
    private Personaje jugador = new Personaje();

    // Puntuador
    private Puntaje puntuador = new Puntaje();

    // El reloj medido en ticks
    int contador = 0;

    // Creación del arreglo de barriles
    private Barril barriles[] = new Barril[]{
            new Barril(suelos[suelos.length - 3]),
            null, null, null, null, null,
            null, null, null, null, null,
            null, null, null, null, null,
            null, null, null, null

    };

    // ...

    Juego() {
        // Inicializa el objeto entorno
        this.entorno = new Entorno(this, "Donkey - Grupo Pereira - Sanchez -
        Tula - V2", 800, 600);

        // Inicializar lo que haga falta para el juego
        // ...

        Herramientas.loop("rsc/sonidos/musica.wav");

        // Inicia el juego!
        this.entorno.iniciar();

```

```
66      }
67
68      /**
69       * Durante el juego, el método tick() será ejecutado en cada instante y
         por lo
70       * tanto es el método más importante de esta clase. Aquí se debe
         actualizar el
71       * estado interno del juego para simular el paso del tiempo (ver el
         enunciado
72       * del TP para mayor detalle).
73       */
74      public void tick() {
75
76
77          // Al inicio de cada ciclo aumentar una unidad el reloj
78          contador++;
79
80
81
82          // Ejecuta la función dibujar por cada miembro del arreglo de vigas.
83          for (int i = 0; i < suelos.length; i++) {
84              suelos[i].dibujar(entorno);
85          }
86
87          // Ejecuta la función dibujar por cada miembro del arreglo de
            escaleras.
88          for (int i = 0; i < escaleras.length; i++) {
89              escaleras[i].dibujar(entorno);
90          }
91
92          // Ejecuta la función dibujar para donkey
93          donkeyKong.gorilear(entorno, contador);
94
95          // Ejecuta la función dibujar para el contador
96          puntuador.dibujar(entorno);
97
98
99          // Ejecuta la función dibujar por cada elemento no NULL del arreglo
            de barriles
100         // , también analiza si un barril debe destruirse.
101         for (int i = 0; i < barriles.length; i++) {
102             if (barriles[i] != null) {
103
104                 barriles[i].dibujar(entorno, contador, suelos, escaleras);
105                 if (barriles[i].deboDestruirme(entorno, suelos)) {
106                     barriles[i] = null;
107                 }
108
109             }
110         }
111
112
113
114         /*
115          * Analisis que ocurren mientras el juego se desarrolla
116          * Es decir que el jugador no ganó ni perdió aún.
117          */
118         if (this.estadoDelJuego.equals("jugando")) {
119
120
121             // Analizar si el personaje se encuentra cerca de escalera
122             jugador.estoyCercaDeEscalera(escaleras, suelos);
123
124
125
126             /*
127              * CAER
```

```
128                 * NO escalera, NO saltando, NO pisando
129                 */
130                if (!jugador.obtenerEstaEnEscalera() && !jugador.
                   obtenerEstaSaltando() && jugador.pisando(suelos) == -1) {
131
132
133                    jugador.cambiarImagen("saltando");
134                    jugador.cambiarY(1);
135
136                }
137
138
139                /*
140                 * SALTAR (Parte del proceso de saltar de una única ejecución)
141                 * Si presionada tecla espacio, Salto anterior dista más de 60
                    ticks, No pisando, No está en Escalera
142                 */
143                if (entorno.sePresiono(entorno.TECLA_ESPACIO) && jugador.
                   obtenerMomentoDeSalto() + 60 < contador
144                        && jugador.pisando(suelos) != -1 && !jugador.
                           obtenerEstaEnEscalera()) {
145
146                    /*
147                     * Esta es la parte de un salto que se ejecuta una sola vez.
                        Es decir que no se
148                     * encarga de la animación de subida o caida a lo largo de
                        los ticks de un salto
149                     * normal.
150                     *
151                     * Cambia el estado de estaSaltando a verdadero. Ejecuta el
                        sonido
152                     * del salto. Indica el tick en el cual se realizó el salto,
                        guardando el valor
153                     * en tiempoSalto.
154                     *
155                     */
156                    jugador.cambiarImagen("saltando");
157                    jugador.cambiarMomentoDeSalto(contador);
158                    jugador.cambiarEstaSaltando(true);
159                    Herramientas.play("rsc/sonidos/jump.wav");
160
161                }
162
163
164                /*
165                 * INGRESAR A ESCALERA
166                 * La única forma de pasar a estar dentro de una escalera
                    (estando cerca de una
167                 * escalera pero no dentro de una)
168                 */
169                if (jugador.obtenerEstaCercaEscalera() && !jugador.
                   obtenerEstaEnEscalera()) {
170
171                    // Entrar subiendo la escalera
172                    if (entorno.sePresiono(entorno.TECLA_ARRIBA) && jugador
173                            .obtenerPosPies() > escaleras[jugador.
                               obtenerSubidoAEscaleraNro()].extremoSuperior()) {
174
175                        jugador.cambiarEstaEnEscalera(true);
176                        jugador.cambiarY(-2);
177                        jugador.cambiarImagen("subiendo");
178
179
180
181                    }
182
183                    // Entrar bajando la escalera
```

```
184                 else if (entorno.sePresiono(entorno.TECLA_ABAJO) && jugador
185                         .obtenerPosPies() < escaleras[jugador.
                            obtenerSubidoAEscaleraNro()].extremoInferior()) {
186
187                     jugador.cambiarEstaEnEscalera(true);
188                     jugador.cambiarY(2);
189                     jugador.cambiarImagen("subiendo");
190
191
192
193                 }
194             }
195
196
197             /*
198              * Moverse dentro de una escalera. Estando dentro de una.
199              */
200             if (jugador.obtenerEstaEnEscalera()) {
201
202
203                 /*
204                  * SUBIR ESCALERA
205                  * Esta función ejecuta las animaciones correspondiente a
                    subir escalera y se
206                  * encarga de informar si ya terminó de subirla. Es decir que
                    sale de la
207                  * escalera y se encuentra en el piso superior.
208                  *
209                  */
210                 if (entorno.estaPresionada(entorno.TECLA_ARRIBA)) {
211
212                     // Subió tanto la esclaera que salió al piso superior
213                     if (jugador.obtenerPosPies() < escaleras[jugador.
                        obtenerSubidoAEscaleraNro()].extremoSuperior()) {
214
215                         // Ya no está en escalera
216                         jugador.cambiarEstaEnEscalera(false);
217
218                         // Para las vigas con indice par, el personaje debe
                            ir hacia izquierda. Para las impares, hacia la derecha.
219                         if (jugador.obtenerSubidoAEscaleraNro() % 2 == 0) {
220                             jugador.cambiarMiraDerecha(false);
221                         } else {
222                             jugador.cambiarMiraDerecha(true);
223                         }
224
225                     }
226
227                     // Si aún no salió de escalera, solamente se desplaza
                        hacia arriba.
228                     else {
229
230                         jugador.cambiarY(-2);
231                         jugador.cambiarEstaEnEscalera(true);
232
233                     }
234
235                 }
236
237                 /*
238                  * BAJAR ESCALERA
239                  * Esta función ejecuta las animaciones correspondiente a
                    subir escalera y se
240                  * encarga de informar si ya terminó de subirla. Es decir que
                    sale de la
241                  * escalera y se encuentra en el piso superior.
242                  *
```

```
243                  */
244                 else if (entorno.estaPresionada(entorno.TECLA_ABAJO)) {
245 
246                     // Bajó tanto la esclaera que salió al piso inferior
247                     if (jugador.obtenerPosPies() >= escaleras[jugador.
                    obtenerSubidoAEscaleraNro()].extremoInferior()
248                             - 5) {
249 
250                         // Ya no está en escalera
251                         jugador.cambiarEstaEnEscalera(false);
252 
253                         // Para las vigas con indice par, el personaje debe
                        ir hacia izquierda. Para las impares, hacia la derecha.
254                         if (jugador.obtenerSubidoAEscaleraNro() % 2 == 0) {
255                             jugador.cambiarMiraDerecha(false);
256                         } else {
257                             jugador.cambiarMiraDerecha(true);
258                         }
259 
260                         // Si aún no salió de escalera, solamente se desplaza
                        hacia abajo.
261                     } else {
262 
263                         jugador.cambiarY(2);
264                         jugador.cambiarEstaEnEscalera(true);
265 
266                     }
267 
268 
269 
270                     /*
271                      * Si está en escalera pero el usuario no presionó ni la
                     tecla Arriba ni la tecla abajo, se queda inmóvil.
272                      */
273                 } else {
274 
275                     jugador.cambiarImagen("quieto");
276 
277                 }
278 
279             }
280 
281 
282 
283             /*
284              * SALTO
285              * Resto de la animación del salto.
286              *
287              * Si no está en escalera,
288              * Si se está saltando y el momento actual dista a menos de 30
                ticks del inicio del salto:
289              * Se está en la parte ascendente del salto.
290              *
291              * Sino, ya no se está saltando (Se informa que ya no se está
                saltando).
292              * La caida se produce por el primer llamado CAER
293              *
294              */
295             if (!jugador.obtenerEstaEnEscalera()) {
296 
297                 if (jugador.obtenerEstaSaltando() && contador - jugador.
                obtenerMomentoDeSalto() < 30) {
298 
299                     jugador.cambiarImagen("saltando");
300                     jugador.cambiarY(-1);
301 
302                 } else {
```

```
303                     jugador.cambiarEstaSaltando(false);
304
305                 }
306             }
307
308
309             /*
310              * DESPLAZARSE
311              * Solo cuando no se cae, no se está saltando y no está en escalera
312              */
313             if (!jugador.obtenerEstaCayendo() && !jugador.obtenerEstaSaltando
                () && !jugador.obtenerEstaEnEscalera()) {
314
315                 // Moverse a Derecha
316                 if (entorno.estaPresionada(entorno.TECLA_DERECHA) && jugador.
                    lateralDerecho() <= 800) {
317                     jugador.cambiarX(2);
318                     jugador.cambiarImagen("caminando");
319                     jugador.hacerSonar(contador);
320                     jugador.cambiarMiraDerecha(true);
321
322                 }
323
324                 // Moverse a izquierda
325                 else if (entorno.estaPresionada(entorno.TECLA_IZQUIERDA) &&
                    jugador.lateralIzquierdo() >= 0) {
326
327                     jugador.cambiarX(-2);
328                     jugador.cambiarImagen("caminando");
329                     jugador.hacerSonar(contador);
330                     jugador.cambiarMiraDerecha(false);
331
332                 }
333
334                 // No moverse (en el suelo)
335                 else {
336
337                     jugador.cambiarImagen("mirando");
338
339                 }
340
341             }
342
343             /*
344              * Luego de analizar todas las posibles situaciones cambiantes
                por cada tick en el personaje
345              * Se ejecuta la función dibujar, que toma el estado de ciertas
                variables para producir
346              * la imagen correcta del personaje.
347              */
348             jugador.dibujar(entorno, 0);
349
350
351             /*
352              * GANAR
353              * Acercarse a determinada posición del juego (sin haber perdido)
354              * Genera la victoria automática.
355              */
356             if (jugador.pisando(suelos) == suelos.length - 1 && jugador.
                lateralIzquierdo() <= 150) {
357                 this.estadoDelJuego = "ganado";
358                 puntuador.ganar();
359             }
360
361
362             /*
363              * PERDER
```

```
364                 * Si la función que reporta si el personaje tocó algún barril da
                    verdadero
365                 * Se genera la derrota automática.
366                 */
367                if (jugador.tocando(barriles)) {
368                    this.estadoDelJuego = "perdido";
369                }
370
371                /*
372                 * PUNTUACIÓN
373                 * Por cada barril se analiza si el jugador lo saltó.
374                 * Para la puntuación.
375                 */
376                for (int i = 0; i < barriles.length; i++) {
377                    if (barriles[i] != null) {
378                        if (jugador.saltandoBarril(barriles[i])) {
379                            puntuador.saltarbarril();
380                        }
381
382                    }
383                }
384
385
386                /*
387                 * Donkey arrojando barriles
388                 *
389                 * Si donkey decide arrojar un barril en el tick actual, se crea
                    uno nuevo en la primera
390                 * posició no NULL del arreglo de barriles.
391                 */
392                if (donkeyKong.decidir(contador)) {
393                    int creados = 0;
394
395                    for (int i = 0; i < barriles.length && creados == 0; i++) {
396                        if (barriles[i] == null) {
397                            barriles[i] = new Barril(suelos[suelos.length +
                                donkeyKong.arribaOabajo()]);
398                            creados = 1;
399                        }
400                    }
401                }
402
403            }
404
405
406            /*
407             * Analisis que ocurren mientras el juego está ganado
408            */
409            else if (this.estadoDelJuego.equals("ganado")) {
410
411                donkeyKong.noMasViolencia();
412
413                jugador.cambiarImagen("mirando");
414                jugador.dibujar(entorno, 0);
415
416                entorno.dibujarRectangulo(400, 300, 200, 75, 0, Color.GREEN);
417                entorno.cambiarFont("terminal", 20, Color.WHITE);
418                entorno.escribirTexto("G A N A S T E", 335, 310);
419
420                /*
421                 * Analisis que ocurren mientras el juego está perdido
422                */
423            } else {
424                entorno.dibujarRectangulo(400, 300, 200, 75, 0, Color.GREEN);
425                entorno.cambiarFont("terminal", 20, Color.WHITE);
426                entorno.escribirTexto("G A M E   O V E R", 315, 310);
427
```

```
428                 jugador.cambiarY(3);
429                 jugador.cambiarImagen("saltando");
430                 jugador.dibujar(entorno, 90);
431 
432                 donkeyKong.noMasViolencia();
433             }
434 
435 
436 
437         }
438 
439         @SuppressWarnings("unused")
440         public static void main(String[] args) {
441             Juego juego = new Juego();
442         }
443     }
444 
```

```java
package juego;

import juego.Viga;
import java.awt.Image;

import entorno.Entorno;
import entorno.Herramientas;

public class Personaje {

    private int posx;
    private int posy;

    // Se reservan las referencias necesarias para cada animacion utilizada
    private Image mirandoIzquierda;
    private Image mirandoDerecha;
    private Image caminandoIzquierda;
    private Image caminandoDerecha;
    private Image saltandoIzquierda;
    private Image saltandoDerecha;
    private Image subiendo;
    private Image subiendo_quieto;

    // Esta referencia realizará un aliasing a la imagen que debe ser mostrada
    // Se muestra una por vez
    private Image imagenMario;

    private int tiempoSalto; // tick en el cual se ejecutó el último salto (o el salto actual)
    private boolean estaSaltando; // Indica si está saltando (ascendiendo) o no.
    private boolean estaCayendo; // Indica si está cayendo (es decir que sus pies no están tocando viga alguna.

    private boolean estaCercaEscalera; // Indica si el personaje se encuentra lo suficientemente cerca de una escalera
                                       // (para poder usarla)
    private boolean estaEnEscalera; // Indica si el personaje se encuentra dentro (usando) una escalera
    private int subidoAEscaleraNro; // Indica que el indice que corresponde a la posición de la escalera que se está
                                    // usando dentro del arreglo de escaleras

    private int sonando; // Ultimo archivo de sonido que se usó para caminar, hay 3 variantes.
    private int sonandoDesde; // tick en el cual se ejecutó el último sonido de caminar (ayuda a evitar que
                                    // suenen sonidos en cada tick)

    private boolean miraDerecha; // Esta variable indica si el personaje está mirando a derecha o no (Vital para
                                    // que se cargue la imagen correcta del personaje según los movimientos que
                                    // indique el usuario)

    public Personaje() {

        // Posición por defecto de spawn
        this.posx = 50;
        this.posy = 530;

        // direcciones URL de las imagenes y animaciones
        this.mirandoIzquierda = Herramientas.cargarImagen(
        "rsc/graficos/marito/mira-izquierda.png");
```

```
54          this.mirandoDerecha = Herramientas.cargarImagen(
            "rsc/graficos/marito/mira-derecha.png");
55          this.caminandoIzquierda = Herramientas.cargarImagen(
            "rsc/graficos/marito/camina-izquierda.gif");
56          this.caminandoDerecha = Herramientas.cargarImagen(
            "rsc/graficos/marito/camina-derecha.gif");
57          this.saltandoIzquierda = Herramientas.cargarImagen(
            "rsc/graficos/marito/salta-izquierda.png");
58          this.saltandoDerecha = Herramientas.cargarImagen(
            "rsc/graficos/marito/salta-derecha.png");
59          this.subiendo = Herramientas.cargarImagen(
            "rsc/graficos/marito/subiendo.gif");
60          this.subiendo_quieto = Herramientas.cargarImagen(
            "rsc/graficos/marito/quieto_subiendo.png");
61
62          // Por defecto no hubo salto y no momentos anterior al cero. Tampoco el
63          // personaje esta saltando, cayendo ni en una escalera ni cerca de alguna.
64          this.tiempoSalto = 0;
65          this.estaSaltando = false;
66          this.estaCayendo = false;
67          this.estaEnEscalera = false;
68          this.estaCercaEscalera = false;
69
70          this.sonando = 1;
71          this.sonandoDesde = 0;
72
73          // Por defecto la imagen a mostrarse es mirando a derecha
74          this.imagenMario = this.mirandoDerecha;
75
76          // Por defecto debe mirar a derecha
77          this.miraDerecha = true;
78      }
79
80
81      /*
82       * Es la función que indica si el jugador está tocando el barril pasado como
         parámetro
83       */
84      public boolean tocando(Barril[] barriles) {
85
86          for (int i = 0; i < barriles.length; i++) {
87
88              if (barriles[i] != null) {
89
90                  if (this.lateralDerecho() - barriles[i].lateralIzquierdo() > 0
91                          && this.lateralIzquierdo() - barriles[i].lateralIzquierdo
                            () < 0
92                          && this.obtenerPosPies() - barriles[i].superior() >= 3
93                          && this.obtenerPosCabeza() - barriles[i].pies() <= -10) {
94
95                      System.out.println("[" + i + "] Colision Derecha");
96                      return true;
97
98                  }
99
100                 if (this.lateralIzquierdo() - barriles[i].lateralDerecho() < 0
101                         && this.lateralDerecho() - barriles[i].lateralDerecho() >
                            0
102                         && this.obtenerPosPies() - barriles[i].superior() >= 3
103                         && this.obtenerPosCabeza() - barriles[i].pies() <= -10) {
104                     System.out.println("[" + i + "] Colision Izquierda");
105                     return true;
106
```

```
107                 }
108
109             }
110         }
111         return false;
112
113     }
114
115     /*
116      * Hacer Sonar.
117      *
118      * Esta función ejecuta el sonido de caminar pero evita que suene en cada tick
119      * donde se está caminando. Sino habría una bola de sonido indistinguible.
120      *
121      * Se le debe indicar el momento actual en ticks como parámetro.
122      *
123      * La función decide hacer sonar alguna de las 3 variantes de sonidos de pasos
124      * que hay. Y sólo hace sonar cuando la distancia entre el sonido anterior y
         el
125      * actual es de 40 ticks.
126      *
127      */
128
129     public void hacerSonar(int contador) {
130         if (this.sonando == 3 && contador > this.sonandoDesde + 40) {
131             Herramientas.play("rsc/sonidos/caminar" + String.valueOf(this.sonando
                ) + ".wav");
132             this.sonando = 1;
133             this.sonandoDesde = contador;
134         }
135
136         else if (this.sonando < 3 && contador > this.sonandoDesde + 40) {
137
138             Herramientas.play("rsc/sonidos/caminar" + String.valueOf(this.sonando
                ) + ".wav");
139             this.sonando++;
140             this.sonandoDesde = contador;
141
142         }
143
144     }
145
146     /*
147      * Pisando
148      *
149      * Esta funcion devuelve el indice que ocupa la viga en el arreglo de
         suelos. Si
150      * no se encuentra pisando, entonces devuelve -1.
151      *
152      * Requiere que se entregue el entorno y el arreglo de vigas como parámetros.
153      *
154      * Para saber si no está pisando la viga, el centro 'y' del personaje + 20
155      * pixeles (para llegar al pie del personaje) obtenerPosPies() debe poseer un
156      * valor distinto para la coordenada 'y' donde comienza cada viga (la posy -
157      * 12px) (int)suelos[i].dondeEmpiezaElSuelo().
158      *
159      * En el caso de que el personaje se encuentra pisando la viga. Queda por
160      * conocer si se encuentra dentro de todos los puntos 'x' que conforman el
         largo
161      * de la viga.
162      *
163      * Por eso la función analiza que el extremo derecho de la viga, sea pisada
         por
```

```
164         * al menos el lateral izquierdo del personaje, y lo mismo de forma invertida.
165         * Si no se cumple esta condición, el personaje está cayendo por estar fuera
            de
166         * la viga a pesar de estar a la altura de alguna de ellas.
167         *
168         *
169         *
170         */
171
172        public int pisando(Viga[] suelos) {
173
174            if (this.obtenerEstaEnEscalera() == false) {
175                for (int i = 0; i < suelos.length; i++) {
176
177                    if (this.obtenerPosPies() == (int) suelos[i].dondeEmpiezaElSuelo
                        ()) {
178
179                        if (this.lateralDerecho() < suelos[i].extremoIzquierdo()
180                                || this.lateralIzquierdo() > suelos[i].extremoDerecho
                                ()) {
181                            this.estaCayendo = true;
182                            return -1;
183
184                        } else {
185                            this.estaCayendo = false;
186                            return i;
187
188                        }
189                    }
190
191                }
192                this.estaCayendo = true;
193                return -1;
194
195            } else {
196
197                if (this.subidoAEscaleraNro > 4) {
198                    return this.subidoAEscaleraNro - 5;
199                } else {
200                    return this.subidoAEscaleraNro;
201                }
202            }
203
204        }
205
206        /*
207         * Esta función cambia el valor de estaCercaEscalera a true o false
            dependiendo
208         * si el personaje está cerca de una escalera como para poder subir o
            descender
209         * por ella.
210         *
211         * Esta función debe llamarse en cada tick del juego pero sólo si el personaje
212         * no se encuentra dentro de una escalera actualmente.
213         *
214         */
215
216        public void estoyCercaDeEscalera(Escaleras[] escaleras, Viga[] suelos) {
217
218            int hallado = 0;
219            int i = pisando(suelos);
220
221            // Sólo analiza la proximidad de una escalera, si la función pisando
```

```
            devuelve el
222         // índice de la viga pisada.
223         // No se analiza proximidad para valores -1 (en el aire) (se está
            cayendo).
224         if (i != -1 && this.estaCayendo == false) {
225
226             // Comprobación de escaleras para todos los pisos excepto el último
227             if (i != suelos.length - 1) {
228
229                 // Se analiza una escalera que comienza en el piso actual y sube
                    al próximo
230                 if ((escaleras[i].extremoInferior() - this.obtenerPosPies() <= 5
                    )) {
231
232                     if (escaleras[i].lateralDerecho() >= this.posx && escaleras[i
                        ].lateralIzquierdo() <= this.posx) {
233                         this.estaCercaEscalera = true;
234                         this.subidoAEscaleraNro = i;
235
236                         hallado += 1;
237                     }
238                 }
239
240                 // Se analiza una escalera adicional que comienza en el piso
                    actual y sube al
241                 // próximo o quizas no sube del todo
242                 if ((escaleras[i + 5].extremoInferior() - this.obtenerPosPies()
                    <= 5)) {
243
244                     if (escaleras[i + 5].lateralDerecho() >= this.posx
245                             && escaleras[i + 5].lateralIzquierdo() <= this.posx) {
246                         this.estaCercaEscalera = true;
247                         this.subidoAEscaleraNro = i + 5;
248
249                         hallado += 1;
250                     }
251                 }
252
253             }
254
255             // Comprobación de escaleras para todos los pisos excepto la planta
                baja
256             if (i != 0) {
257                 if (escaleras[i - 1].extremoSuperior() - this.obtenerPosPies() <=
                     10) {
258
259                     // Se analiza una escalera que termina en el piso actual y
                        desciende al inferior
260                     if (escaleras[i - 1].lateralDerecho() >= this.posx
261                             && escaleras[i - 1].lateralIzquierdo() <= this.posx) {
262                         this.estaCercaEscalera = true;
263                         this.subidoAEscaleraNro = i - 1;
264
265                         hallado += 1;
266                     }
267                 }
268
269                 if (escaleras[i + 4].extremoSuperior() - this.obtenerPosPies() <=
                     10) {
270
271                     // Se analiza una escalera adicional que termina en el piso
                        actual y desciende
272                     // al inferior
```

```
273                     if (escaleras[i + 4].lateralDerecho() >= this.posx
274                             && escaleras[i + 4].lateralIzquierdo() <= this.posx) {
275                         this.estaCercaEscalera = true;
276                         this.subidoAEscaleraNro = i + 4;
277 
278                         hallado += 1;
279                     }
280                 }
281 
282             }
283 
284         }
285 
286         if (hallado == 0) {
287             this.estaCercaEscalera = false;
288 
289         }
290 
291     }
292 
293     /*
294      * Realiza los calculos geométricos para saber que un barril fue correctamente
295      * saltado.
296      */
297 
298     public boolean saltandoBarril(Barril barril) {
299 
300         if ((this.posx + 1 == barril.centroX() || this.posx - 1 == barril.centroX
        () || this.posx == barril.centroX())
301                 && this.obtenerPosPies() - barril.superior() <= 0 && this.
                obtenerPosPies() - barril.superior() > -50
302                 && barril.fueSaltado() == false && this.obtenerEstaEnEscalera()
                == false) {
303 
304             Herramientas.play("rsc/sonidos/salta_barril.wav");
305             barril.saltado();
306             return true;
307 
308         } else {
309             return false;
310         }
311 
312     }
313 
314     /*
315      * Dibuja al personaje. Se deben calcular las situaciones y cambiar las
316      * variables previamente con otros métodos.
317      */
318 
319     public void dibujar(Entorno entorno, int rotacion) {
320         entorno.dibujarImagen(imagenMario, this.posx, this.posy, rotacion, 0.090);
321 
322     }
323 
324     /*
325      * Devuelven la posicion extrema lateral correspondiente
326      */
327     public int lateralDerecho() {
328         return posx + 15;
329     }
330 
331     public int lateralIzquierdo() {
332         return posx - 15;
```

```
333         }
334 
335         /*
336          * Setters de las posiciones X e Y
337          */
338         public void cambiarY(int pixeles) {
339             this.posy = this.posy + pixeles;
340         }
341 
342         public void cambiarX(int pixeles) {
343             this.posx = this.posx + pixeles;
344         }
345 
346         /*
347          * Devuelven la posicion extremas verticales correspondiente
348          */
349 
350         public int obtenerPosPies() {
351             return this.posy + 20;
352         }
353 
354         public int obtenerPosCabeza() {
355             return this.posy - 20;
356         }
357 
358         /*
359          * Getters
360          */
361 
362         public boolean obtenerEstaEnEscalera() {
363             return this.estaEnEscalera;
364         }
365 
366         public int obtenerMomentoDeSalto() {
367             return this.tiempoSalto;
368         }
369 
370         public boolean obtenerEstaCayendo() {
371             return this.estaCayendo;
372         }
373 
374         public boolean obtenerEstaCercaEscalera() {
375             return this.estaCercaEscalera;
376         }
377 
378         public int obtenerSubidoAEscaleraNro() {
379             return this.subidoAEscaleraNro;
380         }
381 
382         public boolean obtenerMiraDerecha() {
383             return this.miraDerecha;
384         }
385 
386         public boolean obtenerEstaSaltando() {
387             return this.estaSaltando;
388         }
389 
390         /*
391          * Setters
392          */
393 
394         public void cambiarMomentoDeSalto(int i) {
395             this.tiempoSalto = i;
```

```
396
397         }
398
399         public void cambiarEstaEnEscalera(boolean escalera) {
400             this.estaEnEscalera = escalera;
401         }
402
403         public void cambiarMiraDerecha(boolean mira) {
404             this.miraDerecha = mira;
405         }
406
407         public void cambiarEstaSaltando(boolean salta) {
408             this.estaSaltando = salta;
409         }
410
411         /*
412          * Ayudan a cambiar por la imagen correcta, según el String indicado, y
             según a
413          * que lado esté mirando el personaje
414          */
415         public void cambiarImagen(String s) {
416
417             if (s.equals("mirando") && !this.imagenMario.equals(mirandoIzquierda) &&
             !this.miraDerecha) {
418                 imagenMario = mirandoIzquierda;
419             }
420
421             else if (s.equals("mirando") && !this.imagenMario.equals(mirandoDerecha)
             && this.miraDerecha) {
422                 imagenMario = mirandoDerecha;
423             }
424
425             else if (s.equals("caminando") && !this.imagenMario.equals(
             caminandoIzquierda) && !this.miraDerecha) {
426                 imagenMario = caminandoIzquierda;
427             }
428
429             else if (s.equals("caminando") && !this.imagenMario.equals(
             caminandoDerecha) && this.miraDerecha) {
430                 imagenMario = caminandoDerecha;
431             }
432
433             else if (s.equals("saltando") && !this.imagenMario.equals(
             saltandoIzquierda) && !this.miraDerecha) {
434                 imagenMario = saltandoIzquierda;
435             }
436
437             else if (s.equals("saltando") && !this.imagenMario.equals(saltandoDerecha
             ) && this.miraDerecha) {
438                 imagenMario = saltandoDerecha;
439             }
440
441             else if (s.equals("subiendo") && !this.imagenMario.equals(subiendo)) {
442                 imagenMario = subiendo;
443             } else if (s.equals("quieto") && !this.imagenMario.equals(subiendo_quieto
             )) {
444                 imagenMario = subiendo_quieto;
445             }
446
447         }
448
449     }
450
```

```java
package juego;

import java.util.Random;

import java.awt.Image;

import entorno.Entorno;
import entorno.Herramientas;

public class Donkey {

    private int ultimoLanzamiento;

    private int lanzarRandom;
    private String violencia;

    public Donkey() {

        this.ultimoLanzamiento = 0;
        this.violencia = "violento";

    }

    public void gorilear(Entorno entorno, int contador) {

        if (contador - ultimoLanzamiento < 30) {

            Image gorila = Herramientas.cargarImagen(
            "rsc/graficos/donkey/tirar.gif");
            Image stock = Herramientas.cargarImagen(
            "rsc/graficos/barriles/stock.png");
            entorno.dibujarImagen(gorila, 100, 30, 0, 0.19);
            entorno.dibujarImagen(stock, 30, 32, 0, 0.13);

        } else {
            Image gorila = Herramientas.cargarImagen(
            "rsc/graficos/donkey/gorilear.gif");
            Image stock = Herramientas.cargarImagen(
            "rsc/graficos/barriles/stock.png");
            entorno.dibujarImagen(gorila, 100, 30, 0, 0.19);
            entorno.dibujarImagen(stock, 30, 32, 0, 0.13);
        }
    }

    public boolean decidir(int contador) {

        Random rnd = new Random();

        if (this.violencia.equals("violento")) {
            if (this.lanzarRandom == contador) {
                this.ultimoLanzamiento = contador;
                this.lanzarRandom = 0;

                return true;
            }

            if (contador >= this.ultimoLanzamiento + rnd.nextInt(220) + 60 &&
            this.lanzarRandom == 0) {

                lanzarRandom = rnd.nextInt(250) + contador;

                return false;
            }
```

```
59
60                  return false;
61
62              }
63
64              return false;
65          }
66
67          public void noMasViolencia() {
68              this.violencia = "noviolento";
69          }
70
71          public int arribaOabajo() {
72              Random rnd = new Random();
73              int eleccion = rnd.nextInt(60);
74              if (eleccion % 3 == 0) {
75                  return -3;
76              } else {
77                  return -1;
78              }
79          }
80
81      }
82
```

```java
package juego;

import java.awt.Color;
import entorno.Entorno;
import entorno.Herramientas;

public class Viga {

    private int pos;
    private double x;
    private double y;
    private double largo;
    private double alto;

    /*
     * Este constructor, ya tiene definida de forma estricta y estática las
     * posiciones de las vigas
     */

    public Viga(int pos) {

        switch (pos) {
        case 1:
            this.x = 400;
            this.y = 575;
            this.largo = 820;
            this.alto = 25;
            break;
        case 2:
            this.x = 325;
            this.y = 475;
            this.largo = 700;
            this.alto = 25;
            break;
        case 3:
            this.x = 475;
            this.y = 375;
            this.largo = 700;
            this.alto = 25;
            break;
        case 4:
            this.x = 325;
            this.y = 275;
            this.largo = 700;
            this.alto = 25;
            break;
        case 5:
            this.x = 475;
            this.y = 175;
            this.largo = 700;
            this.alto = 25;
            break;
        case 6:
            this.x = 325;
            this.y = 75;
            this.largo = 700;
            this.alto = 25;
            break;
        }

        this.pos = pos;

    }
```

```
64
65      /*
66       * Dibujar
67       *
68       * Esta función debe ser llamada en cada tick por cada viga que exista. Por
         cada
69       * tick dibuja la viga.
70       *
71       * La construcción gráfica de la viga tiene un condimento especial. Para dar
         la
72       * sensación de que es una estructura metálica, se dibuja un rectángulo de
         color
73       * rojo de fondo, y sobre el de forma estratégica, triángulos del mismo color
74       * que el fondo, en juegos de a dos. Cada uno invertido 90° con respecto al
75       * anterior.
76       *
77       */
78      public void dibujar(Entorno entorno) {
79
80          // Rectángulo básico de la viga, respetando los valores indicados por el
81          // constructor
82          entorno.dibujarRectangulo(this.x, this.y, this.largo, this.alto, 0.0,
            Color.RED);
83
84          // El extremo izquierdo de la viga corrido 10px
85          double paso = this.x - (this.largo / 2) + 10;
86
87          // Se decide que la suma de la base de un triangulo, la punta del
            triángulo
88          // adyacente y un espacio
89          // extra sea la 25ava parte del ancho de la viga - 4 pixeles
90          double triangulos = (this.largo / 25) - 4;
91
92          // Indica la cantidad de parejas de triangulos dibujados. Una pareja es un
93          // triangulo con la punta hacia arriba
94          // y el otro con la punta hacia abajo.
95          int dibujados = 0;
96
97          // Este bucle dibuja la pareja de triángulos a lo largo de la viga.
98          while (dibujados <= triangulos) {
99
100             entorno.dibujarTriangulo(paso, this.y, 21, 21, Herramientas.radianes(
                90), java.awt.Color.BLACK);
101             paso += 14;
102             entorno.dibujarTriangulo(paso, this.y, 21, 21, Herramientas.radianes(
                270), java.awt.Color.BLACK);
103             paso += 14;
104             dibujados += 1;
105
106         }
107
108     }
109
110     // Devuelve la posX
111     public int getPosx() {
112         return (int) this.x;
113     }
114
115     // Devuelve la posY
116     public int getPosy() {
117         return (int) this.y;
118     }
119
```

```
120         // Devuelve el Ancho
121         public int getAncho() {
122             return (int) this.largo;
123         }
124 
125         public int getPos() {
126             return pos;
127         }
128 
129         /*
130          * DondeEmpiezaElSuelo
131          *
132          * Esta función devuelve el valor en el 'y' en el cual comienza la viga.
133          * Sabiendo que el 'y' se encuentra en el centro.
134          *
135          */
136         public double dondeEmpiezaElSuelo() {
137 
138             return this.y - (this.alto / 2) - 1;
139         }
140 
141         public double dondeTerminaElTecho() {
142 
143             return this.y + (this.alto / 2) + 1;
144         }
145 
146         /*
147          * Esta función indica donde comienza la viga en el eje X.
148          */
149         public int extremoIzquierdo() {
150             return this.getPosx() - this.getAncho() / 2;
151         }
152 
153         /*
154          * Esta función indica donde termina la viga en el eje X.
155          */
156         public int extremoDerecho() {
157             return this.getPosx() + this.getAncho() / 2;
158         }
159 
160     }
161 
```

```java
package juego;

import java.awt.Color;
import java.util.Random;

import entorno.Entorno;

public class Escaleras {

    private double x;
    private double y;
    private double ancho;
    private double alto;

    public Escaleras(int pos, Viga[] suelos) {

        Random rnd = new Random();
        boolean escaleraCompleta;
        this.ancho = 30;

        /*
         * Hay 2 tipos de escaleras. Las obligatorias que deben estar completas y ocupan
         * las "pos" del 0 al 4. Luego las adicionales que pueden estar imcompletas o
         * no. Ocupan las "pos" del 5 al 9.
         */

        // Para las obligatorias
        if (pos < 5) {

            int offsetEscalera = rnd.nextInt(50);

            if (pos % 2 == 0) {

                this.x = suelos[pos + 1].extremoDerecho() - 30 - offsetEscalera;

            } else {

                this.x = suelos[pos + 1].extremoIzquierdo() + 30 + offsetEscalera;
            }

            // El punto mediatriz del segmento que representa la distancia entre la
            // superficie superior de la viga que funciona
            // como el suelo, y la superficie superior de la viga siguiente. Dicho punto es
            // el centro Y de la escalera.
            this.y = ((suelos[pos].dondeEmpiezaElSuelo() - suelos[pos + 1].dondeEmpiezaElSuelo()) / 2)
                    + suelos[pos + 1].dondeEmpiezaElSuelo();

            // El alto es la distancia entre la superficie superior de la viga que funciona
            // como el suelo, y la superficie superior de la viga siguiente.
            this.alto = suelos[pos].dondeEmpiezaElSuelo() - suelos[pos + 1].dondeEmpiezaElSuelo();

        }

        // Para las escaleras adicionales
        else {

```

```
57              int offsetEscalera = rnd.nextInt(150);
58
59              // Las escaleras pares, conectan con vigas superiores que no tocan
                el extremo
60              // derecho.
61              // Las escaleras impares, conectan con vigas superiores que no tocan
                el extremo
62              // izquierdo.
63              // A su vez, la posición de la escalera, está aumentada 4 veces con
                repecto al
64              // indice de la viga superior
65              // con la cual debe conectar.
66              if (pos % 2 == 0) {
67
68                  this.x = suelos[pos - 4].extremoDerecho() - 250 - offsetEscalera;
69
70              } else {
71
72                  this.x = suelos[pos - 4].extremoIzquierdo() + 250 +
                    offsetEscalera;
73              }
74
75              // Para todas las escaleras adicionales, expecto la de planta baja.
76              if (pos != 5) {
77
78                  // Generamos un random que ayudará a elegir si la escalera se
                    presentará de
79                  // forma completa o no
80                  int eleccionEscaleraCompleta = rnd.nextInt(300);
81
82                  // De esta forma existe sólo un 33% de posibilidades de que
                    aparezca completa.
83                  if (eleccionEscaleraCompleta % 3 == 0) {
84
85                      escaleraCompleta = true;
86                  } else {
87                      escaleraCompleta = false;
88                  }
89
90                  // La escalera adicional de planta baja, nunca estará completa
91              } else {
92                  escaleraCompleta = false;
93              }
94
95              if (escaleraCompleta) {
96                  // Método normal para calcular la el centro "Y" y la altura.
97                  this.y = ((suelos[pos - 5].dondeEmpiezaElSuelo() - suelos[pos - 4
                    ].dondeEmpiezaElSuelo()) / 2)
98                          + suelos[pos - 4].dondeEmpiezaElSuelo();
99
100                 this.alto = suelos[pos - 5].dondeEmpiezaElSuelo() - suelos[pos -
                    4].dondeEmpiezaElSuelo();
101
102             }
103
104             else {
105
106                 // Al método normal le corremos 25 pixeles hacia abajo y la
                    altura es la mitad.
107                 // impidiendo que la escalera esté completa
108                 this.y = ((suelos[pos - 5].dondeEmpiezaElSuelo() - suelos[pos - 4
                    ].dondeEmpiezaElSuelo()) / 2)
109                         + suelos[pos - 4].dondeEmpiezaElSuelo() + 25;
```

```
110
111                 this.alto = (suelos[pos - 5].dondeEmpiezaElSuelo() - suelos[pos -
                     4].dondeEmpiezaElSuelo()) / 2;
112
113             }
114
115         }
116
117     }
118
119     public void dibujar(Entorno entorno) {
120
121         // Rectángulo básico de la viga, respetando los valores indicados por el
122         // constructor
123         entorno.dibujarRectangulo(this.x, this.y, this.ancho, this.alto, 0.0,
            Color.BLUE);
124
125         double paso = this.y + (this.alto / 2) - 3;
126
127         // Se decide que la suma de la base de un triangulo, la punta del
            triángulo
128         // adyacente y un espacio
129         // extra sea la 25ava parte del ancho de la viga - 4 pixeles
130         double rectangulos = (this.alto / 10);
131
132         // Indica la cantidad de parejas de triangulos dibujados. Una pareja es un
133         // triangulo con la punta hacia arriba
134         // y el otro con la punta hacia abajo.
135         int dibujados = 0;
136
137         // Este bucle dibuja la pareja de triángulos a lo largo de la viga.
138
139         while (dibujados <= rectangulos) {
140
141             entorno.dibujarRectangulo(this.x, paso, 28, 9, 0.0, java.awt.Color.
                BLACK);
142             paso -= 10;
143
144             dibujados += 1;
145
146         }
147
148     }
149
150     public int lateralDerecho() {
151         return (int) this.x + 15;
152     }
153
154     public int lateralIzquierdo() {
155         return (int) this.x - 15;
156     }
157
158     public int extremoSuperior() {
159         return (int) (this.y - (this.alto / 2));
160     }
161
162     public int extremoInferior() {
163         return (int) (this.y + (this.alto / 2));
164     }
165
166 }
167
```

```java
package juego;

import java.awt.Image;
import java.util.Random;

import juego.Viga;
import entorno.Entorno;
import entorno.Herramientas;

public class Barril {

    private double posx;
    private double posy;
    private int diametro;
    private double escala;

    private Image spin_izquierda;
    private Image spin_derecha;
    private Image cayendo;

    // Conserva la dirección en la que estaba moviendo el barril "derecha" o
    // "izquierda"
    private String ultima;

    /*
     * Es necesaria una variable que indique que ee barril fue saltado para que no
     * sea contado doble en las siguientes situaciones: En el proceso de salto, la
     * función que detecta el correcto salto puede dar varios positivos durante
     una
     * cantidad de ticks cercanos entre si. Atrapando el primer tick donde se
     * detecta que el barril fue saltado, se evita que en los ticks siguientes
     donde
     * también es positivo, se cuenten los puntos inecesariamente. Evitar que el
     * jugador salte el barril en una viga y luego intente saltarlo en una viga
     * inferior para contador doble puntaje.
     */

    private boolean saltado;

    // Conocer si está cayendo por escalera permite detener el movimiento hacia
    // izquierda o derecha. Ayuda a diferenciar una caida desde una viga con
    // respecto a la de escalera.
    private boolean cayendoPorEscalera;

    // Como un barril tarda varios ticks en atravesar el ancho de una escalera.
    // Necesitamos indicar en que tick se tomó la decisión de caer o no por la
    // misma. Para que en el tick siguiente
    // no sobreescriba la decisión. Toma una decisión por escalera,y bloquea
    decidir
    // de nuevo por un cierto tiempo.
    private int ultimaEleccion;

    // Para animar la caida por escalera se utiliza esta variable para
    intercambiar
    // entre 10° y -10°.
    private int anguloRotacion;

    // Asiste en la elección del anguloRotacion
    private boolean sentidoRotacionDerecha;

    public Barril(Viga vigasuelo) {

        this.diametro = 17;
```

```
60          this.escala = (double) this.diametro / 108;
61
62          /*
63           * Se crea el barril en la posición desde donkey decidió arrojarlo
64           */
65          if (vigasuelo.getPos() == 6) {
66              this.posy = (int) vigasuelo.dondeEmpiezaElSuelo() - 20;
67              this.ultima = "derecha";
68              this.posx = 120;
69          } else if (vigasuelo.getPos() == 4) {
70              this.posy = (int) vigasuelo.dondeEmpiezaElSuelo() - 190;
71              this.ultima = "izquierda";
72              this.posx = 120;
73          }
74
75          this.spin_izquierda = Herramientas.cargarImagen(
            "rsc/graficos/barriles/spin-izquierda.gif");
76          this.spin_derecha = Herramientas.cargarImagen(
            "rsc/graficos/barriles/spin-derecha.gif");
77          this.cayendo = Herramientas.cargarImagen(
            "rsc/graficos/barriles/cayendo.png");
78
79          this.saltado = false;
80          this.ultimaEleccion = 0;
81          this.anguloRotacion = 0;
82          this.sentidoRotacionDerecha = true;
83
84      }
85
86      /*
87       * Esta función al ser llamada retorna si el barril debe destruirse por estar
88       * fuera de la pantalla del juego
89       *
90       */
91      public boolean deboDestruirme(Entorno entorno, Viga[] suelos) {
92          if (this.posx < 15 && this.pisando(suelos) == 0) {
93              return true;
94          } else {
95              return false;
96          }
97      }
98
99      /*
100      * Esta función se encarga de dibujar en pantalla al barril y calcular su
101      * moviento.
102      *
103      */
104
105     public void dibujar(Entorno entorno, int contador, Viga[] suelos, Escaleras[]
         escaleras) {
106
107         // Si está rodando sobre el suelo
108         if (pisando(suelos) != -1) {
109
110             // Si la decisión de caer por la escalera es afirmativa
111             if (caerPorEscalera(escaleras, suelos, contador)) {
112
113                 // Este movimiento inicial detendrá en el próximo los
                    movimientos a izquierda o
114                 // derecha
115                 // Porque la función pisando devolverá -1
116                 this.posy = this.posy + 1;
117                 entorno.dibujarImagen(cayendo, this.posx, this.posy, this.
```

```
                    anguloRotacion, this.escala);
118
119             } else {
120
121                 // En vigas con indice par desplazar a izquierda
122                 if (this.posx >= 10 && pisando(suelos) % 2 == 0) {
123                     this.posx = this.posx - 1.7;
124                     entorno.dibujarImagen(spin_izquierda, this.posx, this.posy, 0
                        , this.escala);
125                     this.ultima = "izquierda";
126                 }
127
128                 // En vigas con indice impar desplazar a derecha
129                 else if (this.posx <= 800 && pisando(suelos) % 2 == 1) {
130                     this.posx = this.posx + 1.7;
131                     entorno.dibujarImagen(spin_derecha, this.posx, this.posy, 0,
                        this.escala);
132                     this.ultima = "derecha";
133                 }
134
135             }
136         }
137
138         // Si NO está rodando sobre el suelo
139         if (pisando(suelos) == -1) {
140
141             // cambia la posición con respecto al eje "y" hacia abajo
142             this.posy += 1;
143
144             // Si la caida está producida por caer por escalera
145             if (this.cayendoPorEscalera) {
146
147                 // Cada vez que el contador es divisible por 10, entonces hay
                    que cambiar el
148                 // sentido de rotación
149                 // Esto es puramente gráfico, da la sensación de un barril
                    cayendo por
150                 // escaleras.
151                 if (contador % 10 == 0) {
152                     if (this.sentidoRotacionDerecha) {
153                         this.sentidoRotacionDerecha = false;
154                     } else {
155                         this.sentidoRotacionDerecha = true;
156                     }
157                 }
158
159                 // Dependiendo el sentido de Rotación, el engulo es positivo o
                    negativo
160                 if (this.sentidoRotacionDerecha) {
161                     this.anguloRotacion = 10;
162                 } else {
163                     this.anguloRotacion = -10;
164                 }
165
166                 // dibujar la caida por escalera
167                 entorno.dibujarImagen(cayendo, this.posx, this.posy, this.
                    anguloRotacion, this.escala);
168
169                 // Si la caida es producida por el final de una viga
170             } else {
171
172                 // Si venia desplazandose a derecha pero está cayendo y hay
                    espacio en el x,
```

```
173                 // se sigue desplazando a derecha
174                 if (this.posx <= 800 && this.ultima.equals("derecha")) {
175                     this.posx = this.posx + 1.7;
176                     entorno.dibujarImagen(spin_derecha, this.posx, this.posy, 0,
                        this.escala);
177                 }
178
179                 // De lo contrario hay que indicarle que en el próximo tick se
                    desplace a
180                 // izquierda.
181                 else {
182                     this.ultima = "izquierda";
183                 }
184
185                 // Si venia desplazandose a izquierda pero está cayendo y hay
                    espacio en el x,
186                 // se sigue desplazando a izquierda
187                 if (this.posx >= 10 && this.ultima.equals("izquierda")) {
188                     this.posx = this.posx - 1.7;
189                     entorno.dibujarImagen(spin_izquierda, this.posx, this.posy, 0
                        , this.escala);
190
191                 } else {
192                     // De lo contrario hay que indicarle que en el próximo tick
                        se desplace a
193                     // derecha.
194                     this.ultima = "derecha";
195                 }
196
197             }
198         }
199
200     }
201
202     // Igual que pisando de Personaje
203     public int pisando(Viga[] suelos) {
204
205         for (int i = 0; i < suelos.length; i++) {
206
207             if (this.pies() == (int) suelos[i].dondeEmpiezaElSuelo()) {
208
209                 if (this.lateralDerecho() < suelos[i].extremoIzquierdo()
210                         || this.lateralIzquierdo() > suelos[i].extremoDerecho()) {
211
212                     return -1;
213
214                 } else {
215
216                     return i;
217
218                 }
219             }
220
221         }
222
223         return -1;
224
225     }
226
227     /*
228      * Esta función toma la decisión de decidir si el barril caerá por la
        siguiente
229      * escalera.
```

```
230         */
231
232         public boolean caerPorEscalera(Escaleras[] escaleras, Viga[] suelos, int
            contador) {
233
234             // Sólo se toma la decisión de nuevo, si ha pasado el suficiente tiempo
                (o sea
235             // que se está decisión sobre una escalera
236             // diferente a la anterior decidida.
237             if (this.ultimaEleccion + 30 < contador) {
238
239                 Random rnd = new Random();
240                 int numero = rnd.nextInt(1200);
241
242                 // i es el piso actual por el cual rueda el barril
243                 int i = pisando(suelos);
244
245                 // Comprobación de escaleras para todos los pisos excepto la planta
                    baja
246                 if (i != 0) {
247
248                     /*
249                      * Analisis de una escalera cerca de tipo obligatoria, se toma
                        la decisión si se
250                      * cumple una segunda condición: Estar cerca de una escalera, de
                        forma de que
251                      * exista posibilidad en el proximo tick de caer por ella.
252                      */
253
254                     // A la altura de los pies del barril
255                     if (escaleras[i - 1].extremoSuperior() - this.pies() <= 10) {
256
257                         // Entre las coordinadas x del barril debe haber una escalera
258                         if (escaleras[i - 1].lateralDerecho() - 9 >= this.posx
259                                 && escaleras[i - 1].lateralIzquierdo() + 9 <= this.
                                    posx) {
260
261                             // Si el random es divisible por 6 cae, sino no. 1/6 de
                                posibilidades de caer.
262                             if (numero % 6 == 0) {
263                                 this.cayendoPorEscalera = true;
264                                 this.ultimaEleccion = contador;
265                                 return true;
266                             } else {
267                                 this.cayendoPorEscalera = false;
268                                 this.ultimaEleccion = contador;
269                                 return false;
270                             }
271                         }
272                     }
273
274                     /*
275                      * Mismo analisis de una escalera cerca pero de tipo adicional,
                        se toma la
276                      * decisión si se cumple una segunda condición: Estar cerca de
                        una escalera, de
277                      * forma de que exista posibilidad en el proximo tick de caer
                        por ella.
278                      */
279
280                     // A la altura de los pies del barril (Útil para eliminar
                        escaleras no
281                     // completas)
```

```
282                     if (escaleras[i + 4].extremoSuperior() - this.pies() <= 10) {
283
284                         // Entre las coordinadas x del barril debe haber una escalera
285                         if (escaleras[i + 4].lateralDerecho() >= this.posx
286                                 && escaleras[i + 4].lateralIzquierdo() <= this.posx) {
287
288                             // Si el random es divisible por 6 cae, sino no. 1/6 de
                                posibilidades de caer.
289                             if (numero % 6 == 0) {
290                                 this.cayendoPorEscalera = true;
291                                 this.ultimaEleccion = contador;
292                                 return true;
293                             } else {
294                                 this.cayendoPorEscalera = false;
295                                 this.ultimaEleccion = contador;
296                                 return false;
297                             }
298                         }
299                     }
300
301                 }
302             }
303             // Para todos los demás casos no hay caida posible por escalera
304             cayendoPorEscalera = false;
305             return false;
306
307         }
308
309         public int pies() {
310             return (int) this.posy + diametro / 2 - 2;
311         }
312
313         public int superior() {
314             return (int) this.posy - diametro / 2 + 2;
315         }
316
317         public int lateralDerecho() {
318             return (int) this.posx + diametro / 2;
319         }
320
321         public int lateralIzquierdo() {
322             return (int) this.posx - diametro / 2;
323         }
324
325         public int centroX() {
326             return (int) this.posx;
327         }
328
329         public void saltado() {
330             this.saltado = true;
331         }
332
333         public boolean fueSaltado() {
334             return this.saltado;
335         }
336
337     }
```

```java
package juego;

import entorno.Entorno;
import java.awt.Color;

public class Puntaje {
    private int puntos;

    public Puntaje() {
        this.puntos = 0;
    }

    public void saltarbarril() {

        this.puntos += 15;
    }

    public void ganar() {

        this.puntos += 100;
    }

    public void dibujar(Entorno entorno) {

        entorno.cambiarFont("terminal", 18, Color.GREEN);

        entorno.escribirTexto("Puntos: " + String.valueOf(this.puntos), 685, 15);
    }

}
```