Pilas: Un Primer Juego

disparar_a_monos.py

Entre los ejemplos que incorpora Pilas, tenemos una buena primera aproximación con el proyecto **disparar_a_monos.py**, pues muestra un esqueleto general de cara al jugador.

El juego consiste en un pequeño torreta que ha de disparar a los monos que se generan al azar en pantalla y que intentan llegar hasta él para destruirlo. El juego incluye un sencillo marcador de puntuación, un control de sonido, un sistema de bonus y avisos de texto en pantalla. Tanto sonidos como imágenes están incorporados en Pilas, así que no necesitamos ningún recurso añadido.

En casa paso que demos para llegar a nuestro objetivo, pondremos lo que sea nuevo o lo que modifiquemos con fondo gris y en **negrita**. ¿Estamos preparados?

¡Escribamos código!

paso1.py

El primer paso que vamos a dar es situar el escenario, es decir, elegir un fondo, preparar un marcador, poner un control de sonido:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import pilas

pilas.iniciar()

# Usar un fondo estándar
pilas.fondos.Pasto()

# Añadir un marcador
puntos = pilas.actores.Puntaje(x=230, y=200, color=pilas.colores.blanco)
puntos.magnitud = 40

# Añadir el conmutador de Sonido
pilas.actores.Sonido()

# Arrancar el juego
pilas.ejecutar()
```

Lo primero, como ya sabemos, es importar pilas-engine e inicializarlo, en este caso creando la ventana por defecto de 640x480 pixeles.

A continuación, establecemos como fondo uno de los incorporados en el motor a través del módulo **pilas.fondos**. El elegido, **Pasto**, tiene una textura verde cercana a la selvática jungla donde viven los monos :-)...

Toca el turno del marcador. Entre los actores predefinidos en el módulo **pilas.actores** tenemos uno que se encarga precisamente de ello; **Puntaje**. Al crearlo y almacenarlo en la variable **puntos** para su uso posterior, aprovechamos a colocarlo en la posición adecuada, cerca de la esquina superior derecha (recuerda que el centro de la ventana tiene coordenadas **x=0** e **y=0**). El color blanco que va a tener lo indicamos usando, nuevamente, uno de los predefinidos en otro módulo: **pilas.colores**.

En realidad, la clase de actores **Puntaje** es una clase derivada de otra de Pilas, **Texto**, con lo que hereda sus características, atributos y métodos. Entre ellos, la propiedad **magnitud** indica el tamaño que tendrá el texto (del marcador, en este caso) y como queremos que se vea en grande, lo ponemos a un valor de **40**.

Por último, pasamos al sonido. Como puede verse en la captura de pantalla del juego, queremos poner un botón que permita activar/desactivar el sonido haciendo click sobre él. Nuevamente, Pilas contiene esta funcionalidad de serie...: El actor **Sonido**.

En el intérprete de Pilas puedes probar y escribir

```
pilas.ver(pilas.actores.Sonido)
```

para que veas su código y como está implementado. Allí comprobarás como, efectivamente, el actor se crea directamente en la esquina inferior derecha. Además también incluye la modificación del icono para mostrar el estado del botón y un aviso de texto cuando éste se pulse. El sistema de avisos de texto de Pilas es muy elegante y poco intrusivo; aparece durante un breve periodo de tiempo en la parte inferior con un fondo semitransparente.

¡Ya estamos! Guarda el archivo con el nombre de **paso1.py** y arrástralo sobre la ventana para verlo en ejecución. Pulsa el botón pulsando su icono. ¿No está mal para tan pocas lineas, verdad?

paso2.py

Vamos a añadir ahora la torreta que va a manejar el jugador. Éste es el código:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import pilas

pilas.iniciar()

# Variables y Constantes
balas_simples = pilas.actores.Bala
monos = []


# Funciones
def mono_destruido():
    pass

# Usar un fondo estándar
pilas.fondos.Pasto()

# Añadir un marcador
puntos = pilas.actores.Puntaje(x=230, y=200, color=pilas.colores.blanco)
puntos.magnitud = 40

# Añadir el conmutador de Sonido
pilas.actores.Sonido()

# Añadir la torreta del jugador
torreta = pilas.actores.Torreta(municion_bala_simple=balas_simples,
                                enemigos=monos,
                                cuando_elimina_enemigo=mono_destruido)

# Arrancar el juego
pilas.ejecutar()
```

Estamos usando el actor de Pilas **Torreta** que vamos a almacenar para un futuro uso en la variable **torreta**. En su creación, entre los argumentos que podemos pasarle, hemos usado los tres fundamentales:

• En primer lugar hemos de indicarle cual es la munición que va a emplear con el argumento **municion_bala_simple**. Para ello, al principio del programa, hemos

definido la variable **balas_simples** con el tipo de munición deseado, en este caso la clase correspondiente al actor Bala (que, por cierto, es la que usaría el propio Pilas por defecto):

```
balas_simples = pilas.actores.Bala
```

Observa que escribimos **pilas.actores.Bala** y no **pilas.actores.Bala()** ya que estamos indicando la clase de actor que vamos a usar y no lo estamos creando.

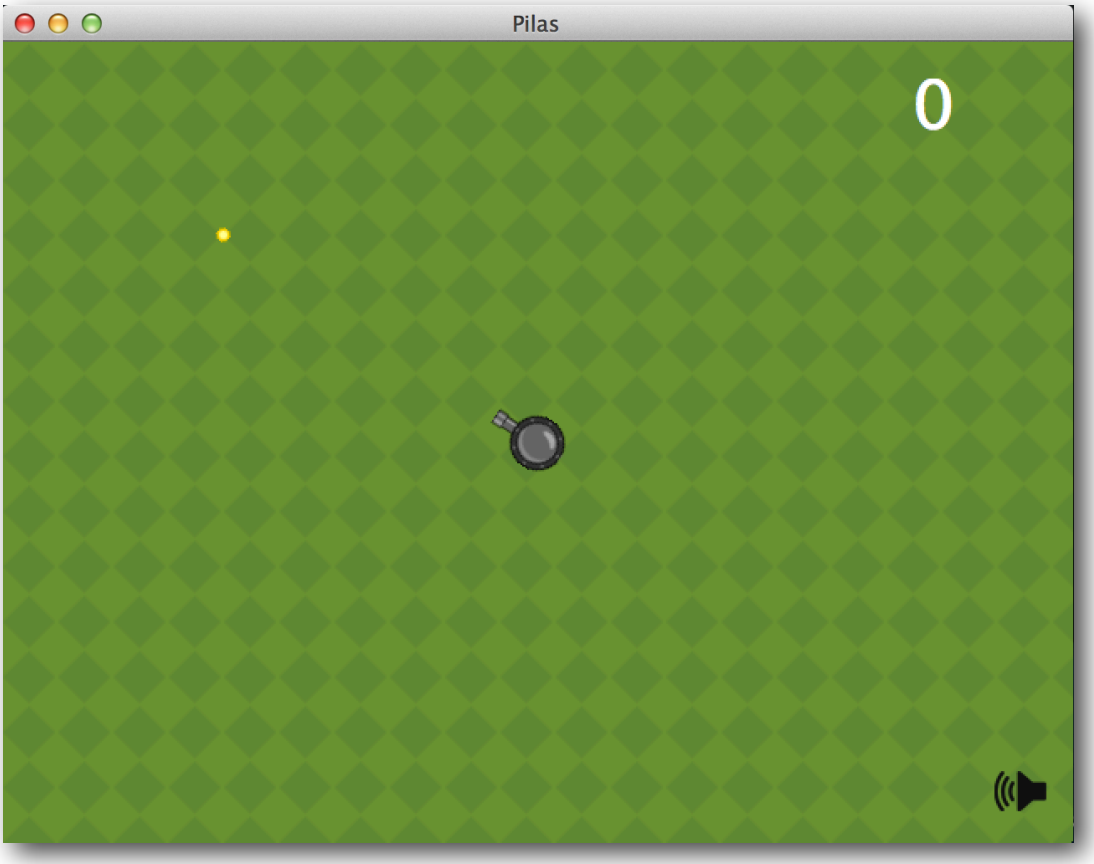
• Lo segundo que le pasamos, el argumento **enemigos**, es el grupo de enemigos a los que se va a disparar. Como en este punto del desarrollo no lo hemos decidido, hemos definido previamente la variable **monos** como una lista vacía.

• El tercer y último parámetro que le proporcionamos al constructor de la torreta, **cuando_elimina_enemigo**, ha de ser una función que se llamará cuando la munición que disparamos impacte con los enemigos. De momento no queremos concretar más detalles, así que hemos definido al principio del código una función **mono_destruido()** que no hace nada, usando **pass**:

```
def mono_destruido():
    pass
```

Observa, de nuevo, que en la creación de la torreta escribimos **mono_destruido** y no **enemigo_destruido()** ya que queremos pasarle la función que ha de usarse y no el resultado de ejecutarla.

Hecho. Guarda el código con el nombre **paso2.py** y ejecútalo. ¡La torreta responde!



paso3.py

Nuestra siguiente tarea va a ser añadir la aparición de los monos, nuestros enemigos en el juego, y su movimiento en busca de nuestra perdición...

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import pilas
import random

pilas.iniciar()

# Variables y Constantes
balas_simples = pilas.actores.Bala
monos = []
tiempo = 6
fin_de_juego = False


# Funciones
def mono_destruido():
    pass


def crear_mono():
    # Crear un enemigo nuevo
    enemigo = pilas.actores.Mono()

    # Hacer que se aparición sea con un efecto bonito
    enemigo.escala = 0
    enemigo.escala = pilas.interpolar(0.5, duracion=0.5, tipo='elastico_final')

    # Dotarle de la habilidad de que explote al ser alcanzado por un disparo
    enemigo.aprender(pilas.habilidades.PuedeExplotar)

    # Situarlo en una posición al azar, no demasiado cerca del jugador
    x = random.randrange(-320, 320)
    y = random.randrange(-240, 240)

    if x >= 0 and x <= 100:
        x = 180
    elif x <= 0 and x >= -100:
        x = -180
```

```
    if y >= 0 and y <= 100:
        y = 180
    elif y <= 0 and y >= -100:
        y = -180

    enemigo.x = x
    enemigo.y = y

    # Dotarlo de un movimiento irregular más impredecible
    tipo_interpolacion = ['lineal',
                          'aceleracion_gradual',
                          'desaceleracion_gradual',
                          'rebote_inicial',
                          'rebote_final']

    enemigo.x = pilas.interpolar(0, tiempo,
                                 tipo=random.choice(tipo_interpolacion))
    enemigo.y = pilas.interpolar(0, tiempo,
                                 tipo=random.choice(tipo_interpolacion))

    # Añadirlo a la lista de enemigos
    monos.append(enemigo)

    # Permitir la creación de enemigos mientras el juego esté en activo
    if fin_de_juego:
        return False
    else:
        return True


# Usar un fondo estándar
pilas.fondos.Pasto()

# Añadir un marcador
puntos = pilas.actores.Puntaje(x=230, y=200, color=pilas.colores.blanco)
puntos.magnitud = 40

# Añadir el conmutador de Sonido
pilas.actores.Sonido()

# Añadir la torreta del jugador
torreta = pilas.actores.Torreta(municion_bala_simple=balas_simples,
                                enemigos=monos,
                                cuando_elimina_enemigo=mono_destruido)

# Crear un enemigo cada segundo
```

```
pilas.mundo.agregar_tarea(1, crear_mono)

# Arrancar el juego
pilas.ejecutar()
```

La manera de conseguir con Pilas que se realice una tarea cada cierto tiempo es usar la función **pilas.mundo.agregar_tarea()**. Esto es lo que hemos hecho en la parte final del código. ¿Qué le hemos pasado como argumentos? En primer lugar, el tiempo en segundos que queremos que pase cada vez que se realice la tarea. Y en segundo lugar, la tarea que queremos que se realice; en nuestro caso le proporcionamos el nombre de una función que debemos definir a su vez, **crear_mono()**. Observa, ya no volveremos a insistir más, entre la diferencia de pasar una función como argumento a pasar el resultado de la ejecución de dicha función, cuando incluimos los paréntesis.

El meollo del asunto está en la función **crear_mono()** como puedes imaginarte. Vamos a mirar su código con detenimiento:

Cada vez que se llame, hay que crear un nuevo mono; eso es lo que hacemos utilizando el módulo **pilas.actores** y almacenándolo en la variable local **enemigo** para su uso. Fíjate que no queremos que simplemente aparezca, si no que lo haga con un efecto vistoso. De eso se encargan las dos líneas siguientes:

```
enemigo.escala = 0
enemigo.escala = pilas.interpolar(0.5, duracion=0.5, tipo='elastico_final')
```

Al poner el atributo **escala** del enemigo creado a 0, conseguimos que inicialmente no se visualice. A continuación, usamos la función **pilas.interpolar()** para que Pilas genere una animación que pase del tamaño 0 antes indicado al **0.5** que indicamos en el primer argumento (es decir, a la mitad del tamaño original de la imagen del enemigo, ya que tal como está predefinido en Pilas, el actor **Mono** es muy grande para el tamaño de la ventana actual). Esta función es muy potente, ya que de forma automática es capaz de generar animaciones muy vistosas y variadas gracias al tercer argumento, el **tipo** de animación. Nuevamente, te sugerimos que abras el intérprete de Pilas y escribas

```
pilas.ver(pilas.interpolar)
```

para observar su código y ver otros detalles y las diferentes opciones a las que nos referimos. La elegida es **'elastico_final'** ya que da un toque de aparición sorpresa muy aparente. El tiempo que dura ese efecto, lo indicamos con el argumento **duracion**, en nuestro caso, medio segundo.

Bien. Lo siguiente que hacemos es dotar al enemigo que hemos creado con un comportamiento específico para cuando lo alcancemos con un disparo. Esto se consigue con el método **aprender** de los actores; en nuestro caso, la habilidad concreta que queremos que aprenda la obtenemos del módulo **pilas.habilidades** (¿tenemos que insistirte en que curiosees su código fuente?). Con la línea

```
enemigo.aprender(pilas.habilidades.PuedeExplotar)
```

conseguimos que los monos que creemos tengan la capacidad de explotar, que incorpora Pilas, cuando se eliminen.

Las siguientes líneas, generan las coordenadas **x** e **y** del enemigo creado para situarlo en una posición aleatoria en la ventana. Para ello hemos tenido que añadir al comienzo del programa la importación del módulo **random** de Python y usar su función **randrange()** que devuelve un número al azar entre los dos dados. Además, para evitar que el enemigo aparezca demasiado cerca de la torreta y haga el juego imposible, si las coordenadas generadas son menores de 100, se le aleja una distancia de **180**. Finalmente, actualizamos la posición del mono modificando **enemigo.x** y **enemigo.y**.

Ya tenemos el enemigo situado. ¿Cómo lo movemos? De la misma manera que creamos una animación del tamaño del enemigo creado con **pilas.interpolar()** aplicado al atributo enemigo.escala, podemos crear la animación de su movimiento si lo aplicamos a los atributos **enemigo.x** y **enemigo.y**. Un detalle que, además, mejorará el aspecto de movimiento 'animal' es el de elegir el tipo de efecto en la animación de forma también al azar. Eso lo podemos lograr creando una lista de tipos de animación

```
tipo_interpolacion = ['lineal',
                      'aceleracion_gradual',
                      'desaceleracion_gradual',
                      'rebote_inicial',
                      'rebote_final']
```

(aquí hemos elegido 5 tipos distintos) y utilizando la función **random.choice()** para tomar uno de ellos al azar. Por ejemplo, con la coordenada x:

```
enemigo.x = pilas.interpolar(0, tiempo,
                             tipo=random.choice(tipo_interpolacion))
```

Si te fijas, para graduar la velocidad con la que el mono se acercará (el tiempo de duración de la animación) hemos usado la variable **tiempo**, definida más arriba en el código, con un valor de **6**. Prueba a poner valores mayores para hacer más fácil el juego y menores para hacerlo difícil...

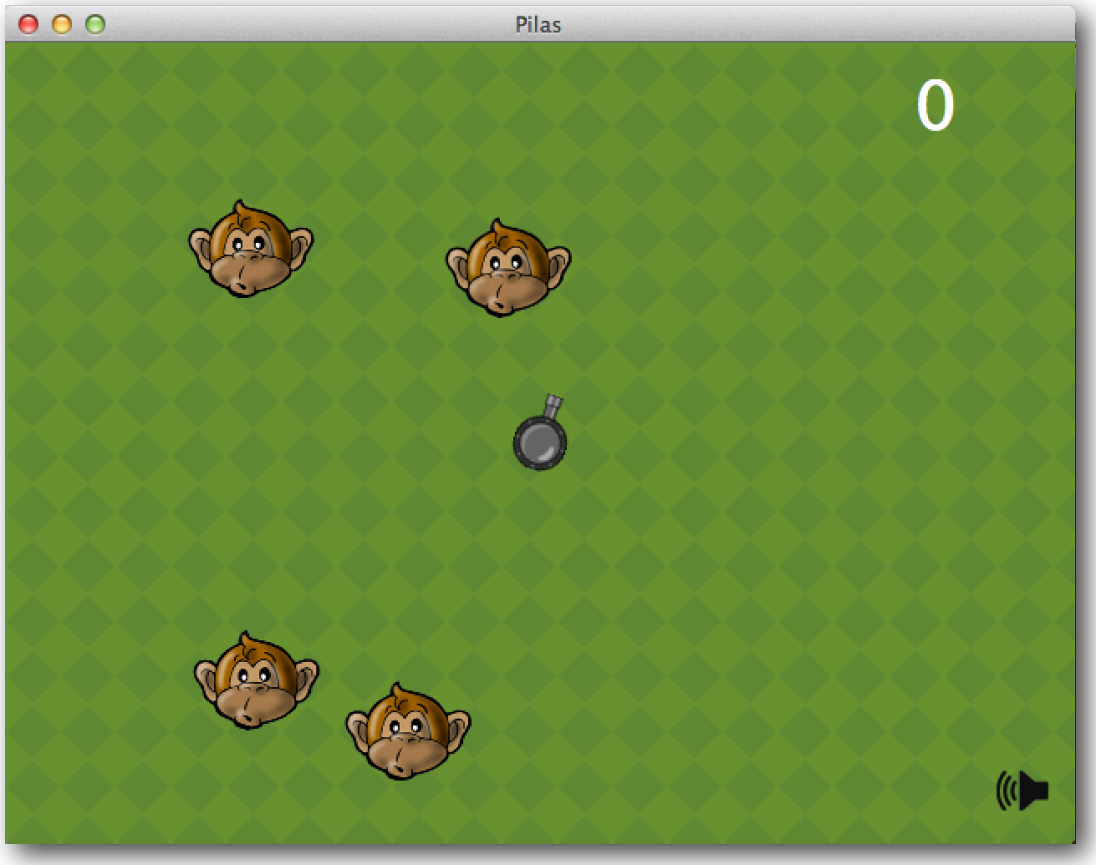
Ya tenemos completado el nuevo enemigo creado; solo falta, para acabar con él, añadirlo a la lista de enemigos (la lista **monos**), de ahí el **monos.append(enemigo)**.

Hay un último detalle que hemos de completar antes de acabar con la función **crear_mono()**. Recuerda que conseguimos que se ejecute cada segundo gracias a que se la hemos pasado al método **pilas.mundo.agregar_tarea()**. Para que, de hecho, la tarea se realice, nos hemos de asegurar que la función devuelve **True** (como es lógico, tiene que existir un mecanismo para cancelar tareas cuando se quiera). Lo que queremos en el fondo es sencillo; mientras dure el juego, se tienen que crear monos (hay que devolver **True**) y cuando éste finalice, no (hay que devolver **False**).

Sin necesidad de concretar más, de momento, definimos una variable booleana que llamaremos **fin_de_juego** y que controlará si el juego ha terminado o no. Inicialmente, por supuesto, el valor será **False**. En consecuencia, las últimas lineas de la función serán

```
if fin_de_juego:
    return False
else:
    return True
```

¡Completado! Guarda el código como **paso3.py** y ejecútalo. ¡Esto ya tiene buena pinta!



paso4.py

Monos atacando, torreta disparando... ¡Algo tiene que pasar cuando choquen! En este paso vamos a dedicarnos precisamente a eso, a implementar la destrucción de los monos y de la torreta cuando éstos lo alcancen.

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import pilas
import random

pilas.iniciar()

# Variables y Constantes
balas_simples = pilas.actores.Bala
monos = []
tiempo = 6
fin_de_juego = False


# Funciones
def mono_destruido(disparo, enemigo):
    # Eliminar el mono alcanzado
    enemigo.eliminar()
    disparo.eliminar()

    # Actualizar el marcador con un efecto bonito
    puntos.escala = 0
    puntos.escala = pilas.interpolar(1, duracion=0.5, tipo='rebote_final')
    puntos.aumentar(1)


def crear_mono():
    # Crear un enemigo nuevo
    enemigo = pilas.actores.Mono()

    # Hacer que se aparición sea con un efecto bonito
    enemigo.escala = 0
    enemigo.escala = pilas.interpolar(0.5, duracion=0.5, tipo='elastico_final')

    # Dotarle de la habilidad de que explote al ser alcanzado por un disparo
    enemigo.aprender(pilas.habilidades.PuedeExplotar)
```

```
    # Situarlo en una posición al azar, no demasiado cerca del jugador
    x = random.randrange(-320, 320)
    y = random.randrange(-240, 240)

    if x >= 0 and x <= 100:
        x = 180
    elif x <= 0 and x >= -100:
        x = -180

    if y >= 0 and y <= 100:
        y = 180
    elif y <= 0 and y >= -100:
        y = -180

    enemigo.x = x
    enemigo.y = y

    # Dotarlo de un movimiento irregular más impredecible
    tipo_interpolacion = ['lineal',
                          'aceleracion_gradual',
                          'desaceleracion_gradual',
                          'rebote_inicial',
                          'rebote_final']

    enemigo.x = pilas.interpolar(0, tiempo,
                                 tipo=random.choice(tipo_interpolacion))
    enemigo.y = pilas.interpolar(0, tiempo,
                                 tipo=random.choice(tipo_interpolacion))

    # Añadirlo a la lista de enemigos
    monos.append(enemigo)

    # Permitir la creación de enemigos mientras el juego esté en activo
    if fin_de_juego:
        return False
    else:
        return True


def perder(torreta, enemigo):
    # Indicar fin de juego y eliminar lo que ya no se necesita
    global fin_de_juego

    enemigo.sonreir()
```

```
    torreta.eliminar()

    pilas.mundo.tareas.eliminar_todas()

    fin_de_juego = True
    pilas.avisar("GAME OVER. Conseguiste %d puntos" % (puntos.obtener()))

# Usar un fondo estándar
pilas.fondos.Pasto()

# Añadir un marcador
puntos = pilas.actores.Puntaje(x=230, y=200, color=pilas.colores.blanco)
puntos.magnitud = 40

# Añadir el conmutador de Sonido
pilas.actores.Sonido()

# Añadir la torreta del jugador
torreta = pilas.actores.Torreta(municion_bala_simple=balas_simples,
                                enemigos=monos,
                                cuando_elimina_enemigo=mono_destruido)

# Crear un enemigo cada segundo
pilas.mundo.agregar_tarea(1, crear_mono)

# Añadir la colisión de enemigos con la torreta para finalizar el juego
pilas.mundo.colisiones.agregar(torreta, monos, perder)

# Arrancar el juego
pilas.ejecutar()
```

En primer lugar, es hora de retomar la función **mono_destruido()** que habíamos dejado sin contenido en **paso2.py**. Para empezar, fíjate que hemos cambiado su definición a

```
def mono_destruido(disparo, enemigo):
```

ya que ésta función la enlazamos en su momento desde la creación de la torreta con el argumento **cuando_elimina_enemigo** (mira el código más arriba). Hecho de este modo, Pilas pasa automáticamente a la función, como argumentos, tanto el **disparo** (una de las balas_simples) como el **enemigo** (uno de los monos).

¿Qué hacemos con ello? Pues lo que es lógico; eliminar el enemigo (**enemigo.eliminar()**, que a su vez será una explosión ya que le hemos dotado de esa habilidad), eliminar de la misma forma el disparo (que desaparecerá discretamente, ya que él no explota) y actualizar la puntuación del marcador.

Esto último es sencillo usando las características de Pilas que empiezan a resultarnos tan familiares. El actor **puntos** (de la clase **Puntaje**, ¿recuerdas?) tiene el método **aumentar()** al que le podemos pasar la cantidad de puntos con la que aumentar el marcador. ¡Es más fácil hacerlo que decirlo! Y de paso, para que sea más evidente el cambio de puntos, aplicamos una interpolación de tamaño... Todo junto:

```
puntos.escala = 0
puntos.escala = pilas.interpolar(1, duracion=0.5, tipo='rebote_final')
puntos.aumentar(1)
```

¡La parte de eliminar a los enemigos ya está conseguida! Vamos a por la torreta...

El jugador pierde cuando no es capaz de eliminar a todos los monos y uno de ellos le alcanza. Así que lo que tenemos que hacer es indicar que cuando choque un mono con la torreta, el juego tiene que realizar las tareas que se encargan de darlo por terminado. Observa cómo lo hemos hecho:

```
pilas.mundo.colisiones.agregar(torreta, monos, perder)
```

Es decir, usamos uno de los módulos predefinidos de Pilas para añadir un tipo de colisión y su respuesta, gracias al método **colisiones.agregar()**. Este método genera el que se lance la función que le pasamos como tercer argumento (**perder()**) cuando colisionen los actores indicados en los otros dos argumentos (**torreta** y **monos**).

Escenas

En realidad, la forma correcta de hacer lo anterior es la siguiente:

```
pilas.escena_actual().colisiones.agregar(torreta, monos, perder)
```

Un juego más elaborado consta de diferentes niveles, pantallas, presentaciones, etc. Ello se orquesta en el diseño de las diferentes **escenas** por nuestra parte, como veremos en otro tutorial posterior.

Usar **escena_actual()** permite separar los comportamientos de una manera más elegante y menos intrusiva y es aconsejable aún cuando tengamos una única escena.

Bien, lo que nos queda, en consecuencia, es definir la función **perder()**. Fíjate que ocurre algo similar a lo de antes; Pilas pasa automáticamente a la función los dos actores que colisionan y así lo reflejamos en el código:

```
def perder(torreta, enemigo):
```

¿Qué es lo que ha de ocurrir cuando somos derrotados? ¿Recuerdas la variable booleana **fin_de_juego**? Acertaste, hemos de darle el valor **True**. (¿entiendes por qué se ha declarado como **global**?). Además, como somos crueles, usamos **enemigo.sonreir()**

para que el mono, que posee esa habilidad predefinida, muestre una sonrisa victoriosa en su cara...

Es el turno de eliminar cosas. Hemos muerto. Así que usamos **torreta.eliminar()**. De paso, eliminamos todas las tareas que se están realizando, pues ya no hacen falta, con la línea **pilas.mundo.tareas.eliminar_todas()** (Ojo: aquí también sería más correcto usar, en lugar de **mundo**, **escena_actual()**. Todo se andará...)

Y queda otro detalle, pues salvo el sonido de la sonrisa del mono, no hay más noticias de que se ha producido la victoria del enemigo. He aquí la solución:

```
pilas.avisar("GAME OVER. Conseguiste %d puntos" % (puntos.obtener()))
```

pilas.avisar() funciona de la misma manera que el aviso automático que vimos con el conmutador de sonido; un breve mensaje no invasivo en la parte inferior de la ventana. Mostramos, de paso, los puntos conseguidos con el método **obtener()** de la clase Puntaje.

Listo. Guarda el código con el nombre **paso4.py** y ejecútalo. ¿Chulo, eh?

paso5.py

Nos quedan los detalles que hacen el juego más visual y algo más variado; mensajes, alicientes de dificultad o ventaja... Veamos el código y lo analizamos después:

```
#! /usr/bin/env python
# -*- encoding: utf-8 -*-

import pilas
import random

pilas.iniciar()

# Variables y Constantes
balas_simples = pilas.actores.Bala
balas_dobles = pilas.municion.BalasDoblesDesviadas
monos = []
tiempo = 6
fin_de_juego = False


# Funciones
def mono_destruido(disparo, enemigo):
    # Eliminar el mono alcanzado
    enemigo.eliminar()
    disparo.eliminar()

    # Actualizar el marcador con un efecto bonito
    puntos.escala = 0
    puntos.escala = pilas.interpolar(1, duracion=0.5, tipo='rebote_final')
    puntos.aumentar(1)


def crear_mono():
    # Crear un enemigo nuevo
    enemigo = pilas.actores.Mono()

    # Hacer que se aparición sea con un efecto bonito
    enemigo.escala = 0
    enemigo.escala = pilas.interpolar(0.5, duracion=0.5, tipo='elastico_final')

    # Dotarle de la habilidad de que explote al ser alcanzado por un disparo
    enemigo.aprender(pilas.habilidades.PuedeExplotar)
```

```
# Situarlo en una posición al azar, no demasiado cerca del jugador
x = random.randrange(-320, 320)
y = random.randrange(-240, 240)

if x >= 0 and x <= 100:
    x = 180
elif x <= 0 and x >= -100:
    x = -180

if y >= 0 and y <= 100:
    y = 180
elif y <= 0 and y >= -100:
    y = -180

enemigo.x = x
enemigo.y = y

# Dotarlo de un movimiento irregular más impredecible
tipo_interpolacion = ['lineal',
                      'aceleracion_gradual',
                      'desaceleracion_gradual',
                      'rebote_inicial',
                      'rebote_final']

enemigo.x = pilas.interpolar(0, tiempo,
                      tipo=random.choice(tipo_interpolacion))
enemigo.y = pilas.interpolar(0, tiempo,
                      tipo=random.choice(tipo_interpolacion))

# Añadirlo a la lista de enemigos
monos.append(enemigo)

# Crear estrella de vez en cuando para el bonus de la munición
if random.randrange(0, 20) > 15:
    if issubclass(torreta.habilidades.DispararConClick.municion,
                  balas_simples):

        estrella = pilas.actores.Estrella(x, y)
        estrella.escala = pilas.interpolar(0.5, duracion=0.5,
                                  tipo='elastico_final')

        pilas.mundo.colisiones.agregar(estrella,
                    torreta.habilidades.DispararConClick.proyectiles,
                    asignar_arma_doble)
```

```
        pilas.mundo.agregar_tarea(3, eliminar_estrella, estrella)

    # Permitir la creación de enemigos mientras el juego esté en activo
    if fin_de_juego:
        return False
    else:
        return True


def perder(torreta, enemigo):
    # Indicar fin de juego y eliminar lo que ya no se necesita
    global fin_de_juego

    enemigo.sonreir()
    torreta.eliminar()

    pilas.mundo.tareas.eliminar_todas()

    fin_de_juego = True
    pilas.avisar("GAME OVER. Conseguiste %d puntos" % (puntos.obtener()))


def asignar_arma_simple():
    # Asignar la munición sencilla
    torreta.municion = balas_simples


def asignar_arma_doble(estrella, disparo):
    # Asignar la munición doble, eliminar la estrella y avisar
    torreta.municion = balas_dobles
    estrella.eliminar()
    pilas.mundo.agregar_tarea(10, asignar_arma_simple)
    pilas.avisar("ARMA MEJORADA")


def eliminar_estrella(estrella):
    # Eliminar estrella
    estrella.eliminar()


def reducir_tiempo():
    # Disminuir la variable tiempo que acelera el movimiento de los monos
    global tiempo
    tiempo -= 1
    pilas.avisar("HURRY UP!!!")
```

```
    if tiempo < 1:
        tiempo = 0.5

    return True

# Usar un fondo estándar
pilas.fondos.Pasto()

# Añadir un marcador
puntos = pilas.actores.Puntaje(x=230, y=200, color=pilas.colores.blanco)
puntos.magnitud = 40

# Añadir el conmutador de Sonido
pilas.actores.Sonido()

# Añadir la torreta del jugador
torreta = pilas.actores.Torreta(municion_bala_simple=balas_simples,
                                enemigos=monos,
                                cuando_elimina_enemigo=mono_destruido)

# Crear un enemigo cada segundo
pilas.mundo.agregar_tarea(1, crear_mono)

# Aumentar la dificultad cada 20 segundos
pilas.mundo.agregar_tarea(20, reducir_tiempo)

# Añadir la colisión de enemigos con la torreta para finalizar el juego
pilas.mundo.colisiones.agregar(torreta, monos, perder)

# Aviso inicial con las instrucciones de juego
pilas.avisar(u"Mueve el ratón y haz click para destruirlos.")

# Arrancar el juego
pilas.ejecutar()
```

Lo primero un pequeño detalle; cuando empieza el juego no hay ninguna instrucción. Como el mecanismo es tan sencillo, simplemente vamos a añadir un aviso de texto indicando que se usa el ratón. Observa que como se incluyen caracteres no anglosajones (en este caso la tilde de 'rat**ó**n'), ponemos una **u** delante de la cadena de texto para pasarle al motor su versión **unicode** (de otra forma, en algunos sistemas se mostraría en pantalla con caracteres extraños).

Vamos a otro punto. Tradicionalmente, en los juegos, cada vez que pasa un cierto tiempo la dificultad suele aumentar. ¿Cómo conseguimos esto? ¿Recuerdas la variable **tiempo**? Controlaba, tal como la definimos y en cierta medida, lo rápido que se movían los monos. Si bajamos su valor, estos se moverán más deprisa... Humm... Dicho y hecho:

```
pilas.mundo.agregar_tarea(20, reducir_tiempo)
```

Es decir, agregamos la tarea de, cada 20 segundos, lanzar la función **reducir_tiempo()**, que se encargará de lo deseado. Observa su definición:

```
def reducir_tiempo():
    # Disminuir la variable tiempo que acelera el movimiento de los monos
    global tiempo
    tiempo -= 1
    pilas.avisar("HURRY UP!!!")

    if tiempo < 1:
        tiempo = 0.5

    return True
```

(¿Entiendes por qué se declara **tiempo** como **global**?). Tres detalles rápidos en los que deberías caer: Primero, para avisar que se aumenta la dificultad, lanzamos un aviso de texto; segundo, no queremos que la variable **tiempo** llegue a **0**, pues entonces el mono ni se movería (basta con que observes el código de su creación para entenderlo), así que nos aseguramos que, como mínimo, valga **0.5**; y tercero, devolvemos **True** para asegurarnos que la tarea repetitiva no deje de realizarse, tal como hemos visto anteriormente.

Bien. Hemos aumentado la dificultad cada cierto tiempo, lo único que nos queda para tener una idea general de un juego típico es crear un bonus para el jugador que le pueda ayudar. Lo elegido es lo siguiente; cada cierto tiempo aparece una estrella en pantalla que, si es destruida, cambia temporalmente la munición que usa la torreta a unas balas dobles mucho más letales.

Hay diferentes maneras de implementar lo anterior. Por variar, y aprovechando que ya tenemos una función que se encarga de crear cada cierto tiempo un mono, vamos a añadir allí mismo la posibilidad de la creación de la estrella. Veamos:

```
    # Crear estrella de vez en cuando para el bonus de la munición
    if random.randrange(0, 20) > 15:
        if issubclass(torreta.habilidades.DispararConClick.municion,
                      balas_simples):

            estrella = pilas.actores.Estrella(x, y)
            estrella.escala = pilas.interpolar(0.5, duracion=0.5,
                                               tipo='elastico_final')

            pilas.mundo.colisiones.agregar(estrella,
                        torreta.habilidades.DispararConClick.proyectiles,
                        asignar_arma_doble)

            pilas.mundo.agregar_tarea(3, eliminar_estrella, estrella)
```

El código anterior aparece al final de la función **crear_mono()**, como hemos indicado. El argumento del **if** tiene la misión de generar un número aleatorio en el rango del **0** al **20** y,

solo si dicho número es mayor de **15**, se procede a continuar (es decir, la proporción esperada es que se generará una estrella, en promedio, cada 4 monos). Pero hay que caer en un detalle más, y es que no se debe crear una estrella si ya estamos en periodo de bonus, ya que ya hemos sido premiados, y hay que esperar a que la torreta vuelva a disparar normal. ¿Cómo averiguarlo? De nuevo, hay muchas maneras de implementarlo (por ejemplo, definiendo una variable booleana que se encargue de ello), pero vamos a optar por ilustrar el uso de la función de Python **issubclass()**. Esta función toma dos argumentos y devuelve **True** si son **instancias** de la misma **clase**. Por lo tanto

```
issubclass(torreta.habilidades.DispararConClick.municion, balas_simples)
```

devolverá **True** (y en consecuencia, se continuará con la ejecución del contenido del bloque **if**) si la torreta está usando la munición de balas simples. ¿Confundido/a? Recuerda que la documentación de Python y Pilas es tu amiga :-). En cualquier caso, lee con naturalidad la **jerarquía de atributos**; la **torreta** es un objeto que posee una serie de **habilidades**, entre ellas la de **DispararConClick** una determinada **municion** que es la que estamos chequeando. ¡Es el código anterior!

Muy bien. Ya sabemos que estamos en el momento adecuado. Toca crear la estrella (y lo hacemos en la misma posición que el mono, como si éste la dejara)

```
        estrella = pilas.actores.Estrella(x, y)
```

usando otro actor predefinido de Pilas. ¿Qué más? Acertaste; la estrella aparece de una forma más linda usando la misma animación que el mono y de ahí la siguiente línea del código. Y por último, para terminar con la estrella, quedan dos cosas:

- Permitir que la torreta pueda destruirla

```
        pilas.mundo.colisiones.agregar(estrella,
                torreta.habilidades.DispararConClick.proyectiles,
                asignar_arma_doble)
```

- Y eliminarla pasado un tiempo

```
        pilas.mundo.agregar_tarea(3, eliminar_estrella, estrella)
```

¿Entiendes las dos líneas anteriores? Respecto de la primera, cuando la **estrella** colisione con uno de los **torreta.habilidades.DispararConClick.proyectiles** se lanzará la función **asignar_arma_doble()**. Y respecto de la segunda, **cada 3 segundos** se lanzará la función **eliminar_estrella()** a la que se le pasará como argumento el objeto **estrella**.

Observa las dos funciones nuevas que acabamos de aludir. Fíjate en el detalle, sobre todo, de que la función **eliminar_estrella()** (que solo hace eso mismo) **no devuelve el valor True** y que, por tanto, **solo se ejecuta una vez** (recuerda, una vez más, que las tareas están activas mientras devuelvan **True**).

La otra función, **asignar_arma_doble()** es, también, interesante:

```
def asignar_arma_doble(estrella, disparo):
    # Asignar la munición doble, eliminar la estrella y avisar
    torreta.municion = balas_dobles
    estrella.eliminar()
    pilas.mundo.agregar_tarea(10, asignar_arma_simple)
    pilas.avisar("ARMA MEJORADA")
```

Ni que decir tiene que lo primero es cambiar la munición de la torreta (para lo que hemos definido previamente la variable **balas_dobles**). Además hemos de avisar de ello con un mensaje de texto y eliminar la estrella a la que hemos disparado y acertado. Pero, y esto es importante, hemos de darle una temporalidad a la munición extra que acabamos de activa. ¿Cómo? En efecto; ¡agregando una nueva tarea! Y ya deberías comprenderlo bien; a los **10** segundos se ejecuta la función **asignar_arma_simple()** que hace lo propio, devolviendo la torreta a su munición estándar. Como has adivinado, también, en la definición de esta última función, **no devolveremos True** para que, así, se ejecute una única vez.

¡Hecho y terminado! Guarda tu código con el nombre **paso5.py** y disfruta de tu obra...

¿Te animas a mejorarlo? Para empezar puedes hacer las correcciones que hemos nombrado en este tutorial. Y puedes simplificar algunas partes del código. Y usar tus propios diseños. Y crear otros enemigos. Y otros comportamientos. Y...

¡Ah, imaginación! ¡Qué grande eres!