

# **pilas engine**

## **Pilas Engine Documentation**

*Release 0.83*

**Hugo Ruscitti**

March 23, 2015



<b>1</b>	<b>Presentación</b>	<b>3</b>
1.1	¿Por qué pilas? . . . . .	3
1.2	Características . . . . .	3
1.3	Sitio oficial . . . . .	4
<b>2</b>	<b>Instalación</b>	<b>5</b>
2.1	Opción 1 - Instalación fácil, para cada sistema operativo . . . . .	5
2.2	Opción 2 - Avanzada, desde repositorios . . . . .	5
<b>3</b>	<b>Empezando, los primeros pasos con pilas</b>	<b>7</b>
3.1	Iniciando pilas . . . . .	7
3.2	Intérprete de pilas . . . . .	8
3.3	Iniciando la biblioteca . . . . .	8
3.4	Creando personajes . . . . .	9
3.5	Cosas en común para los actores . . . . .	10
3.6	Pidiendo ayuda . . . . .	12
3.7	Eliminando a un actor . . . . .	12
3.8	Conclusión . . . . .	12
<b>4</b>	<b>Grupos</b>	<b>13</b>
4.1	Creando grupos de actores . . . . .	13
4.2	Creando un grupo para distintos actores . . . . .	16
<b>5</b>	<b>Colisiones</b>	<b>19</b>
5.1	Un ejemplo sencillo . . . . .	19
<b>6</b>	<b>Física</b>	<b>23</b>
6.1	El protagonista es Box2D . . . . .	23
6.2	Unos ejemplos . . . . .	23
6.3	Modo depuración de física . . . . .	24
6.4	Física personalizada . . . . .	26
6.5	Escala real y tamaño de figuras . . . . .	29
6.6	Cambiando la gravedad interactivamente . . . . .	30
<b>7</b>	<b>Manejo de imágenes</b>	<b>31</b>
7.1	Grillas de imágenes . . . . .	31
7.2	Reproduciendo animaciones . . . . .	32
7.3	Animaciones controladas a mano con una grilla . . . . .	32
7.4	Haciendo actores con animación . . . . .	33
<b>8</b>	<b>Cargar sonidos y música</b>	<b>35</b>
8.1	Reproducir . . . . .	35

<b>9</b>	<b>Dibujado simple en pantalla</b>	<b>37</b>
<b>10</b>	<b>Usando la Tortuga para dibujar</b>	<b>39</b>
10.1	Inspeccionando a la tortuga . . . . .	40
<b>11</b>	<b>Usando una Pizarra</b>	<b>41</b>
11.1	Pintando imágenes . . . . .	42
11.2	Pintando porciones de imágenes . . . . .	42
11.3	Dibujando grillas . . . . .	42
11.4	La pizarra como actor . . . . .	43
<b>12</b>	<b>Dibujado avanzado con Superficies</b>	<b>45</b>
12.1	Dibujando sobre superficies . . . . .	45
12.2	Creación de una superficie . . . . .	46
12.3	Coordenadas de las superficies . . . . .	46
12.4	Métodos para dibujar . . . . .	47
<b>13</b>	<b>Manejo de tiempo con tareas</b>	<b>49</b>
13.1	Tareas . . . . .	49
13.2	Eliminar tareas . . . . .	49
<b>14</b>	<b>Interpolaciones</b>	<b>51</b>
14.1	Girando un actor . . . . .	51
14.2	Escalando un actor . . . . .	52
14.3	Interpolaciones en cadena . . . . .	52
<b>15</b>	<b>Controlando la pantalla</b>	<b>53</b>
15.1	Modo depuración . . . . .	53
15.2	Orden de impresión: atributo z . . . . .	53
15.3	Atributos de posición . . . . .	54
<b>16</b>	<b>Comportamientos</b>	<b>55</b>
16.1	Un ejemplo, ir de un lado a otro . . . . .	55
<b>17</b>	<b>Controles</b>	<b>57</b>
17.1	Investigando al objeto control . . . . .	57
17.2	¿Dónde consultar los controles? . . . . .	57
<b>18</b>	<b>Escenas</b>	<b>59</b>
18.1	Cosas a tener en cuenta . . . . .	59
18.2	La escena Normal . . . . .	59
18.3	Pausando el juego . . . . .	60
18.4	Cambiando el fondo de las escenas . . . . .	60
18.5	Cómo crear nuevas escenas . . . . .	61
18.6	Método sobrescribibles . . . . .	62
<b>19</b>	<b>Como migrar mi juego al nuevo Gestor de Escenas</b>	<b>63</b>
19.1	Iniciar el juego . . . . .	63
19.2	Escenas del juego . . . . .	63
19.3	Cambio de Escena . . . . .	64
19.4	Eventos . . . . .	64
19.5	Fin de la migración . . . . .	65
<b>20</b>	<b>Nuevo Gestor de Escenas</b>	<b>67</b>
20.1	Escena Base . . . . .	67
20.2	Iniciar pilas con una Escena . . . . .	68
20.3	Cambiar entre Escenas . . . . .	68
<b>21</b>	<b>Demos</b>	<b>69</b>
21.1	Piezas . . . . .	69

<b>22</b>	<b>Interfaz de usuario</b>	<b>73</b>
22.1	Propiedades comunes	73
22.2	Deslizador	73
22.3	Selector	74
22.4	Ingreso de texto	75
22.5	Lista de selección	77
<b>23</b>	<b>Como crear menús para tu juegos</b>	<b>79</b>
23.1	Creando funciones de respuesta	80
23.2	Los menús son actores	81
<b>24</b>	<b>Mapas y plataformas</b>	<b>83</b>
24.1	Presentando el actor Mapa	83
24.2	Colisiones con el escenario	85
24.3	Creando mapas con el programa tiled	85
24.4	Creando bloques sólidos con tiled	89
24.5	Un ejemplo completo	89
<b>25</b>	<b>Diálogos</b>	<b>91</b>
25.1	Mensajes de dialogo	91
25.2	Conversaciones	91
25.3	Preguntas	92
<b>26</b>	<b>Manejo de Cámara</b>	<b>93</b>
26.1	Las coordenadas de la cámara	93
26.2	Objetos sensibles a la cámara	93
<b>27</b>	<b>Motores</b>	<b>95</b>
<b>28</b>	<b>Eventos, conexiones y respuestas</b>	<b>97</b>
28.1	¿Que es un Evento?	97
28.2	Conectando la emisión de eventos a funciones	97
28.3	Observando a los eventos para conocerlos mejor	98
28.4	Desconectando señales	98
28.5	Consultado señales conectadas	99
28.6	Creando tus propios eventos	99
28.7	Referencias	99
<b>29</b>	<b>Textos</b>	<b>101</b>
29.1	Crear cadenas de texto	101
29.2	Los textos son actores	102
29.3	Propiedades exclusivas de los textos	102
29.4	Mostrando mensajes en pantalla	102
<b>30</b>	<b>Habilidades</b>	<b>103</b>
30.1	Un ejemplo	103
30.2	Un ejemplo mas: hacer que un actor sea arrastrable por el mouse	103
30.3	Otro ejemplo: un actor que cambia de posición	104
30.4	Mezclar habilidades	104
30.5	Otras habilidades para investigar	104
30.6	¿Cómo funcionan las habilidades?	104
30.7	¿Ideas?	105
<b>31</b>	<b>Depurando y buscando detalles</b>	<b>107</b>
31.1	Modo pausa y manejo de tiempo	107
31.2	Modos depuración	107
31.3	Activar modos desde código	107
31.4	Activando los modos para detectar errores	107

<b>32 Integrando Pilas a una Aplicación Qt</b>	<b>109</b>
32.1 Código	110
32.2 Resultado	113
<b>33 Referencia completa</b>	<b>115</b>
<b>34 Módulo pilas.habilidades</b>	<b>117</b>
<b>35 Guía para desarrolladores</b>	<b>119</b>
35.1 Repositorio	119
35.2 Obteniendo la última versión del repositorio	119
35.3 Primer prueba	119
35.4 Instalación en modo desarrollo	119
35.5 Mantenerse actualizado, siempre...	120
35.6 Mas detalles	120
35.7 Referencias para desarrolladores	120
<b>36 Guía de preguntas avanzadas</b>	<b>121</b>
36.1 Obtengo errores en ingles al iniciar pilas ¿Que anda mal?	121
36.2 ¿Que es OpenGL?, ¿Cómo se configura en mi equipo?	121
36.3 Obtengo errores de AttributeError por parte de pilas	121
36.4 ¿Cómo puedo personalizar el dibujado de un actor?	122
36.5 ¿A veces los sonidos no se reproducen?	122
36.6 Como desinstalo una versión vieja de pilas	122
36.7 Tengo una consulta puntual, ¿quien me puede ayudar?	123
<b>37 Cómo funciona pilas por dentro</b>	<b>125</b>
37.1 Filosofía de desarrollo	125
37.2 API en español	125
37.3 Bibliotecas que usa pilas	125
37.4 Objetos y módulos	126
37.5 Modo interactivo	127
37.6 Motores multimedia	128
37.7 Sistema de actores	128
37.8 Modo depuración	129
37.9 Sistema de eventos	130
37.10 Habilidades	132
37.11 Documentación	135

Introducción:



---

## Presentación

---



# pílas engine

Pílas es una herramienta para construir videojuegos de manera sencilla y didáctica. También conocido como “motor” o “biblioteca” de videojuegos.

El objetivo de este documento es presentar las características del motor, y dar un recorrido general por todos los componentes que se pueden utilizar.

Es una buena idea tener este documento como referencia, e ir experimentando con los ejemplos paso a paso y elaborando un juego.

## 1.1 ¿Por qué pílas?

Pílas está focalizado en ofrecer ayuda a los desarrolladores de juegos casuales y novatos que quieran hacer sus primeros videojuegos, ofreciendo una colección importante de actores, escenas prediseñadas y rutinas para facilitar las tareas más comunes del desarrollo.

Pensamos que pílas es una gran oportunidad de acercar el desarrollo de videojuegos a todas las personas, principalmente jóvenes con interés en aprender a programar computadoras y darle vida a sus ideas.

Pílas está profundamente inspirada en las ideas de Seymour Papert, logo y bibliotecas de videojuegos modernas como `cocos2d`, `pygame`, `rabbyt` y `squeak`.

## 1.2 Características

Estas son algunas de las características técnicas que podemos resumir de pílas.

- Es multiplataforma (Funciona sobre GNU/Linux y Windows)
- Cuenta con objetos prediseñados para agilizar el desarrollo.
- Tiene documentación completamente en español.

- Se utiliza desde python, lo que permite usarla desde sesiones interactivas.
- Incluye interacción con el motor de física pybox2d.
- Es software libre.
- Cuenta con varios ejemplos y tutoriales para ir paso a paso.

A lo largo del documento se dará un repaso mas completo de cada una de estas características y mas.

## **1.3 Sitio oficial**

Para mas información sobre el motor puedes visitar el sitio web:

<http://www.pilas-engine.com.ar>

---

## Instalación

---

Existen 2 formas de instalar la biblioteca, así que veremos cada una por separado.

### 2.1 Opción 1 - Instalación fácil, para cada sistema operativo

La forma mas sencilla de instalar pilas es descargando alguna de las versiones empaquetadas desde nuestro sitio web:

<http://www.pilas-engine.com.ar/descargas.html>

### 2.2 Opción 2 - Avanzada, desde repositorios

Otra opción es descargar pilas e instalarlo directamente desde nuestro repositorio git.

Hay mas información sobre esta forma de instalación en nuestra sección *Guía para desarrolladores*.

Recorrido básico:



---

## Empezando, los primeros pasos con pilas

---

Si ya tienes instalada la biblioteca podemos comenzar a realizar nuestros primeros pasos con la biblioteca.

**pilas** incluye un intérprete interactivo que te permite escribir código de python y autocompletar el nombre de las funciones. Aunque si quieres, puedes usar el intérprete estándar de python, abriendo un terminal de texto y ejecutando el comando `python`.

### 3.1 Iniciando pilas

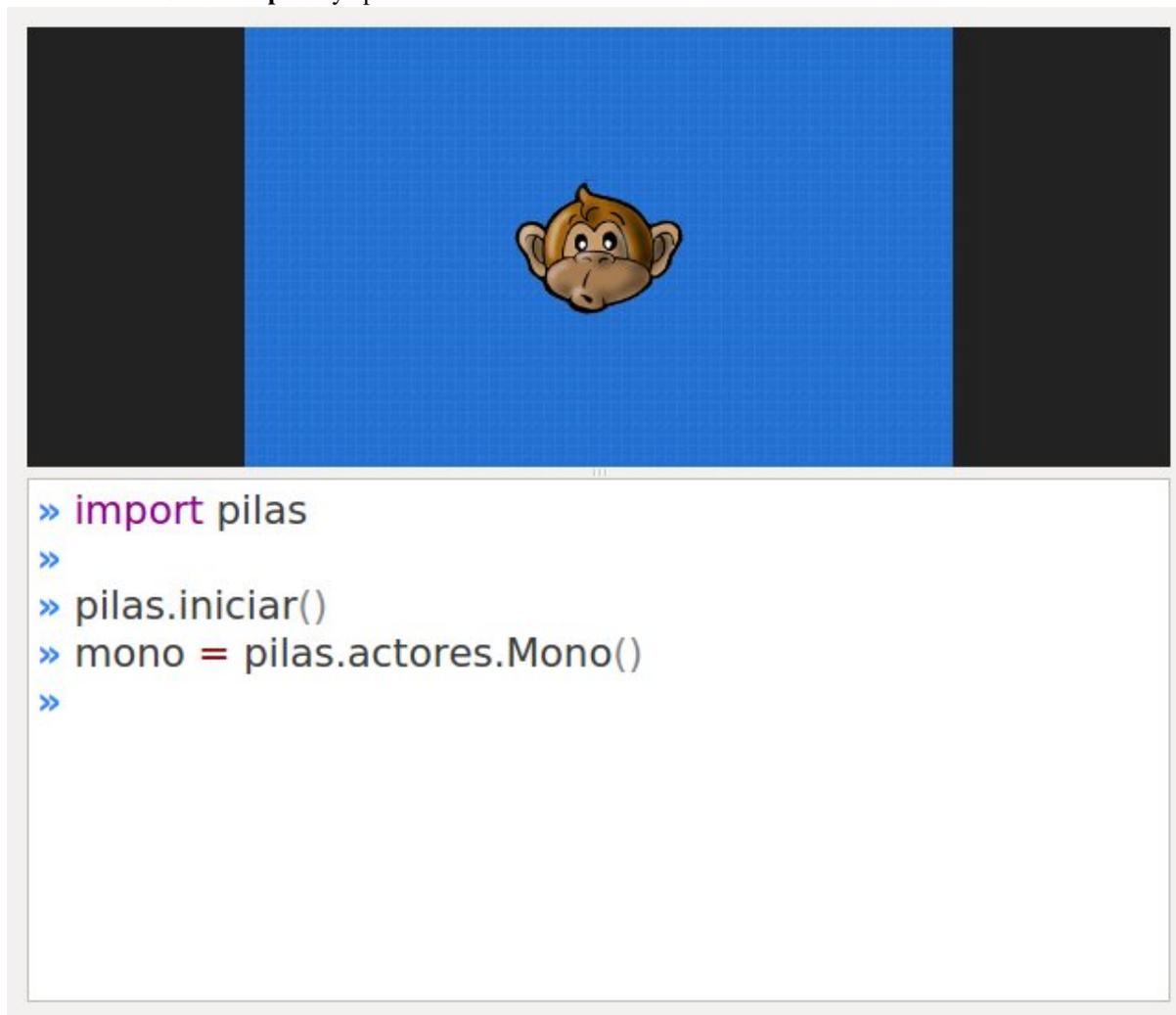
Para ejecutar el asistente de **pilas**, abre un terminal y ejecuta el comando `pilas`.

En tu pantalla tiene que aparecer una ventana como esta.



## 3.2 Intérprete de pilas

Haz click en **Abrir intérprete** y aparecerá esta ventana.



La ventana se divide en dos partes. La parte de abajo muestra el editor. Todo lo que escribas en el editor se ejecutará automáticamente. El resultado se muestra en la parte de arriba.

## 3.3 Iniciando la biblioteca

La parte de arriba es la que utilizaremos para interactuar con el motor. Y mas adelante será la única pantalla que verán los usuarios de nuestros juegos.

Puedes ver que el intérprete viene con algunas líneas de ejemplo.

```
import pilas
```

```
pilas.iniciar()
```

```
mono = pilas.actores.Mono()
```

La línea `import pilas` le dice a Python que use la librería `pilas`. La función `pilas.iniciar()` prepara la ventana (entre otras cosas) para empezar a usar `pilas`. La línea `mono = pilas.actores.Mono()`, por una parte crea un personaje con forma de mono (`pilas.actores.Mono()`) y, por otra parte, le da el nombre de **mono**.

La función `pilas.iniciar()` tiene mas parámetros, pero los veremos mas adelante. Por ahora, continuaremos con lo básico.

## 3.4 Creando personajes

Un concepto importante en `pilas` es el de actores. Un actor en `pilas` es un objeto que aparece en pantalla, tiene una posición determinada y se puede manipular.

Por ejemplo, una nave, un enemigo, una medalla... etc.

Para agilizar el desarrollo de juegos se incluyen varios actores dentro del motor, uno de ellos es `Mono`, un simpático chimpancé.

---

**Note:** Puedes ver todos los actores disponibles en `pilas` ejecutando `pilas.actores.listar_actores()`.

---

Tenemos al actor `mono`. Para indicarle acciones solo tenemos que utilizar su nombre y sentencias simples.

Por ejemplo, para que el personaje cambie su expresión, podemos usar sentencias como:

```
mono.sonreir()
```

o:

```
mono.gritar()
```

En cualquiera de los dos casos, el personaje cambiará su aspecto y emitirá un sonido.



Incluso podríamos decirle al personaje que tiene que hablar algo, por ejemplo:

```
mono.decir("Hey, bienvenido a pilas!!!")
```

y el personaje mostrará exactamente lo que le indicamos que tiene que decir, cómo si fuera un personaje de comic:



A partir de ahora, comenzaremos a escribir algunos ejemplos. Ten en cuenta que no es necesario que escribas todo el código de los ejemplos, el intérprete de `pilas` te permite autocompletar sentencias mientras escribes.

## 3.5 Cosas en común para los actores

Internamente, `Mono` es un actor, así que encontraremos mucha funcionalidad en él que la tendrán el resto de los actores.

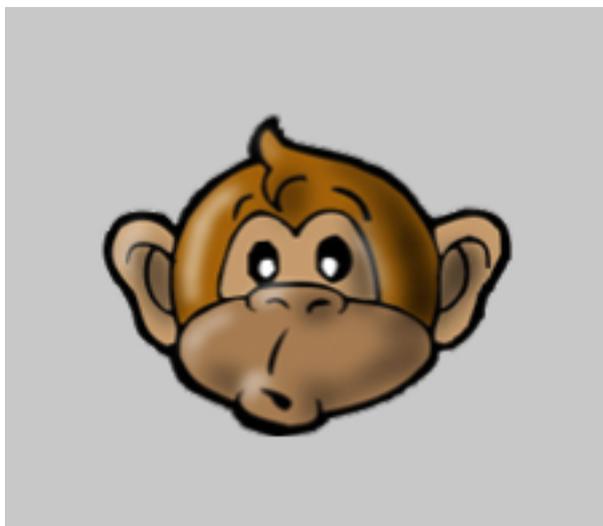
Veamos algunas de estas características:

### 3.5.1 Posición

Podemos cambiar la posición de un actor mediante las propiedades `x` e `y`:

```
mono.x = 100  
mono.y = 100
```

Ten en cuenta que `x` e `y` son las coordenadas de posición en el escenario. Por lo tanto el punto `(0, 0)` es el centro de la ventana. Y `x` aumenta hacia la derecha e `y` hacia arriba.



Este espacio de referencia se puede observar en detalle si pulsas la tecla **F12**, ahí observarás que el movimiento del mouse está asociado a una coordenada y cada actor también.

---

**Note:** Para que tenga efecto, debes tener seleccionada la ventana de resultado (parte de arriba). Haz click sobre ella antes de pulsar F12.

---

### 3.5.2 Escala

Todo actor tiene un atributo para indicar su tamaño en pantalla, el atributo `escala` (que originalmente vale 1):

```
mono.escala = 2
```

### 3.5.3 Rotación

También contamos con un atributo que indica la rotación en grados que debe tener el actor en pantalla. El atributo `rotacion`:

```
mono.rotacion = 40
```



o bien:

```
mono.rotacion = 80
```



La rotación siempre se indica en grados, e indica el grado de inclinación hacia la derecha.

### 3.5.4 Animaciones

Cualquiera de todas las propiedades que vimos anteriormente se pueden usar para hacer animaciones, por ejemplo, con el siguiente código podríamos indicarle al actor que dé una vuelta completa:

```
mono.rotacion = [360]
```

¿por qué?, porque los caracteres [ y ] representan una lista de valores, y cuando pilas ve esta lista asignada a un atributo de pilas, intenta hacer una animación.

Veamos otro ejemplo, si queremos que un personaje como el mono se mueva horizontalmente con una animación podemos escribir esto:

```
mono.x = [-200, 200]
```

Estas animaciones las veremos mas adelante, pero de todas formas es interesante observar que son listas comunes y corrientes.

Python permite multiplicar listas, así que podríamos multiplicarlas para repetir la animación.

```
# Sólo en python.
mono.x = [-200, 200] * 5 # ir de una lado a otro 5 veces.
```

o incluso podríamos alterarles la velocidad, digamos que el ir y regresar se tiene que hacer muy lento, en unos 10 segundos:

```
mono.x = [-200, 200], 10
```

## 3.6 Pidiendo ayuda

Recuerda que cada componente de `pilas` está documentado como un módulo de python. Por lo tanto, puedes ejecutar una sentencia cómo:

```
help(mono)
```

y aparecerán en pantalla todos los instructivos de la funcionalidad del actor.

Incluso puedes usar la función `pilas.ver` para conocer el código fuente de cualquier cosa de `pilas`. Por ejemplo podemos el código completo del `mono` ejecutando la sentencia:

```
pilas.ver(mono)
```

## 3.7 Eliminando a un actor

Para eliminar un actor de la escena tienes que llamar al método `eliminar`:

```
mono.eliminar()
```

## 3.8 Conclusión

Hemos visto los pasos principales para gestionar actores. Ten en cuenta que el módulo `pilas.actores` es donde se guardarán todos los actores.

Un buen ejercicio es ponerse a investigar el módulo **actores**. Hay muchos actores, estos son algunos ejemplos de código para probar:

```
b = pilas.actores.Bomba()
b.explotar()
```

```
p = pilas.actores.Pingu()    # se mueve con el teclado.
```



Es todo por ahora, seguí leyendo o comenzá a explorar por tu cuenta. A divertirse!

---

## Grupos

---

Ahora que podemos manejar a los actores de manera individual. Vamos a ver cómo organizarlos en grupos.

Organizar a los actores en grupo es de utilidad, porque generalmente es una buena idea agrupar a los actores por características y tratarlos a todos por igual. Por ejemplo, en un juego de naves podríamos tener un grupo de naves, un grupo de estrellas y un grupo de disparos.

### 4.1 Creando grupos de actores

Para crear varios actores de una misma clase podríamos ejecutar algo como lo que sigue:

```
bombas = pilas.actores.Bomba() * 5
```

es decir, creamos un actor y luego lo multiplicamos para construir un grupo con muchos actores de la misma especie.

Al crear un grupo de esta forma, todos los actores se colocarán en posiciones aleatorias.

Esto es lo que veríamos en la ventana de pilas:



A partir de ahora, la referencia `bombas` nos servirá para controlar a todas las bombas al mismo tiempo.

Esta referencia es parecida a una lista de python normal. Así que podríamos contar cuantas bombas hay en la escena, o recorrer el grupo haciendo algo:

```
>>> print "hay", len(bombas), "bombas"
hay 5 bombas
```

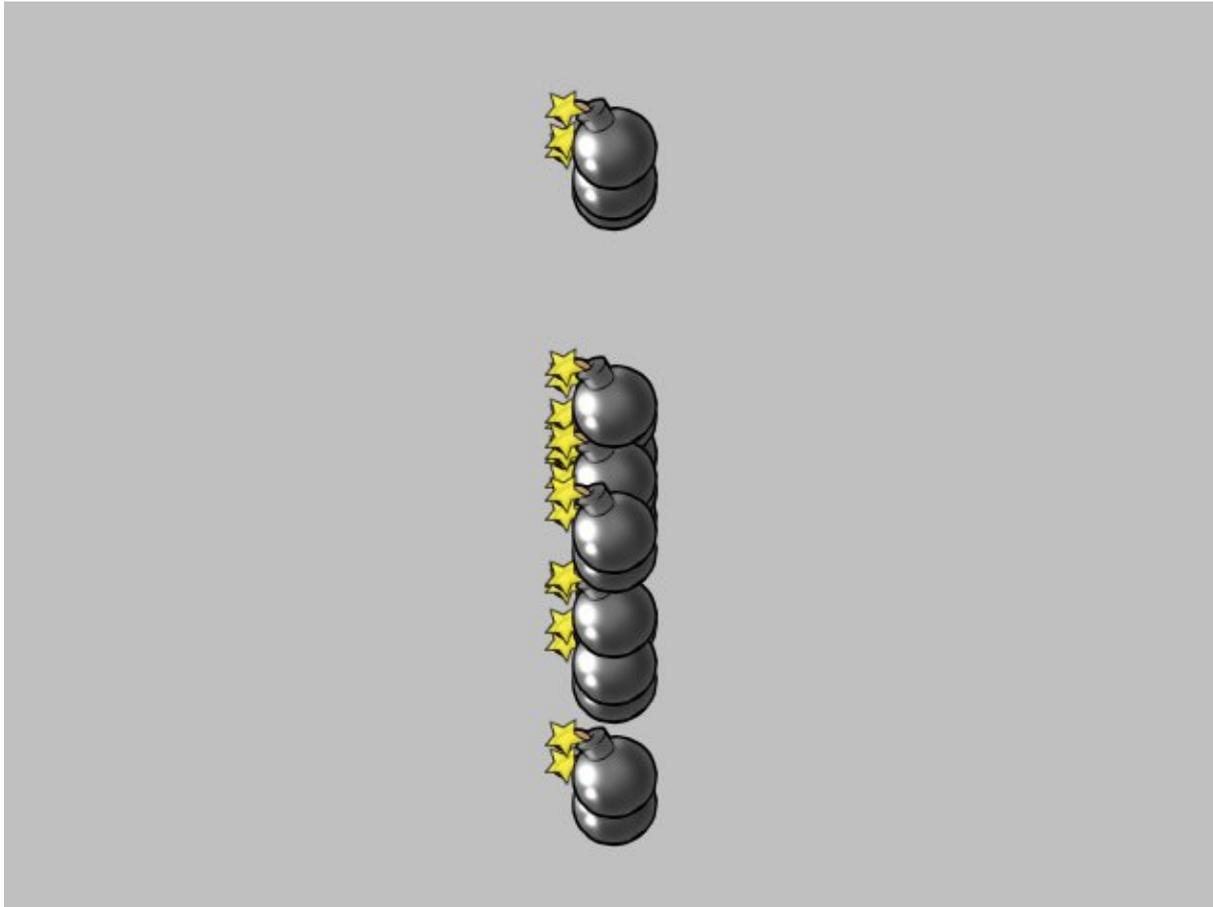
```
>>> for una_bomba in bombas:
...     print una_bomba.x, una_bomba.y
```

Ahora bien, algo que hace un poquito diferente a los grupos de las listas de python, es que los grupos te permiten alterar a varios actores al mismo tiempo con mas facilidad.

Por ejemplo, imagina que quieres hacer que todas las bombas aparezcan en el centro de la ventana. Podrías hacer algo cómo esto:

```
bombas.x = 0
```

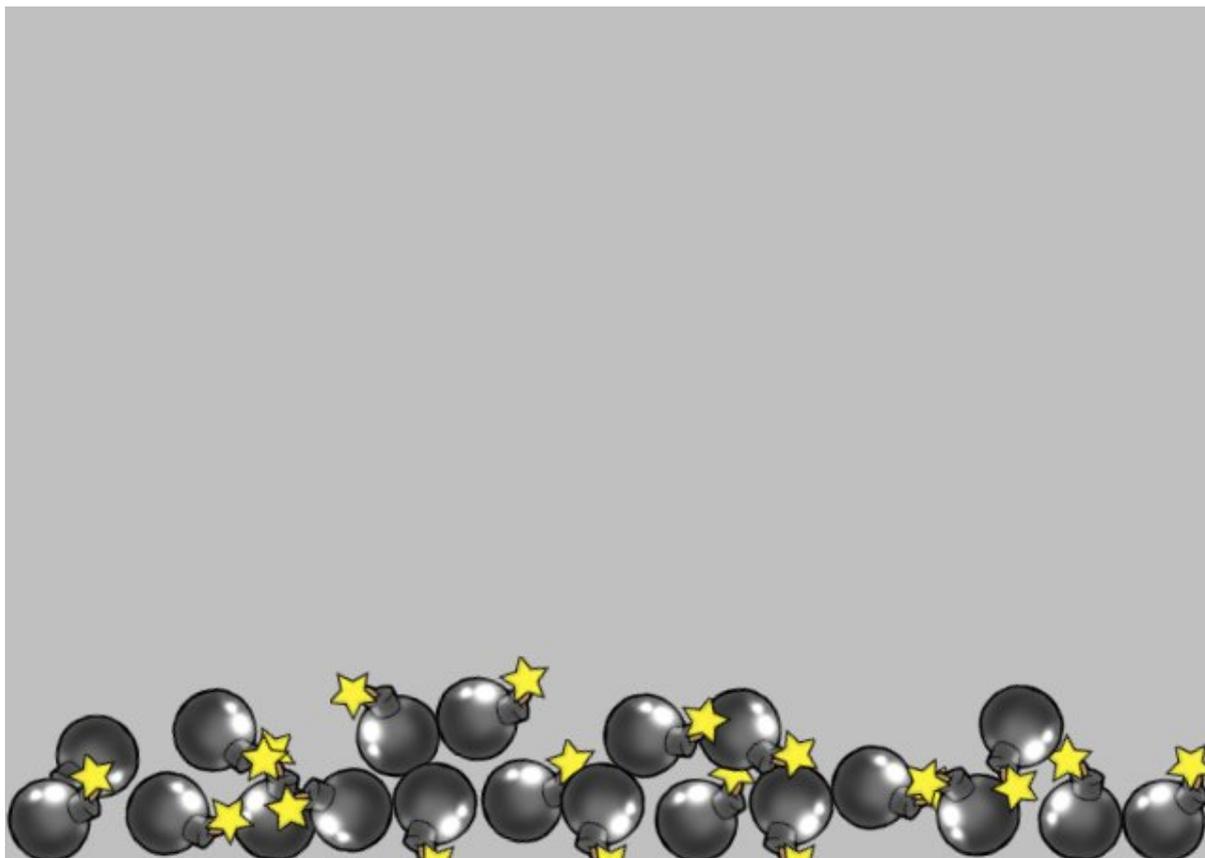
Y en la ventana obtendremos:



Incluso, les podríamos enseñar a las bombas a reaccionar como si fueran pelotas, es decir, que reboten e interactúen con la aceleración gravitatoria:

```
bombas.aprender (pilas.habilidades.RebotarComoPelota)
```

Ahora tendrás algo mucho mas interesante, un montón de actores rebotando entre sí:



Un consejo, la gravedad del escenario se puede modificar usando una sentencia como la que sigue:

```
pilas.atajos.definir_gravedad(200, 0)
```

donde el primer argumento es la gravedad horizontal, en este caso 200 es hacia la derecha, y el segundo argumento es la gravedad vertical, que suele ser de -90 en general.

Pruebalo, es divertido!

## 4.2 Creando un grupo para distintos actores

Hay ocasiones, donde quieres tener un grupo desde cero e ir agregando actores en él.

Esto se puede hacer fácilmente, e incluso abre las puertas a que puedas mezclar actores de distintas especies.

Para crear un grupo vacío tienes que crear un objeto de la clase Grupo:

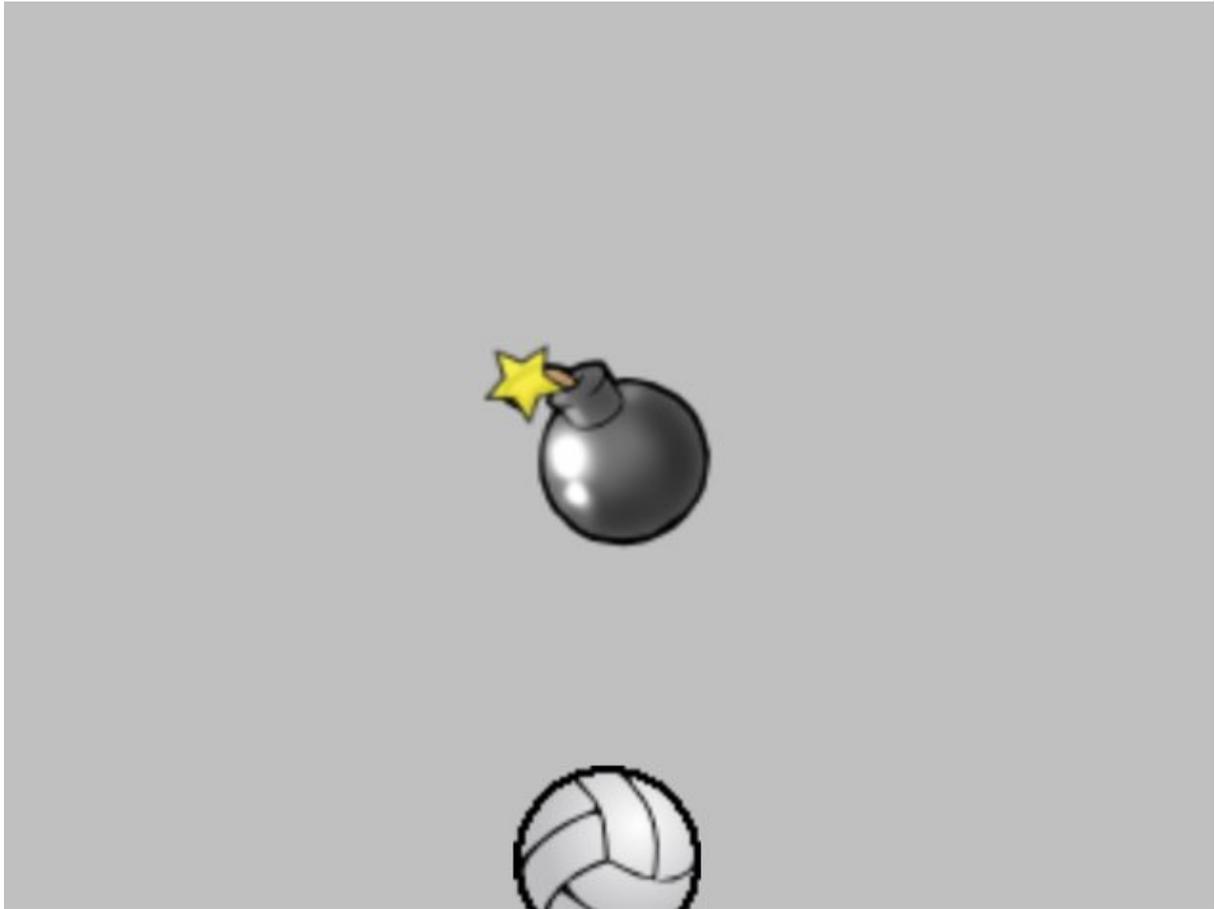
```
mi_grupo = pilas.grupo.Grupo()
```

y luego, para añadir actores al grupo puedes usar el método `append` e indicar la referencia del actor que quieres agregar:

```
bomba = pilas.actores.Bomba()  
pelota = pilas.actores.Pelota()
```

```
mi_grupo.append(bomba)  
mi_grupo.append(pelota)
```

```
mi_grupo.escala = [2]
```





---

## Colisiones

---

En el desarrollo de videojuegos le damos el nombre de `colisión` a lo que ocurre cuando dos actores entran en contacto.

Por ejemplo, cuando un personaje como `Pacman` toca a un `Fantasma` se produce una colisión.

Para programar colisiones en pilas tienes seguir unos pocos pasos.

- Tienes que pensar “qué” quieres hacer cuando se produce una colisión.
- Escribir una función de respuesta a la colisión.
- y, por último, decirle a pilas qué actores son colisionables entre sí.

Ten en cuenta que cada actor tiene un atributo llamado `radio_de_colision`, que se representa como un círculo de color verde cuando pulsas la tecla `F12` sobre la ventana.

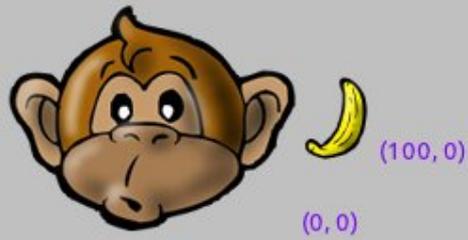
### 5.1 Un ejemplo sencillo

Comencemos con un ejemplo, coloque dos actores en la pantalla de su juego:

```
banana = pilas.actores.Banana()  
banana.x = 100  
mono = pilas.actores.Mono()
```

Ahora pulsá la tecla **F12** para ver la posición de cada uno de los actores:

F12 ModoPosicion habilitado.

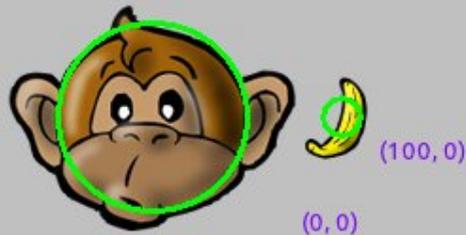


Cuadros por segundo: 60

Posición del mouse: x=-322 y=158

Si pulsas **F9**, aparecerá un círculo verde por cada actor. Ese círculo indica el radio de colisión de cada actor:

F9 ModoRadiosDeColision habilitado.  
F12 ModoPosicion habilitado.



Cuadros por segundo: 60

Posición del mouse: x=-334 y=78

Este círculo se puede alterar cambiando el valor del `radio_de_colision` del actor:

```
banana.radio_de_colision = 30
```



`radio_de_colision = 15`



`radio_de_colision = 30`

Ahora, para poder mover al mono, podemos enseñarle una habilidad:

```
mono.aprender(pilas.habilidades.Arrastrable)
```

Ahora vamos a crear una función con lo que queremos que hagan los dos actores al entrar en contacto:

```
def el_mono_comer(mono, banana):
    mono.sonreir()
    banana.eliminar()
```

y por último crear dos listas de actores y decirle a pilas que asocie la función de mas arriba con la colisión:

```
bananas = [banana]
pilas.escena_actual().colisiones.agregar(mono, bananas, el_mono_comer)
```

Perfecto. Ahora, si mueves al mono por la pantalla con el mouse, podrá comer bananas.

Intenta crear mas actores que representen bananas y agregarlos a la lista que usamos antes, por ejemplo:

```
b = pilas.actores.Banana()
b.x = -100
```

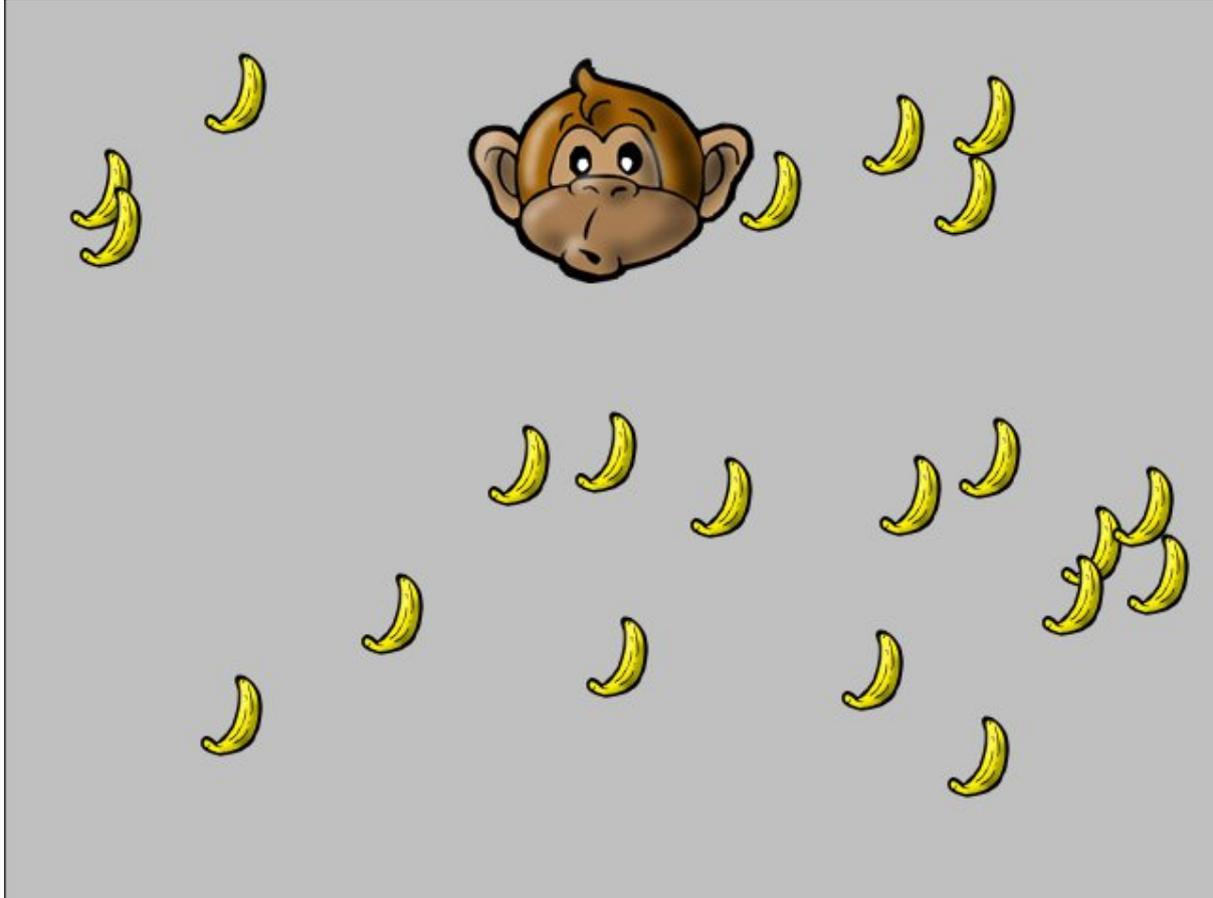
```
bananas.append(b)
```

Ahora intenta nuevamente mover al mono... sí, las colisiones seguirán funcionando, cualquier objeto que agregues a la lista de bananas será alimento del mono...

Bien, ¿y si queremos llenar la pantalla de comida?, una forma sencilla es ejecutar lo siguiente:

```
bananas += pilas.atajos.fabricar(pilas.actores.Banana, 20)
```

La pantalla se llenará de bananas, bah, habrá exactamente 20...



De hecho, si quieres tener mas control sobre las bananas que se crearán, puedes usar esta rutina de código en lugar de llamar a la función `fabricar`:

```
import random
cantidad_de_bananas = 20

for i in range(cantidad_de_bananas):
    banana = pilas.actores.Banana()
    banana.x = random.randrange(-200, +200)
    banana.y = random.randrange(-200, +200)
    bananas.append(banana)
```

---

## Física

---

Pilas incluye integración con un sistema de física para realizar simulaciones y dotar a tus juegos de mas realismo y diversión.

### 6.1 El protagonista es Box2D

El motor de física seleccionado para pilas se llama Box2D, el mismo motor de física utilizado en el juego Angry Birds.

Así, Box2D y PyBox2D son las bibliotecas protagonistas de casi toda la funcionalidad que vas a ver en este módulo.

El módulo `pilas.fisica` es solamente un facilitador para utilizar Box2D, y que puedas comenzar a jugar con físicas rápidamente.

Así que aprovecho este espacio para dar las gracias a **Erin Catto**, y su grupo de desarrollo por haber creado **Box2D**.

### 6.2 Unos ejemplos

El motor de física se puede mostrar en funcionamiento usando un ejemplo, escribe el siguiente código:

```
pelotas = pilas.actores.Pelota() * 10
```

esto creará un grupo de circunferencias que rebotarán hasta la parte inferior de la pantalla.

De manera similar puedes crear un montón de cajas y hacerlas rebotar:

```
cajas = pilas.actores.Caja() * 10
```

Como puedes ver, el resultado es un grupo caótico de actores chocando entre sí. Mas adelante veremos como personalizar y “controlar” un poco el escenario.



Los actores que tienen física son un poco particulares, pero aún así se los puede tratar como a otros actores. Por ejemplo, si quieres poder arrastrar y soltar figuras con el mouse, puedes enseñarles una habilidad:

```

pelotas.aprender(pilas.habilidades.Arrastrable)
cajas.aprender(pilas.habilidades.Arrastrable)

```

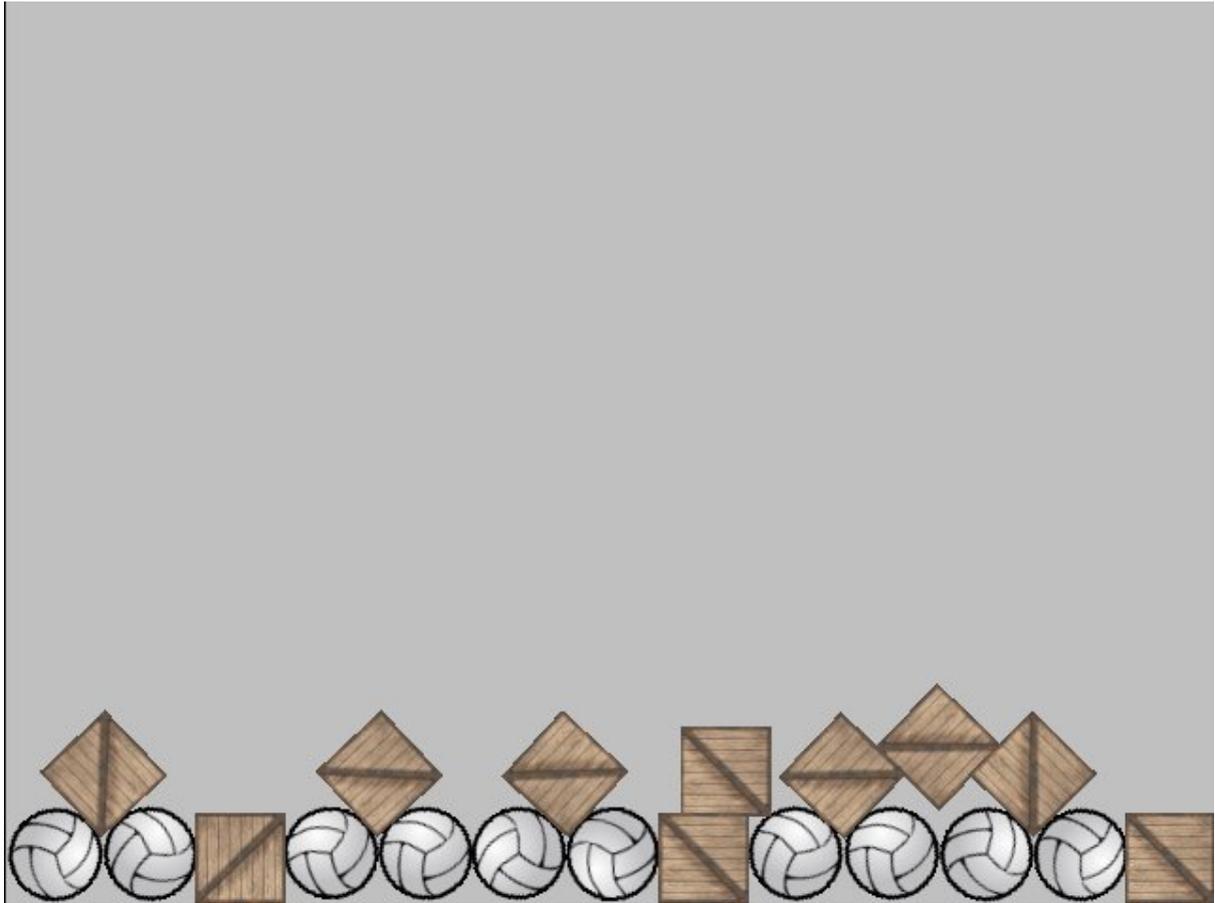
### 6.3 Modo depuración de física

Cuando haces juegos con física o movimientos realistas es muy importante tener en cuenta un concepto importante:

Si bien uno observa pelotas y cajas, en realidad, internamente son solo cuadrados y circunferencias.

Lo que ocurre en pantalla son dos cosas, por un lado vemos imágenes con aspecto de caja o pelota, y por el otro se nos oculta una simulación entre polígonos mucho más primitiva y simple.

Observa esta escena:



Cada uno de esos actores está asociado a una figura geométrica, la física en realidad se da en un nivel muy primitivo de figuras. El aspecto de las cosas es solo eso, un aspecto. Lo que “manda” en el comportamiento físico son las figuras geométricas (cuerpos).

Intenta lo siguiente, pulsa la tecla **F11** y observarás varias líneas de color rojo indicando las figuras de los cuerpos:



F11 ModoFisica habilitado.

Cuadros por segundo: 60

Posición del mouse: x=-234 y=242

Ahora genera dos figuras físicas, una circunferencia estática y otra dinámica:

```
circulo = pilas.fisica.Circulo(0, 0, 50, dinamica=False)
circulo_dinamico = pilas.fisica.Circulo(10, 200, 50)
```

El primer círculo aparecerá en el centro de la ventana, y el segundo comenzará en la posición (10, 200), es decir, en la parte superior de la ventana y luego caerá rebotando. Algo así:



Ahora bien, habrás notado que estas dos circunferencias las podemos ver porque está habilitado el módulo de depuración (que activamos con **F11**), pero esto no lo va a ver alguien que juegue a nuestro juego. El modo depuración es solo para desarrolladores.

Lo que nos falta hacer, es darles apariencia a esas figuras. Algo así como una piel..

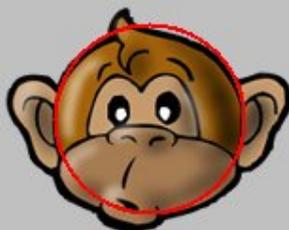
Para esto podemos usar actores. La dinámica es así, tenemos que crear dos actores, y luego decirle a estos actores que se comporten cómo figuras geométricas.

Agreguemos a nuestro programa estas 4 líneas de código, queremos que el primer círculo (el del centro) sea un mono, y el otro círculo que sea una bomba:

```
mono = pilas.actores.Mono()  
mono.aprender(pilas.habilidades.Imitar(circulo))  
  
bomba = pilas.actores.Bomba()  
bomba.aprender(pilas.habilidades.Imitar, circulo_dinamico)
```

Esto es diferente a lo anterior, los objetos físicos ahora tienen apariencia:

F11 ModoFísica habilitado.



Cuadros por segundo: 60

Posición del mouse: x=-232 y=241

Ahora podríamos desactivar el modo depuración física (pulsando nuevamente **F11**) y jugar un poco impulsando la bomba de un lado a otro:

```
circulo_dinamico.y = 200
```

Ten en cuenta que ahora la figura del motor físico es la que determina el movimiento y la rotación, así que ya no funcionará escribir algo como `bomba.y = 200`, ahora tienes que escribir `circulo_dinamico.y = 200` para mover al actor...

Otra cosa a considerar, es que en nuestro ejemplo no ajustamos muy bien el tamaño del `circulo_dinamico` con el de la bomba. Esto es un detalle poco relevante aquí, porque solo quiero explicar cómo se usa el motor, pero cuando hagas tus juegos, recuerda usar el modo depuración de física para detectar estos detalles y corregirlos, son muy importantes para que tus usuarios disfruten del juego. Recuerda que ellos no verán los círculos rojos... solo verán la apariencia de los actores.

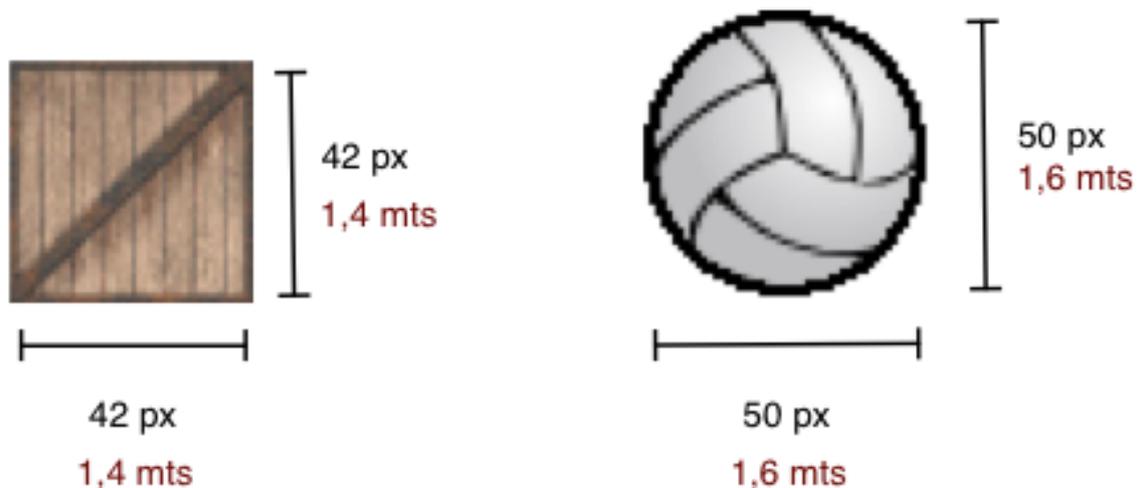
## 6.5 Escala real y tamaño de figuras

Pilas utiliza una pantalla que se puede medir en pixels, de hecho, todas las imágenes tienen un tamaño en pixels predefinido.

Pero dentro del mundo físico de `box2d`, las figuras no tienen tamaño en pixels sino en metros.

¿Cuál es la relación?, pilas convierte pixels a metros para mantener al mundo de `box2D` en coherencia con lo que estamos viendo en pantalla.

30 pixels son equivalentes a 1 metro:



## 6.6 Cambiando la gravedad interactivamente

Por defecto, la gravedad del escenario es de (0, -90), esto significa que los objetos “caen” hacia abajo, y lo hacen con una aceleración de 90 mts/s<sup>2</sup> (metros sobre segundos cuadrados).

Pero no estás obligado a que esto sea siempre así, de hecho si quieres hacer un juego que transcurra en el espacio seguramente vas a querer eliminar por completo la gravedad del escenario para que los objetos puedan “flotar”, ¿no?

Entonces, hay dos formas de cambiar la gravedad del escenario. Podrías cambiar la gravedad en cualquier momento invocando a la función `definir_gravedad` indicando la nueva gravedad, por ejemplo:

```
pilas.atajos.definir_gravedad(200, 0)
```

o directamente especificar la gravedad cuando inicias pilas, por ejemplo:

```
pilas.fisica.definir_gravedad(90, 90)
```

Ten en cuenta que el primer argumento es la aceleración horizontal y la segunda componente es la aceleración vertical. Los valores originales de la gravedad son 0 y -90.

---

## Manejo de imágenes

---

En los videojuegos 2D las imágenes suelen estar en formatos gráficos como **png** o **jpg** ya diseñados con anterioridad.

En `pilas` se pueden cargar estos recursos usando el módulo `imagenes`. Por ejemplo, si tenemos una imagen llamada `hola.png` podríamos incorporarla a nuestro juego así:

```
import pilas

hola = pilas.imagenes.cargar('hola.png')
```

Las imágenes no se imprimen directamente en pantalla, en su lugar tienes que crear un Actor y asignarle la imagen.

Por ejemplo, el siguiente código muestra la imagen en pantalla:

```
imagen = pilas.imagenes.cargar("mi_personaje.png")
actor = pilas.actores.Actor(imagen)
```

otra opción similar es crear al actor, y luego asignarle la imagen:

```
imagen = pilas.imagenes.cargar("mi_personaje.png")
actor = pilas.actores.Actor()

actor.imagen = imagen
```

Cualquiera de las dos opciones produce el mismo resultado, personaje “cambiará” de apariencia cuando se le asigne una nueva imagen.

### 7.1 Grillas de imágenes

Un forma conveniente de almacenar las imágenes de tus personajes es usar una grilla.

La siguiente imagen es una grilla de 10 columnas que utilizamos para crear al personaje “pingu”:



Internamente la imagen se almacena así, pero a la hora de mostrarse en pantalla se puede seleccionar el cuadro.

Este es un ejemplo que carga la grilla de mas arriba y genera un actor para mostrar el cuadro 1:

```
actor = pilas.actores.Actor()
grilla = pilas.imagenes.cargar_grilla("pingu.png", 10)
actor.imagen = grilla
```

Ten en cuenta que el último argumento de la función `pilas.imagenes.cargar_grilla` es la cantidad de columnas que tiene la grilla. También es posible usar funciones que tengan filas y columnas, solo tendrías que indicar un argumento mas con el número de filas. Lo veremos mas adelante.

Puedes ejecutar la siguiente sentencia para ver la documentación completa de esta función:

```
help(pilas.imagenes.cargar_grilla)
```

## 7.2 Reproduciendo animaciones

Tener una grilla de imagenes es una buena forma de comenzar a realizar animaciones.

Si quieres tomar una grilla y mostrar una y otra vez sus cuadros podrías usar el actor `Animación`.

El siguiente código genera un actor que mostrará uno a uno los cuadros de la grilla:

```
grilla = pilas.imagenes.cargar_grilla("explosion.png", 7)
p = pilas.actores.Animacion(grilla, True)
```

El actor `Animacion`, también puede recibir cómo argumento la velocidad con la que tiene que reproducir la animación (medida en cuadros por segundo).

El segundo argumento indica que la animación tiene que ser cíclica (nunca termina).

Observa este ejemplo, muestra la misma animación de antes pero mostrando un cuadro por segundo y se elimina cuando termina:

```
grilla = pilas.imagenes.cargar_grilla("explosion.png", 7)
p = pilas.actores.Animacion(grilla, False, velocidad=1)
```

## 7.3 Animaciones controladas a mano con una grilla

Otra forma de hacer animaciones, es asociar una grilla directamente a un actor y cambiar el cuadro a mostrar.

Por ejemplo, la siguiente sentencia avanza al siguiente cuadro de animación en la grilla. Recuerda que comienza en 1:

```
grilla.avanzar()
actor.imagen = grilla
```

Ten en cuenta que el método `avanzar` va a retornar `True` o `False`. `True` significa que la grilla ha avanzado y ha mostrado un cuadro nuevo. `False` significa que la grilla volvió a mostrar el primer cuadro.

Este valor de retorno es muy útil a la hora de saber si una animación terminó, y poder tomar alguna decisión al respecto.

### 7.3.1 Grillas con filas y columnas

En el ejemplo anterior mencioné que las grillas pueden tener filas y columnas. Esto se logra gracias a que python permite tener funciones y métodos con argumentos opcionales.

En este caso, la función `cargar_grilla` también puede recibir la cantidad de filas que tiene una grilla:

```
animacion = pilas.imagenes.cargar_grilla("grilla.png", 2, 2)
```

el primer número 2 indica que la grilla tiene dos columnas y el segundo 2 indica que la grilla tiene dos filas.

Cuando usas una grilla con pilas y columnas, la función `avanzar` que vimos antes va a recorriendo los cuadros de la misma manera en que se lee una historieta (de izquierda a derecha y de arriba a abajo).

Esta es la apariencia de la imagen que usamos antes y los números indican el orden con que pilas leerá los cuadros:



### 7.3.2 Haciendo animaciones sencillas

En muchas oportunidades nos interesa hacer animaciones simples y que se repitan todo el tiempo sin mucho esfuerzo.

Con lo que vimos hasta ahora, hacer esas animación es cuestión de cargar una grilla y llamar cada un determinado tiempo a la función `avanzar`.

Pero como esta es una tarea muy común, en **pilas** hay una forma mas sencilla de hacer esto.

Existe un actor llamado `Animación` que tiene la capacidad de mostrar una animación cíclica, es decir, que se repita todo el tiempo, comenzando desde el principio cuando llega al final.

Veamos un ejemplo, esta imagen tiene 6 cuadros de animación ordenados en columnas:



Una forma sencilla de convertir esta animación en un actor simple es crear la grilla, construir un actor `Animacion` e indicarle a pilas que será una animación cíclica, es decir, que se tendrá que repetir indefinidamente:

```
grilla = pilas.imagenes.cargar_grilla("fuego.png", 6)
actor = pilas.actores.Animacion(grilla, ciclica=True)
```

El resultado en la ventana será una animación de fuego que no terminará nunca. Cuando el actor termine de mostrar el cuadro 6 de la animación regresará al primero para comenzar nuevamente.

Otra posibilidad es especificar el argumento `ciclica=False`. En ese caso el actor comenzará a mostrar la animación desde el cuadro 1 y cuando termine eliminará al actor de la ventana. Esto es útil para hacer efectos especiales, como explosiones o destellos, cosas que quieres tener en la ventana un instante de tiempo y nada mas...

## 7.4 Haciendo actores con animación

Puede que quieras hacer un actor que tenga múltiples animaciones, y que las muestre en determinados momentos. Por ejemplo, si tienes una nave con motores, es probable que quieras mostrar una animación de motores en funcionamiento cuando la nave avanza y detener la animación de motores cuando finaliza el movimiento.

Una forma de lograr esto de manera sencilla es crear tu propio actor, y que este tenga dos atributos, uno para cada animación:

```
class MiNave(pilas.actores.Actor):
    def __init__(self, x=0, y=0):
        Actor.__init__(self, x=x, y=y)
        self.animacion_detenida = pilas.imagenes.cargar_grilla("nave_detenida.png", 1)
        self.animacion_movimiento = pilas.imagenes.cargar_grilla("nave_en_movimiento.png", 3)
```

Luego, en el método `actualizar` del propio actor podrías avanzar la animación actual y permitirle al programador invocar métodos para intercambiar animaciones:

```
class MiNave(pilas.actores.Actor):  
  
    # [...] codigo anterior  
  
    def poner_en_movimiento(self):  
        self.imagen = self.animacion_movimiento  
  
    def poner_en_reposo(self):  
        self.imagen = self.animacion_detenida  
  
    def actualizar(self):  
        self.imagen.avanzar()
```

Como puedes ver, el concepto inicial es el mismo, cuando queremos cambiar de animación tenemos que cambiar de grilla, y cuando queremos avanzar la animación solamente tenemos que llamar al método `avanzar`.

---

## Cargar sonidos y música

---

Si bien a continuación ejemplificamos todo con el módulo `pilas.sonidos` todos los ejemplos funcionan para el módulo `pilas.musica`

Los sonidos se pueden cargar usando el módulo `sonidos` de la siguiente manera:

```
sonido_de_explosion = pilas.sonidos.cargar('explosion.wav')
```

donde `explosion.wav` es el nombre del archivo de audio.

Ten en cuenta que esta función para cargar sonidos se comporta muy parecido a la función que nos permite cargar imágenes o grillas. El archivo se buscará en el directorio principal de nuestro juego, luego en el directorio `data` y por último en la biblioteca de sonidos que trae `pilas`.

### 8.1 Reproducir

La función `sonidos.cargar` nos retorna un objeto de tipo `Sonido` que tiene un método para reproducirse llamado `reproducir()`.

Entonces, para reproducir un sonido solamente tienes que llamar al método `reproducir`:

```
sonido_de_explosion.reproducir()
```

Si deseas que un sonido se repita indefinidamente debes utilizar el parámetro `repetir=True`. Por ejemplo:

```
sonido_de_explosion.reproducir(repetir=True)
```

Para detener la música o un efecto de sonido, existe el método `detener`. Por ejemplo:

```
sonido_de_explosion.detener()
```



---

## Dibujado simple en pantalla

---

Hasta ahora hemos mostrado en la pantalla fondos y actores que originalmente estaban almacenados en el disco de la computadora cómo imágenes.

En esta sección veremos como dibujar libremente, ya sean líneas, círculos, texto etc..

Comenzaremos con una forma de dibujado muy sencilla, y luego veremos algunas alternativas que ofrecen mas flexibilidad.

Ten en cuenta que el dibujo avanzado y mas flexible lo vamos a ver en el siguiente capítulo **Dibujo avanzado con Superficies**.



---

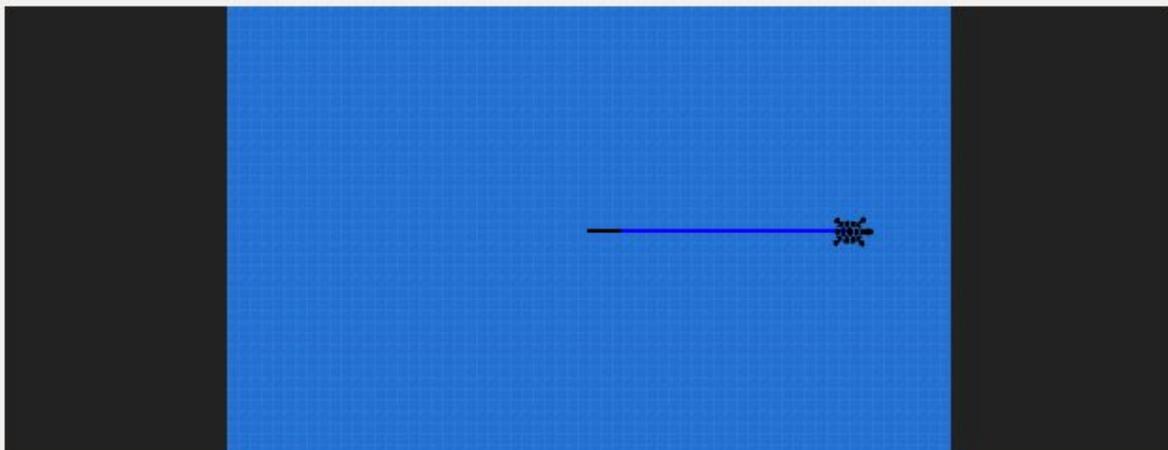
## Usando la Tortuga para dibujar

---

El actor `Tortuga` está inspirado en una de las actividades más divertidas, didácticas y simbólicas del lenguaje de programación **logo**, creado por **Seymour Papert**.

La `Tortuga` básicamente es un actor que sabe dibujar sobre la ventana de pilas. Para ello el programador tiene que indicarle a la tortuga qué movimiento debe realizar.

La siguiente imagen muestra lo que podría dibujar la tortuga con algunas sentencias de movimientos:



```
» import pilas
»
» pilas.iniciar()
» mono = pilas.actores.Mono()
» mono.eliminar()
»
» tortuga = pilas.actores.Tortuga()
» tortuga.avanzar(30)
» tortuga.color = pilas.colores.azul
» tortuga.avanzar(200)
» |
```

La imagen no alcanza a mostrar por completo el concepto, pero en pocas palabras se puede sintetizar lo que realmente hace.

El dibujo de la imagen anterior es una traza que va dibujando la tortuga a partir de su movimiento.

El siguiente código es el que se utilizó para dibujar esa línea de dos colores:

```
import pilas

pilas.iniciar()
tortuga = pilas.actores.Tortuga()
tortuga.avanzar(30)
tortuga.color = pilas.colores.azul
tortuga.avanzar(200)
```

Algo muy valioso en términos didácticos, es que uno podría animarse a realizar dibujos simples como una casa, un hexágono o cualquier otra figura diseñando una estrategia de movimiento para la tortuga.

## 10.1 Inspeccionando a la tortuga

Para manejar a este actor tienes varios comandos inspirados en logo.

Esta es una lista de los comandos más utilizados:

Método completo	nombre corto	ejemplo	¿que hace?
avanzar	av	tortuga.av(10)	avanza en dirección a donde mira la tortuga.
giraderecha	gd	tortuga.gd(45)	gira hacia la derecha los grados indicados.
giraizquierda	gi	tortuga.gi(45)	gira hacia la izquierda los grados indicados.
subelapiz	sl	tortuga.sl()	deja de dibujar cuando se mueve la tortuga.
bajalapiz	bl	tortuga.bl()	comienza a dibujar cuando la tortuga se mueve.
pon_color pintar	pc pintar	tortuga.pc(pilas.colores.rojo) tor- tuga.pintar(pilas.colores.blanco)	dibuja con el color indicado. pinta toda la pantalla del mismo color.

Por supuesto hay algunos más, pero esos quedan para investigar en la clase propiamente dicha. Recuerda que el auto-completado del intérprete de python es bastante útil para estas cosas.

---

## Usando una Pizarra

---

Si quieres dibujar sobre la pantalla pero de forma inmediata y con algunas posibilidades más elaboradas, puedes usar un actor llamado `Pizarra`. Esta no es la forma más avanzada, pero es el siguiente paso después de dominar al actor `Tortuga`.

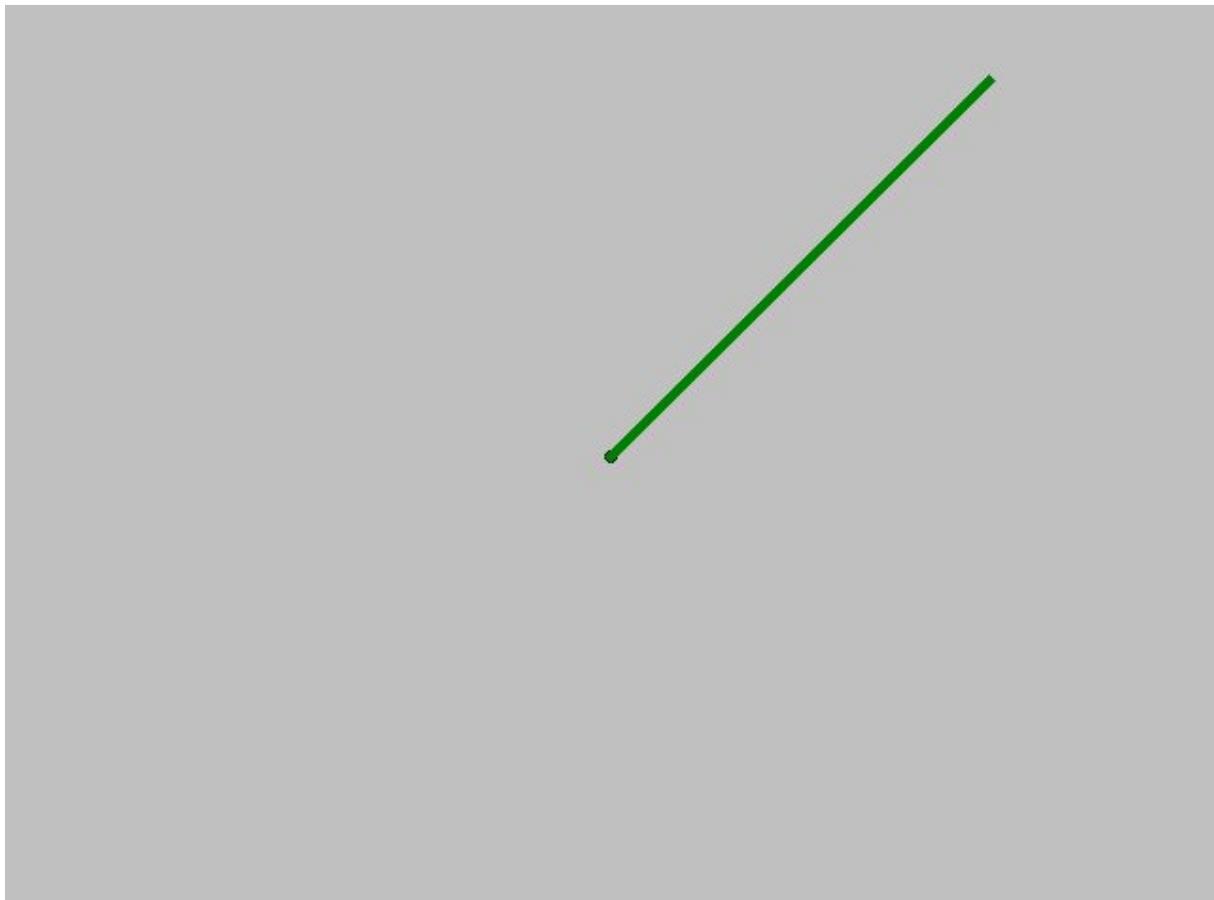
Este actor `Pizarra` es como un lienzo invisible sobre el que podemos pintar imágenes, figuras geométricas y trazos de cualquier tipo. De hecho, el actor `Tortuga` que vimos antes, en realidad estaba dibujando sobre una pizarra, solo que lo hacía con animaciones y algo lento.

Comencemos con algo sencillo: para crear la pizarra y dibujar un punto en el centro de la pantalla se puede usar el siguiente código:

```
pizarra = pilas.actores.Pizarra()
pizarra.dibujar_punto(0, 0)
```

incluso podrías usar el argumento opcional `color` si prefieres otro color, o trazar una línea:

```
pizarra.linea(0, 0, 200, 200, pilas.colores.verdeoscuro, grosor=5)
```



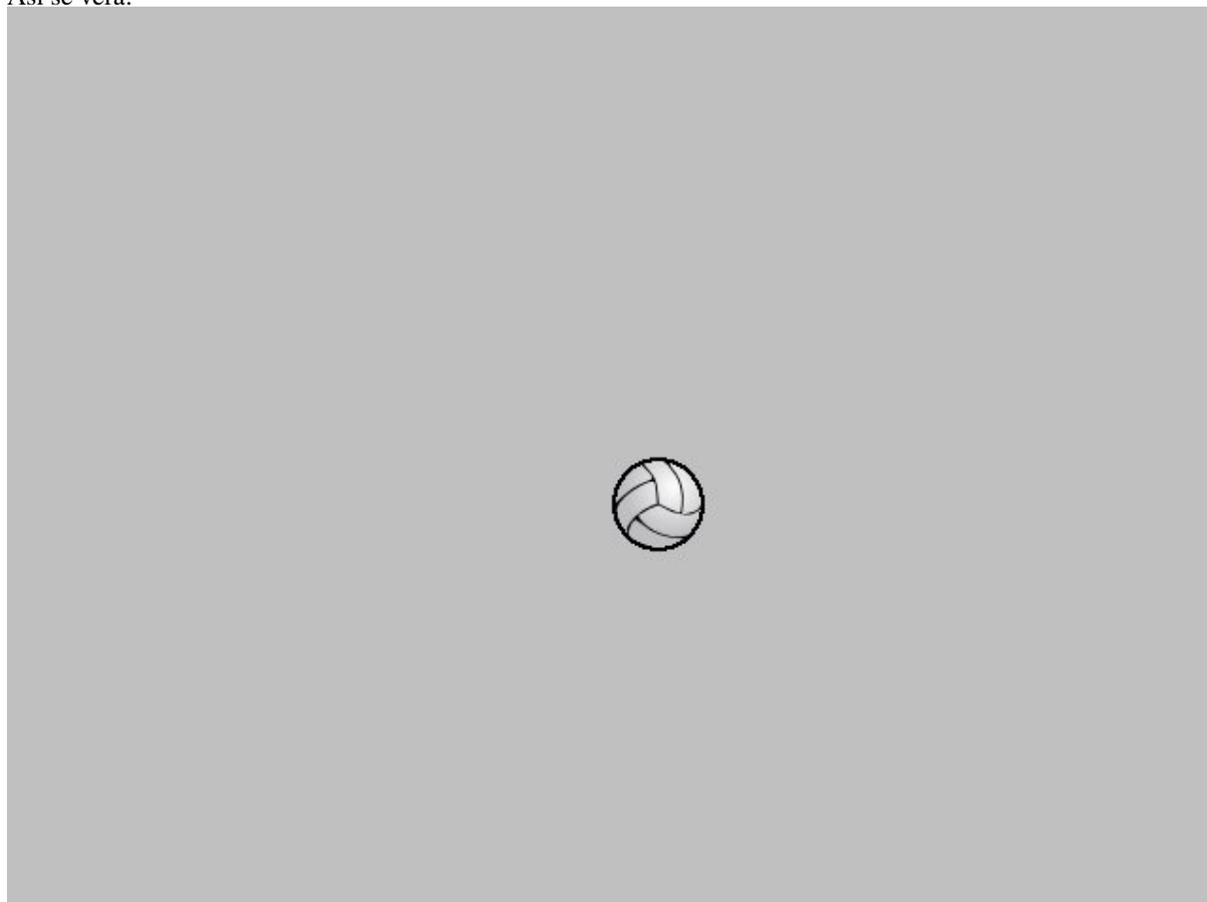
## 11.1 Pintando imágenes

Las pizarras también pueden dibujar imágenes sobre la superficie, y esto es útil cuando quieras crear pinceles especiales sobre la pizarra o construir un escenario usando bloques tipo ladrillos.

Para pintar una imagen solo tienes que cargarla e indicarla a la pizarra que la dibuje en una posición determinada.

```
imagen = pilas.imagenes.cargar("pelota.png")
pizarra.pintar_imagen(imagen, 0, 0)
```

Así se verá:



Ten en cuenta que en estos casos, cuando estamos dibujando una imagen sobre otra, el punto destino (x, y) siempre indica la esquina superior izquierda de la imagen, no el centro u otra posición relativa como en el caso de los actores.

## 11.2 Pintando porciones de imágenes

Hay ocasiones en las que te resultará útil poder pintar solamente porciones de una imagen sobre otra. Para estos casos está el método `pintar_parte_de_imagen`.

Veamos la definición del método:

```
def pintar_parte_de_imagen(self, imagen, origen_x, origen_y, ancho, alto, x, y):
```

## 11.3 Dibujando grillas

De manera similar a las imágenes normales, sobre las pizarras también se pueden pintar grillas.

Solamente tenemos que crear la grilla, seleccionar el cuadro de animación y después decirle a la pizarra que pinte el cuadro actual de la grilla:

```
grilla = pilas.imagenes.cargar_grilla("pingu.png", 10)
pizarra.pintar_grilla(grilla, 0, 0)
```

Así se verá:



Esto es útil cuando se quieren pintar bloques de un escenario completo, por ejemplo podríamos tener una grilla con distintos tipos de suelos (pasto, piedra, tierra) y luego ir imprimiendo sobre una pizarra para formar un escenario completo.

## 11.4 La pizarra como actor

Recuerda que la pizarra también es un actor, así que puedes enseñarle habilidades, cambiar su posición, rotación o lo que quieras.

Recorrido modulos avanzados:



---

## Dibujado avanzado con Superficies

---

Anteriormente vimos que los actores podían tener un aspecto visual, ya sea gracias a una imagen completa, una grilla o un dibujo de pizarra.

Pero hay situaciones donde realmente necesitamos algo más. En muchas ocasiones necesitamos que los actores puedan tener una apariencia que construimos programáticamente (si existe la palabra...).

Por ejemplo, imagina que queremos hacer un indicador de energía, un cronómetro, un indicador de vidas, un botón etc...

### 12.1 Dibujando sobre superficies

En pilas una superficie es una imagen, pero que no se carga directamente desde el disco, sino que se construye en memoria de la computadora, se puede dibujar sobre ella y luego se le puede aplicar a un actor como apariencia.

Comencemos con un ejemplo sencillo, imagina que queremos hacer un actor muy feo, de color “verde” y con dos ojitos. Lo primero que tenemos que hacer es crear una superficie, dibujar sobre ella, y luego crear un actor con esa apariencia:

```
import pilas

pilas.iniciar()

superficie = pilas.imagenes.cargar_superficie(100, 100)

# dibujamos el cuerpo
superficie.circulo(50, 50, 40, color=pilas.colores.verdeoscuro, relleno=True)

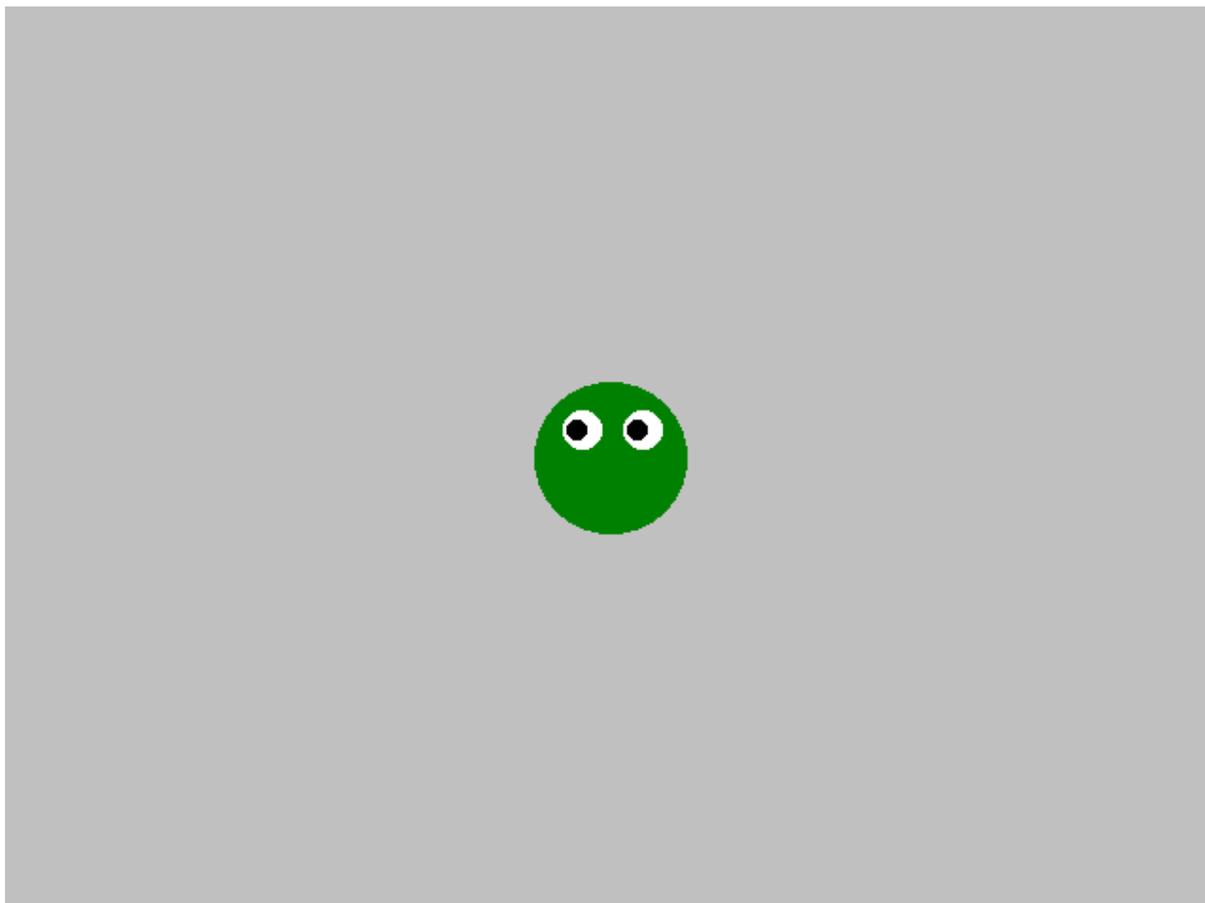
# un ojo
superficie.circulo(35, 35, 10, color=pilas.colores.blanco, relleno=True)
superficie.circulo(32, 35, 5, color=pilas.colores.negro, relleno=True)

# el otro ojo
superficie.circulo(67, 35, 10, color=pilas.colores.blanco, relleno=True)
superficie.circulo(64, 35, 5, color=pilas.colores.negro, relleno=True)

pilas.actores.Actor(superficie)

pilas.ejecutar() # Necesario al ejecutar en scripts.
```

Es decir, una vez que creamos la superficie, en realidad lo que obtenemos es un objeto que se comporta como una imagen, pero con la diferencia que podemos dibujar sobre ella libremente y crear desde el código la imagen que queramos:



Ten en cuenta que también estamos mostrando la superficie gracias a un actor, así que si rotamos el actor o cambiamos su escala la superficie se observará de forma transformada.

Vamos a ver con mas detalle este recurso de pilas, porque ofrece muchas mas funcionalidades de las que vemos en este ejemplo.

## 12.2 Creación de una superficie

Para crear una superficie tenemos que invocar a la función `pilas.imagenes.cargar_superficie` como vimos mas arriba. Esta función admite dos parámetros que indican el ancho y el alto de la superficie.

A partir de ese momento, la superficie será completamente transparente, y lo que dibujemos sobre ella hará que no se note que en realidad es un rectángulo. Vale aclarar que efectivamente todas las imágenes de los videojuegos son rectangulares aunque se disimule...

## 12.3 Coordenadas de las superficies

Las coordenadas que se tienen que especificar para dibujar sobre una superficie son diferentes a las coordenadas cartesianas que usamos en la ventana de pilas.

El motivo de este cambio es que las superficies están en la memoria de la computadora, y es mas sencillo tratar con ellas si usamos el mismo sistema de coordenadas que se usa en casi todas las aplicaciones gráficas. Ten en cuenta que estas son funciones avanzadas y que generalmente se trabaja sobre estas funciones unas pocas veces para lograr lo que ya no está implementado como un actor...

El sistema de coordenadas de las superficies tiene su origen en la esquina superior izquierda  $(0, 0)$ , luego el eje  $x$  crece hacia la derecha y el eje  $y$  crece hacia abajo.

## 12.4 Métodos para dibujar

### 12.4.1 Pintar

Originalmente cuando creamos una superficie es completamente transparente. Si queremos cambiar esto y pintar toda la superficie de un color plano, podemos usar el siguiente método:

```
superficie.pintar(color)
```

Donde el argumento color puede ser algo cómo `pilas.colores.rojo` o un color personalizado indicando las componentes de color rojo, verde y azul. Por ejemplo:

```
superficie.pintar(pilas.colores.Color(100, 255, 0))
```

### 12.4.2 Circulo

Para pintar círculos podemos usar el método `circulo`. Indicando la posición del círculo, su radio y el color.

Ten en cuenta que también debemos indicar si queremos un círculo completamente sólido y pintado o solamente un borde.

Esta es la definición del método:

```
def circulo(self, x, y, radio, color=colores.negro, relleno=False, grosor=1):
```

Si invocamos a la función solamente con sus argumentos principales, obtendremos una silueta de circunferencia sin relleno, por ejemplo:

```
figura.circulo(50, 50, 100)
```

o si queremos un trazo mas grueso:

```
figura.circulo(50, 50, 100, grosor=5)
```

aunque también podemos indicarle que la circunferencia tiene que estar pintada y con otro color:

```
figura.circulo(50, 50, 100, pilas.colores.rojo, relleno=True)
```

### 12.4.3 Rectángulo

El dibujo de rectángulos es muy similar al de círculos, solo que aquí tenemos que indicar la coordenada de la esquina superior izquierda del rectángulo y el tamaño, en ancho y alto.

Esta es la definición del método:

```
def rectangulo(self, x, y, ancho, alto, color=colores.negro, relleno=False, grosor=1):
```

### 12.4.4 Línea

Una línea se compone obligatoriamente de puntos, los que marcan el principio y el final de la línea. Para esto se tienen que usar 4 números, dos para cada punto.

Por ejemplo, el siguiente código dibuja una línea diagonal de color rojo y con 3 píxeles de grosor:

```
superficie.linea(20, 20, 50, 50, pilas.colores.rojo, 3)
```

### 12.4.5 Texto

El dibujo de texto se realiza siempre a partir de una cadena de texto. Y opcionalmente se pueden especificar otros parámetros cómo la posición del texto, el color, el tamaño de las letras y la tipografía.

Este es un ejemplo sencillo que imprime un texto de color azul:

```
superficie.texto("Hola mundo", magnitud=20, fuente="Courier", color=pilas.colores.azul)
```

Ten en cuenta que la fuente se indica como una cadena, y el valor que podemos poner ahí es el de cualquiera de nuestras fuentes del sistema. Si nuestro sistema no tiene la fuente que le solicitamos, se imprimirá usando una tipografía por defecto.

---

## Manejo de tiempo con tareas

---

Una necesidad muy común en los videojuegos es poder planificar tareas para ser ejecutadas por tiempo. Por ejemplo, en un juego de naves podríamos querer que aparezcan naves enemigas cada dos segundos.

### 13.1 Tareas

Las tareas son acciones que elegimos ejecutar en un determinado momento.

Al momento de crear la tarea tenemos que pensar “en qué momento se tiene que ejecutar la tarea”, y dependiendo de lo que queramos, tenemos que escribir algo cómo:

```
pilas.escena_actual().agregar_tarea(tiempo, funcion, parametros)
```

Hay tres tipos de creaciones de tareas:

- tareas que se ejecutan una vez.
- tareas que se ejecutan siempre.
- tareas condicionales.

las tareas condicionales se ejecutarán siempre y cuando la función que las representa retorna True. Si la función retorna False la tarea dejará de ejecutarse.

### 13.2 Eliminar tareas

Una forma sencilla de detener una tarea es iniciarla cómo condicional, y que la función que le asignamos retorne False.

Otra forma es simplemente capturar el retorno de la función que ha creado la tarea y detenerla.

Por ejemplo:

```
una_tarea = pilas.escena_actual().agregar_tarea_siempre(5, funcion)
```

y luego, cuando queramos que la tarea finalice y no se vuelva a ejecutar, tendríamos que ejecutar una sentencia cómo ésta:

```
una_tarea.terminar()
```



---

## Interpolaciones

---

Las interpolaciones nos permiten lograr movimientos de los actores de manera sencilla.

Por ejemplo, tradicionalmente si quisiéramos cambiar posición de un actor en pantalla podemos usar estas sentencias:

```
actor.x = 10
actor.x = 20
actor.x = 30
etc ...
```

una forma de lograr lo mismo con pilas es asignarle todos los valores en forma de lista:

```
actor.x = range(10, 100, 10)
```

o lo que es lo mismo:

```
actor.x = [10, 20, 30, 40, 50 ... etc.]
```

Y a estas interpolaciones, también le puedes decir cuantos segundos puede demorar. Por ejemplo, para dar un giro completo de 360 grados en 10 segundos puedes hacer algo como:

```
actor.rotacion = [360], 10
```

de hecho, puede que te resulte mas conveniente tener mas control sobre la interpolación, así que puedes usar esta forma:

```
actor.x = pilas.interpolar(100)
```

donde el valor inicial será la posición x del actor y el valor final será 100.

La función `interpolar`, como mencioné antes, te da mas control sobre la interpolación, porque admite otros parámetros de ajuste como los siguientes:

- `duracion`: los segundos que durara la interpolacion.
- `demora`: los segundos que tiene que esperar antes de iniciar la interpolacion.
- `tipo`: tipo de interpolaciones, que generalmente es 'lineal'.

Por ejemplo, si queremos que un personaje dé un giro completo de 360 grados en 10 segundos podemos hacer algo así:

```
actor.rotacion = 0
actor.rotacion = pilas.interpolar(360, duracion=10)
```

### 14.1 Girando un actor

Esta herramienta se puede aplicar a muchas situaciones distintas, por ejemplo si queremos hacer girar un personaje podemos hacer algo como:

```
actor.rotacion = 0
actor.rotacion = pilas.interpolar(360, duracion=5)
```

con lo que estaríamos diciendo al personaje que dé un giro completo (de 0 a 360 grados) en 5 segundos.

También existe un argumento `delay` para demorar el inicio de la interpolación.

## 14.2 Escalando un actor

De manera similar a lo que hicimos anteriormente, podemos aplicarla a la propiedad `escala` una nueva interpolación:

```
actor.escala = pilas.interpolar(2, duracion=5)
```

esto duplicará el tamaño del actor en 5 segundos.

¿Y mas simple?, bueno, como hicimos antes:

```
actor.escala = [2]
```

## 14.3 Interpolaciones en cadena

Si queremos que una interpolación pase por distintos valores podemos hacer algo como esto:

```
actor.x = pilas.interpolar([300, 0, 300], duracion=3)
```

lo que llevará al actor de su posición `x` actual, a 300 en un segundo, y luego a 0 en un segundo y por último de nuevo a 300 en un segundo.

En total, ha consumido 3 segundos en pasar por todos los valores que le indicamos.

---

## Controlando la pantalla

---

Para posicionar actores en el escenario principal es importante conocer las propiedades de la pantalla.

La pantalla es lo que contiene la ventana principal de pilas, y que puede observar una porción del escenario y a un grupo de actores.

### 15.1 Modo depuración

El modo depuración te permite ver información de utilidad cuando estás desarrollando un juego o simplemente buscando algún error.

Para iniciar el modo depuración pulsa **F12**. En la ventana principal aparecerán varios textos indicando el rendimiento del juego, las coordenadas de posición de los actores y la posición del mouse.

El centro de la ventana es, inicialmente, el punto  $(0, 0)$ . Este modelo de coordenadas es el cartesiano, y lo hemos elegido porque es el que mejor se adapta a los conocimientos geométricos que se enseñan en las escuelas.

### 15.2 Orden de impresión: atributo z

Cuando tienes varios actores en pantalla notarás que a veces unos aparecen sobre otros.

Para cambiar este comportamiento tienes que modificar el atributo z de cada actor.

Los valores altos de z indican mucha distancia entre el observador y el escenario. Mientras que valores pequeños z harán que los actores tapen a los demás (porque aparecerán mas cerca del usuario).

Este es un ejemplo de dos configuraciones distintas de atributos z:



mono.z = 3  
banana.z = -4



mono.z = -2  
banana.z = 1

Ten en cuenta que inicialmente todos los actores tienen un atributo  $z=0$ .

### 15.3 Atributos de posición

Todos los actores tienen atributos cómo:

- $x$
- $y$

que sirven para cambiar la posición del actor dentro de la escena.

También encontrarás atributos que permiten hacer lo mismo, pero tomando como referencia alguno de los bordes del actor. Por ejemplo:

- izquierda
- derecha
- arriba
- abajo

Ten en cuenta que estos valores indicarán la posición del actor dentro del escenario, no de la ventana. Esto significa que si cambias la posición de la cámara los actores seguirán estando en la misma posición pero se verán un poco corridos.

---

## Comportamientos

---

En el desarrollo de videojuegos es conveniente tener una forma de indicarle a los actores una rutina o tarea para que la realicen.

En pilas usamos el concepto de comportamiento. Un comportamiento es un objeto que le dice a un actor qué debe hacer en todo momento.

La utilidad de usar componentes es que puedes asociarlos y intercambiarlos libremente para lograr efectos útiles.

Por ejemplo: un guardia de un juego de acción puede ir de un lado a otro en un pasillo:

- caminar hacia la izquierda hasta el fin del pasillo.
- dar una vuelta completa.
- caminar hacia la derecha hasta el fin del pasillo.
- dar una vuelta completa.

En este caso hay 4 comportamientos, y queda en nuestro control si queremos que luego de los 4 comportamientos comience nuevamente.

### 16.1 Un ejemplo, ir de un lado a otro

Veamos un ejemplo sencillo, vamos a crear un actor Mono y decirle que se mueva de izquierda a derecha una sola vez:

```
import pilas

pilas.iniciar()
mono = pilas.actores.Mono()

pasos = 200

moverse_a_la_derecha = pilas.comportamientos.Avanzar(pasos)
mono.hacer_luego(moverse_a_la_derecha)

mono.rotacion = [180] # Dar la vuelta.

moverse_a_la_izquierda = pilas.comportamientos.Avanzar(pasos)
mono.hacer_luego(moverse_a_la_izquierda)

pilas.ejecutar() # Necesario al ejecutar en scripts.
```

De hecho, tenemos una variante que puede ser un poco más interesante; decirle al mono que repita estas tareas todo el tiempo:

```
mono.hacer_luego(moverse_a_la_derecha, True)
```

Donde el segundo argumento indica si el comportamiento se tiene que repetir todo el tiempo o no.

---

## Controles

---

Si quieres conocer el estado de los controles en pilas, tienes que usar el objeto `pilas.mundo.control`.

Por ejemplo, para hacer que un actor se mueva por la pantalla simplemente puedes crear al actor y escribir estas sentencias.

```
if pilas.escena_actual().control.izquierda:
    mono.x -= 1
elif pilas.escena_actual().control.derecha:
    mono.x += 1
```

Esta no es la única forma de mover a un personaje por la pantalla, pero suele ser la más conveniente porque es muy directa, y se puede escribir en cualquier parte del código.

---

**Note:** Recuerda que para poder interactuar con el teclado, debes seleccionar la ventana de resultado. Haz click en la parte de arriba del intérprete de pilas para seleccionarla.

---

### 17.1 Investigando al objeto control

En realidad, cuando usamos a `pilas.mundo.control`, accedemos a un objeto que tienen varios atributos.

Estos atributos pueden valer `True` o `False`, dependiendo de la pulsación de las teclas:

- izquierda
- derecha
- arriba
- abajo
- boton

Esto te permite simplificar el código, porque no tienes que conocer específicamente las teclas que activan cada acción, y en todos los juegos serán las mismas teclas.

### 17.2 ¿Dónde consultar los controles?

El mejor lugar para consultar el estado de los controles es en la actualización de un actor.

Esto se logra colocando un método llamado `actualizar` dentro del actor y haciendo la consulta ahí. Veamos un actor sencillo que se pueda mover de izquierda a derecha. El código sería así:

```
import pilas

pilas.iniciar()
```

```
class MiActor(pilas.actores.Actor):

    def __init__(self):
        pilas.actores.Actor.__init__(self)
        self.imagen = pilas.imagenes.cargar("aceituna.png")

    def actualizar(self):
        if pilas.escena_actual().control.izquierda:
            self.x -= 10

        if pilas.escena_actual().control.derecha:
            self.x += 10

MiActor()

pilas.ejecutar() # Necesario al ejecutar en scripts.
```

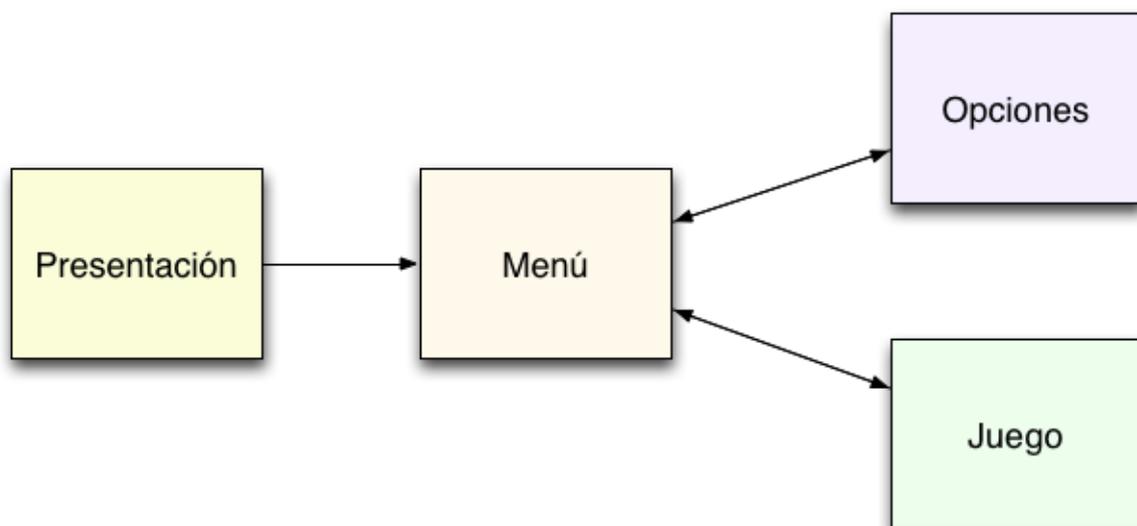
---

## Escenas

---

Las escenas te permiten dividir el juego en partes reconocibles y que interactúan de manera diferente con el usuario.

Un juego típico tendrá al menos una escena como el menú principal, una presentación y una pantalla de juego.



### 18.1 Cosas a tener en cuenta

Hay algunas cosas a tener en cuenta a la hora de manejar escenas, porque simplifican mucho el trabajo posterior:

- La escena actual siempre está señalada por el atributo `pilas.escena_actual()`.
- Solo puede existir una escena activa a la vez.

### 18.2 La escena Normal

Cuando iniciamos pilas por primera vez se creará una escena llamada `Normal`. Esta escena no tiene un comportamiento muy elaborado, simplemente imprime toda la pantalla de gris para que podamos colocar actores sobre ella y veamos una escena limpia.

## 18.3 Pausando el juego

En cualquier momento del juego podemos ejecutar `pilas.escena.pausar()` y el juego quedará pausado hasta que se pulse la tecla ESC y vuelva a la ejecución normal del juego.

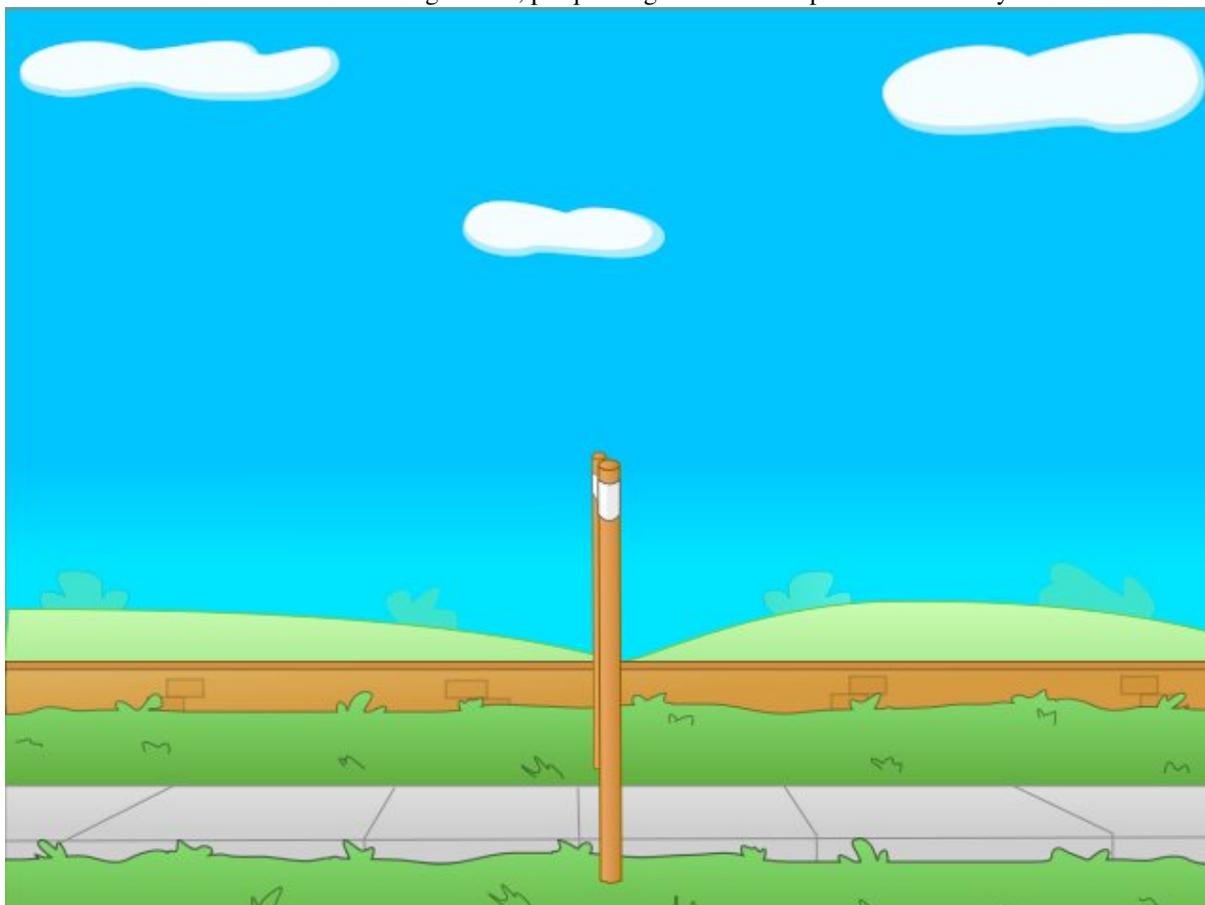
Este comando es un atajo que carga la escena `Pausa`. Puedes crear tu propia pantalla de PAUSA heredando de `pilas.escena.Pausa`.

## 18.4 Cambiando el fondo de las escenas

Para hacer una pequeña prueba sobre una escena, podrías ejecutar la siguiente sentencia de código:

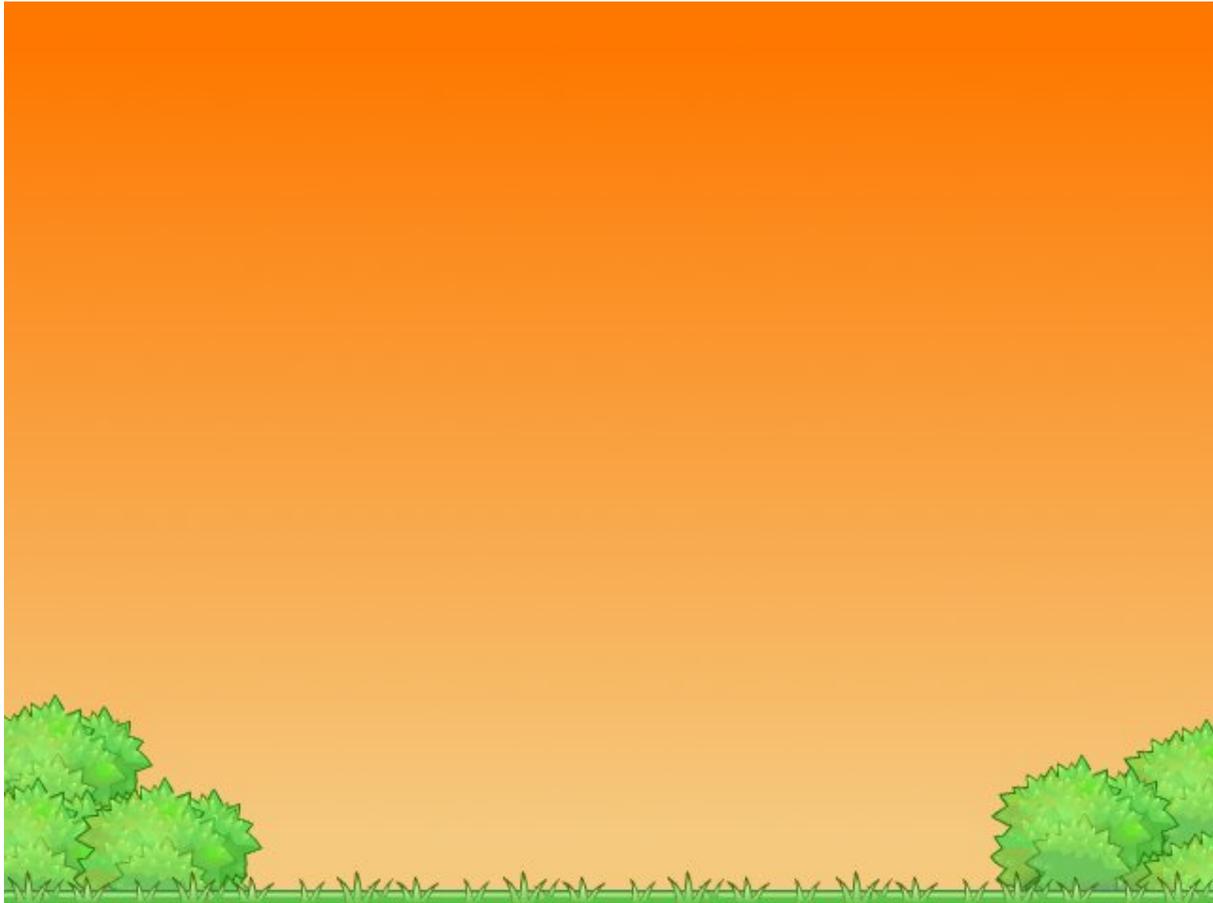
```
pilas.fondos.Volley()
```

Esto le dará a tu escena una vista mas agradable, porque carga un fondo de pantalla colorido y mas divertido:



o podrías usar un fondo de atardecer:

```
pilas.fondos.Tarde()
```



## 18.5 Cómo crear nuevas escenas

Imagina que tienes un juego con dos pantallas, una que simplemente dice “bienvenido” y otra con un personaje para mover.

Claramente tendríamos que hacer dos escenas, e iniciar nuestro juego creando la escena principal.

La primer escena tendríamos que representarla con una clase, que herede de la escena Normal así:

```
class PantallaBienvenida(pilas.escena.Normal):  
  
    def __init__(self):  
        pilas.escena.Normal.__init__(self)  
  
    def iniciar(self):  
        pilas.fondos.Pasto()  
        texto = pilas.actores.Texto("Bienvenido a pilas!!!")
```

Ahora, para poner en funcionamiento esta escena simplemente tienes que decirle a pilas que esta escena es la activa:

```
pilas.cambiar_escena(PantallaBienvenida())
```

Esto eliminará las escenas almacenadas y se colocará como la escena actual y activa:

**See also:**

Mira la documentación *Nuevo Gestor de Escenas* para comprender mejor el apilamiento de escenas.



Ahora, si quieres salir de la escena, simplemente tendrías que hacer un objeto de otra clase que represente otra escena y llamar a uno de estos tres metodos:

- `pilas.cambiar_escena(mi_escena)`
- `pilas.almacenar_escena(mi_escena)`
- `pilas.recuperar_escena()`

## 18.6 Método sobrescribibles

Dentro de cada escena existen 2 métodos que pueden ser sobrescritos.

```
def pausar(self):  
    pass
```

```
def reanudar(self):  
    pass
```

Si quieres tener el control de cuando una escena se queda apilada, deberás sobrescribir el método:

```
def pausar(self):  
    pass
```

Si quieres saber cuando una escena apilada vuelve a ser la escena activa, deberás sobrescribir el método:

```
def reanudar(self):  
    pass
```

---

## Como migrar mi juego al nuevo Gestor de Escenas

---

Antes de migrar tu juego al nuevo sistema de gestión de escenas, es mejor que le des un vistazo a *Nuevo Gestor de Escenas* para comprender mejor el apilamiento de escenas.

Ahora pasamos a explicar los sencillos pasos a seguir para hacer la migración de tu juego.

### 19.1 Iniciar el juego

Tu juego debe tener una estructura de inicio parecida a la siguiente:

```
import pilas
import escena_menu

pilas.iniciar(titulo='Mi titulo')

escena_menu.EscenaMenu()

pilas.ejecutar()
```

Lo único que deberás cambiar aquí es la línea que llama a la escena. Tendrá que quedar de la siguiente forma:

```
import pilas
import escena_menu

pilas.iniciar(titulo='Mi titulo')

# Esta es la línea que debemos cambiar
pilas.cambiar_escena(escena_menu.EscenaMenu())

pilas.ejecutar()
```

### 19.2 Escenas del juego

Todas las escenas de tu juego deben heredar ahora de *pilas.escena.Base*.

```
class MiEscena(pilas.escena.Base):
```

Y el otro cambio que debes realizar en las escenas es que el método `__init__(self)` no debe contener nada más que la llamada al `__init__` de la escena Base

```
def __init__(self):
    pilas.escena.Base.__init__(self)
```

Luego debes definir un método `iniciar(self)` donde podrás crear los nuevos actores y lo necesario para iniciar tu escena.

```
def iniciar(self):
    pilas.fondos.Pasto()
    mono = pilas.actores.Mono()
```

Aquí un ejemplo de como debería ser el cambio.

### Escena antigua

```
class MiEscena(pilas.esenas.Escena):

    def __init__(self):
        pilas.esenas.Escena.__init__(self)

        pilas.fondos.Pasto()
        mono = pilas.actores.Mono()
```

### Escena nueva

```
class MiEscena(pilas.esena.Base):

    def __init__(self):
        pilas.esena.Base.__init__(self)

    def iniciar(self):
        pilas.fondos.Pasto()
        mono = pilas.actores.Mono()
```

## 19.3 Cambio de Escena

En algún punto de tu juego, llamarías a otra escena para cambiarla.

```
escena_juego.Escena_Juego()
```

Debes sustituir esta llamada a la nueva escena por esta otra forma:

```
pilas.cambiar_escena(escena_juego.Escena_Juego())
```

## 19.4 Eventos

Ahora los eventos son individuales por cada escena. Si quieres conectar a algún evento, como *mueve\_mouse*, *actualizar*, *pulsa\_tecla*, puedes hacerlo de cualquiera de las dos siguientes formas:

```
def mi_metodo(evento):
    # Hace algo

pilas.eventos.actualizar.conectar(mi_metodo())

# Otra forma de conectar
pilas.escena_actual().actualizar.conectar(mi_metodo())
```

Ambas formas conectan a los eventos de la escena actualmente activa.

Si deseas crear tu propio evento, lo deberás hacer de la siguiente forma:

```
pilas.eventos.mi_evento_personalizado = pilas.evento.Evento("mi_evento_personalizado")
pilas.eventos.mi_evento_personalizado.conectar(self._mi_evento_personalizado)
```

## 19.5 Fin de la migración

Con estos simples pasos, tu juego debe funcionar sin problemas con el nuevo sistema de gestión de escenas.

Ante cualquier problema no tengas dudas en ponerte en contacto con nosotros mediante el [foro de losersjuegos](#).



---

## Nuevo Gestor de Escenas

---

Pilas contiene un nuevo gestor de escenas que permite tener más de una escena en el juego, aunque sólo una de ellas será la activa.

Esta nueva funcionalidad nos permitiría, por ejemplo, estar jugando y en cualquier momento pulsar una tecla y acceder a las opciones del juego.

Allí quitaríamos el sonido y luego pulsando otra tecla volveríamos al juego, justo donde lo habíamos dejado.

Nuestros actores estarán en la misma posición y estado en el que los habíamos dejado antes de ir a las opciones.

### 20.1 Escena Base

Es la Escena de la cual deben heredar todas las escenas del juego en pilas.

```
pilas.escena.Base
```

El antiguo método para crear una escena era el siguiente:

```
class MiEscena(pilas.escenas.Escena):
    def __init__(self):
        pilas.escenas.Escena.__init__(self)

        pilas.fondos.Pasto()
        mono = pilas.actores.Mono()
```

Ahora el nuevo método para crear una escena es el siguiente:

```
class MiEscena(pilas.escena.Base):
    def __init__(self):
        pilas.escena.Base.__init__(self)

    def iniciar(self):
        pilas.fondos.Pasto()
        mono = pilas.actores.Mono()
```

Como puedes observar, ahora la escena hereda de

```
pilas.escena.Base
```

Otro cambio **muy importante** es que el método `__init__(self)` no debe contener nada más que la llamada al `__init__` de la escena Base.

```
def __init__(self, titulo):
    pilas.escena.Base.__init__(self)
```

```
self._titulo = titulo
self._puntuacion = puntuacion
```

Puedes almacenar unicamente parámetros que quieras pasar a la escena. Por ejemplo así:

```
def __init__(self, titulo):
    pilas.escena.Base.__init__(self)

    self._titulo = titulo
```

Y por último debes definir un método `iniciar(self)` donde podrás crear los nuevos actores y lo necesario para iniciar tu escena.

```
def iniciar(self):
    pilas.fondos.Pasto()
    mono = pilas.actores.Mono()
    texti = pilas.actores.Texto(self._titulo)
```

## 20.2 Iniciar pilas con una Escena

Para iniciar pilas, con el nuevo sistema, debemos ejecutar lo siguiente

```
pilas.cambiar_escena(mi_escena.MiEscena())
pilas.ejecutar()
```

Te habrás fijado que pilas dispone de un nuevo método para realizar esta acción.

```
pilas.cambiar_escena(escena_a_cambiar)
```

En el próximo punto explicaremos su función junto con otros 2 metodos nuevos.

## 20.3 Cambiar entre Escenas

Antes de nada debes comprender que pilas tiene la capacidad de apilar el número de escenas que desees en su sistema.

El método de apilamiento es FILO (First In, Last Out), la primera escena en entrar en la pila será la última en salir.

¿Y como apilamos, recuperamos y cambiamos escenas?, muy sencillo. Pilas dispone de 3 métodos para realizar esta operaciones:

```
pilas.cambiar_escena(mi_escena)

pilas.almacenar_escena(mi_escena)

pilas.recuperar_escena()
```

- `pilas.cambiar_escena(mi_escena)`: VACIA por completo la pila de escenas del sistema e incorporar la escena que pasamos como parámetro. La escena incorporada será la escena activa.
- `pilas.almacenar_escena(mi_escena)`: apila la escena actual y establece como escena activa la que le pasamos como parámetro. La escena que ha sido apilada quedará pausada hasta su recuperación.
- `pilas.recuperar_escena()`: recupera la última escena que fué apilada mediante `almacenar_escena()` y la establece como escena activa.

Por último indicar que si quieres tener acceso a la escena actualmente activa, puedes hacerlo mediante el comando:

```
pilas.escena_actual()
```

---

## Demos

---

Pilas viene con un módulo de ejemplos muy sencillos para que puedas investigar y jugar.

Este módulo de ejemplos es `pilas.demos` y está organizado como un conjunto de clases sencillas.

Para poner en funcionamiento alguna de las demos simplemente tienes que hacer un objeto a partir de alguna de las clases que verás en el módulo y ejecutar su método 'iniciar'.

### 21.1 Piezas

Hay un pequeño rompecabezas que se puede iniciar con el siguiente código:

```
import pilas

pilas.iniciar()
piezas = pilas.demos.piezas.Piezas()
piezas.iniciar()

pilas.ejecutar() # Necesario al ejecutar en scripts.
```

inmediatamente después de evaluar estas sentencias, aparecerá en pantalla un grupo de piezas para que puedas empezar a acomodarlas usando el mouse.



Ten en cuenta que los ejemplos también reciben parámetros, así podemos alterar un poco mas el funcionamiento del minijuego.

Veamos cómo podemos crear un rompecabezas distinto a partir del ejemplo Piezas.

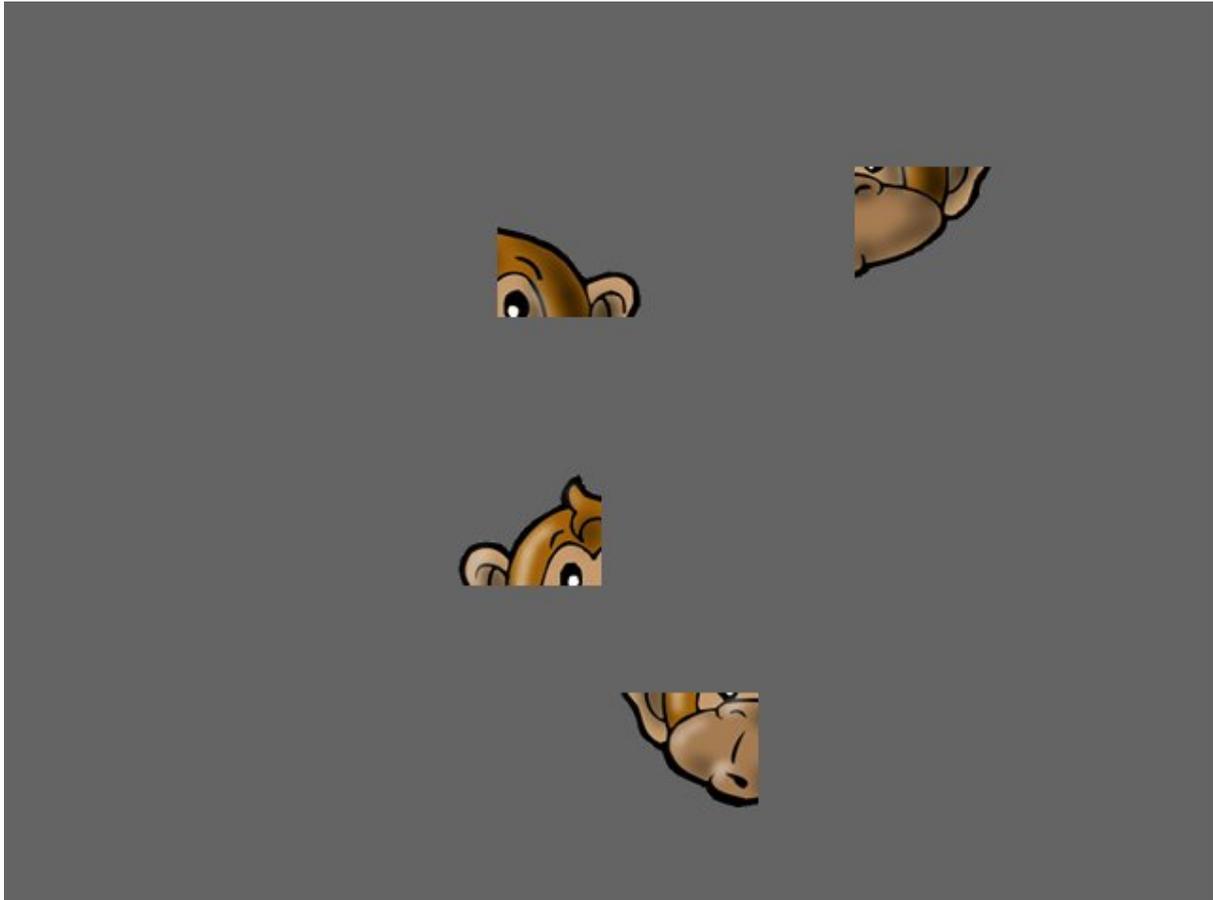
Escribe lo siguiente:

```
import pilas

pilas.iniciar()
piezas = pilas.demos.piezas.Piezas("mono.png", 2, 2)
piezas.iniciar()

pilas.ejecutar() # Necesario al ejecutar en scripts.
```

Si, ahora en pantalla aparece la imagen del mono pero separado en 4 piezas. Dado que hemos especificado 2 (filas) y 2 (columnas).



Puedes usar cualquier imagen que quieras para construir tu ejemplo de piezas.

También se admite una función como argumento al crear el objeto, que se invocará cuando el usuario complete el rompecabezas:

```
import pilas

pilas.iniciar()

def cuando_se_complete():
    pilas.avisar("Lo has completado!!!")

piezas = pilas.demos.piezas.Piezas("mono.png", 2, 2, cuando_se_complete)
piezas.iniciar()

pilas.ejecutar() # Necesario al ejecutar en scripts.
```



Lo has completado!!!

---

## Interfaz de usuario

---

Pilas incluye un submódulo que te permitirá crear una interfaz de usuario con deslizadores, botones, cajas de texto y selectores.

Este submódulo tiene objetos basados en actores, así que lo que conoces sobre actores vas a poder usarlo para construir una interfaz.

### 22.1 Propiedades comunes

Todos los elementos de la interfaz comparten una serie de propiedades como:

**activar** Permite que un control sea seleccionado y que el usuario interactúe con él. Por defecto todos los controles están activados.

```
entrada = pilas.interfaz.IngresoDeTexto()
entrada.texto = "Texto inicial"
```

```
entrada.activar()
```

**desactivar** Bloquea el control para que el usuario no pueda utilizarlo. El control se queda semi-transparente para indicar este estado.

```
entrada.desactivar()
```

**ocultar** Oculta el control en la pantalla.

```
entrada.ocultar()
```

**mostrar** Muestra el control y lo activa.

```
entrada.mostrar()
```

**obtener\_foco** Establece el control como activo.

```
entrada.obtener_foco()
```

### 22.2 Deslizador

El deslizador es útil para que el usuario pueda seleccionar un valor intermedio entre dos números, por ejemplo entre 0 y 1, 0 y 100 etc.

Un ejemplo típico de este componente puedes encontrarlo en las preferencias de audio de algún programa de sonido, los deslizadores te permiten regular el grado de volumen.

Esta es una imagen del ejemplo `deslizador.py` que está en el directorio `ejemplos`. Tiene tres deslizadores, y el usuario puede regular cualquiera de los tres para ver los cambios en el actor:



Para construir un deslizador y asociarlo a una función puedes escribir algo como esto:

```
def cuando_cambia(valor):  
    print "El deslizador tiene grado:", valor
```

```
deslizador = pilas.interfaz.Deslizador()  
deslizador.conectar(cuando_cambia)
```

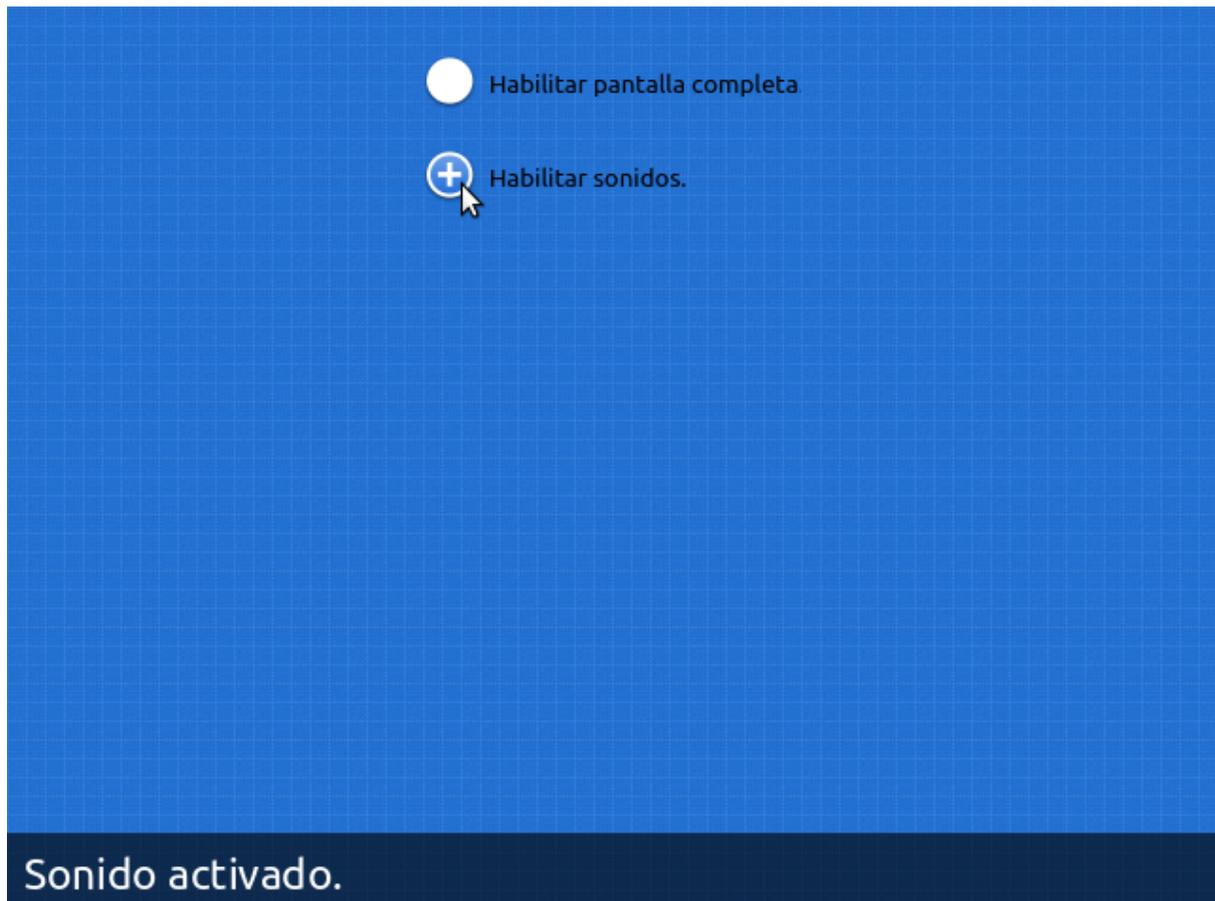
Entonces, a medida que muevas el deslizador se imprimirán en pantalla valores del 0 al 1, por ejemplo 0.25, 0.52777 etc...

Si quieres cambiar los valores iniciales y finales de la escala de valores, lo mas sencillo es multiplicar el argumento `valor` de la función. Por ejemplo, si quieres valores entre 0 y 100:

```
def cuando_cambia(valor):  
    valor_entre_cero_y_cien = valor * 100
```

## 22.3 Selector

El selector te permite seleccionar una opción con dos valores: habilitado, deshabilitado.



Se puede usar para opciones cómo habilitar o deshabilitar el modo pantalla completa o algo así.

Para usarlo, se tiene que crear un objeto de la clase `Selector` y un texto a mostrar:

```
selector = pilas.interfaz.Selector("Habilitar pantalla completa.", x=0, y=200)
```

Y luego, puedes consultar el valor del selector mediante el atributo `seleccionado`:

```
if selector.seleccionado:
    print "El selector esta seleccionado."
else:
    print "El selector no esta seleccionado."
```

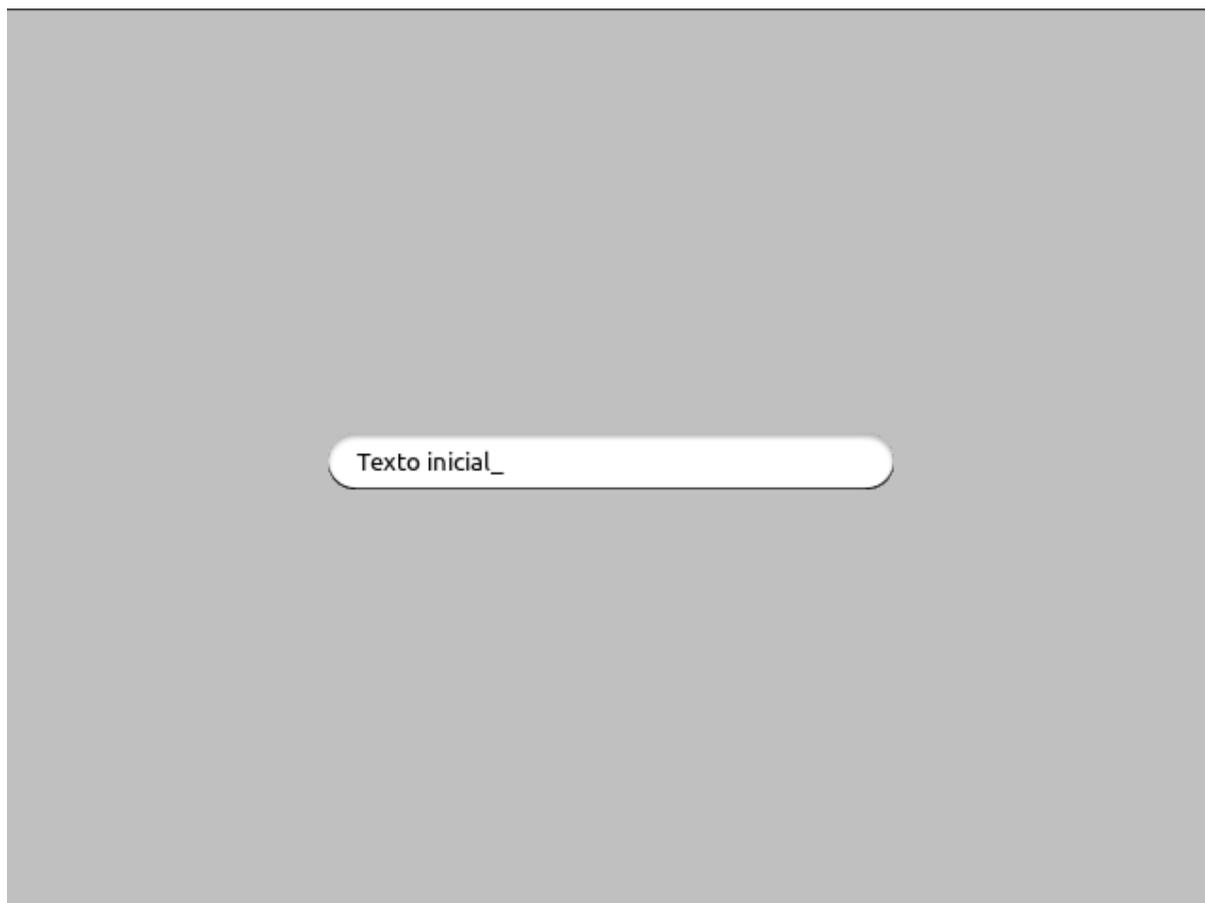
o directamente asociarle una función para que el selector la llame cuando cambia de estado:

```
def cuando_el_selector_cambia(estado):
    print "El selector ahora esta en estado:", estado

selector.definir_accion(cuando_el_selector_cambia)
```

## 22.4 Ingreso de texto

Si quieres solicitar datos, como el nombre del usuario, puedes usar el objeto `IngresoDeTexto`. Ya que muestra una caja y un pequeño cursor para ingresar texto:



Para usar este componente tienes que crearlo y luego leer o escribir el atributo `texto`, que contiene la cadena de texto de la caja:

```
entrada = pilas.interfaz.IngresoDeTexto()
entrada.texto = "Texto inicial"
```

Inicialmente, el objeto `IngresoDeTexto` toma un tamaño y apariencia predeterminado. Pero esto se puede cambiar fácilmente usando argumentos al momento de crear el componente.

Por ejemplo, podríamos enviarle cómo argumento un tamaño mas pequeño y un ícono de búsqueda:

```
entrada = pilas.interfaz.IngresoDeTexto(ancho=100, icono='iconos/lupa.png')
```

u otro ícono:

```
entrada = pilas.interfaz.IngresoDeTexto(ancho=100, icono='iconos/ok.png')
```

La caja también tiene otros métodos para permitir o prohibir el ingreso de datos.

Por ejemplo, podríamos decirle a la caja que solo permita el ingreso de números, letras, o poner un límite de cantidad de caracteres. Los métodos son:

- `solo_numeros()`
- `solo_letras()`

y el límite de caracteres está indicado por la referencia `limite_de_caracteres`:

```
print "El limite de caracteres es"
print entrada.limite_de_caracteres
```

```
entrada.limite_de_caracteres = 50
```

## 22.5 Lista de selección

La lista de selección se utiliza para mostrar al usuario una lista de cadenas, y permitirle seleccionarlas con el mouse.

Para crear un lista de selección, se tiene que crear una lista de cadenas y declarar una función para que sea llamada cuando se termina de seleccionar.

Por ejemplo, el siguiente código muestra una lista e imprime por consola cuando el usuario selecciona con el click del mouse:

```
def cuando_selecciona(opcion):  
    print "Ha seleccionado la opcion:", opcion  
  
consulta = pilas.interfaz.ListaSeleccion(['Uno', 'Dos', 'Tres'], cuando_selecciona)
```



---

## Como crear menús para tu juegos

---

Para crear menús en tus juegos puedes usar el actor `Menu`.

El actor `Menu` tiene la funcionalidad de representar opciones y que le puedas asociar nombres de funciones para invocar.

Un menú sencillo podría tener dos opciones, una para iniciar el juego y otra para salir:

```
import pilas

pilas.iniciar()
pilas.fondos.Selva()

def iniciar_juego():
    print "Tengo que iniciar el juego"

def salir_del_juego():
    print "Tengo que salir..."

pilas.actores.Menu(
    [
        ('iniciar juego', iniciar_juego),
        ('salir', salir_del_juego),
    ])

pilas.ejecutar()
```

Si escribes este texto en un programa, funciona, aunque no es muy útil: solamente creará una ventana con dos opciones, que se pueden seleccionar usando el teclado.

Esta es una imagen de cómo se vé el menú del ejemplo de mas arriba:



Cada vez que selecciones una opción aparecerá un mensaje en la consola de python.

## 23.1 Creando funciones de respuesta

Si observas con atención el primer ejemplo de código, hay dos partes que son muy importantes.

Primero declaramos funciones que hacen algo, como por ejemplo:

```
def iniciar_juego():  
    print "Tengo que iniciar el juego"
```

Y luego, cuando creamos el menú, armamos una lista de tuplas, donde el primer elemento es la cadena de texto que queremos mostrar, y el segundo elemento es la función a invocar:

```
pilas.actores.Menu(  
    [  
        ('iniciar juego', iniciar_juego),  
        ('salir', salir_del_juego),  
    ]  
)
```

Es importante que el argumento se construya usando una lista como la anterior.

Puedes crear tantas opciones como quieras, pero siempre tienen que estar en una tupla de dos elementos, el primer con un texto y el segundo con el nombre de la función que se tiene que invocar.

Cuando colocamos un nombre de función de esa forma, es decir, sin los paréntesis, decimos que esa función será una función de respuesta para el menú. Y aunque parezca un recurso muy simple, funciona bastante bien en casi todos los casos. Por ejemplo, nuestro código anterior se podría poner más interesante si mejoramos la función `iniciar_juego` y la función `salir_del_juego`:

```
def iniciar_juego():  
    pilas.ejemplos.Piezas()  
  
def salir_del_juego():  
    pilas.terminar()
```

## 23.2 Los menús son actores

Ten en cuenta que el menú también es un actor, así que podrás posicionarlo en la ventana, o cambiar su tamaño como si se tratara de cualquier otro personaje del juego:

```
mi_menu.escala = 2  
mi_menu.x = [300, 0]
```

Muchos juegos hace uso de esta característica, por ejemplo, para que el menú aparezca por debajo de la pantalla o que se mueva constantemente como si estuviera flotando.

Ten en cuenta que en realidad no estamos aplicando transformaciones a todo el menú, simplemente estamos transmitiendo las transformaciones a cada uno de los textos que componen el menú. Si haces un cambio de rotación vas a ver a qué me refiero...



---

## Mapas y plataformas

---

En los años 80 uno de los géneros de videojuegos más celebrados ha sido el género de plataformas.

En los juegos de este género el protagonista de la aventura estaba en un escenario armado de bloques y plataformas en donde podía saltar.

Uno de los juegos más populares de esos días era Super Mario Bros.

Pilas incluye un actor llamado `Mapa` que te permite hacer juegos de plataformas fácilmente.

### 24.1 Presentando el actor `Mapa`

El `Mapa` representa un escenario compuesto de bloques que pueden ser plataformas, muros o simplemente adornos del escenario.

Para crear un `Mapa` necesitas una grilla de gráficos con los bloques que se usarán en el escenario. Luego puedes crear el `Mapa`:

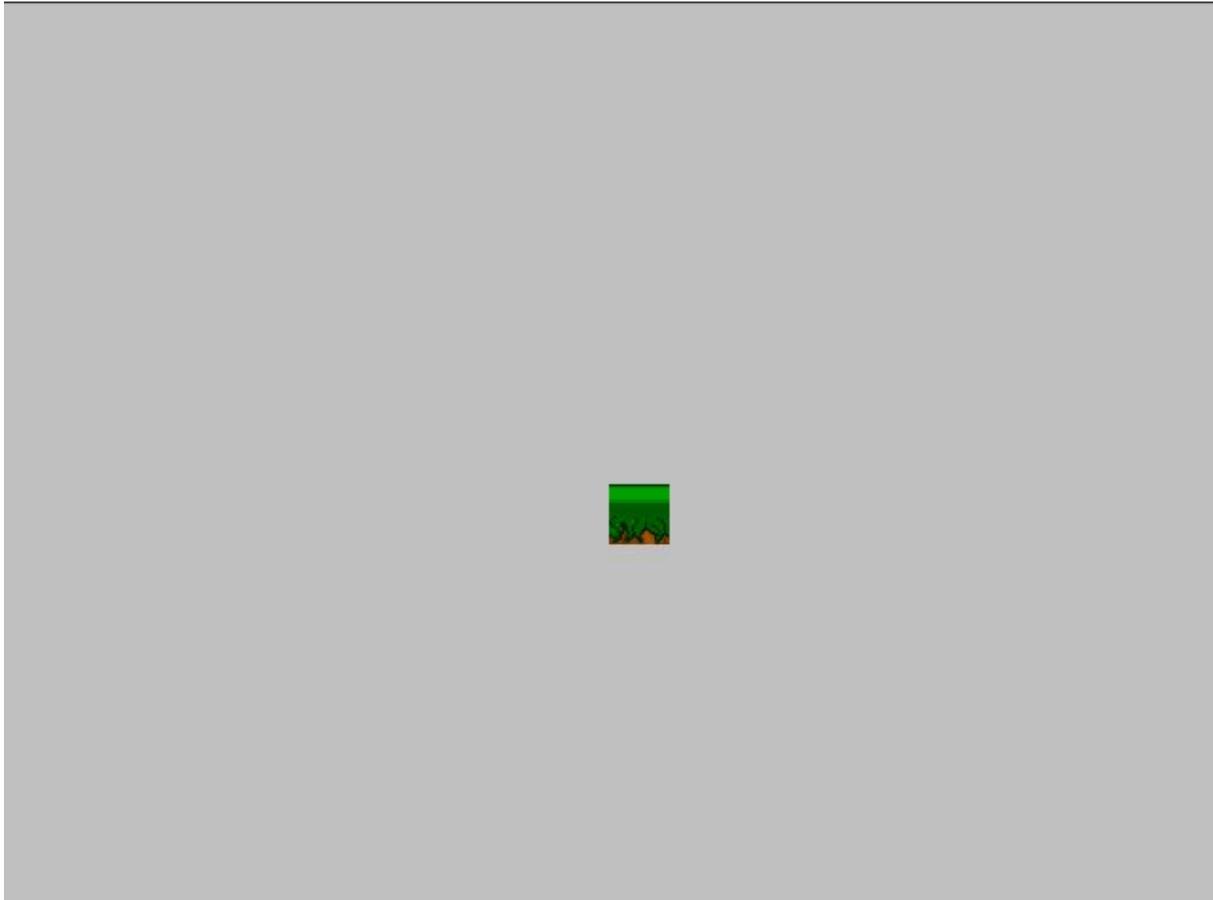
```
grilla = pilas.imagenes.cargar_grilla("grillas/plataformas_10_10.png", 10, 10)
mapa = pilas.actores.Mapa(grilla)
```

Una vez que ejecutas esas sentencias no observarás cambios en la ventana, el mapa está, pero no tiene bloques aún.

Si quieres dibujar bloques solo tienes que indicar un índice de bloque y la posición en pantalla a dibujar.

Por ejemplo, un bloque cerca del centro de la ventana es la posición (8, 10):

```
mapa.pintar_bloque(8, 10, 1)
```



Otro ejemplo: si queremos dibujar en la parte inferior de la ventana, podemos llamar muchas veces al método `pintar_bloque`, una vez por cada bloque que necesitamos:

```
for columna in range(20):  
    mapa.pintar_bloque(14, columna, 1)
```

El primer y segundo argumento del método `pintar_bloque` indica la posición en donde vamos a dibujar el bloque. En este caso la fila será 14 y la columna será 0, 1, 2, 3, 4.. etc

El tercer argumento será el índice de la grilla que indicamos anteriormente.

Este será el resultado:



## 24.2 Colisiones con el escenario

En los juegos de plataformas es muy importante que los bloques puedan interactuar con los jugadores. Por ejemplo habrá bloques que sirvan como plataformas y otros impedirán que avancemos como si se trataran de muros.

Los mapas de pilas te permiten crear esta interacción de manera sencilla. El método que usamos antes `pintar_bloque`, le dice al mapa que dibuje el bloque, pero a la vez te permite indicar si ese bloque es sólido o no.

Diremos que un bloque es sólido cuando un personaje no puede pasar a través de él. Por ejemplo, una plataforma es un bloque sólido.

Entonces, cada vez que invocas al método `pintar_bloque` tienes la posibilidad de indicar si el bloque es sólido o no:

```
mapa.pintar_bloque(14, 10, 1, es_bloque_solido=True)
mapa.pintar_bloque(14, 10, 1, es_bloque_solido=False)
```

Y ten en cuenta que si no especificas el último parámetro, pilas asumirá que el bloque debe ser sólido.

Por cierto, los bloques “no sólidos” son útiles para representar adornos del escenario, como nubes o agua. Incluso en algunos juegos se usan para crear pasadizos secretos entre muros o plataformas...

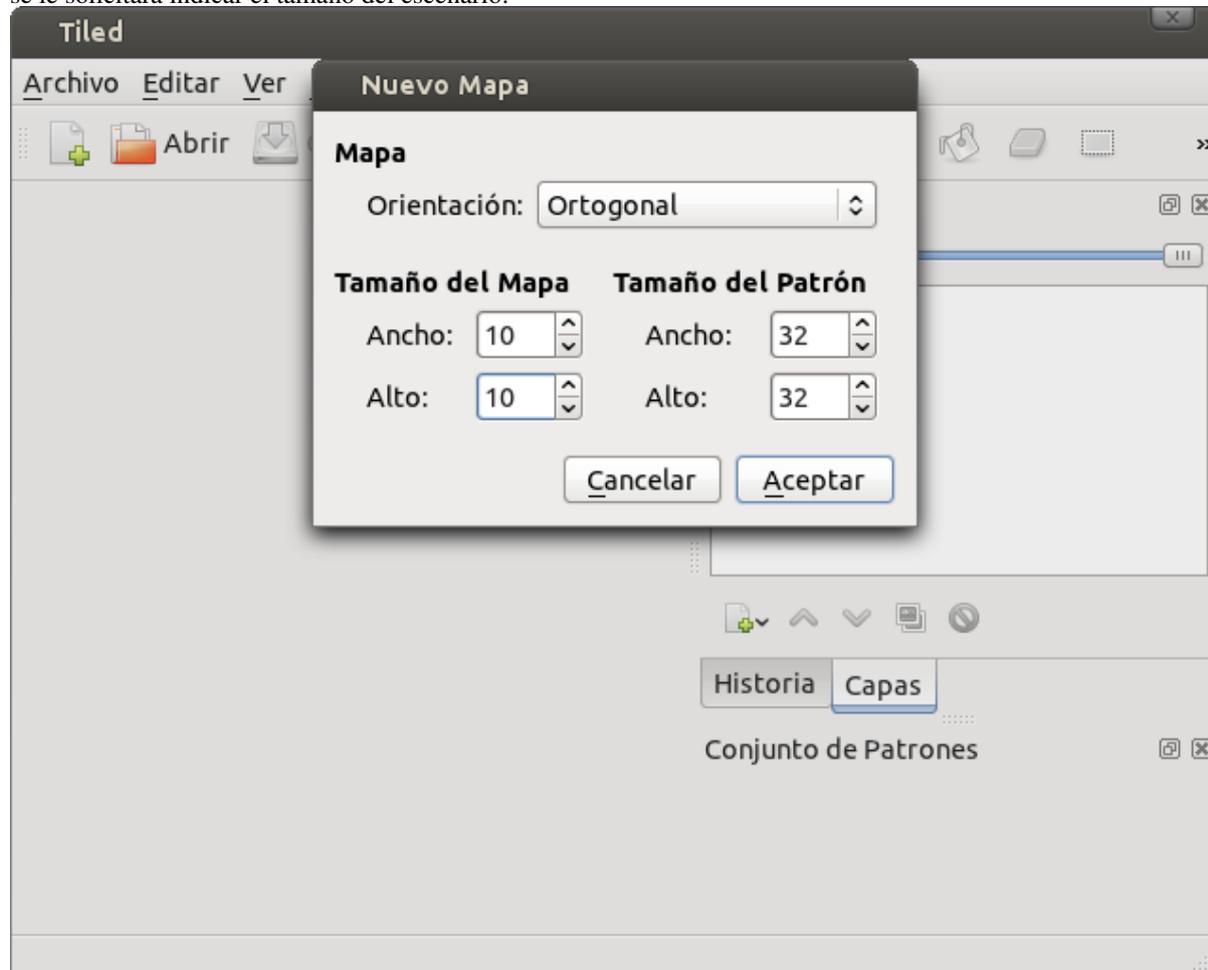
## 24.3 Creando mapas con el programa tiled

Crear los mapas directamente desde el código está bien, pero si tienes que hacer muchos mapas te llevará un montón de tiempo.

Una buena alternativa a esto es usar un software de diseño de escenarios, crear un archivo con todo el escenario y luego cargarlo desde pilas.

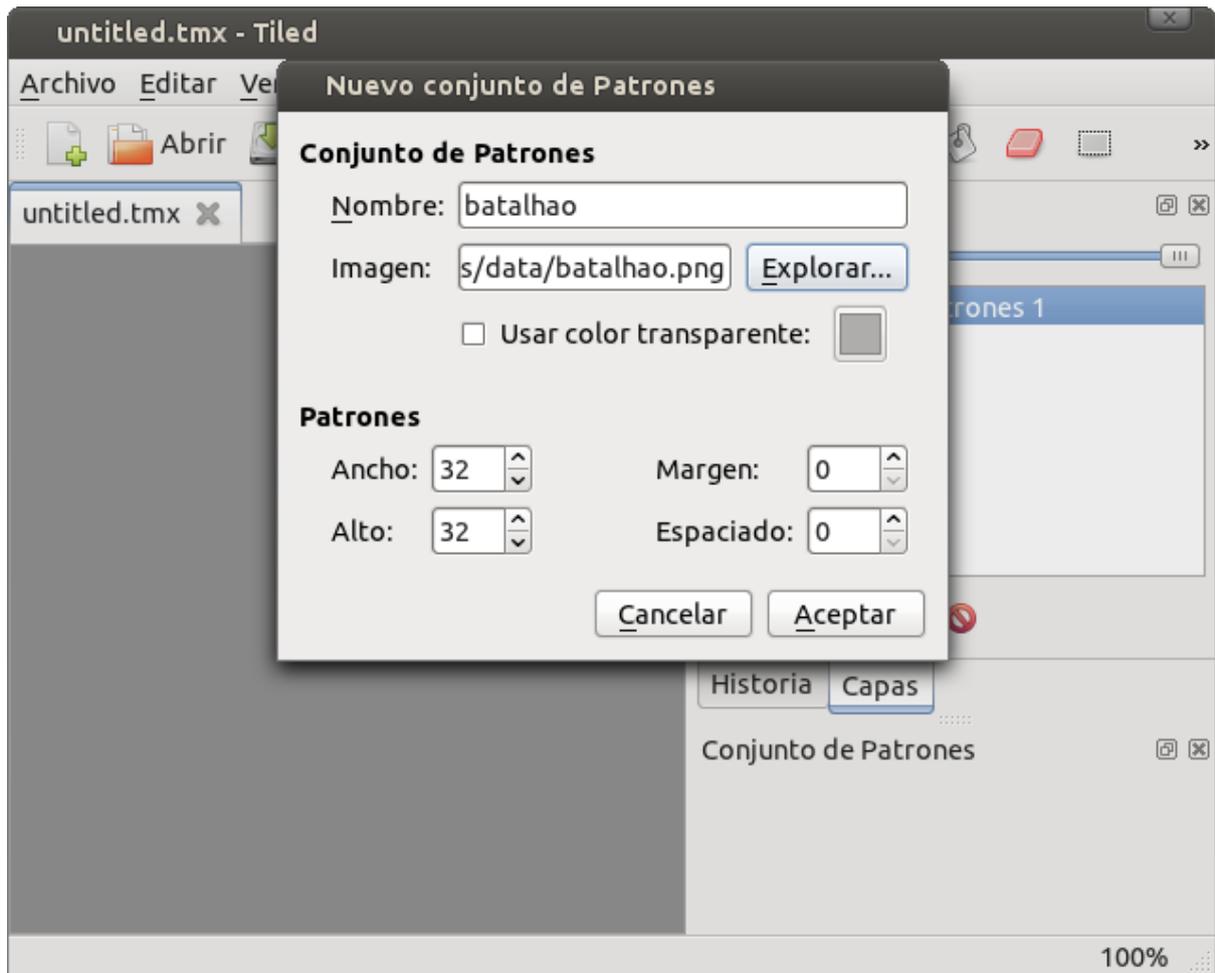
El software que te recomiendo para esta tarea se llama `tiled` (ver <http://www.mapeditor.org>).

Veamos como usar `tiled` para crear un escenario sencillo, primero tienes que crear un mapa desde el menú `File`, se le solicitará indicar el tamaño del escenario:

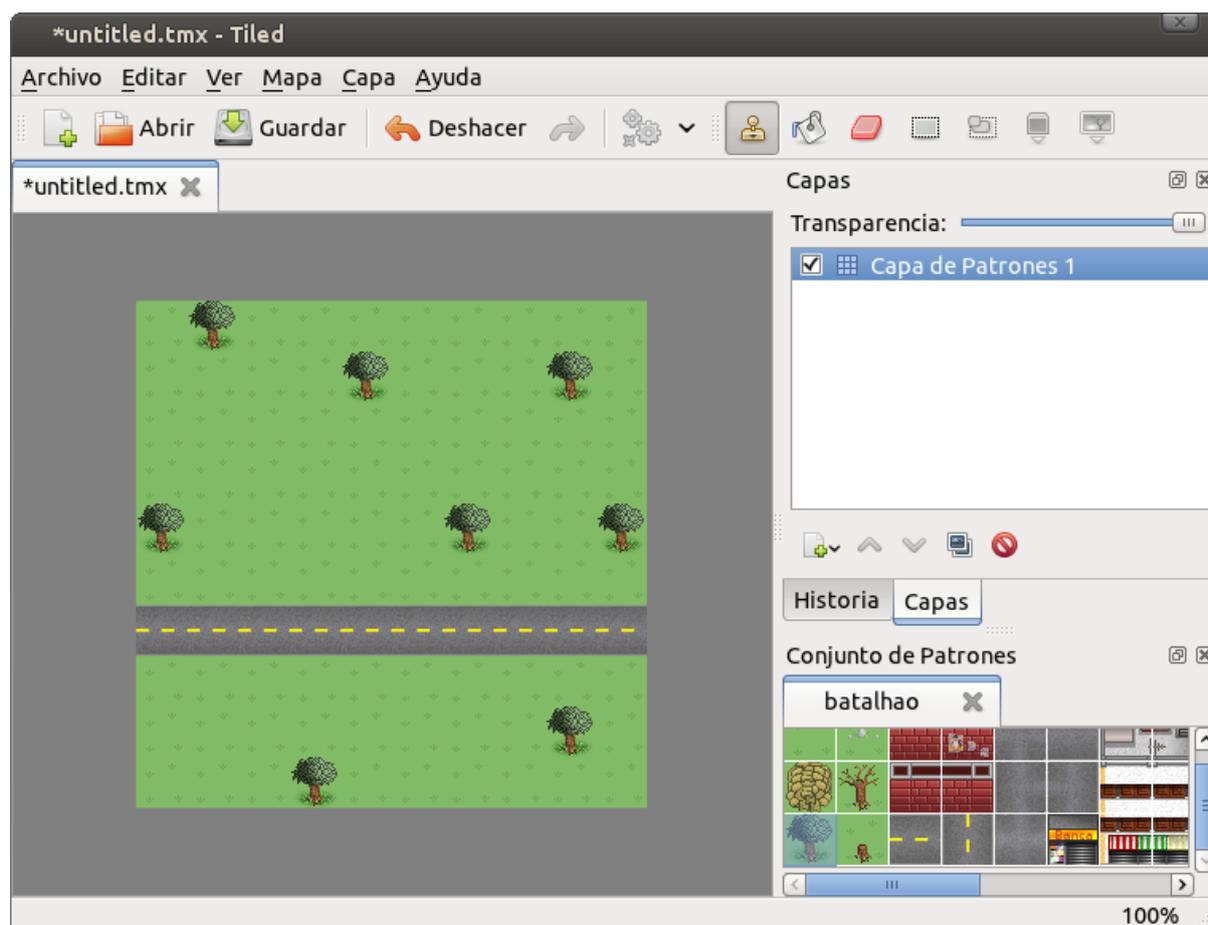


Usa los valores por defecto, al menos por esta vez.

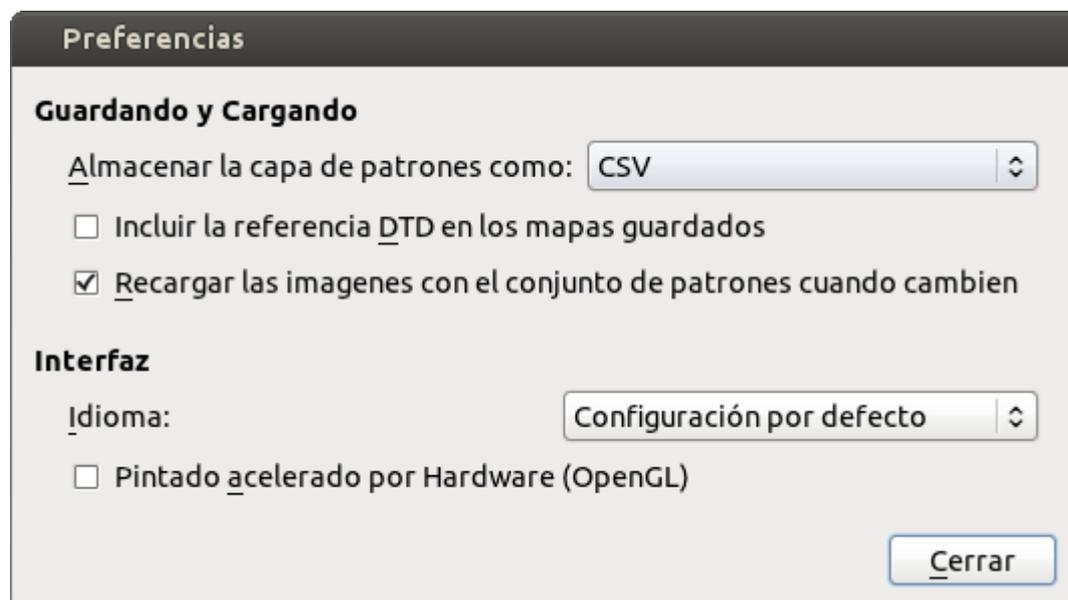
Luego tienes que ir al menú `Map` y luego `New tileset` para indicar cual es la grilla de imágenes que usarás en los bloques. Te recomiendo usar la imagen `batalhao.png` (de Silveins Neto), que está en la carpeta de ejemplos de pilas:



Ahora, lo mas divertido, comienza a dibujar sobre el escenario seleccionando bloques. Observa que el programa tiene varias herramientas para hacer que esto sea mucho mas sencillo:



Luego, asegúrate de que el programa guarda todos los datos en formato CSV, esto es importante para que se pueda vincular con pilas. Para esto tienes que abrir el menú **Edit** y luego **Preferences**, la pantalla de opciones tiene que quedar así:



Listo, ahora solamente hay que guardar el mapa en un archivo. Ve al menú **File** y luego selecciona **Save as**, tienes que darle un nombre al archivo `.tmx`.

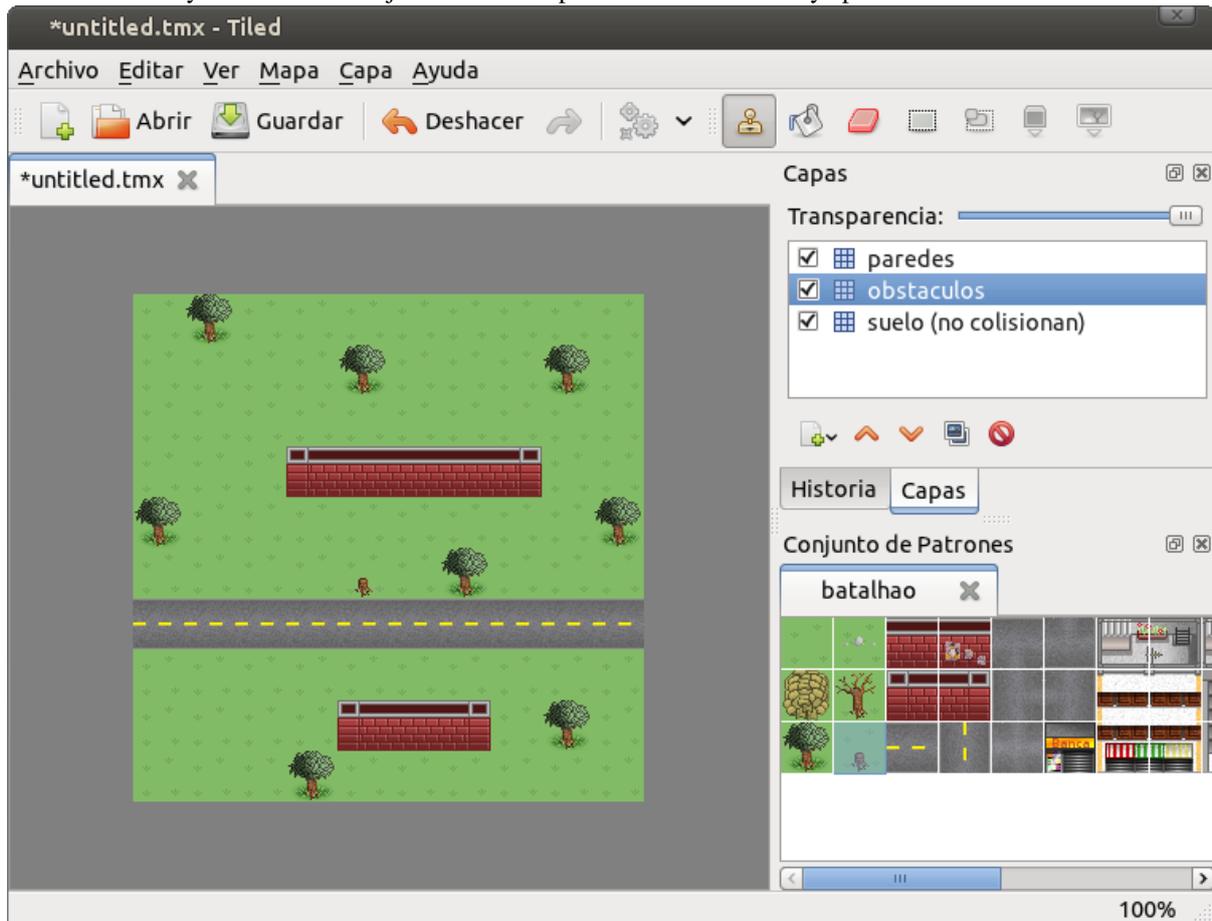
Luego, desde pilas, es muy simple, solamente tienes que crear el actor mapa indicando el nombre del archivo `.tmx` que has generado con el programa **tiled**:

```
import pilas
pilas.iniciar()
mapa_desde_archivo = pilas.actores.MapaTiled("archivo.tmx")
```

## 24.4 Creando bloques sólidos con tiled

Si quieres hacer bloques sólidos desde **tiled** solamente tienes que crear mas capas, la capa 0 se utilizará como decorado (todos los bloques son no-sólidos) y el resto de las capas serán siempre de bloques sólidos.

Por ejemplo, en el escenario anterior, sería interesante colocar los bloques de pasto y la ruta en la capa que he llamado “suelo” y el resto de los objetos en otras capas como “obstáculos” y “paredes”:



## 24.5 Un ejemplo completo

Te recomiendo que observes el ejemplo `mapa_desde_archivo.py` del directorio de ejemplos de pilas, podrás observar un escenario muy simple con obstáculos y un personaje que se puede mover con el teclado:



---

## Diálogos

---

Para contar una historia dentro de un juego podrías hacer que los personajes conversen entre sí. Esto es muy habitual en un género de videojuego llamado aventuras gráficas.

### 25.1 Mensajes de dialogo

Para hacer que un personaje emita un mensaje sencillo puedes usar el método `decir`:

```
actor = pilas.actores.Mono()  
actor.decir("Eh!, ahora puedo hablar...")
```

Esto hará que el personaje muestre un globo similar al de las historietas con las frases que has colocado.



### 25.2 Conversaciones

Los mensajes de dialogo se pueden usar para que dos o mas actores puedan conversar entre sí. Esto es útil para contar una historia, ya que le permites al usuario ir viendo paso a paso lo que se dicen los actores entre sí.

Para crear una conversación entre actores tienes que crear un objeto de la clase `Dialogo`, luego indicarle la secuencia de conversación y por último iniciar el dialogo:

```
dialogo = pilas.actores.Dialogo()  
  
dialogo.decir(mono, "Hola, como estas?")  
dialogo.decir(otro_mono, "Perfecto!!, gracias...")  
dialogo.decir(mono, "genial...")  
  
dialogo.iniciar()
```

Ahora cuando ejecutes este programa, solamente aparecerá el primer mensaje "Hola, cómo estas?" y solo cuando el usuario haga click con el mouse avanzará.

Ten en cuenta que el método `decir` funciona como una cola de mensajes, es decir, si llamas a `decir` el mensaje no aparecerá inmediatamente. El mensaje aparecerá cuando corresponda según el orden de la conversación que se siga.

Si quieres que un botón accione un mensaje y lo haga de manera inmediata tendrías que usar un método como `dialogo.decir_inmediatamente`.

## 25.3 Preguntas

Para desarrollar conversaciones con preguntas también puedes usar a los diálogos. Lo único diferente es que las preguntas traerán asociada una repuesta del usuario, y para manejar el resultado tienes que escribir una función.

La función se invocará cuando el usuario haga click en alguna de las opciones. Y cuando se llame la función se pasará la respuesta que ha elegido como una cadena de texto.

Aquí tienes un ejemplo de una pregunta con 3 respuestas. Cuando el usuario elija una respuesta el personaje volverá a decirlo:

```
def cuando_responde_color_favorito(respuesta):  
    dialogo.decir(mono, "he dicho: " + respuesta)
```

```
dialogo.elegir(mono, "Mi color favorito es el...", ["rojo", "verde", "azul"], cuando_responde_colo
```



---

## Manejo de Cámara

---

En ocasiones queremos que el escenario de nuestro juego sea muy extenso, un bosque, una ciudad repleta de objetos etc...

Nuestros juegos con pilas no están limitados a lo que podemos ver en la ventana, el espacio del escenario puede ser tan grande como queramos. Aquí es donde la `cámara` toma protagonismo.

El objeto `cámara` nos permite desplazar el punto de vista en cualquier parte del escenario, dado que nos brinda dos coordenadas: `x` e `y`, para que le indiquemos qué parte del escenario tenemos que observar.

### 26.1 Las coordenadas de la cámara

Inicialmente la cámara estará mostrando el punto  $(0, 0)$  del escenario, el punto central de la ventana.

Si queremos que muestre otra parte del escenario podemos ejecutar una sentencia como la que sigue:

```
pilas.escena_actual().camara.x = [200]
pilas.escena_actual().camara.y = [200]
```

Con esto le estaríamos diciendo a la cámara que nos muestre el punto  $(200, 200)$  del escenario. Así observaríamos que podemos explorar la parte superior derecha del escenario de forma gradual.

### 26.2 Objetos sensibles a la cámara

Hay casos en donde queremos que los actores no se desplacen junto con el escenario, es decir, puede ocurrir que necesitemos que un actor permanezca fijo en su posición de pantalla aunque la cámara cambie de lugar.

Este es el caso de los contadores de vidas, los textos que vé un usuario o cualquier marcador auxiliar.

Para que un actor no se vea afectado por la cámara, tienes que guardar el valor `True` dentro del atributo `fijo`:

```
actor.fijo = True
```

Por lo general, todos los actores tienen este atributo a `False`, porque viven en el escenario de juego y no se quedan fijos a la pantalla. Excepto los textos que siempre permanecen en la parte superior de la ventana.



---

## Motores

---

Internamente pilas delega toda la tarea de dibujar, manejo de eventos y multimedia en general a un motor llamado Qt.

Actualmente pilas soporta dos motores, y permite que los programadores puedan seleccionar el motor que mejor se adapta a los sistemas que se van a utilizar para ejecutar el juego.

Para indicarle a pilas el motor que tiene que utilizar puede usar la siguiente sentencia:

```
pilas.iniciar(usar_motor='qt')
```

es decir, solamente tienes que cambiar la inicialización de la biblioteca, el resto funcionará normalmente.

Ten en cuenta que generalmente en los tutoriales de pilas o en las presentaciones solamente llamamos a `pilas.iniciar` pero sin indicarle el motor a utilizar. Cuando no le decimos a pilas “qué” motor utilizar, pilas seleccionará a `qtgl`.



---

## Eventos, conexiones y respuestas

---

En el desarrollo de videojuegos es muy importante poder comunicarse con el usuario. Lograr que los personajes del juego puedan interactuar con él y exista una fuerte interacción.

En pilas usamos una estrategia llamada `eventos`, `conexiones` y `respuestas`, no solo porque es muy sencilla de usar, sino también porque es una solución conocida y muy utilizada en otros lugares como en la web.

### 28.1 ¿Que es un Evento?

Los eventos representan algo que esperamos que ocurra dentro de un juego, por ejemplo un `click` del mouse, la pulsación de una tecla, el cierre de la ventana o la `colisión` entre un enemigo y nuestro protagonista.

Lo interesante de los eventos, es que pueden ocurrir en cualquier momento, y generalmente no lo controlamos, solamente los escuchamos y tomamos alguna respuesta predefinida.

Pilas representa a los eventos como objetos, y nos brinda funciones para ser avisados cuando un evento ocurre e incluso emitir y generar eventos nuevos.

Veamos algunos ejemplos:

### 28.2 Conectando la emisión de eventos a funciones

Los `eventos` no disparan ninguna acción automática, nosotros los programadores somos los que tenemos que elegir los eventos importantes y elegir que hacer al respecto.

Para utilizar estas señales, tenemos que vincularlas a funciones, de forma que al emitirse la señal podamos ejecutar código.

#### 28.2.1 La función `conectar`

La función `conectar` nos permite conectar una señal de evento a un método o una función.

De esta forma, cada vez que se emita una determinada señal, se avisará a todos los objetos que hallamos conectado.

Por ejemplo, si queremos que un personaje se mueva en pantalla siguiendo la posición del puntero del mouse, tendríamos que escribir algo como esto:

```
import pilas

mono = pilas.actores.Mono()

def mover_mono_a_la_posicion_del_mouse(evento):
    mono.x = evento.x
```

```
mono.y = evento.y

pilas.eventos.mueve_mouse.conectar(mover_mono_a_la_posicion_del_mouse)

# O puedes utilizar el método abreviado del actor.
mono.mueve_mouse(mover_mono_a_la_posicion_del_mouse)
```

Es decir, la señal de evento que nos interesa es `mueve_mouse` (que se emite cada vez que el usuario mueve el mouse). Y a esta señal le conectamos la función que buscamos ejecutar cada vez que se mueva el mouse.

Ten en cuenta que pueden existir tantas funciones conectadas a una señal como quieras.

Las coordenadas que reporta el mouse son relativas al escenario y no de la ventana. Por lo tanto puedes asignar directamente el valor de las coordenadas del mouse a los actores sin efectos colaterales con respecto a la cámara.

## 28.3 Observando a los eventos para conocerlos mejor

Como puedes ver en la función `mover_mono_a_la_posicion_del_mouse`, hemos definido un parámetro llamado `evento` y accedimos a sus valores `x` e `y`.

Cada evento tiene dentro un conjunto de valores que nos resultará de utilidad conocer. En el caso del movimiento de mouse usamos `x` e `y`, pero si el evento es la pulsación de una tecla, seguramente vamos a querer saber exactamente qué tecla se pulsó.

Entonces, una forma fácil y simple de conocer el estado de un objeto es imprimir directamente su contenido, por ejemplo, en la función de arriba podíamos escribir:

```
def mover_mono_a_la_posicion_del_mouse(evento):
    print evento
```

y en la ventana de nuestra computadora tendríamos que ver algo así:

```
{'y': 2.0, 'x': -57.0, 'dx': 0.0, 'dy': -1.0}
```

donde claramente podemos ver todos los datos que vienen asociados al evento.

Por último, ten en cuenta que este argumento `evento`, en realidad, es un diccionario de python como cualquier otro, solo que puedes acceder a sus valores usando sentencias cómo `diccionario.clave` en lugar de `diccionario['clave']`.

## 28.4 Desconectando señales

Las señales se desconectan por cuenta propia cuando dejan de existir los objetos que le conectamos. En la mayoría de los casos podemos conectar señales y olvidarnos de desconectarlas, no habrá problemas, se desconectarán solas.

De todas formas, puede que quieras conectar una señal, y por algún motivo desconectarla. Por ejemplo si el juego cambia de estado o algo así...

Si ese es tu caso, simplemente asígnale un identificador único al manejador de la señal y luego usa la función `desconectar_por_id` indicando el identificador.

Por ejemplo, las siguientes sentencias muestran eso:

```
pilas.eventos.mueve_mouse.conectar(imprimir_posicion, id='drag')
pilas.eventos.mueve_mouse.desconectar_por_id('drag')
```

En la primera sentencia conecté la señal del evento a una función y le di un valor al argumento `id`. Este valor será el identificador de ese enlace. Y en la siguiente línea se utilizó el identificador para desconectarla.

## 28.5 Consultado señales conectadas

Durante el desarrollo es útil poder observar qué eventos se han conectado a funciones.

Una forma de observar la conexión de los eventos es pulsar la tecla F6. Eso imprimirá sobre consola los nombres de las señales conectadas junto a las funciones.

## 28.6 Creando tus propios eventos

Si tu juego se vuelve mas complejo y hay interacciones entre varios actores, puede ser una buena idea hacer que exista algo de comunicación entre ellos usando eventos.

Veamos cómo crear un evento:

Primero tienes que crear un objeto que represente a tu evento y darle un nombre:

```
evento = pilas.evento.Evento("Nombre")
```

luego, este nuevo objeto `evento` podrá ser utilizado como canal de comunicación: muchos actores podrán conectarse para recibir alertas y otros podrán emitir alertas:

```
def ha_ocurrido_un_evento(datos_evento):
    print "Hola!!!", datos_evento
```

```
evento.conectar(ha_ocurrido_un_evento)
```

```
# En otra parte...
```

```
evento.emitir(argumento1=123, argumento2=123)
```

Cuando se emite un evento se pueden pasar muchos argumentos, tantos como se quiera. Todos estos argumentos llegarán a la función de respuesta en forma de diccionario.

Por ejemplo, para este caso, cuando llamamos al método `evento.emitir`, el sistema de eventos irá automáticamente a ejecutar la función `ha_ocurrido_un_evento` y ésta imprimirá:

```
Hola!!! {argumento1: 123, argumento2: 123}
```

---

**Note:** Para entender mejor cómo se han implementado los eventos, visita este link <http://hugoruscitti.github.com/2012/03/01/rediseñando-el-sistema-de-eventos-pilas/>

---

## 28.7 Referencias

El concepto que hemos visto en esta sección se utiliza en muchos sistemas. Tal vez el mas conocido de estos es la biblioteca GTK, que se utiliza actualmente para construir el escritorio GNOME y Gimp entre otras aplicaciones.

El sistema de señales que se utiliza en pilas es una adaptación del siguiente sistema de eventos:

<http://stackoverflow.com/questions/1092531/event-system-in-python>

Anteriormente usábamos parte del código del sistema `django`, pero luego de varios meses lo reescribimos para que sea mas sencillo de utilizar y no tenga efectos colaterales con los métodos y el módulo `weakref`.

Si quieres obtener mas información sobre otros sistemas de eventos te recomendamos los siguientes documentos:

- <http://pydispatcher.sourceforge.net/>
- <http://www.mercurytide.co.uk/news/article/django-signals/>
- <http://www.boduch.ca/2009/06/sending-django-dispatch-signals.html>
- <http://docs.djangoproject.com/en/dev/topics/signals/>



---

## Textos

---

Los objetos que muestran texto en pantalla se tratan de manera similar a los actores. Por lo tanto, si ya sabes usar actores, no tendrás problemas en usar cadenas de texto.

### 29.1 Crear cadenas de texto

El objeto que representa texto se llama `Texto` y está dentro del módulo `actores`.

Para crear un mensaje tienes que escribir:

```
texto = pilas.actores.Texto("Hola, este es mi primer texto.")
```

y tu cadena de texto aparecerá en pantalla en color negro y con un tamaño predeterminado:



Hola, este es mi primer texto.

Si quieres puedes escribir texto de varias líneas usando el carácter `\n` para indicar el salto de línea.

Por ejemplo, el siguiente código escribe el mismo mensaje de arriba pero en dos líneas:

```
texto = pilas.actores.Texto("Hola (en la primer linea)\n este es mi primer texto.")
```

## 29.2 Los textos son actores

Al principio comenté que los textos también son actores, esto significa que casi todo lo que puedes hacer con un actor aquí también funciona, por ejemplo:

```
texto.x = 100
texto.escala = 2
```

incluso también funcionarán las interpolaciones:

```
texto.rotacion = pilas.interpolar(360)
```

## 29.3 Propiedades exclusivas de los textos

Existen varias propiedades que te permitirán alterar la apariencia de los textos.

Esta es una lista de los mas importantes.

- color
- magnitud
- texto

Por ejemplo, para alterar el texto, color y tamaño de un texto podría escribir algo así:

```
texto.magnitud = 50
texto.color = (0, 0, 0) # Color negro
texto.color = (255, 0, 0, 128) # Color rojo, semi transparente.
texto.texto = "Hola, este texto \n tiene 2 lineas separadas"
```

## 29.4 Mostrando mensajes en pantalla

Si bien podemos crear actores `Texto` y manipularlos como quedarnos, hay una forma aún mas sencilla de imprimir mensajes para los usuarios.

Existe una función llamada `avisar` que nos permite mostrar en pantalla un texto pequeño en la esquina izquierda inferior de la pantalla.

```
pilas.avisar("Hola, esto es un mensaje.")
```

Esto te facilitará mucho el código en los programas pequeños como demostraciones o ejemplos.

---

## Habilidades

---

Pilas permite añadir funcionalidad a tus objetos de manera sencilla, dado que usamos el concepto de habilidades, un enfoque similar a la programación orientada a componentes <sup>1</sup> y mixins <sup>2</sup>.

### 30.1 Un ejemplo

Una habilidad es una funcionalidad que está implementada en alguna clase, y que si quieres la puedes vincular a un actor cualquiera.

Veamos un ejemplo, imagina que tienes un actor en tu escena y quieres que la rueda del mouse te permita cambiarle el tamaño.

Puedes usar la habilidad `AumentarConRueda` y vincularla al actor fácilmente.

El siguiente código hace eso:

```
import pilas

mono = pilas.actores.Mono()
mono.aprender(pilas.habilidades.AumentarConRueda)
```

así, cuando uses la rueda del mouse el tamaño del personaje aumentará o disminuirá.

Nota que aquí usamos la metáfora de “aprender habilidades”, porque las habilidades son algo que duran para toda la vida del actor.

### 30.2 Un ejemplo mas: hacer que un actor sea arrastrable por el mouse

Algo muy común en los juegos es que puedas tomar piezas con el mouse y moverlas por la pantalla.

Esta habilidad llamada `Arrastrable` representa eso, puedes vincularlo a cualquier actor y simplemente funciona:

```
import pilas

mono = pilas.actores.Mono()
mono.aprender(pilas.habilidades.Arrastrable)
```

---

<sup>1</sup> [http://es.wikipedia.org/wiki/Programación\\_orientada\\_a\\_componentes](http://es.wikipedia.org/wiki/Programación_orientada_a_componentes)

<sup>2</sup> <http://es.wikipedia.org/wiki/Mixin>

## 30.3 Otro ejemplo: un actor que cambia de posición

Veamos otro ejemplo sencillo, si queremos que un actor se coloque en la posición del mouse cada vez que hacemos click, podemos usar la habilidad: `SeguirClicks`.

```
import pilas

mono = pilas.actores.Mono()
mono.aprender(pilas.habilidades.SeguirClicks)
```

## 30.4 Mezclar habilidades

En pilas se ha intentado hacer que las habilidades sean lo mas independientes posibles, porque claramente lo mas divertido de este enfoque es poder combinar distintas habilidades para lograr comportamientos complejos.

Así que te invitamos a que pruebes y experimentes mezclando habilidades.

## 30.5 Otras habilidades para investigar

Pilas viene con varias habilidades incluidas, pero lamentablemente este manual no las menciona a todas. Así que te recomendamos abrir un intérprete de python y consultarle directamente a él que habilidades tienes disponibles en tu versión de pilas.

Para esto, abre un terminal de python y escribe lo siguiente:

```
import pilas
pilas.iniciar()
dir(pilas.habilidades)
```

esto imprimirá en pantalla todas las habilidades como una lista de cadenas.

## 30.6 ¿Cómo funcionan las habilidades?

Las habilidades son clases normales de python, solo que se han diseñado para representar funcionalidad y no entidades.

La vinculación con los actores se produce usando herencia múltiple, una de las virtudes de python.

Así que internamente lo que sucede cuando ejecutas una sentencia como:

```
actor.aprender(pilas.habilidades.HabilidadDeEjemplo)
```

es que la instancia de la clase actor pasa a tener una superclase adicional, llamada `HabilidadDeEjemplo`.

A diferencia de la programación orientada a objetos clásica, en pilas los objetos no guardan una estrecha relación con una jerarquía de clases. Por el contrario, los objetos se combinan a conveniencia, y cada clase intenta tener solamente la mínima funcionalidad que se necesita.

Esta idea de combinación de objetos la hemos adoptado de la programación orientada a componentes. Por lo que puedes investigar en la red para conocer mas acerca de ello.

## 30.7 ¿Ideas?

Si encuentras habilidades interesantes para desarrollar te invitamos compartir tus ideas con las personas que hacemos pilas y estamos en el foro de losersjuegos<sup>3</sup>.

---

<sup>3</sup> <http://www.losersjuegos.com.ar/foro>



---

## Depurando y buscando detalles

---

Pilas incluye varios modos de ejecución que te pueden resultar de utilidad para ver en detalle el funcionamiento de tu juego.

La depuración dentro de la programación de juegos permite detectar errores, corregir detalles e incluso comprender algunas interacciones complejas.

### 31.1 Modo pausa y manejo de tiempo

Si pulsas las teclas `ALT + P` durante la ejecución de pilas, el juego completo se detiene. En ese momento puedes pulsar cualquier tecla para avanzar un instante de la simulación o la tecla flecha derecha para avanzar más rápidamente.

Esto es muy útil cuando trabajas con colisiones físicas, porque este modo de pausa y manejo de tiempo te permite ver en detalle la interacción de los objetos y detectar cualquier inconveniente rápidamente.

### 31.2 Modos depuración

Las teclas **F6**, **F7**, **F8**, **F9**, **F10**, **F11** y **F12** te permiten hacer visibles los modos de depuración.

Cada modo representa un aspecto interno del juego que podrías ver. Por ejemplo, el modo que se activa con la tecla **F12** te permite ver la posición exacta de cada actor, mientras que al tecla **F11** te permite ver las figuras físicas.

### 31.3 Activar modos desde código

Si quieres que el juego inicie alguno de los modos, puedes usar la función `pilas.atajos.definir_modos`. Por ejemplo, para habilitar el modo depuración física podrías escribir:

```
pilas.atajos.definir_modos(fisica=True)
```

esta función tiene varios argumentos opcionales, como `posicion`, `radios` etc. Mira la definición de la función para obtener más detalles:

### 31.4 Activando los modos para detectar errores

Ten en cuenta que puedes activar los modos depuración en cualquier momento, incluso en medio de una pausa, ir del modo depuración al modo pausa y al revés. Los dos modos se pueden combinar fácilmente.

Mi recomendación es que ante la menor duda, pulses `alt + p` para ir al modo pausa, y luego comiences a pulsar alguna de las teclas para activar los modos depuración y observar en detalle qué está ocurriendo: **F6**, **F7** etc.



---

## Integrando Pilas a una Aplicación Qt

---

En esta sección vamos a mostrar como integrar Pilas como un widget dentro de tu aplicación desarrollada con *PyQt*.

**Nota:** En este capítulo asumimos que el programador ya conoce *PyQt* y *Pilas*.

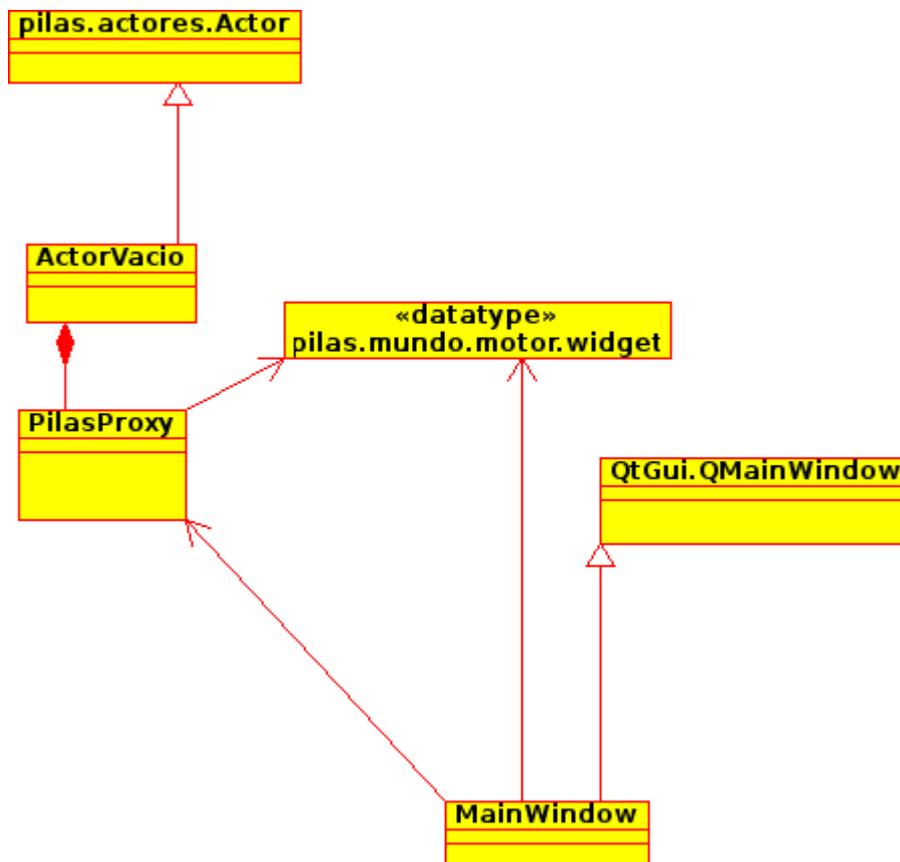
Antes de empezar vamos a establecer algunos objetivos:

- Trataremos que la programación de la parte *Pilas* sea lo mas *Pilas-Like*.
- *Pilas* nos brinda un solo widget; por el objetivo anterior intentaremos mantener esto.
- La programación de la parte *PyQt* trataremos que se mantenga lo mas *PyQt-Like*.

Con esto en mente vamos a proponernos un proyecto:

Desarrollaremos una aplicación *PyQt* que muestre algunos actores en pantalla y al hacerle click sobre alguno nos permita seleccionar una imagen desde un archivo para reemplazar a la del actor.

La estructura de objetos que manejaremos sera la siguiente:



Donde el objetivo de cada clase es el siguiente:

- `MainWindow`: Es un widget PyQt4 que hereda de `PyQt4.QtGui.QMainWindow`. Se encargara de recibir el evento de cuando un `ActorVacio` fue “clickeado” y mostrara la ventana emergente para seleccionar la imagen que luego sera asignada en el actor que lanzo el evento.
- `PilasProxy`: Esta clase es un *singleton* que cada vez que es destruida *finje* su destrucción y solo limpia el widget principal de Pilas, para que cuando sea reutilizada, parezca que esta como nueva. Tendrá 3 métodos/propiedades importantes implementara:
  - `widget`: Propiedad que referencia al widget principal de Pilas.
  - `__getattr__`: Método que delegara todas las llamadas que no posea el proxy al widget principal de Pilas.
  - `destroy`: Método que ocultara la implementación de `destroy` del widget principal de Pilas.
  - `actor_clickeado`: *evento* de pilas que enviara como parámetro el actor que fue clickeado.
  - **agregar\_actor**: permitirá agregar un actor al proxy y conectará las señales del actor con la señal del proxy.
  - `borrar_actor`: borra un actor de los manejados por el proxy
- `ActorVacio`: Subclase de `pilas.actores.Actor` que emitirá un evento al ser clickeada sobre si misma.

## 32.1 Código

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

#=====
# IMPORTS
#=====

import sys
import os
import random

# importamos todos los modulos necesarios de PyQt
from PyQt4 import QtGui, QtCore

#=====
# CONFIGURACIÓN INICIAL
#=====

# antes de importar pilas creamos la app QT del programa
# si tenemos los componentes pilas en otro modulo puede llegar a ser conveniente
# importar ese modulo (el que usa pilas) dentro de un metodo de clase o una
# funcion. Tambien para que phonon no se queje es bueno setearle una nombre a
# nuestro QApplication
app = QtGui.QApplication(sys.argv[1:])
app.setApplicationName(__name__)

# Importamos pilas con un motor que sirva para embeber
# 'qt' y 'qtgl' crean y auto-arrancan la aplicacion
# mientras que 'qtsugar' y 'qtsugargl' solo crean
# los widget necesarios para embeber pilas
import pilas
pilas.iniciar(usar_motor="qtsugar")
```

```

#=====
# CLASE ACTOR
#=====

# nuestra clase actor
class ActorVacio(pilas.actores.Actor):

    def __init__(self, *args, **kwargs):
        super(ActorVacio, self).__init__(*args, **kwargs)

        # El evento que emitiremos cuando clickean al actor
        self.me_clickearon = pilas.evento.Evento("me_clickearon")

        # Conectamos el evento genérico de click del mouse con un
        # validador que se encargara de determinar si el click
        # sucedió sobre el actor
        pilas.eventos.click_de_mouse.conectar(self._validar_click)

    def _validar_click(self, evt):
        # extraemos las coordenadas donde sucedió el click
        x, y = evt["x"], evt["y"]

        # vemos si el actor colisiona con el punto donde
        # se hizo click y de ser asi se lanza el evento
        # me_clickearon pasando como parámetro al mismo
        # actor
        if self.colisiona_con_un_punto(x, y):
            self.me_clickearon.emitir(actor=self)

#=====
# PROXY CONTRA PILAS
#=====

class PilasProxy(object):

    # esta variable de clase guardara la única instancia que genera esta clase.
    _instance = None

    # redefinimos __new__ para que solo haya una instancia de pilas proxy
    @staticmethod
    def __new__(cls, *args, **kwargs):
        if not PilasProxy._instance:
            PilasProxy._instance = super(PilasProxy, cls).__new__(cls, *args, **kwargs)
        return PilasProxy._instance

    def __init__(self):
        self._actores = set() # aca almacenaremos todos los actores
        self.click_en_actor = pilas.evento.Evento("click_en_actor")

    def __getattr__(self, k):
        # todo lo que no pueda resolver la clase se lo delega al widget.
        # Con esto el proxy puede ser usado transparentemente
        return getattr(self.widget, k)

    def agregar_actor(self, actor):
        # Validamos que el actor sea un ActorVacio
        assert isinstance(actor, ActorVacio)

        # conectamos la señal del actor con la señal del proxy
        actor.me_clickearon.conectar(

```

```
        self._clickearon_actor
    )

    # agregamos el actor a la coleccion de actores
    self._actores.add(actor)

def _clickearon_actor(self, evt):
    # método que recibe a que actor clickearon y emite la señal
    # de que clickearon al actor desde el proxy
    self.click_en_actor.emitir(**evt)

def borrar_actor(self, actor):
    if actor in self._actores:
        # si el actor exist en los manejados por el proxy
        # desconectamos las señales y destruimos el actor
        actor.me_clickearon.desconectar(self.click_en_actor)
        self._actores.remove(actor)
        actor.destruir()

# prevenimos que al ejecutarse destroy sobre el widget subyacente
def destroy(self):
    self.widget.setParent(None)
    for act in self._actores:
        self.borrar_actor(act)

@property
def widget(self):
    return pilas.mundo.motor.ventana

#=====
# VENTANA PRINCIPAL
#=====

class MainWindow(QtGui.QMainWindow):

    def __init__(self):
        super(QtGui.QMainWindow, self).__init__()
        self.pilas = PilasProxy() # traemos nuestro proxy
        self.setCentralWidget(self.pilas.widget) # lo agregamos a la ventana
        self.resize(self.pilas.widget.size())

        # creamos entre 5 y 10 actores
        actores = ActorVacio() * random.randint(5,10)
        for a in actores:
            self.pilas.agregar_actor(a)

        # conectamos el evento click en el actor
        self.pilas.click_en_actor.conectar(self.on_actor_clickeado)

    def on_actor_clickeado(self, evt):
        # este slot va a abrir el selector de archivos de imagen
        # y asignar esa imagen al actor que llego como parametro
        actor = evt["actor"]
        filename = QtGui.QFileDialog.getOpenFileName(
            self, self.tr("Imagen de Actor"),
            os.path.expanduser("~"),
            self.tr("Imágenes (*.png *.jpg)")
        )
        if filename:
            actor.imagen = pilas.imagenes.cargar_imagen(
                unicode(filename))
```

```

)

#=====
# PONEMOS A CORRER TODO
#=====

win = MainWindow()
win.show()
sys.exit(app.exec_())

```

## 32.2 Resultado



Si quieren ver en video: <http://www.youtube.com/watch?v=DA1DFTHJ-rE&feature=youtu.be>

Referencia completa (API):



---

## Referencia completa

---

Esta sección resume todas las funciones, métodos y clases que encontrarás en pilas.

Es una buena referencia a la hora explorar en profundidad alguna característica de la biblioteca y conocer las posibilidades que ofrece.



---

**Módulo pilas.habilidades**

---

Avanzado:



---

## Guía para desarrolladores

---

En esta sección veremos como contribuir en el desarrollo de pilas, mostrando las herramientas de desarrollo y dando algunas recomendaciones.

Actualmente utilizamos Git junto a los servicios de [github](#).

### 35.1 Repositorio

Para contribuir en el desarrollo de pilas necesitas una cuenta de usuario en [github](#), nuestros proveedores del servicio de repositorios.

La dirección de acceso web al repositorio es:

- <http://github.com/hugoruscitti/pilas>

Ten en cuenta que el servicio [github](#) es gratuito, y solo lleva unos minutos registrarse.

### 35.2 Obteniendo la última versión del reposito

Para obtener la última versión tienes que ejecutar el siguiente comando desde un terminal:

```
git clone http://github.com/hugoruscitti/pilas
```

Luego aparecerá un directorio llamado `pilas`, con el contenido completo del repositorio.

### 35.3 Primer prueba

Ingresa en el directorio `pilas`, ejecuta el comando:

```
python bin/pilas
```

debería aparecer en pantalla el asistente de primer inicio.

### 35.4 Instalación en modo desarrollo

Si sos desarrollador, la forma de instalación mas recomendable es mediante el comando `develop`. Esta opción es útil porque te permite mantener actualizada la biblioteca en todo momento.

Para usar esta opción de instalación tienes que ejecutar el siguiente comando:

```
sudo python setup.py develop
```

Ten en cuenta que a partir de ahora, cuando uses `pilas` en el sistema, se leerá el código directamente desde ese directorio en donde has clonado la biblioteca.

## 35.5 Mantenerse actualizado, siempre...

Dado que `pilas` está creciendo, es una buena idea mantener tu copia del motor actualizada.

Para ello tienes que ingresar en el directorio `pilas` y ejecutar el siguiente comando de manera periódica:

```
git pull
```

## 35.6 Mas detalles

Usamos el modelo de trabajo de `github`, haciendo `forks` y `pull requests`.

Si quieres obtener mas detalles te recomiendo ver el siguiente artículo:

- <http://www.cocoanetics.com/2012/01/github-fork-fix-pull-request/>

## 35.7 Referencias para desarrolladores

---

## Guía de preguntas avanzadas

---

### 36.1 Obtengo errores en ingles al iniciar pilas ¿Que anda mal?

Si al ejecutar pilas, ves un mensaje cómo el siguiente:

```
X Error: RenderBadPicture (invalid Picture parameter) 163
Extension: 149 (RENDER)
Minor opcode: 8 (RenderComposite)
Resource id: 0x4a0000e
```

Es muy probable que se deba al adaptador de video. Una forma de solucionarlo es cambiar la linea de código:

```
pilas.iniciar()
```

por:

```
pilas.iniciar(usar_motor='qt')
```

El motivo de este problema, es que pilas usa una biblioteca llamada OpenGL, y algunos equipos no lo tienen disponible o con algunos detalles de configuración.

### 36.2 ¿Que es OpenGL?, ¿Cómo se configura en mi equipo?

OpenGL es una biblioteca que usamos en pilas para que los gráficos sean mucho mas rápidos y fluidos. OpenGL utiliza aceleración de hardware y rutinas de optimización avanzadas.

El punto es, que tal vez tu equipo no lo soporte, o no esté correctamente configurado.

Para saber si tu equipo tiene soporte para opengl, es conveniente que ejecutes el comando:

```
glxinfo | grep rende
```

Si tu equipo tiene soporte para opengl, tendrías que ver un mensaje cómo el siguiente:

```
direct rendering: Yes
OpenGL renderer string: Quadro FX 570/PCI/SSE2
```

Luego, si no tienes soporte, puedes ejecutar el siguiente comando y volver a intentar:

```
sudo apt-get install freeglut3 freeglut3-dev
```

### 36.3 Obtengo errores de AttributeError por parte de pilas

El funcionamiento de pilas como módulo de python es un poquito especial... porque sentencias como `pilas.actores` no funcionarán a menos que antes escribas `pilas.iniciar()`.

Por lo tanto, te recomiendo que en tus programas siempre comiences con un archivo que tenga algo como esto:

```
import pilas
pilas.iniciar()
```

es decir, tu programa principal tiene que importar pilas y luego inicializarlo. Recién ahí podrás usar el resto de los módulos de pilas.

## 36.4 ¿Cómo puedo personalizar el dibujado de un actor?

Cada vez que se actualiza el bucle de juego se llama al método `dibujar` de cada actor.

Si quieres personalizar por completo la forma en que se dibuja un actor puedes redefinir el método `dibujar` y listo.

Para mas referencias puedes ver el método `dibujar` de la clase `Actor` o el método `dibujar` de la clase `escena.Normal`, que en lugar de pintar una imagen borra todo el fondo de pantalla.

## 36.5 ¿A veces los sonidos no se reproducen?

sip... a veces los sonidos no se reproducen porque python los libera de memoria mientras están sonando.

Entonces, para solucionar el problema tienes que mantener viva la referencia al objeto `Sonido` cuando quieras reproducir algo. Por ejemplo:

### Ejemplo incompleto

```
def reproducir_sonido():
    mi_sonido_que_no_suena = pilas.sonidos.cargar("sonido.wav.")
    mi_sonido_que_no_suena.reproducir()

reproducir_sonido()
```

### Ejemplo correcto

```
sonido = None

def reproducir_sonido():
    sonido = pilas.sonidos.cargar("sonido.wav")
    sonido.reproducir()

reproducir_sonido()
```

¿Cual es la diferencia?, en el primer ejemplo el sonido no se reproducirá porque la referencia `mi_sonido_que_no_suena` se eliminará cuando termine de ejecutar la función `reproducir_sonido`, mientras que en el segundo la referencia `sonido` seguirá existiendo mientras el sonido esté reproduciéndose.

## 36.6 Como desinstalo una versión vieja de pilas

Pilas se puede desinstalar directamente borrando el cargador e instalando una versión nueva.

Si has instalado pilas en un sistema linux, también podrías desinstalar pilas ubicando el directorio de instalación y borrándolo.

Por ejemplo, con el siguiente comando podemos conocer el directorio de instalación:

```
sudo easy_install -m pilas
```

En pantalla tendría que aparecer un mensaje cómo:

Using `/usr/lib/python2.7/dist-packages`

Este mensaje significa que pilas se buscará dentro de ese contenedor de directorio. Este directorio puede ser distinto en tu sistema.

En mi caso, como el directorio es `/usr/lib/python2.7/dist-packages`, para desinstalar pilas puedo borrar el directorio `pilas` dentro del directorio anterior:

```
rm -r -f /usr/lib/python2.7/dist-packages/pilas
```

(IMPORTANTE: puede variar en tu sistema)

## 36.7 Tengo una consulta puntual, ¿quien me puede ayudar?

Tenemos un foro de mensajes en donde puedes preguntar lo que quieras sobre pilas, esta es la dirección web:

<http://www.losersjuegos.com.ar/foro/viewforum.php?f=22>



---

## Cómo funciona pilas por dentro

---

Pilas es un proyecto con una arquitectura de objetos grande. Tiene mucha funcionalidad, incluye un motor de física, muchos personaje pre-diseñados, eventos, escenas y un enlace al motor multimedia Qt.

Mediante este capítulo quisiera explicar a grandes rasgos los componentes de pilas. Cómo están estructurados los módulos, y qué hacen las clases mas importantes.

El objetivo es orientar a los programadores mas avanzados para que puedan investigar pilas por dentro.

### 37.1 Filosofía de desarrollo

Pilas es un proyecto de software libre, orientado a facilitar el desarrollo de videojuegos a personas que generalmente no hacen juegos... Por ese motivo que gran parte de las decisiones de desarrollo se tomaron reflexionando sobre cómo diseñar una interfaz de programación simple y fácil de utilizar.

Un ejemplo de ello, es que elegimos el lenguaje de programación python, y tratamos de aprovechar al máximo su modo interactivo.

### 37.2 API en español

Dado que pilas está orientado a principiantes, docentes y programadores de habla hispana. Preferimos hacer el motor en español, permitirle a los mas chicos usar su idioma para hacer juegos es alentador, tanto para ellos que observan que el idioma no es una barrera, como para los que enseñamos y queremos entusiasmar.

Esta es una decisión de diseño importante, porque al mismo tiempo que incluye a muchas personas, no coincide con lo que acostumbran muchos programadores (escribir en inglés).

Posiblemente en el futuro podamos ofrecer una versión de pilas alternativa en inglés, pero actualmente no es una prioridad.

### 37.3 Bibliotecas que usa pilas

Hay tres grandes bibliotecas que se utilizan dentro de pilas:

- Box2D
- Qt4



Box2D se utiliza cómo motor de física, mientras que Qt es un motor multimedia utilizado para dibujar, reproducir sonidos y manejar eventos.

## 37.4 Objetos y módulos

Pilas incluye muchos objetos y es un sistema complejo. Pero hay una forma sencilla de abordarlo, porque hay solamente 3 componentes que son indispensables, y han sido los pilares desde las primeras versiones de pilas hasta la fecha:

- Mundo
- Actor
- Motor

Si puedes comprender el rol y las características de estos 3 componentes el resto del motor es mas fácil de analizar.

Veamos los 3 componentes rápidamente:

Mundo es un objeto `singleton`, hay una sola instancia de esta clase en todo el sistema y se encarga de mantener el juego en funcionamiento e interactuando con el usuario.

Los actores (clase `Actor`) representan a los personajes de los juegos, la clase se encarga de representar todos sus atributos como la posición y comportamiento como “dibujarse en la ventana”. Si has usado otras herramientas para hacer juegos, habrás notado que se los denomina `Sprites`.

Luego, el `Motor`, permite que pilas sea un motor multimedia portable y multiplataforma. Básicamente pilas delega la tarea de dibujar, emitir sonidos y controlar eventos a una biblioteca externa. Actualmente esa biblioteca es Qt, pero en versiones anteriores ha sido implementada en `pygame` y `sfml`.

Ahora que lo he mencionado, veamos con un poco mas de profundidad lo que hace cada uno.

### 37.4.1 Inspeccionando: Mundo

El objeto de la clase Mundo se construye cuando se invoca a la función `pilas.iniciar`. Su implementación está en el archivo `mundo.py`:

Mundo
ventana control
iniciar() ejecutar_bucle_principal() terminar()

Su responsabilidad es inicializar varios componentes de pilas, como el sistema de controles, la ventana, etc.

Uno de sus métodos más importantes es `ejecutar_bucle_principal`. Un método que se invoca directamente cuando alguien escribe la sentencia `pilas.ejecutar()`.

Si observas el código, notarás que es el responsable de mantener a todo el motor en funcionamiento.

Esta es una versión muy simplificada del método `ejecutar_bucle_principal`:

```
def ejecutar_bucle_principal(self, ignorar_errores=False):
    while not self.salir:
        pilas.motor.procesar_y_emitir_eventos()

        if not self.pausa_habilitada:
            self._realizar_actualizacion_logica(ignorar_errores)

        self._realizar_actualizacion_grafica()
```

Lo primero que debemos tener en cuenta es que este método contiene un bucle `while` que lo mantendrá en ejecución. Este bucle solo se detendrá cuando alguien llame al método `terminar` (que cambia el valor de la variable `salir` a `True`).

Luego hay tres métodos importantes:

- `procesar_y_emitir_eventos` analiza el estado de los controles y avisa al resto del sistema si ocurre algo externo, como el movimiento del mouse..
- `_realizar_actualizacion_logica` le permite a los personajes realizar una fracción muy pequeña de movimiento, poder leer el estado de los controles o hacer otro tipo de acciones.
- `_realizar_actualizacion_logica` simplemente vuelca sobre la pantalla a todos los actores y muestra el resultado del dibujo al usuario.

Otra tarea que sabe hacer el objeto `Mundo`, es administrar escenas. Las escenas son objetos que representan una parte individual del juego: un menú, una pantalla de opciones, el momento de acción del juego etc...

## 37.5 Modo interactivo

Pilas soporta dos modos de funcionamiento, que técnicamente son muy similares, pero que a la hora de programar hacen una gran diferencia.

- **modo normal:** si estás haciendo un archivo `.py` con el código de tu juego usarás este modo, tu programa comienza con una sentencia como `iniciar` y la simulación se inicia cuando llamas a `pilas.ejecutar` (que se encarga de llamar a `ejecutar_bucle_principal` del objeto mundo).
- **modo interactivo:** el modo que generalmente se usa en las demostraciones o cursos es el modo interactivo. Este modo funciona gracias a una estructura de hilos, que se encargan de ejecutar la simulación pero a la vez no interrumpe al programador y le permite ir escribiendo código mientras la simulación está en funcionamiento.

## 37.6 Motores multimedia

Al principio pilas delegaba todo el manejo multimedia a una biblioteca llamada SFML. Pero esta biblioteca requería que todos los equipos en donde funcionan tengan aceleradoras gráficas (al menos con soporte OpenGL básico).

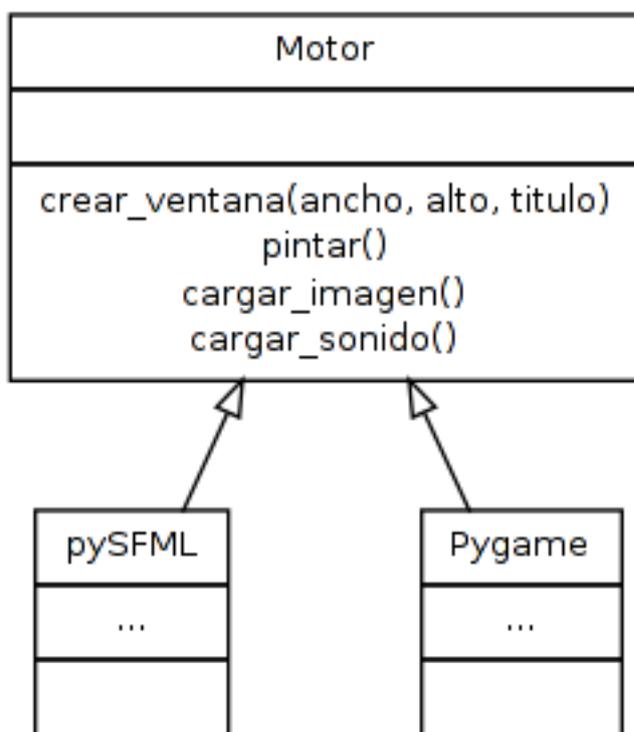
Pero como queremos que pilas funcione en la mayor cantidad de equipos, incluso en los equipos antiguos de algunas escuelas, reemplazamos el soporte multimedia con la biblioteca Qt. Que sabe acceder a las funciones de aceleración de gráficos (si están disponibles), o brinda una capa de compatibilidad con equipos antiguos.

La función que permite iniciar y seleccionar el motor es `pilas.iniciar`.

```
pilas.iniciar(usar_motor='qt')
```

Ahora bien, ¿cómo funciona?. Dado que pilas está realizado usando orientación a objetos, usamos un concepto llamado polimorfismo:

El objeto motor sabe que tiene que delegar el manejo multimedia a una instancia (o derivada) de la clase `Motor` (ver directorio `pilas/motores/`):



El motor expone toda la funcionalidad que se necesita para hacer un juego: sabe crear una ventana, pintar una imagen o reproducir sonidos, entre tantas otras cosas.

El objeto mundo no sabe exactamente qué motor está utilizando, solo tiene una referencia a un motor y delega en él todas las tareas multimedia.

Solo puede haber una instancia de motor en funcionamiento, y se define cuando se inicia el motor.

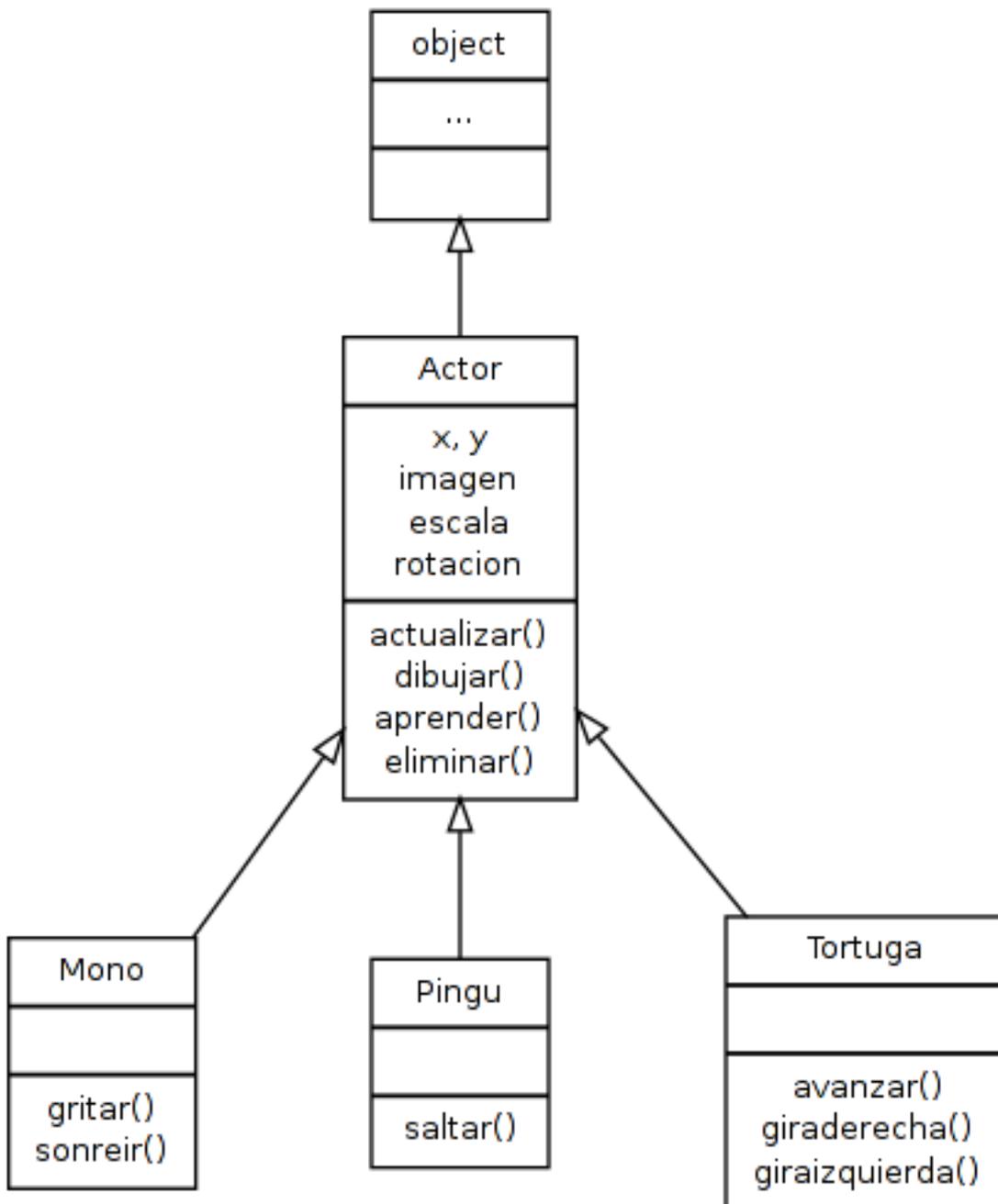
## 37.7 Sistema de actores

Los actores permiten que los juegos cobren atractivo, porque un actor puede representarse con una imagen en pantalla.

La implementación de todos los actores están en el directorio `pilas/actores`.

Todos los actores heredan de la clase `Actor`, que define el comportamiento común de todos los actores.

Por ejemplo, esta sería una versión reducida de la jerarquía de clases de los actores Mono, Pingu y Tortuga:



Hay dos métodos en los actores que se invocarán en todo momento: el método `actualizar` se invocará cuando el bucle de juego del mundo llame al método `_realizar_actualizacion_logica`, esto ocurre unas 60 veces por segundo. Y el otro método es `dibujar`, que se también se invoca desde el objeto mundo, pero esta vez en el método `_realizar_actualizacion_grafica`.

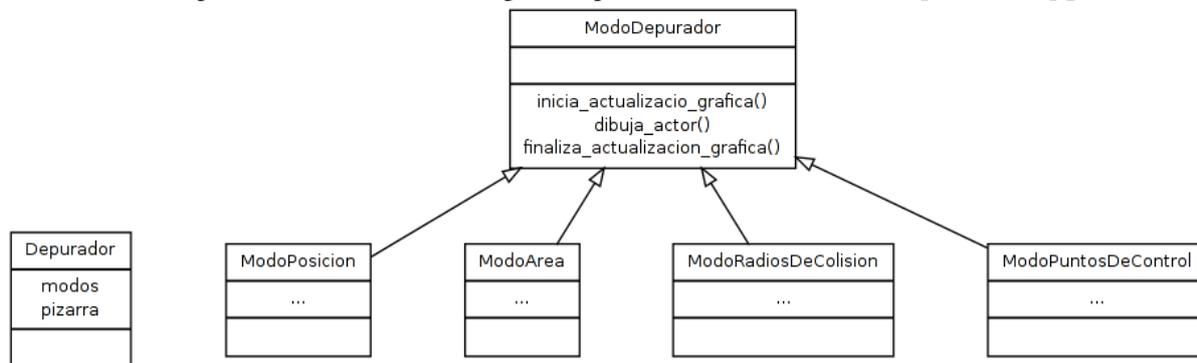
## 37.8 Modo depuración

Cuando pulsas teclas como F8, F9, F10, F11 o F12 durante la ejecución de pilas, vas a ver que la pantalla comienza a mostrar información valiosa para los desarrolladores.

Esta modalidad de dibujo la llamamos **modo depuración**, y ayuda mucho a la hora de encontrar errores o ajustar detalles.

El objeto `Mundo`, que mantiene en ejecución al juego, tiene una instancia de objeto `Depurador` que se encarga de hacer estos dibujos.

Las clases mas importantes a la hora de investigar el depurador están en el archivo `depurador.py`:



El `Depurador` tiene dos atributos, tiene una pizarra para dibujar y una lista de modos. Los modos pueden ser cualquiera de los que están en la jerarquía de `ModoDepuracion`, por ejemplo, podría tener instancias de `ModoArea` y `ModoPuntoDeControl`.

## 37.9 Sistema de eventos

Hay varios enfoques para resolver el manejo de eventos en los videojuegos.

Pilas usa un modelo conocido y elaborado llamado `Observer`, un patrón de diseño. Pero que lamentablemente no es muy intuitivo a primera vista.

En esta sección intentaré mostrar por qué usamos esa solución y qué problemas nos ayuda a resolver.

Comenzaré explicando sobre el problema de gestionar eventos y luego cómo el modelo `Observer` se volvió una buena solución para el manejo de eventos.

### 37.9.1 El problema: pooling de eventos

Originalmente, en un modelo muy simple de aplicación multimedia, manejar eventos de usuario es algo sencillo, pero con el tiempo comienza a crecer y se hace cada vez mas difícil de mantener.

Resulta que las bibliotecas multimedia suelen entregar un objeto `evento` cada vez que ocurre algo y tu responsabilidad es consultar sobre ese objeto en búsqueda de datos.

Imagina que quieres crear un actor `Bomba` cada vez que el usuario hace click en la pantalla. El código podría ser algo así:

```

evento = obtener_evento_actual()

if evento.tipo == 'click_de_mouse':
    crear_bomba(evento.x)
    crear_bomba(evento.x)
else:
    # el evento de otro tipo (teclado, ventana ...)
    # lo descartamos.
    
```

A esta solución podríamos llamarla **preguntar y responder**, porque efectivamente así funciona el código, primero nos aseguramos de que el evento nos importa y luego hacemos algo. En algunos sitios suelen llamar a esta estrategia *pooling*.

Pero este enfoque tiene varios problemas, y cuando hacemos juegos o bibliotecas se hace mas evidente. El código, a medida que crece, comienza a mezclar manejo de eventos y lógica del juego.

Para ver el problema de cerca, imagina que en determinadas ocasiones quieres deshabilitar la creación de bombas, ¿cómo harías?. ¿Y si quieres que las bombas creadas se puedan mover con el teclado?.

### 37.9.2 Otro enfoque, en pilas usamos ‘Observer’

Hay otro enfoque para el manejo de eventos que me parece mas interesante, y lo he seleccionado para el motor pilas:

En lugar de administrar los eventos uno a uno por **consultas**, delegamos esa tarea a un sistema que nos permite **suscribir y ser notificado**.

Aquí no mezclamos nuestro código con el sistema de eventos, si queremos hacer algo relacionado con un evento, escribimos una función y le pedimos al evento que llame a nuestra función cuando sea necesario.

Veamos el ejemplo anterior pero usando este enfoque, se creará una Bomba cada vez que el usuario hace `click` en la pantalla:

```
def crear_bomba(evento):
    pilas.actores.Bomba(x=evento.x, y=evento.y)
    return true
```

```
pilas.eventos.click_de_mouse.conectar(crear_bomba)
```

Si queremos que el mouse deje de crear bombas, podemos ejecutar la función `desconectar`:

```
pilas.eventos.click_de_mouse.conectar(crear_bomba)
```

o simplemente retornar `False` en la función `crear_bomba`.

Nuestro código tendrá *bajo acoplamiento* con los eventos del motor, y no se nos mezclarán.

De hecho, cada vez que tengas dudas sobre las funciones suscritas a eventos pulsa F7 y se imprimirán en pantalla.

### 37.9.3 ¿Cómo funciona?

Ahora bien, ¿cómo funciona el sistema de eventos por dentro?:

El sistema de eventos que usamos es una ligera adaptación del sistema de señales de django (un framework para desarrollo de sitios web) dónde cada evento es un objeto que puede hacer dos cosas:

- suscribir funciones.
- invocar a las funciones que se han suscrito.

#### 1 Suscribir

Por ejemplo, el evento `mueve_mouse` es un objeto, y cuando invocamos la sentencia `pilas.eventos.mueve_mouse.conectar(mi_funcion)`, le estamos diciendo al objeto “quiero que guardes una referencia a `mi_funcion`”.

Puedes imaginar al evento como un objeto contenedor (similar a una lista), que guarda cada una de las funciones que le enviamos con el método `conectar`.

#### 2 Notificar

La segunda tarea del evento es notificar a todas las funciones que se suscribieron.

Esto se hace, retomando el ejemplo anterior, cuando el usuario hace click con el mouse.

Los eventos son objetos `Signal` y se inicializan en el archivo `eventos.py`, cada uno con sus respectivos argumentos o detalles:

```
click_de_mouse = Evento("click_de_mouse")
pulsar_tecla = Evento("pulsar_tecla")
[ etc...]
```

Los argumentos indican información adicional del evento, en el caso del click, observarás que los argumentos son el botón pulsado y la coordenada del puntero.

Cuando se quiere notificar a las funciones conectadas a un evento simplemente se tiene que invocar al método `emitir` del evento y proveer los argumentos que necesita:

```
click_de_mouse.emitir(button=1, x=30, y=50)
```

Eso hará que todas las funciones suscritas al evento `click_de_mouse` se invoquen con el argumento `evento` representando esos detalles:

```
def crear_bomba(evento):
```

```
    print evento.x
    # imprimirá 30
```

```
    print evento.y
    # imprimirá 50
```

```
    [ etc... ]
```

La parte de pilas que se encarga de llamar a los métodos `emitir` es el método `procesar_y_emitir_eventos` del motor.

## 37.10 Habilidades

Los actores de pilas tienen la cualidad de poder ir obteniendo comportamiento desde otras clases.

Esto te permite lograr resultados de forma rápida, y a la vez, es un modelo tan flexible que podrías hacer muchos juegos distintos combinando los mismos actores pero con distintas habilidades.

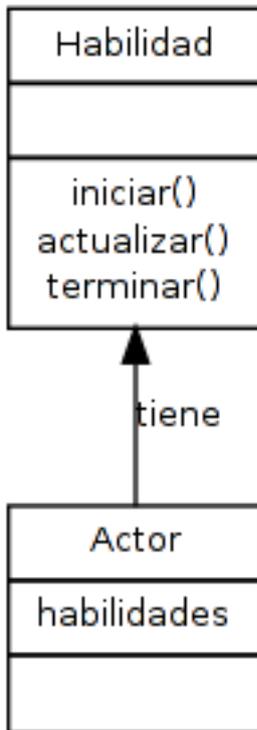
Veamos un ejemplo, un actor sencillo como `Mono` no hace muchas cosas. Pero si escribimos lo siguiente, podremos controlarlo con el mouse:

```
mono = pilas.actores.Mono()
mono.aprender(pilas.habilidades.Arrastrable)
```

Lo que en realidad estamos haciendo, es vincular dos objetos en tiempo de ejecución. `mono` es un objeto `Actor`, y tiene una lista de habilidades que puede aumentar usando el método `aprender`.

El método `aprender` toma la clase que le enviamos como argumento, construye un objeto y lo guarda en su lista de habilidades.

Este es un modelo de cómo se conocen las clases entre sí:



Entonces, una vez que invocamos a la sentencia, nuestro actor tendrá un nuevo objeto en su lista de habilidades, listo para ejecutarse en cada cuadro de animación.

### 37.10.1 ¿Cómo se ejecutan las habilidades?

Retomando un poco lo que vimos al principio de este capítulo, lo que mantiene con *vida* al juego es el bucle principal, la clase `Mundo` tiene un bucle que recorre la lista de actores en pantalla y por cada uno llama al método `actualizar`.

Bien, las habilidades se mantienen en ejecución desde ahí también. Esta es una versión muy simplificada del bucle que encontrarás en el archivo `mundo.py`:

```

def ejecutar_bucle_principal(self, ignorar_errores=False):
    while not self.salir:
        self.actualizar_actores()

        [ etc ... ]

def actualizar_actores(self):
    for actor in pilas.actores.todos:
        actor.actualizar()
        actor.actualizar_habilidades()
  
```

Aquí puedes ver dos llamadas a métodos del actor, el método `actualizar` se creó para que cada programador escriba ahí lo que quiera que el personaje haga (leer el teclado, hacer validaciones, moverse etc). Y el método `actualizar_habilidades` es el encargado de *dar vida* a las habilidades.

Técnicamente hablando, el método `actualizar_habilidades` es muy simple, solamente toma la lista de objetos habilidades y los actualiza, al Actor no le preocupa en lo mas mínimo “qué” hace cada habilidad, solamente les permite ejecutar código (ver código `estudiante.py`, una superclase de `Actor`):

```

def actualizar_habilidades(self):
    for h in self.habilidades:
        h.actualizar()
  
```

Entonces, si queremos que un actor haga muchas cosas, podemos crear un objeto habilidad y vincularlo con el actor. Esto permite generar “comportamientos” re-utilizables, la habilidad se codifica una vez, y se puede usar muchas veces.

### 37.10.2 Objetos habilidad

Las habilidades interactúan con los actores, y por ese motivo tienen que tener una interfaz en común, de modo tal que desde cualquier parte de pilas puedas tratar a una habilidad como a cualquier otra.

La interfaz que toda habilidad debe tener es la que define la clase `Habilidad` del archivo `habilidades.py`:

```
class Habilidad:

    def __init__(self, receptor):
        self.receptor = receptor

    def actualizar(self):
        pass

    def eliminar(self):
        pass
```

Tiene que tener tres métodos, uno que se ejecuta al producirle la relación con un actor, un método que se ejecutará en cada iteración del bucle de juego (`actualizar`) y un último método para ejecutar cuando la habilidad se desconecta del actor. Este método `eliminar` suele ser el que desconecta eventos o cualquier otra cosa creada temporalmente.

Ten en cuenta que el método `__init__`, que construye al objeto, lo invoca el propio actor desde su método `aprender`. Y el argumento `receptor` será una referencia al actor que *aprende* la habilidad.

Veamos un ejemplo muy básico, imagina que quieres hacer una habilidad muy simple, que gire al personaje todo el tiempo, cómo una aguja de reloj. Podrías hacer algo así:

```
class GirarPorSiempre(pilas.habilidades.Habilidad):

    def __init__(self, receptor):
        self.receptor = receptor

    def actualizar(self):
        self.receptor.rotacion += 1
```

```
mono = pilas.actores.Mono()
mono.aprender(GirarPorSiempre)
```

La sentencia `aprender` construirá un objeto de la clase que le indiquemos, y el bucle de pilas (en `mundo.py`) dará la orden para ejecutar los métodos `actualizar` de cada habilidad conocida por los actores.

### 37.10.3 Argumentos de las habilidades

En el ejemplo anterior podríamos encontrar una limitación. El actor siempre girará a la misma velocidad.

Si queremos que los personajes puedan girar a diferentes velocidades tendríamos que agregarle argumentos a la habilidad, esto es simple: solo tienes que llamar al método `aprender` con los argumentos que quieras y asegurarte de que la habilidad los tenga definidos en su método `__init__`.

Este es un ejemplo de la habilidad pero que permite definir la velocidad de giro:

```
class GirarPorSiempre(pilas.habilidades.Habilidad):

    def __init__(self, receptor, velocidad=1):
        self.receptor = receptor
        self.velocidad = velocidad
```

```
def actualizar(self):
    self.receptor.rotacion += self.velocidad

a = pilas.actores.Mono()
a.aprender(GirarPorSiempre, 20)
```

Listo, es casi idéntico al anterior, si llamas a `aprender` con un argumento como `20`, el actor girará mucho más rápido que antes. Y si no especificas la velocidad, se asumirá que la velocidad es `1`, porque así lo indica el método `__init__`.

## 37.11 Documentación

El sistema de documentación que usamos en pilas es Sphinx, un sistema muy interesante porque nos permite gestionar todo el contenido del manual en texto plano, y gracias a varias herramientas de conversión como `structuredText` y `latex`, se producen muchos formatos de salida como HTML y PDF.

Toda la documentación del proyecto está en el directorio `doc`. El directorio `doc/sources` contiene todos los archivos que modificamos para escribir contenido en la documentación.

Para generar los archivos PDF o HTML usamos el comando `make` dentro del directorio `doc`. El archivo que dispara todas las acciones que sphinx sabe hacer están definidas en el archivo `Makefile`.