Desarrollo de videojuegos 2D con PYTHON

Desde **www.ra-ma.es** podrá descargar material adicional.

ALBERTO CUEVAS ÁLVAREZ

Ra-Ma®

Videojuegos 2D

Desarrollo con Python

Videojuegos 2D

Desarrollo con Python

Alberto Cuevas Álvarez

Ra-Ma®

La ley prohíbe
fotocopiar este libro

Editado por:
RA-MA Editorial
Calle Jarama, 3A, Polígono Industrial Igarsa
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: *editorial@ra-ma.com*
Internet: *www.ra-ma.es* y *www.ra-ma.com*
ISBN: 978-84-9964-798-2
Depósito legal: M-37124-2018
Maquetación: Antonio García Tomé
Diseño de portada: Antonio García Tomé
Filmación e impresión: Safekat
Impreso en España en diciembre de 2018

Para mis abuelos.
Por darme tanto cariño y buenos valores
en una infancia feliz.

ÍNDICE

PRÓLOGO

Hablando de forma genérica desarrollar un videojuego puede ser el trabajo de una tarde para un aficionado o de años para un equipo de desarrollo especializado compuesto por decenas de profesionales, ya que el rango de complejidad que podemos tener es muy amplio.

Una clasificación básica de los videojuegos está basada en el número de dimensiones en las que se desarrollan. Tendremos por tanto juegos en dos o tres dimensiones (2D o 3D), al que añadiremos los que simulan la tercera dimensión sobre un plano bidimensional (2.5D).

Los videojuegos suelen estar programados en C o C++. El motivo es que son lenguajes muy rápidos, algo conveniente para determinados desarrollos y fundamental para otros. La práctica totalidad de los motores (núcleos) de videojuegos usan uno o ambos lenguajes.

En nuestro caso hemos elegido desarrollar **videojuegos 2D** sobre **Python**, pensando en un lector que se inicia en el mundo de la programación de este tipo de aplicaciones y considerando que ambas elecciones son perfectas para dicho fin. Haremos uso de la librería **cocos2d** y del editor de mapas **Tiled**, además de varias herramientas que iremos comentando a lo largo del libro.

La elección de cocos2d respecto a otras alternativas se debe a que es software libre, está escrita en Python, tiene fácil instalación, buena documentación, y se adapta perfectamente a nuestros objetivos.

Se presuponen conocimientos fundamentales de Python 3, ya que todos los códigos están realizados sobre la versión **3.6.6**. La versión de cocos2d será **0.6.5** y la de Tiled **1.1.2**.

He usado el sistema operativo **Windows 8.1** sin hacer referencias a cómo actuar en otro, pensando en la simplicidad y considerando que la adaptación a un entorno distinto es muy sencilla si se dispone de conocimientos básicos sobre él.

Como IDE[1] he empleado **PyScripter 3.0.2** (versión 32 bits), que sólo funciona bajo Windows. Considero que se adapta, por velocidad y facilidad de uso, perfectamente a lo que queremos. En el caso de no trabajar con Windows mi recomendación es **PyCharm**[2].

Los códigos presentados en el libro son creaciones personales no exentas de posibles mejoras, por lo que animo al lector a realizarlas. He puesto mucho énfasis en que todos funcionen correctamente y estén libres de errores fundamentales.

En la redacción no he usado por lo general elementos distintivos[3] para variables, funciones o similares (buscando tener un texto más uniforme), algo que debemos tener en cuenta de cara a una correcta interpretación.

Al comentar los códigos, para referenciar de forma más compacta las líneas colocaré el carácter 'L' antes de su número.

La resolución de pantalla que he empleado ha sido 1920x1080 píxeles. El lector podrá modificar los tamaños de las ventanas que aparecen en los códigos para adaptarlos a la resolución de la que disponga, o usar la opción de visualizar la pantalla completa (algo disponible en la práctica totalidad de los casos).

En la web del libro tendremos para descargar todo el material necesario a la hora de seguirlo. Ello incluye la carpeta **Videojuegos_Python** que contiene los ficheros de ejemplo y los necesarios para crear los dos videojuegos principales. Entre ellos hay imágenes, sonidos y animaciones. Todos ellos tienen licencia libre y tenemos la posibilidad de utilizarlos.

La documentación oficial de cocos2d la tenemos en la siguiente dirección web:

http://python.cocos2d.org/doc/index.html

Si queremos consultar también la de pyglet la conseguiremos aquí:

https://pyglet.readthedocs.io/en/pyglet-1.3-maintenance/

Ambas se incluyen (con formato HTML) en el material descargable del libro.

Espero que mediante esta obra el lector pueda finalmente conocer los fundamentos de la programación de videojuegos 2D sobre Python, disfrutando del camino y deseando con posterioridad aplicarlos en sus propios proyectos.

---

1 Entorno integrado de desarrollo (Integrated Development Environment).

2 https://www.jetbrains.com/pycharm/

3 Poner en cursiva, en negrita o de otro color, por ejemplo.

1

FUNDAMENTOS DE COCOS2D

1.1 INTRODUCCIÓN

El mundo de los motores gráficos para videojuegos está dominado por los lenguajes de programación C y C++, habiendo sido la mayoría escritos en ellos. Python no es tan rápido, por lo que si lo usamos en un videojuego será por lo general en un nivel más alto (como puede ser la lógica del juego o la automatización de determinados procesos) o en juegos de menor complejidad como los desarrollados en 2D, que serán los que ocupan este libro.

Motores gráficos o librerías escrito/as (al menos parcialmente) en Python existen muy poco/as:

- **pyglet**

  Es una librería gráfica multimedia y multiplataforma (Windows, OS X y GNU/Linux) para Python, destinada al desarrollo de juegos y otras aplicaciones visuales. Soporta ventanas, manejo de eventos de interfaz de usuario, gráficos OpenGL[4], carga de imágenes y videos, y reproducción de sonidos y música. No tiene dependencias externas ni requisitos de instalación, por lo que es muy fácil de instalar. Se proporciona bajo la licencia de código abierto BSD, permitiéndole usarlo tanto para proyectos comerciales como para otros proyectos de código abierto con muy poca restricción.

4 Open Graphics Library, especificación que define una API (interfaz de programación de aplicaciones) para generar gráficos 2D y 3D.

- **pygame**

  Es una librería de código abierto para Python que nos permitirá crear juegos o aplicaciones multimedia. Está construida sobre la librería SDL[5] y, al igual que ella, es multiplataforma (Windows, OS X y GNU/Linux). No requiere obligatoriamente el uso de OpenGL sino que puede usar otros como DirectX[6]. Está escrito en ensamblador, C y Python. Puede usar varios cores de nuestra CPU. Tiene licencia LGPL[7].

- **cocos2d**

  Veremos sus características principales un poco más adelante.

Entre los motores escritos en C++ pero que están diseñados para usar Python destacamos:

- **Blender Game Engine (BGE)**

  Forma parte de Blender[8] antes de la versión[9] 2.8. Motor con capacidad 3D escrito en C/C++ y Python. Tiene licencia GPL.

- **Panda3D**

  Es un motor de videojuegos con capacidades 3D. El núcleo está escrito en C++ pero permite el uso de Python de forma muy completa, pudiendo acceder a sus interioridades. Tiene licencia BSD.

También hay motores escritos en C/C++ que tienen plugings[10] para poder usar Python en ellos, aunque generalmente no nos permitirán acceder a todas las características y se usarán en un nivel alto (lógica, automatización o similares). Dos de ellos (los que considero más relevantes, ambos muy potentes) son:

---

5 Simple DirectMedia Layer, escrita en C y desarrollada inicialmente para GNU/Linux, nos permite trabajar con imágenes, animaciones, audio y video. Tiene licencia LGPL.

6 API para trabajar con elementos multimedia (principalmente videojuegos) en Windows.

7 En cuanto a restrictiva se coloca entre la GPL (la que más) y las BSD o MIT (las que menos).

8 Programa multiplataforma de creación, modelado, iluminación, renderizado y animación de gráficos tridimensionales. Programado en C, C++ y Python. Con licencia GPL.

9 Su salida está prevista en los primeros meses del 2019 y en él desaparece de forma independiente BGE, aunque sí habrá un modo interactivo. Está por ver cómo evoluciona todo, y cómo lo hacen también forks (bifurcaciones) de BGE como UPBGE.

10 Lo podríamos traducir como “complemento”.

- **Godot**

  Motor de videojuegos 2D y 3D multiplataforma, desarrollado por una comunidad propia. Es de código abierto bajo una Licencia MIT. Programado en C y C++.

  Para programar en él se usa principalmente el lenguaje GDScript[11].

- **Unreal Engine**

  Es un motor de videojuegos para PC y consolas, multiplataforma y con capacidades 3D. Está escrito en C++. Tiene una licencia que permite su uso y, si se comercializa un proyecto basado en él y las ganancias son mayores de 3000€, se deberá pagar un 5% a la compañía que lo produce, Epic Games.

Con **cocos2d** tenemos un framework[12] basado en pyglet y OpenGL usado para desarrollar juegos, demos y aplicaciones gráficas (generalmente interactivas). Está escrito en Python (y por lo tanto pensado para trabajar con él), es multiplataforma (Windows, OS X y GNU/Linux) y tiene una licencia libre tipo BSD. Incorpora además un intérprete de Python que nos será útil en el proceso de depuración.

Existen versiones de cocos2d para otras plataformas:

- Cocos2d-X : C++
- Cocos2d-Js: Javascript
- Cocos2D-Swift: Objective-C
- Cocos2d-XNA: C#

Para instalar Python y cocos2d en nuestro sistema seguiremos el Apéndice A del libro.

1.2 ELEMENTOS BÁSICOS DE COCOS2D

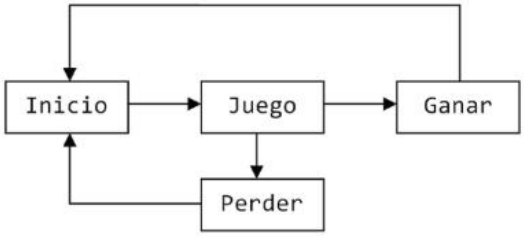
Hay una serie de elementos que debemos conocer de cara a crear nuestra aplicación con cocos2d. Son los siguientes:

11 Lenguaje de alto nivel (muy similar a Python) creado especialmente para Godot.

12 Entorno de trabajo, generalmente relacionado con un determinado lenguaje, que contiene las herramientas necesarias para la realización de determinadas aplicaciones.

- Una **escena** (también denominada fase o etapa) es una parte independiente de la aplicación. A pesar de que puede haber en ésta varias de ellas, solamente una está activa en un momento dado. Por ejemplo podemos tener en nuestro juego una escena de menú inicial, varias correspondientes a cada uno de los niveles y una final indicando las puntuaciones máximas. Una escena del juego podría ser algo así:

Existe una conexión lógica entre ellas (por ejemplo que del nivel 1 pasamos al nivel 2, o del menú al nivel 1) pero podremos crear cada una de forma bastante independiente. Un sencillo ejemplo es:



Una escena se implementa mediante un objeto de la clase **Scene**, que es una subclase de CocosNode. Tenemos también una subclase de Scene llamada **TransitionScene** que nos permitirá hacer transiciones entre dos escenas.

- Una **capa** es una de las partes que pueden componer la escena. Podemos pensar en ellas como hojas de fondo transparente[13] que superpuestas en un determinado orden forman una escena. Nos ayuda a organizar ésta,

13 También las hay de fondo opaco.

teniendo por ejemplo una capa para el paisaje de fondo, otra para los personajes y una tercera para el HUD[14].

Se implementa con la clase **Layer** (subclase de CocosNode) y en las capas es donde se definen los manejadores de eventos[15], que se propagan entre ellas (de adelante hacia atrás) hasta que alguna captura y acepta el evento. Tenemos varias capas especializadas como Menu, ColorLayer o RectMapLayer, que implementan respectivamente menús, capas con fondo sólido de color y mapas de baldosas rectangulares[16].

- El **director** es una instancia única[17] encargada de dirigir las escenas (almacenadas en una pila), reemplazándolas, llamándolas, pausándolas o finalizándolas, conociendo además en cada momento cuál de ellas está activa. También inicializa la ventana principal. Se implementa importando **director** desde el módulo cocos2d.director.

14 Head Up Display, que podemos traducir como "visualización frontal" y que se refiere a la representación de información por pantalla (por ejemplo para el número de vidas o la puntuación).

15 Indicaré qué son los eventos un poco más adelante, en esta misma clasificación de elementos fundamentales.

16 Los veremos más adelante. Básicamente son mapas creados en base a elementos rectangulares a los que llamaremos baldosas.

17 Singleton object. Solamente existe una instancia de ese tipo en nuestra aplicación.

- Un **sprite** es una imagen 2D representada en la pantalla que se puede escalar, mover, rotar, animar o variar su opacidad. Se implementa mediante la clase **Sprite**, también subclase de CocosNode. Un sprite puede contener otros sprite hijo, que son todos transformados si el sprite padre lo hace.

- Las **acciones** son órdenes que se dan a los objetos CocosNode[18] y que generalmente modifican alguno de sus atributos.

- Los **eventos** podríamos describirlos como cosas que pasan en nuestra aplicación, por ejemplo la pulsación de una tecla o el clic en un botón del ratón. En cocos2d los manejamos mediante el framework de eventos de pyglet. Se obtienen las entradas del usuario (teclado, ratón) recibiendo en la capa[19] los eventos generados en director.window, es decir, en la ventana principal. Tras recibirlos serán tratados mediante los manejadores de eventos correspondientes.

Una escena es descrita en cocos2d como un árbol de objetos CocosNode donde la raíz es un objeto Scene, los descendientes más cercanos usualmente son capas, y ellas sostienen y organizan elementos individuales (por ejemplo, los sprites).

Como en las capas se define la apariencia y el comportamiento de los elementos principales de la aplicación, la mayoría del tiempo lo emplearemos en programar subclases de Layer para conseguir lo que deseamos. Para cargar los elementos que queramos podremos sobrecargar el método __init__() de la subclase, ya que se llamará al crear la capa. También tenemos una serie de capas especializadas:

- Menu: nos permite implementar menús simples.

- ColorLayer: capa con un fondo sólido de color.

- RectMapLayer, HexMapLayer, TmxObjectLayer: representan respectivamente mapas de baldosas (rectangulares o hexagonales) u objetos TMX[20].

- ScrollableLayer, ScrollingManager: nos permiten, respectivamente, crear capas con scroll y manejarlo posteriormente.

---

18 Recordemos que Scene, Layer y Sprite derivan de ella, por lo que las acciones pueden aplicarse a prácticamente todos los elementos.

19 Para ello la capa debe tener el atributo de clase is_event_handler con valor True.

20 Posteriormente veremos a qué nos referiremos con esta terminología.

- MultiplexLayer: grupo de capas en las que sólo una está visible en un momento dado.
- PythonInterpreterLayer: usado por el director para ejecutar una consola[21] interactiva en la que poder inspeccionar los objetos de nuestra escena.

La clase **CocosNode** (derivada de object) es el elemento básico de cocos2d. Todo lo que se pueda dibujar o contenga elementos que se puedan dibujar deriva de CocosNode. Sus principales características son las siguientes:

- Puede contener otros objetos CocosNode.
- Puede planificar callbacks[22] periódicas.
- Puede ejecutar una serie de acciones.

Crear una clase a partir de CocosNode suele significar realizar alguna de las siguientes tareas:

- Sobrecargar __init__ para inicializar recursos y planificar callbacks.
- Crear callbacks para manejar el avance del tiempo.
- Sobrecargar el método draw() para renderizar el nodo.

Para instanciar el **director** no lo debemos hacer directamente sino de la siguiente manera:

```
from cocos.director import director
```

Con ello conseguimos acceder al único objeto de la clase Director. Posteriormente lo inicializaremos mediante el método init() para más adelante ejecutarlo mediante el método run().

Veamos a continuación cómo es una sencilla aplicación en cocos2d donde simplememte visualizaremos en la pantalla un texto centrado. El fichero es **ejemplo_sencillo_1.py**:

21 Se activa/desactiva mediante Ctrl+i.

22 Son funciones que se llaman dentro de otras funciones para ser ejecutadas en un instante determinado.

```
1
2  import cocos
3
4  class VerTexto(cocos.layer.Layer):
5      def __init__(self):
6          super().__init__()
7          mi_etiqueta = cocos.text.Label('Visualización de un texto simple',
8                                         font_name = 'Consolas',
9                                         font_size = 18,
10                                        anchor_x = 'center', anchor_y = 'center')
11
12         mi_etiqueta.position = (320, 240)
13         self.add(mi_etiqueta)
14
15
16 if __name__ == "__main__":
17     cocos.director.director.init()
18     capa_texto = VerTexto()
19     escena_principal = cocos.scene.Scene(capa_texto)
20     cocos.director.director.run(escena_principal)
21
```

Genera la siguiente salida:

Comentario del código:

- En L2 importamos el módulo cocos, que contendrá todos los elementos que necesitamos.

- En L4-13 definimos la clase VerTexto, que deriva de la clase Layer del módulo cocos.layer. Contiene una etiqueta[23] que creamos en L7-10 con una serie de características[24] (tipo de letra 'Consolas', tamaño 18 puntos y centrada tanto horizontal como verticalmente) y que colocamos en L12 justo en el centro de la pantalla. En L13 añadimos la etiqueta a la capa mediante el método add().
- En L17 inicializamos el director mediante el método init().
- El L18 creamos una capa (instancia de nuestra clase VerTexto) que denominamos capa_texto.
- En L19 creamos una escena (instancia de la clase Scene del módulo cocos.scene) llamada escena_principal, que contiene a capa_texto.
- En L20 ejecutamos escena_principal mediante el método run() del director.

En este primer ejemplo tenemos una sola escena. En el caso de que haya varias el director podrá manejar el flujo de escenas, por ejemplo:

- Reemplazar la escena actual (que finalizará) por otra mediante el método director.**replace**(nueva_escena).
- Ejecutar una nueva escena y colocar la escena actual en una pila de escenas suspendidas mediante el método director.**push**(nueva_escena).
- Sacar una escena de la pila de escenas, reemplazando la escena actual (que finalizará) usando el método director.**pop**().
- Finalizar la escena actual proporcionando un valor de finalización. La nueva escena que ejecutaremos será sacada desde la pila de escenas. En este caso usaremos el método director.scene**.end**(valor_fin).

23 Instancia de la clase Label de cocos.

24 Para saber más sobre la clase Label ver el módulo cocos.text en el Apéndice B.

En el cambio de escenas ocurre lo siguiente:

- Se llama al método on_exit() de la escena saliente.
- Se inhabilitan los manejadores de eventos de la escena saliente.
- Se llama al método on_enter() de la escena entrante.
- Se habilitan los manejadores de eventos de la escena entrante.

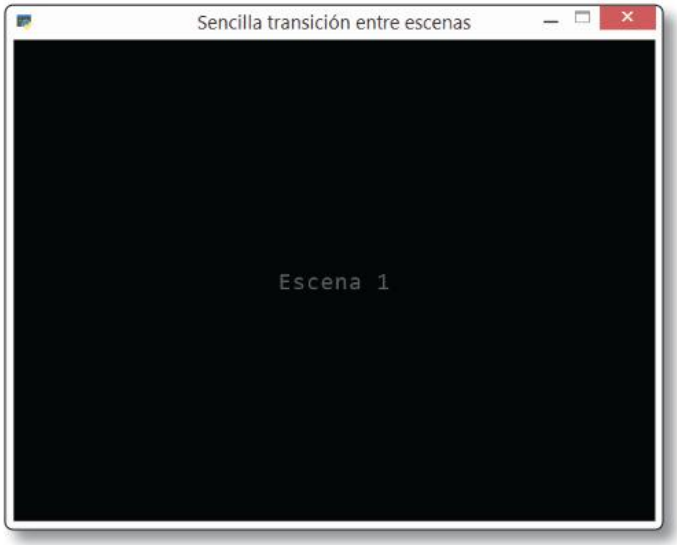
Tendremos también la posibilidad de hacer **transiciones** entre escenas para dotar al cambio de mayor atractivo. Eso se logra mediante una **escena de transición** que realiza un efecto visual desde la escena que sustituimos a la que colocamos. Existen varios efectos de distinto tipo, que los tendremos en forma de clases en el módulo cocos.scenes.transitions[25].

Veamos un sencillo ejemplo de transición entre dos escenas (**ejemplo_transicion.py**):

```
1
2 import cocos
3 from cocos.scenes.transitions import FadeTransition
4
5 class VerTexto(cocos.layer.Layer):
6     def __init__(self):
7         super().__init__()
8         mi_etiqueta = cocos.text.Label('Escena 1',
9                                        font_name = 'Consolas',
10                                       font_size = 18,
11                                       anchor_x = 'center', anchor_y = 'center')
12        mi_etiqueta.position = (320, 240)
13        self.add(mi_etiqueta)
14
15
16 class VerTexto2(cocos.layer.Layer):
17     def __init__(self):
18         super().__init__()
19         mi_etiqueta = cocos.text.Label('Escena 2',
20                                        font_name = 'Consolas',
21                                        font_size = 18,
22                                        anchor_x = 'center', anchor_y = 'center')
23        mi_etiqueta.position = (320, 240)
24        self.add(mi_etiqueta)
25
26
27 if __name__ == "__main__":
28     cocos.director.director.init(caption= 'Sencilla transición entre escenas')
29
30     escena_1 = cocos.scene.Scene(VerTexto())
31     escena_2 = cocos.scene.Scene(VerTexto2())
32
33     cocos.director.director.run(FadeTransition(escena_2, 5, escena_1))
34
```

25 Consultar Apéndice B.

Un punto intermedio del paso de una a otra será el siguiente:



Ahora tenemos dos capas muy similares a las del ejemplo anterior, con un texto centrado que indica la escena que las contendrá. En L3 importamos y en L33 ejecutamos FadeTransition(), que irá desvaneciendo progresivamente la escena saliente para posteriormente dibujar cada vez más nítidamente la entrante. Hemos indicado que la transición dure 5 segundos. Veremos ejemplos más complejos de transiciones entre escenas en el apartado 1.2.3.

1.2.1 Menús

Un elemento típico de cualquier videojuego son los menús. Para trabajar con ellos en cocos2d tenemos la clase **Menu**, que implementa una capa para representarlos. Dentro de los menús hay una serie de ítems que pueden ser de distinto tipo, estando representados por las siguientes clases:

- **MenuItem**
  Nos da la capacidad de mostrar una etiqueta con texto.
- **MultipleMenuItem**
  Podremos cambiar en él entre varios valores dados previamente.
- **ToggleMenuItem**
  Para trabajar con opciones booleanas.

- **EntryMenuItem**
  Nos permitirá introducir una cadena.
- **ImageMenuItem**
  Nos mostrará una imagen.
- **ColorMenuItem**
  Tendremos la posibilidad de elegir entre varios colores preseleccionados.

En cada una de ellas podremos indicar una función que se ejecute cuando seleccionemos el ítem.

La forma habitual para trabajar con menús en cocos2d será:

1. Crear una subclase de Menu.
2. Sobrescribir su método __init__() para configurar los diferentes ítems del menú y añadirlos mediante el método **create_menu**()[26].
3. Añadir el menú a otra capa o a la escena.

Para ejemplificar el uso de un menú y usar toda la variedad de posibles tipos de ítems crearemos uno en el que podamos elegir lo siguiente:

- Activar/desactivar el sonido.
- Elegir entre 5 posibles resoluciones de pantalla.
- Elegir entre 4 colores.
- Elegir la dificultad del juego, de 1 a 10.
- Tener la opción de iniciar el juego haciendo clic en la imagen de un guerrero.
- Tener la opción de salir del menú y cerrar la ventana.

26 Para saber más sobre él consultar el módulo cocos.menu en el Apéndice B.

El código lo tenemos en **ejemplo_menu.py**:

```
1
2 from cocos.director import director
3 from cocos.scene import Scene
4 from cocos.menu import *
5
6 class MiMenu(Menu):
7     def __init__(self):
8         super().__init__("Menú principal")
9
10        item1= ToggleMenuItem('Sonido: ', self.eleccion_sonido, True)
11
12        resoluciones = ['640x480','800x600', '1024x768', '1280x720', '1600x900']
13        item2= MultipleMenuItem('Resolución: ', self.eleccion_resolucion,
14                                resoluciones)
15
16        colores = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (0, 200, 200)]
17        item3 = ColorMenuItem('Color:', self.eleccion_color, colores)
18
19        item4 = EntryMenuItem('Dificultad (1-10): ', self.eleccion_dificultad,
20                              '', max_length=2)
21        item5 = ImageMenuItem('mi_guerrero_2.png', self.on_image_callback)
22        item6 = MenuItem('Salir', self.salir )
23
24        self.create_menu( [item1, item2, item3 ,item4, item5, item6] )
25
26    def eleccion_sonido(self, b):
27        if b:
28            sel= 'activado'
29        else:
30            sel = 'desactivado'
31        print('Tu elección de audio es: ', sel)
32
33    def eleccion_resolucion(self, valor):
34        print('Has elegido la resolución número {}'.format(valor+1))
35
36    def eleccion_color(self, valor):
37        print('Has elegido el color número {}'.format(valor+1))
38
39    def eleccion_dificultad(self, valor):
40        print('Has elegido el nivel de dificultad', valor)
41
42    def on_image_callback(self):
43        print('Has elegido comenzar el juego')
44
45    def salir(self):
46        director.window.close()
47        print('Has elegido salir')
48
49
50 def main():
51     ventana = director.init(width=800, height=600, caption='')
52     ventana.set_location(500,200)
53     director.run(Scene(MiMenu()))
54
55
56 if __name__ == '__main__':
57     main()
58
```

El resultado de su ejecución será el siguiente:

Definimos la clase MiMenu (basada en Menu) y sobrescribimos su método __init__(), en el que creamos 6 instancias de las distintas clases de ítems que son añadidas en L24 mediante el método create_menu(). Definimos además 6 métodos asociados a la selección de cada uno de los ítems. En nuestro caso simplemente se ha mostrado una información por pantalla, salvo en el caso de la opción "Salir", pero en el desarrollo del videojuego ejecutaríamos elementos o configuraríamos variables.

1.2.2 Acciones

Las **acciones** son órdenes dadas a los objetos CocosNode que generalmente modifican alguno de sus atributos, como la escala, posición o rotación. En relación al tiempo las acciones pueden ser de tres tipos:

- Aplicadas en un solo paso (**InstantAction**).
- Aplicadas en una serie de pasos dentro de un tiempo prefijado (**IntervalAction**).
- Aplicadas por tiempo indefinido (**Action**).

Las acciones de intervalo (IntervalAction) tienen algunas características interesantes:

- El flujo temporal puede ser modificado mediante las acciones **Accelerate**, **AccelDeccel** y **Speed**.

- Todas las acciones relativas (las terminadas en "By") y algunas de las absolutas (las terminadas en "To") tienen la función **Reverse**() que la ejecuta en sentido inverso.

Las acciones pueden ser divididas en varios grupos, dependiendo de la naturaleza de lo que realizan. Entre ellos destacamos:

- De posición:
  - **MoveBy**: Mueve el objeto, durante un periodo de tiempo, una serie de píxeles relativas a la posisión que ocupa en ese momento.
  - **MoveTo**: Mueve el objeto, durante un periodo de tiempo, a unas coordenadas absolutas que le proporcionamos.
  - **JumpBy:** Mueve el objeto, simulando una serie de saltos y durante un tiempo determinado, a una posición relativa a la que ocupa en ese instante.
  - **JumpTo:** Mueve el objeto, simulando una serie de saltos y durante un tiempo determinado, a unas coordenadas absolutas.
  - **Bezier:** Mueve el objeto, durante un tiempo determinado, a través de una curva de Bezier.
  - **Place:** Coloca el objeto en unas coordenadas absolutas.
- De escala:
  - **ScaleBy:** Escala el objeto un determinado valor y durante un determinado periodo de tiempo.
  - **ScaleTo:** Escala el objeto un determinado valor y durante un determinado periodo de tiempo.
- De rotación:
  - **RotateBy:** Rota el objeto, en un tiempo determinado, una serie de grados medidos en sentido horario.
  - **RotateTo:** Rota el objeto, en un tiempo determinado, a un valor absoluto de grados medidos en sentido horario. El sentido de rotación dependerá de ese valor.
- De visibilidad:
  - **Show:** Muestra el objeto.
  - **Hide:** Oculta el objeto.

- **Blink:** Hace parpadear al objeto durante un determinado tiempo, al finalizar el cual vuelve a su valor original de visibilidad.
- **ToggleVisibility:** Conmuta el valor de visibilidad del objeto.

◤ De opacidad:

- **FadeIn:** Hace aparecer el objeto, dibujándolo con progresiva nitidez en un determinado intervalo de tiempo.
- **FadeOut:** Hace desaparecer el objeto difuminándolo progresivamente durante un determinado intervalo de tiempo.
- **FadeTo:** Difumina el objeto a un valor determinado de alpha (un valor entre 0 y 255 que indica la opacidad).

◤ De tiempo:

- **Delay:** No hace nada durante una serie de segundos. La usaremos para retardar acciones.
- **RandomDelay:** No hace nada durante una serie de segundos elegidos aleatoriamente dentro de un intervalo dado.

◤ De control:

- **CallFunc:** Llama a una determinada función.
- **CallFuncS:** Llama a una determinada función a la que se le envía el objeto que ejecuta la acción.

Tenemos la posibilidad de combinar o modificar acciones:

◤ Operador '+'**:** mediante él conseguiremos realizar acciones de forma secuencial (una después de la otra).

◤ Operador '|'**:** nos permite realizar acciones en paralelo (varias a la vez).

◤ Operador '*'**:** conseguiremos la repetición de la acción un número entero de veces.

◤ **Repeat:** su uso hace repetir la acción de forma continuada.

También existen los modificadores del flujo temporal:

- **Accelerate:** Cambia la aceleración de la acción.
- **AccelDeccel:** Cambia la velocidad de la acción, pero mantiene la original al inicio y al final.
- **Speed:** Cambia la velocidad de una acción.
- **Reverse:** Invierte la acción

Podremos crear además nuestras propias acciones compuestas combinando varias de ellas.

Un primer ejemplo del uso de acciones es **ejemplo_acciones_1.py**:

```
1
2 import cocos
3 from cocos.actions import *
4
5 class MisAcciones(cocos.layer.ColorLayer):
6
7     def __init__(self):
8         super().__init__(64, 200, 255, 255)
9
10        mi_sprite = cocos.sprite.Sprite('mi_ovni_1.png')
11        mi_sprite.position = 320, 240
12        mi_sprite.scale = 1.5
13        self.add(mi_sprite)
14
15        escalar1 =ScaleBy(2, duration = 2)
16        escalar2 =ScaleBy(0.5, duration = 1)
17        mover1 = MoveTo((150,150), 2)
18        mover2 = MoveTo((150,350), 2)
19        mover3 = MoveTo((450,150), 1)
20        rotar1 = RotateBy(180, 2)
21        rotar2 = RotateBy(180, 0.5)
22
23        mi_sprite.do(escalar1 + Reverse(escalar1))
24        mi_sprite.do(Delay(4) + mover1)
25        mi_sprite.do(Delay(6) + rotar1)
26        mi_sprite.do(Delay(8) + mover2)
27        mi_sprite.do(Delay(10) + mover3)
28        mi_sprite.do(Delay(11) + rotar2)
29        mi_sprite.do(Delay(11.5) + mover1)
30        mi_sprite.do(Delay(11.5) + rotar2 * 4)
31        mi_sprite.do(Delay(13.5) + MoveTo((320,240), 1))
32        mi_sprite.do(Delay(13.5) + escalar2)
33        mi_sprite.do(Delay(16.5) + Place((100,100)))
34
35
36 if __name__ == "__main__":
37     cocos.director.director.init(caption = 'Ejemplo 1 de acciones')
38     mi_capa = MisAcciones()
39     mi_escena = cocos.scene.Scene(mi_capa)
40     cocos.director.director.run(mi_escena)
41
```

En un instante dado la salida será como la siguiente:

En este caso creamos una capa llamada MisAcciones con fondo de color azul (en L8 le pasamos los parámetros para conseguirlo) basada en ColorLayer. Incluimos en ella (L13) un sprite que contiene la imagen del ovni[27] (L10), lo escalamos un 50% más grande (L12) y lo colocamos en la mitad de la pantalla (L11). En L15-L21 creamos una serie de acciones[28] que posteriormente (L23-33) aplicaremos al sprite. Las líneas L37-40 son las habituales de inicializar el director, crear una escena con la capa y hacer que ésta se ejecute.

Es interesante analizar la sintaxis de las acciones, y el uso de operadores (especialmente '+' en este primer ejemplo) .

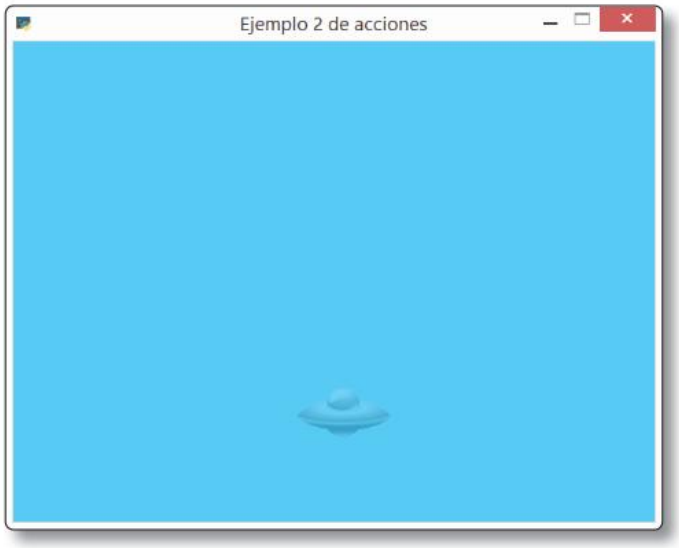
Sigamos viendo el uso de acciones (**ejemplo_acciones_2.py**):

27 Lo denominaré en todo el libro (por comodidad) "ovni" aunque lo correcto sería "OVNI".

28 Hemos importado en la L3 todas las acciones del módulo actions. Para saber más sobre él consultar el Apéndice B.

```
import cocos
from cocos.actions import *

class MisAcciones(cocos.layer.ColorLayer):

    def __init__(self):
        super().__init__(64, 200, 255, 255)

        mi_sprite = cocos.sprite.Sprite('mi_ovni_1.png')
        mi_sprite.position = 320, 240
        mi_sprite.scale = 1.5
        self.add(mi_sprite)

        mover1 = MoveBy((250, 0), 1.5)
        mover2 = MoveTo((580, 50), 2)
        mover3 = MoveBy((-250, 0), 2)
        mover4 = MoveBy((0, 250), 2)

        saltar1 = JumpTo((70,240), 100, 10, 8)
        saltar2 = JumpBy((180,150), 50, 3, 3)

        parpadeo = Blink(12, 2)
        difuminado = FadeOut(2)
        aparicion = FadeIn(2)

        mi_sprite.do(mover1)
        mi_sprite.do(Delay(1.5) + saltar1)
        mi_sprite.do(Delay(9.5) + saltar2)
        mi_sprite.do(Delay(12.5) + (parpadeo | mover2))
        mi_sprite.do(Delay(14.5) + (difuminado | mover3))
        mi_sprite.do(Delay(16.5) + (aparicion | mover4))


if __name__ == "__main__":
    cocos.director.director.init(caption = 'Ejemplo 2 de acciones')
    mi_capa = MisAcciones()
    mi_escena = cocos.scene.Scene(mi_capa)
    cocos.director.director.run(mi_escena)

```

Un instante de la salida generada es:



El código es similar al ejemplo anterior pero aplicando otras acciones al sprite. De interés especial es ver cómo se ejecutan en paralelo dos acciones mediante el operador '|'.

Finalizaremos los ejemplos de acciones con **ejemplo_acciones_3.py**:

```
1
2 import cocos
3 from cocos.actions import *
4
5 class MisAcciones(cocos.layer.ColorLayer):
6
7     def __init__(self):
8         super().__init__(64, 200, 255, 255)
9
10        mi_sprite = cocos.sprite.Sprite('mi_ovni_1.png')
11        mi_sprite.position = 590, 240
12        mi_sprite.scale = 1.5
13        self.add(mi_sprite)
14
15        mi_sprite.do(CallFuncS(lambda obj: self.discontinuo(obj, 5, (-100, 0), 1)))
16        mi_sprite.do(Delay(6) + CallFuncS(lambda obj: self.mover_girando(obj, (550, 80), 3, 3)))
17        mi_sprite.do(Delay(9) + AccelDeccel(MoveBy((0,350), 3) | RotateBy(360, 3)))
18        mi_sprite.do(Delay(12) + Accelerate((MoveBy((-450,0), 3) | RotateBy(360, 3)), 4))
19        mi_sprite.do(Delay(16) + (MoveBy((0,-400), 3) | FadeOut(3)))
20
21    def discontinuo(self, obj, n_pasos, vector, tiempo):
22        coor_x = obj.x
23        coor_y = obj.y
24        for i in range(1, n_pasos + 1):
25            obj.do(Delay(i * tiempo) + Place((coor_x + i*vector[0], coor_y)))
26
27    def mover_girando(self, obj, coor, n_giros, tiempo):
28        obj.do(MoveTo((coor[0], coor[1]), tiempo) | RotateBy(360 * n_giros, tiempo))
29
30
31 if __name__ == "__main__":
32     cocos.director.director.init(caption = 'Ejemplo 3 de acciones')
33     mi_capa = MisAcciones()
34     mi_escena = cocos.scene.Scene(mi_capa)
35     cocos.director.director.run(mi_escena)
36
```

Su salida en algún momento será como en la siguiente imagen:

En este tercer ejemplo sobre acciones tenemos varios elementos novedosos respecto a los dos anteriores:

- Creamos el método discontinuo() en la clase MisAcciones. Lo usaremos para realizar una serie de saltos instantáneos de una determinada longitud en un periodo de tiempo dado. El número de saltos estará indicado por el parámetro n_pasos, la longitud y dirección del salto por vector y el tiempo en el que se realizan todos los saltos por el parámetro del mismo nombre. En el parámetro obj pasamos el objeto sobre el que se realiza la acción, es decir, nuestro ovni.

- Creamos el método mover_girando() en la clase MisAcciones, que nos permitirá desplazar el ovni a la vez que gira. Con el parámetro coor indicaremos el punto de llegada, con n_giros proporcionamos el número de giros de 360 grados que daremos y con tiempo el intervalo en el que transcurre todo.

- Para hacer la llamada a los dos métodos indicados usaremos la acción CallFuncS junto con la función lambda. Notar que CallFuncS nos proporciona como parámetro el objeto que ejecuta la acción (en nuestro caso el ovni), lambda lo recibe y llama a los métodos con los parámetros deseados.

1.2.3 Eventos

Los **eventos** que conoceremos a continuación son los generados en la ventana principal de nuestra aplicación (director.window) y recibidos y tratados en una escena o una capa. En el caso de la escena, cuando ésta está activa, se permite a sus capas "escuchar" los eventos de la ventana principal, algo que hará (siguiendo el árbol de capas de la escena) la que tenga el atributo de clase is_event_handler con valor True. Cuando la escena se inactiva se pierde la capacidad de escucha de sus capas. Los tipos de evento más habituales que se pueden generar en la ventana principal son:

- **on_activate**()
  La ventana fue activada.

- **on_close**()
  El usuario intentó cerrar la ventana.

- **on_deactivate**()
  La ventana fue desactivada.

- **on_draw**()

  El contenido de la ventana fue redibujado.

- **on_hide**()

  La ventana fue ocultada.

- **on_key_press**(symbol, modifiers)

  Se pulsó una tecla del teclado.

  Nos envía la tecla pulsada (symbol) y los modificadores de tecla (modifiers), ambos en forma de.

- **on_key_release**(symbol, modifiers)

  Una tecla del teclado se soltó.

  Nos envía la tecla pulsada (symbol) y los modificadores de tecla (modifiers), ambos en forma de números enteros.

- **on_mouse_drag**(x, y, dx, dy, buttons, modifiers)

  El ratón se movió con uno o más de sus botones pulsados.

  Nos envía las coordenadas[29] (x e y) en píxeles, las distancias relativas desde la posición previa (dx y dy), los botones pulsados[30] (buttons) y los modificadores de tecla (modifiers).

- **on_mouse_enter**(x, y)

  El puntero del ratón entró en la ventana.

  Nos envía las coordenadas en píxeles (x e y).

- **on_mouse_leave**(x, y)

  El puntero del ratón abandonó la ventana.

  Nos envía las coordenadas en píxeles (x e y).

- **on_mouse_motion**(x, y, dx, dy)

  El ratón se movió sin estar pulsados ninguno de sus botones.

  Nos envía las coordenadas[31] (x e y) en píxeles, además de las distancias relativas desde la posición previa (dx y dy).

---

29 Se toma como origen la esquina inferior izquierda de la ventana.

30 También nos lo dará como un número entero.

31 Se toma como origen la esquina inferior izquierda de la ventana.

- **on_mouse_press**(x, y, button, modifiers)

  Un botón del ratón fue pulsado.
  Nos envía las coordenadas[32] (x e y) en píxeles, los botones pulsados (buttons) y los modificadores de tecla (modifiers).

- **on_mouse_release**(x, y, button, modifiers)

  Un botón del ratón fue soltado.
  Nos envía las coordenadas[33] (x e y) en píxeles, los botones pulsados (buttons) y los modificadores de tecla (modifiers).

- **on_mouse_scroll**(x, y, scroll_x, scroll_y)

  La rueda del ratón fue movida.
  Nos envía las coordenadas[34] (x e y) en píxeles y el número de clics en el eje x (scroll_x) o y (scroll_y), pudiendo ser estos dos últimos números negativos.

- **on_move**(x, y)

  La ventana fue movida.
  Nos envía la distancia desde la parte izquierda de la ventana a la parte izquierda de la pantalla (x) y la distancia desde la parte superior de la ventana hasta la parte superior de la pantalla (y).

- **on_resize**(width, height)

  La ventana fue redimensionada.
  Nos envía los nuevos ancho y alto de la pantalla (width y height respectivamente).

- **on_show**()

  La ventana fue mostrada.

- **on_text**(text)

  El usuario introdujo algún texto.
  Nos envía el texto introducido por el usuario.

- **on_text_motion**(motion)

  El usuario movió el cursor de entrada de texto.

  Nos envía la dirección del movimiento.

---

32 Se toma como origen la esquina inferior izquierda de la ventana.

33 Se toma como origen la esquina inferior izquierda de la ventana.

34 Se toma como origen la esquina inferior izquierda de la ventana.

- **on_text_motion_select**(motion)

  El usuario movió el cursor de entrada de texto mientras extendía la selección.

  Nos envía la dirección del movimiento de selección.

Para manejar cualquiera de estos eventos solo tendríamos que crear unos métodos del mismo nombre en nuestra capa.

Mencionar además que disponemos de una serie de manejadores por defecto para la pulsación de las siguientes combinaciones de teclas:

- **CTRL + f**

  Activa/desactiva el modo de pantalla completa.

- **CTRL + s**

  Toma una foto de la ventana y la guarda en un fichero de nombre "screenshot-xxxxx" en el directorio de trabajo actual.

- **CTRL + p**

  Activa/desactiva el modo pausa.

- **CTRL + w**

  Activa/desactiva el modo wire-frame, donde se representan determinados elementos en una especie de esqueleto.

- **CTRL + x**

  Activa/desactiva el modo FPS, donde se nos representan por pantalla el número de frames (fotogramas) por segundo que tenemos en cada momento.

- **CTRL + i**

  Activa/desactiva el modo intérprete incorporado de Python, que nos puede ser útil en tareas de depuración.

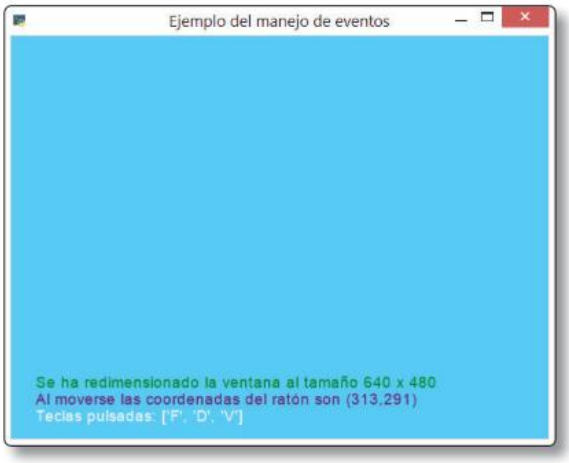
Especialmente útil puede ser CTRL+f para adaptar todos los códigos que aparecen en el libro a la resolución máxima de pantalla de la que disponga el lector. CTRL+i nos pueden ayudar en un momento dado a inspeccionar elementos del código para comprobar su correcto funcionamiento.

Conoceremos un poco más cómo funcionan los eventos con **ejemplo_eventos.py**:

```
1
2 import cocos
3 import pyglet
4 from cocos.director import director
5
6 class EventosTeclado(cocos.layer.ColorLayer):
7     is_event_handler = True
8
9     def __init__(self):
10        super().__init__(64, 200, 255, 255)
11        self.texto = cocos.text.Label('', x = 30, y = 20,
12                                      font_name = 'arial', font_size = 14)
13        self.teclas_pulsadas = set()
14        self.actualiza_texto()
15        self.add(self.texto)
16
17    def actualiza_texto(self):
18        nombres_teclas = [pyglet.window.key.symbol_string(k)
19                          for k in self.teclas_pulsadas]
20        self.texto.element.text = 'Teclas pulsadas: {}'.format(nombres_teclas)
21
22    def on_key_press(self, key, modifiers):
23        self.teclas_pulsadas.add(key)
24        self.actualiza_texto()
25
26    def on_key_release(self, key, modifiers):
27        self.teclas_pulsadas.discard(key)
28        self.actualiza_texto()
29
30
31 class EventosRaton(cocos.layer.Layer):
32     is_event_handler = True
33
34     def __init__(self):
35         super().__init__()
36         self.texto = cocos.text.Label('', font_name = 'arial', font_size=14,
37                                       x=30, y=40, color =(100, 0, 100, 255))
38         self.add(self.texto)
39
40     def actualiza_texto(self, x, y, texto):
41         self.texto.element.text = texto
42
43     def on_mouse_leave(self,x,y):
44         self.actualiza_texto(x, y, 'Se ha salido de la ventana')
45
46     def on_mouse_motion(self, x, y, dx, dy):
47         self.actualiza_texto(x, y, 'Al moverse las coordenadas '
48                                    'del ratón son ({},{})'.format(x,y))
49
50     def on_mouse_drag(self, x, y, dx, dy, buttons, modifiers):
51         self.actualiza_texto(x, y, 'Al arrastrar con el botón {} las coordenadas'
52                                    ' del ratón son ({},{})'.format(buttons,x,y))
53
54     def on_mouse_release(self, x, y, buttons, modifiers):
55         self.actualiza_texto(x, y, 'Se ha pulsado el botón número {}'
56                                    ' del ratón'.format(buttons))
57
58     def on_mouse_scroll(self, x, y, scroll_x, scroll_y):
59         if scroll_y == 1.0:
60             self.actualiza_texto(x, y, 'Se ha movido la rueda'
61                                        ' del ratón hacia adelante')
62         elif scroll_y == -1.0:
63             self.actualiza_texto(x, y, 'Se ha movido la rueda'
64                                        ' del ratón hacia atrás')
65
```

```
66
67 class OtrosEventos(cocos.layer.Layer):
68     is_event_handler = True
69
70     def __init__(self):
71         super().__init__()
72         self.texto = cocos.text.Label('', font_name = 'arial', font_size=14,
73                                       x=30, y=60, color =(0, 120, 0, 255))
74         self.add(self.texto)
75
76     def actualiza_texto(self, x, y, texto):
77         self.texto.element.text = texto
78
79     def on_move(self,x,y):
80         self.actualiza_texto(x, y,'Se ha movido la ventana a las '
81                                   'coordenadas ({},{})'.format(x,y))
82
83     def on_resize(self,width, height):
84         self.actualiza_texto(None, None, 'Se ha redimensionado la ventana '
85                                          'al tamaño {} x {}'.format(width,height))
86
87
88 if __name__ == "__main__":
89     director.init(resizable=True, caption = 'Ejemplo del manejo de eventos')
90     director.run(cocos.scene.Scene(EventosTeclado(), EventosRaton(), OtrosEventos()))
91
```

La salida será similar a la siguiente:



Tenemos una escena con tres capas (dos basadas en Layer y una en ColorLayer):

- En EventosTeclado definimos los métodos on_key_press() y on_key_release(), que se ejecutan cuando se presiona o se suelta una tecla. Almacenamos las teclas pulsadas (en forma de números enteros) en el atributo teclas_pulsadas, un conjunto al que iremos añadiendo o quitando elementos dependiendo del caso. Usaremos la función symbol_string() del módulo pyglet.window.key para obtener, en base a un número entero, el símbolo de la tecla pulsada.

- En EventosRaton definimos los métodos que tienen que ver con el ratón, que se ejecutarán cuando ocurran los eventos correspondientes.
- En OtrosEventos definimos los métodos on_move() y on_resize(), que actuarán cuando la ventana se mueva o redimensione.

Las tres capas tienen el atributo de clase is_event_handler con valor True para poder manejar los eventos, y una línea de texto de distinto color (actualizada mediante el método actualiza_texto() de cada capa) con la que informaremos sobre ellos.

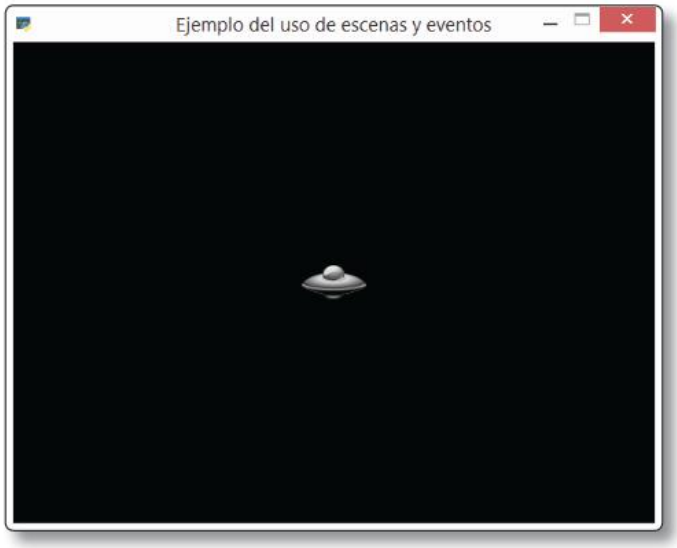
Como nota decir que la documentación de cocos2d nos previene sobre la posibilidad de que el método on_mouse_press() pueda no funcionar de forma correcta, como ha sido mi caso particular[35]. Es el motivo por el que he usado en su lugar on_mouse_release(), que ha funcionado correctamente.

En este punto podríamos combinar la creación de dos escenas con el uso del evento de pulsación del botón del ratón para pasar de una a otra. El código es **ejemplo_escenas_eventos.py**:

```
1
2  import cocos
3
4  class VerTexto(cocos.layer.Layer):
5      is_event_handler = True
6      def __init__(self):
7          super().__init__()
8          mi_etiqueta = cocos.text.Label('Escena 1',
9                                         font_name = 'Consolas',
10                                        font_size = 18,
11                                        anchor_x = 'center', anchor_y = 'center')
12
13         mi_etiqueta.position = (320, 240)
14         self.add(mi_etiqueta)
15
16     def on_mouse_release(self, x, y, buttons, modifiers):
17         cocos.director.director.replace(cocos.scene.Scene(VerSprite()))
18
19
20 class VerSprite(cocos.layer.Layer):
21     is_event_handler = True
22     def __init__(self):
23         super().__init__()
24         mi_sprite = cocos.sprite.Sprite('mi_ovni_1.png', (320,240) )
25         self.add(mi_sprite)
26
27     def on_mouse_release(self, x, y, buttons, modifiers):
28         cocos.director.director.replace(cocos.scene.Scene(VerTexto()))
29
30
31 if __name__ == "__main__":
32     cocos.director.director.init(caption = 'Ejemplo del uso de escenas y eventos')
33     cocos.director.director.window.set_location(600,200)
34     escena_1 = cocos.scene.Scene(VerTexto())
35     cocos.director.director.run(escena_1)
36
```

35 Puede que al lector sí le funcione bien, que sería evidentemente lo ideal.

Haciendo clic con el ratón alternaremos la visualización de la escena 1 (contiene un texto indicando que estamos en ella) o la escena 2 (incluye el sprite del ovni). Una instantánea de la salida es:

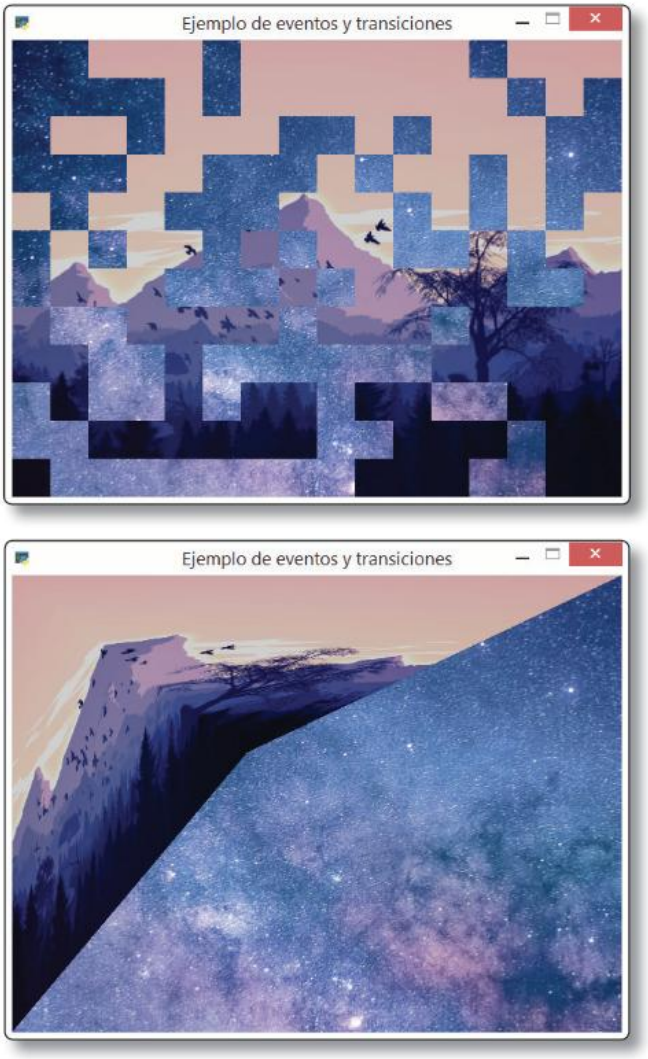


Ambas capas contienen ahora el método on_mouse_release(), que reemplaza la escena actual por la que contiene la otra capa. En este ejemplo he colocado, mediante la función set_location() de la línea L33, la ventana en las coordenadas (600,200) de la pantalla[36].

Mezclaremos ahora el uso de eventos con la transición entre escenas, para hacer un repaso bastante completo[37] de las opciones que tenemos con estas últimas. Dispondremos de dos escenas, en la primera tendremos la imagen de una montaña y en la segunda del universo. Mediante el clic de cualquier botón del ratón se realizará una transición entre ambas escenas. Dos ejemplos de ellas son los siguientes:

36 El lector puede variar estos valores para adecuarlos mejor a su resolución particular de pantalla, o si lo desea activar la opción de pantalla completa (mediante CTRL+f, algo válido para prácticamente todos los códigos que veamos a continuación).

37 Se visualizan casi todas las presentes en el módulo cocos.scenes.transitions.



El código está en **ejemplo_eventos_transiciones.py**:

```
1
2 import cocos
3 from cocos.director import director
4 from cocos.scene import Scene
5 from cocos.sprite import Sprite
6 from cocos.scenes.transitions import *
7
8 class Capa1(cocos.layer.Layer):
9     is_event_handler = True
10    contador = 0
11    def __init__(self):
12        super().__init__()
13        mi_sprite = Sprite('mi_fondo.png', (320,240) )
14        self.add(mi_sprite)
```

```
15
16      def on_mouse_release(self, x, y, buttons, modifiers):
17          if Capa1.contador == 0:
18              Capa1.contador += 1
19              director.replace(RotoZoomTransition(Scene(Capa2()), 3, self))
20          elif Capa1.contador == 1:
21              Capa1.contador += 1
22              director.replace(SlideInBTransition(Scene(Capa2()), 2, self))
23          elif Capa1.contador == 2:
24              Capa1.contador += 1
25              director.replace(TurnOffTilesTransition(Scene(Capa2()), 2, self))
26          elif Capa1.contador == 3:
27              Capa1.contador += 1
28              director.replace(EnvelopeTransition(Scene(Capa2()), 2, self))
29          elif Capa1.contador == 4:
30              Capa1.contador += 1
31              director.replace(JumpZoomTransition(Scene(Capa2()), 2, self))
32          elif Capa1.contador == 5:
33              Capa1.contador += 1
34              director.replace(FadeDownTransition(Scene(Capa2()), 2, self))
35          elif Capa1.contador == 6:
36              Capa1.contador += 1
37              director.replace(CornerMoveTransition(Scene(Capa2()), 2, self))
38          elif Capa1.contador == 7:
39              Capa1.contador += 1
40              director.replace(MoveInLTransition(Scene(Capa2()), 2, self))
41          elif Capa1.contador == 8:
42              Capa1.contador += 1
43              director.replace(FlipX3DTransition(Scene(Capa2()), 2, self))
45
46  class Capa2(cocos.layer.Layer):
47      is_event_handler = True
48      def __init__(self):
49          super().__init__()
50          mi_sprite = Sprite('universo_original.jpg', (320,240) )
51          self.add(mi_sprite)
52
53      def on_mouse_release(self, x, y, buttons, modifiers):
54          if Capa1.contador == 1:
55              director.replace(FlipAngular3DTransition(Scene(Capa1()), 2, self))
56          elif Capa1.contador == 2:
57              director.replace(ShuffleTransition(Scene(Capa1()), 2, self))
58          elif Capa1.contador == 3:
59              director.replace(ShrinkGrowTransition(Scene(Capa1()), 2, self))
60          elif Capa1.contador == 4:
61              director.replace(SplitColsTransition(Scene(Capa1()), 2, self))
62          elif Capa1.contador == 5:
63              director.replace(MoveInRTransition(Scene(Capa1()), 2, self))
64          elif Capa1.contador == 6:
65              director.replace(FlipY3DTransition(Scene(Capa1()), 2, self))
66          elif Capa1.contador == 7:
67              director.replace(FadeTRTransition(Scene(Capa1()), 2, self))
68          elif Capa1.contador == 8:
69              director.replace(MoveInTTransition(Scene(Capa1()), 2, self))
70          elif Capa1.contador == 9:
71              Capa1.contador = 0
72              director.replace(SplitRowsTransition(Scene(Capa1()), 2, self))
73
74
75  if __name__ == "__main__":
76      director.init(caption = 'Ejemplo de eventos y transiciones')
77      director.window.set_location(600,200)
78      director.run(Scene(Capa1()))
79
```

Su análisis queda como ejercicio para el lector.

También podremos usar los eventos para desplazar mediante teclado un determinado sprite por pantalla. El código es **ejemplo_movimiento_esfera.py**:

```
1
2  import cocos
3  from pyglet import image
4  from pyglet.window import key
5  from collections import defaultdict
6
7  class Actor(cocos.sprite.Sprite) :
8      def __init__(self, imagen, x, y) :
9          super().__init__(image.load(imagen))
10         self.position = (x, y)
11
12 class MiCapa(cocos.layer.Layer) :
13     is_event_handler = True
14     def __init__(self) :
15         super().__init__()
16         self.mi_actor = Actor('mi_esfera.png', 320, 240)
17         self.add(self.mi_actor)
18         self.velocidad = 100.0
19         self.teclas_pulsadas = defaultdict(int)
20         self.schedule(self.update)
21
22     def on_key_press(self, k, m) :
23         self.teclas_pulsadas[k] = 1
24     def on_key_release(self, k, m) :
25         self.teclas_pulsadas[k] = 0
26
27     def update(self, dt) :
28         x = self.teclas_pulsadas[key.RIGHT] - self.teclas_pulsadas[key.LEFT]
29         y = self.teclas_pulsadas[key.UP] - self.teclas_pulsadas[key.DOWN]
30         if x != 0 or y != 0:
31             posicion = self.mi_actor.position
32             nueva_x = posicion[0] + self.velocidad * x * dt * 2
33             nueva_y = posicion[1] + self.velocidad * y * dt * 2
34             self.mi_actor.position = (nueva_x, nueva_y)
35
36 if __name__ == '__main__':
37     cocos.director.director.init(caption='Movimiento de esfera mediante teclado')
38     mi_escena = cocos.scene.Scene(MiCapa())
39     cocos.director.director.run(mi_escena)
40
```

La salida será la mostrada a continuación:

En este caso hemos usado en L19 la clase defaultdict del módulo collections, que es una subclase de dict que tiene la particularidad de llamar a una función si indicamos una clave que no existe. Con defaultdict(int) llamará a int() y por tanto nos devolverá un 0. Nada más comenzar el programa, el L28-29 llamará al atributo teclas_pulsadas con 4 llaves que no tenemos aún inicializadas, y en ese instante lo hará a valor 0. Posteriormente, cuando pulsemos una tecla, el método on_key_press() colocará a 1 el valor asociado a ella en teclas_pulsadas. En el caso de dejar de pulsar la tecla, el método on_key_release() asociará un 0 a la tecla. Los valores 0 o 1 asociados a las pulsaciones de las teclas son usados posteriormente (en L28-29) para dar valor 1, 0 o -1 a las variables x e y (indican la dirección en ambos ejes), que posteriormente se usará el L32-33 para calcular la nueva posición del sprite.

Si queremos limitar el movimiento de la esfera a la pantalla sustituiremos la línea 34 por las siguientes:

```
34            if self.mi_actor.width / 2 < nueva_x < 640 - self.mi_actor.width /2 and\
35               self.mi_actor.height / 2 < nueva_y < 480 - self.mi_actor.height /2:
36                    self.mi_actor.position = (nueva_x, nueva_y)
```

Si hubiésemos querido crear un código más genérico podríamos haber obtenido las dimensiones en píxeles de la pantalla mediante los atributos width y height de director.window.

El código resultante lo guardaremos en **ejemplo_movimiento_esfera_limitado.py**.

1.2.4 Modelo de colisión

Hasta este momento hemos aprendido a crear sencillas aplicaciones donde hacíamos uso de los conceptos fundamentales de cocos2d (escenas, capas, acciones, eventos) tratando solamente con un sprite. En los videojuegos tendremos varios, y nos preguntaremos cosas como las siguientes:

- ¿Hay algún enemigo que toca al jugador?
- ¿Hay algún enemigo o disparo cerca del jugador?
- ¿Qué enemigo está más cerca del jugador?
- ¿Hay algún actor bajo el cursor del ratón?

El módulo **collision_model** (modelo de colisión) nos permitirá encontrar respuestas a esas preguntas.

Los actores tienen por lo general una forma irregular, por lo que de manera ideal se deberían tener en cuenta (por ejemplo para saber si se tocan dos de ellos) todos los píxeles que lo componen, algo nada práctico ya que su cómputo ralentizaría mucho la ejecución del código. En su lugar se asocia una forma geométrica simple a cada actor de cara a calcular sus posibles colisiones, y posteriormente se analiza si esas formas se superponen. Las posibles formas geométricas que tenemos en cocos2d son:

- Círculos.
- Rectángulos (con los lados paralelos a los ejes).

Para que se puedan considerar posibles colisiones en un objeto es **obligatorio** que tenga un atributo llamado **cshape** que sea una instancia de las clases **CircleShape** (para los círculos) o **AARectShape** (para los rectángulos)[38].

El manejador de colisión (**collision manager**) deberá llevar un control sobre los objetos que puedan colisionar. Tendrá por tanto que conocerlos y almacenarlos, para posteriormemte indicarnos la información que precisemos. Pensemos en cuando preguntamos qué enemigo está más cerca de nuestro jugador. Habrá una serie de enemigos que incluiremos dentro de esa categoría de "conocidos" de cara a una elección. Crearemos una instancia de la clase **CollisionManager** (o que implemente su interfaz) para generar nuestro manejador de colisión. Tiene los siguientes métodos para añadir o eliminar objetos:

- **add**(obj)
  Añade obj a los objetos conocidos.
- **remove_tricky**(obj)
  Elimina obj de los objetos conocidos[39].
- **clear**()
  Se eliminan todos los objetos conocidos que pudiese haber en el manejador de colisión.

38 Para más información sobre ellos consultar el módulo cocos.collision_model en el Apéndice B.

39 Para proceder de forma correcta el valor de obj.cshape debe ser el mismo que tenía cuando añadimos obj mediante el método add().

También tenemos los siguientes métodos útiles para depuración y testeo:

- **knows**(obj)
  Indica si el objeto obj es conocido o no.

- **known_objs**()
  Nos proporciona todos los objetos conocidos.

Obtener respuestas satisfactorias del manejador de colisión requiere que los objetos conocidos tengan el mismo valor de cshape en el momento de la pregunta y en el momento en que se añadieron, algo que podremos logar de dos maneras[40]:

- Haciendo mc.remove_tricky(obj) → obj.update_cshape() → mc.add(obj) en el momento en que obj necesite actualizar su atributo cshape.

- Haciendo, en cada frame, mc.clear() → actualizar los atributos cshape de todos los objetos colisionables → añadir todos los objetos colisionables a mc → seguir con la lógica del juego relativa a la colisión del actor.

La primera forma es lenta, y sólo aceptable si pocos elementos colisionables deben actualizar su atributo cshape en cada frame.

La segunda forma es la adecuada cuando la mayoría de los elementos colisionables deben actualizar su atributo cshape en cada frame, y por lo tanto será la que usemos habitualmente.

En algunos casos puede ser conveniente tener dos manejadores de colisión, uno para actores que raramente cambian su cshape (por ejemplo comida, armas o monedas) y otro para los que los cambian en cada frame (como pueden ser el jugador y los enemigos).

Una vez que tengamos el manejador de colisión con todos los elementos cargados usaremos los siguientes métodos para obtener información sobre colisiones:

- **any_near**(obj, near_distance)
  Devuelve None si ningún objeto conocido (salvo él mismo) está más cerca que la distancia near_distance, o un objeto que sí lo esté. El objeto obj no tiene por qué ser un objeto conocido.

---

40 Consideramos que mc es una instancia de CollisionManager, es decir, el manejador de colisión. Con el método update_cshape() simbolizamos la actualización del atributo cshape del objeto.

- **iter_all_collisions**()

  Nos devuelve un iterador que identifica todas las colisiones entre objetos conocidos. En cada iteración nos devolverá una tupla (obj1, obj2) donde obj1 y obj2 son los objetos que colisionan. Si aparece (obj1, obj2) no lo hará (obj2, obj1) dado que ya habría sido indicada.

- **iter_colliding**(obj)

  Nos devuelve iterador sobre los objetos que colisionan con el objeto obj, donde obj no tiene por qué ser un objeto conocido.

- **objs_colliding**(obj)

  Nos devuelve un contenedor con los objetos conocidos que se superponen al objeto obj (sin incluirse él, que no debe ser obligatoriamente un objeto conocido).

- **objs_into_box**(minx, maxx, miny, maxy)

  Devuelve un contenedor con los objetos conocidos que caben por completo en el rectángulo alineado con los ejes determinado por los parámetros minx, maxx, miny y maxy.

- **objs_near**(obj, near_distance)

  Devuelve un contenedor con los objetos conocidos que están a una distancia menor o igual a near_distance del objeto obj (algo que incluye por supuesto a los que colisionan con él), sin incluir el propio obj (no requiriéndose que éste sea un objeto conocido).

- **objs_near_wdistance**(obj, near_distance)

  Devuelve una lista de tuplas (objeto_conocido, distancia) que cumplen que la distancia de objeto_conocido a obj es menor o igual que near_distance. Eso incluye los que colisionan con él. No se requiere que obj sea un objeto conocido. Si necesitásemos que esa lista estuviese ordenada ascendentemente por las distancias usaríamos el método ranked_objs_near().

- **objs_touching_point**(x, y)

  Devuelve un contenedor con los objetos conocidos que tocan el punto (x,y).

- **ranked_objs_near**(obj, near_distance)

  Igual que objs_near_wdistance pero ordenando ascendentemente la lista teniendo como referencia las distancias.

- **they_collide**(obj1, obj2)

  Devuelve booleano indicando si los objetos obj1 y obj2 se superponen. Ninguno de ellos es obligatorio que sean objetos conocidos.

Hay dos clases que implementan la interfaz CollisionManager:

- **CollisionManagerBruteForce**, que lo hace con el menor código posible. Su rendimiento es bajo, por lo que se usa generalmente como referencia, en ejemplos sencillos, o para depuración.

- **CollisionManagerGrid**, basado en el esquema denominado "spatial hashing"[41], que se fundamenta en dividir el plano en rectángulos de un determinado ancho y alto, teniendo una tabla que indique qué objetos se superponen a cada rectángulo. Posteriormente, cuando hagamos una consulta referente al objeto obj, sólo examinaremos los objetos que superponen los rectángulos sobre los que está ubicado obj, o los que están a una cierta distancia.

  Su sintaxis es la siguiente:

  ```
  CollisionManagerGrid(xmin, xmax, ymin, ymax, cell_width, cell_height)
  ```

  Los cuatro primeros parámetros nos delimitan el plano en el que será efectivo el manejador de colisiones, y los dos siguientes nos marcan la anchura y altura de los rectángulos (o celdas) en que dividimos ese plano.

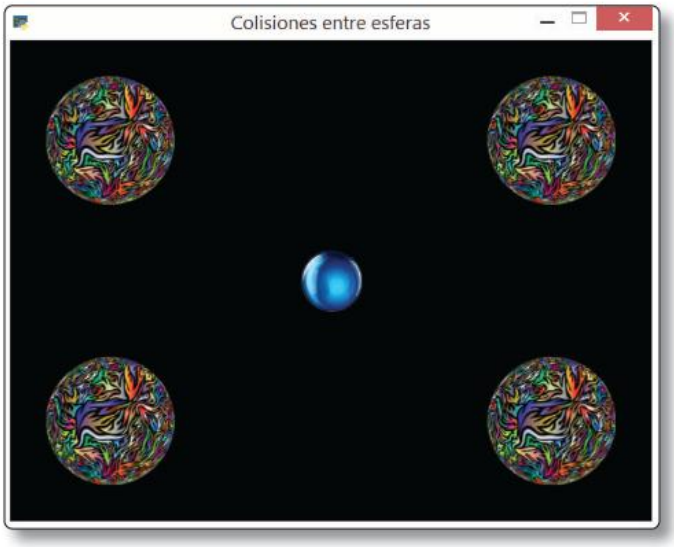
Para profundizar un poco más en todos los aspectos relacionados con el modelo de colisión, consultar el módulo cocos.collision_model en el Apéndice B.

Veamos a continuación un sencillo ejemplo de manejador de colisiones en el cual moveremos la esfera vista con anterioridad, teniendo además 4 de ellas más en los vértices de un cuadrado imaginario. Detectaremos la colisión entre la esfera móvil y cualquiera de las otras cuatro. Cuando eso ocurra se eliminará la esfera fija correspondiente. El código (**ejemplo_manejador_colision_esfera.py**) es el siguiente:

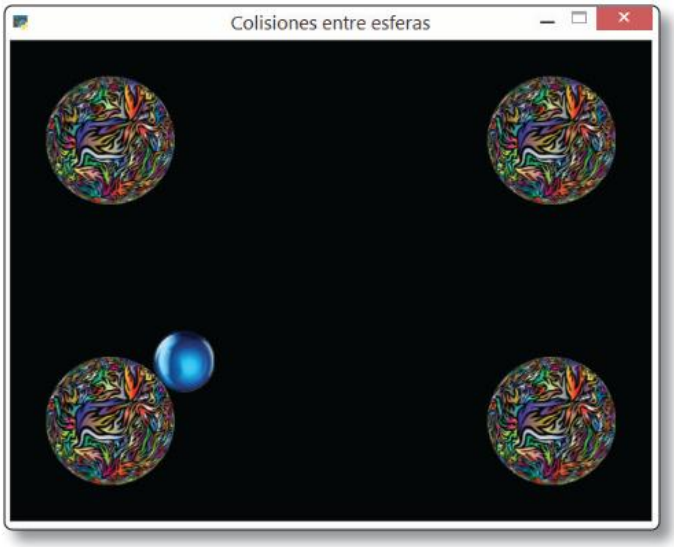
41 Podríamos traducirlo como "cálculo espacial".

```
1
2 import cocos
3 import cocos.collision_model as cm
4 import cocos.euclid as eu
5 from pyglet import image
6 from pyglet.window import key
7 from collections import defaultdict
8
9 class Actor(cocos.sprite.Sprite) :
10     def __init__(self, imagen, x, y) :
11         super().__init__(image.load(imagen))
12         self.position = (x, y)
13         self.cshape = cm.CircleShape(eu.Vector2(x, y), self.width/2)
14
15
16 class MiCapa(cocos.layer.Layer) :
17     is_event_handler = True
18     def __init__(self) :
19         super().__init__()
20         self.esfera_movil = Actor('mi_esfera.png', 320, 240)
21         self.add(self.esfera_movil)
22         for pos in [(100, 100), (540, 380), (540, 100), (100, 380)]:
23             a = Actor('mi_esfera_2.png', pos[0] , pos[1])
24             self.add(a)
25         self.manejador_colision = cm.CollisionManagerBruteForce()
26         self.velocidad = 100.0
27         self.teclas_pulsadas = defaultdict(int)
28         self.schedule(self.update)
29
30     def on_key_press(self, k, m) :
31         self.teclas_pulsadas[k] = 1
32     def on_key_release(self, k, m) :
33         self.teclas_pulsadas[k] = 0
34
35     def update(self, dt) :
36         self.manejador_colision.clear()
37         for _, node in self.children:
38             self.manejador_colision.add(node)
39         for other in self.manejador_colision.iter_colliding(self.esfera_movil):
40             self.remove(other)
41
42         x = self.teclas_pulsadas[key.RIGHT] - self.teclas_pulsadas[key.LEFT]
43         y = self.teclas_pulsadas[key. UP] - self.teclas_pulsadas[key.DOWN]
44         if x != 0 or y != 0:
45             pos = self.esfera_movil.position
46             nueva_x = pos[0] + self.velocidad * x * dt * 2
47             nueva_y = pos[1] + self.velocidad * y * dt * 2
48             self.esfera_movil.position = (nueva_x, nueva_y)
49             self.esfera_movil.cshape.center = eu.Vector2(nueva_x, nueva_y)
50
51
52 if __name__ == '__main__':
53     cocos.director.director.init(caption = 'Colisiones entre esferas')
54     mi_escena = cocos.scene.Scene(MiCapa())
55     cocos.director.director.run(mi_escena)
56
```

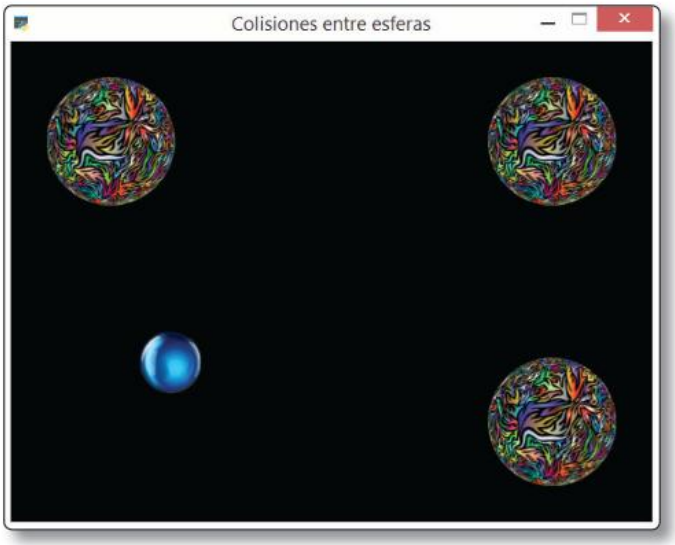
Inicialmente la salida será:



Podremos acercarnos enormemente a cualquiera de las cuatro esferas fijas:



Pero en cuanto la toquemos, desaparecerá:



Comentario general del código:

- Se define la clase Actor, basada en Sprite, que nos permitirá en su inicilización cargar la imagen del Sprite y colocarla en las coordenadas que indiquemos. Además se añade el atributo cshape, obligatorio para incluir la instancia en el manejador de colisiones. En nuestro caso es un círculo (instancia de CircleShape) con las coordenadas del sprite y un radio igual a la mitad de su anchura.
- Se define la clase MiCapa, basada en Layer, que dispondrá de los métodos on_key_press() y on_key_release() para tratar las pulsasiones de teclas, y update() para ejecutar un código en cada frame.

Comentario pormenorizado:

- L2-7: importamos los módulos que necesitaremos. Respecto a ejemplos anteriores destacar la inclusión de collision_model (renombrado como cm) para el manejador de colisión y euclid[42] (renombrado como eu) para tratar posteriormente con vectores (elemento de geometría euclídea).

42 Para saber más sobre el módulo euclid consultar el Apéndice B.

- L17: colocamos el atributo de clase is_event_handler con valor True para que la capa pueda manejar eventos.
- L20-21: creamos (haciendo uso de la clase Actor) la esfera central y la añadimos a MiCapa en las coordenadas (320,240). Una referencia a ella se guarda en el atributo esfera_movil de la capa.
- L22-24: creamos cuatro esferas (usando la clase Actor con otra imagen distinta) colocadas en los vértices de un cuadrado imaginario alrededor de la esfera móvil.
- L25: creamos el manejador de colisión basado en la clase CollisionManagerBruteForce, teniendo una referencia a él en al atributo manejador_colisión de la capa.
- L26-28: creamos los atributos velocidad y teclas_pulsadas, además de programar la ejecución del método update() de la capa mediante el método schedule().
- L30-33: definimos los métodos on_key_press() y on_key_release() que se ejecutarán al pulsar o soltar teclas.
- L36: borramos todos los objetos que tuviésemos en ese momento almacenados en el manejador de colisión.
- L37-38: añadimos todos los objetos de la capa (es decir, las 5 esferas) al manejador de colisión.
- Mediante el método iter_colliding() del manejador de colisión recibimos todos los objetos que colisionan en ese instante con la esfera móvil. Usando un bucle for los recorremos y los vamos eliminando mediante el método remove() de la capa.
- L42-49: es el bloque de código ya conocido usado para actualizar la posición de la esfera móvil al que hemos añadido en la L49 la actualización de las coordenadas de cshape.
- L52-55: de la forma habitual inicializamos el director, creamos una escena con la capa MiCapa y la ejecutamos.

En **ejemplo_manejador_colision_esfera_2.py** tendremos el mismo código pero aplicando CollisionManagerGrid en lugar de CollisionManagerBruteForce, por lo que cambiaremos la línea 25 original por la 25 y 26 del nuevo fichero.

1.2.5 Animaciones, efectos de sonido y música

A continuación aprenderemos a crear **animaciones**, introducir efectos de **sonido** y añadir una **música** de fondo en nuestras escenas.

Ya vimos que los sprites nos permiten representar por pantalla una imagen en un área rectangular, que podremos posteriormente (entre otras cosas) mover, rotar o escalar.

Tenemos la posibilidad de realizar **animaciones** mediante sprites al estilo de los dibujos animados, es decir, reemplazando la imagen lo suficientemente rápido por otra para dar sensación de movimiento. Lo lograremos de las siguientes maneras:

1. Pasando como fuente para la imagen del sprite un objeto de la clase pyglet.image.Animation, que contiene una colección de imágenes individuales que irá mostrando sucesivamente.
2. Usando un GIF[43] animado como fuente para la imagen del sprite.
3. Teniendo un array de imágenes y asignando mediante código la visualización de cada una de ellas en los instantes adecuados.

Para llevar a cabo la primera opción conoceremos tres clases del módulo pyglet.image: ImageGrid, Animation y AnimationFrame.

Mediante **ImageGrid** colocamos sobre una determinada imagen una rejilla imaginaria con la cual poder acceder a cada una de las partes en las que ésta divide a la imagen. La sintaxis es la siguiente:

```
ImageGrid(image, rows, colums)
```

En él image será la imagen sobre la que actuar, mientras que rows y colums indicarán, respectivamente, el número de filas y columnas en los que dividiremos la imagen.

Destacamos los atributos:

- **image**

  La imagen sobre la que actuamos.

- **rows**

  Número de filas en las que dividimos la imagen.

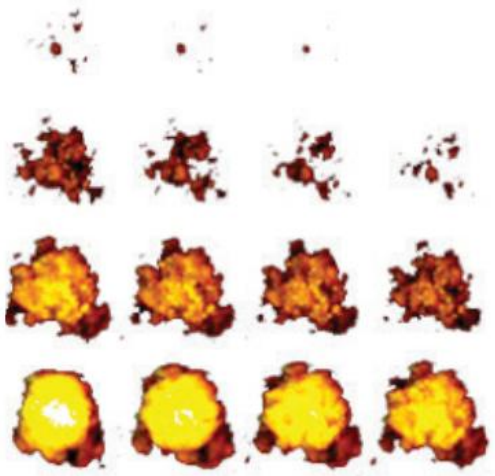
43 Graphics Interchange Format (formato de intercambio de gráficos), es un tipo de gráfico muy utilizado, tanto para imágenes como para animaciones.

- **columns**
  Número de columnas en las que dividimos la imagen.
- **height**
  Alto en píxeles de la imagen que tratamos.
- **width**
  Ancho en píxeles de la imagen que tratamos.

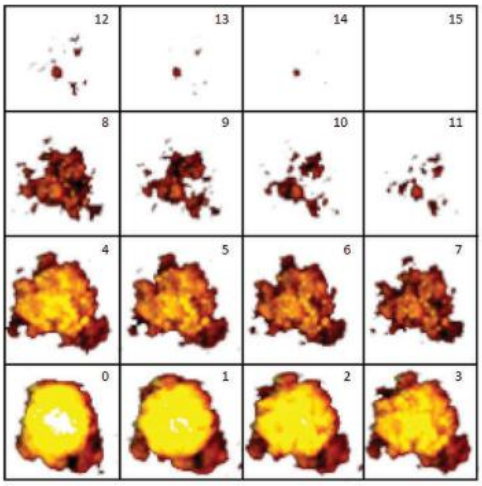
Y los métodos:

- **get_animation**(period, loop=True)
  Nos devolverá un objeto Animation que representará cada frame en un tiempo en segundos indicado por period (un número real). Con el parámetro loop indicamos mediante un booleano si la animación se repite de manera continua (o se detiene) al finalizar todos los frames.
- **get_region**(x, y, width, height)
  Nos devuelve una región rectangular de la imagen. Los parámetros x e y indican las coordenadas de la esquina inferior izquierda de esa región, mientras que width y height marcan la anchura y altura que tendrá.

Si disponemos de una imagen como **secuencia_explosion.png**, que nos muestra 16 instantes de una explosión, podremos dividirla en esos 16 fotogramas (4 filas x 4 columnas) para ir mostrándolos posteriormente. La imagen es:



Un aspecto importante es la numeración de las partes en que dividimos la imagen. En nuestro caso sería de la siguiente manera:



Es decir, se recorren las filas de abajo arriba y las columnas de izquierda a derecha.

La clase **Animation** nos permitirá realizar una animación fotograma a fotograma con un ritmo temporal que indicaremos. Tiene la siguiente sintaxis:

```
Animation(Frames)
```

El parámetro Frames es una lista de los fotogramas (instancias de pyglet.image.AnimationFrame) que componen la animación. Si ningún fotograma de ella tiene una duración de valor None, ésta continúa cíclicamente de forma ininterrumpida. En caso contrario, se detendrá en el fotograma de valor None.

Destacamos los métodos:

- **from_image_sequence**(sequence, period, loop=True)

  Es un método de clase que nos creará la animación basándose en una lista de imágenes y una tasa constante de fotogramas. Con sequence indicamos la lista, con period (un número real) el instante en segundos entre la visualización de un fotograma y otro, y mediante loop (valor booleano) configuramos si la animación se repetirá o no de forma cíclica. Se nos devolverá el objeto Animation correspondiente.

- **get_duration**()

  Nos devolverá un número real indicando la duración en segundos de la animación.

La clase **AnimationFrame** nos valdrá para almacenar cada uno de los fotogramas de la animación. Tendrá la siguiente sintaxis:

```
AnimationFrame(image, duration)
```

En ella image será la imagen contenida y duration un número real que indica el tiempo en segundos que permanece visible.

Veamos a continuación dos ejemplos de la animación de la explosión que tenemos almacenada en la imagen secuencia_explosion.png. En nuestro caso se ejecutará cuando hagamos clic con el ratón en la pantalla.

En el primero (**ejemplo_animacion.py**) usaremos las clases ImageGrid y Animation:

```
1
2 import cocos
3 from pyglet.image import load, ImageGrid, Animation
4
5 class MiCapa(cocos.layer.Layer):
6     is_event_handler = True
7
8     def __init__(self):
9         super().__init__()
10
11    def on_mouse_release(self, x, y, buttons, modifiers):
12        mi_secuencia = ImageGrid(load('secuencia_explosion.png'), 4, 4)
13        mi_animacion = Animation.from_image_sequence(mi_secuencia, 0.05, False)
14        self.mi_sprite = cocos.sprite.Sprite(mi_animacion, (x, y))
15        self.add(self.mi_sprite)
16
17 if __name__ == '__main__':
18     cocos.director.director.init(caption = 'Ejemplo animación 1')
19     mi_escena = cocos.scene.Scene(MiCapa())
20     cocos.director.director.run(mi_escena)
21
```

La salida tras tres clics seguidos en la pantalla será similar a la siguiente, en la que visualizamos distintos instantes de las explosiones:

Comentarios sobre el código:

- En L3 importamos desde el módulo pyglet.image las clases load, ImageGrid y Animation.
- Creamos la capa MiCapa y definimos el método on_mouse_release(), que se ejecutará cuando hagamos clic con el ratón sobre la pantalla. Dentro de él crearemos y colocaremos en la capa cada una de las animaciones.
- En L12 creamos mediante ImageGrid una secuencia de 16 fotogramas (4 filas x 4 columnas) a partir de la imagen secuencia_explosion.png.
- En L13 creamos la animación pasando esa secuencia de fotogramas al método from_image_sequence() de la clase Animation, indicando en los argumentos que la sucesión de fotogramas se hará cada 0.05 segundos y que no habrá repetición de la animación.
- En L14-15 pasamos la animación como argumento del sprite, y añadimos a la capa en las coordenadas donde hemos hecho clic con el ratón.

En el segundo ejemplo usaremos solamente ImageGrid, empleando su método get_animation() para generar la animación. El código es **ejemplo_animacion_2.py**:

```
1
2 import cocos
3 from pyglet.image import load, ImageGrid
4
5 class MiCapa(cocos.layer.Layer):
6     is_event_handler = True
7
8     def __init__(self):
9         super().__init__()
10
11    def on_mouse_release(self, x, y, buttons, modifiers):
12        mi_secuencia = ImageGrid(load('secuencia_explosion.png'), 4, 4)
13        mi_animacion = mi_secuencia.get_animation(0.05, loop = False )
14        self.mi_sprite = cocos.sprite.Sprite(mi_animacion, (x, y))
15        self.add(self.mi_sprite)
16
17 if __name__ == '__main__':
18     cocos.director.director.init(caption = 'Ejemplo animación 2')
19     mi_escena = cocos.scene.Scene(MiCapa())
20     cocos.director.director.run(mi_escena)
21
```

La salida es la misma que en el ejemplo anterior, y el código sólo difiere en L13 y en que no importamos la clase Animation.

Para realizar una animación mediante un gif animado (la segunda forma que indicamos al inicio) veremos la función **animation**() del módulo pyglet.

resource, que nos cargará la animación y podremos realizar sobre ella determinadas transformaciones. Su sintaxis es la siguiente:

```
animation(name, flip_x=False, flip_y=False, rotate=0)¶
```

Los parámetros son:

- **name**
  Cadena en la que indicamos en nombre del fichero de la animación a cargar.

- **flip_x**
  Booleano que indica si a la imagen devuelta se le dará o no la vuelta horizontalmente.

- **flip_y**
  Booleano que indica si a la imagen devuelta se le dará o no la vuelta verticalmente.

- **rotate**
  Número entero (múltiplo de 90) que indica la rotación de la imagen devuelta (en grados y en el sentido de las agujas del reloj).

La función devolverá una instancia de la clase Animation.

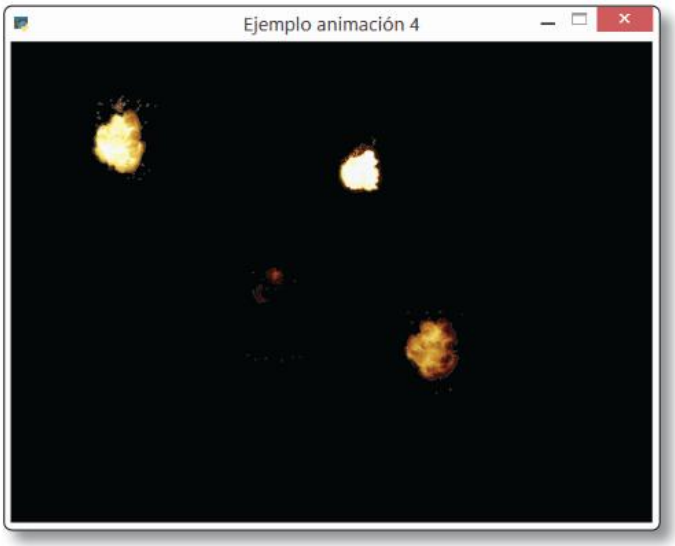
Por lo general un gif animado se reproduce de forma cíclica e ininterrumpida. En el fichero **gif_animado_fuego.gif** tenemos la animación continua de un fuego. Veamos a continuación un ejemplo que coloca una de ellas en cada pulsación del botón del ratón sobre la pantalla (**ejemplo_animacion_3.py**):

```
1
2  import cocos
3  from cocos.sprite import Sprite
4  from pyglet.resource import animation
5
6  class MiCapa(cocos.layer.Layer):
7      is_event_handler = True
8
9      def __init__(self):
10         super().__init__()
11
12     def on_mouse_release(self, x, y, buttons, modifiers):
13             mi_sprite = Sprite(animation('gif_animado_fuego.gif'), (x, y))
14             self.add(mi_sprite)
15
16 if __name__ == '__main__':
17     cocos.director.director.init(caption = 'Ejemplo animación 3')
18     mi_escena = cocos.scene.Scene(MiCapa())
19     cocos.director.director.run(mi_escena)
20
```

Si disponemos de un gif animado que se repite cíclicamente pero queremos que sólo se reproduzca una vez podríamos eliminar el sprite una vez que se han representado todos los fotogramas que componen el gif. Por ejemplo, en **gif_animado_explosion.gif** disponemos de una explosión doble (una más luminosa, otra más oscura). Un código para su reproducción unitaria al hacer clic con el ratón sobre la pantalla es **ejemplo_animacion_4.py**:

```
1
2  import cocos
3  from cocos.actions import CallFunc, Delay
4  from cocos.sprite import Sprite
5  from pyglet.resource import animation
6
7  class MiCapa(cocos.layer.Layer):
8      is_event_handler = True
9
10     def __init__(self):
11         super().__init__()
12
13     def on_mouse_release(self, x, y, buttons, modifiers):
14         mi_animacion = animation('gif_animado_explosion.gif')
15         mi_sprite =Sprite(mi_animacion, (x, y))
16         self.add(mi_sprite)
17         tiempo_animacion = mi_animacion.get_duration()
18         self.do(Delay(tiempo_animacion) + CallFunc(self.elimina_sprite))
19
20     def elimina_sprite(self):
21         if self.children:
22             self.children[0][1].kill()
23
24 if __name__ == '__main__':
25     cocos.director.director.init(caption = 'Ejemplo animación 4')
26     mi_escena = cocos.scene.Scene(MiCapa())
27     cocos.director.director.run(mi_escena)
28
```

La salida podría ser así, con ambas explosiones:

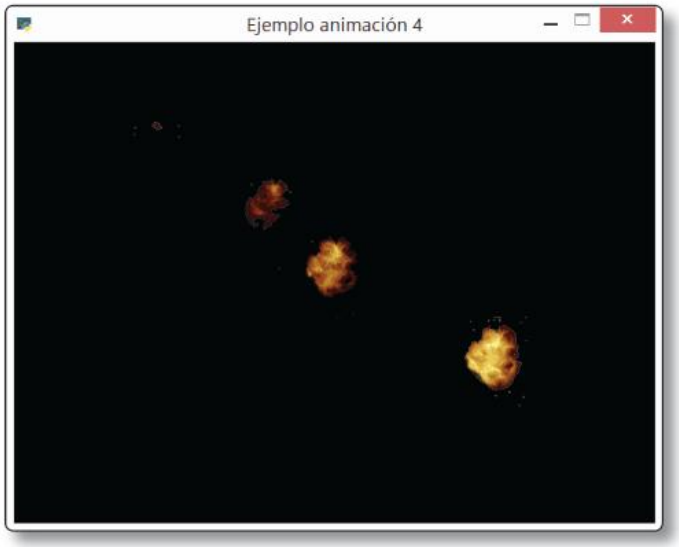


El código es muy similar al ejemplo anterior, con la particularidad de que:

- En L17 obtenemos la duración en segundos de la animación mediante el método get_duration() de la clase Animation[44].
- En L18, tras esperar el tiempo que dura la animación usando la acción Delay, llamamos mediante la acción CallFunc al método elimina_sprite() que hemos creado.
- El método elimina_sprite() comprueba si hay objetos en la capa, y de ser así elimina el primero de ellos mediante el método kill().
- En L22 usamos self.children para acceder a los objetos que tenemos en la capa, obteniendo una lista de tuplas de dos elementos (uno es el nivel y otro el objeto), de ahí la necesidad de un doble índice.

Si en la línea L19 pusiésemos tiempo_animacion/2 en lugar de tiempo_animacion sólo aparecería el primer tipo de explosión. Esa modificación la tendremos en **ejemplo_animacion_4_2.py.** Un ejemplo de su salida es:

44 Recordemos que la función animation() del módulo pyglet.resource nos devuelve un objeto de la clase Animation.



Para añadir efectos de sonido y música nos basaremos por simplicidad únicamente en la clase **Effect** del módulo cocos.audio.effect y en ficheros de sonido en formato **wav**[45]. El tratamiendo del audio nos lo proporcionará la librería pygame, de ahí la necesidad de tenerla instalada.

La sintaxis de Effect, que está basada en Object, es muy sencilla:

```
Effect(filename)
```

La cadena filename nos indica el fichero de audio que cargaremos.

Tiene los siguientes atributos:

- **action**

  Es una acción que ejecuta el fichero de audio.

- **sound**

  Es un objeto de la clase **Sound** (módulo cocos.audio.pygame.mixer) mediante el que podremos configurar, controlar u obtener información del audio cargado.

45 Para profundizar en el tratamiento del audio en cocos2d consultar el conjunto de módulos cocos.audio en la documentación oficial.

Y el método:

- **play**()
  Reproduce el audio cargado.

El objeto Sound almacenado en el atributo sound tiene los siguientes métodos interesantes:

- **fadeout**(time)
  El sonido se irá desvaneciendo en el número de milisegundos marcados por el número entero time.

- **get_length**()
  Nos devuelve un número real indicando la longitud temporal del sonido en segundos.

- **get_volume**()
  Nos devuelve un número real entre 0.0 y 1.0 indicando el volumen al que tenemos configurado el sonido.

- **play**(loops, maxtime)
  Reproduce el sonido cargado. Mediante el parámetro loops (un número entero) indicamos el número de repeticiones del sonido tras la primera reproducción[46], y con maxtime configuramos un tiempo máximo (valor entero dado en milisegundos) tras el cual se detendrá la reproducción de sonido.

- **set_volume**(volume)
  Configura el volumen dando un valor real entre 0.0 y 1.0 al parámetro volume.

- **stop**()
  Detiene la reproducción del sonido.

Tendremos por tanto varias opciones para reproducir el sonido cargado en Effect:

- Mediante el método play() de la clase Effect.

- Ejecutando la acción que contiene el atributo action de la instancia de Effect.

46 Si el valor es 0 no se repite ninguna vez, y si es -1 lo hace de forma ininterrumpida.

- Usando el método play() del objeto Sound almacenado en el atributo sound de la instancia de la clase Effect.

Para el correcto uso de Effect debemos añadir el argumento audio_backend = 'sdl' al inicializar el director[47].

Añadiremos al fichero ejemplo_animacion.py el sonido de una explosión[48] a la vez que se ejecuta la animación de ésta. Se recomienda bajar el volumen de los altavoces antes de su ejecución, ya que el estruendo podría ser alto. Tenemos el código en **ejemplo_animacion_y_sonido.py**:

```
1
2  import cocos
3  from cocos.audio.effect import Effect
4  from pyglet.image import load, ImageGrid, Animation
5
6  class MiCapa(cocos.layer.Layer):
7      is_event_handler = True
8
9      def __init__(self):
10         super().__init__()
11
12     def on_mouse_release(self, x, y, buttons, modifiers):
13         mi_efecto_sonoro = Effect('explosion.wav')
14         mi_efecto_sonoro.sound.set_volume(0.1)
15         mi_efecto_sonoro.play()
16         mi_secuencia = ImageGrid(load('secuencia_explosion.png'), 4, 4)
17         mi_animacion = Animation.from_image_sequence(mi_secuencia, 0.05, False)
18         self.mi_sprite = cocos.sprite.Sprite(mi_animacion, (x, y))
19         self.add(self.mi_sprite)
20
21
22 if __name__ == '__main__':
23     cocos.director.director.init(caption = 'Ejemplo animación y sonido',
24                                  audio_backend='sdl')
25     mi_escena = cocos.scene.Scene(MiCapa())
26     cocos.director.director.run(mi_escena)
27
```

Comentarios sobre el código:

- En L3 importamos Effect del módulo cocos.audio.effect.
- En L13 cargamos el efecto sonoro con el fichero explosion.wav.
- En L14 configuramos el volumen al 10% mediante el método set_volume() de la clase Sound.

47 Recordemos que la librería SDL (sobre la que está construida pygame) nos permite trabajar con elementos multimedia.

48 Almacenada en el fichero explosion.wav en nuestra carpeta.

- En L15 ejecutamos el efecto sonoro mediante el método play() de la clase Effect.
- En L23-24 colocamos el argumento audio_backend con valor 'sdl' al iniciar el director.

Si ahora queremos añadir una música de fondo a nuestra escena haremos también uso de Effect. Veamos cómo en **ejemplo_animacion_y_sonido_2.py**:

```
1
2  import cocos
3  from cocos.audio.effect import Effect
4  from pyglet.image import load, ImageGrid, Animation
5
6  class MiCapa(cocos.layer.Layer):
7      is_event_handler = True
8
9      def __init__(self):
10         super().__init__()
11         global musica_de_fondo
12         musica_de_fondo = Effect('musica_fondo_retro.wav')
13         musica_de_fondo.sound.set_volume(0.1)
14         musica_de_fondo.sound.play(-1)
15
16     def on_mouse_release(self, x, y, buttons, modifiers):
17         sonido_explosion = Effect('explosion.wav')
18         sonido_explosion.sound.set_volume(0.1)
19         sonido_explosion.sound.play()
20         mi_secuencia = ImageGrid(load('secuencia_explosion.png'), 4, 4)
21         mi_animacion = Animation.from_image_sequence(mi_secuencia, 0.05, False)
22         self.mi_sprite = cocos.sprite.Sprite(mi_animacion, (x, y))
23         self.add(self.mi_sprite)
24
25
26 def cierra_ventana():
27     musica_de_fondo.sound.stop()
28     cocos.director.director.window.close()
29
30
31 if __name__ == '__main__':
32     cocos.director.director.init(caption = 'Ejemplo animación y sonido 2',
33                                  audio_backend='sdl')
34     mi_escena = cocos.scene.Scene(MiCapa())
35     cocos.director.director.window.on_close = cierra_ventana
36     cocos.director.director.run(mi_escena)
37
```

Comentarios sobre el código:

- En L11-14 hemos colocado en la reproducción de música (al 10% de volumen) en el método __init__() de la capa, para que se ejecute nada más inicializarse. En L14 pasamos el argumento -1 al método play() de la clase Sound. Con ello conseguimos que la música (que no suele tener una duración por ciclo de demasiados segundos) se repita indefinidamente. Eso nos genera un problema, ya que al cerrar la ventana la música sigue reproduciéndose.

- Para solucionarlo hemos creado la función cierra_ventana() (fuera de la definición de la clase) que detiene la música de fondo antes de que se cierre la ventana. Para poder acceder desde esta función a la música se declara la variable musica_de_fondo como global en L11. En L27 paramos la música de fondo y en L28 cerramos (mediante el método close() de director.window) la ventana principal de la forma habitual.

- Para que se ejecute la función cierra_ventana() al intentar cerrar la ventana principal se ha sobrescrito en L35 el método on_close() de director.window.

Podemos conseguir efectos sonoros como ir desvaneciendo la música de fondo que tengamos en una escena. Un código que lo crea es **ejemplo_animacion_y_sonido_3.py**:

```
1
2  import cocos
3  from cocos.audio.effect import Effect
4
5  class MiCapa(cocos.layer.Layer):
6
7      def __init__(self):
8          super().__init__()
9          musica_de_fondo = Effect('musica_fondo_retro.wav')
10         musica_de_fondo.sound.play()
11         musica_de_fondo.sound.fadeout(5000)
12
13
14 if __name__ == '__main__':
15     cocos.director.director.init(caption = 'Ejemplo animación y sonido 3',
16                                  audio_backend='sdl')
17     mi_escena = cocos.scene.Scene(MiCapa())
18     cocos.director.director.run(mi_escena)
19
```

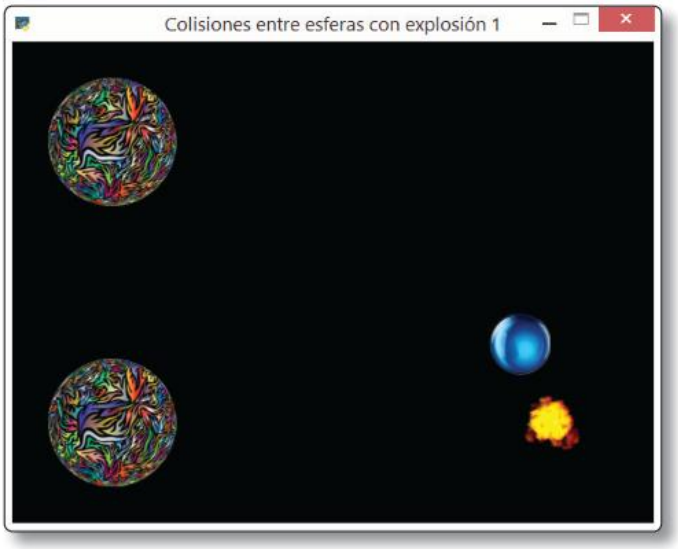
Hemos usado el método fadeout() de la clase Sound para indicar que la música cargada en L9 en la clase Effect, y reproducida en L10, se vaya desvaneciendo progresivamente hasta desaparecer en 5 segundos.

Podríamos ahora hacer que en el código ejemplo_manejador_colision_esfera.py, visto con anterioridad en el apartado 1.2.3, en lugar de simplemente desaparecer la esfera inmóvil contra la que chocamos, exlote (con animación y sonido). El lector puede como ejercicio intentar crear el código que lo haga posible[49]. El mío es el siguiente (**ejemplo_colision_animacion_sonido.py**):

49 Con ello será más consciente de las dificultades que pueden ir apareciendo.

```
1
2  import cocos
3  import cocos.collision_model as cm
4  import cocos.euclid as eu
5  from pyglet.window import key
6  from pyglet.image import load, ImageGrid, Animation
7  from cocos.audio.effect import Effect
8  from collections import defaultdict
9
10
11 class Actor(cocos.sprite.Sprite) :
12     def __init__(self, imagen, x, y) :
13         super().__init__(load(imagen))
14         self.position = (x, y)   #
15         self.cshape = cm.CircleShape(eu.Vector2(x, y), self.width/2)
16
17
18 class MiCapa(cocos.layer.Layer) :
19     is_event_handler = True
20
21     def __init__(self) :
22         super().__init__()
23         self.esfera_movil = Actor('mi_esfera.png', 320, 240)
24         self.add(self.esfera_movil)
25         for pos in [(100, 100), (540, 380), (540, 100), (100, 380)]:
26             a = Actor('mi_esfera_2.png', pos[0] , pos[1])
27             self.add(a)
28         self.manejador_colision = cm.CollisionManagerBruteForce()
29         self.velocidad = 100.0
30         self.teclas_pulsadas = defaultdict(int)
31         self.schedule(self.update)
32
33     def on_key_press(self, k, m) :
34         self.teclas_pulsadas[k] = 1
35     def on_key_release(self, k, m) :
36         self.teclas_pulsadas[k] = 0
37
38     def update(self, dt) :
39         self.manejador_colision.clear()
40         for _, node in self.children:
41             if isinstance(node, Actor):
42                 self.manejador_colision.add(node)
43         for other in self.manejador_colision.iter_colliding(self.esfera_movil):
44             self.remove(other)
45             mi_efecto_sonoro = Effect('explosion.wav')
46             mi_efecto_sonoro.sound.set_volume(0.1)
47             mi_efecto_sonoro.play()
48             mi_secuencia = ImageGrid(load('secuencia_explosion.png'), 4, 4)
49             mi_animacion = Animation.from_image_sequence(mi_secuencia, 0.05, False)
50             self.mi_sprite = cocos.sprite.Sprite(mi_animacion, (other.x, other.y))
51             self.add(self.mi_sprite)
52
53         x = self.teclas_pulsadas[key.RIGHT] - self.teclas_pulsadas[key.LEFT]
54         y = self.teclas_pulsadas[key. UP] - self.teclas_pulsadas[key.DOWN]
55         if x != 0 or y != 0:
56             pos = self.esfera_movil.position
57             nueva_x = pos[0] + self.velocidad * x * dt * 2
58             nueva_y = pos[1] + self.velocidad * y * dt * 2
59             self.esfera_movil.position = (nueva_x, nueva_y)
60             self.esfera_movil.cshape.center = eu.Vector2(nueva_x, nueva_y)
61
62
63 if __name__ == '__main__':
64     cocos.director.director.init(caption = 'Colisiones entre esferas con explosión 1',
65                                  audio_backend='sdl')
66     mi_escena = cocos.scene.Scene(MiCapa())
67     cocos.director.director.run(mi_escena)
68
```

Un instante de la salida puede ser el que aparece a continuación:



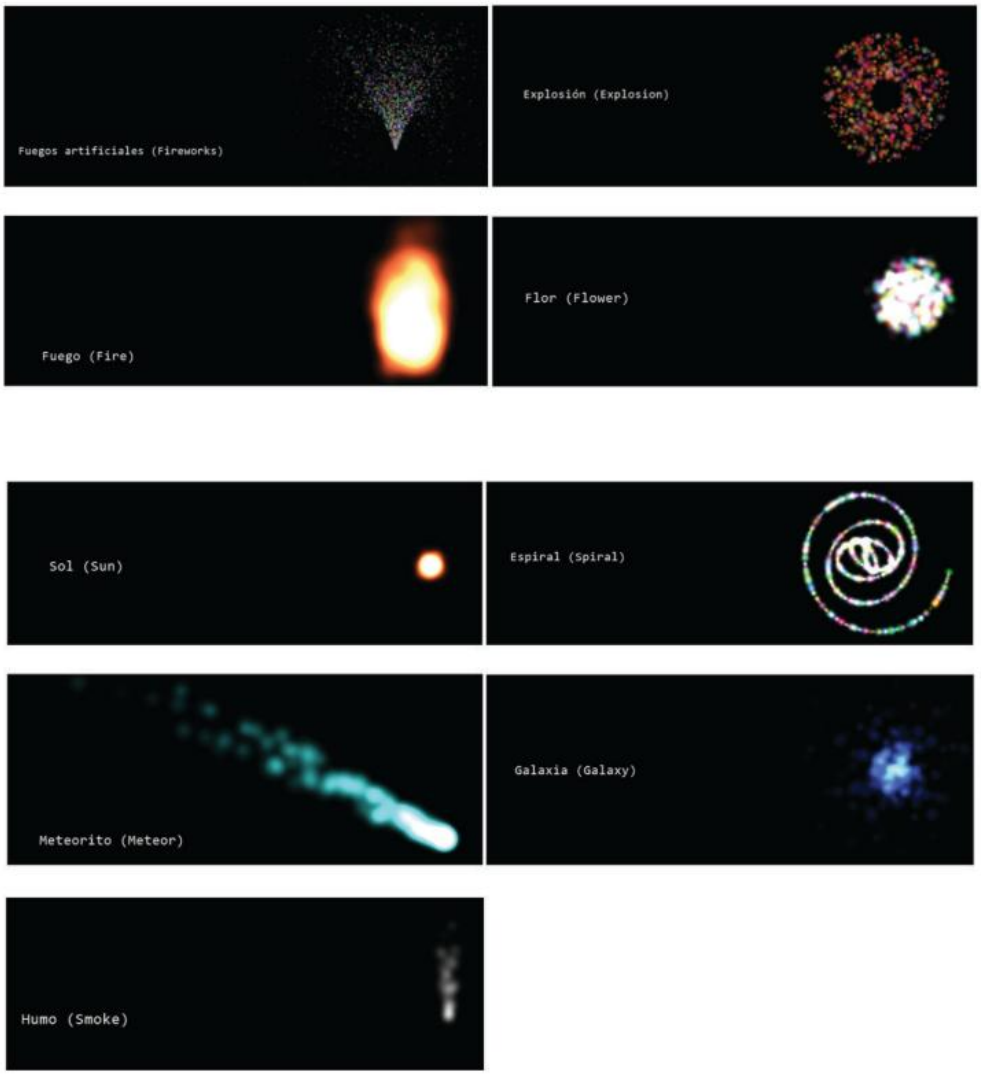
Por economía de código y simplicidad no he incluido la existencia de música de fondo, quedando como ejercicio para el lector en caso de querer añadirlo.

El código es muy sencillo de entender en base a lo que ya sabemos, por lo que su análisis queda como ejercicio para el lector.

1.2.6 Sistemas de partículas

En cocos2d tenemos la posibilidad de simular el comportamiento de determinados elementos como fuego, humo o explosiones mediante **sistemas de partículas**, que es una técnica de gráficos por ordenador que utiliza un gran número de objetos de pequeño tamaño (partículas) para simular ciertos tipos de fenómenos físicos que pueden ser difícil de reproducir con otras técnicas. Los sistemas de partículas de cocos2d son bidimensionales y los proporciona NumPy, por lo que debemos tenerlo instalado.

En cocos2d tenemos una clase base (**ParticleSystem**) sobre la que se desarrollan todos los sistemas de partículas predefinidos, que son un total de 9:



La clase Particlesystem[50] tiene una multitud de atributos (y métodos) que podemos configurar en base a nuestras necesidades, de cara a crear un sistema de partículas personalizado.

50 Para saber más sobre ella consultar el módulo particle.system en el Apéndice B.

Un sencillo ejemplo donde se muestra el sistema de partículas "fuegos artificiales" (fireworks) es **ejemplo_particulas_fireworks.py**:

```
1
2 from cocos.director import director
3 from cocos.scene import Scene
4 from cocos.layer import Layer
5 from cocos.text import Label
6 from cocos.particle_systems import Fireworks
7
8 class P1(Layer):
9     def __init__(self):
10        super().__init__()
11        mi_etiqueta = Label('Fuegos artificiales (Fireworks)',
12                                    font_name='Consolas',
13                                    font_size=18,
14                                    anchor_x='center', anchor_y='center')
15
16        mi_etiqueta.position = (320,384)
17        p1 = Fireworks()
18        p1.position = (900,384)
19        self.add(mi_etiqueta)
20        self.add(p1)
21
22
23 if __name__ == "__main__":
24     director.init(width=1280, height=768,
25                   caption='Sistemas de partículas. Fuegos artificiales.')
26     director.window.set_location(300,200)
27     director.run(Scene(P1()))
28
```

El código crea una capa llamada P1 donde se colocan una etiqueta y el origen de un sistema de partículas.

En **ejemplo_partículas.py**[51] recorreremos (al hacer clic con el ratón) los 9 tipos distintos de sistemas de partículas que tenemos preconfigurados.

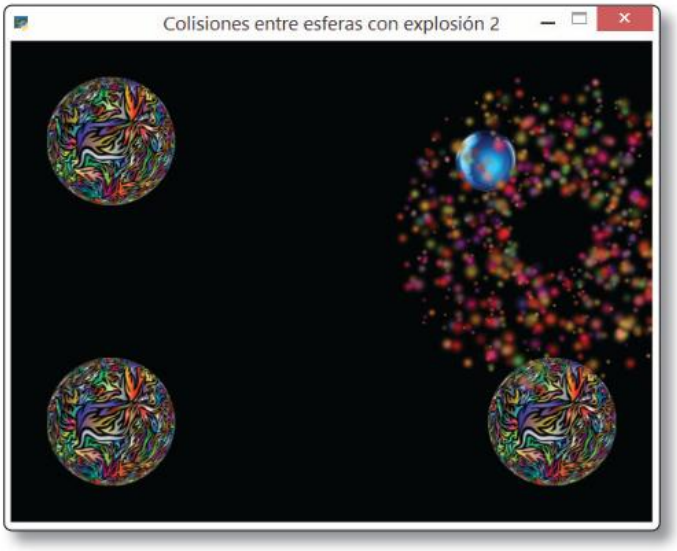
Intentaremos ahora que la explosión de ejemplo_colision_animacion_sonido.py (que vimos en el apartado anterior) no sea una animación en base a fotogramas sino un sistema de partículas (queda de nuevo como opción para el lector intentar su resolución previa a modo de ejercicio). El código (**ejemplo_colision_animacion_sonido_2.py**) es muy similar a ejemplo_colision_animacion_sonido.py, por lo que solamente incluyo la parte que varía[52]:

51 El análisis queda como ejercicio para el lector, al ser sencillo y repetirse la estructura de la clase P1 vista en el ejemplo anterior.

52 Al margen de la importación de la clase Explosion del módulo cocos.particle_system en detrimento de las clases ImageGrid y Animation del módulo pyglet.image.

```
37    def update(self, dt) :
38        self.manejador_colision.clear()
39        for _, node in self.children:
40            if isinstance(node, Actor):
41                self.manejador_colision.add(node)
42        for other in self.manejador_colision.iter_colliding(self.esfera_movil):
43            self.remove(other)
44            mi_efecto_sonoro = Effect('explosion.wav')
45            mi_efecto_sonoro.sound.set_volume(0.1)
46            mi_efecto_sonoro.play()
47            explosion_p = Explosion()
48            explosion_p.position = (other.x, other.y)
49            self.add(explosion_p)
```

La salida podrá tener la siguiente apariencia en una colisión entre esferas:



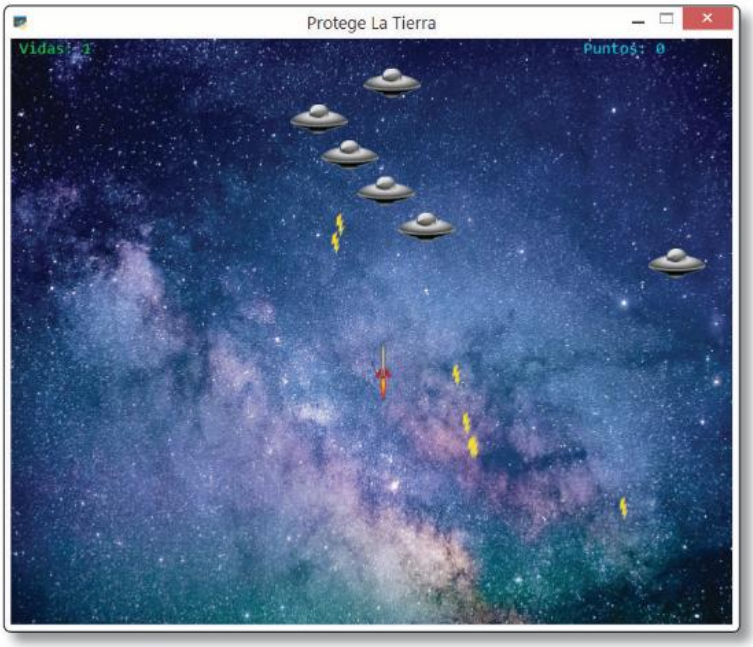
Cuando se tocan creamos el sistema de partículas Explosion con los argumentos por defecto, pero podríamos haberlos variado para generar una explosión personalizada.

2

DESARROLLO DE UN VIDEOJUEGO DE ARCADE

En este capítulo crearemos nuestro primer sencillo videojuego, que consistirá en el típico arcade "matamarcianos" con algunas características personales. Por ejemplo, en lugar de tener una nave espacial con posibilidad de disparar tendremos un misil que lanzaremos contra una serie de ovnis que se mueven horizontalmente de forma aleatoria y nos atacan mediante rayos.

Una imagen del resultado final del videojuego será la siguiente:



Para llegar a conseguirlo iremos paso a paso añadiendo características:

1. En un primer lugar crearemos el sprite del misil y controlaremos su movimiento horizontal (sólo será permitido en ese eje) mediante teclado dentro de los límites de la pantalla.

2. Añadimos la posibiliad de lanzamiento en vertical del misil, con su sonido asociado y sumándole el fuego del sistema de propulsión. Si sale de la pantalla creamos un nuevo misil en la posición original.

3. Añadimos un alien moviéndose en horizontal de forma aleatoria.

4. Añadimos capacidad al alien para disparar rayos de forma aleatoria.

5. Añadimos manejo de colisión entre el misil y el resto de objetos (ovni y sus disparos). En caso de colisionar desaparecen ambos objetos, y se crea un nuevo misil en la posición original.

6. Añadimos animación y sonido de explosión en colisiones. Además, retrasamos un segundo la aparición del misil tras un impacto.

7. Creamos una serie de aliens (6 exactamente) escalonados. Añadimos el HUD (dinámico).

8. Añadimos 3 capas más (inicio, ganador y perdedor). Inicialmente estaremos en la capa inicio, pasaremos a la capa del juego propiamente dicho al hacer clic en el ratón, y terminaremos en la capa ganador o perdedor, desde la que nos enviará de vuelta (en 3 segundos) al inicio.

9. Añadimos como imagen de fondo una del universo. Además incorporamos música de fondo a la capa principal del juego.

Iremos analizando el código de cada una de las etapas. En cada una de ellas el fichero asociado será **arcade_x.py**, donde x indicará el número de etapa:

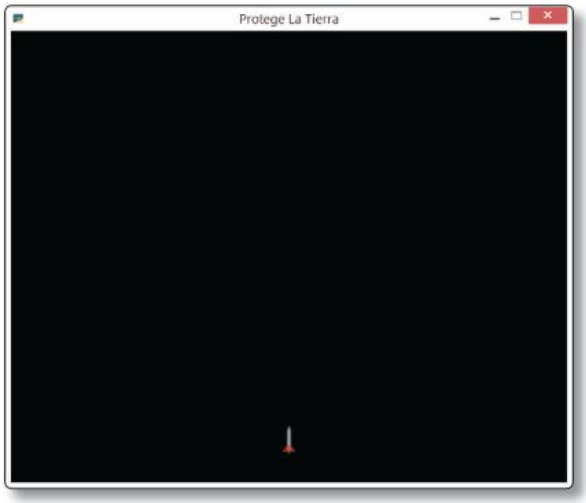
Etapa 1

El código de **arcade_1.py** es:

```
1
2 from pyglet.window import key
3 from cocos.director import director
4 from cocos.layer import Layer
5 from cocos.scene import Scene
6 from cocos.sprite import Sprite
7 from cocos.euclid import Vector2
8 from collections import defaultdict
```

```
9
10 class MiObjeto(Sprite):
11     def __init__(self, image, x, y):
12         super().__init__(image)
13         self.position = Vector2(x, y)
14
15     def move(self, offset):
16         self.position += offset
17
18     def update(self, delta_t):
19         pass
20
21
22 class MiMisil(MiObjeto):
23     TECLAS_PULSADAS = defaultdict(int)
24
25     def __init__(self,imagen, x, y):
26         super().__init__(imagen, x, y)
27         self.velocidad = Vector2(400, 0)
28
29     def update(self, delta_t):
30         pulsadas = MiMisil.TECLAS_PULSADAS
31
32         movimiento = pulsadas[key.RIGHT] - pulsadas[key.LEFT]
33         if movimiento != 0:
34             delta_x = (self.velocidad * movimiento * delta_t)[0]
35             if self.x  <= self.parent.ancho_ventana - self.width/2 - delta_x:
36                 if self.x - self.width/2 + delta_x > abs(delta_x):
37                     self.move(self.velocidad * movimiento * delta_t)
38
39
40 class MiCapa(Layer):
41     is_event_handler = True
42
43     def on_key_press(self, k, _):
44         MiMisil.TECLAS_PULSADAS[k] = 1
45
46     def on_key_release(self, k, _):
47         MiMisil.TECLAS_PULSADAS[k] = 0
48
49     def __init__(self):
50         super().__init__()
51         self.ancho_ventana, _ = director.get_window_size()
52         self.crear_misil()
53         self.schedule(self.update)
54
55     def crear_misil(self):
56         self.misil = MiMisil('mi_misil.png', self.ancho_ventana * 0.5, 50)
57         self.add(self.misil)
58
59     def update(self, dt):
60         for _, node in self.children:
61             node.update(dt)
62
63
64 if __name__ == '__main__':
65     ventana = director.init(caption='Protege La Tierra', width=800, height=650)
66     ventana.set_location(500,200)
67     mi_escena = Scene(MiCapa())
68     director.run(mi_escena)
69
```

Su ejecución genera la siguiente salida:



En el código tenemos tres clases definidas:

- MiObjeto
  Será la clase base para todos los sprites que usaremos en el juego. Además de __init__() tendrá los métodos move() y update() para, respectivamente, desplazar o actualizar el sprite.

- MiMisil
  Clase usada para el sprite del misil. Dispondrá de los métodos __init__() y update()

- MiCapa
  Basada en Layer, es la capa principal del juego (inicialmente la única). Dispone del método __init__(), de on_key_press() y on_key_release() para manejar las pulsaciones de teclas, de crear_misil() que realiza lo que indica su nombre, y de update().

Comentarios sobre el código:

- Importaremos solamente las clases que necesitemos, algo que hacemos en L2-8.

- En el método update() de la clase MiMisil creamos el atributo de clase TECLAS_PULSADAS en forma de instancia de defaultdict(int). De L30-37 creamos la variable movimiento, que puede tener 3 posibles valores en base a la pulsación de las teclas de cursor izquierda o derecha[53](0=sin

53 Son las teclas que tienen el símbolo de una flecha que apunta hacia la izquierda o la derecha, respectivamente.

moverse, 1=moverse hacia la derecha, -1=moverse hacia la izquierda). Si no es 0 calculamos delta_x, que es el potencial desplazamiento del misil sobre el eje x. Si al hacer ese desplazamiento estamos dentro de los límites de la pantalla realizamos el movimiento mediante el método move().

- El atributo de clase is_event_handler de la clase MiCapa tiene valor True para poder recibir los eventos[54].
- Mediante los métodos on_key_press() y on_key_release() almacenamos en TECLAS_PULSADAS un 1 o un 0 asociado con la tecla que hayamos, respectivamente, pulsado o soltado.
- Calculamos el ancho en puntos de la ventana principal mediante la función get_window_size() del director, y lo almacenamos en el atributo ancho_ventana.
- El método crear_misil() crea el sprite del misil (en el centro del eje horizontal[55] y desplazado 50 puntos en el eje vertical) y lo añade a la capa.
- El método update() de la capa simplemente llama a todos los métodos update() de los objetos contenidos en ella.
- Al inicializar mediante __init__() la capa llamamos a crear_misil() y mediante el método schedule() planificamos la llamada al método update() de la capa en cada fotograma[56].
- Hacer notar que al inicializar el director creamos una ventana de 800 x 650 puntos.

Etapa 2

Como a partir de ahora se va añadiendo progresivamente código al ya existente, presentaré en varios casos únicamente el bloque en el que hay algún tipo de modificación y/o añadido[57]. Es el caso de **arcade_2.py**[58]:

54 En este caso nos interesan solamente los de teclado.

55 Para ello hacemos uso del atributo ancho_ventana recogido con anterioridad.

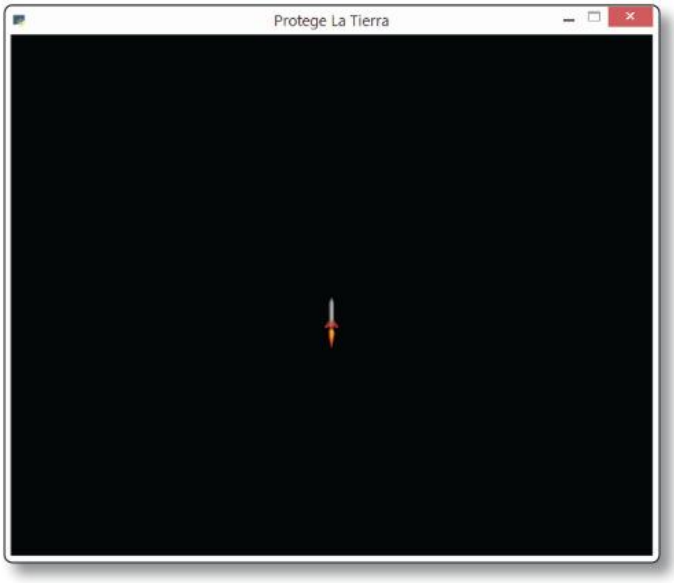
56 Si no le indicamos argumento temporal será cada aproximadamente 0.018 segundos.

57 Omitiendo las adicionales importaciones de clases desde los módulos.

58 Bajar el volumen de los altavoces del ordenador antes de ejecutar el código.

```
23
24 class MiMisil(MiObjeto):
25     TECLAS_PULSADAS = defaultdict(int)
26
27     def __init__(self,imagen, x, y):
28         super().__init__(imagen, x, y)
29         self.velocidad = Vector2(400, 0)
30         self.esta_lanzado = False
31
32     def update(self, delta_t):
33         pulsadas = MiMisil.TECLAS_PULSADAS
34         if pulsadas[key.SPACE] and self.esta_lanzado == False:
35             self.image = load('mi_misil_2.png')
36             sonido_misil = Effect('misil.wav')
37             sonido_misil.play()
38             self.esta_lanzado = True
39         elif self.esta_lanzado == True:
40             self.move(Vector2(0,10))
41         else:
42             movimiento = pulsadas[key.RIGHT] - pulsadas[key.LEFT]
43             if movimiento != 0:
44                 delta_x = (self.velocidad * movimiento * delta_t)[0]
45                 if self.x  <= self.parent.ancho_ventana - self.width/2 - delta_x:
46                     if self.x - self.width/2 + delta_x > abs(delta_x):
47                         self.move(self.velocidad * movimiento * delta_t)
48         if self.y > self.parent.alto_ventana:
49             self.kill()
50
51     def on_exit(self):
52         self.parent.crear_misil()
53
```

Una instantánea del lanzamiento del misil es la siguiente:



Comentarios sobre el código:

- Para contar con la posibilidad de lanzar el misil al pulsar la barra espaciadora (junto a su sonido) añadiremos código a la clase MiMisil, ampliando el método update() y sobrescribiendo on_exit().
- Crearemos el atributo esta_lanzado (inicialmente con valor False) para indicar si el misil está por el aire. En caso de no estarlo y si pulsamos la barra espaciadora, cambiamos la imagen del misil para incluir el fuego de la propulsión y damos valor True a esta_lanzado. Además, reproducimos el sonido de lanzamiento de misil guardado en el fichero **misil.wav**, para lo que es necesario en la inicialización del director que se le pase el argumento audio_backend con valor 'sdl'.
- En cada ejecución de update(), si se mantiene el valor True para esta_lanzado, se desplazará el sprite hacia arriba en 10 píxeles mediante el método move() de la clase MiMisil, que hereda de MiObjeto.
- En L48-49 indicamos que si el misil sale de la zona de pantalla (valor que tenemos almacenado en el atributo alto_ventana de la clase MiCapa) se elimine de la capa, ya que en caso contrario seguiría (a pesar de no verlo) calculando sobre él, una carga computacional innecesaria y que podría (si existen muchos misiles) ralentizar la ejecución del juego.
- Sobrescribimos el método on_exit() de la clase MiMisil para que, una vez el misil haya sido eliminado de la capa, se cree uno nuevo en la posición original.

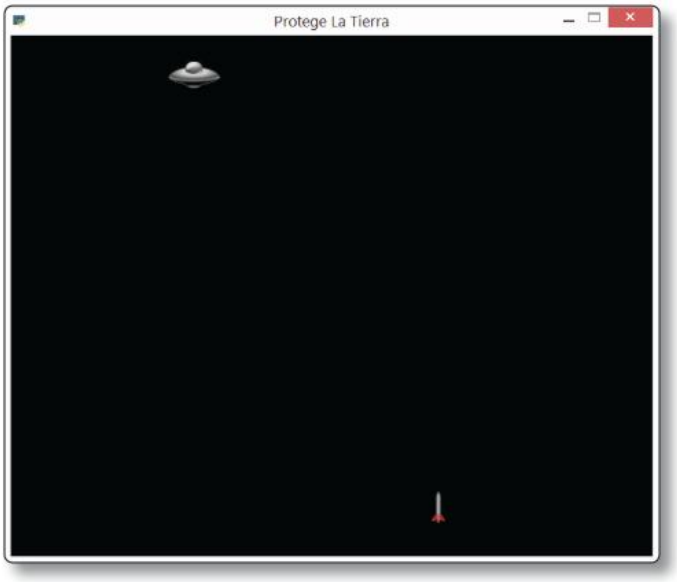
Etapa 3

Código añadido en **arcade_3.py**:

```
55
56 class MiAlien(MiObjeto):
57     def __init__(self, img, x, y):
58         super().__init__(img, x, y)
59         self.velocidad = Vector2(150, 0)
60         self.contador = 0
61         self.dir = choice([-1,1])
62
63     def update(self, delta_t):
64         if 50 < (self.position[0] + (self.dir * self.velocidad * delta_t)[0]) < 750:
65             if self.contador < 100:
66                 self.move(self.dir * self.velocidad * delta_t)
67                 self.contador += 1
68             else:
69                 self.contador = 0
70                 self.dir = choice([-1,1])
71         else:
```

```
            self.dir *= -1


class MiCapa(Layer):
    is_event_handler = True

    def on_key_press(self, k, _):
        MiMisil.TECLAS_PULSADAS[k] = 1

    def on_key_release(self, k, _):
        MiMisil.TECLAS_PULSADAS[k] = 0

    def __init__(self):
        super().__init__()
        self.ancho_ventana, self.alto_ventana = director.get_window_size()
        self.crear_misil()
        self.crear_alien()
        self.schedule(self.update)

    def crear_misil(self):
        self.misil = MiMisil('mi_misil.png', self.ancho_ventana * 0.5, 50)
        self.add(self.misil)

    def crear_alien(self):
        self.mi_alien = MiAlien('mi_ovni_1.png', self.ancho_ventana * 0.5, 600)
        self.add(self.mi_alien)

    def update(self, dt):
        for _, node in self.children:
            node.update(dt)

```

La salida en un instante dado es:



Comentarios sobre el código:

- Para añadir un ovni moviéndose de forma aleatoria creamos una clase llamada MiAlien, que hereda de MiObjeto. Al ya conocido atributo velocidad le incorporamos dos más: dir (que tendrá valor 1 o -1) lo usaremos para indicar la dirección de desplazamiento horizontal, y contador, empleado para calcular cada cuántos fotogramas[59] el ovni recalcula (de forma aleatoria y mediante el uso de la función choice() del módulo random) la dirección de su movimiento. En el caso de acercarse a cualquiera de los dos lados de la pantalla, automáticamente cambiaría su dirección (L72).
- En la clase MiCapa añadimos el método crear_alien(), que añade a la capa un ovni en la parte superior de la pantalla, en las coordenadas (400, 600).
- Ahora en el método update() de MiCapa se llamará a los dos métodos del mismo nombre de las clases MiMisil y MiAlien, para que se actualicen los movimientos de ambos sprites.

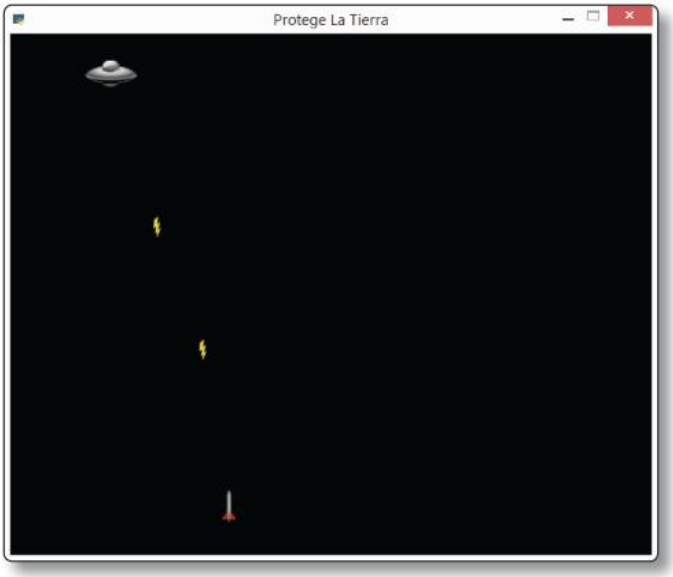
Etapa 4

Código añadido en **arcade_4.py**:

```
55
56 class MiRayo(MiObjeto):
57     def __init__(self,imagen, x, y):
58         super().__init__(imagen, x, y)
59         self.velocidad = Vector2(0, -400)
60
61     def update(self, delta_t):
62         self.move(self.velocidad * delta_t)
63         if self.y < 0:
64             self.kill()
65
66
67 class MiAlien(MiObjeto):
68     def __init__(self, img, x, y):
69         super().__init__(img, x, y)
70         self.velocidad = Vector2(150, 0)
71         self.contador = 0
72         self.dir = choice([-1,1])
73
74     def update(self, delta_t):
75         if 50 < (self.position[0] + (self.dir * self.velocidad * delta_t)[0]) < 750:
76             if self.contador < 100:
77                 self.move(self.dir * self.velocidad * delta_t)
78                 self.contador += 1
79             else:
80                 self.contador = 0
81                 self.dir = choice([-1,1])
82         else:
83             self.dir *= -1
84         if randint(1,1000) > 990:
85             a = MiRayo('mi_rayo.png', self.x, self.y -30)
86             self.parent.add(a)
87
```

59 En nuestro caso el valor elegido ha sido 100.

La salida será algo similar a la siguiente:



Comentarios sobre el código:

- Para que el ovni nos pueda disparar rayos crearemos una clase llamada MiRayo para el propio rayo, y añadiremos código en la clase MiAlien para que lo lance de forma aleatoria cada cierto tiempo.
- La clase MiRayo está basada en MiObjeto. Sus instancias descienden verticalmente en cada fotograma una serie de píxeles (400 * delta_t) hasta el momento en que salen de la pantalla (atributo y negativo), momento en el que son eliminadas de la capa (L63).
- En el método update() de MiAlien se crea y añade a la capa (cada cierto instante aleatorio de tiempo) una instancia de MiRayo a 30 píxeles por debajo de la posición del ovni. Para ello usamos la función randint() del módulo random, que nos proporciona un número entero aleatorio dentro del rango que le indiquemos. En nuestro caso el rango es de 1 a 1000 y lanzamos el rayo si el entero que nos devuelve la función es mayor que 990. Puede parecer que es un rango muy pequeño, pero debemos recordar que el método se ejecuta cada menos de 0.02 segundos.

Etapa 5

En **arcade_5.py** añadimos el siguiente código:

Bloque1:

```
13
14 class MiObjeto(Sprite):
15     def __init__(self, image, x, y):
16         super().__init__(image)
17         self.position = Vector2(x, y)
18         self.cshape = AARectShape(self.position,
19                                   self.width * 0.5,
20                                   self.height * 0.5)
21
22     def move(self, offset):
23         self.position += offset
24         self.cshape.center += offset
25
```

Bloque2:

```
93
94 class MiCapa(Layer):
95     is_event_handler = True
96
97     def on_key_press(self, k, _):
98         MiMisil.TECLAS_PULSADAS[k] = 1
99
100    def on_key_release(self, k, _):
101        MiMisil.TECLAS_PULSADAS[k] = 0
102
103    def __init__(self):
104        super().__init__()
105        self.ancho_ventana, self.alto_ventana = director.get_window_size()
106        self.man_col = CollisionManagerBruteForce()
107        self.crear_misil()
108        self.crear_alien()
109        self.schedule(self.update)
110
111    def crear_misil(self):
112        self.misil = MiMisil('mi_misil.png', self.ancho_ventana * 0.5, 50)
113        self.add(self.misil)
114
115    def crear_alien(self):
116        self.mi_alien = MiAlien('mi_ovni_1.png', self.ancho_ventana * 0.5, 600)
117        self.add(self.mi_alien)
118
119    def update(self, dt):
120        self.man_col.clear()
121        for _, node in self.children:
122            node.update(dt)
123        for _, node in self.children:
124            self.man_col.add(node)
125
126        self.collide(self.misil)
127
128    def collide(self, node):
129        if node is not None:
130            for other in self.man_col.iter_colliding(node):
131                other.kill()
132                node.kill()
133
```

La salida será la misma que en arcade_4.py, pero ahora cuando haya una colisión misil-ovni o misil-rayo ambos se eliminarán, para instantáneamente crear un nuevo misil en la posición inicial.

Comentarios sobre el código:

- Habilitamos la capacidad de detectar colisiones entre el misil y los dos otros tipos de objeto que podemos tener: ovnis y rayos. Para ello añadimos a la clase MiObjeto el (obligatorio) atributo cshape, que será un objeto de la clase AARectShape. Además, en el método move(), además de actualizar la posición (como hacíamos hasta ahora) también lo hacemos con la de cshape.

- En la clase MiCapa crearemos en su inicialización un atributo llamado man_col (una instancia de CollisionManagerBruteForce[60]) que será propiamente el manejador de colisiones.

- En el método update() de MiCapa realizamos el proceso para "cargar" de forma correcta en el manejador de colisiones todos los elementos que queremos considerar. Inicialmente (L120) borramos todos los elementos que pueda contener mediante el método clear(), posteriormente actualizamos los atributos cshape de todos los objetos colisionables (L121-122) y finalmente los añadimos al manejador de colisiones (L123-124).

- Hemos creado un nuevo método en MiCapa llamado collide(), al que pasamos un objeto como argumento y comprueba mediante el método iter_colliding() del manejador de colisiones si hay objetos colisionando con él, en cuyo caso elimina ambos (se creará entonces un nuevo misil ya que se ejecutará el método on_exit() de MiMisil). El método collide() lo llamamos dentro del método update() con nuestro objeto misil como argumento.

60 Importado desde el módulo cocos.collision_model.

Etapa 6

En **arcade_6.py** añadimos:

Bloque 1:

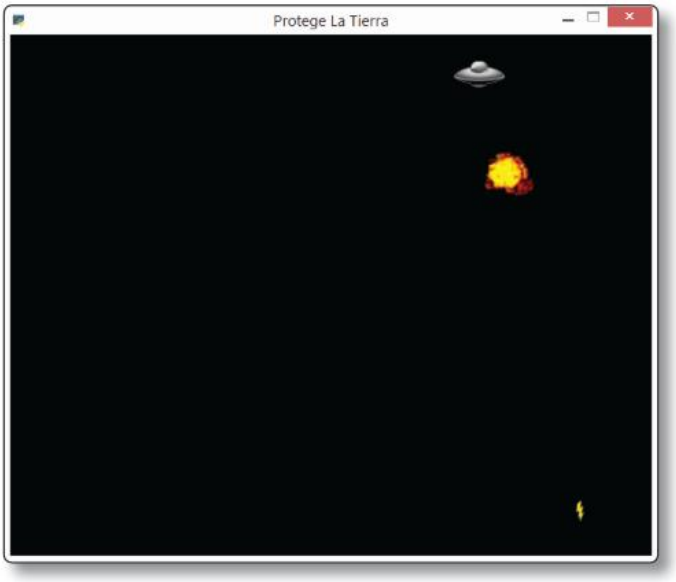
```
30
31 class mi_misil(mi_objeto):
32     TECLAS_PULSADAS = defaultdict(int)
33
34     def __init__(self,imagen, x, y):
35         super().__init__(imagen, x, y)
36         self.velocidad = Vector2(400, 0)
37         self.esta_lanzado = False
38
39     def update(self, delta_t):
40         pulsadas = mi_misil.TECLAS_PULSADAS
41         if pulsadas[key.SPACE] and self.esta_lanzado == False:
42             self.image = load('mi_misil_2.png')
43             sonido_misil = Effect('misil.wav')
44             sonido_misil.play()
45             self.esta_lanzado = True
46         elif self.esta_lanzado == True:
47             self.move(Vector2(0,10))
48         else:
49             movimiento = pulsadas[key.RIGHT] - pulsadas[key.LEFT]
50             if movimiento != 0:
51                 delta_x = (self.velocidad * movimiento * delta_t)[0]
52                 if self.x  <= self.parent.ancho_ventana - self.width/2 - delta_x:
53                     if self.x - self.width/2 + delta_x > abs(delta_x):
54                         self.move(self.velocidad * movimiento * delta_t)
55         if self.y > self.parent.alto_ventana:
56             self.kill()
57
58     def on_exit(self):
59         self.do(Delay(1) + CallFunc(self.parent.crear_misil))
60
```

Bloque 2:

```
95 class MiCapa(Layer):
96     is_event_handler = True
97
98     def on_key_press(self, k, _):
99         MiMisil.TECLAS_PULSADAS[k] = 1
100
101    def on_key_release(self, k, _):
102        MiMisil.TECLAS_PULSADAS[k] = 0
103
104    def __init__(self):
105        super().__init__()
106        self.ancho_ventana, self.alto_ventana = director.get_window_size()
107        self.man_col = CollisionManagerBruteForce()
108        self.crear_misil()
109        self.crear_alien()
110        self.schedule(self.update)
111
112    def crear_misil(self):
113        self.misil = MiMisil('mi_misil.png', self.ancho_ventana * 0.5, 50)
114        self.add(self.misil)
115
116    def crear_alien(self):
```

```
        self.mi_alien = MiAlien('mi_ovni_1.png', self.ancho_ventana * 0.5, 600)
        self.add(self.mi_alien)

    def update(self, dt):
        self.man_col.clear()
        for _, node in self.children:
            if isinstance(node, MiObjeto):
                node.update(dt)
        for _, node in self.children:
            if isinstance(node, MiObjeto):
                self.man_col.add(node)

        self.collide(self.misil)

    def collide(self, node):
        if node is not None:
            for other in self.man_col.iter_colliding(node):
                effect = Effect('explosion.wav')
                effect.play()
                if self.children.count((0, other)) != 0:
                    other.kill()
                if self.children.count((0, node)) != 0:
                    node.kill()
                seq = ImageGrid(load('secuencia_explosion.png'), 4, 4)
                anim = Animation.from_image_sequence(seq, 0.05, False)
                self.mi_sprite = Sprite(anim, (other.x, other.y))
                self.add(self.mi_sprite)
                self.do(Delay(0.8) + CallFunc(self.mi_sprite.kill))
```

A continuación presento una imagen del momento de la colisión entre el misil y un rayo del ovni:



Comentarios sobre el código:

- Hemos añadido (en L59) un retraso de un segundo desde que el misil colisiona hasta que vuelve a aparecer en su posición original.

- Para añadir la animación y el sonido de una explosión cada vez que tengamos una colisión misil-ovni o misil-rayo ampliaremos el método collide() de MiCapa. El caso del sonido es muy sencillo y simplemente crearemos un efecto sonoro con la clase Effect y lo reproduciremos en el caso de colisión.

- Para crear la animación de la explosión tendremos más problemas. Hay que tener en cuenta que añadimos un sprite con ella y debemos eliminarlo al terminarse ya que, de lo contrario, quedaría ahí y podría colisionar (a pesar de no visualizarse) con posterioridad. Es por ello por lo que en L144 esperamos el tiempo que tarda la explosión (0.8 segundos) y posteriormente eliminamos el sprite.

- También debemos tener en cuenta que, al ejecutarse tan rápido el programa, el método collide() puede llamarse seguidamente varias veces. De ahí los dos condicionales en L136-139. En el caso de no colocarlos se intentará eliminar varias veces dos sprites que ya han sido borrados previamente de la capa, lanzando el consiguiente error.

- En L122-127 colocamos código para actualizar y añadir al manejador de colisiones sólo los objetos que derivan de MiObjeto, dejando con ello fuera a los sprites de las animaciones de explosión.

- Tal y como tenemos el código está la posibilidad de que haya varias explosiones cuando haya colisiones, algo que he mantenido por considerarlo más atractivo visualmente. En caso de sólo querer una explosión, deberemos introducir (en éste y los ficheros siguientes) el código self.misil = None tras L139.

Etapa 7

En **arcade_7.py** se ha añadido:

Bloque 1:

```
95
96 class MiEtiqueta(Label):
97     def __init__(self, texto, x, y):
98         super().__init__(texto, (x, y), font_name = 'Consolas', font_size = 14,
99                          anchor_x ='center', anchor_y = 'center')
100
101
102 class MiCapa(Layer):
103     is_event_handler = True
104
105     def on_key_press(self, k, _):
106         MiMisil.TECLAS_PULSADAS[k] = 1
107
108     def on_key_release(self, k, _):
109         MiMisil.TECLAS_PULSADAS[k] = 0
110
111     def __init__(self, HUD):
112         super().__init__()
113         self.mi_HUD = HUD
114         self.ancho_ventana, self.alto_ventana = director.get_window_size()
115         self.man_col = CollisionManagerBruteForce()
116         self.crear_misil()
117         self.crear_aliens()
118         self.schedule(self.update)
119
120     def crear_misil(self):
121         self.misil = MiMisil('mi_misil.png', self.ancho_ventana * 0.5, 50)
122         self.add(self.misil)
123
124     def crear_aliens(self):
125         for i in range(6):
126             alien = MiAlien('mi_ovni_1.png',self.ancho_ventana * 0.5, 600 - i*40)
127             self.add(alien)
128
```

Bloque 2:

```
139
140     def collide(self, node):
141         if node is not None:
142             for other in self.man_col.iter_colliding(node):
143                 effect = Effect('explosion.wav')
144                 effect.play()
145                 if self.children.count((0, other)) != 0:
146                     other.kill()
147                     if isinstance(other, MiAlien):
148                         self.mi_HUD.puntos += 20
149                         self.mi_HUD.update()
150                 if self.children.count((0, node)) != 0:
151                     node.kill()
152                     if node.y == 50:
153                         self.mi_HUD.vidas -= 1
154                         self.mi_HUD.update()
155                 seq = ImageGrid(load('secuencia_explosion.png'), 4, 4)
156                 anim = Animation.from_image_sequence(seq, 0.05, False)
157                 self.mi_sprite = Sprite(anim, (other.x, other.y))
158                 self.add(self.mi_sprite)
159                 self.do(Delay(0.8) + CallFunc(self.mi_sprite.kill))
```

```
160
161
162 class MiHUD(Layer):
163     def __init__(self):
164         super().__init__()
165         self.vidas = 3
166         self.puntos = 0
167         self.update()
168
169     def update(self):
170         self.children = []
171         texto_vidas = 'Vidas: ' + str(self.vidas)
172         etiqueta_vidas = MiEtiqueta(texto_vidas, 50, 640)
173         texto_puntos = 'Puntos: ' + str(self.puntos)
174         etiqueta_puntos = MiEtiqueta(texto_puntos, 680, 640)
175         self.add(etiqueta_vidas)
176         self.add(etiqueta_puntos)
177
178
179 if __name__ == '__main__':
180     director.init(caption='Protege La Tierra',
181                   width=800, height=650, audio_backend='sdl')
182     mi_HUD = MiHUD()
183     mi_capa = MiCapa(mi_HUD)
184     mi_escena = Scene(mi_capa, mi_HUD)
185     director.run(mi_escena)
186
```

Comentarios sobre el código:

- Para tener 6 ovnis en la pantalla añadiremos el método crear_aliens() de la clase MiCapa, donde simplemente los añadimos de forma escalonada a la capa.
- Para el HUD definiremos la clase MiHUD (basada en Layer), donde nos apoyaremos en nuestra clase MiEtiqueta (basada en Label) que nos ayudará a tratar con textos. En el HUD representaremos (en las coordenadas correspondientes) el número de vidas (inicialmente 3) y el de puntos, que irán variando a medida que nos alcanzan o derribamos ovnis, respectivamente. MiHUD dispondrá del método update() para cuando necesitemos actualizar el contenido del HUD.
- Tendremos ahora en nuestra escena dos capas: la principal que hemos tratado hasta ahora, y la del HUD. Para poder acceder a ésta última la instancia de MiCapa recibe en su creación una instancia de MiHUD (L183), que almacenará en el atributo mi_HUD (L113).
- Para modificar dinámicamente el número de vidas y/o puntos ampliamos el método collide() de MiCapa, identificando el tipo de objeto que colisiona y actuando en consecuencia. Notar que, por no hacer más complejo el código, no hemos puesto tope inferior al número de vidas, por lo que si nos derriban más de tres veces podrá ser un número negativo. Es algo que solucionaremos en la siguiente etapa.

- No confundir la variable mi_HUD definida en L182 con el atributo mi_HUD de la clase MiCapa, definido en L113.

Veamos un ejemplo de cómo aparecerá la pantalla tras la ejecución del código:

Etapa 8

En **arcade_8.py** incorporamos el siguiente código adicional:

Bloque 1:

```
96
97 class MiEtiqueta(Label):
98     def __init__(self, texto, x, y, c = (255,255,255,255)):
99         super().__init__(texto, (x, y), font_name = 'Consolas', font_size = 14,
100                          color = c,
101                          anchor_x ='center', anchor_y = 'center')
102
103
104 class CapaInicio(Layer):
105     is_event_handler = True
106     def __init__(self):
107         super().__init__()
108         self.add(MiEtiqueta('Pulsa botón de ratón para iniciar', 400, 325))
109
110     def on_mouse_release(self, x, y, buttons, modifiers):
111         a = MiHUD()
112         director.replace(Scene(MiCapa(a), a))
113
114
115 class CapaGanador(Layer):
```

```
    def __init__(self):
        super().__init__()
        self.add(MiEtiqueta('¡Enhorabuena, has ganado!', 400, 325))
        f1 = lambda : director.replace(Scene(CapaInicio()))
        self.do(Delay(3) + CallFunc(f1))


class CapaGameOver(Layer):
    def __init__(self):
        super().__init__()
        self.add(MiEtiqueta('GAME OVER', 400, 325))
        f1 = lambda : director.replace(Scene(CapaInicio()))
        self.do(Delay(3) + CallFunc(f1))

```

Bloque 2:

```

class MiCapa(Layer):
    is_event_handler = True

    def on_key_press(self, k, _):
        MiMisil.TECLAS_PULSADAS[k] = 1

    def on_key_release(self, k, _):
        MiMisil.TECLAS_PULSADAS[k] = 0

    def __init__(self, HUD):
        super().__init__()
        self.mi_HUD = HUD
        self.ancho_ventana, self.alto_ventana = director.get_window_size()
        self.man_col = CollisionManagerBruteForce()
        self.crear_misil()
        self.crear_aliens()
        self.schedule(self.update)

    def crear_misil(self):
        self.misil = MiMisil('mi_misil.png', self.ancho_ventana * 0.5, 50)
        self.add(self.misil)

    def crear_aliens(self):
        for i in range(6):
            alien = MiAlien('mi_ovni_1.png',self.ancho_ventana * 0.5, 600 - i*40)
            self.add(alien)

    def update(self, dt):
        sin_ovnis = True
        for data in self.children:
            if isinstance(data[1], MiAlien):
                sin_ovnis = False
        if sin_ovnis:
            director.replace(Scene(CapaGanador()))

        self.man_col.clear()
        for _, node in self.children:
            if isinstance(node, MiObjeto):
                node.update(dt)
        for _, node in self.children:
            if isinstance(node, MiObjeto):
                self.man_col.add(node)

        self.collide(self.misil)

```

Bloque 3:

```
198
199 class MiHUD(Layer):
200     def __init__(self):
201         super().__init__()
202         self.vidas = 3
203         self.puntos = 0
204         self.update()
205
206     def update(self):
207         self.children = []
208         texto_vidas = 'Vidas: ' + str(self.vidas)
209         etiqueta_vidas = MiEtiqueta(texto_vidas, 50, 640, (0,255,0,255))
210         texto_puntos = 'Puntos: ' + str(self.puntos)
211         etiqueta_puntos = MiEtiqueta(texto_puntos, 680, 640, (0,255,255,255))
212         self.add(etiqueta_vidas)
213         self.add(etiqueta_puntos)
214         if self.vidas == 0:
215             director.replace(Scene(CapaGameOver()))
216
217
218 if __name__ == '__main__':
219     director.init(caption='Protege La Tierra',
220                   width=800, height=650, audio_backend='sdl')
221     director.run(Scene(CapaInicio()))
222
```

La escena principal del juego es muy similar a la de arcade_7.py (salvo los colores del texto del HUD), por lo que no adjuntaré una imagen de ella.

Comentarios sobre el código:

- La clase MiEtiqueta ha sido cambiada para incluir el color a la hora de crear sus instancias.
- Ahora distribuimos el juego en 4 escenas, cada una de ellas con una capa: la propia del juego y vista hasta el momento, una de inicio, una de ganador y otra de juego finalizado. Para la primera tenemos la clase MiCapa, mientras que para las restantes definiremos, respectivamente, las clases CapaInicio, CapaGanador y CapaGameOver (basadas en Layer). En ellas se representa un texto centrado con información.
- Para saber si tenemos que ejecutar la escena de juego finalizado comprobamos en cada actualización del HUD si el número de vidas llega a 0, mientras que para ejecutar la capa de ganador inspeccionaremos en cada fotograma (dentro del método update() de la instancia de MiCapa) si hay objetos de la clase MiAlien, ya que en caso negativo será la prueba de haber terminado con todos los ovnis. En ambos casos usamos el método replace() del director y la clase Scene para cambiar a esas sencillas escenas finales.

- De la escena inicial (conteniendo una instancia de CapaInicio) iremos por tanto (al hacer clic con el ratón) a la del juego, que desembocará en las escenas con las capas ganador o juego finalizado, desde donde (tras tres segundos) volveremos de nuevo a la inicial.

- En CapaInicio configuramos el atributo de clase is_event_handler con valor True para manejar mediante el método on_mouse_release() el evento pulsación tecla del ratón, en el que cambiaremos (haciendo uso del método replace() del director) a escena principal del juego.

Etapa 9 (y final)

Al ser ya la etapa final tendremos la versión final del código, por lo que lo presentaré en su totalidad (**arcade_9.py**):

```
1
2  from pyglet.window import key
3  from pyglet.image import load, ImageGrid, Animation
4  from cocos.director import director
5  from cocos.layer import Layer
6  from cocos.scene import Scene
7  from cocos.sprite import Sprite
8  from cocos.euclid import Vector2
9  from cocos.audio.effect import Effect
10 from cocos.collision_model import CollisionManagerBruteForce, AARectShape
11 from cocos.actions import Delay, CallFunc
12 from cocos.text import Label
13 from cocos.scenes.transitions import FadeTransition
14 from random import choice, randint
15 from collections import defaultdict
16
17 class MiObjeto(Sprite):
18     def __init__(self, image, x, y):
19         super().__init__(image)
20         self.position = Vector2(x, y)
21         self.cshape = AARectShape(self.position,
22                                   self.width * 0.5,
23                                   self.height * 0.5)
24
25     def move(self, offset):
26         self.position += offset
27         self.cshape.center += offset
28
29     def update(self, delta_t):
30         pass
31
32
33 class MiMisil(MiObjeto):
34     TECLAS_PULSADAS = defaultdict(int)
35
36     def __init__(self,imagen, x, y):
37         super().__init__(imagen, x, y)
38         self.velocidad = Vector2(400, 0)
39         self.esta_lanzado = False
```

```
40
41      def update(self, delta_t):
42          pulsadas = MiMisil.TECLAS_PULSADAS
43          if pulsadas[key.SPACE] and self.esta_lanzado == False:
44              self.image = load('mi_misil_2.png')
45              sonido_misil = Effect('misil.wav')
46              sonido_misil.sound.set_volume(0.1)
47              sonido_misil.play()
48              self.esta_lanzado = True
49          elif self.esta_lanzado == True:
50              self.move(Vector2(0,10))
51          else:
52              movimiento = pulsadas[key.RIGHT] - pulsadas[key.LEFT]
53              if movimiento != 0:
54                  delta_x = (self.velocidad * movimiento * delta_t)[0]
55                  if self.x  <= self.parent.ancho_ventana - self.width/2 - delta_x:
56                      if self.x - self.width/2 + delta_x > abs(delta_x):
57                          self.move(self.velocidad * movimiento * delta_t)
58          if self.y > self.parent.alto_ventana:
59              self.kill()
60
61      def on_exit(self):
62          self.do(Delay(1) + CallFunc(self.parent.crear_misil))
63
64
65  class MiRayo(MiObjeto):
66      def __init__(self,imagen, x, y):
67          super().__init__(imagen, x, y)
68          self.velocidad = Vector2(0, -400)
69
70      def update(self, delta_t):
71          self.move(self.velocidad * delta_t)
72          if self.y < 0:
73              self.kill()
74
75
76  class MiAlien(MiObjeto):
77      def __init__(self, img, x, y):
78          super().__init__(img, x, y)
79          self.velocidad = Vector2(100, 0)
80          self.contador = 0
81          self.dir = choice([-1,1])
82
83      def update(self, delta_t):
84          if 50 < (self.position[0] + (self.dir * self.velocidad * delta_t)[0]) < 750:
85              if self.contador < 100:
86                  self.move(self.dir * self.velocidad * delta_t)
87                  self.contador += 1
88              else:
89                  self.contador = 0
90                  self.dir = choice([-1,1])
91          else:
92              self.dir *= -1
93          if randint(1,1000) > 990:
94              a = MiRayo('mi_rayo.png', self.x, self.y -30)
95              self.parent.add(a, z = 1)
96
97
98  class MiEtiqueta(Label):
99      def __init__(self, texto, x, y, c = (255,255,255,255)):
100         super().__init__(texto, (x, y), font_name = 'Consolas', font_size = 14,
101                          color = c,
102                          anchor_x ='center', anchor_y = 'center')
103
104
105 class CapaInicio(Layer):
106     is_event_handler = True
```

```
    def __init__(self):
        super().__init__()
        self.add(MiEtiqueta('Pulsa botón de ratón para iniciar', 400, 325))

    def on_mouse_release(self, x, y, buttons, modifiers):
        a = MiHUD()
        director.replace(Scene(MiCapa(a), a))


class CapaGanador(Layer):
    def __init__(self):
        super().__init__()
        self.add(MiEtiqueta('¡Enhorabuena, has ganado!', 400, 325))
        f1 = lambda : director.replace(Scene(CapaInicio()))
        self.do(Delay(3) + CallFunc(f1))


class CapaGameOver(Layer):
    def __init__(self):
        super().__init__()
        self.add(MiEtiqueta('GAME OVER', 400, 325))
        f1 = lambda : director.replace(Scene(CapaInicio()))
        self.do(Delay(3) + CallFunc(f1))


class MiCapa(Layer):
    is_event_handler = True

    def on_key_press(self, k, _):
        MiMisil.TECLAS_PULSADAS[k] = 1

    def on_key_release(self, k, _):
        MiMisil.TECLAS_PULSADAS[k] = 0

    def __init__(self, HUD):
        super().__init__()
        global musica_de_fondo
        self.mi_HUD = HUD
        self.ancho_ventana, self.alto_ventana = director.get_window_size()
        self.man_col = CollisionManagerBruteForce()
        self.crear_misil()
        self.crear_aliens()
        self.schedule(self.update)

        musica_de_fondo = Effect('.\GameLoops\MelodicHouse_1.wav')
        musica_de_fondo.sound.set_volume(0.1)
        musica_de_fondo.sound.play(-1)

        self.image = Sprite(load('universo_2.jpg'), (400,325))
        self.add(self.image, z = 0)

    def crear_misil(self):
        self.misil = MiMisil('mi_misil.png', self.ancho_ventana * 0.5, 50)
        self.add(self.misil, z = 1)

    def crear_aliens(self):
        for i in range(6):
            alien = MiAlien('mi_ovni_1.png',self.ancho_ventana * 0.5, 600 - i*40)
            self.add(alien, z = 1)

    def update(self, dt):
        sin_ovnis = True
        for data in self.children:
            if isinstance(data[1], MiAlien):
                sin_ovnis = False
        if sin_ovnis:
            director.replace(Scene(CapaGanador()))
```

```
174
175         self.man_col.clear()
176         for _, node in self.children:
177             if isinstance(node, MiObjeto):
178                 node.update(dt)
179         for _, node in self.children:
180             if isinstance(node, MiObjeto):
181                 self.man_col.add(node)
182
183         self.collide(self.misil)
184
185     def collide(self, node):
186         if node is not None:
187             for other in self.man_col.iter_colliding(node):
188                 sonido_explosion = Effect('explosion.wav')
189                 sonido_explosion.sound.set_volume(0.05)
190                 sonido_explosion.play()
191                 if self.children.count((1, other)) != 0:
192                     other.kill()
193                     if isinstance(other, MiAlien):
194                         self.mi_HUD.puntos += 20
195                         self.mi_HUD.update()
196                 if self.children.count((1, node)) != 0:
197                     node.kill()
198                     if node.y == 50:
199                         self.mi_HUD.vidas -= 1
200                         self.mi_HUD.update()
201                 seq = ImageGrid(load('secuencia_explosion.png'), 4, 4)
202                 anim = Animation.from_image_sequence(seq, 0.05, False)
203                 self.mi_sprite = Sprite(anim, (other.x, other.y))
204                 self.add(self.mi_sprite)
205                 self.do(Delay(0.8) + CallFunc(self.mi_sprite.kill))
206
207     def on_exit(self):
208         self.pause_scheduler()
209         if 'musica_de_fondo' in globals():
210             musica_de_fondo.sound.stop()
211
212
213 class MiHUD(Layer):
214     def __init__(self):
215         super().__init__()
216         self.vidas = 3
217         self.puntos = 0
218         self.update()
219
220     def update(self):
221         self.children = []
222         texto_vidas = 'Vidas: ' + str(self.vidas)
223         etiqueta_vidas = MiEtiqueta(texto_vidas, 50, 640, (0,255,0,255))
224         texto_puntos = 'Puntos: ' + str(self.puntos)
225         etiqueta_puntos = MiEtiqueta(texto_puntos, 680, 640, (0,255,255,255))
226         self.add(etiqueta_vidas)
227         self.add(etiqueta_puntos)
228         if self.vidas == 0:
229             director.replace(Scene(CapaGameOver()))
230
231
232 def cierra_ventana():
233     if 'musica_de_fondo' in globals():
234         musica_de_fondo.sound.stop()
235     director.window.close()
236
237
238 if __name__ == '__main__':
239     director.init(caption='Protege La Tierra',
240                   width=800, height=650, audio_backend='sdl')
241     director.window.on_close = cierra_ventana
242     director.run(Scene(CapaInicio()))
243
```

La escena principal del juego tendrá ahora el siguiente aspecto, donde presento dos instantes de su desarrollo:

Comentarios sobre el código:

- Ahora añadiremos música de fondo[61] en la escena principal del juego, para lo cual actuaremos de forma similar a la que vimos en

61 He elegigo MelodicHouse_1.wav dentro de la carpeta GameLoops, que contiene varias melodías. El lector puede sustituirla por una que le guste más si lo considera oportuno.

ejemplo_animacion_y_sonido_2.py, es decir, reproduciendo de forma ininterrumpida la música al iniciar la instancia de MiCapa y definiendo una función cierra_ventana() para que en el caso de cerrar la ventana principal (se lo indicamos el L241 al sobrescribir el método on_close() de director.window) no siga la reproducción. Definimos la variable musica_de_fondo como global para poder posteriormente acceder a ella desde el exterior de la clase MiCapa.

- Además de ello sobrescribimos la clase on_exit() de MiCapa para, además de pausar el planificador, parar la reproducción de la música cuando pasemos a cualquiera de las restantes escenas.

- Para añadir un fondo al juego incorporamos a la escena principal un sprite con la imagen universo.jpg, que tiene el mismo tamaño que la ventana principal (800 x 650), y lo colocamos justo en el centro con coordenadas (400, 325). La adición del fondo podría haberse logrado también creando una capa para albergarlo.

Consideraciones finales:

- La dificultad del arcade que hemos programado no es muy alta. Si quisiésemos aumentarla podríamos por ejemplo incrementar la frecuencia con la que nos disparan los ovnis, cambiando en L93 en valor 990 por uno más bajo[62]. También lo lograríamos incrementando la velocidad de los ovnis cambiando en L68 el valor[63] -100. Incluso podemos combinar la variación de estos dos parámetros para logar varios niveles de dificultad, a los que iríamos pasando a medida que superamos el anterior. Como sugerencia final para aumentar la complejidad del juego está la posibilidad de disponer de un número limitado de misiles para lanzar. Animo al lector a intentar generar un juego más completo, con las sugerencias indicadas y/o con las que le hayan surgido a nivel personal.

- El ejemplo que hemos puesto se limita a una sola pantalla donde nos podemos mover solamente dentro de ese marco estático. Para desarrollar juegos que superen esa limitación aprenderemos las herramientas necesarias en el siguiente capítulo.

---

62 Por ejemplo, si lo sustituimos por 975 notaremos al instante que la complicación para superar la prueba se incrementa bastante.

63 Probar el cambio de velocidad si colocamos en su lugar -500.

3

MÁS SOBRE COCOS2D

En este capítulo seguiremos viendo elementos de cocos2d que nos permitirán ampliar las posibilidades de nuestros potenciales videojuegos.

3.1 SCROLL

Ya hemos realizado nuestro primer videojuego 2D, que transcurre en los límites de una ventana. Si queremos construir juegos que operen en un espacio mayor tenemos dos opciones principales:

1. Crear varias escenas de una sola pantalla e ir moviéndonos entre ellas.
2. Crear una escena de un tamaño superior a una pantalla e ir desplazándonos por ella.

Para realizar la primera opción conocemos en este momento muchos de los elementos necesarios. Para la segunda necesitamos poder desplazarnos por la escena con una "cámara" que siga los movimientos del actor en el eje x (sin considerar rotaciones), siendo lo que ve esa cámara lo que se representa en la ventana. Es la técnica denominada **scrolling**, y realizarla se denomina "hacer scroll"[64].

En el módulo **cocos.layer.scrolling** tendremos los elementos para trabajar en el caso de querer crear un videojuego con scroll. Hay varios elementos fundamentales:

64 El término scroll lo traduciríamos como desplazamiento, pero usaremos habitualmente la palabra en inglés.

- Las **capas** implicadas, que serán instancias de **ScrollableLayer**, un tipo de capa que nos permite realizar scroll sobre ella.
- El **manejador** de scroll, una instancia de la clase **ScrollingManager**, que coordinará (tras añadirlas previamente) las capas sobre las que se puede realizar scroll.
- El **foco**, ligado a los atributos fx y fy de ScrollingManager, nos marca el punto central a partir del que se realiza la vista, y por tanto lo que aparece en pantalla.

Tanto ScrollableLayer como ScrollingManager están basadas en Layer (módulo cocos.layer.base_layers), que a su vez deriva de CocosNode (módulo cocos.cocosnode).

Si en ScrollableLayer se define el atributo px_width también se debe definir px_height; el scroll se limitará a mostrar sólo las áreas con coordenadas x e y que cumplan:

origin_x <= x < = px_width
origin_y <= y <= px_height

Los valores de origin_x y origin_y toman valor 0 por defecto.

Si px_width no está definido, entonces la capa no limitará el scroll.

Entre otras cosas ScrollingManager limita el scroll para que se respeten todas las restricciones de visibilidad impuestas por las instancias ScrollableLayer; al menos una de ellas debe definir una restricción para que el scroll sea limitado. La representación puede reducirse al rectángulo de una ventana específica mediante el parámetro viewport.

Destacaremos los siguientes métodos de ScrollingManager:

- **add**(child, z=0, name=None)

  Añade el objeto hijo child (de tipo CocosNode, generalmente una instancia de ScrollableLayer) y actualiza la vista y el foco del manejador de scroll. El parámetro z (número entero) indica el nivel (eje z) en el que se coloca el hijo, mientras que name (una cadena) nos proporciona opcionalmente su nombre.

- **force_focus**(fx, fy)

  Fuerza al manejador de scroll a focalizar en el punto de coordenadas (fx, fy), independientemente de los límites visibles de cualquier capa que gestione. Los parámetros fx y fy son de tipo entero.

- **screen_to_world**(x, y)

  Transforma las coordenadas "screen" (tienen como referencia la esquina inferior izquierda de la ventana, luego son relativas a ella) en las coordenadas "world" (su referencia es la esquina inferior izquierda del propio manejador, por lo que las consideraremos coordenadas absolutas), que nos son devueltas en forma de tupla de enteros. Importante para las transformaciones de vista, capa y pantalla.

- **set_focus**(fx, fy, force=False)

  Hace que el punto de coordenadas (fx, fy) se muestre lo más cerca posible del centro de la vista. Cambia los objetos hijo manejados para que el punto (fx, fy) en las coordenadas "world" se vea tan cerca del centro de la vista como sea posible, mientras que al mismo tiempo no muestra áreas fuera de los límites de los objetos hijo. Los parámetros fx y fy son de tipo entero, mientras que force es un booleano (por defecto de valor False) que, si es True, fuerza la actualización del enfoque aunque el punto de enfoque o la escala no hayan cambiado.

- **world_to_screen**(x, y)

  Transforma las coordenadas "world" en las coordenadas "screen", que nos son devueltas en forma de tupla de enteros. Importante para las transformaciones de vista, capa y pantalla.

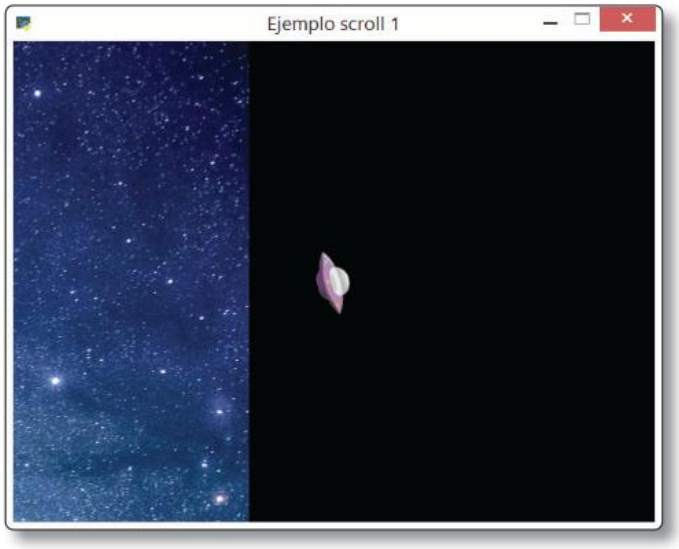
Para profundizar un poco más consultaremos el módulo cocos.layer.scrolling en el Apéndice B.

Veamos a continuación dos ejemplos del uso de scroll. En el primero conduciremos un ovni por la pantalla (mediante las teclas de desplazamiento) teniendo como fondo una imagen del universo almacenada en **universo_original.jpg**. El código lo tenemos en **ejemplo_scroll_1.py**:

```
1
2  from pyglet.window import key
3  from cocos.director import director
4  from cocos.scene import Scene
5  from cocos.layer import Layer, ScrollableLayer, ScrollingManager
6  from cocos.sprite import Sprite
7  from cocos.actions import Driver
8
9  class MueveOvni(Driver):
10     def __init__(self):
11         super().__init__()
12
13     def step(self, dt):
14         super().step(dt)
15         self.target.rotation += (teclas[key.RIGHT] - teclas[key.LEFT]) * 150 * dt
16         self.target.acceleration = (teclas[key.UP] - teclas[key.DOWN]) * 400
17
```

```
        manejador_scroll.force_focus(self.target.x, self.target.y)


if __name__ == '__main__':

    teclas = key.KeyStateHandler()

    director.init(width = 640, height = 480, caption = 'Ejemplo scroll 1')
    director.window.set_location(300, 80)
    director.window.push_handlers(teclas)

    capa_sprite = ScrollableLayer()
    mi_ovni = Sprite('mi_ovni_2.png')
    mi_ovni.max_forward_speed = 300
    mi_ovni.max_reverse_speed = -300
    capa_sprite.add(mi_ovni)

    capa_fondo =ScrollableLayer()
    mi_fondo = Sprite('universo_original.jpg')
    mi_fondo.position = (0, 0)
    capa_fondo.add(mi_fondo)

    manejador_scroll = ScrollingManager()
    manejador_scroll.add(capa_fondo)
    manejador_scroll.add(capa_sprite)

    mi_ovni.do(MueveOvni())

    mi_escena = Scene(manejador_scroll)
    director.run(mi_escena)

```

Al ejecutarlo podremos mover el ovni incluso fuera de los límites de la imagen de fondo, como se muestra a continuación:



Comentarios sobre el código:

- He distribuido el código de una forma poco habitual hasta ahora, creando una clase MueveOvni basándome en la acción Driver, que nos permitirá desplazar el ovni conduciéndolo. Su método step() se ejecutará en cada fotograma, y en él fijaremos el valor de los atributos rotation y acceleration en base a las teclas de desplazamiento pulsadas. En L18 forzamos al manejador de scroll (que veremos a continuación) a "apuntar" al ovni.
- El resto del código está en el bloque principal. En L23 creamos una instancia de la clase KeyStateHandler (en el módulo pyglet.window.key), y en L27 la incluimos en los manejadores de director.window. Mediante ella rastreamos las posibles pulsaciones de teclas (KeyStateHandler crea un diccionario donde a cada tecla le asocia True o False dependiendo de si está o no pulsada), que usaremos en L15-16 para calcular.
- En L25-26 inicializamos el director y colocamos la ventana (de tamaño 640 x 480) en las coordenadas (300,800).
- En L29-33 creamos una capa con scroll (instancia de ScrollableLayer) a la que añadimos el sprite del ovni, tras configurar su velocidad máxima, tanto hacia adelante como hacia atrás.
- En L35-38 creamos otra capa con scroll a la que añadimos, en la posición (0,0), un sprite con la imagen del universo.
- En L40-42 creamos el manejador de scroll y añadimos las dos capas creadas. Al ser manejador_scroll una variable global será accesible desde dentro de la definición de MueveOvni.
- En L44 ejecutamos la acción MueveOvni, que será la que controle de forma periódica el movimiento de nuestro ovni.
- En L46-47 creamos la escena principal (con el manejador de scroll, que recordemos está basado en Layer) y la ejecutamos.

En algunos casos querremos que el scroll se limite únicamente a la zona en la que existe la imagen del universo, no pudiendo sobrepasar sus límites. Una forma de conseguirlo la tenemos en **ejemplo_scroll_2.py**:

```
1
2 from pyglet.window import key
3 from cocos.director import director
4 from cocos.scene import Scene
5 from cocos.layer import Layer, ScrollableLayer, ScrollingManager
6 from cocos.sprite import Sprite
7 from cocos.actions import Driver
8
```

```
class MueveOvni(Driver):
    def __init__(self):
        super().__init__()
        self.ancho_fondo = mi_fondo.width
        self.alto_fondo = mi_fondo.height
        self.x_max = self.ancho_fondo - mi_ovni.height / 2
        self.x_min = mi_ovni.height / 2
        self.y_max = self.alto_fondo - mi_ovni.height / 2
        self.y_min = mi_ovni.height / 2

    def step(self, dt):
        super().step(dt)
        self.target.rotation += (teclas[key.RIGHT] - teclas[key.LEFT]) * 150 * dt
        self.target.acceleration = (teclas[key.UP] - teclas[key.DOWN]) * 400

        manejador_scroll.set_focus(self.target.x, self.target.y, force = True)

        if self.target.x > self.x_max:
            self.target.speed = 0
            self.target.x = self.x_max
        if self.target.x < self.x_min:
            self.target.speed = 0
            self.target.x = self.x_min
        if  self.target.y < self.y_min:
            self.target.speed = 0
            self.target.y = self.y_min
        if self.target.y > self.y_max:
            self.target.speed = 0
            self.target.y = self.y_max


if __name__ == '__main__':

    teclas = key.KeyStateHandler()

    director.init(width = 640, height = 480, caption = 'Ejemplo scroll 2')
    director.window.set_location(300,80)
    director.window.push_handlers(teclas)

    capa_sprite = ScrollableLayer()
    mi_ovni = Sprite('mi_ovni_2.png')
    mi_ovni.position = (640, 436)
    mi_ovni.max_forward_speed = 300
    mi_ovni.max_reverse_speed = -300
    capa_sprite.add(mi_ovni)

    capa_fondo = ScrollableLayer()
    capa_fondo.px_width = 1280
    capa_fondo.px_height = 873
    mi_fondo = Sprite('universo_original.jpg')
    mi_fondo.position = (640, 436)
    capa_fondo.add(mi_fondo)

    manejador_scroll = ScrollingManager()
    manejador_scroll.add(capa_fondo)
    manejador_scroll.add(capa_sprite)

    mi_ovni.do(MueveOvni())

    mi_escena = Scene(manejador_scroll)
    director.run(mi_escena)

```

Si intentamos superar los límites de la imagen de fondo, no podremos:

Comentarios sobre el código:

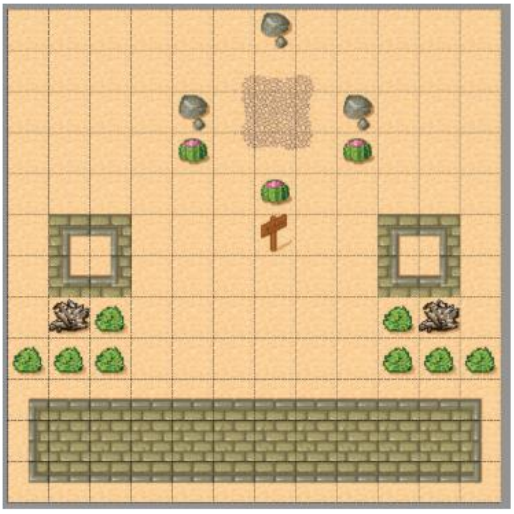
- Crearemos los atributos ancho_fondo y alto_fondo de MueveOvni para contener, respectivamente, la anchura y altura en píxeles de la imagen del universo. Con x_max, x_min, y_max, y_min almacenamos los valores máximos y mínimos que pueden tener las coordenadas del ovni en ambos ejes (he considerado para ello también el tamaño del propio ovni).
- En cada ejecución de step() analizaremos si las coordenadas del ovni superan el límite, en cuyo caso las mantenemos en él, colocando la coordenada correspondiente a su valor máximo y el atributo speed a 0.
- En L56-57 configuramos los valores de px_width y px_height de la capa del fondo.
- Es importante ver que el sprite del universo lo colocamos en las coordenadas (640,436), las mismas en las que hemos insertado el sprite del ovni.

3.2 MAPAS DE BALDOSAS

Los **mapas de baldosas**[65] (tile maps[66]) son elementos importantes en el desarrollo de nuestros juegos 2D. Consisten en una cuadrícula 2D dividida en celdas donde cada una de ellas puede albergar una imagen. Disponemos de otra cuadrícula 2D (llamada **conjunto de patrones**) donde estarán los bloques principales de construcción a partir de los cuales crearemos el mapa.

Veámoslo gráficamente con un ejemplo. Partiendo del siguiente conjunto de patrones:

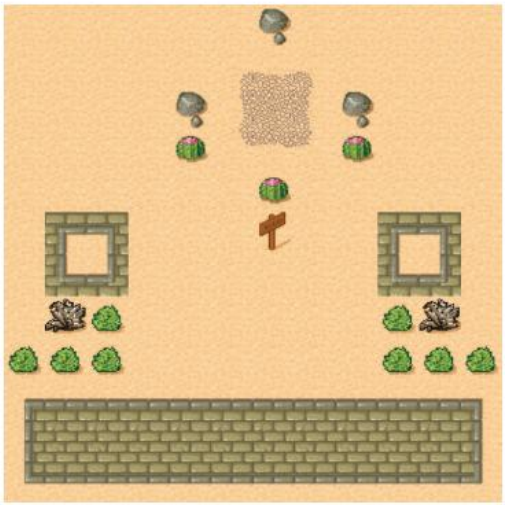
Construimos el mapa de baldosas mostrado a continuación:



65 O mapas de azulejos.

66 Usaré habitualmente tanto la expresión española como la inglesa.

Obteniendo el siguiente aspecto final:



Es una técnica muy útil para almacenar grandes cantidades de información basadas en pequeños trozos de gráficos reutilizables.

Para tratar con mapas de baldosas usaremos el programa Tiled Map Editor[67]. En el Apéndice C se comenta cómo instalarlo y usarlo, además de indicar la forma de crear el mapa anterior[68], por lo que insto al lector a su lectura antes de seguir adelante[69].

Una vez que tengamos el tile map creado y almacenado en un fichero deberemos posteriormente trabajar con él desde nuestro programa, para lo cual usaremos las clases incluidas en el módulo cocos.tiles[70]:

- **MapLayer**, clase base para los mapas.
- **RectMapLayer**, para los mapas (renderizables y en los que se puede hacer scroll) de mosaicos rectangulares.

67 Tiled de forma abreviada. Las tres imágenes superiores han sido obtenidas desde él.

68 Lo usaremos en varios de los siguientes códigos que se presentarán.

69 La inclusión del material sobre Tiled en un apéndice aparte se debe a que considero que aporta claridad a la presentación.

70 Para profundizar en todas sus características consultar el Apéndice B.

- **HexMapLayer**, para los mapas (renderizables y en los que se puede hacer scroll) de mosaicos hexagonales.
- **RegularTesselationMap**, para los mapas teselados regularmente y que permite acceder a sus celdas mediante los índices (i,j).
- **RectMap**, para los mapas rectangulares.
- **HexMap**, para los mapas hexagonales.
- **Cell**, clase base para celdas de mapas rectangulares y hexagonales.
- **RectCell**, para las celdas de un mapa rectangular, que son rectángulos.
- **HexCell**, para las celdas de un mapa hexagonal, que son hexágonos regulares.
- **Resource**, para cargar recursos del mapa de mosaicos desde un fichero XML.
- **Tile**, para tratar las baldosas, que contienen una imagen y algunas propiedades opcionales.
- **TileSet**, para contener un conjunto de objetos baldosa referenciados por algún identificador.
- **TmxObject**, que representa un objeto en una capa de objetos TMX.
- **TmxObjectLayer**, para tratar con una capa compuesta por formas básicas primitivas.

También usaremos las siguientes funciones del módulo:

- **load**(filename), carga recursos desde un fichero XML[71] de nombre filename.
- **load_tiles**(filename), que carga los recursos del mapa desde un fichero XML de nombre filename.
- **load_tmx**(filename), carga los recursos del mapa desde un fichero TMX de nombre filename.

71 Tiled almacena por defecto los datos en formato XML.

Conseguiremos cargar nuestro tile map mediante la función load() del módulo cocos.tiles. Veamos un ejemplo de ello con **ejemplo_tile_map_1.py**:

```
1
2 from cocos.tiles import load
3 from cocos.layer import ScrollingManager
4 from cocos.director import director
5 from cocos.scene import Scene
6
7 if __name__ == '__main__':
8     ventana = director.init(width=384, height=384, caption="Cargar tile map")
9     ventana.set_location(600, 200)
10
11    mi_mapa = load("mapa1.tmx")["capa0"]
12
13    manejador_scroll = ScrollingManager()
14    manejador_scroll.add(mi_mapa)
15
16    director.run(Scene(manejador_scroll))
17
```

Su salida genera la siguiente ventana:

En el código observamos que:

- La ventana creada tiene un tamaño de 384 x 384 puntos, ya que ese es el tamaño del tile map creado.
- La variable mi_mapa apunta a un objeto de tipo MapLayer, que deriva de la clase ScrollableLayer, por lo que he tenido que usar un manejador

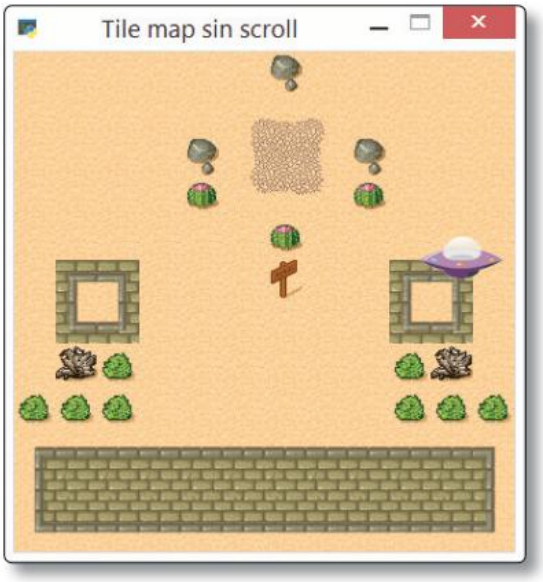
de scroll como paso previo a su inclusión en la escena a pesar de que no realizaremos ningún tipo de scroll en el ejemplo.

Al ejecutar **ejemplo_tile_map_2.py**[72] tendremos la posibilidad de desplazar un ovni sobre el mapa que hemos creado:

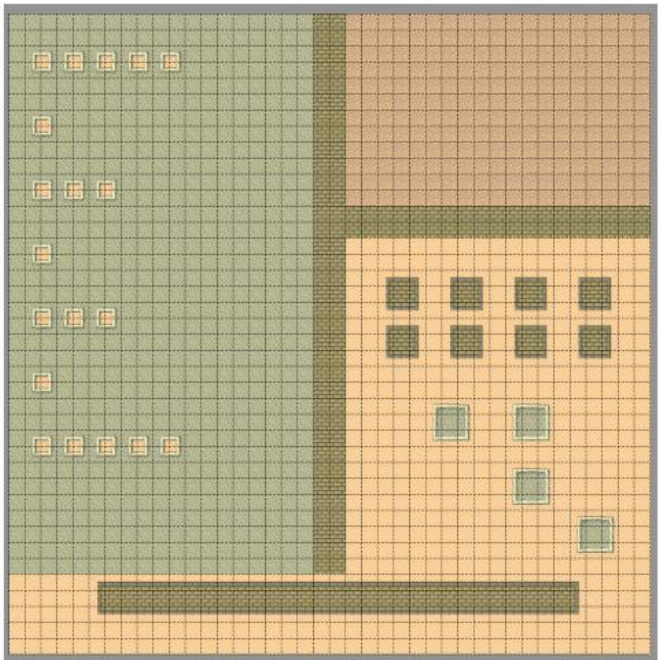
```
1
2 from pyglet.window import key
3 from cocos.director import director
4 from cocos.scene import Scene
5 from cocos.layer import ScrollableLayer, ScrollingManager
6 from cocos.tiles import load
7 from cocos.sprite import Sprite
8 from cocos.actions import Move
9
10 class MueveOvni(Move):
11     def __init__(self):
12         super().__init__()
13
14     def step(self, dt):
15         super().step(dt)
16         velocity_x = 200 * (teclas[key.RIGHT] - teclas[key.LEFT])
17         velocity_y = 200 * (teclas[key.UP] - teclas[key.DOWN])
18         self.target.velocity = (velocity_x, velocity_y)
19         manejador_scroll.set_focus(self.target.x, self.target.y)
20
21
22 if __name__ == '__main__':
23   teclas = key.KeyStateHandler()
24
25   ventana = director.init(width=384, height=384, caption='Tile map sin scroll')
26   ventana.set_location(600, 200)
27   director.window.push_handlers(teclas)
28
29   capa_ovni = ScrollableLayer()
30   mi_ovni = Sprite('mi_ovni_2.png')
31   mi_ovni.position = (200, 125)
32   mi_ovni.velocity = (0, 0)
33   capa_ovni.add(mi_ovni)
34
35   mi_fondo = load('mapa1.tmx')['capa0']
36
37   manejador_scroll = ScrollingManager()
38   manejador_scroll.add(mi_fondo)
39   manejador_scroll.add(capa_ovni)
40
41   mi_ovni.do(MueveOvni())
42
43   mi_escena = Scene(manejador_scroll)
44   director.run(mi_escena)
45
```

72 El análisis del código, junto al resto de los que aparecerán en el tema, queda como ejercicio para el lector.

Su salida es:

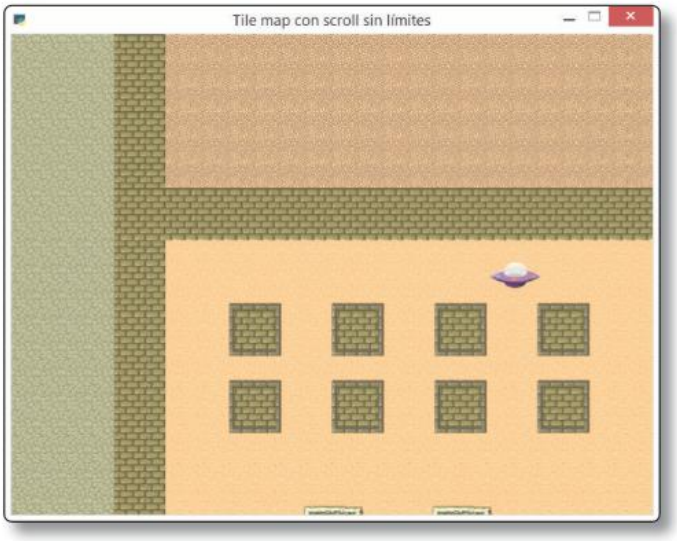


Si ahora creamos un mapa mucho mas grande como **mapa2.tmx** (40 x 40 patrones de 32 x 32 puntos para un total de 1280 x 1280 puntos):



En **ejemplo_tile_map_3.py** lo recorreremos mediante una ventana más grande (800 x 600 puntos):

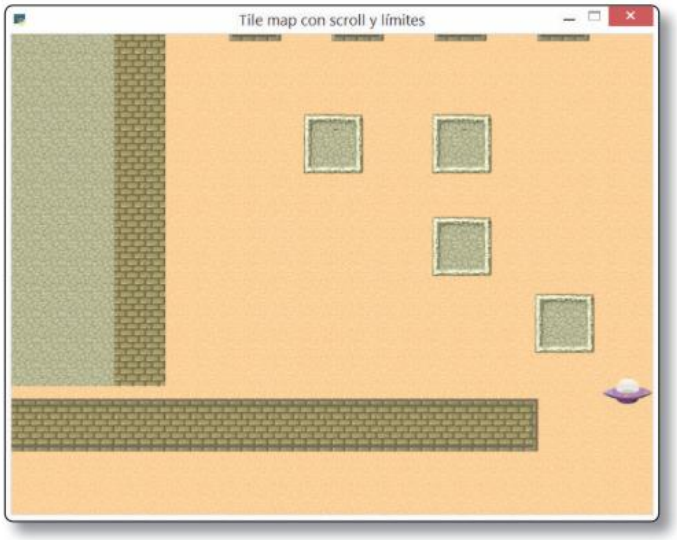
```
1
2 from pyglet.window import key
3 from cocos.director import director
4 from cocos.scene import Scene
5 from cocos.layer import ScrollableLayer, ScrollingManager
6 from cocos.tiles import load
7 from cocos.sprite import Sprite
8 from cocos.actions import Move
9
10
11 class MueveOvni(Move):
12     def __init__(self):
13         super().__init__()
14
15     def step(self, dt):
16         super().step(dt)
17         velocity_x = 200 * (teclas[key.RIGHT] - teclas[key.LEFT])
18         velocity_y = 200 * (teclas[key.UP] - teclas[key.DOWN])
19         self.target.velocity = (velocity_x, velocity_y)
20
21         manejador_scroll.set_focus(self.target.x, self.target.y)
22
23
24 if __name__ == '__main__':
25   teclas = key.KeyStateHandler()
26
27   ventana = director.init(width=800, height=600,
28                           caption = 'Tile map con scroll sin límites')
29   ventana.set_location(500, 200)
30   director.window.push_handlers(teclas)
31
32   capa_ovni = ScrollableLayer()
33   mi_ovni = Sprite('mi_ovni_2.png')
34   mi_ovni.position = (650, 250)
35   mi_ovni.velocity = (0, 0)
36   capa_ovni.add(mi_ovni)
37
38   mi_fondo = load('mapa2.tmx')['capa0']
39
40   manejador_scroll = ScrollingManager()
41   manejador_scroll.add(mi_fondo)
42   manejador_scroll.add(capa_ovni)
43
44   mi_ovni.do(MueveOvni())
45
46   mi_escena = Scene(manejador_scroll)
47   director.run(mi_escena)
48
```



En **ejemplo_tile_map_4.py** ponemos límites al desplazamiento del ovni para no poder sobrepasar los propios del mapa:

```
1
2  from pyglet.window import key
3  from cocos.director import director
4  from cocos.scene import Scene
5  from cocos.layer import ScrollableLayer, ScrollingManager
6  from cocos.tiles import load
7  from cocos.sprite import Sprite
8  from cocos.actions import Move
9
10 class MueveOvni(Move):
11     def __init__(self):
12         super().__init__()
13         self.ancho_fondo = mi_fondo.px_width
14         self.alto_fondo = mi_fondo.px_height
15         self.x_max = self.ancho_fondo - mi_ovni.width / 2
16         self.x_min = mi_ovni.width / 2
17         self.y_max = self.alto_fondo - mi_ovni.height / 2
18         self.y_min = mi_ovni.height / 2
19
20     def step(self, dt):
21         super().step(dt)
22         if self.x_min <= self.target.x <= self.x_max and\
23            self.y_min <= self.target.y <= self.y_max:
24             velocity_x = 200 * (teclas[key.RIGHT] - teclas[key.LEFT])
25             velocity_y = 200 * (teclas[key.UP] - teclas[key.DOWN])
26         else:
27             if self.target.x >= self.x_max:
28                 velocity_x = velocity_y = 0
29                 self.target.x = self.x_max
30             if self.x_min >= self.target.x:
31                 velocity_x = velocity_y = 0
32                 self.target.x = self.x_min
33             if self.target.y >= self.y_max:
34                 velocity_x = velocity_y = 0
35                 self.target.y = self.y_max
```

```
            if self.y_min >self.target.y:
                velocity_x = velocity_y = 0
                self.target.y = self.y_min

        self.target.velocity = (velocity_x, velocity_y)
        manejador_scroll.set_focus(self.target.x, self.target.y)


if __name__ == '__main__':
    teclas = key.KeyStateHandler()

    ventana = director.init(width=800, height=600,
                caption = 'Tile map con scroll y límites')
    ventana.set_location(500, 200)
    director.window.push_handlers(teclas)

    capa_ovni = ScrollableLayer()
    mi_ovni = Sprite('mi_ovni_2.png')
    mi_ovni.position = (650, 250)
    mi_ovni.velocity = (0, 0)
    capa_ovni.add(mi_ovni)

    mi_fondo = load('mapa2.tmx')['capa0']

    manejador_scroll = ScrollingManager()
    manejador_scroll.add(mi_fondo)
    manejador_scroll.add(capa_ovni)

    mi_ovni.do(MueveOvni())

    mi_escena = Scene(manejador_scroll)
    director.run(mi_escena)


```



Hasta este momento los mapas de baldosas han actuado como un simple fondo, pero nos gustaría que varios de sus elementos fuesen "sólidos". Pensemos en el típico videojuego de plataformas, donde hay elementos del mapa (las plataformas) que impiden el paso del actor y limitan su movimiento. Para conseguir esa solidez de elementos del tile map veremos el siguiente tema.

3.3 MAPAS DE COLISIÓN

Ya hemos visto que si en una escena tenemos a un actor moviéndose podremos actualizar su posición fotograma a fotograma de la siguiente manera:

velocidad = velocidad + aceleración * dt
posición = posición + velocidad * dt

Con dt indicamos el diferencial de tiempo entre los dos fotogramas. Lo indicado funciona perfectamente si el actor no se encuentra con ningún obstáculo, como puede ser un muro. En el caso de sí encontrarlo generalmente querremos que ocurran algunas de las opciones siguientes:

1. Detener el cambio de posición para que el actor no se superponga con el obstáculo.
2. Cambiar la velocidad del actor: que rebote, se detenga...
3. Realizar alguna acción sobre el actor dependiendo del tipo de obstáculo con el que colisionemos.

Representaremos al actor y a los obstáculos como rectángulos de lados paralelos a los ejes de cara a hacer los cálculos. Usaremos lo que se denominan **mapas de colisión**, que de alguna manera conocen la posición de los obstáculos con los que el actor puede colisionar.

Algunos mapas de colisión[73] (RectMapCollider y RectMapWithProps Collider) están pensados para usarse en los mapas de baldosas (tiled maps) que acabamos de ver, mientras que otros (TmxObjectMapCollider) pueden usarse sin ellos ya que tratan con un tipo de objetos geométricos (TmxObject) que veremos más adelante.

73 Para saber más sobre las citadas clases consultar el Apéndice B.

Tendremos por tanto tres variantes de mapas de colisión[74]:

- **RectMapCollider**: los obstáculos son las baldosas rectangulares (rectangular tiles) no vacías que están en una capa RectMapLayer.
- **RectMapWithPropsCollider**: los obstáculos son baldosas rectangulares (rectangular tiles) en una capa RectMapLayer. Las celdas usan las propiedades left, right, top y bottom para indicar que cara/s bloquean el movimiento.
- **TmxObjectMapCollider**: los obstáculos son objetos TmxObject dentro de una capa TmxObjectLayer destinados a bloquear el movimiento.

Los mapas de colisión, a través del método collide_map() que veremos a continuación, recibirán:

- La velocidad del actor, junto a su rectángulo anterior y al potencial rectángulo posterior.
- Una capa de mapa (map layer) que contiene una serie de obstáculos.

Y harán lo siguiente:

- Devolver la actualización correcta de la velocidad y la posición del actor.
- Llamar a determinados métodos si detectan que el actor ha chocado con un obstáculo.
- Indicar si ha habido alguna colisión en el eje x e/o y.

La forma en la que la velocidad del actor cambia en una colisión es manejada por el método **on_bump_handler**() del mapa de colisión, en el que podremos colocar código personalizado o usar el ya disponible en los siguientes métodos:

- **on_bump_bounce**(), hace que el actor rebote al colisionar con un obstáculo.
- **on_bump_stick**(), detiene cualquier movimiento tras la colisión.
- **on_bump_slide**(), bloquea el movimiento solo en una determinada dirección de un eje.

74 A veces nos referiremos a ellos como modelos de colisión.

El comentado método **collide_map**()[75], presente en las clases que definen los mapas de colisión (RectMapCollider, RectMapWithPropsCollider y TmxObjectMapCollider), será el elemento fundamental de cara a trabajar con mapas de colisión. Tiene la siguiente sintaxis:

```
collide_map(maplayer, last, new, vx, vy)
```

Usando el método collide_map() del mapa de colisión limitaremos el posible movimiento del actor en base a los obstáculos que pueda encontrar. Se consideran las posiciones (mediante los rectángulos comentados con anterioridad) en dos instantes de tiempo separados por dt, una "actual" (last) y otra "potencial" (new). Se asume que en la posición inicial no se colisiona con ningún objeto, y que dt es suficientemente pequeño para que no exista ningún objeto entero entre las dos posiciones. Con maplayer indicamos una capa con objetos sólidos[76] con los que poder colisionar, y con vx y vy los valores de velocidad usados para todo el cálculo.

Si el rectángulo new se superpone a cualquier elemento de maplayer, el método collide_map():

- Modifica new para estar lo más cerca del new original pero sin solapar ningún objeto de maplayer.
- Devuelve unos modificados (vx, vy) procedentes del método on_bump_handler().
- Asigna a los atributos bumped_x y/o bumped_y el valor True si, respectivamente, ha ocurrido una colisión horizontal o vertical.
- Llama al método collide_top(), collide_bottom(), collide_left() o collide_right() si la colisión se ha realizado, respectivamente, en la parte superior, inferior, izquierda o derecha del actor.
- Forzará el desplazamiento mínimo en los dos ejes (si la colisión se ha detectado en ambos) para que deje de haber colisión.

Si el rectángulo new no se superpone a ningún elemento de maplayer el método collide_map():

- No modifica new.
- Devuelve (vx, vy) sin modificar.

---

75 Lo denominaremos manejador de colisión en algunos instantes.

76 Recordemos las dos formas básicas de indicar cuales son los elementos sólidos de la capa: mediante las celdas de mapas de baldosas o mediante objetos TmxObject.

- Asigna a los atributos bumped_x y bumped_y el valor False.
- No llama a ningún método.

Con la ayuda de collide_map() modificaremos en nuestro código la velocidad y posición del actor (al margen de poder realizar operaciones adicionales) en base a los obstáculos que tengamos.

Para incluir la funcionalidad de los mapas de colisión a la clase del actor podemos hacerlo básicamente de dos maneras:

1. Como componentes (atributos) de ella, generalmente pasando el mapa de colisión y la capa de mapa (mapcollider y maplayer) en el método __ init__() de la clase del actor. Posteriormente usaremos el método collide_ map() de mapcollider pasando a maplayer como uno de sus argumentos.

2. Con un estilo mixto, usando RectMapCollider (o una de sus subclases indicadas) como clase base secundaria para el actor. Podremos acceder directamente desde la clase al método collide_map(), y deberíamos poder acceder de alguna manera a la capa de mapa para pasarla como parámetro.

No obstante, podemos también usar la función **make_collision_handler**() del módulo cocos.mapcoliders[77], que recibe como parámetros un mapa de colisión y una capa de mapa y nos devuelve una función equivalente a collide_map() pero a la que no tenemos ya que pasar el mapa de colisión al tenerlo "incorporado"[78]. Tendremos la posibilidad de almacenar esa función en un atributo de la clase del actor, o incluso en un atributo de la instancia particular de nuestro actor. En los códigos que presentaré más adelante opto mayoritariamente por esta última opción.

Para tener una instancia (actor) funcional al trabajar con un mapa de colisión debemos **obligatoriamente**[79] definir el comportamiento de su velocidad en una colisión, algo que realiza el método on_bump_handler(). Si queremos usar alguna de las variantes que tenemos por defecto, nos valdrá con poner:

```
mi_mapa_colision.on_bump_handler = mi_mapa_colision.variante
```

Donde variante puede tomar el valor on_bump_bounce,on_bump_stick u on_bump_slide.

---

77 Ver Apéndice B.

78 Por lo tanto solamente le pasaremos los 4 parámetros restantes.

79 En caso contrario se lanzará una excepción.

También podríamos pasarlo en tiempo de instanciación:

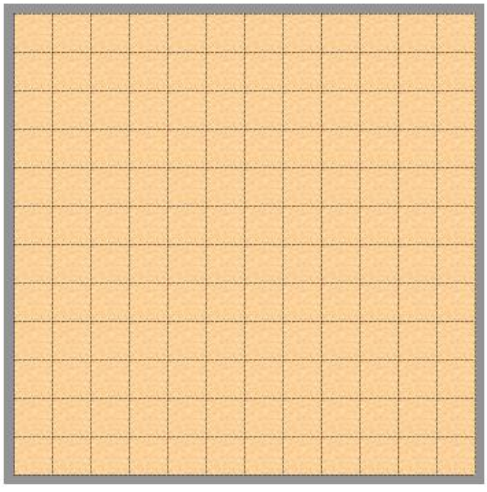
```
mi_mapa_colision = MapaColision(variante)
```

De esta manera variante tiene los posibles valores ya conocidos y MapaColisión será RectMapCollider, RectMapWithPropsCollider o TmxObjectMapCollider.

En el caso de querer colocar nuestro código personalizado en on_bump_handler() podemos crear una subclase de cualquier mapa de colisión y escribirlo allí, o incluso modificar el método en las clases propias de los mapas de colisión que tenemos en el módulo cocos.mapcolliders si ello no nos conlleva perjuicio y lo hacemos de forma totalmente controlada. Si elegimos este último caso, mi consejo es guardar el módulo modificado con otro nombre (que posteriormente importaremos en nuestro programa) y mantener el original siempre sin cambios[80].

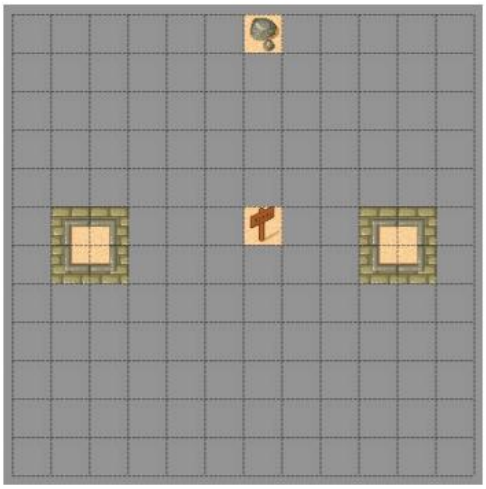
Veamos un primer ejemplo que ponga en práctica varios de los conceptos indicados y nos aporte claridad en su comprensión. Trabajaremos sobre un pequeño mapa de baldosas (**mapa_base.tmx**) que contendrá dos capas. En una (capa0) tendremos el fondo y en otra (capa1) las baldosas que se comportarán como objetos sólidos, a partir de la cual se creará el mapa de colisión. A continuación se presentan imágenes de las dos capas:

capa0



80 En uno de los códigos presentados con posterioridad lo aplicaré de forma práctica, aunque aplicado a otro método.

capa1



El código (**ejemplo_mapa_colision.py**) es el siguiente:

```
1
2 from pyglet.window import key
3 from cocos.sprite import Sprite
4 from cocos.scene import Scene
5 from cocos.layer import ScrollableLayer, ScrollingManager
6 from cocos.tiles import load_tmx
7 from cocos.mapcolliders import RectMapCollider, make_collision_handler
8 from cocos.actions import Action
9 from cocos.director import director
10
11 class MiNave(Sprite):
12     def __init__(self, image):
13         super().__init__(image)
14
15     def update(self, dt):
16         vx = 200 * (man_tec[key.RIGHT] - man_tec[key.LEFT])
17         vy = 200 * (man_tec[key.UP] - man_tec[key.DOWN])
18
19         dx = vx * dt
20         dy = vy * dt
21
22         antes = self.get_rect()
23         despues = antes.copy()
24         despues.x += dx
25         despues.y += dy
26
27         self.man_col(antes, despues, vx, vy)
28         self.position = despues.center
29
30
31 class Control(Action):
32     def step(self, dt):
33         self.target.update(dt)
34
35
36 if __name__ == '__main__':
37     ventana = director.init(width=384, height=384,
```

```
                          caption='Mapa de colisión')
ventana.set_location(500,300)

man_tec = key.KeyStateHandler()
director.window.push_handlers(man_tec)

manejador_scroll = ScrollingManager()
mi_mapa0 = load_tmx('mapa_base.tmx')['capa0']
mi_mapa1 = load_tmx('mapa_base.tmx')['capa1']
manejador_scroll.add(mi_mapa0)
manejador_scroll.add(mi_mapa1)

capa_personaje = ScrollableLayer()
personaje = MiNave('mi_ovni_2.png')
personaje.position = (200,100)
personaje.do(Control())
capa_personaje.add(personaje)

mapa_colision = RectMapCollider()
mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
personaje.man_col = make_collision_handler(mapa_colision, mi_mapa1)

mi_escena = Scene()
mi_escena.add(manejador_scroll, z=0)
mi_escena.add(capa_personaje, z=1)
director.run(mi_escena)
```

Su ejecución genera la siguiente salida:

Ahora podremos desplazar nuestro ovni por la pantalla, pero no por encima de los 4 objetos que tenemos en la escena, que aparecerán como sólidos.

Comentarios sobre el código:

- En L45-48 cargamos las dos capas en el manejador de scroll (que posteriormente agregaremos en L61 a la escena, junto a la capa que contiene al ovni). Al cargar primero la capa0 será ésta la que actúe como fondo, mientras que la capa1 se colocará encima[81].

- En L56 creamos el mapa de colisión de tipo RectMapCollider, por lo que los obstáculos serán las baldosas no vacías que están en una capa RectMapLayer[82]. En L57 indicamos que al chocar con esas baldosas se detendrá el movimiento.

- En L58 añadimos un atributo a nuestro ovni que contenga el manejador de colisión para el mapa contenido en la capa1 de mapa_base.tmx, que es el que contiene las baldosas sólidas.

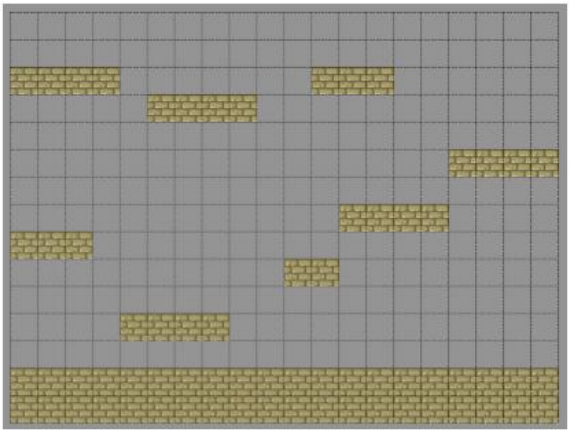
- El método update() de la clase MiNave trabaja se forma similar a la vista en ejemplos anteriores, con el añadido de que ahora tendremos en cuenta la posible colisión del ovni con los elementos sólidos que hemos indicado. Será en L27, donde pasamos al manejador de colisión el rectángulo actual, el potencial y las velocidades en ambos ejes. Si existe colisión modificará (para evitarla) el rectángulo potencial, que es el que posteriormente (L28) usamos para colocar el ovni en pantalla.

Tras este primer ejemplo de los mapas de colisión intentaremos crear la base para un juego de plataformas. Nuevamente lo dividiremos en etapas, concretamente 5, donde en cada una de ellas añadiremos elementos y/o características nuevas.

Para la primera etapa crearemos mediante Tiled un mapa (de nombre **mapa3.tmx**) con una sola capa (**capa0**) consistente en 20 x 15 patrones de 32 x 32 píxeles cada uno, para un tamaño total de 640 x 480 píxeles. Estará basado en el conjunto de patrones tmw_desert.tsx y tendrá la siguiente apariencia:

81 De haberlo hecho al revés sólo visualizaríamos el contenido de la capa 0.

82 Recordemos que la función load_tmx() nos devuelve una instancia de RectMapLayer.

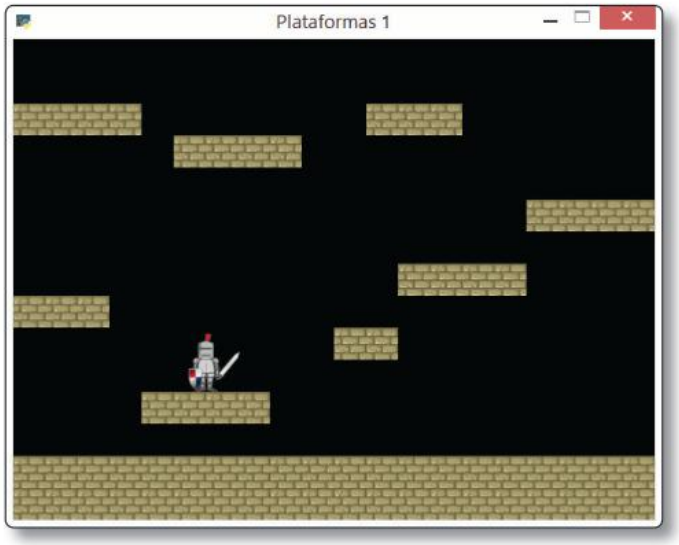


Partiendo de esa base crearemos **plataformas_1.py**:

```
1
2 from pyglet.window import key
3 from cocos.sprite import Sprite
4 from cocos.scene import Scene
5 from cocos.layer import ScrollableLayer, ScrollingManager
6 from cocos.tiles import load_tmx
7 from cocos.mapcolliders import RectMapCollider, make_collision_handler
8 from cocos.actions import Action
9 from cocos.director import director
10
11 class MiGuerrero(Sprite):
12     en_suelo = True
13     VEL_MOV = 200
14     VEL_SALTO = 400
15     GRAVEDAD = -800
16
17     def __init__(self, image):
18         super().__init__(image)
19         self.velocidad = (0,0)
20
21     def update(self, dt):
22         vx, vy = self.velocidad
23
24         vx = (man_tec[key.RIGHT] - man_tec[key.LEFT]) * self.VEL_MOV
25         vy += self.GRAVEDAD * dt
26         if self.en_suelo and man_tec[key.SPACE]:
27             vy = self.VEL_SALTO
28
29         dx = vx * dt
30         dy = vy * dt
31
32         antes = self.get_rect()
33         despues = antes.copy()
34         despues.x += dx
35         despues.y += dy
36
37         self.velocidad = self.man_col(antes, despues, vx, vy)
38         self.en_suelo = (despues.y == antes.y)
39         self.position = despues.center
40
41
42 class Control(Action):
43     def step(self, dt):
44         self.target.update(dt)
45
```

```
46
47 def main():
48     global man_tec
49
50     ventana = director.init(width=640, height=480, caption='Plataformas 1')
51     ventana.set_location(500,300)
52
53     man_tec = key.KeyStateHandler()
54     director.window.push_handlers(man_tec)
55
56     manejador_scroll = ScrollingManager()
57     mi_mapa = load_tmx('mapa3.tmx')['capa0']
58     manejador_scroll.add(mi_mapa)
59
60     capa_personaje = ScrollableLayer()
61     personaje = MiGuerrero('mi_guerrero_1.png')
62     personaje.position = (200,200)
63     personaje.do(Control())
64     capa_personaje.add(personaje)
65
66     mapa_colision = RectMapCollider()
67     mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
68     personaje.man_col = make_collision_handler(mapa_colision, mi_mapa)
69
70     mi_escena = Scene()
71     mi_escena.add(manejador_scroll, z=0)
72     mi_escena.add(capa_personaje, z=1)
73     director.run(mi_escena)
74
75
76 if __name__ == '__main__':
77     main()
78
```

Nada más ejecutar el programa aparecerá la siguiente ventana:



Podemos mover libremente al guerrero por la pantalla con las teclas de cursor izquierda y derecha, haciendo que salte pulsando la barra espaciadora. Las plataformas actuarán como elementos sólidos, pero no tenemos limitado el movimiento únicamente a la ventana que visualizamos. Si llevamos al guerrero a cualquiera de sus dos extremos horizontales e intentamos traspasarlo se caerá y perderemos el control sobre él. Si estamos en cualquiera de las plataformas más altas, al saltar, la parte superior del guerrero saldrá de la pantalla. Incluso tenemos la posibilidad de, estando en una de las plataformas de los laterales, saltar, salir momentáneamente de la pantalla y volver a entrar en ella moviendo adecuadamente las teclas de cursor.

Comentarios sobre el código:

- En la función main() colocaremos gran parte de los elementos.
- Definiremos la clase MiGuerrero, que controlará el movimiento del personaje. La clase Control nos servirá básicamente para actualizar en cada fotograma su posición.
- En las líneas L2-9 importamos todas las clases que vamos a necesitar en el programa.
- La clase MiGuerrero tiene una serie de atributos de clase que serán usados posteriormente. Con en_suelo almacenamos si el guerrero está en cualquiera de las plataformas, VEL_MOV será un parámetro para calcular la velocidad de desplazamiento del guerrero, VEL_SALTO para su velocidad de salto, y GRAVEDAD marcará la fuerza con la que es impulsado hacia abajo el guerrero debido a la gravedad.
- MiGuerrero tiene el método update(), que actualiza en cada fotograma la posición del guerrero en base a las teclas que pulsemos y a dónde se encuentre. Hace uso del manejador de teclado man_tec. En L22-35, en base a todos los elementos que intervienen en el movimiento del guerrero, calculamos su potencial siguiente posición, que junto a la posición actual son pasadas al manejador de colisión que hemos creado (y añadido a la instancia de MiGuerrero) en L68 teniendo como referencia la capa0 de mapa3.tmx que nos indica los objetos sólidos a tener en cuenta. La salida del manejador de colisión indicará la velocidad del guerrero, datos que almacenamos en el atributo velocidad. En L38 calculamos el atributo en_suelo y en L39 actualizamos la posición del guerrero teniendo como referencia el rectángulo[83] después, que ha podido ser modificado (o no) por el manejador de colisión.

83 Instancia de la clase Rect, del módulo cocos.rect.

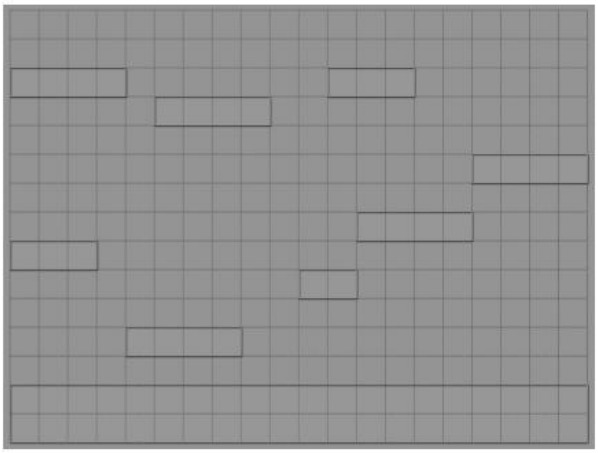
- En la función main() declaramos como global el manejador de teclado man_tec (creado en L53 mediante la clase KeyStateHandler del módulo pyglet.window.key y añadido a la ventana principal en L54) para que se pueda acceder a ella desde la clase MiGuerrero.
- En L51 he colocado la ventana del juego en las coordenadas (500,300). El lector puede modificarlas de cara a adaptarlas mejor a la resolución de pantalla que pueda tener, algo que se repetirá en las sucesivas etapas.
- En L56-58 creamos un manejador de scroll al que añadimos el mapa de baldosas contenido en capa0 de mapa3.tmx, que cargamos mediante la función load_tmx().
- En L60-64 creamos una capa con posibilidad de scroll a la que añadimos el sprite del guerrero. En L63 indicamos que se ejecute la acción Control, algo que será efectivo en cuanto se ejecute la escena principal.
- En L66 y L68 creamos el manejador de colisión (de tipo RectMapCollider y basado en el mapa que hemos cargado) y lo añadimos como atributo a la instancia de MiGuerrero. En L67 le indicamos que el comportamiento al chocar con los objetos "sólidos" sea detener su movimiento.
- En L70-73 creamos una escena con el manejador de scroll y la capa del personaje, tras lo cual la reproducimos.

En nuestra carpeta de ficheros incluyo dos pequeñas variaciones sobre plataformas_1.py, en las que cambio la forma de crear y usar el manejador de colisión[84]:

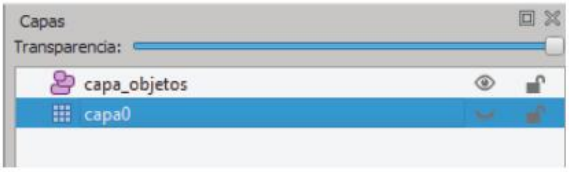
- **plataformas_1_2.py**, donde la clase MiGuerrero hereda no solo de Sprite sino también de RectMapCollider, permitiéndonos usar internamente el método collide_map(). Necesitamos para ello conocer la instancia de RectMapLayer sobre la que trabajaremos.
- **plataformas_1_3.py**, en el cual, al instanciar MiGuerrero, pasamos tanto el modelo de colisión como la capa en la que se producen las colisiones (en nuestro caso particular instancias de RectMapCollider y RectMapLayer) para posteriormente usar el método collide_map().

84 El lector interesado puede ver y analizar ambos ficheros.

Como hemos comentado, en este primer código (y sus variantes) el manejador de colisión se ha construido en base a instancias de RectMapLayer y RectMapCollider. También podremos construirlo sobre instancias de TmxObjectLayer y TmxObjectMapCollider. Para ello inicialmente crearemos, mediante Tiled Map Editor y partiendo de mapa3.tmx, un mapa llamado **mapa3_2.tmx**. A su inicial capa de patrones de nombre capa0 le añadiremos[85] otra superior de tipo capa de objetos llamada **capa_objetos** que contendrá instancias TmxObject rectangulares[86] que solapen de forma perfecta las baldosas no vacías que tenemos en capa0. Su apariencia es:



Para que aparezca únicamente la citada capa tendremos desactivada la opción de visualizar[87] la capa restante:



85 De la forma vista en Apéndice C.

86 A pesar de que Tiled Map Editor nos permite trabajar con figuras geométricas como elipses, círculos y polígonos, a efectos prácticos para trabajar con cocos2d sólo tendremos la opción de hacerlo con rectángulos y líneas.

87 Tiene forma de ojo.

En el fichero **plataformas_1_objetos.py** tenemos el código para trabajar con los elementos que hemos comentado. Presentaré los bloques principales de código que varían respecto de plataformas_1.py:

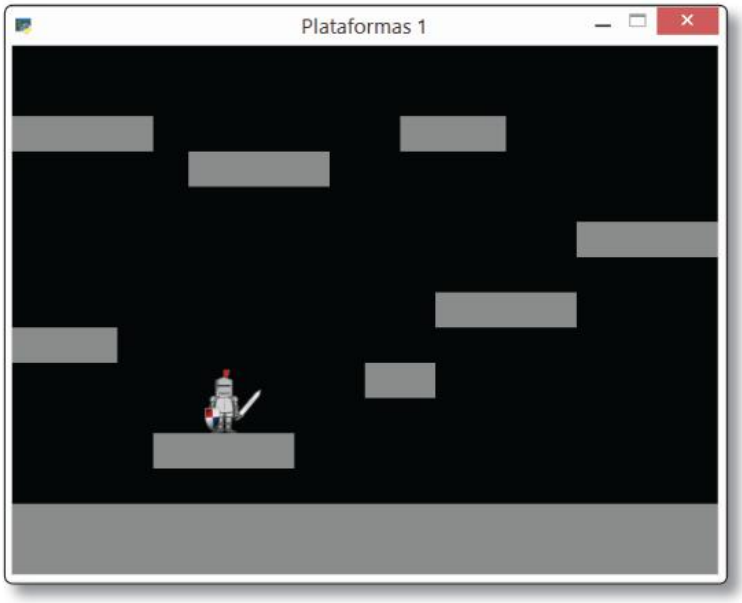
Bloque 1:

```
55
56    manejador_scroll = ScrollingManager()
57    mi_mapa1 = load_tmx('mapa3_2.tmx')['capa0']
58    mi_mapa2 = load_tmx('mapa3_2.tmx')['capa_objetos']
59    manejador_scroll.add(mi_mapa1)
60    manejador_scroll.add(mi_mapa2)
61
```

Bloque 2:

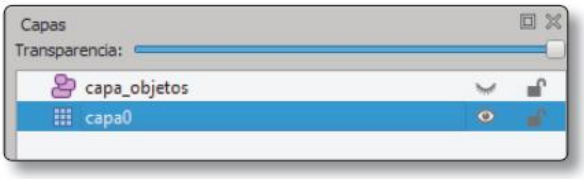
```
67
68    mapa_colision = TmxObjectMapCollider()
69    mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
70    personaje.man_col = make_collision_handler(mapa_colision, mi_mapa2)
71
```

La pantalla que aparecerá al ejecutar el código será la siguiente:



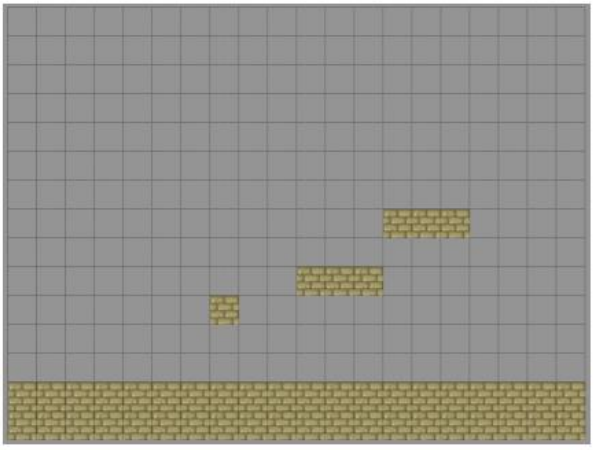
El comportamiento de los elementos será igual al que ya conocemos. Hemos desactivado la visualización de la capa0 de mapa3_2.tmx para hacer más patente que

ahora trabajamos con instancias de TmxObjectLayer y TmxObjectMapCollider. Si ahora intercambiásemos las visualizaciones de ambas capas[88]:



Tendremos entonces la misma apariencia que en plataformas_1.py a pesar de seguir trabajando con los mismos elementos (TmxObjectLayer, TmxObjectMapCollider y manejador de colisión generado en base a ellos).

En la siguiente etapa[89] crearemos una segunda pantalla, a la que pasaremos al traspasar el límite derecho de la pantalla ya creada. Por lo tanto antes de nada generaremos un nuevo mapa (de nombre **mapa4.tmx**) con una sola capa (**capa0**) del mismo tamaño y basado en el mismo conjunto de patrones que el anterior. Tendrá la apariencia mostrada a continuación:



88 No nos olvidemos en tal caso de guardar previamente las modificaciones de mapa3_2.tmx.

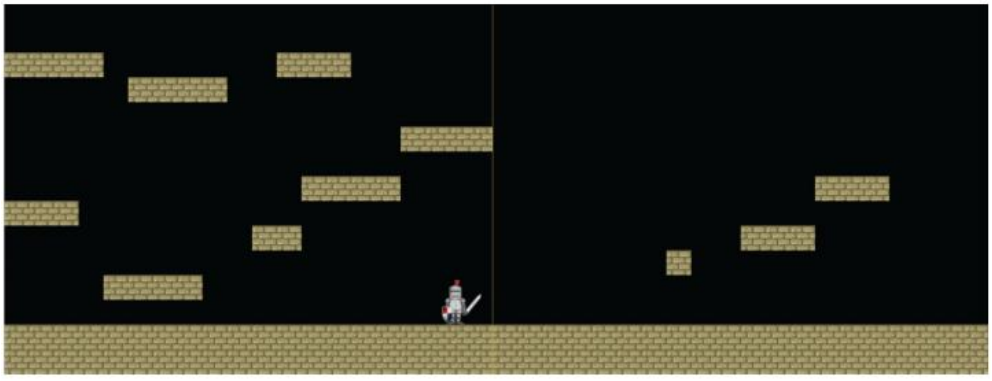
89 Seguiremos sobre la base de plataformas_1.py

El código estará en **plataformas_2.py**:

```
1
2 from pyglet.window import key
3 from cocos.sprite import Sprite
4 from cocos.scene import Scene
5 from cocos.layer import ScrollableLayer, ScrollingManager
6 from cocos.tiles import load_tmx
7 from cocos.mapcolliders import RectMapCollider, make_collision_handler
8 from cocos.actions import Action
9 from cocos.director import director
10
11 class MiGuerrero(Sprite):
12     en_suelo = True
13     VEL_MOV = 200
14     VEL_SALTO = 400
15     GRAVEDAD = -800
16
17     def __init__(self, image):
18         super().__init__(image)
19         self.velocidad = (0,0)
20
21     def update(self, dt):
22         vx, vy = self.velocidad
23         vx = (man_tec[key.RIGHT] - man_tec[key.LEFT]) * self.VEL_MOV
24         vy += self.GRAVEDAD * dt
25         if self.en_suelo and man_tec[key.SPACE]:
26             vy = self.VEL_SALTO
27
28         dx = vx * dt
29         dy = vy * dt
30
31         antes = self.get_rect()
32         despues = antes.copy()
33         despues.x += dx
34         despues.y += dy
35
36         self.velocidad = self.man_col(antes, despues, vx, vy)
37         self.en_suelo = (despues.y == antes.y)
38         self.position = despues.center
39
40         if personaje.pista == 1 and self.x > 640:
41             pista_2()
42         if personaje.pista == 2 and self.x < 0:
43             pista_1()
44
45
46 class Control(Action):
47     def step(self, dt):
48         self.target.update(dt)
49
50
51 def pista_1():
52     global personaje, mapa_colision
53     mi_mapa = load_tmx('mapa3.tmx')['capa0']
54     if 'personaje' not in globals():
55         personaje = MiGuerrero('mi_guerrero_1.png')
56         personaje.position = (200,200)
57         personaje.do(Control())
58     else:
59         personaje.x = 640
60     personaje.pista = 1
61
62     manejador_scroll = ScrollingManager()
63     manejador_scroll.add(mi_mapa)
64
65     capa_personaje = ScrollableLayer()
```

```
    capa_personaje.add(personaje)

    if 'mapa_colision' not in globals():
        mapa_colision = RectMapCollider()
        mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
    personaje.man_col = make_collision_handler(mapa_colision, mi_mapa)

    mi_escena = Scene()
    mi_escena.add(manejador_scroll, z=0)
    mi_escena.add(capa_personaje, z=1)
    director.scene_stack.clear()
    director.run(mi_escena)


def pista_2():
    mi_mapa2 = load_tmx('mapa4.tmx')['capa0']
    personaje.x = 0
    personaje.pista = 2

    manejador_scroll2 = ScrollingManager()
    manejador_scroll2.add(mi_mapa2)

    capa_personaje2 = ScrollableLayer()
    capa_personaje2.add(personaje)

    personaje.man_col = make_collision_handler(mapa_colision, mi_mapa2)

    mi_escena2 = Scene()
    mi_escena2.add(manejador_scroll2, z=0)
    mi_escena2.add(capa_personaje2, z=1)
    director.run(mi_escena2)


if __name__ == '__main__':
    ventana = director.init(width=640, height=480, caption='Plataformas 2')
    ventana.set_location(500,300)
    man_tec = key.KeyStateHandler()
    director.window.push_handlers(man_tec)
    pista_1()
```

Ahora el escenario completo constará de las siguientes dos pantallas[90]:



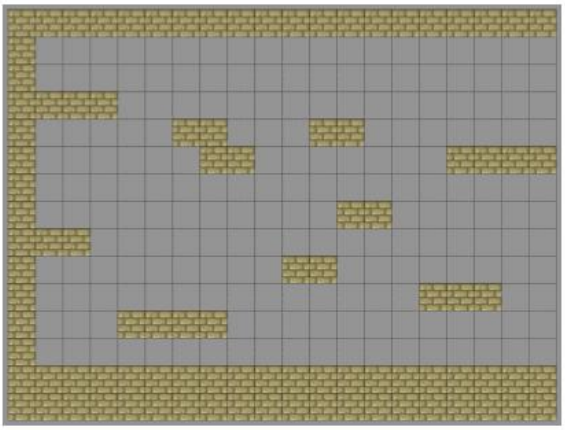
90 He colocado una línea vertical para visualizar mejor los límites de ambas.

Comentarios sobre el código:

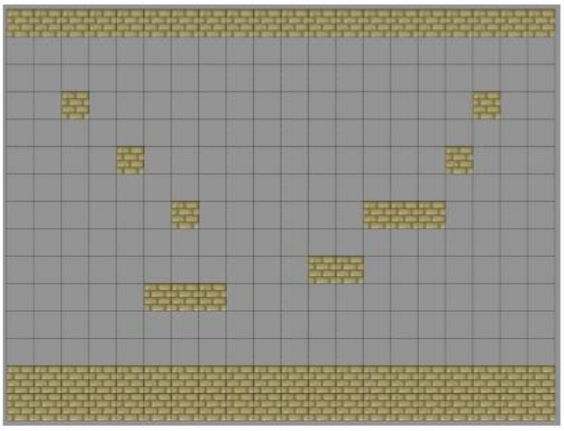
- Tendremos dos funciones (pista_1() y pista_2()), una para cada pantalla. En ellas se realizará lo que en el ejemplo anterior hacíamos en main().
- La clase Control actualiza solamente al guerrero ya que es el único personaje que tenemos en la escena. Más adelante lo ampliaremos a todos los que pueda haber en ella.
- A la instancia de MiGuerrero se le añadirá en L60 un atributo llamado pista (inicialmente con valor 1) para indicarnos en qué pantalla nos encontramos.
- Dentro de la clase MiGuerrero, en su método update() que se ejecuta en cada fotograma, es donde comprobamos si traspasamos el límite derecho de la pista 1 o el izquierdo de la pista 2, en cuyo caso cambiaríamos de pista.
- En L52 declaramos globales una serie de variables para que sean accesibles para la función pista_2().
- En L54-59 comprobamos (haciendo uso de la función globals()) si la variable personaje está ya creada. Si no es así (significa que es la primera ejecución de pista_1()) creamos el sprite del guerrero y lo configuramos con las coordenadas (200,200). De lo contrario no es la primera vez que ejecutamos pista_1(), y volvemos a la pista 1 desde la pista 2, por lo que damos a la coordenada x del guerrero el valor 640 (límite derecho de la pantalla).
- Dentro de la función pista_2() ponemos la coordenada x del guerrero a 0 y damos valor 2 al atributo pista. En L81 cargamos el mapa que almacenamos en mapa4.tmx y en L91 creamos un manejador de colisión basado en él al que apuntará ahora el atributo man_col del guerrero. Se tendrán por tanto en cuenta los objetos sólidos de la segunda pantalla de cara al movimiento del guerrero.

Seguimos sin tener límites laterales para el movimiento del guerrero. En el siguiente programa nos planteamos crear una tercera pista (de forma similar, aunque no igual, a la vista en este segundo ejemplo) y colocar en el diseño de cada una de ellas límites en los lados de cada pantalla que no sean los que permiten pasar a la siguiente (o a la anterior). Crearemos por tanto los mapas **mapa5.tmx**, **mapa6.tmx** y **mapa7.tmx**. Son los siguientes:

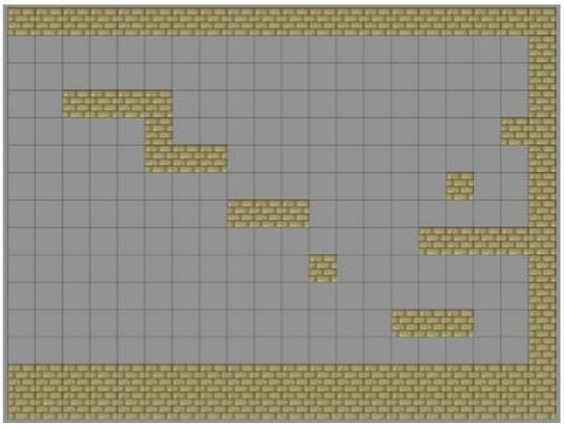
mapa5.tmx



mapa6.tmx



mapa7.tmx

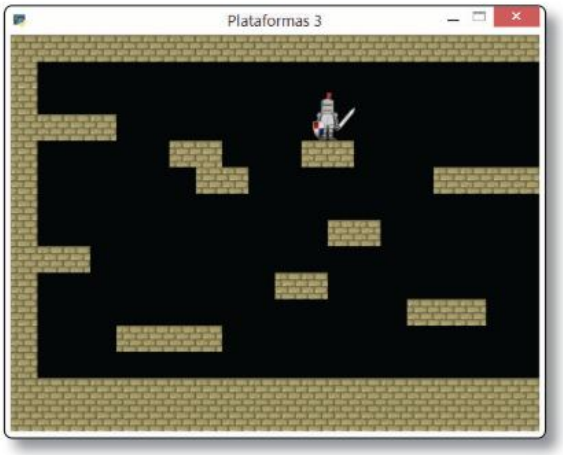


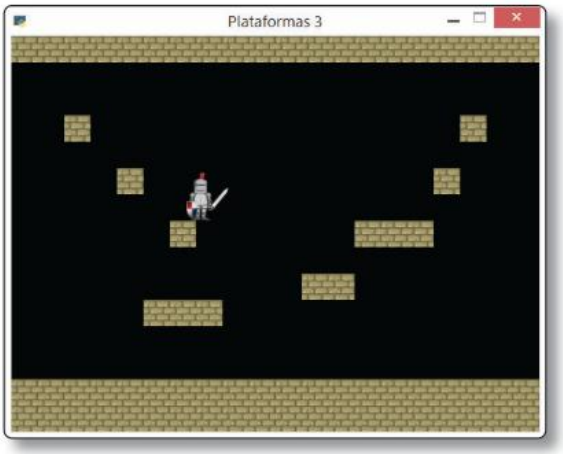
El código lo tendremos en **plataformas_3.py**:

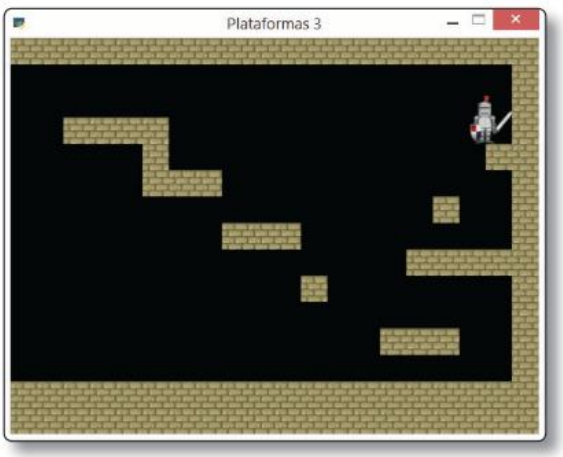
```
1
2 from pyglet.window import key
3 from cocos.sprite import Sprite
4 from cocos.scene import Scene
5 from cocos.layer import ScrollableLayer, ScrollingManager
6 from cocos.tiles import load_tmx
7 from cocos.mapcolliders import RectMapCollider, make_collision_handler
8 from cocos.actions import Action
9 from cocos.director import director
10
11 class MiGuerrero(Sprite):
12     en_suelo = True
13     VEL_MOV = 200
14     VEL_SALTO = 400
15     GRAVEDAD = -800
16
17     def __init__(self, image):
18         super().__init__(image)
19         self.velocidad = (0, 0)
20
21     def update(self, dt):
22         vx, vy = self.velocidad
23         vx = (man_tec[key.RIGHT] - man_tec[key.LEFT]) * self.VEL_MOV
24         vy += self.GRAVEDAD * dt
25         if self.en_suelo and man_tec[key.SPACE]:
26             vy = self.VEL_SALTO
27
28         dx = vx * dt
29         dy = vy * dt
30
31         antes = self.get_rect()
32         despues = antes.copy()
33         despues.x += dx
34         despues.y += dy
35
36         self.velocidad = self.man_col(antes, despues, vx, vy)
37         self.en_suelo = (despues.y == antes.y)
38         self.position = despues.center
39
40         if personaje.pista == 1 and self.x > 640:
41             personaje.x = 0
42             Escena2()
43         if personaje.pista == 2 and self.x < 0:
44             personaje.x = 640
45             Escena1()
46         if personaje.pista == 2 and self.x > 640:
47             personaje.x = 0
48             Escena3()
49         if personaje.pista == 3 and self.x < 0:
50             personaje.x = 640
51             Escena2()
52
53
54 class Control(Action):
55     def step(self, dt):
56         self.target.update(dt)
57
58
59 class Escena1(Scene):
60     def __init__(self):
61         global personaje, mapa_colision
62         super().__init__()
63         mi_mapa = load_tmx('mapa5.tmx')['capa0']
64         if 'personaje' not in globals():
```

```
            personaje = MiGuerrero('mi_guerrero_1.png')
            personaje.position = (200, 200)
            personaje.do(Control())
        personaje.pista = 1

        manejador_scroll = ScrollingManager()
        manejador_scroll.add(mi_mapa)

        capa_personaje = ScrollableLayer()
        capa_personaje.add(personaje)

        if 'mapa_colision' not in globals():
            mapa_colision = RectMapCollider()
            mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
        personaje.man_col = make_collision_handler(mapa_colision, mi_mapa)

        self.add(manejador_scroll, z=0)
        self.add(capa_personaje, z=1)
        director.run(self)


class Escena2(Scene):
    def __init__(self):
        super().__init__()
        personaje.pista = 2
        mi_mapa2 = load_tmx('mapa6.tmx')['capa0']

        manejador_scroll = ScrollingManager()
        manejador_scroll.add(mi_mapa2)

        capa_personaje = ScrollableLayer()
        capa_personaje.add(personaje)

        personaje.man_col = make_collision_handler(mapa_colision, mi_mapa2)

        self.add(manejador_scroll, z=0)
        self.add(capa_personaje, z=1)
        director.run(self)


class Escena3(Scene):
    def __init__(self):
        super().__init__()
        personaje.pista = 3
        mi_mapa3 = load_tmx('mapa7.tmx')['capa0']

        manejador_scroll = ScrollingManager()
        manejador_scroll.add(mi_mapa3)

        capa_personaje = ScrollableLayer()
        capa_personaje.add(personaje)

        personaje.man_col = make_collision_handler(mapa_colision, mi_mapa3)

        self.add(manejador_scroll, z=0)
        self.add(capa_personaje, z=1)
        director.run(self)


if __name__ == '__main__':
    ventana = director.init(width=640, height=480, caption='Plataformas 3')
    ventana.set_location(500,300)
    man_tec = key.KeyStateHandler()
    director.window.push_handlers(man_tec)
    Escena1()
```

Veamos a continuación imágenes de cada una de las pistas:







Comentarios sobre el código:

- En este caso hemos definido clases en lugar de funciones para cada una de las pistas, aunque el código que contienen es muy similar.
- En L40-51 es donde comprobamos la posición del guerrero y si debemos cambiar de pista.

Recordemos que el mapa de colisión RectMapCollider está diseñado para usarse con los mapas de baldosas (tiled maps). Los obstáculos sólidos serán las baldosas rectangulares (rectangular tiles) no vacías que estén en una capa RectMapLayer. Eso nos limita si, por ejemplo, queremos que una plataforma (o una pared) tenga solamente uno de sus lados sólido (o incluso si lo es únicamente al intentar traspasarlo en un determinado sentido). Pensemos en una plataforma como las que acabamos de ver en los ejemplos anteriores. Tal y como la tenemos diseñada no nos permitirá saltar desde debajo y aterrizar en su parte superior, algo para lo cual el lado superior de la plataforma nos debería permitir el paso si nuestro sentido es abajo-arriba pero no en el arriba-abajo.

En plataformas_1_objetos.py trabajamos con TmxObjectMapCollider y TmxObjectLayer, pero los objetos TmxObject seguían teniendo sólidas todas sus caras, por lo que el comportamiento en ese sentido era el mismo.

Nuestro objetivo es controlar el comportamiento (respecto a un actor) de cada una de las caras del objeto TmxObject. Hacerlas sólidas dependiendo de si el actor llega a ellas en un determinado sentido. Para logarlo sobrescribiremos[91], dentro del módulo cocos.mapcolliders, el método **detect_collision**() de la clase **TmxObjectMapCollider** con el objetivo de poder hacer sólida la cara interna o externa de cada uno de los lados del rectángulo, lo que nos proporcionará mucha flexibilidad de cara a diseñar y configurar obstáculos. El módulo modificado lo guardaremos (en la misma carpeta y manteniendo intacto el módulo cocos.mapcolliders original) como **mapcolliders_plus.py**[92], presentando a continuación el método añadido:

```
380     def detect_collision(self, obj, last, new):
381         g = obj.get
382         dx_correction = dy_correction = 0.0
383         if  obj.properties['solidas']=='arriba1':
384             if last.bottom >= obj.top > new.bottom:
385                 dy_correction = obj.top - new.bottom
386             return dx_correction, dy_correction
387         elif  obj.properties['solidas']=='arriba2':
388             if new.top >= obj.top > last.top:
389                 dy_correction = obj.top - new.top
```

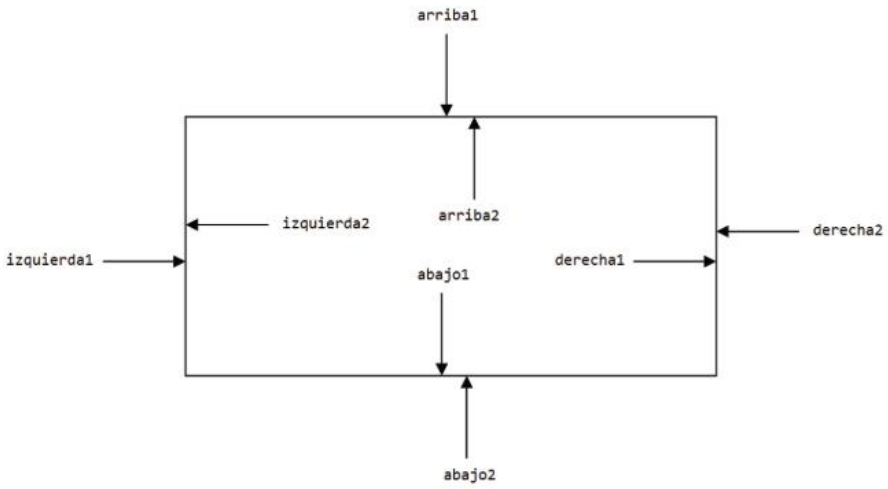
91 Recordemos que TmxObjectMapCollider deriva de RectMapCollider (y por tanto hereda su método detect_collision()), de ahí que el comportamiento de solidez de sus elementos sea por defecto el mismo en ambos.

92 Tendremos una copia del fichero en el material descargable del libro.

```
            return dx_correction, dy_correction
        elif  obj.properties['solidas']=='abajo1':
            if last.bottom >= obj.bottom > new.bottom:
                dy_correction = obj.bottom - new.bottom
            return dx_correction, dy_correction
        elif  obj.properties['solidas']=='abajo2':
            if new.top >= obj.bottom > last.top:
                dy_correction = obj.bottom - new.top
            return dx_correction, dy_correction
        elif  obj.properties['solidas']=='izquierda1':
            if last.right <= obj.left < new.right:
                dx_correction = obj.left - new.right
            return dx_correction, dy_correction
        elif  obj.properties['solidas']=='izquierda2':
            if new.left < obj.left <= last.left:
                dx_correction = obj.left - new.left
            return dx_correction, dy_correction
        elif  obj.properties['solidas']== 'derecha1':
            if new.right > obj.right >= last.right:
                dx_correction = obj.right - new.right
            return dx_correction, dy_correction
        elif  obj.properties['solidas']== 'derecha2':
            if last.left >= obj.right > new.left:
                dx_correction = obj.right - new.left
            return dx_correction, dy_correction
        elif  obj.properties['solidas']== 'izquierda1+derecha1':
            if last.right <= obj.left < new.right:
                dx_correction = obj.left - new.right
            if new.right > obj.right >= last.right:
                dx_correction = obj.right - new.right
            return dx_correction, dy_correction
        elif  obj.properties['solidas']== 'izquierda1+derecha2':
            if last.right <= obj.left < new.right:
                dx_correction = obj.left - new.right
            if last.left >= obj.right >= new.left:
                dx_correction = obj.right - new.left
            return dx_correction, dy_correction
        elif  obj.properties['solidas']== 'izquierda2+derecha1':
            if new.left < obj.left <= last.left:
                dx_correction = obj.left - new.left
            if new.right > obj.right >= last.right:
                dx_correction = obj.right - new.right
            return dx_correction, dy_correction
        elif  obj.properties['solidas']== 'izquierda2+derecha2':
            if new.left < obj.left <= last.left:
                dx_correction = obj.left - new.left
            if last.left >= obj.right > new.left:
                dx_correction = obj.right - new.left
            return dx_correction, dy_correction
        elif  obj.properties['solidas']== 'arriba1+abajo1':
            if last.bottom >= obj.top > new.bottom:
                dy_correction = obj.top - new.bottom
            if last.bottom >= obj.bottom > new.bottom:
                dy_correction = obj.bottom - new.bottom
            return dx_correction, dy_correction
        elif  obj.properties['solidas']== 'arriba1+abajo2':
            if last.bottom >= obj.top > new.bottom:
                dy_correction = obj.top - new.bottom
            if new.top >= obj.bottom > last.top:
                dy_correction = obj.bottom - new.top
            return dx_correction, dy_correction
        elif  obj.properties['solidas']== 'arriba2+abajo1':
            if new.top >= obj.top > last.top:
                dy_correction = obj.top - new.top
            if last.bottom >= obj.bottom > new.bottom:
                dy_correction = obj.bottom - new.bottom
```

```
            return dx_correction, dy_correction
        elif  obj.properties['solidas']== 'arriba2+abajo2':
            if new.top >= obj.top > last.top:
                dy_correction = obj.top - new.top
            if new.top >= obj.bottom > last.top:
                dy_correction = obj.bottom - new.top
            return dx_correction, dy_correction
        else:
            if last.bottom >= obj.top > new.bottom:
                dy_correction = obj.top - new.bottom
            elif last.top <= obj.bottom < new.top:
                dy_correction = obj.bottom - new.top
            if last.right <= obj.left < new.right:
                dx_correction = obj.left - new.right
            elif last.left >= obj.right > new.left:
                dx_correction = obj.right - new.left
            return dx_correction, dy_correction
```

En el código obj representa al objeto TmxObject. En una capa TmxObjectLayer puede haber muchos de ellos, y es posible que queramos que cada uno se comporte de forma determinada en cuanto a la solidez de sus lados se refiere. Es por ello por lo que debemos añadir en Tiled un atributo[93] (de tipo string y al que he llamado **solidas**) a cada uno de los objetos TmxObject para indicarlo. En el código accederemos a él mediante el atributo properties, que es un diccionario. A continuación se presentan los 8 valores principales del atributo solidas[94] del objeto TmxObject, donde la flecha asociada indica el sentido en el que la cara del objeto será sólida y por tanto bloqueará el paso del personaje:



93 Ver cómo hacerlo en el Apéndice C.

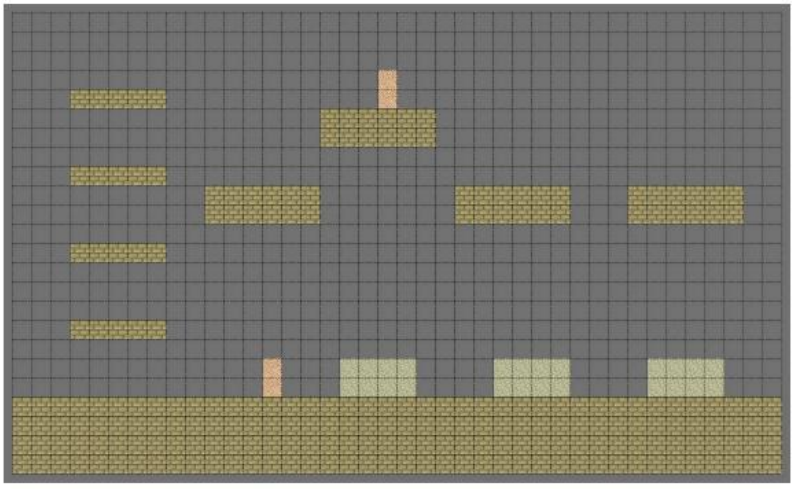
94 Tengamos en cuenta que son cadenas.

Además de ellos tenemos en el código añadido los siguientes, combinaciones[95] que bloquearán el paso en los dos lados y direcciones indicadas:

- izquierda1+derecha1
- izquierda1+derecha2
- izquierda2+derecha1
- izquierda2+derecha2
- arriba1+abajo1
- arriba1+abajo2
- arriba2+abajo1
- arriba2+abajo2

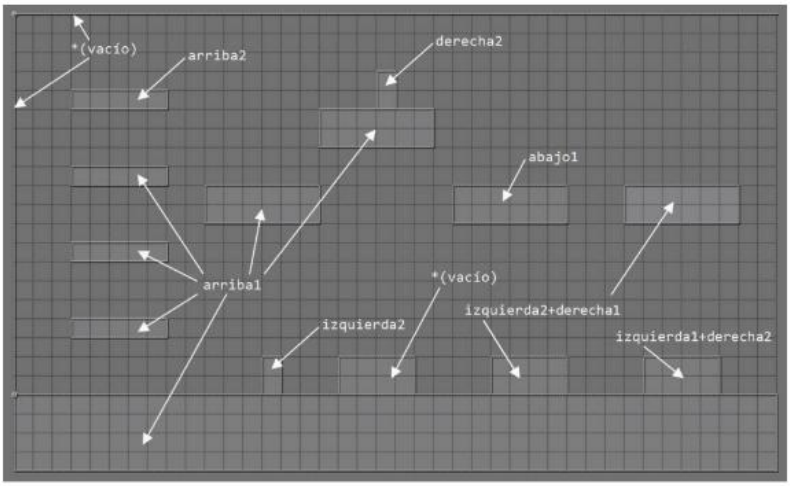
Una vez que hemos dado esta cualidad añadida a los objetos TmxObject tendremos la posibilidad de crear plataformas con distintas configuraciones. Para ello previamente debemos configurar los objetos con el adecuado valor del atributo sólidas. Crearemos con Tiled **mapa8.tmx**, que contendrá dos capas:

capa0 (capa de patrones)



95 El lector podrá incluir, añadiendo código de forma similar a la realizada, las combinaciones adicionales (de los 8 valores fundamentales) que considere oportuno.

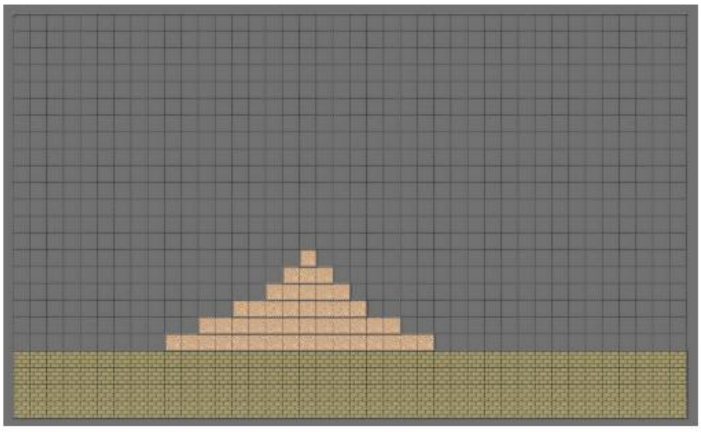
objetos (capa de objetos)



En la última imagen se indica el valor del atributo solidas de cada uno de los objetos[96], que son todos de tipo rectángulo salvo las dos líneas del lado izquierdo y superior.

Adicionalmente crearemos **mapa9.tmx** y **mapa10.tmx**, que constan de dos capas del mismo tipo y del mismo nombre que mapa8.tmx. Sus imágenes visualizando ambas capas son las siguientes[97]:

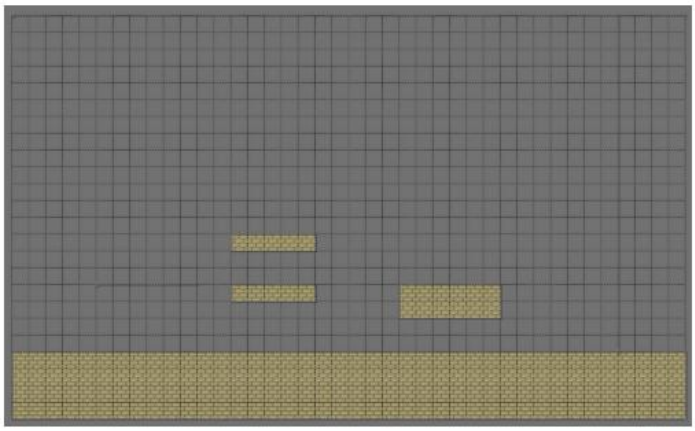
mapa9.tmx



96 Con *(vacío) indicamos que el atributo solidas tiene como valor la cadena vacía.

97 Mantendremos la visualización de ambas capas de cara a la ejecución posterior del código.

mapa10.tmx



En ambos mapas los objetos de su capa objetos tienen el atributo solidas con valor cadena vacía, salvo las 4 plataformas de mapa10.tmx que tienen valor 'arriba1'.

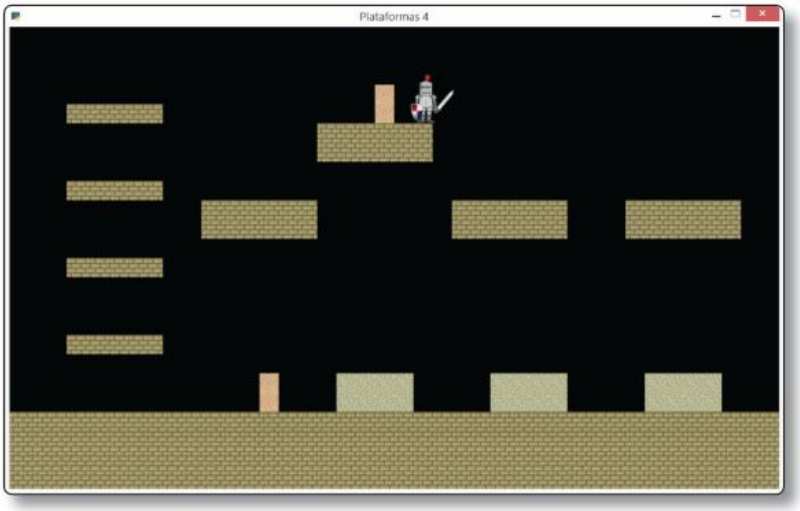
Teniendo ya los 3 mapas el código final que hace uso de ellos es **plataformas_4.py**, cuyo análisis queda como ejercicio para el lector:

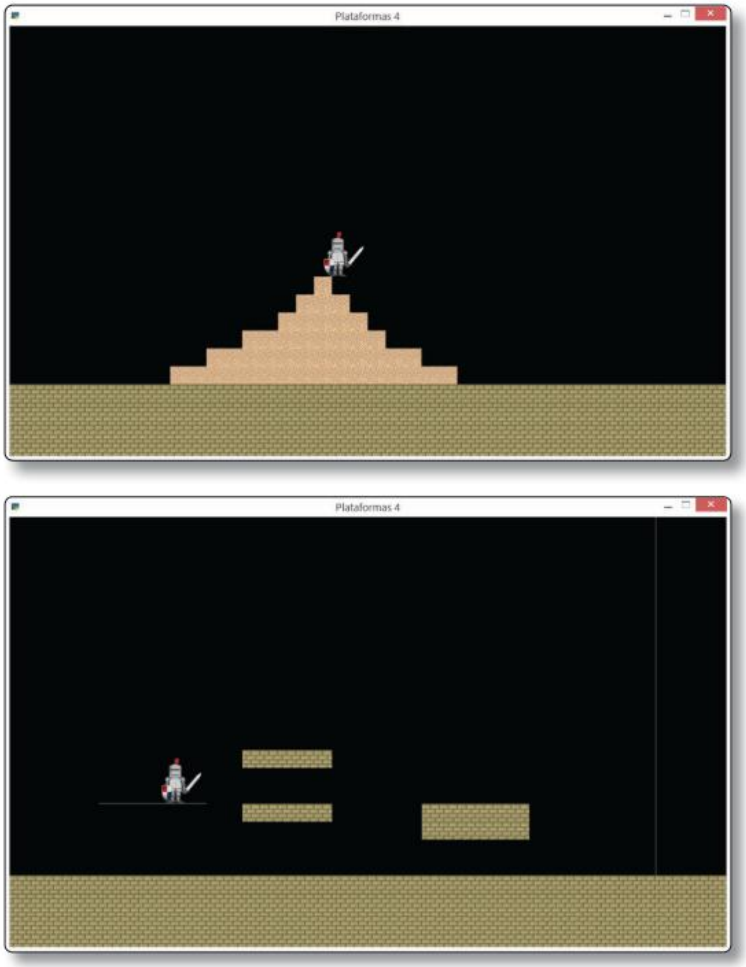
```
1
2  from pyglet.window import key
3  from cocos.sprite import Sprite
4  from cocos.scene import Scene
5  from cocos.layer import ScrollableLayer, ScrollingManager
6  from cocos.tiles import load_tmx
7  from cocos.mapcolliders_plus import TmxObjectMapCollider, make_collision_handler
8  from cocos.actions import Action
9  from cocos.director import director
10
11 class MiGuerrero(Sprite):
12     en_suelo = True
13     VEL_MOV = 200
14     VEL_SALTO = 500
15     GRAVEDAD = -800
16
17     def __init__(self, image):
18         super().__init__(image)
19         self.velocidad = (0,0)
20
21     def update(self, dt):
22         vx, vy = self.velocidad
23         vx = (man_tec[key.RIGHT] - man_tec[key.LEFT]) * self.VEL_MOV
24         vy += self.GRAVEDAD * dt
25         if self.en_suelo and man_tec[key.SPACE]:
26             vy = self.VEL_SALTO
27
28         dx = vx * dt
29         dy = vy * dt
30
31         antes = self.get_rect()
32         despues = antes.copy()
```

```
        despues.x += dx
        despues.y += dy

        self.velocidad = self.man_col(antes, despues, vx, vy)
        self.en_suelo = (despues.y == antes.y)
        self.position = despues.center

        if personaje.pista == 1 and self.x > 1280:
            personaje.x = 0
            Escena2()
        if personaje.pista == 2 and self.x < 0:
            personaje.x = 1280
            Escena1()
        if personaje.pista == 2 and self.x > 1280:
            personaje.x = 0
            Escena3()
        if personaje.pista == 3 and self.x < 0:
            personaje.x = 1280
            Escena2()


class Control(Action):
    def step(self, dt):
        self.target.update(dt)


class Escena1(Scene):
    def __init__(self):
        global personaje, mapa_colision
        super().__init__()
        mi_mapa1 = load_tmx('mapa8.tmx')['objetos']
        mi_mapa1_1 = load_tmx('mapa8.tmx')['capa0']
        if 'personaje' not in globals():
            personaje = MiGuerrero('mi_guerrero_2.png')
            personaje.position = (200,300)
            personaje.do(Control())
        personaje.pista = 1

        manejador_scroll = ScrollingManager()
        manejador_scroll.add(mi_mapa1)
        manejador_scroll.add(mi_mapa1_1)

        capa_personaje = ScrollableLayer()
        capa_personaje.add(personaje)

        if 'mapa_colision' not in globals():
            mapa_colision = TmxObjectMapCollider()
            mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
        personaje.man_col = make_collision_handler(mapa_colision, mi_mapa1)

        self.add(manejador_scroll, z=0)
        self.add(capa_personaje, z=1)
        director.run(self)

    def __init__(self):
        super().__init__()
        personaje.pista = 2
        mi_mapa2 = load_tmx('mapa9.tmx')['objetos']
        mi_mapa2_1 = load_tmx('mapa9.tmx')['capa0']

        manejador_scroll = ScrollingManager()
        manejador_scroll.add(mi_mapa2)
        manejador_scroll.add(mi_mapa2_1)

        capa_personaje = ScrollableLayer()
        capa_personaje.add(personaje)
```

```
101
102         personaje.man_col = make_collision_handler(mapa_colision, mi_mapa2)
103
104         self.add(manejador_scroll, z=0)
105         self.add(capa_personaje, z=1)
106         director.run(self)
107
108
109 class Escena3(Scene):
110     def __init__(self):
111         super().__init__()
112         personaje.pista = 3
113         mi_mapa3 = load_tmx('mapa10.tmx')['objetos']
114         mi_mapa3_1 = load_tmx('mapa10.tmx')['capa0']
115
116         manejador_scroll = ScrollingManager()
117         manejador_scroll.add(mi_mapa3)
118         manejador_scroll.add(mi_mapa3_1)
119
120         capa_personaje = ScrollableLayer()
121         capa_personaje.add(personaje)
122
123         personaje.man_col = make_collision_handler(mapa_colision, mi_mapa3)
124
125         self.add(manejador_scroll, z=0)
126         self.add(capa_personaje, z=1)
127         director.run(self)
128
129
130 if __name__ == '__main__':
131     ventana = director.init(width=1280, height=768, caption='Plataformas 4')
132     ventana.set_location(300,200)
133     man_tec = key.KeyStateHandler()
134     director.window.push_handlers(man_tec)
135     Escena1()
136
```

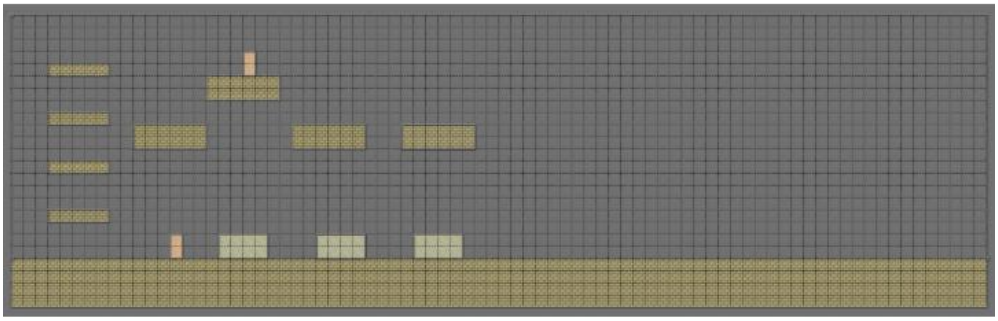
Las tres pantallas que tenemos si ejecutamos el programa son:





En la primera pantalla los obstáculos se comportarán con las características de solidez que les hemos indicado. En la tercera aparecen visibles dos líneas: la horizontal hará de plataforma y la vertical limitará el desplazamiento horizontal del guerrero. Si no quisiésemos que se viesen desactivaríamos en mapa10.tmx la opción de visualizar la capa objetos y guardaríamos los cambios antes de ejecutar de nuevo el código.

Tenemos ya un ejemplo de la base para un juego de plataformas con tres pantallas, pero no hemos hecho uso del scroll. Nuestra intención será ahora la construcción de los elementos básicos para un juego de plataformas que sí lo use. Crearemos inicialmente un escenario de tamaño mayor que una pantalla de las que ya conocemos. Exactamente será del tamaño de dos de ellas, por lo que el tamaño del mapa (de nombre **mapa11.tmx**) será de 80 x 24 patrones. Contendrá las habituales capas (capa0 y objetos) y su aspecto visualizando ambas se muestra a continuación:



Los objetos tienen los mismos valores de su atributo solidas que los indicados en mapa8.tmx.

Teniendo el mapa el código[98] será **plataformas_5.py**:

```
from pyglet.window import key
from cocos.sprite import Sprite
from cocos.scene import Scene
from cocos.layer import ScrollableLayer, ScrollingManager
from cocos.tiles import load_tmx
from cocos.mapcolliders_plus import TmxObjectMapCollider, make_collision_handler
from cocos.actions import Action
from cocos.director import director

class MiGuerrero(Sprite):
    en_suelo = True
    VEL_MOV = 200
    VEL_SALTO = 500
    GRAVEDAD = -800

    def __init__(self, image):
        super().__init__(image)
        self.velocidad = (0,0)

    def update(self, dt):
        vx, vy = self.velocidad
        vx = (man_tec[key.RIGHT] - man_tec[key.LEFT]) * self.VEL_MOV
        vy += self.GRAVEDAD * dt
        if self.en_suelo and man_tec[key.SPACE]:
            vy = self.VEL_SALTO

        dx = vx * dt
        dy = vy * dt

        antes = self.get_rect()
        despues = antes.copy()
        despues.x += dx
        despues.y += dy

        self.velocidad = self.man_col(antes, despues, vx, vy)
        self.en_suelo = (despues.y == antes.y)
        self.position = despues.center

        manejador_scroll.set_focus(despues.x, 384)
```

98 Su análisis queda como ejercicio para el lector.

```
42
43 class Control(Action):
44     def step(self, dt):
45         self.target.update(dt)
46
47
48 class Escena(Scene):
49     def __init__(self):
50         global  manejador_scroll
51         super().__init__()
52         mi_mapa1 = load_tmx('mapa11.tmx')['objetos']
53         mi_mapa1_1 = load_tmx('mapa11.tmx')['capa0']
54
55         personaje = MiGuerrero('mi_guerrero_2.png')
56         personaje.position = (200,300)
57         personaje.do(Control())
58
59         capa_personaje = ScrollableLayer()
60         capa_personaje.add(personaje)
61
62         manejador_scroll = ScrollingManager()
63         manejador_scroll.add(mi_mapa1, z=0)
64         manejador_scroll.add(mi_mapa1_1, z=1)
65         manejador_scroll.add(capa_personaje, z=2)
66
67         mapa_colision = TmxObjectMapCollider()
68         mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
69         personaje.man_col = make_collision_handler(mapa_colision, mi_mapa1)
70
71         self.add(manejador_scroll)
72         director.run(self)
73
74
75 if __name__ == '__main__':
76     ventana = director.init(width=1280, height=768, caption='Plataformas 5')
77     ventana.set_location(300,200)
78     man_tec = key.KeyStateHandler()
79     director.window.push_handlers(man_tec)
80     Escena()
81
```

Tras ejecutarlo podremos desplazarnos por el escenario de forma continua:

4

DESARROLLO DE UN JUEGO DE PLATAFORMAS

Disponemos ya de la base para hacer un videojuego completo (aunque sencillo) de plataformas con scroll que incorpore sus elementos principales. Tomando como referencia inicial el fichero plataformas_5.py visto en el capítulo anterior, dividiremos nuevamente su creación en varias etapas (en este caso 6), añadiendo elementos y/o características en cada uno de ellos. Ahora los códigos de cada etapa serán **juego_x.py**, siendo x el número de la etapa en cuestión. Una pequeña explicación de qué se aporta en cada etapa es:

- Etapa 1

  Creamos mapa del juego y añadimos fondo.

- Etapa 2

  Añadimos la posibilidad de que el guerrero lance cuchillos en ambas direcciones del eje horizontal.

- Etapa 3

  Incorporamos un dragón que se mueve verticalmente y nos lanza fuego de forma aleatoria.

- Etapa 4

  Añadimos las colisiones cuchillo-fuego y cuchillo-dragón y les asociamos una explosión (distinta en cada caso).

- Etapa 5

Colocamos elementos realizados mediante sistemas de partículas (sol, humo) y añadimos dos nuevos enemigos (arañas) en una plataforma.

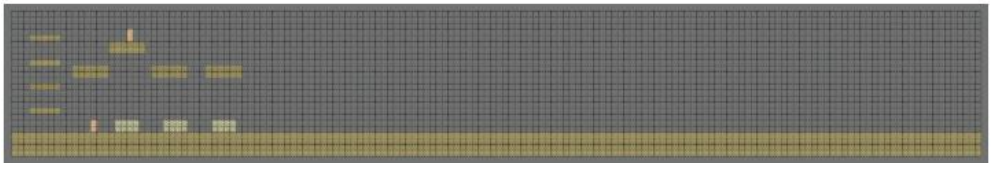
- Etapa 6

  Completamos elementos del juego:

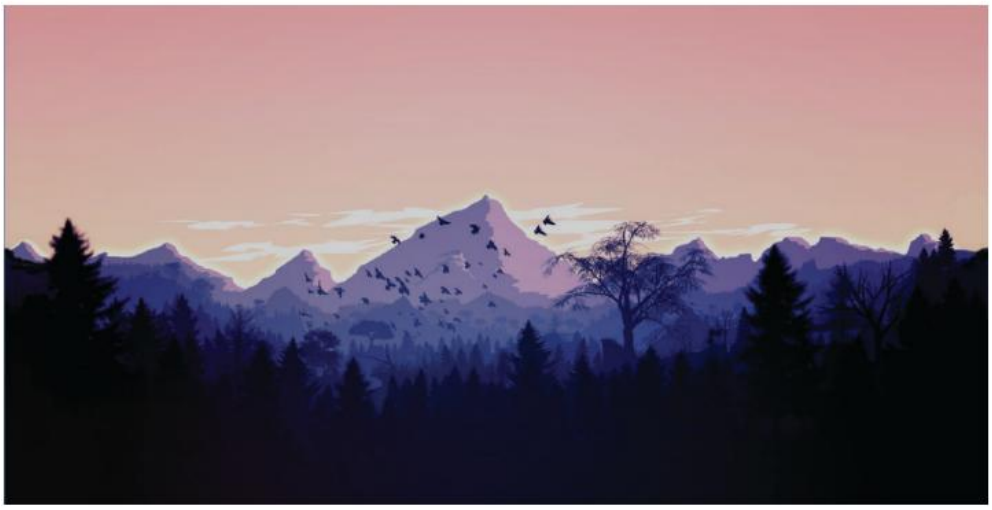
  - Tendremos en cuenta las colisiones cuchillo-araña, guerrero-araña, guerrero-dragón y guerrero-fuego, con sus correspondientes animaciones y/o sonidos asociados.
  - Añadimos música de fondo, y sonido para las explosiones, el lanzamiento de cuchillos y el de fuego, y si alcanzan a nuestro guerrero.
  - Tendremos la posibilidad de completar la misión si llegamos al límite derecho del escenario.
  - Creamos escenas de inicio, ganador y perdedor.
  - Visualizaremos un HUD que nos indique el número de vidas y los puntos.

Etapa 1

En **juego_1.py**, que usa gran parte del código creado en plataformas_5.py, trabajaremos sobre un mapa más amplio (4 veces mayor que el usado en cada una de las pantallas de plataformas_4.py y el doble del que tenemos en plataformas_5.py, para un tamaño de 160 x 24 patrones) de nombre **mapa_plataformas.tmx**. Contiene las dos capas habituales (capa0 y objetos) con la configuración ya conocida. La imagen del mapa es la siguiente:



Queremos disponer de un fondo representando un bosque y montañas. No es nada práctico (por tamaño del fichero y por la dificultad de crear o encontrar imágenes grandes) que ese fondo sea una imagen del tamaño de todo el mapa. En nuestro caso lo que haremos es colocar una repetida 4 veces a lo largo del escenario. La imagen es **mi_fondo.png**, tiene un tamaño de 1294 x 659 píxeles y el siguiente aspecto:



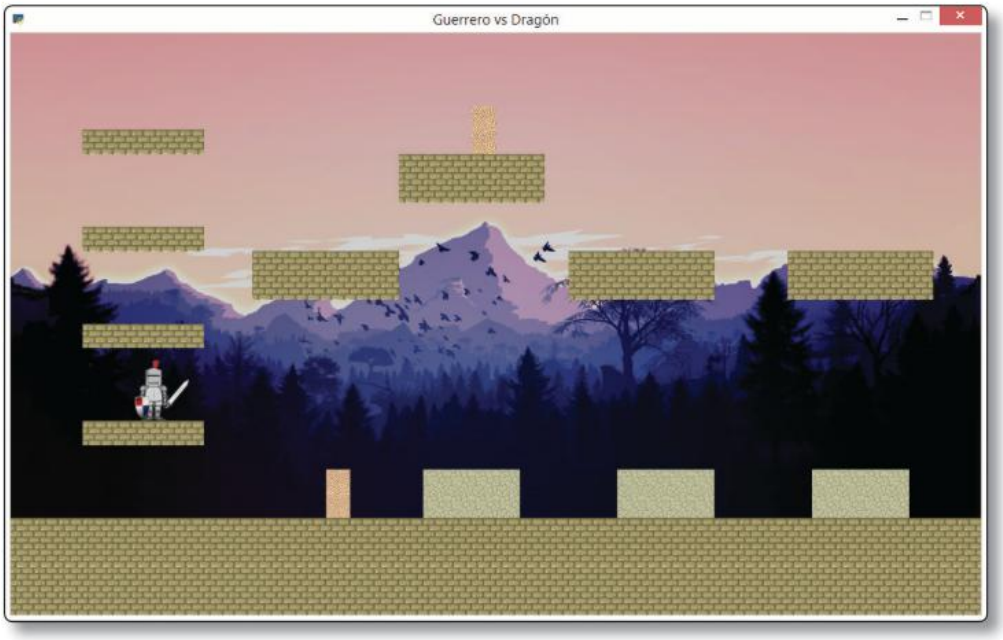
Para lograr que el fondo no presente discontinuidades la parte derecha de la imagen debe encajar con la izquierda. Para ello suele ser necesario programas de retoque fotográfico. En nuestro caso se ha modificado usando GIMP[99], eliminando algún elemento, dándole un tono más oscuro y cambiando ligeramente el tamaño original.

El código difiere en pocas líneas de plataformas_5.py, por lo que solamente mostraré a continuación el bloque en el que se añaden elementos:

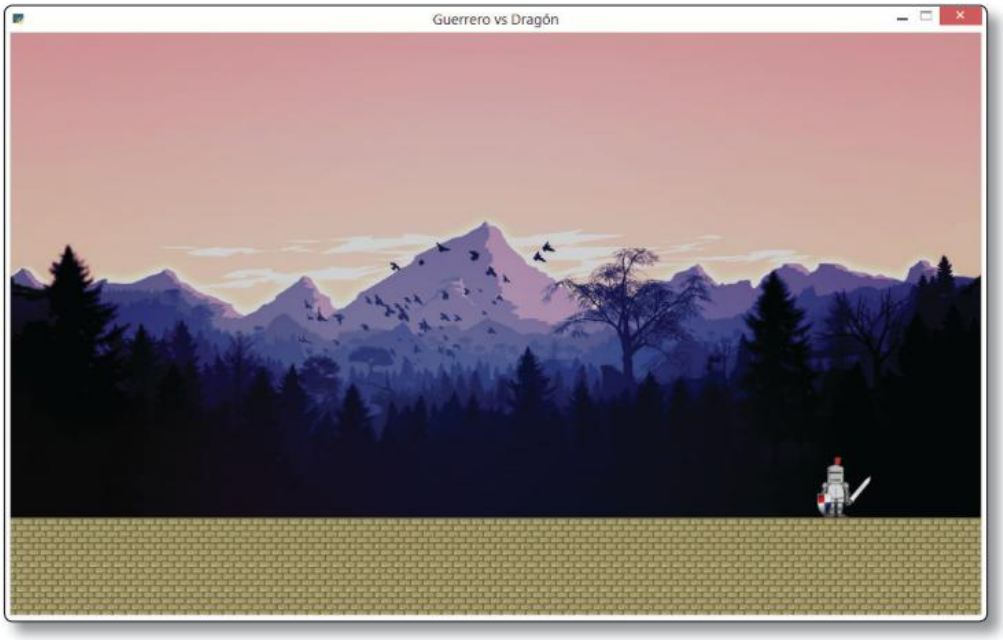
```
59          capa_fondo = ScrollableLayer()
60          for i in range(4):
61              a = Sprite('mi_fondo.png')
62              a.position = (640 + 1280*i, 440)
63              capa_fondo.add(a)
64
65          capa_personaje = ScrollableLayer()
66          capa_personaje.add(personaje)
67
68          manejador_scroll = ScrollingManager()
69          manejador_scroll.add(mi_mapa1, z=0)
70          manejador_scroll.add(capa_fondo, z=1)
71          manejador_scroll.add(mi_mapa1_1, z=2)
72          manejador_scroll.add(capa_personaje, z=3)
```

99 GNU Image Manipulation Program, editor de imagen multiplataforma. Forma parte del proyecto GNU y está disponible de forma libre bajo licencias GPL o LGPL.

La imagen inicial del juego al ejecutarlo es como se muestra a continuación:



Tendremos la posibilidad de desplazarnos por todo el escenario hasta llegar a su extremo derecho:



Etapa 2

El fichero **juego_2.py**, en el que añadimos la posibilidad de lanzar cuchillos horizontalmente en ambas direcciones mediante la pulsación de la tecla Enter, contiene el siguiente código:

```
1
2 from pyglet.window import key
3 from cocos.sprite import Sprite
4 from cocos.scene import Scene
5 from cocos.layer import ScrollableLayer, ScrollingManager
6 from cocos.tiles import load_tmx
7 from cocos.mapcolliders_plus import TmxObjectMapCollider, make_collision_handler
8 from cocos.actions import Action, Delay, CallFunc
9 from cocos.director import director
10 from cocos.euclid import Vector2
11
12 class MiGuerrero(Sprite):
13     en_suelo = True
14     VEL_MOV = 200
15     VEL_SALTO = 500
16     GRAVEDAD = -800
17
18     def __init__(self, image):
19         super().__init__(image)
20         self.velocidad = (0, 0)
21         self.direccion = 'derecha'
22         self.hay_disparo = False
23
24     def update(self, dt):
25         vx, vy = self.velocidad
26         vx = (man_tec[key.RIGHT] - man_tec[key.LEFT]) * self.VEL_MOV
27         vy += self.GRAVEDAD * dt
28         if self.en_suelo and man_tec[key.SPACE]:
29             vy = self.VEL_SALTO
30         if man_tec[key.LEFT]:
31             self.direccion = 'izquierda'
32         if man_tec[key.RIGHT]:
33             self.direccion = 'derecha'
34
35         if man_tec[key.ENTER] and not self.hay_disparo:
36             self.hay_disparo = True
37             if self.direccion == 'derecha':
38                 self.disparo = MiCuchillo('mi_cuchillo_1.png', 'd',
39                                           self.position[0], self.position[1])
40                 self.parent.add(self.disparo)
41             elif self.direccion == 'izquierda':
42                 self.disparo = MiCuchillo('mi_cuchillo_2.png', 'i',
43                                           self.position[0], self.position[1])
44                 self.parent.add(self.disparo)
45             self.do(Delay(0.2)+ CallFunc(self.nuevo_disparo))
46
47         dx = vx * dt
48         dy = vy * dt
49
50         antes = self.get_rect()
51         despues = antes.copy()
52         despues.x += dx
53         despues.y += dy
54
55         self.velocidad = self.man_col(antes, despues, vx, vy)
56         self.en_suelo = (despues.y == antes.y)
57         self.position = despues.center
58
```

```
        manejador_scroll.set_focus(despues.x, 384)

    def nuevo_disparo(self):
        self.hay_disparo = False


class MiCuchillo(Sprite):
    def __init__(self, image, dir, x, y):
        super().__init__(image)
        self.position = (x,y)
        self.direccion = dir

    def update(self, dt):
        if self.direccion == 'd':
            self.position += Vector2(20,0)
        elif self.direccion == 'i':
            self.position -= Vector2(20,0)


class Control(Action):
    def step(self, dt):
        for objeto in self.target.parent.children:
            objeto[1].update(dt)


class Escena(Scene):
    def __init__(self):
        global manejador_scroll
        super().__init__()
        mi_mapa1 = load_tmx('mapa_plataformas.tmx')['objetos']
        mi_mapa1_1 = load_tmx('mapa_plataformas.tmx')['capa0']

        personaje = MiGuerrero('mi_guerrero_2.png')
        personaje.position = (200,300)
        personaje.do(Control())

        capa_fondo = ScrollableLayer()
        for i in range(4):
            a =Sprite('mi_fondo.png')
            a.position = (640 + 1280*i, 440)
            capa_fondo.add(a)

        capa_personaje = ScrollableLayer()
        capa_personaje.add(personaje)

        manejador_scroll = ScrollingManager()
        manejador_scroll.add(mi_mapa1, z=0)
        manejador_scroll.add(capa_fondo, z=1)
        manejador_scroll.add(mi_mapa1_1, z=2)
        manejador_scroll.add(capa_personaje, z=3)

        mapa_colision = TmxObjectMapCollider()
        mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
        personaje.man_col = make_collision_handler(mapa_colision, mi_mapa1)

        self.add(manejador_scroll)
        director.run(self)


if __name__ == '__main__':
    ventana = director.init(width=1280, height=768, caption='Guerrero vs Dragón')
    ventana.set_location(300,200)
    man_tec = key.KeyStateHandler()
    director.window.push_handlers(man_tec)
    Escena()
```

Una instantánea de un lanzamiento de cuchillo es la siguiente:

Comentarios sobre el código:

- Definimos una nueva clase llamada MiCuchillo, donde en cada fotograma desplazamos 20 píxeles el sprite con la imagen del cuchillo (**mi_cuchillo_1.png** o **mi_cuchillo_2.png**) en el sentido indicado por el parámetro dir. Hacemos uso para ello de la clase Vector2 del módulo cocos.euclid.

- Añadiremos a la clase MiGuerrero el atributo direccion, una cadena que nos indica en realidad el sentido hacia donde nos vamos desplazando horizontalmente y que marcará también el sentido de lanzamiento del cuchillo. A medida que vamos pulsando las teclas de desplazamiento horizontales cambiaremos su valor.

- Uno de los problemas que podemos tener al lanzar los cuchillos es que, como el método update() de MiGuerrero se ejecuta muchas veces en un corto espacio de tiempo, al pulsar la tecla Enter se lanzarán muchos cuchillos de golpe. La forma de solucionarlo que he usado ha sido crear un atributo de nombre hay_disparo que indica si ya hay un cuchillo lanzado, algo que evita lanzar otro. En L45 configuro un retraso de 0.2 segundos hasta que se pueda lanzar otro cuchillo, ayudado en el nuevo método creado nuevo_disparo(), que simplemente da valor False al atributo hay_disparo. En el bloque L35-45 está la mayor parte del código dedicado al tratamiento del lanzamiento de cuchillos.

- Ahora ya tenemos (al menos potencialmente) dos objetos que actualizar en cada fotograma (el guerrero y el cuchillo), por lo que la clase Control actualizará mediante el método update() todos los elementos de la capa del personaje.

Etapa 3

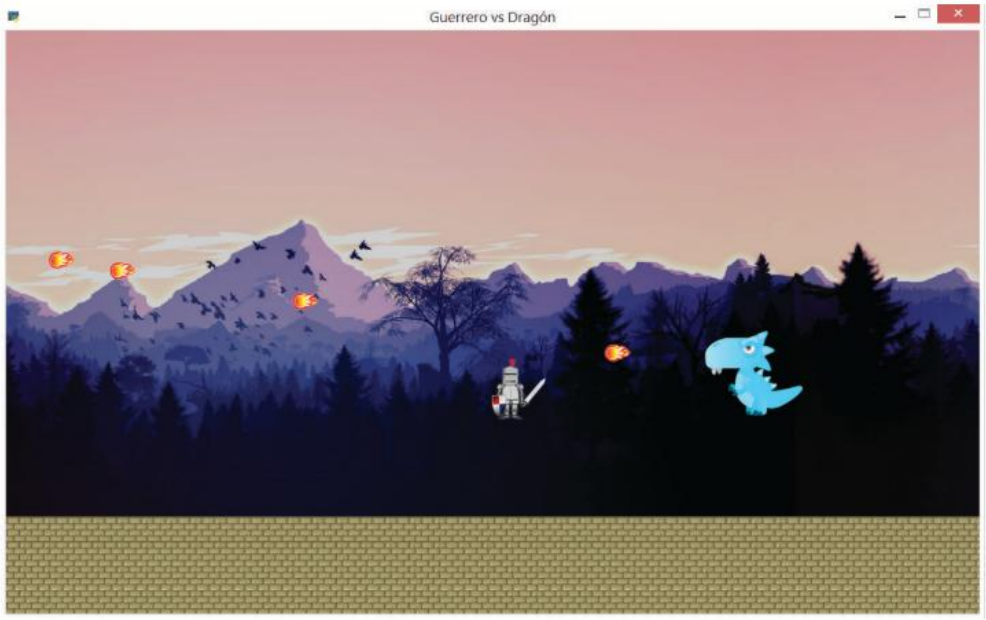
Como ejemplo de enemigo crearemos inicialmente un dragón de movimiento vertical que nos lanza fuego en instantes aleatorios. El código de **juego_3.py** es:

```
1
2  from pyglet.window import key
3  from cocos.sprite import Sprite
4  from cocos.scene import Scene
5  from cocos.layer import ScrollableLayer, ScrollingManager
6  from cocos.tiles import load_tmx
7  from cocos.mapcolliders_plus import TmxObjectMapCollider, make_collision_handler
8  from cocos.collision_model import CircleShape
9  from cocos.actions import Action, Delay, CallFunc, MoveBy, Repeat
10 from cocos.director import director
11 from cocos.euclid import Vector2
12 from random import randint
13
14 class MiGuerrero(Sprite):
15     en_suelo = True
16     VEL_MOV = 200
17     VEL_SALTO = 500
18     GRAVEDAD = -800
19
20     def __init__(self, image):
21         super().__init__(image)
22         self.velocidad = (0,0)
23         self.direccion = 'derecha'
24         self.hay_disparo = False
25
26     def update(self, dt):
27         vx, vy = self.velocidad
28         vx = (man_tec[key.RIGHT] - man_tec[key.LEFT]) * self.VEL_MOV
29         vy += self.GRAVEDAD * dt
30         if self.en_suelo and man_tec[key.SPACE]:
31             vy = self.VEL_SALTO
32         if man_tec[key.LEFT]:
33             self.direccion = 'izquierda'
34         if man_tec[key.RIGHT]:
35             self.direccion = 'derecha'
36         if man_tec[key.ENTER] and not self.hay_disparo:
37             self.hay_disparo = True
38             if self.direccion == 'derecha':
39                 self.disparo = MiCuchillo('mi_cuchillo_1.png', 'd',
40                                           self.position[0], self.position[1])
41                 self.parent.add(self.disparo)
42             elif self.direccion == 'izquierda':
43                 self.disparo = MiCuchillo('mi_cuchillo_2.png', 'i',
44                                           self.position[0], self.position[1])
45                 self.parent.add(self.disparo)
46             self.do(Delay(0.2)+ CallFunc(self.nuevo_disparo))
47
48         dx = vx * dt
49         dy = vy * dt
50
51         antes = self.get_rect()
52         despues = antes.copy()
53         despues.x += dx
54         despues.y += dy
55
56         self.velocidad = self.man_col(antes, despues, vx, vy)
57         self.en_suelo = (despues.y == antes.y)
58         self.position = despues.center
59
60         manejador_scroll.set_focus(despues.x, 384)
```

```
61
62      def nuevo_disparo(self):
63          self.hay_disparo = False
64
65
66  class MiCuchillo(Sprite):
67      def __init__(self, image, dir, x, y):
68          super().__init__(image)
69          self.position = (x,y)
70          self.direccion = dir
71      def update(self, dt):
72          if self.direccion == 'd':
73              self.position += Vector2(20,0)
74          elif self.direccion == 'i':
75              self.position -= Vector2(20,0)
76
77
78  class MiDragon(Sprite):
79      def __init__(self, image):
80          super().__init__(image)
81      def update(self, dt):
82          if randint(1,1000) > 980:
83              llama = MiFuegoDragon('mi_fuego_dragon.png', self.x -50, self.y + 40)
84              self.parent.add(llama)
85
86
87  class MiFuegoDragon(Sprite):
88      def __init__(self, image, x, y):
89          super().__init__(image)
90          self.position = (x,y)
91
92      def update(self, dt):
93          self.position += Vector2(-10,0)
94
95
96  class Control(Action):
97      def step(self, dt):
98          for objeto in self.target.parent.children:
99              objeto[1].update(dt)
100
101
102 class Escena(Scene):
103     def __init__(self):
104         global manejador_scroll
105         super().__init__()
106         mi_mapa1 = load_tmx('mapa_plataformas.tmx')['objetos']
107         mi_mapa1_1 = load_tmx('mapa_plataformas.tmx')['capa0']
108
109         dragon = MiDragon('mi_dragon.png')
110         dragon.position = (2500,300)
111
112         personaje = MiGuerrero('mi_guerrero_2.png')
113         personaje.position = (200,300)
114
115         dragon.do(Repeat(MoveBy((0,300), 1) + MoveBy((0,-300), 1)))
116         personaje.do(Control())
117
118         capa_fondo = ScrollableLayer()
119         for i in range(4):
120             a =Sprite('mi_fondo.png')
121             a.position = (640 + 1280*i, 440)
122             capa_fondo.add(a)
123
124         capa_personaje = ScrollableLayer()
125         capa_personaje.add(personaje)
126         capa_personaje.add(dragon)
127
```

```
        manejador_scroll = ScrollingManager()
        manejador_scroll.add(mi_mapa1, z=0)
        manejador_scroll.add(capa_fondo, z=1)
        manejador_scroll.add(mi_mapa1_1, z=2)
        manejador_scroll.add(capa_personaje, z=3)

        mapa_colision = TmxObjectMapCollider()
        mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
        personaje.man_col = make_collision_handler(mapa_colision, mi_mapa1)

        self.add(manejador_scroll)
        director.run(self)


if __name__ == '__main__':
    ventana = director.init(width=1280, height=768, caption='Guerrero vs Dragón')
    ventana.set_location(300,200)
    man_tec = key.KeyStateHandler()
    director.window.push_handlers(man_tec)
    Escena()
```

Una instantánea donde aparece el guerrero y el dragón es:



Comentarios sobre el código:

- Aún no hemos implementado nada referente a las colisiones, por lo que no ocurrirá nada si existen entre los distintos elementos.

- Se definen dos nuevas clases: MiFuegoDragon y MiDragon. Con la primera desplazaremos hacia la izquierda un sprite con la imagen del fuego (**mi_fuego_dragon.png**) , y con la segunda crearemos un sprite con la imagen del dragón (**mi_dragon.png**) que lance de forma aleatoria (L82-84, apoyados en la función randint()) las llamas. Para el movimiento del dragón se usan acciones en L115.

- Si transcurre un cierto tiempo desde la ejecución del código, y más si hemos hecho muchos lanzamientos de cuchillos, podremos observar que el scroll ya no funciona tan fluido, algo que se nota especialmente en los saltos del personaje. Es debido a la gran cantidad de elementos que existen en la escena, ya que no hemos eliminado ni los cuchillos ni los fuegos que salen de la pantalla, por lo que se sigue haciendo cálculos sobre ellos y aumentando la carga computacional que debemos soportar, lo que ralentiza la ejecución del juego. En la próxima etapa solucionaremos este problema, eliminando de la escena los objetos que salen de la pantalla.

Etapa 4

Tendremos ahora en cuenta las colisiones cuchillo-fuego y cuchillo-dragón, reproduciendo una explosión cada vez que alguna de ellas ocurra. El código lo tendremos en **juego_4.py**:

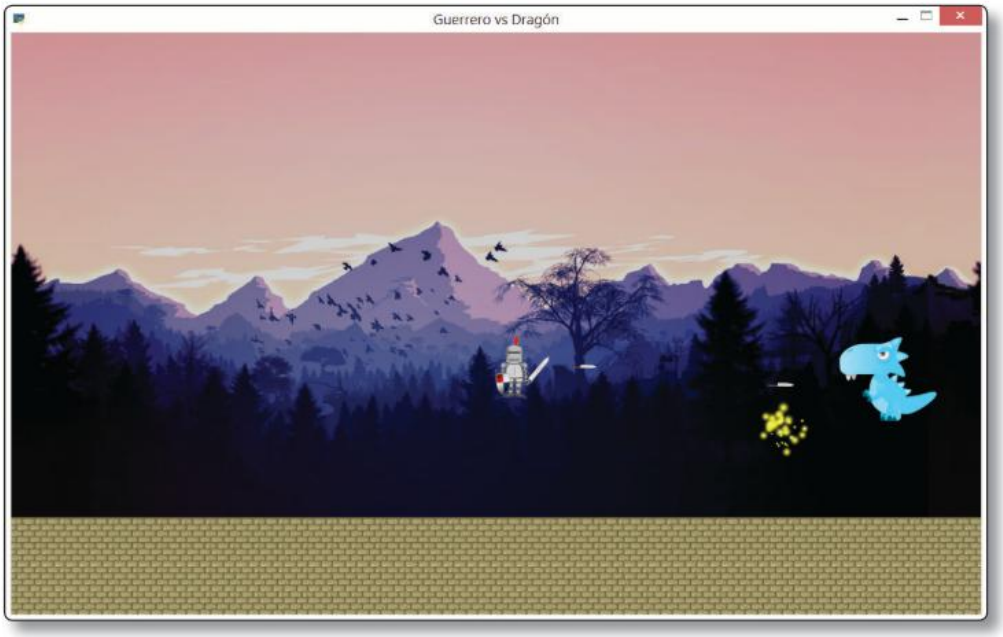
```
1
2  from pyglet.window import key
3  from cocos.sprite import Sprite
4  from cocos.scene import Scene
5  from cocos.layer import ScrollableLayer, ScrollingManager
6  from cocos.tiles import load_tmx
7  from cocos.mapcolliders_plus import TmxObjectMapCollider, make_collision_handler
8  from cocos.collision_model import AARectShape, CollisionManagerBruteForce
9  from cocos.particle_systems import Explosion
10 from cocos.particle import Color
11 from cocos.actions import Action, Delay, CallFunc, MoveBy, Repeat
12 from cocos.director import director
13 from cocos.euclid import Vector2
14 from random import randint
15
16 class MiGuerrero(Sprite):
17     en_suelo = True
18     VEL_MOV = 200
19     VEL_SALTO = 500
20     GRAVEDAD = -800
21
22     def __init__(self, image):
23         super().__init__(image)
24         self.velocidad = (0,0)
25         self.direccion = 'derecha'
26         self.hay_disparo = False
27
28     def update(self, dt):
29         vx, vy = self.velocidad
30         vx = (man_tec[key.RIGHT] - man_tec[key.LEFT]) * self.VEL_MOV
31         vy += self.GRAVEDAD * dt
```

```
        if self.en_suelo and man_tec[key.SPACE]:
            vy = self.VEL_SALTO
        if man_tec[key.LEFT]:
            self.direccion = 'izquierda'
        if man_tec[key.RIGHT]:
            self.direccion = 'derecha'
        if man_tec[key.ENTER] and not self.hay_disparo:
            self.hay_disparo = True
            if self.direccion == 'derecha':
                self.disparo = MiCuchillo('mi_cuchillo_1.png', 'd',
                                          self.position[0], self.position[1])
                self.parent.add(self.disparo)
            elif self.direccion == 'izquierda':
                self.disparo = MiCuchillo('mi_cuchillo_2.png', 'i',
                                          self.position[0], self.position[1])
                self.parent.add(self.disparo)
            self.do(Delay(0.2)+ CallFunc(self.nuevo_disparo))

        dx = vx * dt
        dy = vy * dt

        antes = self.get_rect()
        despues = antes.copy()
        despues.x += dx
        despues.y += dy

        self.velocidad = self.man_col(antes, despues, vx, vy)
        self.en_suelo = (despues.y == antes.y)
        self.position = despues.center

        manejador_scroll.set_focus(despues.x, 384)

    def nuevo_disparo(self):
        self.hay_disparo = False


class MiCuchillo(Sprite):
    def __init__(self, image, dir, x, y):
        super().__init__(image)
        self.position = (x,y)
        self.direccion = dir
        self.cshape = AARectShape(self.position, self.width/5, self.height/5)

    def update(self, dt):
        coor_cuchillo = manejador_scroll.world_to_screen(self.x, self.y)[0]
        if coor_cuchillo> 1280 or coor_cuchillo < 0:
            self.kill()
        if self.direccion == 'd':
            self.position += Vector2(20,0)
        elif self.direccion == 'i':
            self.position -= Vector2(20,0)
        self.cshape.center = Vector2(self.position[0], self.position[1])


class MiDragon(Sprite):
    def __init__(self, image):
        super().__init__(image)
        self.cshape = AARectShape(self.position, self.width/2, self.height/2)

    def update(self, dt):
        if randint(1,1000) > 980:
            llama = MiFuegoDragon('mi_fuego_dragon.png', self.x -50, self.y + 40)
            self.parent.add(llama)
        self.cshape.center = Vector2(self.position[0], self.position[1])

```

```
97
98 class MiFuegoDragon(Sprite):
99     def __init__(self, image, x, y):
100        super().__init__(image)
101        self.position = (x,y)
102        self.cshape = AARectShape(self.position, self.width/5, self.height/5)
103
104    def update(self, dt):
105        self.position += Vector2(-10,0)
106        if self.x < 0:
107            self.kill()
108        self.cshape.center = Vector2(self.position[0], self.position[1])
109
110
111 class MiExplosion1(Explosion):
112    def __init__(self, pos):
113        super().__init__()
114        self.position = pos
115        self.auto_remove_on_finish = True
116        self.total_particles = 700
117        self.emission_rate = 700
118        self.size = 2
119        self.life = 1
120        self.scale = 2
121        self.start_color = Color(255,255,0,255)
122
123    def update(self, dt):
124        pass
125
126
127 class MiExplosion2(Explosion):
128    def __init__(self, pos):
129        super().__init__()
130        self.position = pos
131        self.auto_remove_on_finish = True
132        self.total_particles = 700
133        self.emission_rate = 1200
134        self.size = 10
135        self.life = 3
136        self.start_color = Color(0,255,100,255)
137
138    def update(self, dt):
139        pass
140
141
142 class Control(Action):
143    def start(self):
144        self.mc = CollisionManagerBruteForce()
145
146    def step(self, dt):
147        self.mc.clear()
148        for objeto in self.target.parent.children:
149            if isinstance(objeto[1], (MiCuchillo, MiFuegoDragon, MiDragon)):
150                self.mc.add(objeto[1])
151
152        for elemento in list(self.mc.iter_all_collisions()):
153            a = set([type(elemento[0]), type(elemento[1])])
154            b = set([MiCuchillo, MiFuegoDragon])
155            c = set([MiCuchillo, MiDragon])
156            if  a == b:
157                self.target.parent.add(MiExplosion1(elemento[0].position))
158                if (0, elemento[0]) in self.target.parent.children:
159                    elemento[0].kill()
160                if (0, elemento[1]) in self.target.parent.children:
161                    elemento[1].kill()
162            if a == c:
```

```
                self.target.parent.add(MiExplosion2(elemento[0].position))
                if (0, elemento[0]) in self.target.parent.children:
                    elemento[0].kill()
                if (0, elemento[1]) in self.target.parent.children:
                    elemento[1].kill()

        for objeto in self.target.parent.children:
            objeto[1].update(dt)


class Escena(Scene):
    def __init__(self):
        global manejador_scroll
        super().__init__()
        mi_mapa1 = load_tmx('mapa_plataformas.tmx')['objetos']
        mi_mapa1_1 = load_tmx('mapa_plataformas.tmx')['capa0']

        dragon = MiDragon('mi_dragon.png')
        dragon.position = (2500,180)
        dragon.do(Repeat(MoveBy((0,300), 1) + MoveBy((0,-300), 1)))

        personaje = MiGuerrero('mi_guerrero_2.png')
        personaje.position = (200,300)
        personaje.do(Control())

        capa_fondo = ScrollableLayer()
        for i in range(4):
            a = Sprite('mi_fondo.png')
            a.position = (640 + 1280*i, 440)
            capa_fondo.add(a)

        capa_personaje = ScrollableLayer()
        capa_personaje.add(dragon)
        capa_personaje.add(personaje)

        manejador_scroll = ScrollingManager()
        manejador_scroll.add(mi_mapa1, z=0)
        manejador_scroll.add(capa_fondo, z=1)
        manejador_scroll.add(mi_mapa1_1, z=2)
        manejador_scroll.add(capa_personaje, z=3)

        mapa_colision = TmxObjectMapCollider()
        mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
        personaje.man_col = make_collision_handler(mapa_colision, mi_mapa1)

        self.add(manejador_scroll)
        director.run(self)


if __name__ == '__main__':
    ventana = director.init(width=1280, height=768, caption='Guerrero vs Dragón')
    ventana.set_location(300,200)
    man_tec = key.KeyStateHandler()
    director.window.push_handlers(man_tec)
    Escena()
```

En la siguiente imagen podemos visualizar el instante ligeramente posterior a una colisión cuchillo-fuego:



Comentarios sobre el código:

- Como ahora vamos a considerar colisiones entre instancias de las clases MiCuchillo, MiDragon y MiFuegoDragon, debemos incorporar en cada una de ellas un atributo cshape, que en nuestro caso serán instancias de AARectShape[100]. En cada ejecución de sus métodos update() estaremos atentos a actualizar la posición de cshape.

- Creamos dos clases nuevas (MiExplosion1 y MiExplosion2) basadas en la clase Explosion del módulo cocos.particle_systems. Con ellas simularemos explosiones asociadas, respectivamente, a la colisión cuchillo-fuego y cuchillo-dragón.

- Ampliamos bastante la clase Control, ya que es allí donde comprobaremos las posibles colisiones entre los objetos que hemos considerado inicialmente. Mediante el método start() inicializo la instancia, creando un atributo que albergará el manejador de colisiones, un objeto de tipo CollisionManagerBruteForce ya que el número de elementos no es elevado.

100 En MiCuchillo y MiFuegoDragon en lugar de la mitad del tamaño del sprite he usado una quinta parte, para hacer las colisiones entre sus instancias un poco más precisas.

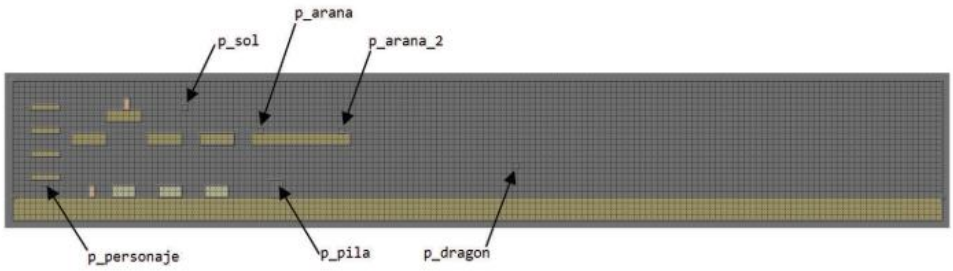
- En L147-150 vacío el manejador y le añado los objetos que tengamos de las tres clases. En L152-167 escaneamos todas las posibles colisiones que tengamos y comprobamos si son de los dos tipos elegidos, en cuyo caso eliminamos las instancias colisionadoras[101] y reproducimos la explosión pertinente. Con L169-170 actualizamos todos los objetos que tenemos en la capa del personaje[102].

- Hemos usado en L76 el método world_to_screen() del manejador de scroll para conseguir las coordenadas relativas a la pantalla que tiene en cada momento el cuchillo, con el propósito de que cuando salga de ella (algo que comprobamos en L77) sea eliminado (L78). Para el caso del fuego de dragón simplemente lo eliminamos si sus coordenada x absoluta es menor que 0 (L106-107).

Etapa 5

Incorporaremos ahora dos elementos basados en sistemas de partículas (sol y humo) además de añadir un par de nuevos enemigos (arañas) que se mueven en una de las plataformas. Seguiremos considerando de cara a colisiones solamente los dos casos vistos en la etapa 4.

Añadiremos previamente a mapa_plataformas.tmx una serie de objetos rectangulares, a los que daremos atributos de tipo booleano con el nombre que indicamos a continuación[103], donde se muestran las dos capas habituales:



101 Hacer notar que son necesarios los condicionales de L158-161 y L164-167 para que, como step() se ejecuta muchas veces en un corto espacio de tiempo, no se intente borrar una instancia que ya ha sido borrada, algo que nos lanzaría un error.

102 Para ser más ortodoxos con lo que comentamos en el Capítulo 1 colocaríamos estas dos líneas de código tras L147, pero donde están también nos garantiza que los objetos añadidos al manejador tengan el mismo valor de cshape en el momento de la consulta y en el momento en que se añadieron.

103 El objeto con el atributo p_pila también tiene el atributo solidas, de tipo string y conteniendo la cadena vacía.

El resultado lo guardamos en **mapa_plataformas_2.tmx**.

Dada la extensión del código incluiré sólo los bloques de **juego_5.py** que se añaden y/o modifican respecto del anterior.

Bloque 1:

```
85
86 class MiObjeto(Sprite):
87     def __init__(self, image):
88         super().__init__(image)
89
90
91 class MiEnemigo(Sprite):
92     def __init__(self, image):
93         super().__init__(image)
94         self.cshape = AARectShape(self.position, self.width/2, self.height/2)
95
96     def update(self, dt):
97         self.cshape.center = Vector2(self.position[0], self.position[1])
98
99
100 class MiDragon(MiEnemigo):
101     def __init__(self, image):
102         super().__init__(image)
103
104     def update(self, dt):
105         if randint(1,1000) > 980:
106             llama = MiFuegoDragon('mi_fuego_dragon.png', self.x - 50, self.y + 40)
107             self.parent.add(llama)
108         self.cshape.center = Vector2(self.position[0], self.position[1])
109
```

Bloque 2:

```
181
182         for objeto in self.target.parent.children:
183             if not isinstance(objeto[1], (MiObjeto, Smoke, Sun)):
184                 objeto[1].update(dt)
185
```

Bloque 3:

```
186
187 class Escena(Scene):
188     def __init__(self):
189         global  manejador_scroll
190         super().__init__()
191         mi_mapa1 = load_tmx('mapa_plataformas_2.tmx')['objetos']
192         mi_mapa1_1 = load_tmx('mapa_plataformas_2.tmx')['capa0']
193
194         dragon = MiDragon('mi_dragon.png')
195         ref_dra = mi_mapa1.find_cells(p_dragon=True)[0]
196         dragon.position = (ref_dra.x, ref_dra.y)
197         mi_mapa1.objects.remove(ref_dra)
198         dragon.do(Repeat(MoveBy((0, 300), 1) + MoveBy((0, -300), 1)))
199
200         araña1 = MiEnemigo('mi_araña_2.png')
201         ref_ara = mi_mapa1.find_cells(p_arana=True)[0]
```

```
        araña1.position = (ref_ara.x, ref_ara.y +10)
        mi_mapa1.objects.remove(ref_ara)
        araña1.do(Repeat(MoveBy((450, 0), 3) + MoveBy((-450, 0), 3)))

        araña2 = MiEnemigo('mi_araña_2.png')
        ref_ara = mi_mapa1.find_cells(p_arana_2=True)[0]
        araña2.position = (ref_ara.x, ref_ara.y +10)
        mi_mapa1.objects.remove(ref_ara)
        araña2.do(Repeat(MoveBy((-450, 0), 3) + MoveBy((450, 0), 3)))

        pila = MiObjeto('mi_pila.png')
        ref_pila = mi_mapa1.find_cells(p_pila=True)[0]
        pila.position = ref_pila.center

        humo_pila = Smoke()
        humo_pila.position = pila.position + Vector2(0, 40)

        sol = Sun()
        ref_sol = mi_mapa1.find_cells(p_sol=True)[0]
        sol.position = Vector2(ref_sol.x, ref_sol.y)
        mi_mapa1.objects.remove(ref_sol)

        personaje = MiGuerrero('mi_guerrero_2.png')
        ref_per = mi_mapa1.find_cells(p_personaje=True)[0]
        personaje.position = (ref_per.x, ref_per.y)
        mi_mapa1.objects.remove(ref_per)
        personaje.do(Control())

        capa_fondo = ScrollableLayer()
        for i in range(4):
            a = Sprite('mi_fondo.png')
            a.position = (640 + 1280*i, 440)
            capa_fondo.add(a)

        capa_personaje = ScrollableLayer()
        capa_personaje.add(dragon)
        capa_personaje.add(araña1)
        capa_personaje.add(araña2)
        capa_personaje.add(pila)
        capa_personaje.add(humo_pila)
        capa_personaje.add(sol)
        capa_personaje.add(personaje)

        manejador_scroll = ScrollingManager()
        manejador_scroll.add(mi_mapa1, z=0)
        manejador_scroll.add(capa_fondo, z=1)
        manejador_scroll.add(mi_mapa1_1, z=2)
        manejador_scroll.add(capa_personaje, z=3)

        mapa_colision = TmxObjectMapCollider()
        mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
        personaje.man_col = make_collision_handler(mapa_colision, mi_mapa1)

        self.add(manejador_scroll)
        director.run(self)
```

Se muestra a continuación una imagen del juego donde se visualizan los nuevos elementos:

Comentarios sobre el código:

- Definiremos la clase MiObjeto para ser la base de los objetos estáticos que aparecen en el juego, como puede ser la pila que aparece a la derecha del guerrero en la imagen superior.
- También definiremos la clase MiEnemigo para que los personajes que atacan al guerrero sean instancias suyas (por ejemplo las dos arañas) o de clases que heredan de ella (como ocurre ahora con MiDragon).
- En L182-184 añadimos ahora al manejador de colisiones todas las instancias que no sean de la clase MiObjeto, Smoke o Sun.
- En la clase Escena colocamos el dragón, las arañas, la pila, el humo de la pila, el sol y el guerrero (todos ellos en capa_personaje) en las posiciones marcadas en mapa_plataformas_2.tmx por los atributos de los objetos Tmx. Mediante el método find_cells() conseguimos el objeto que queramos y, tras usarlo para obtener sus coordenadas, lo borramos, con la única excepción del usado para la pila, que lo mantendremos para hacerla sólida ante el guerrero. El movimiento de las arañas lo conseguimos, igual que hicimos con el dragón, mediante acciones.

Etapa 6 (y última)

En la versión final del juego trataremos las colisiones (y sus efectos) cuchillo-araña, guerrero-araña, guerrero-dragón y guerrero-fuego. Habrá varias escenas: de inicio, de misión completada, de "Game Over", y en la principal del juego un HUD que nos informe de vidas y puntos. Añadiremos además música de fondo, sonidos (explosiones, lanzamiento de cuchillos, emisión de fuego, y si alcanzan a nuestro guerrero).

El contenido de **juego_6.py** es el siguiente:

```
1
2  from pyglet.window import key
3  from cocos.sprite import Sprite
4  from cocos.scene import Scene
5  from cocos.layer import Layer, ScrollableLayer, ScrollingManager
6  from cocos.tiles import load_tmx
7  from cocos.mapcolliders_plus import TmxObjectMapCollider, make_collision_handler
8  from cocos.collision_model import AARectShape, CollisionManagerBruteForce
9  from cocos.particle_systems import Explosion, Smoke, Sun
10 from cocos.particle import Color
11 from cocos.actions import Action, Delay, CallFunc, MoveBy, Repeat, FadeOut
12 from cocos.director import director
13 from cocos.euclid import Vector2
14 from cocos.text import Label
15 from cocos.audio.effect import Effect
16 from random import randint
17
18 class MiGuerrero(Sprite):
19     en_suelo = True
20     VEL_MOV = 200
21     VEL_SALTO = 500
22     GRAVEDAD = -800
23
24     def __init__(self, image, x, y):
25         super().__init__(image)
26         self.position = (x, y)
27         self.velocidad = (0,0)
28         self.direccion = 'derecha'
29         self.hay_disparo = False
30         self.reviviendo = False
31         self.cshape = AARectShape(self.position, self.width/2, self.height/2)
32
33     def update(self, dt):
34         if self.reviviendo == False:
35             vx, vy = self.velocidad
36             vx = (man_tec[key.RIGHT] - man_tec[key.LEFT]) * self.VEL_MOV
37             vy += self.GRAVEDAD * dt
38             if self.en_suelo and man_tec[key.SPACE]:
39                 vy = self.VEL_SALTO
40             if man_tec[key.LEFT]:
41                 self.direccion = 'izquierda'
42             if man_tec[key.RIGHT]:
43                 self.direccion = 'derecha'
44             if man_tec[key.ENTER] and not self.hay_disparo:
45                 self.hay_disparo = True
46                 if self.direccion == 'derecha':
47                     sonido = Effect('cuchillo.wav')
48                     sonido.sound.set_volume(0.5)
49                     sonido.sound.play()
50                     self.disparo = MiCuchillo('mi_cuchillo_1.png', 'd',
51                                               self.position[0], self.position[1])
52                     self.parent.add(self.disparo)
```

```
                elif self.direccion == 'izquierda':
                    sonido = Effect('cuchillo.wav')
                    sonido.sound.set_volume(0.5)
                    sonido.sound.play()
                    self.disparo = MiCuchillo('mi_cuchillo_2.png', 'i',
                                              self.position[0], self.position[1])
                    self.parent.add(self.disparo)
                self.do(Delay(0.2)+ CallFunc(self.nuevo_disparo))

            dx = vx * dt
            dy = vy * dt

            antes = self.get_rect()
            despues = antes.copy()
            despues.x += dx
            despues.y += dy

            self.velocidad = self.man_col(antes, despues, vx, vy)
            self.en_suelo = (despues.y == antes.y)

            if despues.x < ref_salida.x -self.width:
                self.position = despues.center
            else:
                director.replace(Scene(CapaGanador()))
            self.cshape.center = Vector2(self.position[0], self.position[1])
            if despues.x >= 640:
                manejador_scroll.set_focus(despues.x, 384)

    def nuevo_disparo(self):
        self.hay_disparo = False


class MiCuchillo(Sprite):
    def __init__(self, image, dir, x, y):
        super().__init__(image)
        self.position = (x,y)
        self.direccion = dir
        self.cshape = AARectShape(self.position, self.width/5, self.height/5)

    def update(self, dt):
        coor_cuchillo = manejador_scroll.world_to_screen(self.x, self.y)[0]
        if coor_cuchillo> 1280 or coor_cuchillo < 0:
            self.kill()
        if self.direccion == 'd':
            self.position += Vector2(20,0)
        elif self.direccion == 'i':
            self.position -= Vector2(20,0)
        self.cshape.center = Vector2(self.position[0], self.position[1])


class MiObjeto(Sprite):
    def __init__(self, image, x, y):
        super().__init__(image)
        self.position = (x, y)


class MiEnemigo(Sprite):
    def __init__(self, image, x, y):
        super().__init__(image)
        self.position = (x, y)
        self.cshape = AARectShape(self.position, self.width/2, self.height/2)

    def update(self, dt):
        self.cshape.center = Vector2(self.position[0], self.position[1])


class MiDragon(MiEnemigo):
```

```
    def __init__(self, image, x, y):
        super().__init__(image, x, y)
        self.cshape = AARectShape(self.position, self.width/2, self.height/2)

    def update(self, dt):
        if randint(1,1000) > 980:
            sonido_llama = Effect('fuego_dragon.wav')
            sonido_llama.sound.set_volume(0.1)
            sonido_llama.sound.play()
            llama = MiFuegoDragon('mi_fuego_dragon.png', self.x -50, self.y + 40)
            self.parent.add(llama)
        self.cshape.center = Vector2(self.position[0], self.position[1])


class MiFuegoDragon(Sprite):
    def __init__(self, image, x, y):
        super().__init__(image)
        self.position = (x,y)
        self.cshape = AARectShape(self.position, self.width/5, self.height/5)

    def update(self, dt):
        self.position += Vector2(-10,0)
        if self.x < 0:
            self.kill()
        self.cshape.center = Vector2(self.position[0], self.position[1])


class MiExplosion1(Explosion):
    def __init__(self, pos):
        super().__init__()
        self.position = pos
        self.auto_remove_on_finish = True
        self.total_particles = 700
        self.emission_rate = 700
        self.size = 2
        self.life = 1
        self.scale = 2
        self.start_color = Color(255,255,0,255)

    def update(self, dt):
        pass


class MiExplosion2(Explosion):
    def __init__(self, pos):
        super().__init__()
        self.position = pos
        self.auto_remove_on_finish = True
        self.total_particles = 700
        self.emission_rate = 1200
        self.size = 10
        self.life = 3
        self.start_color = Color(0,255,100,255)

    def update(self, dt):
        pass


class MiExplosion3(Explosion):
    def __init__(self, pos):
        super().__init__()
        self.position = pos
        self.total_particles = 100
        self.size = 2
        self.life = 1
        self.auto_remove_on_finish = True
        self.start_color = Color(0,0,255,255)
```

```
187
188     def update(self, dt):
189         pass
190
191
192 class Control(Action):
193     def start(self):
194         self.rel = [set([MiGuerrero, MiFuegoDragon]), set([MiGuerrero, MiDragon]),
195                     set([MiGuerrero, MiEnemigo])]
196         self.mc = CollisionManagerBruteForce()
197
198     def step(self, dt):
199         self.mc.clear()
200         for objeto in self.target.parent.children:
201             if isinstance(objeto[1], (MiGuerrero, MiCuchillo,
202                                       MiEnemigo, MiDragon, MiFuegoDragon)):
203                 self.mc.add(objeto[1])
204
205         for elemento in list(self.mc.iter_all_collisions()):
206             a = set([type(elemento[0]), type(elemento[1])])
207             b = set([MiCuchillo, MiFuegoDragon])
208             c = set([MiCuchillo, MiDragon])
209             d = set([MiCuchillo, MiEnemigo])
210
211             if a in self.rel and personaje.reviviendo == False:
212                 sonido = Effect('golpeo_personaje.wav')
213                 sonido.play()
214                 personaje.visible = False
215                 personaje.reviviendo = True
216                 personaje.do(Delay(3) + CallFunc(self.f1))
217             if a == b :
218                 sonido= Effect('col_cuchillo_enemigo.wav')
219                 sonido.sound.set_volume(0.5)
220                 sonido.sound.play()
221                 self.target.parent.add(MiExplosion1(elemento[0].position))
222                 if (0, elemento[0]) in self.target.parent.children:
223                     elemento[0].kill()
224                 if (0, elemento[1]) in self.target.parent.children:
225                     elemento[1].kill()
226                 hud.puntos += 5
227             if a == c :
228                 sonido= Effect('col_cuchillo_dragon.wav')
229                 sonido.sound.set_volume(0.5)
230                 sonido.sound.play()
231                 self.target.parent.add(MiExplosion2(elemento[0].position))
232                 if (0, elemento[0]) in self.target.parent.children:
233                     elemento[0].kill()
234                 if (0, elemento[1]) in self.target.parent.children:
235                     elemento[1].kill()
236                 hud.puntos += 50
237             if a == d:
238                 sonido= Effect('col_cuchillo_enemigo.wav')
239                 sonido.sound.set_volume(0.5)
240                 sonido.sound.play()
241                 self.target.parent.add(MiExplosion3(elemento[0].position))
242                 if (0, elemento[0]) in self.target.parent.children:
243                     elemento[0].kill()
244                 if (0, elemento[1]) in self.target.parent.children:
245                     elemento[1].kill()
246                 hud.puntos += 15
247
248         for objeto in self.target.parent.children:
249             if not isinstance(objeto[1], (MiObjeto, Smoke, Sun)):
250                 objeto[1].update(dt)
251
252     def f1(self):
253         hud.vidas -= 1
```

```
        personaje.reviviendo = False
        if hud.vidas > 0:
            personaje.visible = True


class MiEtiqueta(Label):
    def __init__(self, texto, x, y, c = (255,255,255,255)):
        super().__init__(texto, (x, y), font_name = 'Consolas', font_size = 14,
                         color = c, anchor_x ='center', anchor_y = 'center')


class MiHUD(Layer):
    def __init__(self):
        super().__init__()
        self.vidas = 2
        self.puntos = 0

    def update(self, dt):
        self.children = []
        texto_vidas = 'Vidas: ' + str(self.vidas)
        etiqueta_vidas = MiEtiqueta(texto_vidas, 70, 740, (0,255,0,255))
        texto_puntos = 'Puntos: ' + str(self.puntos)
        etiqueta_puntos = MiEtiqueta(texto_puntos, 1180, 740, (0,255,255,255))
        self.add(etiqueta_vidas)
        self.add(etiqueta_puntos)
        delta_scroll = manejador_scroll.world_to_screen(70, 740)[0]
        if self.vidas == 0:
            director.replace(Scene(CapaGameOver()))
        self.x = 70 - delta_scroll


class CapaInicio(Layer):
    is_event_handler = True
    def __init__(self):
        super().__init__()
        self.add(MiEtiqueta('Pulsa botón de ratón para iniciar', 640, 384))

    def on_mouse_release(self, x, y, buttons, modifiers):
        global musica_de_fondo
        musica_de_fondo = Effect('musica_fondo_retro.wav')
        musica_de_fondo.sound.set_volume(0.1)
        musica_de_fondo.sound.play(-1)
        director.replace(Escena())


class CapaGanador(Layer):
    def __init__(self):
        super().__init__()
        musica_de_fondo.sound.stop()
        self.add(MiEtiqueta('¡Enhorabuena, has completado la misión!', 640, 384))
        self.do(Delay(3) + CallFunc(lambda : director.replace(Scene(CapaInicio()))))


class CapaGameOver(Layer):
    def __init__(self):
        super().__init__()
        musica_de_fondo.sound.stop()
        self.add(MiEtiqueta('GAME OVER', 640, 384))
        self.do(Delay(3) + CallFunc(lambda: director.replace(Scene(CapaInicio()))))


class Escena(Scene):
    def __init__(self):
        global personaje, manejador_scroll, hud, ref_salida
        super().__init__()
        mi_mapa1 = load_tmx('mapa_plataformas_2.tmx')['objetos']
        mi_mapa1_1 = load_tmx('mapa_plataformas_2.tmx')['capa0']
```

```
321
322         ref_dra = mi_mapa1.find_cells(p_dragon=True)[0]
323         dragon = MiDragon('mi_dragon.png', ref_dra.x, ref_dra.y)
324         mi_mapa1.objects.remove(ref_dra)
325         dragon.do(Repeat(MoveBy((0, 300), 1) + MoveBy((0, -300), 1)))
326
327         ref_ara = mi_mapa1.find_cells(p_arana=True)[0]
328         araña1 = MiEnemigo('mi_araña_2.png', ref_ara.x, ref_ara.y +10)
329         mi_mapa1.objects.remove(ref_ara)
330         araña1.do(Repeat(MoveBy((450, 0), 3) + MoveBy((-450, 0), 3)))
331
332         ref_ara = mi_mapa1.find_cells(p_arana_2=True)[0]
333         araña2 = MiEnemigo('mi_araña_2.png', ref_ara.x, ref_ara.y +10)
334         mi_mapa1.objects.remove(ref_ara)
335         araña2.do(Repeat(MoveBy((-450, 0), 3) + MoveBy((450, 0), 3)))
336
337         ref_pila = mi_mapa1.find_cells(p_pila=True)[0]
338         pila = MiObjeto('mi_pila.png', ref_pila.center[0], ref_pila.center[1])
339
340         humo_pila = Smoke()
341         humo_pila.position = pila.position + Vector2(0, 40)
342
343         sol = Sun()
344         ref_sol = mi_mapa1.find_cells(p_sol=True)[0]
345         sol.position = Vector2(ref_sol.x, ref_sol.y)
346         mi_mapa1.objects.remove(ref_sol)
347
348         ref_per = mi_mapa1.find_cells(p_personaje=True)[0]
349         personaje = MiGuerrero('mi_guerrero_2.png', ref_per.x, ref_per.y)
350         mi_mapa1.objects.remove(ref_per)
351         personaje.do(Control())
352
353         ref_salida = mi_mapa1.find_cells(p_salida=True)[0]
354
355         capa_fondo = ScrollableLayer()
356         for i in range(4):
357             a = Sprite('mi_fondo.png')
358             a.position = (640 + 1280*i, 440)
359             capa_fondo.add(a)
360
361         hud = MiHUD()
362
363         capa_personaje = ScrollableLayer()
364         capa_personaje.add(dragon)
365         capa_personaje.add(araña1)
366         capa_personaje.add(araña2)
367         capa_personaje.add(personaje)
368         capa_personaje.add(pila)
369         capa_personaje.add(humo_pila)
370         capa_personaje.add(sol)
371         capa_personaje.add(hud)
372
373         manejador_scroll = ScrollingManager()
374         manejador_scroll.add(mi_mapa1, z=0)
375         manejador_scroll.add(capa_fondo, z=1)
376         manejador_scroll.add(mi_mapa1_1, z=2)
377         manejador_scroll.add(capa_personaje, z=3)
378
379         mapa_colision = TmxObjectMapCollider()
380         mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
381         personaje.man_col = make_collision_handler(mapa_colision, mi_mapa1)
382
383         self.add(manejador_scroll)
384         director.run(self)
385
386 def cierra_ventana():
387     musica_de_fondo.sound.stop()
```

```
388     director.window.close()
389
390
391 if __name__ == '__main__':
392     ventana = director.init(width=1280, height=768, caption='Guerrero vs Dragón',
393                             audio_backend='sdl')
394     ventana.set_location(300,200)
395     man_tec = key.KeyStateHandler()
396     director.window.push_handlers(man_tec)
397     director.window.on_close = cierra_ventana
398     director.run(Scene(CapaInicio()))
399
```

Comentarios sobre el código:

- Definimos la clase MiEtiqueta para poder crear etiquetas de color al instanciarla. Las usaremos en las distintas escenas y en el HUD.
- Definimos tres clases (CapaInicio, CapaGanador y CapaGameOver) para tres capas en las que se presentará un texto por pantalla. La primera (dentro de su escena) aparecerá nada más ejecutar el código, y tras pulsar un botón del ratón accederemos a la escena principal del juego. Si llegamos a su límite derecho se ejecutará CapaGanador, y si nos alcanzan dos veces será CapaGameOver la que lo haga. Tras cualquiera de estas dos últimas, y con tres segundos de retraso, se volverá a ejecutar CapaInicio.
- Definimos la clase MiHud, una capa donde colocaremos texto informando del número de vidas restante y de los puntos acumulados. Su instancia la incluiremos en capa_personaje.
- La música de fondo será añadida en la escena principal del juego, y la desactivaremos en el resto de escenas o al cerrar la ventana (de la forma que ya conocemos, apoyándonos en la función cierra_ventana()).
- Crearemos el atributo adicional reviviendo en la clase MiGuerrero. Con él indicaremos si, tras ser alcanzado nuestro personaje, está en el periodo de 3 segundos de espera en los que no es visible.
- Ahora la clase Control es un poco más compleja, ya que tenemos que considerar un mayor número de posibles colisiones, además de los sonidos, explosiones u operaciones asociado/as a cada una de ellas. Haremos uso de la nueva clase MiExplosion3 cuando haya un impacto cuchillo-araña.

En este punto el lector podrá considerar qué elementos mejorar y/o añadir para la creación de un juego más completo. Por ejemplo incorporar un menú al inicio, o transiciones entre escenas, serían tareas muy sencillas. Implementar otras características podría ser una labor más compleja, como veremos en el siguiente capítulo.

5

UN PASO MÁS Y OTROS USOS DE COCOS2D

En el presente capítulo comentaré cómo añadir más elementos a nuestro juego de plataformas, poniendo dos ejemplos concretos. Además se mostrará el uso de cocos2d en una sencilla presentación y una aplicación interactiva basada en un juego. Los cuatro casos se presentan como potenciales ejercicios para resolver previamente a visualizar la solución dada por mí[104], algo que proporcionará una mejor medida de la dificultad exacta que conlleva cada uno.

Debido a los conocimientos de los que ya disponemos, la totalidad de los códigos que aparecen en este capítulo no serán analizados, quedando esa labor para el lector.

5.1 AÑADIENDO NUEVAS CARACTERÍSTICAS A NUESTRO JUEGO

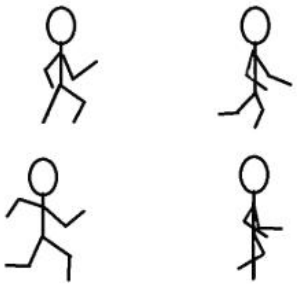
Hemos conseguido finalmente nuestro objetivo de crear un sencillo juego de plataformas. Tendremos la posibilidad de añadir, con los conocimientos de los que ya disponemos, muchas características adicionales para conseguir juegos más completos. Dos de ellas podrían ser las siguientes:

1. Animar el personaje para representar cosas como correr, saltar o trepar.
2. Tener la posibilidad de interactuar con elementos para, por ejemplo, agarrar una liana o escalar una pared.

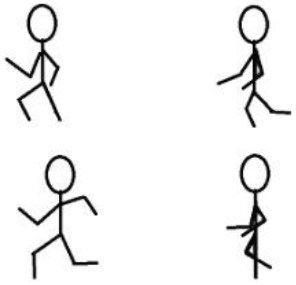
104 Con la posibilidad clara de mejorarla.

Hemos de tener en cuenta que nuestro guerrero siempre tiene la misma apariencia, dando igual si se mueve, salta o lanza un cuchillo. Para dar más realismo al juego sería interesante que los **personajes estuviesen animados**. Por ejemplo un enemigo que nos lanza una piedra y hace los movimientos necesarios para ello, o en el caso del guerrero que al moverse simulase correr.

Para conseguirlo usaré las animaciones, vistas en el apartado 1.2.4, por lo que previamente tenemos que conseguir o crear los fotogramas que componen la acción de correr. Como el propósito es simplemente didáctico los crearé usando Paint[105] (un total de 4) y compondré (usando la misma herramienta) una imagen que los contenga, **fotogramas_dcha.png**:



La secuencia corresponde a correr hacia la derecha[106]. Como queremos también implementar correr en el otro sentido modificamos la imagen[107] con la ayuda de GIMP para ello, obteniendo **fotogramas_izqda.png**:



105 Sencillo editor de imágenes que viene con Windows.

106 Con un estilo no demasiado elegante.

107 La volteamos mediante Imagen → Transformar → Voltear horizontalmente y posteriormente intercambiamos las columnas.

También tendremos una imagen para cuando no corramos en ninguna dirección. Será **fotograma_base.png**[108]:

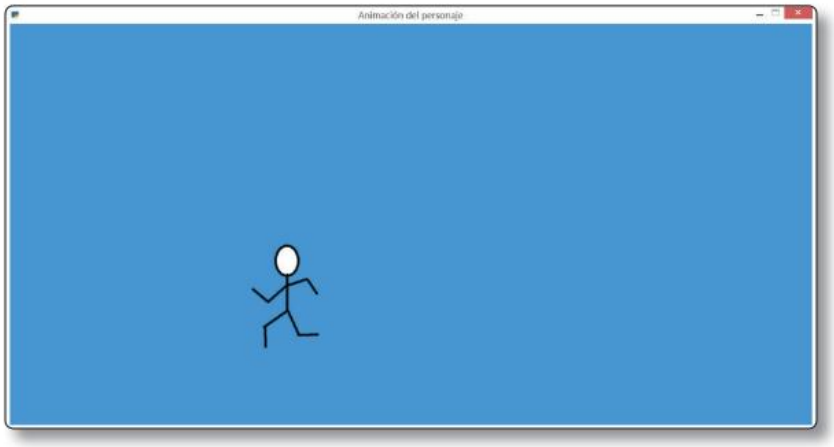
El código es **ejemplo_animacion_personaje.py**:

```
1
2  from pyglet.window import key
3  from pyglet.image import load, ImageGrid
4  from cocos.director import director
5  from cocos.layer import Layer, ColorLayer
6  from cocos.scene import Scene
7  from cocos.sprite import Sprite
8  from cocos.euclid import Vector2
9  from collections import defaultdict
10
11 class MiObjeto(Sprite):
12     def __init__(self, image, x, y):
13         super().__init__(image)
14         self.position = (x, y)
15
16     def move(self, offset):
17         self.position += offset
18
19     def update(self, delta_t):
20         pass
21
22
23 class MiPersonaje(MiObjeto):
24     TECLAS_PULSADAS = defaultdict(int)
25     pulsado = False
26
27     def __init__(self,imagen, x, y):
28         super().__init__(imagen, x, y)
29         self.velocidad = Vector2(400, 0)
30
31     def update(self, delta_t):
32         pulsadas = MiPersonaje.TECLAS_PULSADAS
33
34         movimiento = pulsadas[key.RIGHT] - pulsadas[key.LEFT]
35         if movimiento != 0:
36             delta_x = (self.velocidad * movimiento * delta_t)[0]
37             if self.x  <= self.parent.ancho_ventana - self.width/5 - delta_x:
38                 if self.x - self.width/5 + delta_x > abs(delta_x):
39                     self.move(self.velocidad * movimiento * delta_t)
40
41         if pulsadas[key.RIGHT] and not self.pulsado:
42             self.pulsado = True
43             self.image = mi_animacion
44         if pulsadas[key.LEFT] and not self.pulsado:
45             self.pulsado = True
46             self.image = mi_animacion2
47         if not pulsadas[key.RIGHT] and not pulsadas[key.LEFT]:
48             self.image = load('fotograma_base.png')
49             self.pulsado = False
50
```

108 Sí, simula estar de brazos cruzados :) .

```
51
52 class MiCapa(ColorLayer):
53     is_event_handler = True
54
55     def __init__(self):
56         super().__init__(0, 150, 255, 255)
57         global mi_animacion, mi_animacion2
58         self.ancho_ventana, _ = director.get_window_size()
59
60         mi_secuencia = ImageGrid(load('fotogramas_dcha.png'), 2, 2)
61         mi_secuencia2 = ImageGrid(load('fotogramas_izqda.png'), 2, 2)
62         mi_animacion = mi_secuencia.get_animation(0.2)
63         mi_animacion2 = mi_secuencia2.get_animation(0.2)
64
65         self.personaje = MiPersonaje(mi_animacion, self.ancho_ventana * 0.5, 250)
66         self.add(self.personaje)
67         self.schedule(self.update)
68
69     def on_key_press(self, k, _):
70         MiPersonaje.TECLAS_PULSADAS[k] = 1
71
72     def on_key_release(self, k, _):
73         MiPersonaje.TECLAS_PULSADAS[k] = 0
74
75     def update(self, dt):
76         self.children[0][1].update(dt)
77
78
79 if __name__ == '__main__':
80     ventana = director.init(caption='Animación del personaje',
81                             width=1600, height=800)
82     ventana.set_location(150,100)
83     director.run(Scene(MiCapa()))
84
```

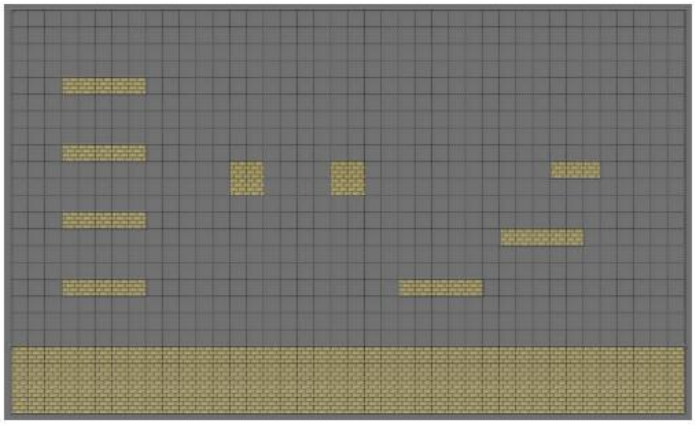
Al ejecutarlo podremos desplazar el personaje horizontalmente (en ambos sentidos) dentro de los límites de la ventana, simulando correr:



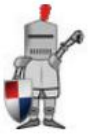
El tamaño elegido para la ventana ha sido de 1600 x 800 píxeles y la he colocado en las coordenadas (150,100). El lector puede adaptar esos valores a la resolución de pantalla que tenga o usar, como siempre si lo deseamos, el modo de pantalla completa (CTRL+f).

Adicionalmente, es muy interesante que tengamos la capacidad de **interactuar con objetos** para implementar cosas como recogerlos o agarrarlos. Como ejemplos pensemos en el primer caso en monedas y en el segundo en lianas. Trabajaremos sobre el modelo de colisión para alcanzar nuestro objetivo.

En el siguiente código que presentaré he añadido al guerrero la posibilidad de agarrar lianas[109] y saltar entre ellas[110], además de trepar o descender por setos[111]. El mapa base será **mapa_plataformas_3.tmx**:



Ahora tendremos dos imágenes para el guerrero: la habitual hasta el momento y la que representa agarrar (tanto una liana como el seto), que se ha creado de forma bastante artesanal mediante GIMP y que tenemos en **guerrero_agarrando.png**:



109 Cambiando la imagen del guerrero para representar el agarre.

110 Pulsando la barra espaciadora teniendo pulsada cualquiera de las dos teclas de cursor de desplazamiento horizontal.

111 Usaremos para ello las dos teclas de cursor de desplazamiento vertical.

También he modificado ligeramente mediante GIMP la imagen de fondo usada en el juego de plataformas para darle un tono más luminoso. Está en **mi_fondo_2.png**.

Usaremos también las imágenes **liana.png** y **seto.png** que contienen lo que indica su nombre.

El código (**ejemplo_interactuar_objetos.py**) es el siguiente:

```
1
2  from pyglet.window import key
3  from pyglet.image import load
4  from cocos.sprite import Sprite
5  from cocos.scene import Scene
6  from cocos.layer import ScrollableLayer, ScrollingManager
7  from cocos.tiles import load_tmx
8  from cocos.mapcolliders_plus import make_collision_handler, TmxObjectMapCollider
9  from cocos.collision_model import CollisionManagerBruteForce
10 from cocos.actions import Action, JumpBy
11 from cocos.particle_systems import Sun
12 from cocos.director import director
13 from cocos.collision_model import AARectShape
14 from cocos.euclid import Vector2
15
16 class MiGuerrero(Sprite):
17     VEL_MOV = 200
18     VEL_SALTO = 500
19     GRAVEDAD = -800
20     en_suelo = True
21     modo = 'normal'
22
23     def __init__(self, image):
24         super().__init__(image)
25         self.velocidad = (0, 0)
26         self.cshape = AARectShape(Vector2(0,0), self.width/3, self.height/3)
27
28     def update(self, dt):
29         if self.modo == 'normal':
30             vx, vy = self.velocidad
31             vx = (man_tec[key.RIGHT] - man_tec[key.LEFT]) * self.VEL_MOV
32             vy += self.GRAVEDAD * dt
33             if self.en_suelo and man_tec[key.SPACE]:
34                 vy = self.VEL_SALTO
35
36             dx = vx * dt
37             dy = vy * dt
38
39             antes = self.get_rect()
40             despues = antes.copy()
41             despues.x += dx
42             despues.y += dy
43
44             self.velocidad = self.man_col(antes, despues, vx, vy)
45             self.en_suelo = (despues.y == antes.y)
46             self.position = despues.center
47             self.cshape.center = Vector2(self.position[0], self.position[1])
48
49         if self.modo == 'liana':
50             if man_tec[key.SPACE] and man_tec[key.RIGHT]:
51                 self.do(JumpBy((185,-20), 80, 1, 1))
52                 self.cshape.center = Vector2(self.position[0], self.position[1])
```

```
                self.image = load('mi_guerrero_2.png')
            elif man_tec[key.SPACE] and man_tec[key.LEFT]:
                self.do(JumpBy((-195,-20), 80, 1, 1))
                self.cshape.center = Vector2(self.position[0], self.position[1])
                self.image = load('mi_guerrero_2.png')

        if self.modo == 'seto':
            despues_subiendo = self.position[1] + 5 * man_tec[key.UP]
            despues_bajando = self.position[1] - 5 * man_tec[key.DOWN]
            if despues_subiendo < 720 and despues_bajando > 200:
                delta_y = 5 * man_tec[key.UP] - 5 * man_tec[key.DOWN]
                self.position += Vector2(0, delta_y)
            if man_tec[key.SPACE] and man_tec[key.LEFT]:
                self.do(JumpBy((-195,-20), 80, 1,1))
                self.cshape.center = Vector2(self.position[0], self.position[1])
                self.image = load('mi_guerrero_2.png')


class MiObjetoInteractuable(Sprite):
    def __init__(self, image, x, y):
        super().__init__(image)
        self.position = (x, y)
        rect = self.get_rect()
        self.cshape = AARectShape(Vector2(0,0), self.width/3, self.height/3)
        self.cshape.center = Vector2(rect.center[0], rect.center[1])

    def update(self, dt):
        pass


class Control(Action):
    def start(self):
        self.mc = CollisionManagerBruteForce()
    def step(self, dt):
        for objeto in self.target.parent.children:
            objeto[1].update(dt)
        self.mc.clear()
        for objeto in self.target.parent.children:
            self.mc.add(objeto[1])

        colisiones_personaje = set(self.mc.iter_colliding(self.target))

        if objetos_colisionables.intersection(colisiones_personaje):
            liana_agarrada = list(lianas & colisiones_personaje)
            if liana_agarrada:
                self.target.GRAVEDAD = 0
                self.target.modo = 'liana'

                self.target.position = (liana_agarrada[0].position[0]-12,
                                        liana_agarrada[0].position[1]-35)
                self.target.image = load('guerrero_agarrando.png')
            if seto in colisiones_personaje:
                self.target.GRAVEDAD = 0
                self.target.modo = 'seto'
                self.target.image = load('guerrero_agarrando.png')
                self.target.position = (seto.position[0]-35,
                                        self.target.position[1])
        else:
            self.target.modo = 'normal'
            self.target.GRAVEDAD = -800


class Escena(Scene):
    def __init__(self):
        global personaje, mapa_colision, man_tec,\
```

```
                objetos_colisionables, lianas, seto
        super().__init__()

        mi_mapa1 = load_tmx('mapa_plataformas_3.tmx')['objetos']
        mi_mapa1_1 = load_tmx('mapa_plataformas_3.tmx')['capa0']

        personaje = MiGuerrero('mi_guerrero_2.png')
        personaje.position = (200,300)
        personaje.do(Control())

        fondo = Sprite('mi_fondo_2.png')
        fondo.position = (640, 384)

        sol = Sun()
        sol.position = (800, 650)

        liana_1 = MiObjetoInteractuable('liana.png', 440, 404)
        liana_2 = MiObjetoInteractuable('liana.png', 640, 404)
        seto = MiObjetoInteractuable('seto.png', 1260, 520)

        objetos_colisionables = {liana_1, liana_2, seto}
        lianas = {liana_1, liana_2}

        capa_fondo = ScrollableLayer()
        capa_fondo.add(fondo)
        capa_fondo.add(sol)

        manejador_scroll = ScrollingManager()
        manejador_scroll.add(mi_mapa1)
        manejador_scroll.add(mi_mapa1_1)

        capa_personaje = ScrollableLayer()
        capa_personaje.add(personaje)
        capa_personaje.add(liana_1)
        capa_personaje.add(liana_2)
        capa_personaje.add(seto)

        mapa_colision = TmxObjectMapCollider()
        mapa_colision.on_bump_handler = mapa_colision.on_bump_slide
        personaje.man_col = make_collision_handler(mapa_colision, mi_mapa1)

        self.add(capa_fondo, z=0)
        self.add(manejador_scroll, z=1)
        self.add(capa_personaje, z=2)
        director.run(self)


if __name__ == '__main__':
    ventana = director.init(width=1280, height=768,
                            caption='Interactuando con objetos')
    ventana.set_location(300,200)
    man_tec = key.KeyStateHandler()
    director.window.push_handlers(man_tec)
    Escena()
```

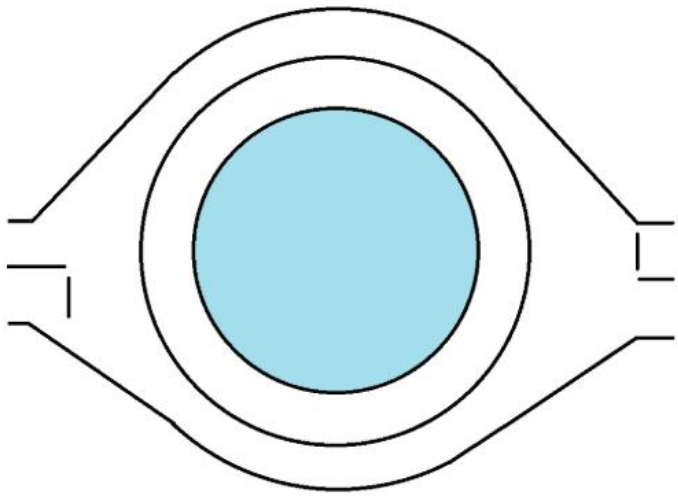
Dos instantáneas de las interacciones guerrero-liana y guerrero-seto son:

El lector puede comprobar el funcionamiento general del programa, detectar imperfecciones e intentar solucionarlas.

5.2 USO DE COCOS2D PARA OTRO TIPO DE APLICACIONES

El libro se ha orientado principalmente al desarrollo de dos tipos muy concretos de videojuegos 2D: arcade "matamarcianos" y de plataformas. El uso de cocos2d nos permite además, como comenté en el Capítulo 1, la creación de presentaciones, demos y aplicaciones gráficas interactivas. De entre los innumerables ejemplos que podríamos poner he elegido el de una sencilla presentación donde se visualiza el paso de un coche por una rotonda y la creación de una aplicación donde podemos interactuar con los elementos de un tablero de ajedrez (agarrando, arrastrando y con la posibilidad de comer fichas) para simular jugar.

Como base para el ejemplo inicial tendremos la imagen (generada, nuevamente de forma artesanal, mediante Paint) **mi_rotonda.png**:

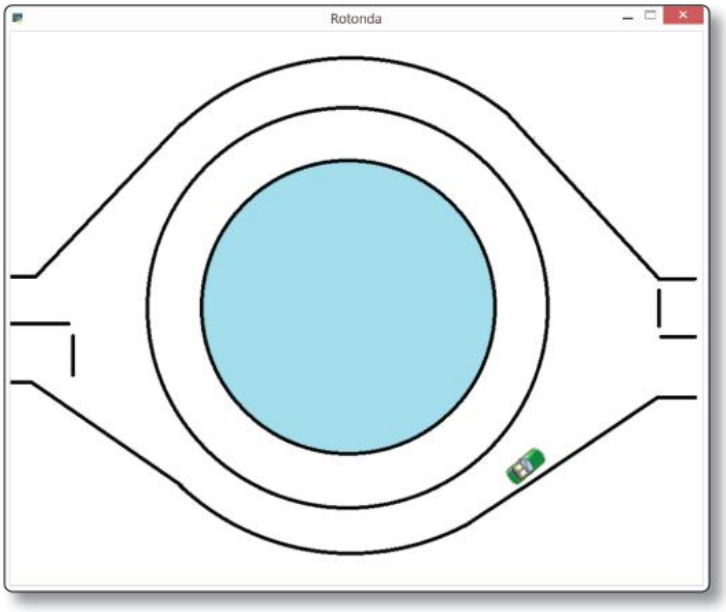


El código estará en **rotonda.py**:

```
1
2 from cocos.director import director
3 from cocos.scene import Scene
4 from cocos.layer import Layer
5 from cocos.sprite import Sprite
6 from cocos.actions import MoveBy, RotateBy, JumpBy, Delay
7
8 class MisAcciones(Layer):
9
10     def __init__(self):
11         super().__init__()
12
13         mi_sprite = Sprite('mi_coche.png')
14         mi_rotonda = Sprite('mi_rotonda.png')
15         mi_sprite.position = 0, 320
```

```
        mi_rotonda.position = 480, 384

        self.add(mi_rotonda)
        self.add(mi_sprite)

        mover1 = MoveBy((60, 0), 1)
        mover2 = MoveBy((120, -100), 2)
        mover3 = MoveBy((100, 45), 1)
        mover4 = MoveBy((100, 0), 1)

        rotar1 = RotateBy(60, 2)
        rotar2 = RotateBy(-120, 5)
        rotar3 = RotateBy(60, 1)

        saltar1 = JumpBy((640,35), -170, 1, 5)

        paso1 = mover1
        paso2 = mover2 |rotar1
        paso3 = saltar1 | rotar2
        paso4 = mover3 | rotar3
        paso5 = mover4

        mi_sprite.do(paso1 + Delay(2) + paso2 + paso3 +  paso4 + paso5 )


if __name__ == "__main__":
    ventana = director.init(width=960, height=768, caption = 'Rotonda')
    ventana.set_location(450,100)
    mi_capa = MisAcciones()
    mi_escena = Scene(mi_capa)
    director.run(mi_escena)
```

Su ejecución presenta a un coche parando en un STOP y circulando posteriormente por una rotonda. Una instantánea se muestra a continuación:



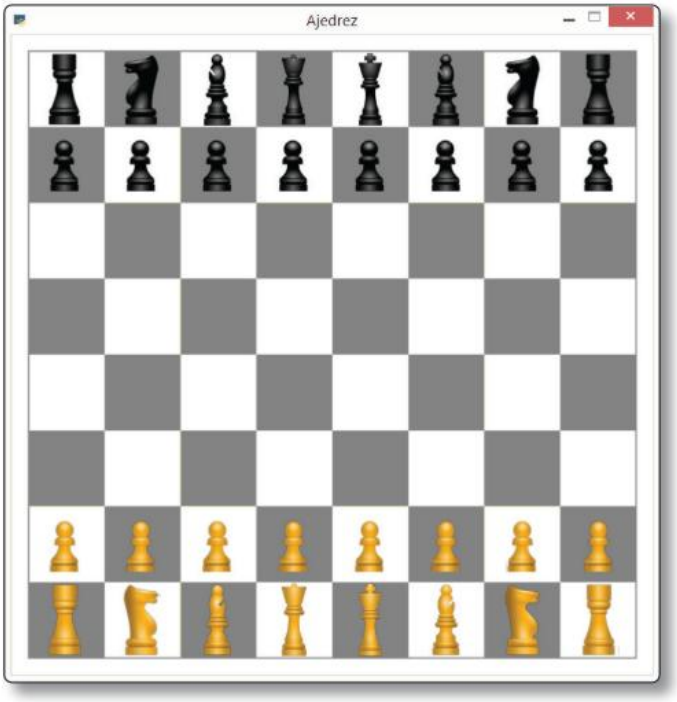
Ahora intentaremos crear un tablero de ajedrez con la posibilidad de mover las figuras, eliminando la que quede en un momento dado solapada por otra. Nos basaremos en las imágenes del tablero y de cada una de las figuras que tenemos en nuestra carpeta. El código está en **ajedrez.py**[112]**:**

```
from cocos.director import director
from cocos.scene import Scene
from cocos.layer import Layer
from cocos.sprite import Sprite

class EventosRaton(Layer):
    is_event_handler = True

    def __init__(self):
        super().__init__()
        self.agarrado = False
        self.total_piezas = []

        self.tablero = Sprite('tablero_ajedrez.png')
        self.tablero.position=(400,400)
        self.add(self.tablero, z = 0)

        linea = 'TCARYACT'
        paso = (director.window.width - 40) / 8
        x = 20 + paso/2
        for e in linea:
            if e == 'T':
                pieza1 = Sprite('torre_b.png')
                pieza2 = Sprite('torre_n.png')
                pieza3 = Sprite('peon_b.png')
                pieza4 = Sprite('peon_n.png')
            elif e == 'C':
                pieza1 = Sprite('caballo_b.png')
                pieza2 = Sprite('caballo_n.png')
                pieza3 = Sprite('peon_b.png')
                pieza4 = Sprite('peon_n.png')
            elif e == 'A':
                pieza1 = Sprite('alfil_b.png')
                pieza2 = Sprite('alfil_n.png')
                pieza3 = Sprite('peon_b.png')
                pieza4 = Sprite('peon_n.png')
            elif e == 'Y':
                pieza1= Sprite('rey_b.png')
                pieza2 = Sprite('rey_n.png')
                pieza3 = Sprite('peon_b.png')
                pieza4 = Sprite('peon_n.png')
            elif e == 'R':
                pieza1 = Sprite('reina_b.png')
                pieza2 = Sprite('reina_n.png')
                pieza3 = Sprite('peon_b.png')
                pieza4 = Sprite('peon_n.png')

            pieza1.position=(x, 70)
            pieza2.position=(x, 70 + (7*paso-4))
            pieza3.position=(x, 70 + (1*paso-4))
            pieza4.position=(x, 70 + (6*paso-4))

            self.add(pieza1, z=1)
            self.add(pieza2, z=1)
```

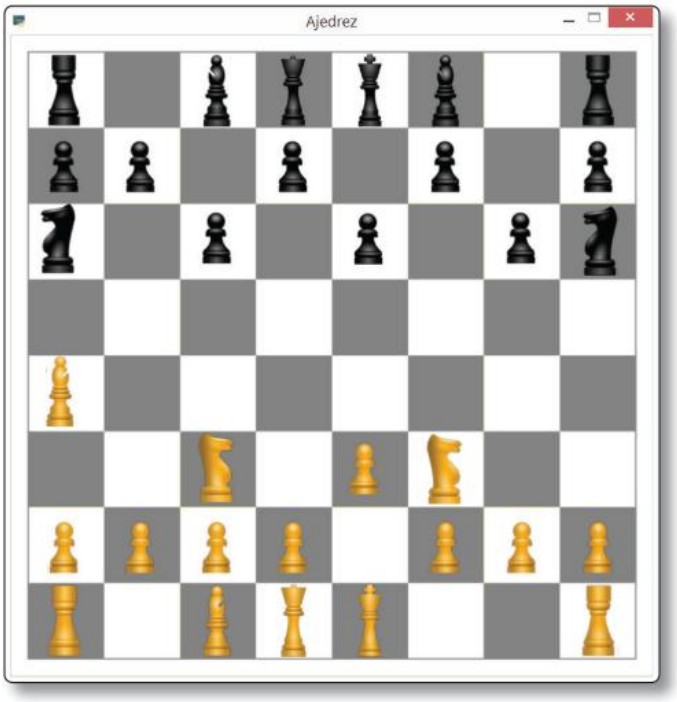
112 No está diseñado para trabajar en pantalla completa.

```
            self.add(pieza3, z=1)
            self.add(pieza4, z=1)

            x += paso

    def on_mouse_drag(self, x, y, dx, dy, buttons, modifiers):
        for elem in self.children:
            if elem[0] == 1 and elem[1].contains(x,y) and not self.agarrado:
                self.agarrado = True
                self.elem_agarrado = elem[1]
        if self.agarrado:
            self.elem_agarrado.position = (x,y)

    def on_mouse_release(self, x, y, buttons, modifiers):
        self.agarrado = False
        for elem in self.children:
            if elem[0] == 1 and elem[1].contains(x,y) and not self.agarrado:
                if elem[1] != self.elem_agarrado:
                    elem[1].kill()


if __name__ == "__main__":
    director.init(width=800, height=800, caption="Ajedrez")
    director.window.set_location(500,100)
    director.run(Scene(EventosRaton()))

```

Su ejecución inicial genera la siguiente salida:



Tras unos pocos movimientos la distribución podría ser:



5.3 CONSIDERACIONES FINALES

Como comenté en el prólogo, para empezar en mundo de la programación de videojuegos es aconsejable en mi opinión hacerlo en 2D, para lo cual la librería cocos2d es ideal si queremos tener a Python como lenguaje. Hemos conocido sus elementos principales (dejando al margen los más sofisticados y complejos[113]), desarrollando videojuegos sencillos. Hay mil características que podríamos añadir a ellos (enemigos que nos siguen o que tienen distintos comportamientos, objetos que realizan movimientos complejos...) y la forma de conseguirlo es, en base a las herramientas que conocemos, programarlas. Si llegado a este punto el lector está deseoso de pulir, mejorar o añadir elementos a las cosas que hemos visto, o crear su propia idea de videojuego, el objetivo fundamental del libro habrá sido satisfecho.

113 Para profundizar más en ellos consultar la documentación oficial de cocos2d.

Puede que en un momento dado queramos hacer cosas más complejas en 2D que incluyan física de (múltiples) objetos sólidos. Su programación directa será una tarea enorme, por lo que mi consejo en ese caso es aprender el uso de **Pymunk**, una librería para tratar la física de cuerpos rígidos en 2D con Python. Está construido sobre la librería **Chipmunk**, escrita en C y Objective-C, que tiene licencia MIT. Su estudio está fuera de los objetivos del libro, aunque su instación sería tan sencilla como abrir una ventana de comandos y teclear:

```
pip3.6 install pymunk
```

En el caso de necesitar programar juegos en 2.5D **pyglet** y **Tiled** (que puede crear mapas con perspectiva isométrica) nos lo permitirían.

Si lo que queremos es trabajar en 3D sobre Python la librería pyglet nos da también esa posibilidad, pero mi consejo sería entonces aprender **BGE** (pendientes de qué ocurre con la versión 2.8 o de forks como **UPBGE**) o **Panda3D**. Incluso trabajar con **Blender** directamente podría ser buena idea, ya que su interacción con Python es, por diseño, muy natural. También tenemos la opción de usar **Godot** o **Unreal Engine** e instalar los plugins que nos permiten el uso de Python sobre ellos, aunque el acceso a sus elementos internos será más limitado.

Espero que el lector haya disfrutado del libro (en el fondo es lo que importa) y le sirva como base e iniciación en el mundo de la programación de videojuegos, un tema apasionante y extremadamente vasto en el que siempre encontraremos nuevos retos.

Apéndice A

INSTALACIÓN DE PYTHON, COCOS2D Y PYSCRIPTER

En este apéndice veremos cómo instalar en nuestro ordenador los siguientes elementos:

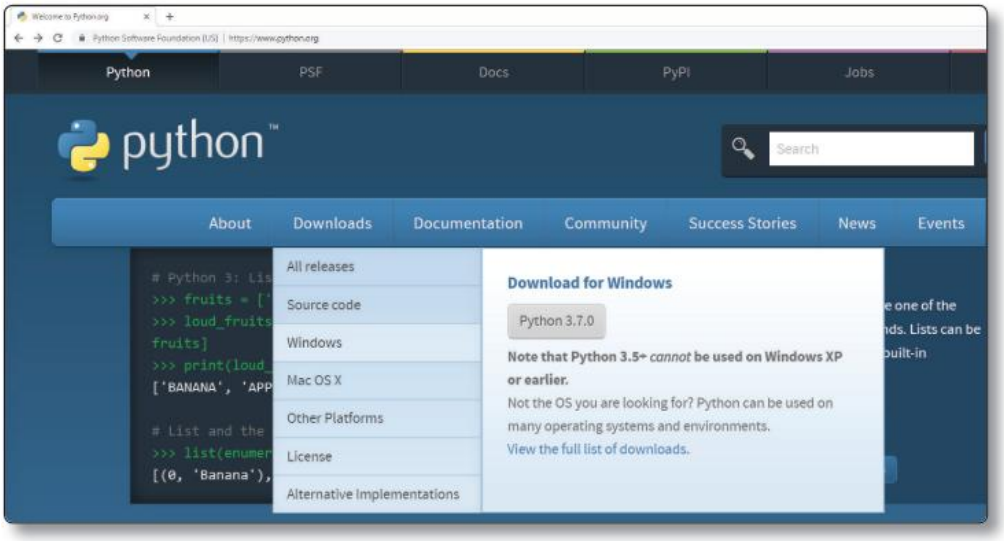
- El intérprete **Python**, en su versión **3.6.6**.
- La librería **cocos2d** (junto a otras librerías necesarias), versión **0.6.5**.
- La versión **3.0.2** del IDE **PyScriter**.

A.1 INSTALAR PYTHON

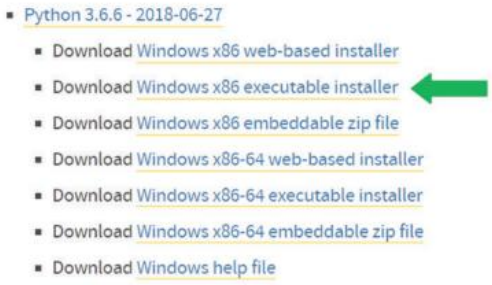
Accederemos mediante nuestro navegador a la siguiente dirección web:

https://www.python.org/

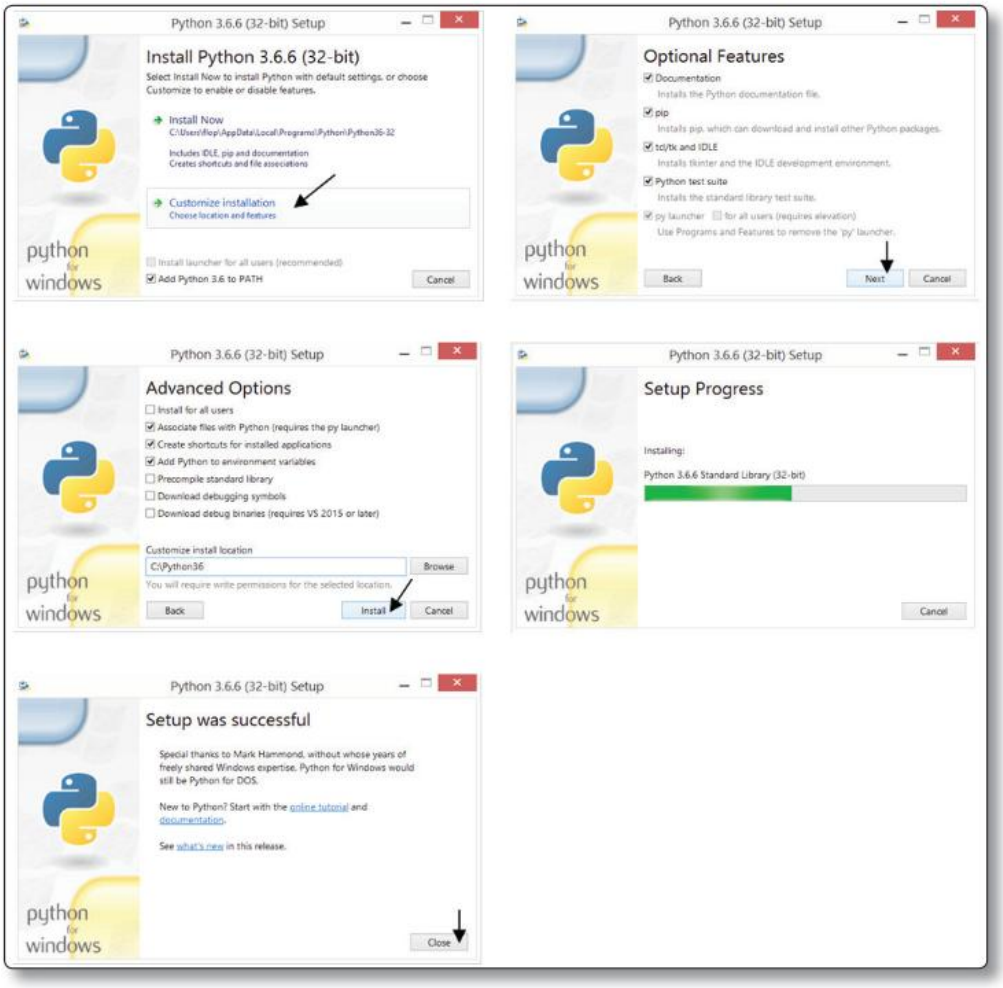
En la sección "Downloads", dedicada a la descarga de las distintas versiones de Python, podremos colocar el cursor del ratón y seleccionar nuestra plataforma:



A pesar de que en Octubre de 2018 ya tenemos disponible Python 3.7.0, el libro se ha realizado enteramente sobre la versión 3.6, por lo que descargaremos su última actualización, la 3.6.6. Para ello hacemos clic el "Windows" y la buscamos:



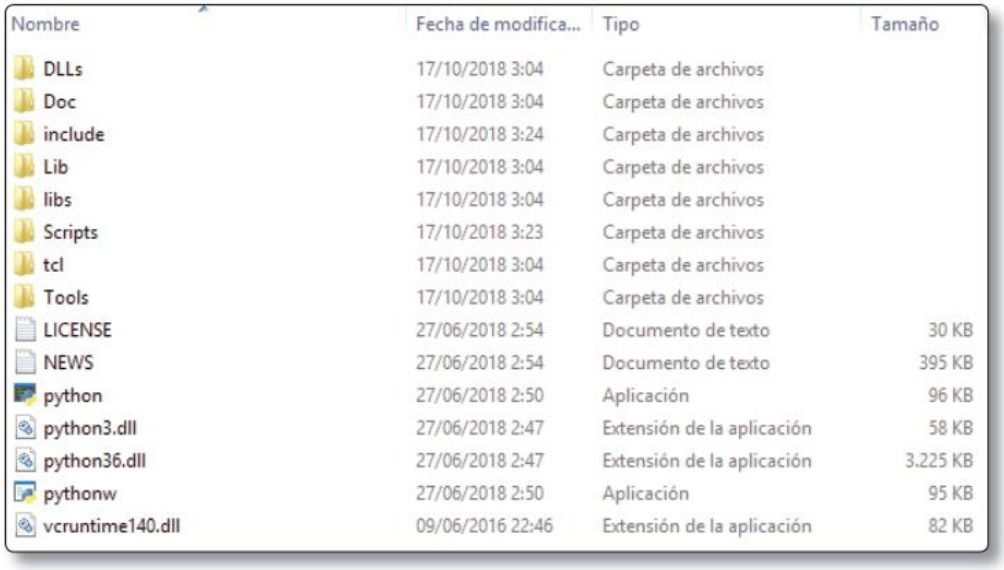
Haremos clic sobre el elemento indicado con una flecha en la imagen superior, tras lo cual se procederá a la descarga de **python-3.6.6.exe**, un fichero que también encontraremos en el material descargable del libro. Lo ejecutamos y comenzará un asistente de instalación donde pulsaremos los botones de los cuadros de diálogo que se indican con flechas tras rellenar las opciones que se muestran si fuese necesario:



En la primera ventana podríamos haber pulsado directamente en “Install Now”, pero he preferido hacerlo de forma personalizada, por dos motivos:

- Para instalar todo en la carpeta C:\Python36, de más fácil acceso y que sigue la línea de versiones anteriores de Python.
- Para conocer de forma exacta qué se instala y qué elementos configura (importante añadir Python a PATH para poder acceder directamente a sus elementos, entre ellos las librerías).

Si todo ha salido bien tendremos el siguiente contenido dentro de C:\Python36:



Nombre	Fecha de modifica...	Tipo	Tamaño
DLLs	17/10/2018 3:04	Carpeta de archivos	
Doc	17/10/2018 3:04	Carpeta de archivos	
include	17/10/2018 3:24	Carpeta de archivos	
Lib	17/10/2018 3:04	Carpeta de archivos	
libs	17/10/2018 3:04	Carpeta de archivos	
Scripts	17/10/2018 3:23	Carpeta de archivos	
tcl	17/10/2018 3:04	Carpeta de archivos	
Tools	17/10/2018 3:04	Carpeta de archivos	
LICENSE	27/06/2018 2:54	Documento de texto	30 KB
NEWS	27/06/2018 2:54	Documento de texto	395 KB
python	27/06/2018 2:50	Aplicación	96 KB
python3.dll	27/06/2018 2:47	Extensión de la aplicación	58 KB
python36.dll	27/06/2018 2:47	Extensión de la aplicación	3.225 KB
pythonw	27/06/2018 2:50	Aplicación	95 KB
vcruntime140.dll	09/06/2016 22:46	Extensión de la aplicación	82 KB

A.2 INSTALAR COCOS2D

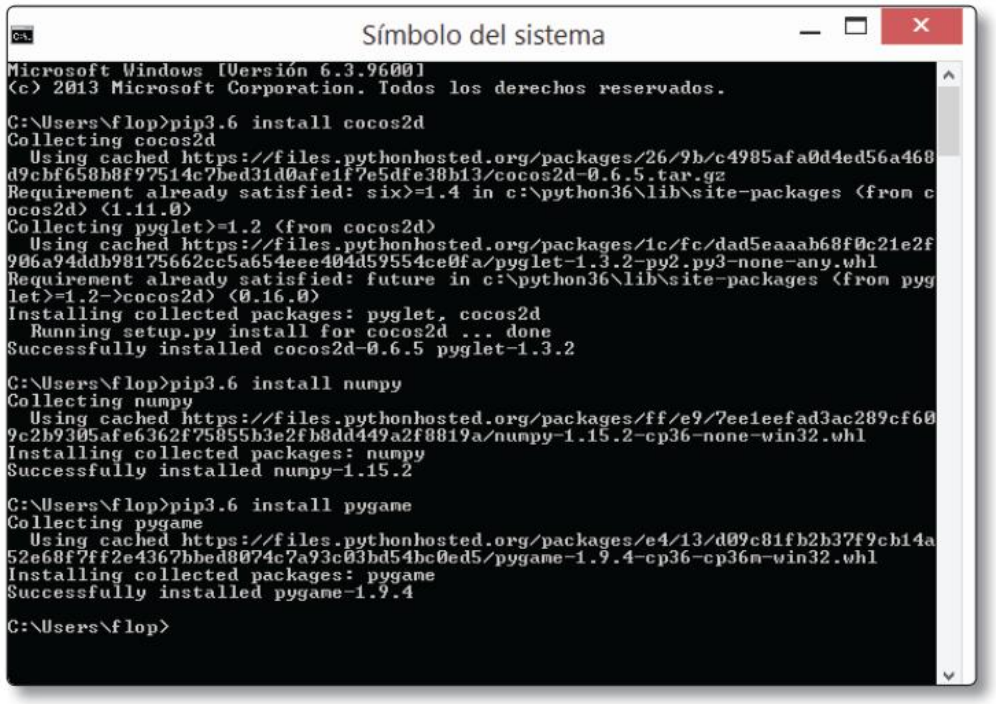
Pasaremos a continuación a instalar las librería cocos2d, además de las dos adicionales que necesitaremos:

- **NumPy**, dará soporte a los sistemas de partículas.
- **pygame**, para el soporte del audio.

Como hemos añadido Python 3.6 al PATH del sistema podremos ejecutar sus elementos desde cualquier carpeta. Abriremos una ventana de comandos e introduciremos uno a uno los tres siguientes:

- pip3.6 install cocos2d
- pip3.6 install numpy
- pip3.6 install pygame

Mostramos el resultado:



Observamos que al instalar inicialmente cocos2d nos incluye **pyglet** en su versión 1.3.2, ya que es una de sus dependencias.

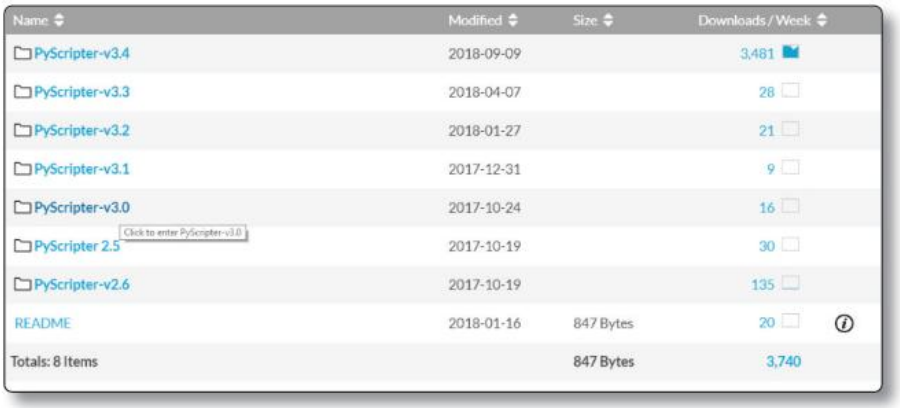
Tendremos ya instaladas las librerías cocos2d, NumPy y pygame en las carpetas correspondientes dentro de C:\Python36. En el material descargable del libro se incluye una copia de ella.

A.3 INSTALAR Y CONFIGURAR PYSCRIPTER

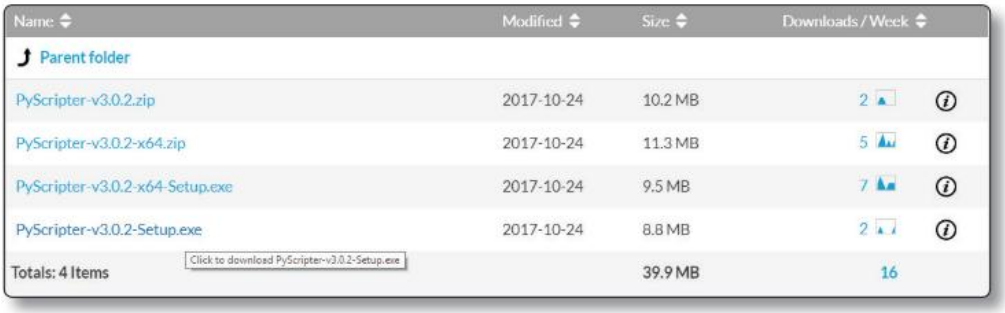
Accedemos desde nuestro navegador a la siguiente dirección:

https://sourceforge.net/projects/pyscripter/files/

Aparecerá lo siguiente:



Haciendo clic sobre la carpeta PyScripter-v3.0 tendremos:

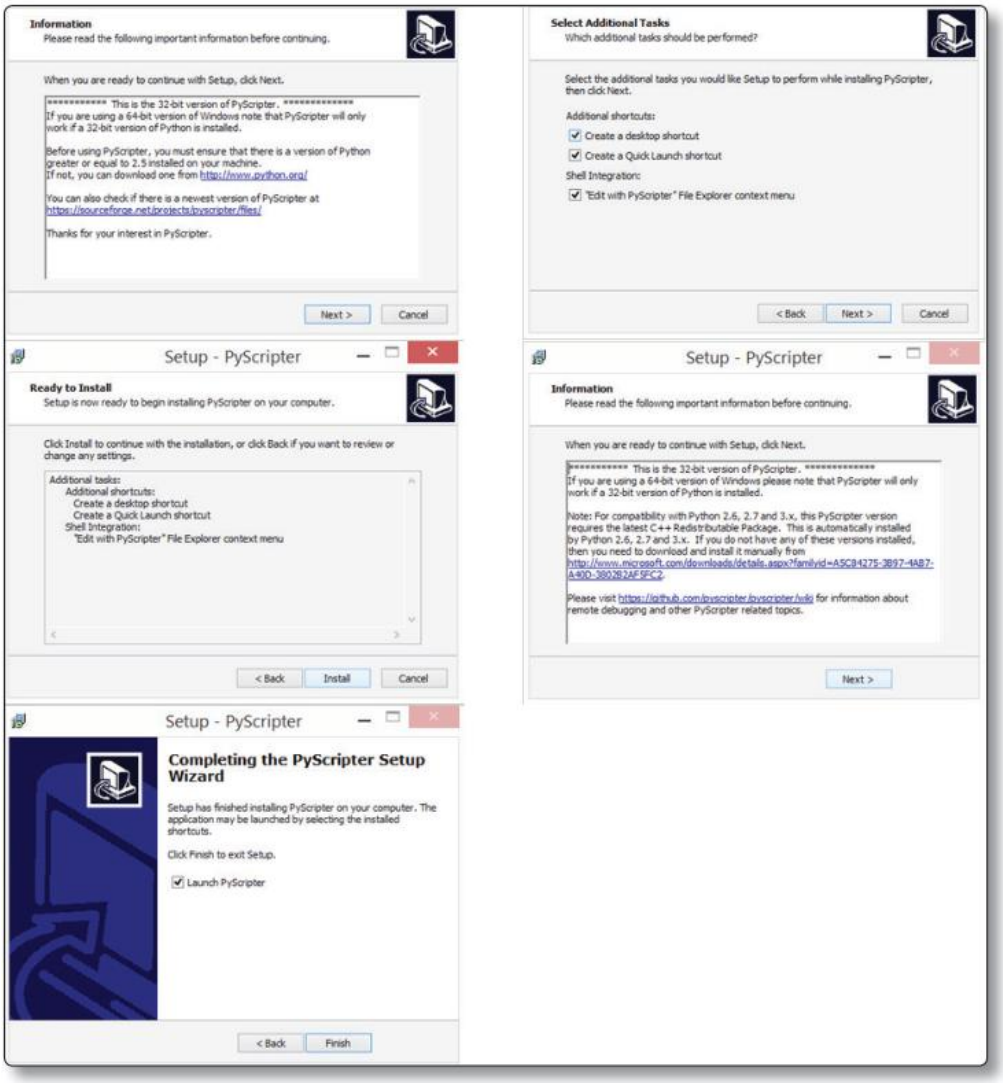


Pulsaremos ahora sobre **PyScripter-v3.0.2-Setup.exe**[114] y, si todo funciona correctamente comenzará a descargar el fichero. Al finalizar, lo abriremos y aparecerá el asistente de instalación. Los pasos a seguir serán simples, limitándonos a hacer clic en Next o Install. En la segunda ventana marcamos las tres opciones, que nos permitirán posteriormente ejecutar el IDE desde un icono en escritorio y que la opción de editar un fichero con PyScripter aparezca en los menús contextuales[115],

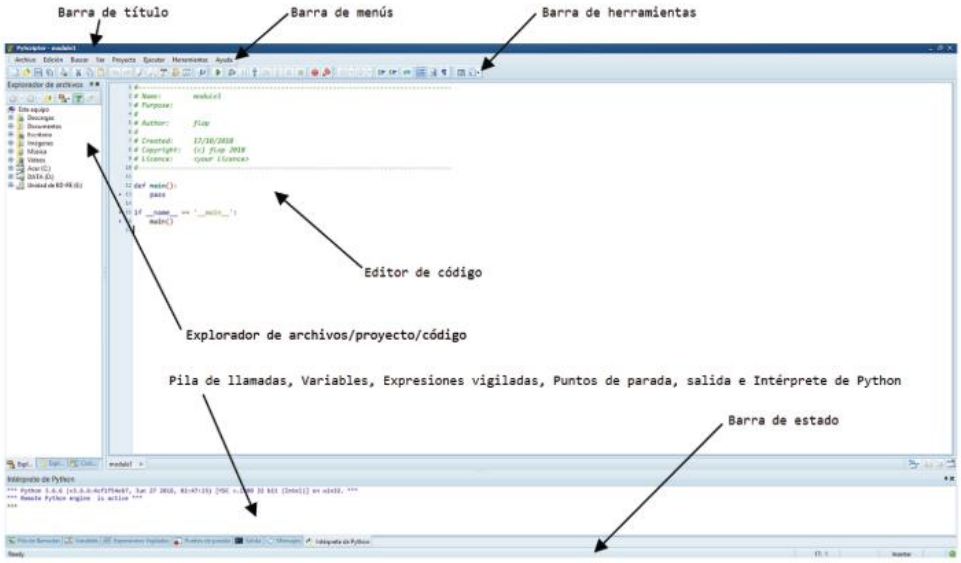
114 Incluido en el material descargable del libro.

115 Un menú contextual es el que cambia dependiendo del contexto. Por ejemplo cuando hacemos clic con el botón derecho del ratón, el menú es distinto dependiendo del elemento concreto al que estemos apuntando.

algo bastante útil en determinadas circunstancias. Las distintas ventanas que recorreremos serán:



Tras hacer clic en Finish con la opción "Launch PyScripter" activada, obtendríamos una pantalla como la siguiente[116], donde señalamos los elementos fundamentales:



Observamos que una serie de ventanas están agrupadas (por ejemplo la que incluye al intérprete de Python, que ya ha configurado automáticamente con nuestra versión) pero podemos distribuirlas a nuestro gusto haciendo clic sobre la pestaña correspondiente y arrastrando hacia la parte de la pantalla que queramos. Al moverla por ella hay determinadas zonas en las que se nos permite anclarlas si soltamos el botón del ratón. También podremos modificar la zona dedicada a una y otra ventana, simplemente colocándonos en el límite entre ellas y, cuando la flecha del ratón cambie a un icono con dos líneas verticales paralelas y dos flechas horizontales, hacer clic y arrastrar. Es interesante jugar con ello sin miedo ya que siempre podremos volver a la configuración de ventanas por defecto mediante el menú Ver→Distribución→Default. El lector podrá distribuirlas como desee.

Hay dos zonas diferenciadas para el intérprete de Python y para el editor integradas en nuestro IDE. Otras, como la Pila de llamadas, Variables, Expresiones Vigiladas, Puntos de parada, Salida y Mensajes están relacionadas de una manera

116 Una vez ejecutado PyScripter podemos anclarlo a la barra de tareas de Windows usando el botón derecho del ratón sobre el icono del programa y seleccionando "Anclar este programa a la barra de tareas".

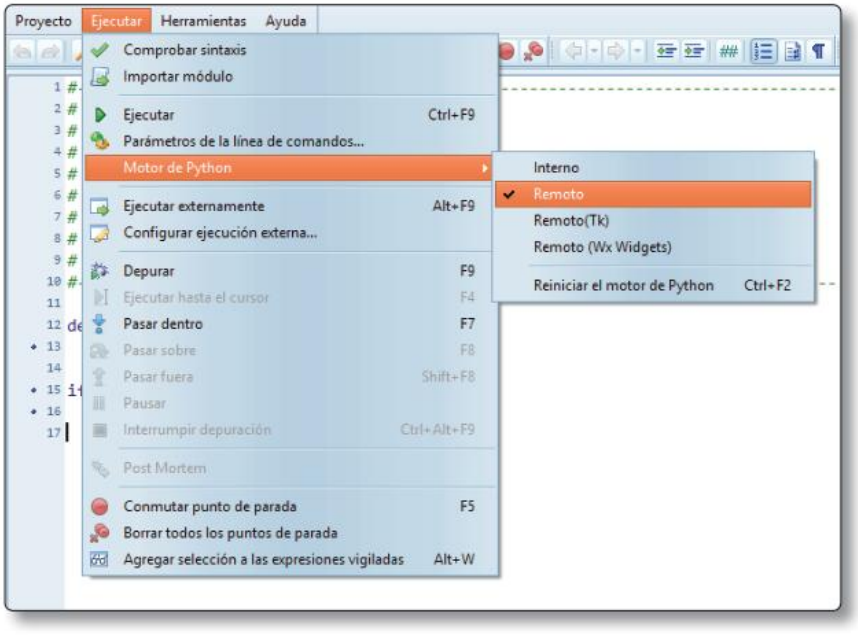
u otra con el depurador, que nos permitirá cosas como ejecutar el programa paso a paso (o hasta un punto determinado), y entrar (o no) en las funciones y métodos, visualizando los elementos que deseemos.

En base a esos tres elementos principales (intérprete, editor y depurador) trabajaremos. Son elementos similares a los encontrados en cualquier IDE, y no entraré a comentar cada uno de ellos de forma minuciosa.

PyScripter puede **ejecutar** scripts de Python mediante la opción "Ejecutar"[117], de dos maneras básicas:

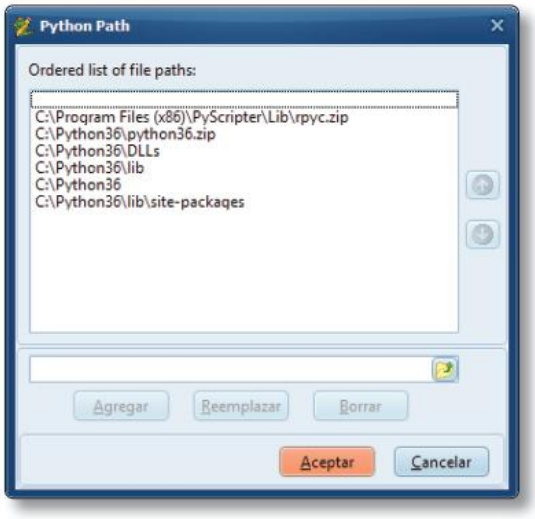
1. Con un intérprete interno (**motor interno**) que viene con PyScripter. Es la opción por defecto, aunque también la podríamos seleccionar en "Motor de Python → interno" en el menú "Ejecutar".
2. Con el intérprete que tengamos en nuestra instalación Python (**motor remoto**), que recordemos hemos guardado en la carpeta C:\Python36. Lo seleccionaremos mediante "Motor de Python → remoto" en el menú "Ejecutar".

Configuraremos por tanto la opción 2:



117 Asociada al botón de ejecutar usado habitualmente en PyScripter o a las teclas Ctrl+F9.

Para comprobar que las carpetas de nuestro intérprete Python 3.6.6 están incluidas en la ruta de búsqueda de PyScripter iremos al menú “ Herramientas” y seleccionaremos “Directorio de Python”, obteniendo:



Ello nos permitirá acceder directamente a las librerías que hemos instalado.

Apéndice B

MÓDULOS DE COCOS2D

En este Apéndice conoceremos los elementos fundamentales de los módulos de cocos2d más importantes para los propósitos del libro. En algunos casos se hará una simple presentación de las clases, métodos y funciones, mientras que en otros se acompañará de una breve explicación de conceptos.

Los módulos de cocos2d son los siguientes[118]:

- cocos package
  - Subpackages
    - cocos.actions package
    - cocos.audio package
    - cocos.layer package
    - cocos.scenes package
  - Submodules
    - cocos.batch module
    - cocos.camera module
    - cocos.cocosnode module
    - cocos.collision_model module
    - cocos.compat module
    - cocos.custom_clocks module
    - cocos.director module
    - cocos.draw module
    - cocos.euclid module
    - cocos.fps module
    - cocos.framegrabber module
    - cocos.gl_framebuffer_object module
    - cocos.grid module
    - cocos.mapcolliders module
    - cocos.menu module
    - cocos.particle module
    - cocos.particle_systems module
    - cocos.path module
    - cocos.rect module
    - cocos.scene module
    - cocos.shader module
    - cocos.skeleton module
    - cocos.sprite module
    - cocos.text module
    - cocos.tiles module
    - cocos.utils module
    - cocos.wired module

118 Sacado directamente de la documentación oficial de cocos2d.

Dentro de cocos.actions veremos los siguientes módulos:

- cocos.actions.base_actions
- cocos.actions.instant_actions
- cocos.actions.interval_actions

De cocos.audio:

- cocos.audio.effect

De cocos.layer:

- cocos.layer.base_layers
- cocos.layer.python_interpreter
- cocos.layer.scrolling
- cocos.layer.util_layers

De cocos.scenes:

- cocos.scenes.pause
- cocos.scenes.sequences
- cocos.scenes.transitions

B.1 MÓDULO COCOS.ACTIONS.BASE_ACTIONS

La clase **Action**, basada en Object, representa la clase más general de la acciones. Su sintaxis es:

```
Action(*args, **kwargs)
```

Tiene los siguientes atributos y métodos principales:

- **Atributos:**
  - **target**
    Objeto CocosNode que es el objetivo de la acción, sobre el que se ejecuta.
- **Métodos:**
  - **done**()
    Nos devolverá False mientras el método step() deba seguir siendo llamado.

- **init**(*args, **kwargs)
  Es llamado por __init__ con todos los parámetros que recibe. En ese momento se desconoce el objetivo de la acción, por lo que su uso típico es almacenar los parámetros necesarios para realizarla.
- **start**()
  Tras configurarse mediante código externo self.target se llama a este método, que nos permite una inicialización extra.
- **step**(dt)
  Es llamado cada frame. El parámetro dt es un número real que indica el tiempo en segundos transcurrido desde la última llamada.
- **stop**()
  Cuando la acción debe cesar su ejecución se llama desde código externo a este método, tras lo cual ningún otro método debe ser llamado.

La clase **IntervalAction**, basada en cocos.actions.base_actions.Action, representa las acciones que tienen una determinada duración fija. Su sintaxis es:

```
IntervalAction(*args, **kwargs)
```

La clase **InstantAction**, basada en cocos.actions.base_actions. IntervalAction, representa las acciones que se realizan de forma instantánea. Su sintaxis es:

```
InstantAction(*args, **kwargs)
```

Comentaremos algo sobre sus siguientes **métodos**:

- **start**()
  Es aquí donde debemos realizar todo el trabajo.
- **step**(dt)
  No hace nada, por lo que no debemos sobrescribirlo.
- **stop**()
  No hace nada, por lo que no debemos sobrescribirlo.
- **update**(t)
  No hace nada, por lo que no debemos sobrescribirlo.

La clase **Repeat**, basada en cocos.actions.base_actions.Action, representa repetición continuada[119] de una acción. Aplicado a InstantAction significa una vez por fotograma. Su sintaxis es:

```
Repeat(*args, **kwargs)
```

Como **método** a destacar comentaremos:

- **init**(action)
  Inicializa el método. El parámetro action es la acción a repetir.

Tendremos las siguientes **funciones** en el módulo:

- **sequence**(action_1, action_2)
  Devuelve una acción (lo más sencilla posible) que ejecuta primero la acción_1 y luego la acción_2.
- **spawn**(action_1, action_2)
  Devuelve una acción (lo más sencilla posible) que ejecuta la acción_1 y la acción_2 en paralelo.
- **loop**(action, times)
  Repite la acción action un número de veces indicado por times.
- **Reverse**(action)
  Invierte el comportamiento de la acción.

B.2 MÓDULO COCOS.ACTIONS.INSTANT_ACTIONS

En este módulo se incluyen las acciones instantáneas, es decir, las que ocurren de forma inmediata, y no tienen una duración determinada como las acciones de intervalo.

La clase **Place**, basada en cocos.actions.base_actions.InstantAction, coloca el objeto CocosNode en una determinada posición de pantalla indicada por sus coordenadas. Su sintaxis es:

```
Place(*args, **kwargs)
```

119 Si quisiésemos la repetición de una acción un número determinado de veces haríamos una multiplicación de la acción por el citado número.

Destacaremos únicamente el siguiente **método**:

- **init**(position)
  Método de inicialización, donde position es una tupla de dos enteros que indican sus coordenadas en pantalla.

La clase **CallFunc**, basada en cocos.actions.base_actions.InstantAction, es una acción que llamará a una determinada función que indiquemos. Su sintaxis es:

```
CallFunc(*args, **kwargs)
```

Destacamos únicamente el siguiente **método**:

- **init**(func, *args, **kwargs)
  Método de inicialización, donde func es la función a la que llamaremos.

La clase **CallFuncS**, basada en cocos.actions.instant_actions.CallFunc, es una acción que llamará a una determinada función que indiquemos, mandándole como primer argumento el objeto sobre el que se realiza la acción. Su sintaxis es:

```
CallFuncS(*args, **kwargs)
```

Seguiremos teniendo el siguiente **método**:

- **init**(func, *args, **kwargs)
  Método de inicialización, donde func es la función a la que llamaremos.

La clase **Hide**, basada en cocos.actions.base_actions.InstantAction, es una acción que oculta el objeto CocosNode. Para volver a mostrarla debemos llamar a la acción show. Su sintaxis es:

```
Hide(*args, **kwargs)
```

La clase **Show**, basada en cocos.actions.base_actions.InstantAction, es una acción que muestra el objeto CocosNode. Para ocultarlo debemos llamar a la acción hide(). Su sintaxis es:

```
Show(*args, **kwargs)
```

La clase **ToggleVisibility**, basada en cocos.actions.base_actions.InstantAction, es una acción que conmuta la visibilidad del objeto CocosNode. Su sintaxis es:

```
ToggleVisibility(*args, **kwargs)
```

B.3 MÓDULO COCOS.ACTIONS.INTERVAL_ACTIONS

En este módulo se incluyen las acciones de intervalo, es decir, las que tienen una duración determinada.

Tendremos las siguientes clases principales:

- **MoveTo**
- **MoveBy**
- **JumpBy**
- **JumpTo**
- **Bezier**
- **Blink**
- **RotateTo**
- **RotateBy**
- **ScaleTo**
- **ScaleBy**
- **FadeOut**
- **FadeIn**
- **FadeTo**
- **Delay**
- **RandomDelay**

Y los modificadores (que también son clases):

- **Accelerate**
- **AccelDeccel**
- **Speed**

La clase **MoveTo**, basada en cocos.actions.base_actions.IntervalAction, mueve el objeto CocosNode a una determinada posición absoluta de pantalla indicada por sus coordenadas. Su sintaxis es:

```
MoveTo(dst_coords, duration=5)
```

En ella dst_coords son las coordenadas absolutas (en forma de tupla de dos enteros) a las que se trasladará el objeto CocosNode, y duration un número real que indica el tiempo en segundos que tardará en realizar el desplazamiento.

La clase **MoveBy**, basada en cocos.actions.interval_actions.MoveTo, mueve el objeto CocosNode una serie de píxeles en el eje x y en el eje y respecto a la posición actual del objeto. Su sintaxis es:

```
MoveBy(delta, duration=5)
```

En ella delta es una de tupla de dos enteros indicando el desplazamiento en cada uno de los ejes, y duration un número real que indica el tiempo en segundos que tardará en realizar el desplazamiento.

La clase **Jump**, basada en cocos.actions.base_actions.IntervalAction, mueve el objeto CocosNode simulando un salto. Su sintaxis es:

```
Jump(y=150, x=120, jumps=1, duration=5)
```

En ella y es un entero que indica la altura del salto, x un entero que marca el desplazamiento horizontal del salto, jumps un entero que fija el número de saltos a realizar, y duration un número real que nos proporciona el tiempo en segundos que tardará en realizar el desplazamiento.

La clase **JumpBy**, basada en cocos.actions.base_actions.IntervalAction, mueve el objeto CocosNode simulando un salto proporcionando desplazamientos relativos respecto a la posición del objeto. Su sintaxis es:

```
JumpBy(position=(0, 0), height=100, jumps=1, duration=5)
```

En ella position es una tupla de dos enteros que indica el desplazamiento horizontal y vertical respecto a la posición del objeto, height un entero que indica la altura del/los salto/s, jumps un entero que fija el número de saltos a realizar y duration un número real que nos proporciona el tiempo en segundos que tardará en realizar el desplazamiento.

La clase **JumpTo**, basada en cocos.actions.interval_actions.JumpBy, mueve el objeto CocosNode simulando un salto. Su sintaxis es:

```
JumpTo(position=(0, 0), height=100, jumps=1, duration=5)
```

En este caso position es una tupla de dos enteros que indica las coordenadas absolutas a las que saltará el objeto, height un entero que indica la altura del/los salto/s, jumps un entero que fija el número de saltos a realizar y duration un número real que nos proporciona el tiempo en segundos que tardará en realizar el desplazamiento.

La clase **Bezier**, basada en cocos.actions.base_actions.IntervalAction, mueve el objeto CocosNode a través de una curva de Bezier. Su sintaxis es:

```
Bezier(bezier, duration=5, forward=True)
```

En ella bezier es una instancia de configuración de la curva de Bezier, duration un número real que nos proporciona el tiempo en segundos que tardará en realizar el desplazamiento y forward un booleano que nos indica si nos desplazamos hacia adelante o no.

La clase **RotateTo**, basada en cocos.actions.base_actions.IntervalAction, rota el objeto CocosNode un cierto ángulo y en una dirección que estará marcada por el ángulo más pequeño. Su sintaxis es:

```
RotateTo(angle, duration)
```

En ella angle es un número real que indica el ángulo en grados y duration un número real que nos proporciona el tiempo en segundos que tardará en realizar la rotación.

La clase **RotateBy** (o **Rotate**), basada en cocos.actions.base_actions.IntervalAction, rota el objeto CocosNode un cierto ángulo en la dirección de las agujas del reloj. Su sintaxis es:

```
RotateBy(angle, duration)
```

En ella angle es un número real que indica el ángulo en grados que rota el objeto (números positivos significan sentido horario) y duration un número real que nos proporciona el tiempo en segundos que tardará en realizar la rotación.

La clase **ScaleTo**, basada en cocos.actions.base_actions.IntervalAction, cambia la escala del objeto CocosNode en un determinado factor. Su sintaxis es:

```
ScaleTo(scale, duration=5)
```

En ella scale es un número real que indica el factor de escalado y duration un número real que nos proporciona el tiempo en segundos que tardará en realizar la operación.

La clase **ScaleBy**, basada en cocos.actions.interval_actions.ScaleTo, cambia la escala del objeto CocosNode en un determinado factor. Su sintaxis es:

```
ScaleBy(scale, duration=5)
```

En ella scale es un número real que indica factor de escalado y duration un número real que nos proporciona el tiempo en segundos que tardará en realizar la operación.

La clase **Delay**, basada en cocos.actions.base_actions.IntervalAction, retrasa la acción un determinado tiempo. Su sintaxis es:

```
Delay(duration)
```

En ella duration es un número real que nos proporciona el tiempo en segundos del retardo.

La clase **RandomDelay**, basada en cocos.actions.interval_actions.Delay, retrasa la acción un determinado tiempo aleatorio entre dos valores dados. Su sintaxis es:

```
RandomDelay(low, hi)
```

En ella low y hi son, respectivamente, los números reales que nos proporcionan el tiempo mínimo y máximo posible (en segundos) del retardo.

La clase **FadeOut**, basada en cocos.actions.base_actions.IntervalAction, hace desvanecer progresivamente al objeto CocosNode en un determinado tiempo. Su sintaxis es:

```
FadeOut(duration)
```

En ella duration es un número real que nos proporciona el tiempo en segundos del desvanecimiento.

La clase **FadeIn**, basada en cocos.actions.interval_actions.FadeOut, hace aparecer gradualmente al objeto CocosNode (hasta una opacidad total) en un determinado tiempo. Su sintaxis es:

```
FadeIn(duration)
```

En ella duration es un número real que nos proporciona el tiempo en segundos que tardará la aparición progresiva.

La clase **FadeTo**, basada en cocos.actions.base_actions.IntervalAction, hace aparecer gradualmente al objeto CocosNode (hasta una opacidad indicada) en un determinado tiempo. Su sintaxis es:

```
FadeTo(alpha, duration)
```

En ella alpha es un entero entre 0 y 255 que nos indica la opacidad[120] y duration es un número real que nos proporciona el tiempo en segundos de la aparición gradual.

La clase **Blink**, basada en cocos.actions.base_actions.IntervalAction, hace parpadear al objeto CocosNode un determinado tiempo. El estado de visibilidad del objeto al finalizar la accción es el mismo que al inicio de ésta. Su sintaxis es:

```
Blink(times, duration)
```

En ella times es un entero que nos indica el número de parpadeos y duration es un número real que nos proporciona el tiempo en segundos que estará parpadeando.

La clase **Accelerate**, basada en cocos.actions.base_actions.IntervalAction, cambia la aceleración de una acción. Su sintaxis es:

```
Accelerate(other, rate=2)
```

En ella other es la acción sobre la que se actuará y rate el índice de aceleración[121].

La clase **AccelDeccel**, basada en cocos.actions.base_actions.IntervalAction, cambia la velocidad de desplazamiento pero la mantiene la velocidad normal al principio y al final del trayecto. Su sintaxis es:

```
AccelDeccel(other)
```

En ella other es la acción sobre la que se actuará.

La clase **Speed**, basada en cocos.actions.base_actions.IntervalAction, cambia la velocidad de una acción. Su sintaxis es:

```
Speed(other, speed)
```

En ella other es la acción sobre la que se actuará y speed un número real que indica el factor multiplicador de la velocidad[122].

120 El valor 0 sería transparente por completo y el 255 opaco total.

121 Por ejemplo el valor 1 sería aceleración lineal.

122 Por ejemplo si es 2 significa el doble de rápido (tardaría la mitad de tiempo). Si es 0.5 significa la mitad de rápido (tardaría la mitad de tiempo). De ser 1 la velocidad no cambia.

B.4 MÓDULO COCOS.AUDIO.EFFECT

La sintaxis de la clase **Effect**, basada en Object, es:

```
Effect(filename)
```

La cadena filename nos indica el fichero de audio que cargaremos.

Tiene los siguientes **atributos**:

- **action**
  Es una acción que ejecuta el fichero de audio.
- **sound**
  Es un objeto de la clase **Sound** (módulo cocos.audio.pygame.mixer) mediante el que podremos configurar, controlar u obtener información del audio cargado.

Y el **método**:

- **play**()
  Reproduce el audio cargado.

El objeto **Sound** almacenado en el atributo sound tiene los siguientes **métodos** interesantes:

- **fadeout**(time)
  El sonido se irá desvaneciendo en el número de milisegundos marcados por el número entero time.
- **get_length**()
  Nos devuelve un número real indicando la longitud temporal del sonido en segundos.
- **get_volume**()
  Nos devuelve un número real entre 0.0 y 1.0 indicando el volumen al que tenemos configurado el sonido.
- **play**(loops, maxtime)
  Reproduce el sonido cargado. Mediante el parámetro loops (un número entero) indicamos el número de repeticiones del sonido tras la primera

reproducción[123], y con maxtime configuramos un tiempo máximo (valor entero dado en milisegundos) tras el cual se detendrá la reproducción de sonido.

- **set_volume**(volume)
  Configura el volumen dando un valor real entre 0.0 y 1.0 al parámetro volume.

- **stop**()
  Detiene la reproducción del sonido.

B.5 MÓDULO COCOS.LAYER.BASE_LAYERS

En este módulo se incluyen las clases Layer y MultiplexLayer (subclase de Layer) que nos permite trabajar con capas. Las capas se usan habitualmente como manejadores de eventos y/o como contenedores de los distintos elementos del juego, ayudando a su organización, tanto a nivel visual como lógico.

El atributo **anchor**[124] se establece de forma predeterminada en el centro de la ventana, que la mayoría de las veces proporciona el comportamiento deseado para la rotación y escala.

La clase **Layer**, basada en cocos.cocosnode.CocosNode[125], nos permitirá escuchar los eventos generados desde director.window. Para conocer sus atributos y métodos consultaremos la clase CocosNode, destacando la presencia adicional del siguiente **atributo**:

- **is_event_handler**
  Booleano que nos indica si los eventos generados en director.window son manejados en la capa. Por defecto no lo son, teniendo el valor False.

La clase **MultiplexLayer**, basada en cocos.layer.base_layers.Layer, nos permite tener una capa compuesta de varias capas, teniendo habilitada solo una de ellas en cada momento[126]. Su sintaxis es la siguiente:

---

123 Si el valor es 0 no se repite ninguna vez, y si es -1 lo hace de forma ininterrumpida.

124 También denominado transform_anchor.

125 Por lo tanto tiene todos sus atributos y métodos. Ver el módulo cocos.cocosnode.

126 Útil por ejemplo si tenemos varios menús pero solo queremos representar uno en cada instante.

```
MultiplexLayer(*layers)
```

En ella *layers es un iterable con las capas que manejaremos. Tras la instanciación la capa habilitada será layers[0].

Destacaremos el siguiente **método**:

- **switch_to**(layer_number)

  Cambiará a otra de las capas manejadas. El parámetro layer_number es un entero (entre 0 y el número de capas menos uno) que nos indicará a qué capa cambiar. En caso de que el número esté fuera del rango se lanzará una excepción. La capa que se esté ejecutando dejará de hacerlo (se llamará además a su método on_exit()) y la nueva se empezará a ejecutar (su método on_enter() será llamado).

B.6 MÓDULO COCOS.LAYER.PYTHON_INTERPRETER

Contendrá la clase **PythonInterpreterLayer**, basada en cocos.layer.util_layers.ColorLayer, que ejecutará un intérprete Python interactivo como una capa hijo de la escena actual. Destacamos únicamente el siguiente **atributo**:

- **cfg** = {'code.font_name': 'Arial', 'code.font_size': 12, 'code.color': (255, 255, 255, 255), 'caret.color': (255, 255, 255)}

  Mediante él configuramos el formato del texto del intérprete.

B.7 MÓDULO COCOS.LAYER.SCROLLING

En este módulo tendremos herramientas para manejar lo que será visible en pantalla cuando el videojuego en su totalidad no quepa en una sola. Daremos soporte a la técnica **parallax**[127], que consiste en dar una falsa sensación de perspectiva usando capas que se deslizan más lentamente cuanto más lejos están.

Tendremos varios elementos fundamentales:

- Las capas implicadas, que serán instancias de **ScrollableLayer**, un tipo de capas que nos permiten realizar scroll sobre ellas.

127 Lo traduciremos como paralaje.

- El manejador, una instancia de la clase **ScrollingManager**, que coordinará las capas sobre las que se puede realizar scroll.
- El **foco**, ligado a los atributos fx y fy de ScrollingManager, nos marca el punto central a partir del que se realiza la vista, y por tanto lo que aparece en pantalla.

Veamos a continuación las dos clases indicadas.

La clase **ScrollableLayer**, basada en cocos.layer.base_layers.Layer[128], tiene la siguiente sintaxis:

```
ScrollableLayer(parallax=1)
```

El parámetro parallax es el paralaje de la capa. Su valor por defecto es 1.

Si en una instancia ScrollableLayer se define el atributo **px_width** también se debe definir **px_height**; el scroll se limitará a mostrar sólo las áreas con coordenadas x e y que cumplan:

origin_x <= x < = px_width
origin_y <= y <= px_height

Los valores de los atributos **origin_x** y **origin_y** toman valor 0 por defecto.

Si px_width no está definido, entonces la capa no limitará el scroll.

Además de los atributos indicados destacaremos de la clase ScrollableLayer los siguientes **métodos**:

- **draw**()
  Se dibuja a sí mismo.
- **on_cocos_resize**(usable_width, usable_height)
  Manejador para el evento redimensionar la ventana. Los dos parámetros son los valores usables de ancho y alto de la ventana.
- **on_enter**()
  Se llama cada vez que la capa entra en escena.
- **on_exit**()
  Se llama cada vez que la capa sale de escena.

128 A su vez deriva de cocos.cocosnode.CocosNode. Ver cocos.layer.base_layers.Layer.

- **set_dirty**()
  La vista ha cambiado de alguna manera.

- **set_view**(x, y, w, h, viewport_ox=0, viewport_oy=0)
  Configura la posición de la vista para la capa. Con x e y indicamos el punto de la vista, con w y h la anchura y altura de ella, y con viewport_ox y viewport_oy el origen de la vista. Todos los parámetros son números reales.

La clase **ScrollingManager**, basada en cocos.layer.base_layers.Layer[129], maneja el scroll para sus hijos, que son instancias de ScrollableLayer. Por lo tanto lo denominaremos manejador de scroll.

Su sintaxis es:

```
ScrollingManager(viewport=None, do_not_scale=None)
```

Restringe el scroll para que se respeten todas las restricciones de visibilidad impuestas por las instancias ScrollableLayer; al menos una de ellas debe definir una restricción para que el scroll sea limitado. La representación puede limitarse al rectángulo de una ventana específica mediante el parámetro viewport. El manejador de scroll también proporciona cambios de coordenadas entre las coordenadas "screen" (relativas a la ventana) y las "world" (absolutas respecto al manejador).

El parámetro (opcional) viewport es un rectángulo (instancia de Rect) que define la vista, y do_not_scale un booleano que indica si el manejador de scroll debe escalar la vista durante los cambios de tamaño de la ventana[130]. Tiene los siguientes atributos y métodos principales:

- **Atributos**:
  - **scale**
    Factor de escalado del objeto.

- **Métodos**:
  - **add**(child, z=0, name=None)
    Añade el objeto hijo child (de tipo CocosNode, generalmente una instancia de ScrollableLayer) y actualiza la vista y el foco del manejador de scroll. El parámetro z (número entero) indica el nivel (eje z) en el que se inserta el hijo, mientras que name (una cadena) nos proporciona opcionalmente su nombre.

---

129 A su vez deriva de cocos.cocosnode.CocosNode. Ver cocos.layer.base_layers.Layer.

130 Por defecto su valor es None, lo que significa que tiene el mismo valor que director.autoscale.

- **force_focus**(fx, fy)
  Fuerza al manejador de scroll a focalizar en el punto de coordenadas (fx, fy), independientemente de los límites visibles de cualquier capa que gestione. Los parámetros fx y fy son de tipo entero.
- **on_cocos_resize**(usable_width, usable_height)
  Manejador para el evento redimensionar la ventana. Los dos parámetros son los valores usables de ancho y alto de la ventana.
- **on_enter**()
  Se llama cada vez que la capa entra en escena.
- **on_exit**()
  Se llama cada vez que la capa sale de escena.
- **pixel_from_screen**(x, y)
  Desactualizado, renombrado como screen_to_world.
- **pixel_to_screen**(x, y)
  Desactualizado, renombrado como world_to_screen.
- **refresh_focus**()
  Restablece el enfoque en el punto de enfoque.
- **screen_to_world**(x, y)
  Transforma las coordenadas "screen" en las coordenadas "world", que nos son devueltas en forma de tupla de enteros. Importante para las transformaciones de vista, capa y pantalla.
- **set_focus**(fx, fy, force=False)
  Hace que el punto de coordenadas (fx, fy) se muestre lo más cerca posible del centro de la vista. Cambia los objetos hijo manejados para que el punto (fx, fy) en las coordenadas "world" se vea tan cerca del centro de la vista como sea posible, mientras que al mismo tiempo no muestra áreas fuera de los límites de los objetos hijo. Los parámetros fx y fy son de tipo entero, mientras que force es un booleano (por defecto de valor False) que, si es True, fuerza la actualización del enfoque aunque el punto de enfoque o la escala no hayan cambiado.
- **update_view_size**()
  Actualiza el tamaño de la vista basándose en el ancho y alto utilizable por el director, y en la opcional vista (viewport).
- **visit**()
  Se dibuja a sí mismo y a sus hijos en la vista (viewport). Opera igual que CocosNode.visit() pero limitándose al rectángulo de la vista.

- **world_to_screen**(x, y)
  Transforma las coordenadas "world" en las coordenadas "screen", que nos son devueltas en forma de tupla de enteros. Importante para las transformaciones de vista, capa y pantalla.

B.8 MÓDULO COCOS.LAYER.UTIL_LAYERS

En este módulo se incluye únicamente la clase **ColorLayer**, que nos permitirá crear una capa con fondo sólido de color. Está basada en cocos.layer.base_layers.Layer[131] y tiene la siguiente sintaxis:

```
ColorLayer(r, g, b, a, width=None, height=None)
```

Los **parámetros** son los siguientes:

- **r**
  Entero entre 0 y 255 que indica la cantidad de color rojo que tiene nuestro color.
- **g**
  Entero entre 0 y 255 que indica la cantidad de color verde que tiene nuestro color.
- **b**
  Entero entre 0 y 255 que indica la cantidad de color azul que tiene nuestro color.
- **a**
  Entero entre 0 y 255 que indica el valor de alpha[132] que tiene nuestro color.
- **width**
  Entero que nos indica la anchura en píxeles de la capa de color. Opcional.
- **height**
  Entero que nos indica la altura en píxeles de la capa de color. Opcional.

131 Por lo tanto hereda de ella sus atributos y métodos.

132 Nos indica el nivel de opacidad. Cuanto más alto es el número, mayor la opacidad.

Si no indicamos los parámetros opcionales width y height se creará una capa de tamaño 640x480.

Disponemos de los siguientes **atributos** adicionales a los que ya teníamos en la clase cocos.layer.base_layers.Layer:

- **color**

  Tupla de tres enteros entre 0 y 255 que nos indica en formato RGB[133] el color de la capa.

- **opacity**

  Entero entre 0 y 255 que nos indica el nivel de opacidad del color de la capa. El valor 0 es totalmente transparente y 255 totalmente opaco.

B.9 MÓDULO COCOS.SCENES.PAUSE

Contiene las siguientes dos clases:

- **PauseLayer**, basada en cocos.layer.base_layers.Layer, que es una capa que representa el texto 'PAUSED'.
- **PauseScene**, basada en cocos.scene.Scene, una escena en la que se realiza una pausa.

B.10 MÓDULO COCOS.SCENES.SEQUENCES

Contiene la clase **SequenceScene**, basada en cocos.scene.Scene, que es una escena usada para reproducir una serie de escenas, una detrás de otra. Su sintaxis es:

```
SequenceScene(*scenes)
```

El parámetro *scenes es una lista con las escenas a reproducir, en el orden indicado. El método on_enter() de cada escena no será llamado hasta que ésta esté activa. Usaremos el método pop() del director para avanzar hacia la siguiente escena. Cuando se llegue a la última y se reproduzca se eliminará la instancia SequenceScene de la pila de escenas.

133 Red, Green, Blue.

B.11 MÓDULO COCOS.SCENES.TRANSITIONS

En este módulo tendremos las clases que nos permitirán tratar con las transiciones entre escenas. Cuando hablemos de alejar o acercar la escena nos referiremos a hacer zoom hacia fuera o hacia dentro de ella.

La clase base es **TransitionScene**, cuya sintaxis es:

```
TransitionScene(dst, duration=1.25, src=None)
```

Basada en cocos.scene.Scene, toma dos escenas y hace una transición entre ellas. Con dst y src indicamos (respectivamente) la escena final e inicial, y duration nos marca el tiempo en segundos que dura la transición. Tiene los siguientes **atributos**:

- **duration** = None
  Duración en segundos de la transición
- **in_scene** = None
  Indicamos la escena que sustituirá a la anterior.
- **out_scene** = None
  Indicamos la escena que será reemplazada

Y destacamos los **métodos** que se indican:

- **finish**()
  Se llama cuando el tiempo de transición termina. En ese momento la escena que ejecuta el director será dst.
- **hide_all**()
  Oculta ambas escenas: la entrante y la saliente.
- **hide_out_show_in**()
  Oculta la escena saliente y muestra la escena entrante.
- **start**()
  Añade la escena entrante con z=1 y la escena saliente con z=0

Se presentan a continuación las sintaxis de las restantes clases del módulo, junto a una breve descripción de lo que hacen:

```
RotoZoomTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, gira y aleja la escena saliente y, a continuación, gira y acerca la escena entrante.

```
JumpZoomTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, aleja y realiza saltos en la escena saliente, y luego realiza saltos y amplía la escena entrante.

```
MoveInLTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, se mueve desde la izquierda la escena entrante.

```
MoveInRTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.MoveInLTransition, se mueve desde la derecha la escena entrante.

```
MoveInBTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.MoveInLTransition, se mueve desde abajo la escena entrante.

```
MoveInTTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.MoveInLTransition, se mueve desde arriba la escena entrante.

```
SlideInLTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, desliza la escena entrante desde el borde izquierdo.

```
SlideInRTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.SlideInLTransition, desliza la escena entrante desde el borde derecho.

```
SlideInBTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.SlideInLTransition, desliza la escena entrante desde el borde inferior.

```
SlideInTTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.SlideInLTransition, desliza la escena entrante desde el borde superior.

```
FlipX3DTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, voltea la pantalla horizontalmente, siendo la cara frontal la escena saliente y la cara posterior la escena entrante.

```
FlipY3DTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, voltea la pantalla verticalmente, siendo la cara frontal la escena saliente y la cara posterior la escena entrante.

```
FlipAngular3DTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, voltea la pantalla la mitad horizontalmente y la mitad verticalmente, siendo la cara frontal la escena saliente y la cara posterior la escena entrante.

```
ShuffleTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, divide en forma de mosaico y desordena la escena saliente, y luego reordena los azulejos con la escena entrante.

```
TurnOffTilesTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, va desactivando cada una de los azulejos de la escena saliente en orden aleatorio.

```
FadeTRTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, va desvaneciendo los azulejos[134] de la escena saliente desde la esquina inferior izquierda hasta la esquina superior derecha.

```
FadeBLTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.FadeTRTransition, va desvaneciendo los azulejos de la escena saliente desde la esquina superior derecha hasta la esquina inferior izquierda.

---

134 O baldosas, término que he usado habitualmente en el libro.

```
FadeUpTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.FadeTRTransition, va desvaneciendo los azulejos de la escena saliente de abajo hacia arriba.

```
FadeDownTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.FadeTRTransition, va desvaneciendo los azulejos de la escena saliente de arriba hacia abajo.

```
ShrinkGrowTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, va reduciendo la escena saliente mientras crece la escena entrante.

```
CornerMoveTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, mueve la esquina inferior derecha de la escena entrante a la esquina superior izquierda.

```
EnvelopeTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, desde la escena saliente mueve la esquina superior derecha y la esquina inferior izquierda hacia el centro y desde la escena entrante realiza la acción inversa de la escena saliente.

```
SplitRowsTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.SplitColsTransition, divide la pantalla en filas. Las filas impares van a la izquierda y las pares a la derecha.

```
SplitColsTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, divide la pantalla en columnas. Las columnas impares van hacia arriba y las pares hacia abajo.

```
FadeTransition(*args, **kwargs)
```

Basada en cocos.scenes.transitions.TransitionScene, va desvaneciendo la escena saliente y apareciendo progresivamente de forma más nítida la escena entrante. Opcionalmente se suministra un color RGB para que se desvanezca en el medio.

```
ZoomTransition(*args, **kwargs)
```

Basado en cocos.scenes.transitions.TransitionScene, se amplía y desvanece la escena saliente.

B.12 MÓDULO COCOS.COCOSNODE

La clase **CocosNode**, basada en Object, tiene los siguientes atributos y métodos principales:

- **Atributos**:
  - **actions**, lista de objetos Action que se están ejecutando. El valor inicial es None.
  - **anchor**, coordenadas en forma de tupla de números enteros (relativas a position) del punto de anclaje usado para transformaciones como rotaciones o escalados. Los objetos hijo serán añadidos en este punto. También se le denomina transform_anchor.
  - **anchor_x**, coordenada en eje x del atributo anchor. También se le denomina transform_anchor_x.
  - **anchor_y**, coordenada en eje y del atributo anchor. También se le denomina transform_anchor_y.
  - **children**, lista de tuplas (a,b) donde a es un entero que nos indica el z-order y b la referencia del objeto hijo. Se ordena de menor a mayor z-order (desde el fondo hacia el frente).
  - **children_names**, diccionario que mapea los nombres de los objetos hijo con su referencia.
  - **is_running**, booleano que indica si el objeto se está ejecutando.
  - **parent**, indica el padre del objeto.
  - **position**, tupla de enteros (x,y) que nos indica las coordenadas del objeto.
  - **rotation**, número real que nos indica la rotación del objeto en grados. Valor inicial 0.0.
  - **scale**, valor real que indica el factor de escala del objeto.
  - **scale_x**, valor real que indica el factor de escala del objeto en el eje x.
  - **scale_y**, valor real que indica el factor de escala del objeto en el eje y.
  - **scheduled_calls**, lista de callbacks programadas.
  - **scheduled_interval_calls**, lista de los intervalos de las callbacks programadas. Su valor por defecto es None.
  - **skip_frame**, booleano que indica si el próximo frame se saltará o no.

- **to_remove**, lista de objetos Action que serán eliminados.
- **transform_anchor**, ver atributo anchor.
- **transform_anchor_x**, ver atributo anchor_x.
- **transform_anchor_y**, ver atributo anchor_y.
- **visible**, booleano que indica si el objeto (y todos sus hijos) es visible o no. Su valor por defecto es True.
- **x**, entero que indica la coordenada x del objeto.
- **y**, entero que indica la coordenada y del objeto.

Métodos:

- **add**(child, z=0, name=None)
  Añade un hijo child (objeto CocosNode) y, si forma parte de la escena activa, se llama a su método on_enter(). El parámetro z es el nivel en el que se inserta el hijo y name su nombre.
- **are_actions_running**()
  Nos devuelve booleano indicando si hay acciones ejecutándose.
- **do**(action, target=None)
  Ejecuta la acción action aplicada a target (que por defecto es el propio objeto) y nos devuelve un clon de la acción. Cuando la acción finaliza será eliminada del contenedor de acciones. Si quisiésemos eliminar una acción usaremos el valor de retorno de do() y se lo pasaremos al método remove_action().
- **draw**(*args, **kwargs)
  Es el método que debemos sobrescribir si queremos que una subclase de CocosNode dibuje algo en la pantalla, respetando los atributos de posición, escala, rotación y ancla.
- **get**(name)
  Nos devuelve el objeto hijo de nombre name, pasado como cadena. De no existir se lanzaría una excepción.
- **get_ancestor**(klass)
  Recorre ascendentemente el árbol de nodos hasta que encuentra uno de la clase Klass, que nos devuelve. En caso de no encontrarlo devuelve None.
- **get_children**()
  Nos devuelve una lista con los hijos del objeto, ordenados desde el fondo al frente.

- **kill**()
  Elimina el objeto de su objeto padre, y muy probablemente sea eliminado de todos sitios.
- **on_enter**()
  Método llamado justo después de que el nodo entre en la escena activa. Las llamadas y acciones programadas se iniciarán o continuarán.
- **on_exit**()
  Método llamado justo después de que el nodo salga de la escena activa. Las llamadas y acciones programadas se suspenderán.
- **pause**()
  Suspende la ejecución de acciones.
- **pause_scheduler**()
  Suspende las llamadas y acciones programadas.
- **remove**(obj)
  Elimina un objeto hijo dando su nombre (en forma de cadena) o referencia. Si el hijo fue añadido por su nombre es conveniente que lo eliminemos también por su nombre ya que de lo contrario intentar añadir un objeto con ese nombre lanzará una excepción.
  Si el objeto fuese parte de la escena activa, se llama a su método on_exit().
- **remove_action**(action)
  Elimina la acción action del contenedor de acciones del objeto, potencialmente llamando al método action.stop().
- **resume**()
  Reanuda la ejecución de acciones.
- **resume_scheduler**()
  Reanuda las llamadas y acciones programadas.
- **schedule**(callback, *args, **kwargs)
  Programa la llamada a la función callback en cada frame. Ésta debe incluir un parámetro de entrada dt que recoge el tiempo transcurrido (en segundos) desde el último tic del reloj. Los argumentos adicionales que indiquemos serán pasados a la función.
  Es un wrapper a pygle.clock.schedule, con el añadido de que todas las funciones son reanudadas o suspendidas si el objeto entra o sale de la escena.

- **schedule_interval**(callback, interval, *args, **kwargs)
  Programa la llamada a la función callback en cada intervalo de tiempo (en segundos) interval. Si el valor de éste es 0 no se vuelve a llamar a la función.
  Es un wrapper a pygle.clock.schedule_interval. El formato por lo demás es igual que en el método schedule().
- **stop**()
  Elimina todos los elementos de la lista de acciones que están ejecutándose, llamando previamente al método stop() de cada acción.
- **unschedule**(callback)
  Elimina la función callback del programador de llamadas. Si no apareciese en él no se lanzaría una excepción. Si la función tuviese varias apariciones en el programador, todas ellas se eliminarían. Es un wrapper a pygle.clock.unschedule.
- **visit**()
  Visita los objetos hijo de forma recursiva. Primero visita los que tienen un nivel z menor que 0, después llama a su método draw(), y finalmente visita a los que tienen nivel z igual o mayor que 0. Antes de visitar cualquier hijo llama al método transform() para aplicar posibles transformaciones.
- **walk**(callback, collect=None)
  Ejecuta la función callback en todos los elementos del árbol del objeto, devolviendo una lista de todos los valores de retorno que no son None. La función debe tomar como argumento un objeto CocosNode, y collect hace referencia a la lista.

B.13 MÓDULO COCOS.COLLISION_MODEL

El módulo dispone de las siguientes clases:

- **Cshape**
- **AARectShape**
- **CircleShape**
- **CollisionManager**
- **CollisionManagerBruteForce**
- **CollisionManagerGrid**

Y de las siguientes funciones[135]:

- **aa_rect_distance_aa_rect**()
- **aa_rect_distance_circle**()
- **aa_rect_overlaps_aa_rect**()
- **aa_rect_overlaps_circle**()
- **circle_distance_circle**()
- **circle_overlaps_circle**()
- **clamp**()

La clase **Cshape**, basada en Object, representa una forma geométrica abstracta en 2D y nos puede dar información sobre la proximidad o intersección con otras formas. Implementaciones particulares pueden restringir la forma geométrica que se acepta. Tiene los siguientes **métodos** principales:

- **copy**()

  Devuelve una copia de sí mismo.

- **distance**(other)

  Devuelve un número real que es la distancia desde la forma a otra forma. No es necesariamente una distancia euclídea, sino distancia entre delimitaciones.

- **fits_in_box**(packed_box)

  Devuelve un booleano que será True si la forma cabe totalmente en el rectángulo de lados paralelos a los ejes definido por packed_box, y False en caso contrario. En packed_box pasamos una tupla de 4 números reales (x_min, x_max, y_min, y_max) que indican los valores mínimos y máximos de las coordenadas de los lados del rectángulo.

- **minmax**()

  Devuelve el menor rectángulo (de lados paralelos a los ejes) que contiene todos los puntos de la forma geométrica. El rectángulo se devuelve en forma de tupla de 4 números reales (x_min, x_max, y_min, y_max) que marcan los valores mínimos y máximos de las coordenadas de los lados del rectángulo.

135 No representamos los posibles parámetros de entrada. Se verán más adelante.

- **near_than**(other, near_distance)

  Devuelve un booleano que será True si la distancia entre la forma propia y otra forma other es menor o igual que el valor near_distance, y False en caso contrario.

- **overlaps**(other)

  Devuelve un booleano que será True si la forma propia se superpone a otra forma other, y False en caso contrario.

- **touches_point**(x,y)

  Devuelve un booleano que será True si el punto (x,y) se superpone a la forma propia, y False en caso contrario.

La clase **AARectShape** implementa la interfaz[136] de Cshape usando rectángulos (con sus lados paralelos a los ejes) como forma geométrica. Lo leeríamos como "Axis Aligned Rectangle Shape". Es aconsejable si los actores no rotan.

Su sintaxis es la siguiente:

```
AARectShape(center, half_width, half_height)
```

En ella center es una tupla con las coordenadas del centro del rectángulo, y half_width y half_height la mitad de su anchura y su altura, respectivamente.

La clase **CircleShape** implementa la interfaz de Cshape[137] usando círculos como forma geométrica. La distancia es la distancia euclídea. Es aconsejable si los actores rotan.

Su sintaxis es la siguiente:

```
CircleShape(center, r)
```

En ella center es una tupla con las coordenadas del centro del círculo, y r su radio.

La clase **CollisionManager**, basada en Object, nos permitirá implementar el manejador de colisiones, y responder preguntas sobre proximidad o colisión entre objetos conocidos por él. Podríamos querer saber qué objetos están a menos de una determinada distancia o colisionan con uno dado.

---

136 Por lo tanto, tiene los métodos comentados en Cshape.

137 Por lo tanto, tiene los métodos comentados en Cshape.

En el momento de la instanciación, o al llamar a su método clear(), el manejador no conoce ningún objeto. Para que lo haga tendremos que añadirlo mediante el método add().

Puede haber varios manejadores de colisión en un mismo ámbito, y un mismo objeto puede ser conocido por varios de ellos al mismo tiempo.

Los objetos, para poder ser conocidos por el manejador de colisiones, o presentados ante él, de cara a posteriores consultas, deben tener las siguientes características:

- Tener un atributo llamado cshape.
- Que el atributo cshape soporte la interfaz Cshape.

Cumpliéndose esos requisitos el objeto es "colisionable". Está permitido que los objetos colisionables sean de cualquiera de las clases AARectShape o CircleShape, aunque a menudo son sólo de una de ellas.

La manipulación de los distintos objetos conocidos en un manejador de colisión se realiza mediante los siguientes métodos:

- **clear**(), que elimina todos los objetos conocidos del manejador y vacía las estructuras de datos que los contienen.
- **add**(obj), que añade el objeto obj a los objetos conocidos.
- **remove_tricky**(obj), que elimina obj de los objetos conocidos.

Internamente las estructuras de datos del manejador se basan en el valor del atributo cshape del objeto a añadir, por lo que deberemos actualizarlo (cuando éste cambia) antes de añadir el objeto, algo que es responsabilidad del programador.

Los **métodos** de CollisionManager son:

- **add**(obj)
  Añade el objeto obj como conocido.
- **any_near**(obj, near_distance)
  Devuelve None si ningún objeto conocido (salvo él mismo) está más cerca que la distancia near_distance, o un objeto que sí lo esté. El objeto obj no tiene por qué ser un objeto conocido.

- **clear**()

  Borra el conjunto de objetos conocidos.

- **iter_all_collisions**()

  Nos devuelve iterador que identifica todas las colisiones entre objetos conocidos. En cada iteración nos devolverá una tupla (obj1, obj2) donde obj1 y obj2 son los objetos que colisionan. Si aparece (obj1, obj2) no aparecerá (obj2, obj1) ya que ya ha indicado su colisión.

- **iter_colliding**(obj)

  Nos devuelve iterador sobre los objetos que colisionan con el objeto obj, donde obj no tiene por qué ser un objeto conocido.

- **known_objs**()

  Devuelve conjunto con todos los objetos conocidos por el manejador de colisión.

- **knows**(obj)

  Nos devuelve booleando indicando si el objeto obj es conocido por el manejador de colisiones.

- **objs_colliding**(obj)

  Nos devuelve un contenedor con los objetos conocidos que se superponen al objeto obj (sin incluirse él, que no debe ser obligatoriamente un objeto conocido).

- **objs_into_box**(minx, maxx, miny, maxy)

  Devuelve un contenedor con los objetos conocidos que caben por completo en el rectángulo alineado con los ejes determinado por los parámetros minx, maxx, miny y maxy.

- **objs_near**(obj, near_distance)

  Devuelve un contenedor con los objetos conocidos que están a una distancia menor o igual a near_distance del objeto obj (algo que incluye por supuesto a los que colisionan con él), sin incluir el propio obj (no requiriéndose que éste sea un objeto conocido).

- **objs_near_wdistance**(obj, near_distance)

  Devuelve una lista de tuplas (objeto_conocido, distancia) que cumplen que la distancia de objeto_conocido a obj es menor o igual que near_distance. Eso incluye los que colisionan con él. No se requiere que obj sea un objeto conocido. Si necesitásemos que esa lista estuviese ordenada

ascendentemente por las distancias usaríamos el método ranked_objs_near().

- **objs_touching_point**(x, y)
  Devuelve un contenedor con los objetos conocidos que tocan el punto (x,y).

- **ranked_objs_near**(obj, near_distance)
  Igual que objs_near_wdistance pero ordenando ascendentemente la lista teniendo como referencia las distancias.

- **remove_tricky**(obj)
  Hace que el manejador de colisiones olvide al objeto obj. Como nota importante recordar que obj debe tener el mismo valor del atributo cshape que cuando el manejador añadió obj mediante el método add().

- **they_collide**(obj1, obj2)
  Devuelve booleano indicando si los objetos obj1 y obj2 se superponen. Ninguno de ellos es obligatorio que sean objetos conocidos.

La clase **CollisionManagerBruteForce**, basada en Object, implementa la interfaz CollisionManager[138] con el menor código posible. Su rendimiento es bajo, por lo que se usa generalmente en casos sencillos, como referencia o para depuración.

La clase **CollisionManagerGrid**, basada en Object, implementa la interfaz CollisionManager[139] basada en el esquema denominado "spatial hashing"[140], que se fundamenta en dividir el plano en rectángulos de un determinado ancho y alto, teniendo una tabla que indique qué objetos se superponen a cada rectángulo. Posteriormente, cuando hagamos una consulta referente al objeto obj, sólo examinaremos los objetos que superponen los rectángulos superpuestos por obj, o los que están a una cierta distancia.

Su sintaxis es la siguiente:

```
CollisionManagerGrid(xmin, xmax, ymin, ymax, cell_width, cell_height)
```

138 Por lo tanto, tiene los métodos comentados en CollisionManager.

139 Por lo tanto, tiene los métodos comentados en CollisionManager.

140 Podríamos traducirlo como "cálculo espacial".

Los cuatro primeros parámetros nos delimitan el plano en el que será efectivo el manejador de colisiones, y los dos siguientes nos marcan la anchura y altura de los rectángulos (o celdas) en que dividimos ese plano.

La función **aa_rect_distance_aa_rect**() nos devuelve la distancia entre dos rectángulos de lados alineados con los ejes[141]. Los rectángulos deben tener los atributos center (para el centro), rx (la mitad del ancho) y ry (la mitad del alto).

Tiene la siguiente sintaxis:

```
aa_rect_distance_aa_rect(aa_rect, other)
```

Los parámetros aa_rect y other son los dos objetos rectángulo.

La función **aa_rect_distance_circle**() nos devuelve la distancia entre un rectángulo de lados alineados con los ejes y un círculo. El rectángulo debe tener los atributos center (para el centro), rx (la mitad del ancho) y ry (la mitad del alto) mientras que el círculo debe tener los atributos center (para el centro) y r (para el radio). Tiene la siguiente sintaxis:

```
aa_rect_distance_circle(aa_rect, circle)
```

En ella aa_rect es el rectángulo y circle es el círculo.

La función **circle_distance_circle**() nos devuelve la distancia entre dos círculos, que deben tener los atributos center (para el centro) y r (para el radio).

Tiene la siguiente sintaxis:

```
circle_distance_circle(circle, other)
```

Los parámetros circle y other son los dos círculos.

La función **aa_rect_overlaps_aa_rect**() nos devuelve booleano indicando si dos rectángulos de lados alineados con los ejes se solapan. Los rectángulos deben tener los atributos center (para el centro), rx (la mitad del ancho) y ry (la mitad del alto).

Tiene la siguiente sintaxis:

```
aa_rect_overlaps_aa_rect(aa_rect, other)
```

En ella aa_rect y other son los dos rectángulos.

141 "aa_rect" lo leeríamos como "axis aligned rectangle".

La función **aa_rect_overlaps_circle**() nos devuelve un booleano indicando si un rectángulo de lados alineados con los ejes y un círculo se solapan. Los rectángulos deben tener los atributos center (para el centro), rx (la mitad del ancho) y ry (la mitad del alto) mientras que el círculo debe tener los atributos center (para el centro) y r (para el radio).

Tiene la siguiente sintaxis:

```
aa_rect_overlaps_circle(aa_rect, circle)
```

En ellal aa_rect es el rectángulo y circle el círculo.

La función **circle_overlaps_circle**() nos devuelve un booleano indicando si dos círculos se solapan. Los círculos deben tener los atributos center (para el centro) y r (para el radio).

Tiene la siguiente sintaxis:

```
circle_overlaps_circle(circle, other)
```

Los parámetros circle y other son los dos círculos.

La función **clamp**() es usada para el cálculo de distancias entre los objetos. Tiene el siguiente formato:

```
clamp(value, minimum, maximum)
```

Nos devolverá el resultado de la siguiente expresión:

max(min(value, maximum), minimum)

B.14 MÓDULO COCOS.DIRECTOR

La clase **Director**, basada en pyglet.event.EventDispatcher, tiene los siguientes atributos y métodos principales:

- **Atributos**:
  - **window**, la ventana principal que maneja el director. Es una instancia de la clase pyglet.window.Window, que por su importancia será comentada en este módulo.
  - **scene**, la escena activa en ese momento.
  - **scene_stack**, la pila de escenas.

- **event_types**, lista que almacena los tipos de evento para el director. Su valor es [‘on_push’, ‘on_pop’, ‘on_resize’, ‘on_cocos_resize’].
- **show_FPS**, booleano que indica si se está representando por pantalla el número de frames por segundo.
- **show_interpreter**, indica si se está representando por pantalla el intérprete interactivo de Python.
- **return_value**, almacena el valor de finalización devuelto por la última escena que fue terminada al llamar al método end().

◤ **Métodos**:

- **get_virtual_coordinates**(x, y)
  Transforma las coordenadas (x,y) pertenecientes al tamaño real de la ventana a las correspondientes al tamaño virtual de la ventana, devolviéndolas en forma de tupla.
- **get_window_size**()
  Devuelve en forma de tupla el tamaño de la ventana cuando fue creada, no el tamaño actual de la ventana.
- **init**(*args, **kwargs)
  Inicializa el director, creando la ventana principal. La mayoría de parámetros corresponden a los de pyglet.window.Window.__init__. Destacamos los siguientes:
  - autoscale, booleano que indica si se realiza autoescalado del contenido de la ventana. Por defecto el valor es False.
  - audio_backend, cadena que nos indica el backend de audio. Podemos elegir entre ‘pyglet’ (valor por defecto) o ‘sdl’.
  - fullscreen, booleano que nos indica si la ventana ocupa toda la pantalla o no. Por defecto su valor es False.
  - resizable, booleano que nos indica si podemos o no redimensionar el tamaño de la ventana. Por defecto su valor es False.
  - width, entero que nos indica el tamaño en puntos del ancho de la ventana. Su valor por defecto es 640.
  - height, entero que nos indica el tamaño en puntos del alto de la ventana. Su valor por defecto es 480.
  - caption, cadena que nos indica el título de la ventana.
  - visible, booleano que nos indica si la ventana es o no visible. Por defecto tiene valor True.

- **on_draw**()
  Maneja el evento 'on_draw' ("al dibujar") de pyglet.window.Window.
- **pop**()
  Saca una escena de la pila de escenas, reemplazando la escena actual. Si la pila estuviese vacía termina la aplicación. Internamente pop() lanza el evento 'on_pop' que es manejado por el método on_pop().
- **push**(scene)
  Ejecuta la escena scene, colocando la escena actual en la pila de escenas suspendidas. Internamente push(scene) lanza el evento 'on_push' junto al parámetro scene, que es manejado por el método on_push().
- **replace**(scene)
  Reemplaza la escena actual (finalizándola) con la escena scene.
- **run**(scene)
  Ejecuta la escena scene, para a continuación entrar en el bucle principal del director.
- **scaled_resize_window**(width, height)
  Se llama cuando tenemos la opción de escalado en la ventana principal. Si redimensionamos la ventana los nuevos valores de anchura y altura son pasados como parámetros width y height, respectivamente.
- **set_show_FPS**(value)
  Activa/desactiva la representación de los fotogramas por segundo que tenemos en ese momento.
- **unscaled_resize_window**(width, height)
  Se llama cuando no tenemos la opción de escalado en la ventana principal. Si redimensionamos la ventana los nuevos valores de anchura y altura son pasados como parámetros width y height, respectivamente.

La clase **Window** del módulo **pyglet.window** será la base para la ventana de nuestra aplicación, por lo que merece que comentemos su sintaxis, atributos y métodos:

Sintaxis:

```
Window(*kwargs)
```

Los parámetros nombrados que podemos pasarle coinciden con los **atributos**, entre los cuales destacamos:

- **width**=None

  Entero que nos indica el tamaño en puntos del ancho de la ventana. Su valor por defecto es 640.

- **height**=None

  Entero que nos indica el tamaño en puntos del alto de la ventana. Su valor por defecto es 480.

- **caption**=None

  Cadena que nos indica el título de la ventana.

- **resizable**=False

  Booleano que nos indica si podemos o no redimensionar el tamaño de la ventana.

- **style**=None

  Nos indica el estilo de la ventana.

- **fullscreen**=False

  Booleano que nos indica si la ventana ocupa o no toda la pantalla.

- **visible**=True

  Booleano que nos indica si la ventana es o no visible

Los **métodos** más interesantes son:

- **clear**()

  Borra la ventana.

- **close**()

  Cierra la ventana.

- **get_location**()

  Nos devuelve tupla de enteros con la posición actual de la ventana (su esquina superior izquierda), en píxeles y tomando como referencia la esquina superior izquierda de la pantalla.

- **get_size**()

  Nos devuelve tupla de enteros con el tamaño actual de la ventana, que no incluye su barra de título.

- **maximize**()

  Maximiza el tamaño de la ventana.

- **minimize**()

  Minimiza el tamaño de la ventana.

- **set_caption**(caption)

  Configura el título de la ventana con la cadena caption.

- **set_fullscreen**(*kwargs)

  Configura la pantalla completa.

- **set_icon**(*images)

  Configura mediante la lista de imágenes images el icono (o iconos) de la ventana.

- **set_location**(x,y)

  Configura la posición de la ventana. Con x e y indicamos las coordenadas en píxeles de la esquina superior izquierda de la ventana respecto de la esquina superior izquierda de la pantalla.

- **set_maximum_size**(width, height)

  Configura el máximo tamaño de la ventana. Con width y height indicamos, respectivamente, la anchura y altura (en píxeles).

- **set_minimum_size**(width, height)

  Configura el mínimo tamaño de la ventana. Con width y height indicamos, respectivamente, la anchura y altura (en píxeles).

- **set_mouse_visible**(visible=True)

  Configura mediante un booleano la visibilidad del cursor del ratón.

- **set_size**(width, height)

  Configura el tamaño de la ventana. Con width y height indicamos, respectivamente, la anchura y altura (en píxeles).

- **set_visible**(visible=True)

  Configura mediante un booleano la visibilidad de la ventana.

B.15 MÓDULO COCOS.DRAW

Es un módulo con elementos relacionados con la posibilidad de dibujar en pantalla. Contiene cuatro clases:

- **Canvas**, basada en cocos.cocosnode.CocosNode.
- **Context**, basada en object.
- **Line**, basada en cocos.draw.Canvas.
- **Segment**, basada en object.

Entre ellas destacamos la clase **Line**, con la cual podremos dibujar líneas en pantalla. Tiene la siguiente sintaxis:

```
Line(start, end, color, stroke_width=1)
```

Los parámetros de entrada coinciden con **atributos**:

- **start**
  Tupla que indica el punto inicial de la línea.
- **end**
  Tupla que indica el punto final de la línea.
- **color**
  Tupla RGBA que indica el color de la línea.
- **stroke_width**
  Número real que indica la anchura de la línea.

Un sencillo ejemplo de su uso lo tenemos en **ejemplo_linea.py**.

B.16 MÓDULO COCOS.EUCLID

Es un módulo donde tendremos clases para tratar con vectores, matrices y cuaterniones en geometría euclídea, tanto en 2D como en 3D. En el libro hemos hecho uso únicamente de la clase **Vector2**, que nos permite trabajar con vectores 2D. Tiene la siguiente sintaxis:

```
Vector2(x=0, y=0)
```

En ella x e y son las coordenadas del vector.

Para más información consultar la documentación oficial de cocos2d.

B.17 MÓDULO COCOS.MAPCOLLIDERS

El módulo contiene tres clases y una función destinados a manejar las colisiones entre el actor y un contenedor de objetos.

Las clases son:

- **RectMapCollider**
  Para manejar colisiones entre actor y elementos de una capa RectMapLayer.
- **RectMapWithPropsCollider**
  Para manejar colisiones entre actor y elementos de una capa RectMapLayer que tengan unas determinadas propiedades.
- **TmxObjectMapCollider**
  Para manejar colisiones entre actor y elementos de una capa TmxObjectLayer.

Y la función[142]:

- **make_collision_handler**()
  Crea un manejador de colisión indicándole una capa y un modelo de colisión.

La clase **RectMapCollider**, basada en Object, nos da la posibilidad de manejar colisiones entre un actor y los objetos de una capa de tipo RectMapLayer. Su sintaxis es la siguiente:

```
RectMapCollider(velocity_on_bump=None)
```

El parámetro velocity_on_bump es una cadena que puede tener uno de los siguientes valores: 'bounce', 'stick' o 'slide'. Nos indica el comportamiento del actor cuando existe una colisión, como veremos a continuación.

142 No representamos los posibles parámetros de entrada. Se verán más adelante.

Tendremos los siguientes **atributos** en la clase:

- **on_bump_handler**
  Indicará el método usado para cambiar la velocidad del actor cuando existe una colisión.
- **bumped_x**
  Booleano que indica si se detecta colisión en el eje x.
- **bumped_y**
  Booleano que indica si se detecta colisión en el eje y.

Y los **métodos**:

- **collide_bottom**(obj)
  Se llama cuando existe colisión de la parte inferior del actor con el objeto obj.
- **collide_left**(obj)
  Se llama cuando existe colisión de la parte izquierda del actor con el objeto obj.
- **collide_map**(maplayer, last, new, vx, vy) --> (vx, vy)
  Limita el movimiento del actor (last-->new) considerando los posibles obstáculos. Los parámetros de entrada son:
  - maplayer, capa de tipo RectMapLayer que contiene los objetos con los que potencialmente colisionar.
  - last, rectángulo (objeto Rect) que indica la posición del actor en el instante last.
  - new, rectángulo (objeto Rect) que indica la potencial posición del actor en el instante new.
  - vx, número real que indica la velocidad en el eje x usada para calcular last-->new.
  - vy, número real que indica la velocidad en el eje x usada para calcular last-->new.

  Se nos devolverá una tupla de enteros (vx,vy) con los valores (modificados o no) de velocidad.
- **collide_right**(obj)
  Se llama cuando existe colisión de la parte derecha del actor con el objeto obj.
- **collide_top**(obj)
  Se llama cuando existe colisión de la parte superior del actor con el objeto obj.

- **detect_collision**(obj, last, new)

  Nos devuelve la mínima corrección en cada eje para que el actor no colisione con el objeto obj.

  Los parámetros son:

  - obj, el potencial objeto colisionador con nuestro actor.
  - last, rectángulo (objeto Rect) que indica la posición del actor en el instante last.
  - new, rectángulo (objeto Rect) que indica la posición del actor en el instante new.

  Puede ser sobrescrito para afinar en la consideración de existencia de colisión.

- **on_bump_bounce**(vx, vy)

  Hace que el actor rebote al colisionar con un obstáculo. Los parámetros de entrada son los mismos que en on_bump_handler().

- **on_bump_handler**(vx, vy) --> (vx, vy)

  Indicaremos el método que controla la velocidad del actor cuando hay una colisión (on_bump_bounce, on_bump_stick u on_bump_slide) o un código personalizado.

  Los parámetros de entrada son:

  - vx, número real que indica la velocidad en el eje x antes de la colisión.
  - vy, número real que indica la velocidad en el eje y antes de la colisión.

  Nos devolverá la tupla (vx, vy), donde:

  - vx, número real que indica la velocidad en el eje x después de la colisión.
  - vy, número real que indica la velocidad en el eje y después de la colisión.

- **on_bump_slide**(vx, vy)

  Bloquea el movimiento sólo en una determinada dirección de un eje. Los parámetros de entrada son los mismos que en on_bump_handler().

- **on_bump_stick**(vx, vy)

  Detiene cualquier movimiento tras la colisión. Los parámetros de entrada son los mismos que en on_bump_handler().

- **resolve_collision**(obj, new, dx_correction, dy_correction)

Corrije new para evitar que el actor colisione con el objeto obj.

Los parámetros son:

- obj, el potencial objeto colisionador con nuestro actor.
- new, rectángulo (objeto Rect) que indica la potencial posición del actor en el instante new, antes de considerar la colisión con obj.
- dx_correction, número entero que indica la mínima corrección en el eje x para que no haya colisión con obj.
- dy_correction, número entero que indica la mínima corrección en el eje y para que no haya colisión con obj.

Si existe una colisión a lo largo de los ejes x e/o y, se colocarán a True las banderas bumped_x y/o bumped_y, respectivamente. También se llamaría a collide_left, collide_right, collide_top o collide_bottom dependiendo del caso.

La clase **RectMapWithPropsCollider**, basada en RectMapCollider, nos permite manejar colisiones entre un actor y los objetos de una capa de tipo RectMapLayer que tengan la propiedad 'left', 'right', 'top' o 'bottom', que indicará el lado del objeto que se tendrá en cuenta para una posible colisión. Por lo tanto nos marcan qué lado es sólido. Tiene todas las características de RectMapCollider (consultar sus atributos y métodos) teniendo en cuenta la particularidad comentada.

La clase **TmxObjectMapCollider**, basada en RectMapCollider, nos permite manejar colisiones entre un actor y los objetos de una capa de tipo TmxObjectLayer, que son instancias de la clase TmxObject y que representan formas geométricas.

Para conocer sus atributos y métodos consultar la clase RectMapCollider.

La única función de la que disponemos en el módulo es **make_collision_handler**(), que tiene la siguiente sintaxis:

```
make_collision_handler(collider, maplayer)
```

Sus **parámetros** de entrada son los siguientes:

- **collider**

  Indica el modelo de colisión, pudiendo ser RectMapCollider, RectMapWithPropsCollider o TmxObjectMapCollider.

- **maplayer**

  Indica la capa donde estarán los objetos potencialmente colisionables con nuestro actor.

Se nos devolverá:

- función f(last, new, vx, vy) -> (vx, vy)

  Corresponde con el método collide_map() de la clase indicada por collider al que le pasamos el argumento adicional indicado por maplayer. Tenemos por tanto un modelo de colisión aplicado a una capa mediante el que podremos controlar el comportamiento de nuestro actor considerando las potenciales colisiones entre él y los objetos de la capa.

Usaremos esta función para crear un manejador de colisión personalizado. En muchos casos en forma de atributo del actor, para poder acceder a él de forma sencilla.

B.18 MÓDULO COCOS.MENU

Módulo para trabajar con menús. Tendremos una clase para el propio menú (Menu) y seis para los posibles tipos de ítem que puede contener (MenuItem, MultipleMenuItem, ToggleMenuItem, EntryMenuItem, ImageMenuItem y ColorMenuItem).

La clase **Menu**, basada en cocos.layer.base_layers.Layer, es la base para las capas de menú. Su sintaxis es:

```
Menu(title='')
```

Con el parámetro title indicamos el título que tendrá el menú.

Destacaremos únicamente el siguiente método:

- **create_menu**(items, selected_effect=None, unselected_effect=None, activated_effect=None, layout_strategy=<function verticalMenuLayout>)

  Los parámetros son:

  - items, lista de instancias BaseMenuItem que compondrán el menú.
  - selected_effect, función que será ejecutada cuando la instancia BaseMenuItem sea seleccionada.
  - unselected_effect, función que será ejecutada cuando la instancia BaseMenuItem sea deseleccionada.
  - activated_effect, función que será ejecutada cuando la instancia BaseMenuItem sea activada (pulsando Enter o haciendo clic sobre ella).

La clase **MenuItem**, basada en cocos.menu.BaseMenuItem, nos permite trabajar con un ítem de menú que muestra una etiqueta. Su sintaxis es:

```
MenuItem(label, callback_func, *args, **kwargs)
```

Con el parámetro label indicamos el texto de la etiqueta, y con callback_func la función que se ejecutará cuando se seleccione.

La clase **MultipleMenuItem**, basada en cocos.menu.MenuItem, nos permite trabajar con un ítem de menú que pueda cambiar entre múltiples valores.

```
MultipleMenuItem(label, callback_func, items, default_item=0)
```

Con el parámetro label indicamos el texto de la etiqueta, con callback_func la función que se ejecutará cuando se seleccione, con items los posibles valores que tendremos y con default_item el índice del ítem inicial.

La clase **ToggleMenuItem**, basada en cocos.menu.MultipleMenuItem, nos permite trabajar con un ítem de menú que muestra una opción booleana. Su sintaxis es:

```
ToggleMenuItem(label, callback_func, value=False)
```

Con el parámetro label indicamos el texto de la etiqueta, con callback_func la función que se ejecutará cuando se seleccione, y con value el valor booleano inicial.

La clase **EntryMenuItem**, basada en cocos.menu.MenuItem, nos permite trabajar con un ítem de menú en el que poder introducir una cadena.

```
EntryMenuItem(label, callback_func, value, max_length=0)
```

Con el parámetro label indicamos el texto de la etiqueta, con callback_func la función que se ejecutará cuando se seleccione, con value el valor inicial, y con max_length el tamaño máximo en caracteres de la casilla de entrada (0 significa sin límite). Cuando introducimos un valor cambia el contenido de value y se llama a callback_func con él como argumento.

La clase **ImageMenuItem**, basada en cocos.menu.BaseMenuItem, nos permite trabajar con un ítem de menú que muestra una imagen seleccionable.

```
ImageMenuItem(image, callback_func, *args, **kwargs)
```

Con el parámetro label indicamos el texto de la etiqueta y con callback_func la función que se ejecutará cuando se seleccione.

La clase **ColorMenuItem**, basada en cocos.menu.MenuItem, nos permite trabajar con un ítem de menú para seleccionar un color entre varios preseleccionados.

```
ColorMenuItem(label, callback_func, items, default_item=0)
```

Con el parámetro label indicamos el texto de la etiqueta, con callback_func la función que se ejecutará cuando se seleccione, con items una lista con los posibles colores (en forma de tupla RGB) que tendremos, y con default_item el índice del color inicial.

B.19 MÓDULO COCOS.PARTICLE

Módulo para tratar con los sistemas de partículas. Contiene las dos siguientes clases:

- **Color**
- **ParticleSystem**

También tres funciones:

- **PointerToNumpy**()
- **point_sprites_available**()
- **rand**()

Y una excepción:

- **ExceptionNoEmptyParticle**

La clase Color, basada en object, representa un color RGBA. Su sintaxis es:

```
Color(r, g, b, a)
```

Los parámetros coinciden con los **atributos**:

- **r**
  Número entero de 0 a 255 que indica la cantidad de color rojo.
- **g**
  Número entero de 0 a 255 que indica la cantidad de color verde.

- **b**
  Número entero de 0 a 255 que indica la cantidad de color azul.
- **a**
  Número entero de 0 a 255 que indica el valor de alpha (marca la opacidad).

Tiene el siguiente método:

- **to_array**()
  Convierte el color a una tupla RGBA.

La clase **ParticleSystem**, basada en cocos.cocosnode.CocosNode, es la clase base para los distintos sistemas de partículas que tenemos en cosos2d. La forma más sencilla de personalizarla es crear una subclase y redefinir algunos de sus atributos. Su sintaxis es:

```
ParticleSystem(fallback=None, texture=None)
```

Sus **atributos** más interesantes son:

- **active** = True
  Booleano que indica si el sistemas de partículas está o no activo.
- **angle** = 0.0
  El ángulo (medido en grados) de las partículas.
- **color_modulate** = True
  Booleano que indica si hay o no modulación de color.
- **duration** = 0
  Duración en segundos del sistema de partículas. Un valor -1 significa tiempo infinito.
- **elapsed** = 0
  Tiempo transcurrido (en segundos) desde el inicio del sistema de partículas.
- **emit_counter** = None
  Indica cuántas partículas pueden ser emitidas por segundo.
- **end_color** = Color(0.00, 0.00, 0.00, 0.00)
  Color final de las partículas.

- **gravity** = Point2(0.00, 0.00)
  Intensidad de la gravedad aplicada a las partículas. La clase Point2, basada en Vector2, la tenemos en el módulo cocos.euclid.
- **life** = 0
  Indica cuántos segundos vivirán las partículas.
- **particle_count** = None
  Conteo de las parrtículas.
- **radial_accel** = 0.0
  Aceleración radial, en grados por segundo.
- **scale**
  Factor de escala (número real) del objeto.
- **size** = 0.0
  Tamaño de las partículas.
- **speed** = 0.0
  Velocidad de las partículas.
- **start_color** = Color(0.00, 0.00, 0.00, 0.00)
  Color inicial de las partículas.
- **tangential_accel** = 0.0
  Aceleración tangencial.
- **total_particles** = 0
  Máximo número de partículas.

Entre sus **métodos** destacamos:

- **draw**()
  Dibuja el sistema de partículas.
- **init_particle**()
  Configura el estado inicial de las partículas.
- **on_cocos_resize**(usable_width, usable_height)
  Manejador para el redimensionamiento de la ventana. Los parámetros usable_width y usable_height son el nuevo ancho y alto de la ventana.

- **on_enter**()
  Llamado cada vez que el sistema de partículas entra en escena.
- **on_exit**()
  Llamado cada vez que el sistema de partículas sale de la escena.
- **reset_system**()
  Resetea el sistema de partículas.
- **step**(delta)
  Se llama en cada fotograma para crear (si es necesario) nuevas partículas y actualizar la posición de las que ya tenemos, o borrar alguna de ellas. Si hemos indicado una duración del sistema de partículas, comprueba si hemos llegado a él y lo elimina. Con el parámetro delta (número real) tenemos el tiempo entre fotograma y fotograma.
- **stop_system**()
  Para el sistema de partículas.
- **update_particles**(delta)
  Actualiza la posición de las partículas. Con el parámetro delta (número real) tenemos el tiempo entre fotograma y fotograma.

B.20 MÓDULO COCOS.PARTICLE_SYSTEMS

En este módulo tendremos las clases para los sistemas de partículas predefinidos de los que dispone cocos2d, que son:

- **Fireworks**
- **Spiral**
- **Meteor**
- **Sun**
- **Fire**
- **Galaxy**
- **Flower**
- **Explosion**
- **Smoke**

Todos están basados en la clase ParticleSystem y sobrescriben con determinados valores los atributos.

La clase **Fireworks**, basada en cocos.particle.ParticleSystem, es la base para los fuegos artificiales. Su sintaxis es:

```
Fireworks(fallback=None, texture=None)
```

Tiene los siguientes valores de sus **atributos**:

- angle = 90
- angle_var = 20
- blend_additive = False
- color_modulate = True
- duration = -1
- emission_rate = 857.1428571428571
- end_color = Color(0.10, 0.10, 0.10, 0.20)
- end_color_var = Color(0.10, 0.10, 0.10, 0.20)
- gravity = Point2(0.00, -90.00)
- life = 3.5
- life_var = 1
- pos_var = Point2(0.00, 0.00)
- radial_accel = 0
- radial_accel_var = 0
- size = 8.0
- size_var = 2.0
- speed = 180
- speed_var = 50
- start_color = Color(0.50, 0.50, 0.50, 1.00)
- start_color_var = Color(0.50, 0.50, 0.50, 1.00)
- total_particles = 3000

La clase **Spiral**, basada en cocos.particle.ParticleSystem, es la base para la espiral. Su sintaxis es:

```
Spiral(fallback=None, texture=None)
```

Los valores de sus **atributos** son:

- angle = 90.0
- angle_var = 0.0
- blend_additive = True
- color_modulate = True
- duration = -1
- emission_rate = 41.666666666666664
- end_color = Color(0.50, 0.50, 0.50, 1.00)
- end_color_var = Color(0.50, 0.50, 0.50, 0.00)
- gravity = Point2(0.00, 0.00)
- life = 12.0
- life_var = 0.0
- pos_var = Point2(0.00, 0.00)
- radial_accel = -380
- radial_accel_var = 0
- size = 20.0
- size_var = 10.0
- speed = 150.0
- speed_var = 0.0
- start_color = Color(0.50, 0.50, 0.50, 1.00)
- start_color_var = Color(0.50, 0.50, 0.50, 0.00)
- tangential_accel = 45.0
- tangential_accel_var = 0.0
- total_particles = 500

La clase **Meteor**, basada en cocos.particle.ParticleSystem, es la base para el meteorito. Su sintaxis es:

```
Meteor(fallback=None, texture=None)
```

Los valores de sus **atributos** son:

- angle = 90.0
- angle_var = 360.0
- blend_additive = True
- color_modulate = True
- duration = -1
- emission_rate = 75.0
- end_color = Color(0.00, 0.00, 0.00, 1.00)
- end_color_var = Color(0.00, 0.00, 0.00, 0.00)
- gravity = Point2(-200.00, 100.00)
- life = 2.0
- life_var = 1.0
- pos_var = Point2(0.00, 0.00)
- radial_accel = 0
- radial_accel_var = 0
- size = 60.0
- size_var = 10.0
- speed = 15.0
- speed_var = 5.0
- start_color = Color(0.20, 0.70, 0.70, 1.00)
- start_color_var = Color(0.00, 0.00, 0.00, 0.20)
- tangential_accel = 0.0
- tangential_accel_var = 0.0
- total_particles = 150

La clase **Sun**, basada en cocos.particle.ParticleSystem, es la base para el sol. Su sintaxis es:

```
Sun(fallback=None, texture=None)
```

Los valores de sus **atributos** son:

- angle = 90.0
- angle_var = 360.0
- blend_additive = True
- color_modulate = True
- duration = -1
- emission_rate = 350.0
- end_color = Color(0.00, 0.00, 0.00, 0.00)
- end_color_var = Color(0.00, 0.00, 0.00, 0.00)
- gravity = Point2(0.00, 0.00)
- life = 1.0
- life_var = 0.5
- pos_var = Point2(0.00, 0.00)
- radial_accel = 0
- radial_accel_var = 0
- size = 40.0
- size_var = 0.0
- speed = 20.0
- speed_var = 5.0
- start_color = Color(0.75, 0.25, 0.12, 1.00)
- start_color_var = Color(0.00, 0.00, 0.00, 0.00)
- tangential_accel = 0.0
- tangential_accel_var = 0.0
- total_particles = 350

La clase **Fire**, basada en cocos.particle.ParticleSystem, es la base para el fuego. Su sintaxis es:

```
Fire(fallback=None, texture=None)
```

Los valores de sus **atributos** son:

- angle = 90.0
- angle_var = 10.0
- blend_additive = True
- color_modulate = True
- duration = -1
- emission_rate = 83.33333333333333
- end_color = Color(0.00, 0.00, 0.00, 1.00)
- end_color_var = Color(0.00, 0.00, 0.00, 0.00)
- gravity = Point2(0.00, 0.00)
- life = 3.0
- life_var = 0.25
- pos_var = Point2(40.00, 20.00)
- radial_accel = 0
- radial_accel_var = 0
- size = 100.0
- size_var = 10.0
- speed = 60.0
- speed_var = 20.0
- start_color = Color(0.76, 0.25, 0.12, 1.00)
- start_color_var = Color(0.00, 0.00, 0.00, 0.00)
- total_particles = 250

La clase **Galaxy**, basada en cocos.particle.ParticleSystem, es la base para la galaxia. Su sintaxis es:

```
Galaxy(fallback=None, texture=None)
```

Los valores de sus **atributos** son:

- angle = 90.0
- angle_var = 360.0
- blend_additive = True
- color_modulate = True
- duration = -1
- emission_rate = 50.0
- end_color = Color(0.00, 0.00, 0.00, 0.00)
- end_color_var = Color(0.00, 0.00, 0.00, 0.00)
- gravity = Point2(0.00, 0.00)
- life = 4.0
- life_var = 1.0
- pos_var = Point2(0.00, 0.00)
- radial_accel = -80.0
- radial_accel_var = 0
- size = 37.0
- size_var = 10.0
- speed = 60.0
- speed_var = 10.0
- start_color = Color(0.12, 0.25, 0.76, 1.00)
- start_color_var = Color(0.00, 0.00, 0.00, 0.00)
- tangential_accel = 80.0
- tangential_accel_var = 0.0
- total_particles = 200

La clase **Flower**, basada en cocos.particle.ParticleSystem, es la base para la flor. Su sintaxis es:

```
Flower(fallback=None, texture=None)
```

Los valores de sus **atributos** son:

- angle = 90.0
- angle_var = 360.0
- blend_additive = True
- color_modulate = True
- duration = -1
- emission_rate = 125.0
- end_color = Color(0.00, 0.00, 0.00, 1.00)
- end_color_var = Color(0.00, 0.00, 0.00, 0.00)
- gravity = Point2(0.00, 0.00)
- life = 4.0
- life_var = 1.0
- pos_var = Point2(0.00, 0.00)
- radial_accel = -60
- radial_accel_var = 0
- size = 30.0
- size_var = 0.0
- speed = 80.0
- speed_var = 10.0
- start_color = Color(0.50, 0.50, 0.50, 1.00)
- start_color_var = Color(0.50, 0.50, 0.50, 0.00)
- tangential_accel = 15.0
- tangential_accel_var = 0.0
- total_particles = 500

La clase **Explosion**, basada en cocos.particle.ParticleSystem, es la base para la explosión. Su sintaxis es:

```
Explosion(fallback=None, texture=None)
```

Los valores de sus **atributos** son:

- angle = 90.0
- angle_var = 360.0
- blend_additive = False
- color_modulate = True
- duration = 0.1
- emission_rate = 7000.0
- end_color = Color(0.50, 0.50, 0.50, 0.00)
- end_color_var = Color(0.50, 0.50, 0.50, 0.00)
- gravity = Point2(0.00, -90.00)
- life = 5.0
- life_var = 2.0
- pos_var = Point2(0.00, 0.00)
- radial_accel = 0
- radial_accel_var = 0
- size = 15.0
- size_var = 10.0
- speed = 70.0
- speed_var = 40.0
- start_color = Color(0.70, 0.20, 0.10, 1.00)
- start_color_var = Color(0.50, 0.50, 0.50, 0.00)
- total_particles = 700

La clase **Smoke**, basada en cocos.particle.ParticleSystem, es la base para el humo. Su sintaxis es:

```
Smoke(fallback=None, texture=None)
```

Los valores de sus **atributos** son:

- angle = 90.0
- angle_var = 10.0
- blend_additive = True
- color_modulate = False
- duration = -1
- emission_rate = 20.0
- end_color = Color(0.50, 0.50, 0.50, 0.10)
- end_color_var = Color(0.00, 0.00, 0.00, 0.10)
- gravity = Point2(0.00, 0.00)
- life = 4.0
- life_var = 1.0
- pos_var = Point2(0.10, 0.00)
- radial_accel = 5
- radial_accel_var = 0
- size = 40.0
- size_var = 10.0
- speed = 25.0
- speed_var = 10.0
- start_color = Color(0.50, 0.50, 0.50, 0.10)
- start_color_var = Color(0.00, 0.00, 0.00, 0.10)
- tangential_accel = 0.0
- tangential_accel_var = 0.0
- total_particles = 80

B.21 MÓDULO COCOS.RECT

Tendremos únicamente la clase Rect, que nos definirá un área rectangular, que incluye sus lados izquierdo e inferior pero no el superior y derecho.

La clase **Rect**, basada en object, tiene la siguiente sintaxis[143]:

```
Rect(x, y, width, height)
```

Los parámetros x e y nos indican las coordenadas (en píxeles) de la esquina inferior izquierda del rectángulo, mientras que width y height nos marcan, respectivamente, su anchura y altura (también en píxeles). Disponemos de los siguientes atributos y métodos principales:

- **Atributos**:
  - **top**
    Entero que indica el valor de la coordenada y del lado superior del rectángulo.
  - **bottom**
    Entero que indica el valor de la coordenada y del lado inferior del rectángulo.
  - **left**
    Entero que indica el valor de la coordenada x del lado izquierdo del rectángulo.
  - **right**
    Entero que indica el valor de la coordenada x del lado derecho del rectángulo.
  - **position**
    Tupla de enteros que indica las coordenadas en píxeles de la esquina inferior izquierda del rectángulo.
  - **x**
    Entero que indica (en píxeles) la coordenada x de la esquina inferior izquierda del rectángulo.
  - **y**
    Entero que indica (en píxeles) la coordenada y de la esquina inferior izquierda del rectángulo.

143 Los parámetros pasados son los valores de los atributos de igual nombre.

- **origin**
  Tupla de enteros que indica las coordenadas en píxeles de la esquina inferior izquierda del rectángulo.
- **center**
  Tupla de enteros que indica las coordenadas en píxeles del centro del rectángulo.
- **topleft**
  Tupla de enteros que indica las coordenadas en píxeles de la esquina superior izquierda del rectángulo.
- **topright**
  Tupla de enteros que indica las coordenadas en píxeles de la esquina superior derecha del rectángulo.
- **bottomleft**
  Tupla de enteros que indica las coordenadas en píxeles de la esquina inferior izquierda del rectángulo.
- **bottomright**
  Tupla de enteros que indica las coordenadas en píxeles de la esquina inferior derecha del rectángulo.
- **midtop**
  Tupla de enteros que indica las coordenadas en píxeles del punto medio del lado superior del rectángulo.
- **midbottom**
  Tupla de enteros que indica las coordenadas en píxeles del punto medio del lado superior del rectángulo.
- **midleft**
  Tupla de enteros que indica las coordenadas en píxeles del punto medio del lado izquierdo del rectángulo.
- **midright**
  Tupla de enteros que indica las coordenadas en píxeles del punto medio del lado derecho del rectángulo.
- **size**
  Tupla de enteros que indica el ancho y alto (en píxeles) del rectángulo.
- **height**
  Entero que indica (en píxeles) la altura del rectángulo.
- **width**
  Entero que indica (en píxeles) la anchura del rectángulo.

- **Métodos**:
  - **clippedBy**(other)
    Nos devuelve booleano indicando si nuestro rectángulo no encaja completamente dentro del rectángulo other[144]. Es decir, devolverá True si la intersección de ambos es menor que nuestro rectángulo, y False en caso contrario.
  - **contains**(x, y)
    Devuelve booleano indicando si el punto indicado por las coordenadas (x,y) en píxeles está dentro del área de nuestro rectángulo.
  - **copy**()
    Devuelve una copia de nuestro rectángulo.
  - **get_bottom**()
    Nos devuelve el atributo bottom.
  - **get_bottomleft**()
    Nos devuelve el atributo bottomleft.
  - **get_bottomright**()
    Nos devuelve el atributo bottomright.
  - **get_center**()
    Nos devuelve el atributo center.
  - **get_left**()
    Nos devuelve el atributo left.
  - **get_midbottom**()
    Nos devuelve el atributo midbottom.
  - **get_midleft**()
    Nos devuelve el atributo midleft.
  - **get_midright**()
    Nos devuelve el atributo midright.
  - **get_midtop**()
    Nos devuelve el atributo midtop.
  - **get_origin**()
    Nos devuelve el atributo origin.
  - **get_right**()
    Nos devuelve el atributo right.

144 Por lo tanto debe ser instancia de Rect.

- **get_top**()¶
  Nos devuelve el atributo top.
- **get_topleft**()
  Nos devuelve el atributo topleft.
- **get_topright**()
  Nos devuelve el atributo topright.
- **intersect**(other)
  Nos devuelve la intersección entre nuestro propio rectángulo y el rectángulo other[145]. Si existe nos devuelve una instancia de Rect, en caso contrario None.
- **intersects**(other)
  Nos devuelve booleano indicando si el rectángulo other[146] intersecta al nuestro.
- **set_bottom**(y)
  Configura el atributo bottom. El parámetro y es un entero.
- **set_bottomleft**(position)
  Configura el atributo bottom. El parámetro position es una tupla de dos enteros.
- **set_bottomright**(position)
  Configura el atributo bottomright. El parámetro position es una tupla de dos enteros.
- **set_center**(center)
  Configura el atributo center. El parámetro center es una tupla de dos enteros.
- **set_height**(value)
  Configura el atributo height. El parámetro value es un entero.
- **set_left**(x)
  Configura el atributo left. El parámetro x es un entero.
- **set_midbottom**(midbottom)
  Configura el atributo midbottom. El parámetro midbottom es una tupla de dos enteros.

145 Por lo tanto debe ser instancia de Rect.

146 Por lo tanto debe ser instancia de Rect.

- **set_midleft**(midleft)
  Configura el atributo midleft. El parámetro midleft es una tupla de dos enteros.
- **set_midright**(midright)
  Configura el atributo midright. El parámetro midright es una tupla de dos enteros.
- **set_midtop**(midtop)
  Configura el atributo midtop. El parámetro midtop es una tupla de dos enteros.
- **set_origin**(origin)
  Configura el atributo origin. El parámetro origin es una tupla de dos enteros.
- **set_position**(value)
  Configura el atributo position. El parámetro value es una tupla de dos enteros.
- **set_right**(x)
  Configura el atributo right. El parámetro x es un entero.
- **set_size**(value)
  Configura el atributo size. El parámetro value es una tupla de dos enteros.
- **set_top**(y)
  Configura el atributo top. El parámetro y es un entero.
- **set_topleft**(position)
  Configura el atributo topleft. El parámetro position es una tupla de dos enteros.
- **set_topright**(position)
  Configura el atributo topright. El parámetro position es una tupla de dos enteros.
- **set_width**(value)
  Configura el atributo width. El parámetro x es un entero.
- **set_x**(value)
  Configura el atributo x. El parámetro value es un entero.
- **set_y**(value)
  Configura el atributo y. El parámetro value es un entero.

B.22 MÓDULO COCOS.SCENE

En este módulo tendremos la clase que nos permitirá tratar con las escenas.

La clase **Scene**, basada en cocos.cocosnode.CocosNode, creará una escena con capas y/o escenas, controlando el envío de eventos a las capas y la reproducción de música de fondo. Tiene la siguiente sintaxis:

```
Scene(*children)
```

El parámetro children será la capa o escena que contendrá la escena. En el caso de contener varias de ellas será una lista que las incluya, asignándose automáticamente un número entero que indica su nivel (eje z) y que irá de 0 al número de elementos hijo menos uno de la escena.

Destacaremos los siguientes **métodos**:

- **end**(value=None)
  Finaliza la escena en curso, pasando como parámetro opcional el valor de retorno value[147]. Será acompañada por la llamada a Director.pop() y la configuración de director.return_value al valor value.

- **load_music**(filename)
  Carga el fichero musical de nombre filename[148] para poder ser posteriormente reproducido de forma continua en la escena[149], deteniendo la música que pudiese estar sonando en ese momento.

- **on_enter**()
  Se llama cada vez que la escena entra en ejecución.

- **on_exit**()
  Se llama cada vez que la escena deja de ejecutarse.

- **play_music**()
  Activa la reproducción de música en la escena. Si ya estaba sonando no hará nada. Si se llama a este método en una escena inactiva, la música comenzará a reproducirse sólo si se activa la escena (y en ese justo momento).

147 Puede ser cualquier cosa, como un tipo o una instancia.

148 Por lo tanto será una cadena.

149 Dependiendo de las bibliotecas instaladas, los formatos soportados pueden ser WAV, MP3, OGG o MOD. También tenemos la opción de indicar None para desactivar la música.

- **stop_music**()
  Desactiva la reproducción de música en la escena.

B.23 MÓDULO COCOS.SPRITE

La clase **Sprite**, basada en cocos.batch.BatchableNode[150] y pyglet.sprite.Sprite, tiene la siguiente sintaxis[151]:

```
Sprite(image, position=(0, 0), rotation=0, scale=1, opacity=255,
               color=(255, 255, 255), anchor=None)
```

Disponemos de los siguientes atributos y métodos principales:

- **Atributos**:
  - **image**
    Cadena que nos indica el nombre de la imagen del sprite. También puede ser un objeto de la clase pyglet.image.AbstractImage.
  - **position**
    Tupla de números enteros que nos indica las coordenadas iniciales[152] del punto de referencia o anclaje (anchor) del sprite. Por defecto su valor es (0,0).
  - **rotation**
    Número real que nos indica la rotación en grados del sprite. Su valor por defecto es 0.
  - **scale**
    Número real que nos indica el factor de escala del sprite. Su valor por defecto es 1.
  - **scale_x**
    Número real que indica el factor de escala horizontal. Su valor por defecto es 1.
  - **scale_y**
    Número real que indica el factor de escala vertical. Su valor por defecto es 1.

150 Deriva a su vez de cocos.cocosnode.CocosNode.

151 Los parámetros pasados son los valores de los atributos de igual nombre.

152 Puede ser posteriormente modificado mediante el atributo anchor.

- **opacity**
  Número entero que marca el nivel de opacidad del sprite. Sus valores van desde 0 (transparente) a 255 (totalmente opaco, valor por defecto).
- **color**
  Tupla de 3 números enteros entre 0 y 255 que nos marcan la cantidad de color rojo, verde o azul (RGB, red, green, blue) del color. El valor por defecto es (255,255,255).
- **Anchor**
  Tupla de dos números enteros que nos marca las variaciones respecto a position del punto de referencia del sprite que usaremos para posicionar, rotar o escalar el sprite.
- **height**
  Entero que nos indica el alto en píxeles del sprite. Sólo lectura. Invariable ante rotaciones.
- **image_anchor**
  Tupla de enteros que nos indica las coordenadas (relativas a position) en píxeles del punto a partir del cual la imagen del sprite será posicionada, rotada o escalada.
- **image_anchor_x**
  Entero que indica la coordenada x del atributo image_anchor.
- **image_anchor_y**
  Entero que indica la coordenada y del atributo image_anchor.
- **supported_classes**
  Alias de la clase Sprite.
- **width**
  Entero que nos indica el ancho en píxeles del sprite. Sólo lectura. Invariable ante rotaciones.
- **x**
  Entero que nos indica la coordenada x (en píxeles) del sprite.
- **y**
  Entero que nos indica la coordenada y (en píxeles) del sprite.

◤ **Métodos**:

- **contains**(x, y)
  Devuelve booleano indicando si el punto de coordenadas (x,y) está en el área rectangular (sin tranformación) que ocupa el sprite.

- **draw**()
  Si el sprite no está dentro de un batch se dibujará mediante este método. Si lo está será el batch el que lo dibuje y no se llamará al método.
- **get_AABB**()
  Nos devuelve (en forma de instancia de cocos.rect.Rect) el rectángulo de lados paralelos a los ejes que delimita al sprite. Notar que la posición de este rectángulo[153] no es la posición del sprite, indicada por defecto por el punto central del rectángulo.
- **get_rect**()
  Nos devuelve (en forma de instancia de cocos.rect.Rect) el rectángulo que delimita al sprite. Notar que la posición de este rectángulo[154] no es la posición del sprite, indicada por defecto por el punto central del rectángulo.

B.24 MÓDULO COCOS.TEXT

Es el módulo para el soporte del texto. Destacamos las 4 siguientes clases:

- **TextElement**
- **Label**
- **RichLabel**
- **HTMLLabel**

La clase **TextElement**, basada en cocos.cocosnode.CocosNode, es la clase base para todos los textos de cocos2d. Su sintaxis es:

```
TextElement(text='', position=(0, 0), **kwargs)
```

En ella text es el texto y position la posición.

La clase **Label**, basada en cocos.text.TextElement, proporciona el soporte para el texto plano. Su sintaxis es:

```
Label(text='', position=(0, 0), **kwargs)
```

153 Las coordenadas de su esquina inferior izquierda.

154 Las coordenadas de su esquina inferior izquierda.

En ella text es el texto y position la posición. Los **parámetros** restantes son:

- **font_name**
  Nombre de la fuente del texto.
- **font_size**
  Tamaño en puntos de la fuente.
- **bold**
  Booleano indicando si está en negrita o no.
- **italic**
  Booleano indicando si está en cursiva o no.
- **color**
  Tupla de cuatro enteros RGBA que indica el color de la fuente.
- **width**
  Ancho de la etiqueta en píxeles, o None.
- **height**
  Alto de la etiqueta en píxeles, o None.
- **anchor_x**
  Anclaje en el eje x. Puede tener los valores “left, “center” o “right” (izquierda, centrado o derecha).
- **anchor_y**
  Anclaje en el eje y. Puede tener los valores “bottom”, “baseline”, “center” or “top” (abajo, en la base, centrado o arriba).
- **align**
  Indica la alineación. Se aplica cuando se indica width. Puede tomar los valores “left”, “center” o “right” (izquierda, centrada o derecha).
- **multiline**
  Booleano indicando si puede haber varias líneas.
- **dpi**
  Resolución de las fuentes. Su valor por defecto es 96.

La clase **RichLabel**, basada en cocos.text.TextElement, proporciona el soporte para texto enriquecido. Su sintaxis es:

```
RichLabel(text='', position=(0, 0), **kwargs)
```

Sus posibles argumentos son los mismos que en el caso de Label.

La clase **HTMLLabel**, basada en cocos.text.TextElement, proporciona las etiquetas de texto con formato HTML[155] (soporta un subconjunto de HTML 4.01). Su sintaxis es:

```
HTMLLabel(text='', position=(0, 0), **kwargs)
```

En ella text es el texto y position la posición. Los **parámetros** restantes son:

- **location**

  Objeto de localización para cargar las imágenes a las que se hace referencia en el documento. Por defecto, se utiliza el directorio de trabajo.

- **width**

  Ancho de la etiqueta en píxeles, o None.

- **height**

  Alto de la etiqueta en píxeles, o None.

- **anchor_x**

  Anclaje en el eje x. Puede tener los valores "left, "center" o "right" (izquierda, centrado o derecha).

- **anchor_y**

  Anclaje en el eje y. Puede tener los valores "bottom", "baseline", "center" or "top" (abajo, en la base, centrado o arriba).

- **multiline**

  Booleano indicando si puede haber varias líneas.

- **dpi**

  Resolución de las fuentes. Su valor por defecto es 96.

155 HyperText Markup Language, lenguaje de marcado de hipertexto.

B.25 MÓDULO COCOS.TILES

En este módulo dispondremos de clases que nos permitirán manejar mapas de baldosas. Podremos cargarlos, guardarlos y renderizarlos mediante la API que nos proporciona.

Dispone de las siguientes clases:

- **MapLayer**, clase base para los mapas.
- **RectMapLayer**, para los mapas (renderizables y en los que se puede hacer scroll) de mosaicos rectangulares.
- **HexMapLayer**, para los mapas (renderizables y en los que se puede hacer scroll) de mosaicos hexagonales.
- **RegularTesselationMap**, para los mapas teselados regularmente y que permite acceder a sus celdas mediante los índices (i,j).
- **RectMap**, para los mapas rectangulares.
- **HexMap**, para los mapas hexagonales.
- **Cell**, clase base para celdas de mapas rectangulares y hexagonales.
- **RectCell**, para las celdas de un mapa rectangular, que son rectángulos.
- **HexCell**, para las celdas de un mapa hexagonal, que son hexágonos regulares.
- **RectMapCollider**, movido al módulo cocos.mapcolliders.
- **Resource**, para cargar recursos del mapa de mosaicos desde un fichero XML.
- **Tile**, para tratar las baldosas, que contienen una imagen y algunas propiedades opcionales.
- **TileSet**, para contener un conjunto de objetos baldosa referenciados por algún identificador.
- **TmxObject**, que representa un objeto en una capa de objetos TMX.
- **TmxObjectLayer**, para tratar con una capa compuesta por formas básicas primitivas.

Y destacaremos, de entre todas las funciones disponibles, las siguientes:

- **load**(filename), carga recursos desde un fichero XML de nombre filename.
- **load_tiles**(filename), que carga los recursos del mapa desde un fichero XML de nombre filename.
- **load_tmx**(filename), carga los recursos del mapa desde un fichero TMX de nombre filename.

También dispondremos de las siguientes excepciones:

- **ResourceError**
- **TilesPropertyWithoutName**
- **TilesPropertyWithoutValue**
- **TmxUnsupportedVariant**

La clase **MapLayer**, basada en cocos.layer.scrolling.ScrollableLayer, es la clase base para los mapas (que están compuestos de baldosas, pudiendo determinar cuáles de ellas son renderizables por pantalla). Su sintaxis es:

```
MapLayer(properties)
```

Con properties indicamos los valores de los posibles **atributos**, que son:

- **id**
  Identifica el mapa en el fichero XML y en los recursos.
- **(width, height)**
  Tamaño del mapa en celdas.
- **(px_width, px_height)**
  Tamaño del mapa en píxels.
- **(tw, th)**
  Tamaño (en píxels) de cada celda.
- **(origin_x, origin_y, origin_z)**
  Offset (desde el origen) de la esquina superior izquierda del mapa en píxels.
- **cells**
  Array de las celdas que componen el mapa.

- **debug**

  Booleano que indica si representamos o no información de depuración en las celdas. Su valor por defecto es False.

- **properties**

  Diccionario de propiedades del mapa.

Los **métodos** mas relevantes son:

- **find_cells**(**requirements)

  Busca las celdas que tengan una propiedad determinada que especificaremos.

- **get_cell**(i, j)

  Nos devuelve la celda que ocupa la coordenada (i,j), o None si está fuera de los límites.

- **get_at_pixel**(x, y)

  Nos devuelve la celda que está en las coordenadas en píxeles (x,y) del mapa, o None si está fuera de los límites.

- **get_in_region**(left, bottom, right, top)

  Nos devuelve una lista de celdas que intersectan el rectángulo marcado por los parámetros, donde (left,bottom) indican la esquina inferior izquierda y (right,top) la esquina superior derecha.

- **is_visible**(rect)

  Nos devuelve booleano indicando si el rectángulo rect de esta capa es visible o no.

- **set_cell_color**(i, j, color)

  Configuramos el color de la celda del mapa con coordenadas (i,j).

- **set_cell_opacity**(i, j, opacity)

  Configuramos la opacidad de la celda del mapa con coordenadas (i,j).

- **set_debug**(debug)

  Configuramos mediante el parámetro booleano debug el valor del atributo debug.

- **set_view**(x, y, w, h, viewport_x=0, viewport_y=0)

  Configura la vista del mapa.

La clase **RegularTesselationMap**, basada en object, nos permite teselar de forma regular los mapas y nos permite acceder a sus celdas mediante los índices (i,j). De ella destacaremos simplemente el siguiente **método**:

- **get_cell**(i, j)
  Nos devuelve la celda que ocupa la coordenada (i,j).

La clase **RectMap**, basada en cocos.tiles.RegularTesselationMap, nos permite tener un mapa teselado con rectángulos. Las celdas se referencian de la siguiente manera:

IJKL
EFGH
ABCD

Y se almacenan así:

[[A,E,I], [B,F,J], [C,G,K], [D,H,L]]

Por lo tanto la celda (1,2) será J.

Tiene los siguientes **atributos**[156]:

- **LEFT** = (-1, 0)
- **RIGHT** = (1, 0)
- **UP** = (0, 1)
- **DOWN** = (0, -1)

Destacaremos los siguientes **métodos**:

- **get_at_pixel**(x, y)
  Nos devuelve la celda que está en las coordenadas en píxeles (x,y) del mapa, o None si está fuera de los límites.
- **get_in_region**(left, bottom, right, top)
  Nos devuelve una lista de celdas que intersectan el rectángulo marcado por los parámetros, donde (left, bottom) indican la esquina inferior izquierda y (right, top) la esquina superior derecha.

156 Veremos su utilidad al comentar los métodos.

- **get_neighbor**(cell, direction)

  Nos devuelve la celda colindante a la celda cell pasada como parámetro, en base a lo indicado por direction (cuyo valor puede ser self.LEFT, self.RIGHT, self.UP o self.DOWN). Devolverá None si está fuera de los límites.

- **get_neighbors**(cell, diagonals=False)

  Nos devuelve un diccionario que incluye las celdas colindantes a los lados de la proporcionada mediante el parámetro cell. Si diagonals es True se incluyen además las que tocan sus esquinas. El diccionario incluye como llaves self.UP, self.DOWN...y como valores las celdas.

Su sintaxis es la siguiente:

```
RectMap(id, tw, th, cells, origin=None, properties=None)
```

Donde los **parámetros** son:

- **id**

  Cadena que identifica el mapa. En formato XML.

- **tw**

  Entero que indica el número de columnas de celdas.

- **th**

  Entero que indica el número de filas de celdas.

- **cells**

  Contenedor que alberga las celdas y que soporta indexado [i][j].

- **origin**

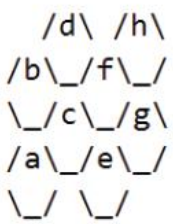
  Tupla de tres enteros que indican el offset del mapa. Por defecto (0, 0, 0).

- **properties**

  Diccionario donde se almacenan propiedades arbitrarias[157].

157 Si posteriormente guardamos en formato XML las llaves deben tener formato Unicode o ASCII de 8 bits.

La clase **HexMap**, basada en cocos.tiles.RegularTesselationMap, nos permite tener un mapa teselado con baldosas hexagonales. Las celdas se referencian de la siguiente manera:



Y se almacenan así:

[[a, b], [c,d], [e,f], [g, h]]

Por lo tanto la celda (2,1) será f.

Su sintaxis es:

```
HexMap(id, th, cells, origin=None, properties=None)
```

Donde los **parámetros** son:

- **id**
  Cadena que identifica el mapa.
- **th**
  Altura (en píxeles) de una baldosa.
- **cells**
  Contenedor que alberga las celdas y que soporta indexado [i][j].
- **origin**
  Tupla de tres enteros que indican el offset del mapa. Por defecto (0, 0, 0).
- **properties**
  Diccionario donde se almacenan propiedades arbitrarias[158].

158 Si posteriormente guardamos en formato XML las llaves deben tener formato Unicode o ASCII de 8 bits.

Tiene los siguientes **atributos**[159]:

- **th**
  Altura de la baldosa en píxeles.
- **tw** = edge_length * 2
  Anchura de la baldosa en píxeles.
- **edge_length** = int(th / sqrt(3))
  Longitud (en píxeles) de un lado de hexágono.
- **UP** = 'up'
- **UP_LEFT** = 'up left'
- **UP_RIGHT** = 'up right'
- **DOWN** = 'down'
- **DOWN_LEFT** = 'down left'
- **DOWN_RIGHT** = 'down right'

Entre los **métodos** destacamos:

- **get_at_pixel**(x, y)
  Nos devuelve la celda que está en las coordenadas en píxeles (x,y) del mapa, o None si está fuera de los límites.
- **get_in_region**(left, bottom, right, top)
  Nos devuelve una lista de celdas que intersectan el hexágono marcado por los parámetros, donde (left, bottom) indican la esquina inferior izquierda y (right, top) la esquina superior derecha.
- **get_neighbor**(cell, direction)
  Nos devuelve la celda colindante a la celda cell pasada como parámetro, en base a lo indicado por direction (que puede ser self.UP, self.DOWN, self.UP_LEFT, self.UP_RIGHT, self.DOWN_LEFT or self.DOWN_RIGHT). Devolverá None si está fuera de los límites.
- **get_neighbors**(cell)
  Nos devuelve un diccionario que incluye las celdas colindantes a los lados de la proporcionada mediante el parámetro cell. El diccionario incluye como llaves self.UP, self.DOWN, ... y como valores las celdas.

159 Veremos su utilidad al comentar los métodos.

La clase **RectMapLayer**, que deriva de cocos.tiles.RectMap y cocos.tiles.MapLayer, nos permite tratar con mapas (renderizables y en los que se puede hacer scroll) de baldosas rectangulares**.**

Tiene la siguiente sintaxis:

```
RectMapLayer(id, tw, th, cells, origin=None, properties=None)
```

Siendo los **parámetros**:

- **id**
  Cadena que identifica el mapa.
- **tw**
  Entero que indica el número de columnas de celdas.
- **th**
  Entero que indica el número de filas de celdas.
- **cells**
  Contenedor que alberga las celdas y que soporta indexado [i][j].
- **origin**
  Tupla de tres enteros que indican el offset del mapa. Por defecto (0, 0, 0).
- **properties**
  Diccionario donde se almacenan propiedades arbitrarias[160].

La clase **HexMapLayer**, que deriva de cocos.tiles.HexMap y cocos.tiles.MapLayer, nos permitirá trabajar con mapas (renderizables y en los que se puede hacer scroll) de mosaicos hexagonales.

Su sintaxis es:

```
HexMapLayer(id, ignored, th, cells, origin=None, properties=None)
```

Los parámetros son los mismos que en cocos.tiles.HexMap con el añadido de ignored, que se añade simplemente para que coincida con el formato de RectMapLayer y que como su nombre indica será ignorado.

160 Si posteriormente guardamos en formato XML las llaves deben tener formato Unicode o ASCII de 8 bits.

La clase **Cell** es la clase base para celdas de mapas rectangulares y hexagonales. Su sintaxis es:

```
Cell(i, j, width, height, properties, tile)
```

Sus parámetros tienen que ver con sus **atributos**:

- **i, j**
  Índices (coordenadas) de la celda en el mapa.
- **position**
  Coordenadas de la celda en forma de tupla.
- **width**, **height**
  Ancho y alto (en píxeles) de la celda.
- **properties**
  Propiedades arbitrarias de la celda.
- **tile**
  celda correspondiente de MapLayer.

Como **método** destacaremos:

- **get**(key, default=None)
  Nos devuelve el valor correspondiente a la llave key en el diccionario de propiedades. De no existir nos devolverá el valor de default, que por defecto es None.

La clase **RectCell**, basada en cocos.rect.Rect y cocos.tiles.Cell, nos permite trabajar con las celdas de un mapa teselado rectangularmente. Su sintaxis es:

```
RectCell(i, j, width, height, properties, tile)
```

Sus parámetros tienen que ver con sus **atributos**:

- **i**, **j**
  Índices de la celda en el mapa.
- **x**, **y**
  Coordenadas, en píxeles, de la esquina inferior izquierda de la celda.

- **width**, **height**
  Anchura y altura, en píxeles, de la celda.
- **properties**
  Propiedades arbitrarias de la celda.
- **tile**
  celda correspondiente de MapLayer.

Los siguientes atributos son de solo lectura:

- **top**
  Coordenada y (en píxeles) del lado superior de la celda.
- **bottom**
  Coordenada y (en píxeles) del lado inferior de la celda
- **left**
  Coordenada x (en píxeles) del lado izquierdo de la celda.
- **right**
  Coordenada x (en píxeles) del lado derecho de la celda.
- **center**
  Coordenadas (x, y) en píxeles del centro de la celda.
- **origin**
  Coordenadas (x, y) en píxeles de la esquina inferior izquierda de la celda.
- **topleft**
  Coordenadas (x, y) en píxeles la esquina superior izquierda de la celda.
- **topright**
  Coordenadas (x, y) en píxeles la esquina superior derecha de la celda.
- **bottomleft**
  Coordenadas (x, y) en píxeles la esquina inferior izquierda de la celda.
- **bottomright**
  Coordenadas (x, y) en píxeles la esquina inferior derecha de la celda.

- **midtop**

  Coordenadas (x, y) en píxeles del punto intermedio del lado superior de la celda.

- **midbottom**

  Coordenadas (x, y) en píxeles del punto intermedio del lado inferior de la celda.

- **midleft**

  Coordenadas (x, y) en píxeles del punto intermedio del lado izquierdo de la celda.

- **midright**

  Coordenadas (x, y) en píxeles del punto intermedio del lado derecho de la celda.

La celda puede tener las propiedades estándar "top", "left", "bottom" y "right", que son booleanos indicativos de la intransitabilidad de los lados superior, izquierdo, inferior y derecho.

Los atributos indicados en píxeles no están adaptados a las transformaciones de pantalla, vista o capa.

Los métodos simplemente nos devolverán el atributo indicado por el nombre:

- **get_top**()
- **get_bottom**()
- **get_left**()
- **get_right**()
- **get_center**()
- **get_origin**()
- **get_topleft**()
- **get_topright**()
- **get_bottomleft**()
- **get_bottomright**()
- **get_midtop**()
- **get_midbottom**()
- **get_midleft**()
- **get_midright**()

La clase **HexCell**, basada en cocos.tiles.Cell, hará que podamos trabajar con las celdas de un mapa teselado mediante hexágonos regulares. Su sintaxis es:

```
HexCell(i, j, ignored, height, properties, tile)
```

Donde los parámetros son los valores de los **atributos**:

- **i**, **j**
  Índices de la celda en el mapa.
- **x**, **y**
  Coordenadas, en píxeles, de la esquina inferior izquierda de la celda.
- **width**, **height**
  Anchura y altura, en píxeles, de la celda.
- **properties**
  Propiedades arbitrarias de la celda.
- **tile**
  celda correspondiente de MapLayer.

Los siguientes atributos son de solo lectura:

- **top**
  Coordenada y (en píxeles) del lado superior de la celda.
- **bottom**
  Coordenada y (en píxeles) del lado inferior de la celda
- **left**
  Coordenada x (en píxeles) del lado izquierdo de la celda.
- **right**
  Coordenada x (en píxeles) del lado derecho de la celda.
- **center**
  Coordenadas (x, y) en píxeles del centro de la celda.
- **origin**
  Coordenadas (x, y) en píxeles de la esquina inferior izquierda de la celda.

- **topleft**
  Coordenadas (x, y) en píxeles la esquina superior izquierda de la celda.
- **topright**
  Coordenadas (x, y) en píxeles la esquina superior derecha de la celda.
- **bottomleft**
  Coordenadas (x, y) en píxeles la esquina inferior izquierda de la celda.
- **bottomright**
  Coordenadas (x, y) en píxeles la esquina inferior derecha de la celda.
- **midtop**
  Coordenadas (x, y) en píxeles del punto intermedio del lado superior de la celda.
- **midbottom**
  Coordenadas (x, y) en píxeles del punto intermedio del lado inferior de la celda.
- **midtopleft**
  Coordenadas (x, y) en píxeles del punto intermedio del lado superior izquierdo de la celda.
- **midtopright**
  Coordenadas (x, y) en píxeles del punto intermedio del lado superior derecho de la celda.
- **midbottomleft**
  Coordenadas (x, y) en píxeles del punto intermedio del lado inferior izquierdo de la celda.
- **midbottomright**
  Coordenadas (x, y) en píxeles del punto intermedio del lado inferior derecho de la celda.

Los atributos indicados en píxeles no están adaptados a las transformaciones de pantalla, vista o capa.

Los **métodos** devolverán el atributo indicado por el nombre:

- **get_top**()
- **get_bottom**()
- **get_left**()
- **get_right**()
- **get_center**()
- **get_origin**()
- **get_topleft**()
- **get_topright**()
- **get_bottomleft**()
- **get_bottomright**()
- **get_midtop**()
- **get_midbottom**()
- **get_midtopleft**()
- **get_midtopright**()
- **get_midbottomleft**()
- **get_midbottomright**()

La clase **Resource**, que deriva de object, nos permite cargar recursos del mapa procedentes de un fichero XML.

Destacaremos los siguientes **métodos**:

- **add_resource**(id, resource)
  Añade un recurso resource con el identificador id.
- **find**(cls)
  Busca todos los elementos de la clase cls en los recursos.
- **find_file**(filename)
  Busca el fichero de nombre filename en los recursos.
- **get_resource**(ref)
  Nos devuelve el recurso referenciado como ref.

- **save_xml**(filename)
  Graba los recursos en el fichero de nombre filename.

La sintaxis es:

```
Resource(filename)
```

En ella filename es en nombre del fichero XML desde el que cargamos los recursos.

La clase **Tile**, basada en object, nos permite trabajar con las baldosas, que contienen una imagen e información adicional.

Su sintaxis es:

```
Tile(id, properties, image, offset=None)
```

En ella id es el identificador, properties son las propiedades, image la imagen almacenada y offset el ajuste.

La clase **TileSet**, basada en dict, contendrá (en forma de diccionario) un grupo de baldosas referenciadas por un identificador. La sintaxis es:

```
TileSet(id, properties)
```

Los parémetros id y properties indican, respectivamemte, el identificador y las propiedades del grupo de baldosas.

Destacamos el siguiente **método**:

- **add**(properties, image, id=None)
  Añade una nueva baldosa con la imagen image, las propiedades properties y el identificador id[161]. Nos devolverá la instancia de la clase Tile.

La clase **TmxObject**, basada en cocos.rect.Rect, representa un objeto TMX dentro de una capa que contiene ese tipo de elementos. Su sintaxis es:

```
TmxObject(tmxtype, usertype, x, y, width=0, height=0, name=None,
          id=None, tile=None, visible=1, points=None)
```

161 Se generará un identificador único si no lo indicamos explícitamente.

En ella los parámetros corresponden con los **atributos**:

- **tmxtype**
  Tipo de objeto, pudiendo ser las cadenas 'ellipse', 'polygon', 'polyline', 'rect' o 'tile' (elipse, polígono, línea poligonar, rectángulo o baldosa).

- **name**
  Cadena arbitraria que indica el nombre del objeto. Es el campo 'name' en el editor de Tiled.

- **usertype**
  Cadena arbitraria que indica el tipo del objeto indicado por el usuario. Es el campo 'type' en el editor de Tiled.

- **x**
  La coordenada x de la esquina inferior izquierda del rectángulo de lados paralelos a los ejes que delimita el objeto.

- **y**
  La coordenada y de la esquina inferior izquierda del rectángulo de lados paralelos a los ejes que delimita el objeto.

- **width**
  La anchura del objeto, en píxeles. Por defecto su valor es 0.

- **height**
  La altura del objeto, en píxeles. Por defecto su valor es 0.

- **tile**
  Referencia a una baldosa asociada. Opcional.

- **gid**
  Identificador para la baldosa asociada. Opcional.

- **visible**
  Indica si el objeto es visible (1, valor por defecto) o no (0).

- **points**
  Secuencia de coordenadas (x, y) relativas[162] que representan los vértices en un polígono ('polygon') o línea poligonar ('polyline').

162 Respecto a la esquina inferior izquierda del rectángulo de lados paralelos a los ejes que delimita el objeto.

La clase **TmxObjectLayer**, basada en cocos.tiles.MapLayer, nos servirá para tratar con una capa compuesta por objetos TMX (formas básicas primitivas). Tenemos la siguiente sintaxis:

```
TmxObjectLayer(name, color, objects, opacity=1, visible=1,
                    position=(0, 0))
```

Los parámetros corresponden con los **atributos**:

- **position**
  Tupla que indica la posición de la capa de objetos.
- **name**
  Nombre de la capa de objetos.
- **color**
  Color usado para representar los objetos de la capa.
- **opacity**
  Número real entre 0 y 1 que indica la opacidad de la capa. Por defecto vale 1, totalmemte opaco.
- **visible**
  Entero que indica si la capa es mostrada (1, valor por defecto) o no (0).
- **objects**
  Lista que contiene los objetos (instancias de TmxObject) que componen la capa.

Los **métodos** son:

- **collide**(rect, propname)
  Busca todos los objetos de la capa que se toquen con el rectángulo rect y que tengan configurada la propiedad propname. Se devuelve una lista.
- **find_cells**(**requirements)
  Busca todos los objetos de la capa que tengan configuradas las propiedades dadas por requirements. Se devuelve una lista.
- **get_at**(x, y)
  Devuelve el primer objeto encontrado en las coordenadas (x, y). En caso de no haber ninguno devuelve None.

- **get_in_region**(left, bottom, right, top)

  Devuelve una lista de objetos TMX que superpone cualquier punto interior del rectángulo indicado por los puntos (left, bottom) y (right, top).

- **match**(**properties)

  Devuelve una lista de objetos TMX que tengan unas determinadas propiedades configuradas con unos determinados valores.

Apéndice C

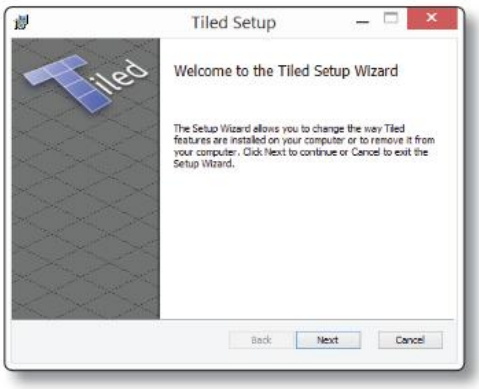
TILED MAP EDITOR

Los **mapas de baldosas** (tile maps) serán parte fundamental en nuestros juegos 2D. Son cuadrículas 2D donde cada celda puede albergar una imagen (o parte de ella). Tendremos otra cuadrícula 2D llamada **conjunto de patrones** a partir de cuyos bloques crearemos el mapa.

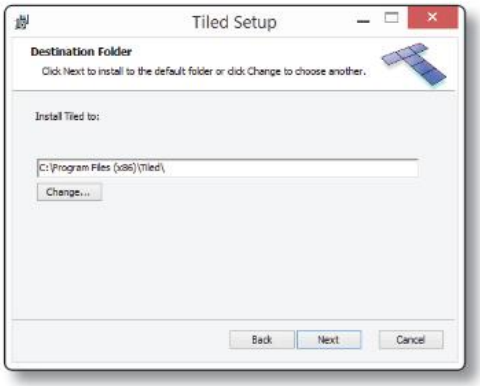
Para la construcción de mapas de baldosas he usado en el libro el programa Tiled Map Editor (Tiled de forma abreviada). La última versión[163] se puede descargar desde la siguiente dirección web:

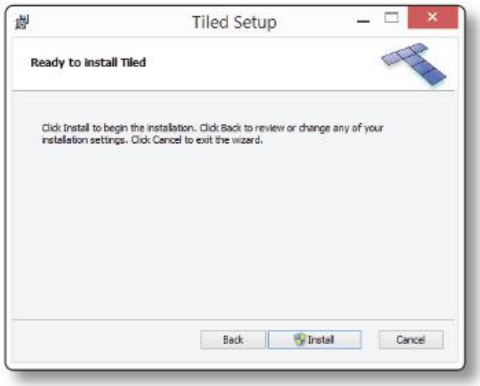
https://www.mapeditor.org/

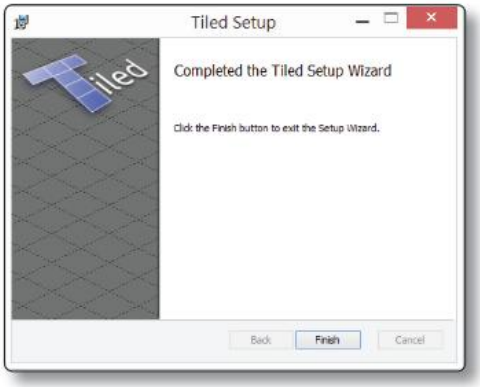
En el libro se ha usado la versión 1.1.2. En el material adicional del libro se podrá encontrar el fichero **Tiled-1.1.2-win32.msi**, cuya ejecución nos permitirá instalarla.



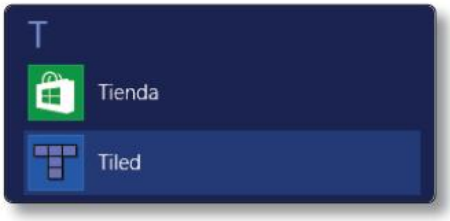
163 En el momento de escribir estas líneas (Octubre de 2018) es la 1.2.







Una vez instalado buscaremos Tiled en las aplicaciones de Windows y lo ejecutaremos:



Desplazaremos su icono en la barra de tareas hasta colocarlo en el lugar que queramos, y lo anclaremos:

Haré a continuación un breve repaso de algunos conceptos fundamentales que debemos saber para trabajar con él y su uso sencillo para nuestros propósitos, sin pretender profundizar en demasía. Al ser un programa bastante intuitivo, se anima al lector a probar todas las herramientas que se vayan comentando.

Ya sabemos qué son los mapas de baldosas y los conjuntos de patrones. Un fichero TMX[164] está compuesto de varias **capas**, cada una de ellas en un nivel, que pueden ser de varios tipos:

- De patrones: alberga un mapa de baldosas.
- De objetos: contiene objetos geométricos de distintos tipos, como rectángulos, elipses, líneas o polígonos.
- De imagen: en ella tendremos una imagen.

Podemos hacer visible u ocultar cada una de las capas, además de variar su nivel de opacidad.

Tenemos la opción de cambiar de nombre a las capas, así como de configurar varios de sus atributos por defecto y crear unos nuevos personalizados.

Los elementos de la capa de objetos podrán a su vez visualizarse/ocultarse de forma individual. Tendrán también una serie de atributos, así como la opción de añadir nuevos.

Nadas más ejecutar Tiled aparecerá lo siguiente en el centro de la ventana:

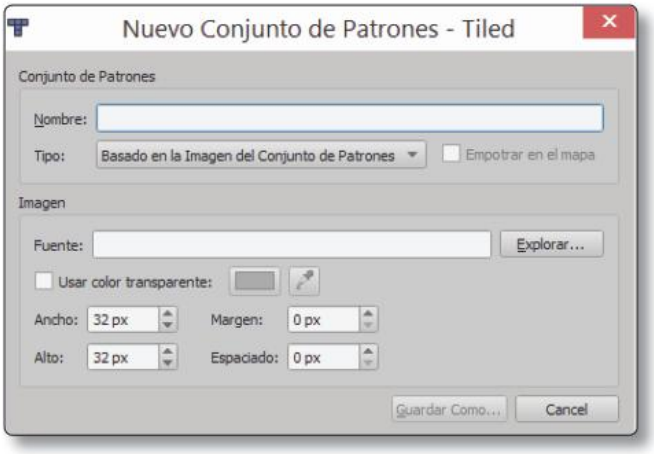
Podemos crear un nuevo mapa[165] o abrir un archivo ya creado. Nosotros empezaremos creando el conjunto de patrones. Al hacer clic sobre "Nuevo Conjunto de Patrones" aparecerá la siguiente pantalla[166]:

---

164 Tile Map XML, es el formato XML que usa Tiled para almacenar los mapas.

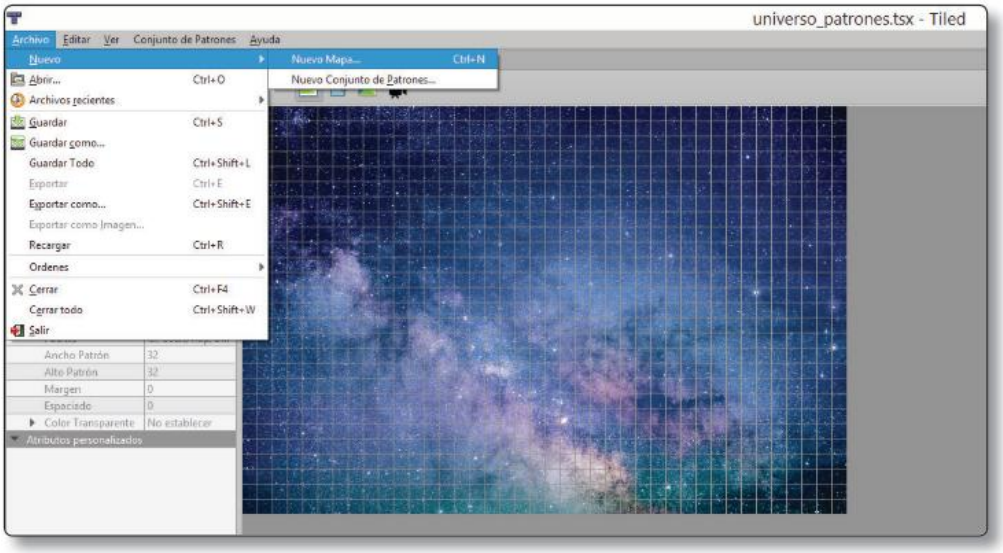
165 Posteriormente nos dará la posibilidad de crear un nuevo conjunto de patrones, como veremos más adelante.

166 Si en las casillas Ancho y Alto aparecen otros valores, cambiarlos por los indicados.



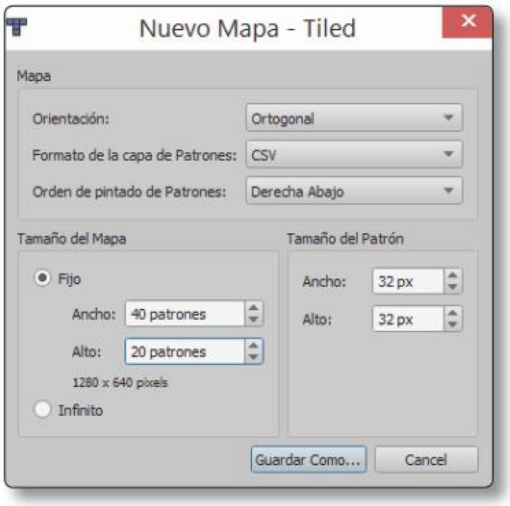
Pulsando "Explorar..." y buscando (haciendo doble clic sobre él cuando lo encontremos) **foto_universo.jpg** en nuestra carpeta tendremos la opción de hacer clic en "Guardar como..." y crear (de nuevo en nuestra carpeta) el fichero **universo_patrones.tsx** tras la pulsación del botón "Guardar". Nos aparecerá el conjunto de patrones[167].

Ahora crearemos un nuevo mapa, haciendo clic en la opción indicada a continuación:

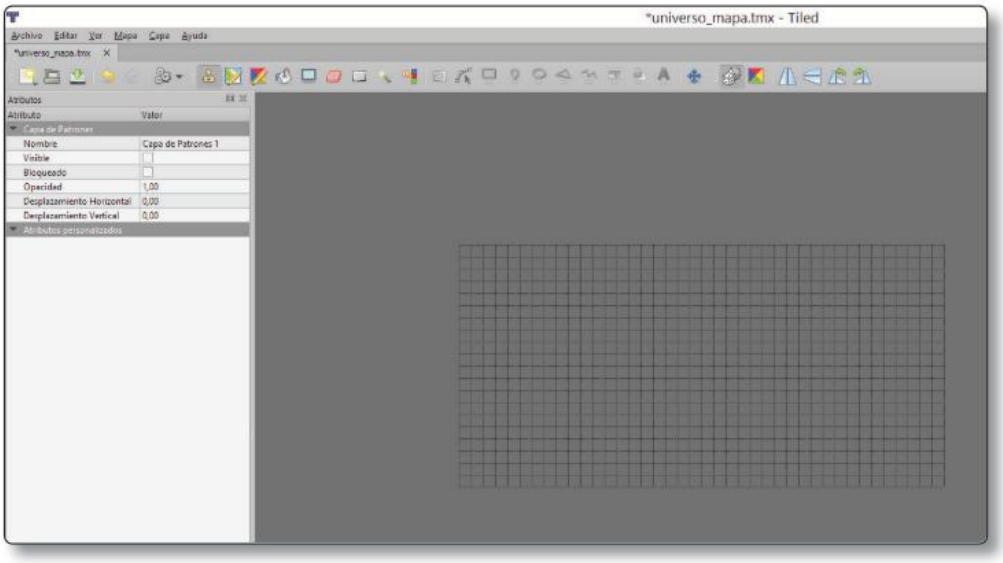


167 Si el tamaño es muy grande cambiarlo mediante el indicador de porcentaje que aparece en la esquina inferior derecha. En mi caso lo he colocado al 50%.

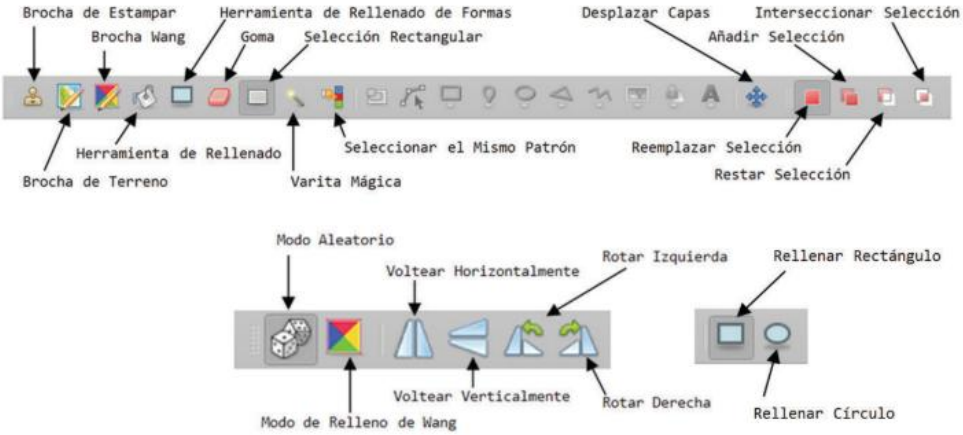
Nos aparecerá un cuadro de diálogo, que se configurará como se indica:



Haremos clic el "Guardar Como...", le pondremos de nombre universo_mapa y lo guardaremos en nuestra carpeta. Tendremos entonces en ella **universo_mapa.tmx**. Una imagen de la parte central e izquierda de la ventana será:



Por defecto tenemos una única capa de patrones del tamaño indicado, cuyos atributos visualizamos. Los botones que podemos tener inicialmente en la barra de herramientas son:



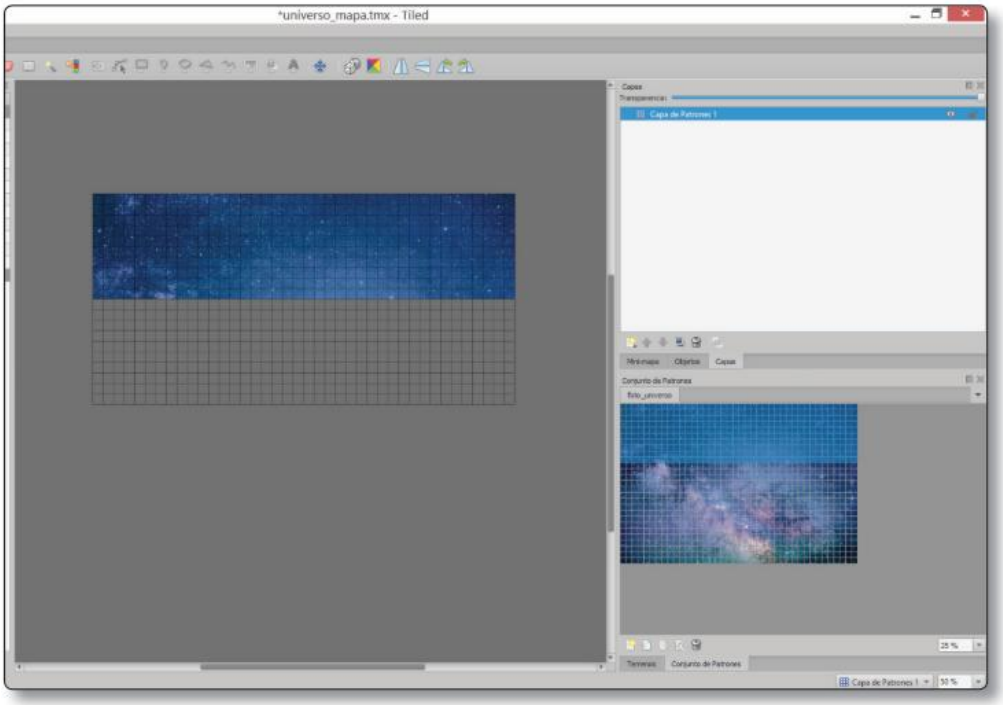
Una breve descripción de los elementos que usaremos[168] es la siguiente:

- Brocha de estampar: nos permite insertar elementos del conjunto de patrones en la cuadrícula de la capa.
- Herramienta de rellenado: mediante ella podremos rellenar de golpe amplias zonas (generalmente toda) de la cuadrícula.
- Herramienta de rellenado de formas: rellenamos la zona marcada con los patrones que tenemos en ese momento seleccionados[169].
- Goma: con ella lograremos eliminar el contenido de las celdas de la cuadrícula.
- Selección rectangular: selecciona una zona rectangular de la cuadrícula para, por ejemplo, borrar todo su contenido tras pulsar la tecla Supr.

168 Para conocer todos los elementos consultar el Manual de Usuario de Tiled.

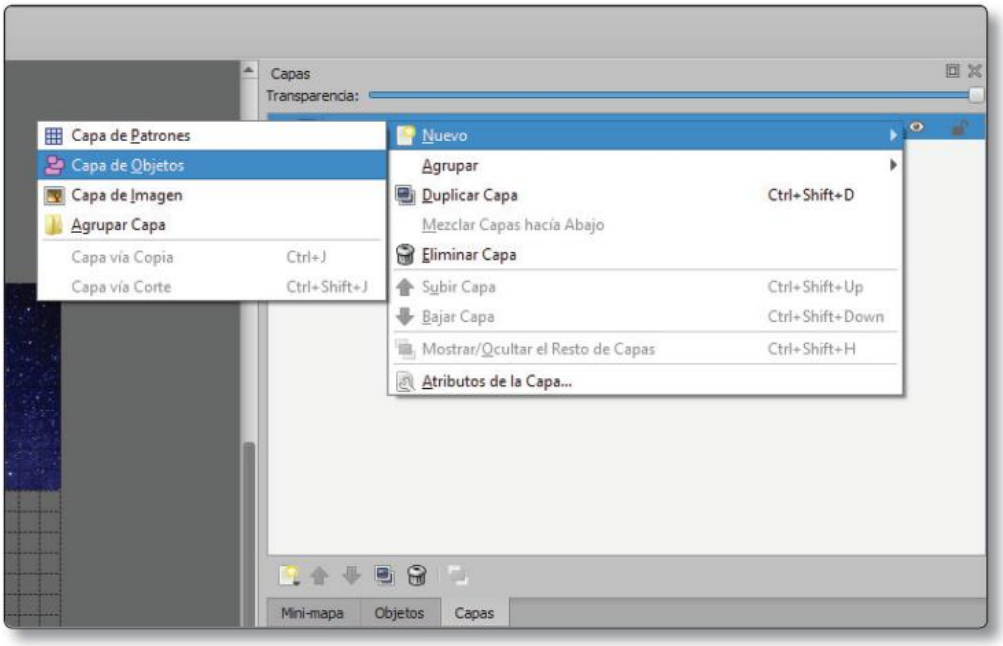
169 Para ello es necesario tener el botón "Modo Aleatorio" desactivado.

Si ahora (con el botón Brocha de Estampar pulsado) seleccionamos las 11 primeras filas del conjunto de patrones, las llevamos hacia la cuadrícula central y hacemos clic en ella, obtendremos lo siguiente:

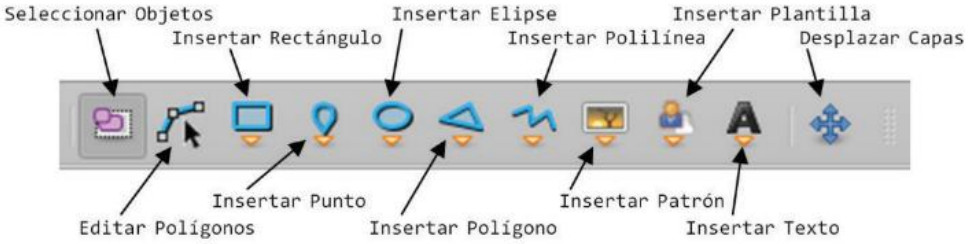


Tenemos en este momento en universo_mapa.tmx una capa de patrones que hemos rellenado parcialmente con celdas del conjunto de patrones universo_patrones.tsx. Para visualizar el resultado final que vayamos teniendo pulsaremos en la pestaña “Mini-mapa” colocada en la parte derecha de la ventana.

Añadiremos ahora una capa de objetos. Haremos clic con el botón derecho del ratón sobre la única capa que tenemos. Junto a varias opciones sobre las capas (agrupar, duplicar, mezclar, eliminar, subir/bajar de nivel, mostrar/ocultar y ver atributos de la capa) tenemos la opción de crear una nueva. Aparecen los tres tipos indicados, y seleccionamos “Capa de Objetos”:

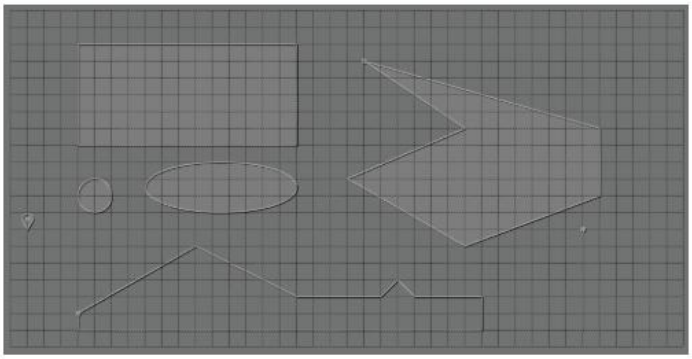


Se nos activarán los siguientes botones de la barra de herramientas:

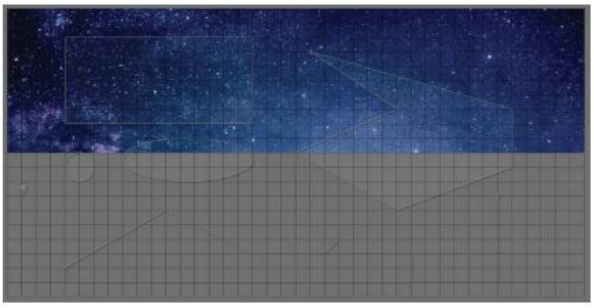


Al seleccionar objetos se nos permite hacerlo tanto individualmente como múltiple. Al editar polígonos, cuando tengamos un objeto de tipo polígono o polilínea, podremos trabajar sobre sus vértices. Observamos que también podemos introducir patrones y texto en una capa de objetos.

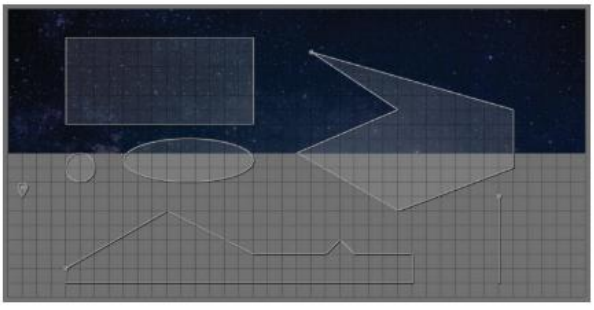
Insertaremos un rectángulo, un punto, un círculo[170], una elipse, un polígono, una línea[171] y una polilínea. Para visualizar mejor la capa de objetos desactivaremos la visualización de la capa de patrones:



Con la capa de patrones visualizada y seleccionada tendríamos:



Si seleccionamos la capa de objetos la apariencia varía:

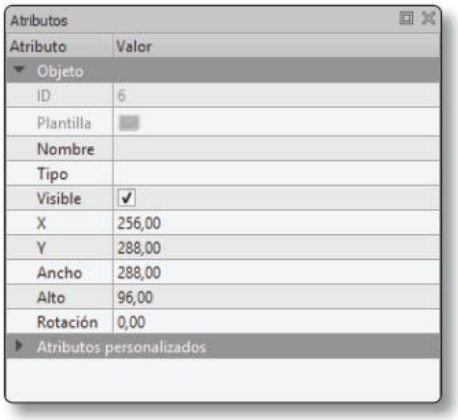


170 Considerado un caso particular de elipse.

171 Considerado un caso particular de polilínea.

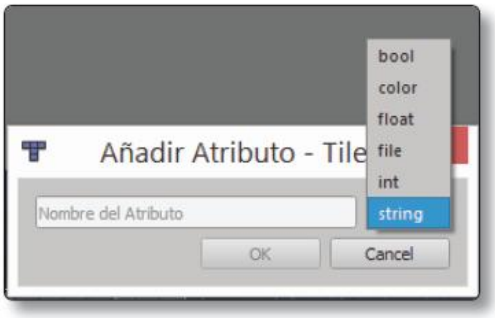
Al disponer de varias capas tenemos la posibilidad de colocar cada una de ellas en el nivel que queramos. Lo haremos haciendo clic con el botón derecho del ratón en la capa deseada y seleccionando Subir Capa/ Bajar Capa, o arrastrando la capa para colocarla en el lugar deseado[172].

Estando en la capa de objetos con la opción "Seleccionar Objetos" activada, si hacemos clic sobre la elipse la seleccionaremos y aparecerán sus atributos en la parte izquierda de la pantalla, entre ellos el nombre, tipo, si es o no visible, coordenadas x e y, ancho, alto y rotación:



En la parte inferior tendremos los botones para Añadir/Eliminar/Renombrar atributos personalizados:

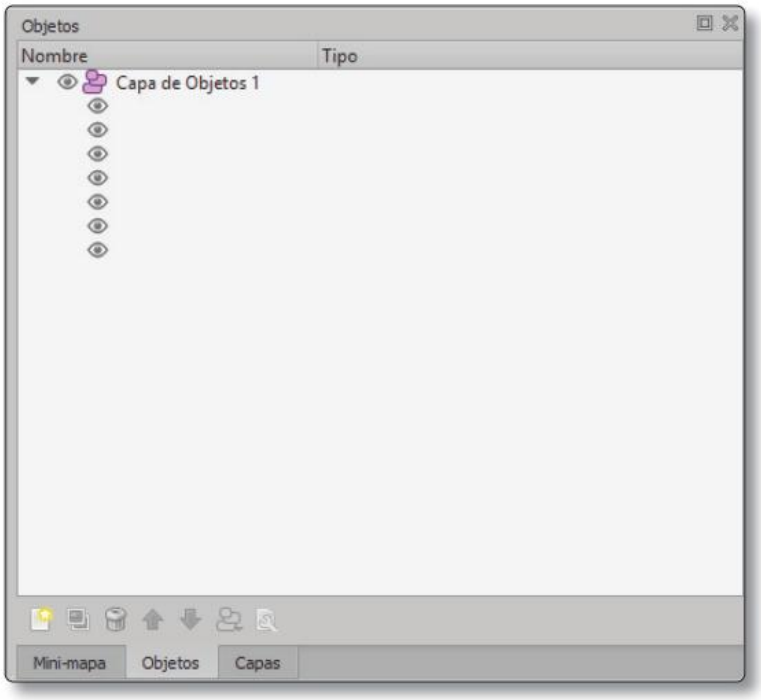
Si hacemos clic en el primero obtenemos:



172 El lector puede probar a intercambiar el nivel de las capas y ver su resultado final.

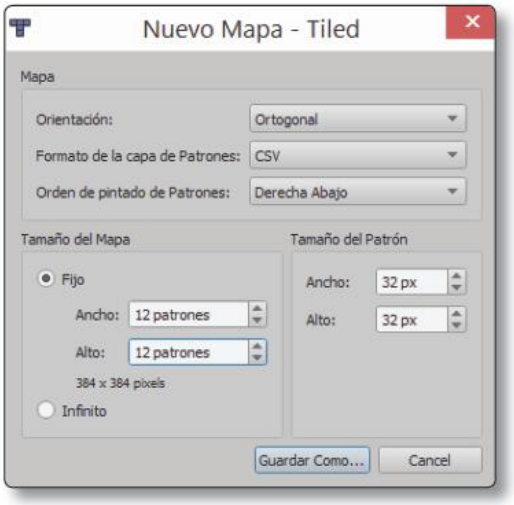
De esta manera añadiremos atributos a medida que usaremos posteriormente en el código Python.

Observamos que en el objeto elipse la casilla del nombre por defecto está vacía, algo que ocurre con el resto de objetos de la capa. Si hacemos clic sobre la pestaña “Objetos” en la parte derecha de la pantalla aparece:

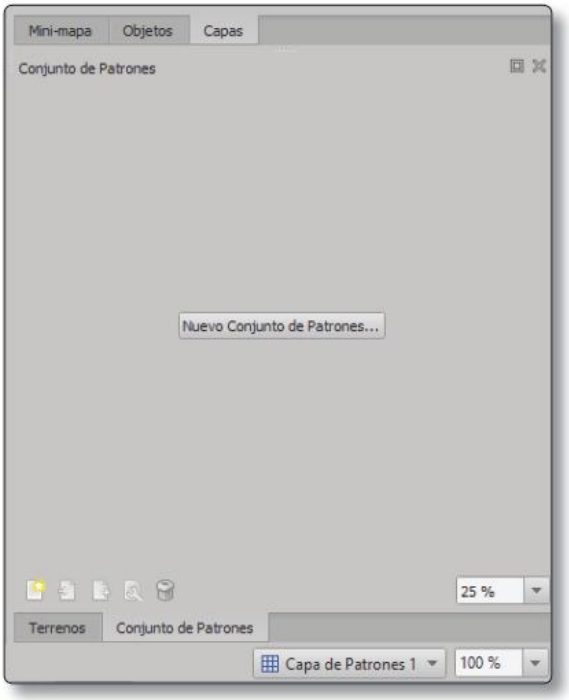


Se listan los objetos que tenemos, contando con la opción de visualizar/ocultar cada uno de ellos. Como no hemos colocado nombre a ninguno no aparecerá ninguno. Si ahora (teniendo aún seleccionada la elipse) rellenamos la casilla “Nombre”, observaremos cómo aparece en la lista. Habrá casos en los que será útil poner nombre a los objetos, otros que no tanto.

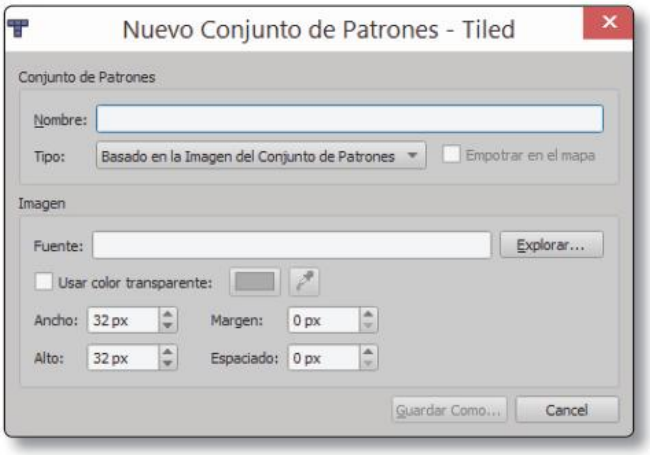
Crearemos a continuación el mapa usado en primeros ejemplos con código Python. En este caso al abrir Tiled pulsamos en el botón “Nuevo Mapa...” y rellenamos como se indica las casillas de la ventana que aparece:



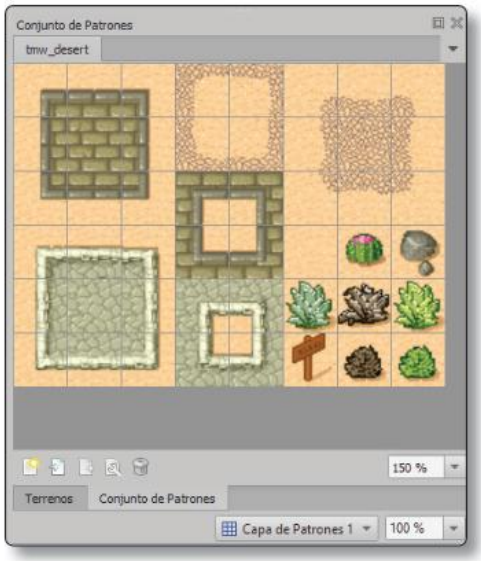
Lo guardamos en nuestra carpeta con nombre mapa1, teniendo finalmente **mapa1.tmx**. En la parte inferior derecha de la pantalla tendremos:



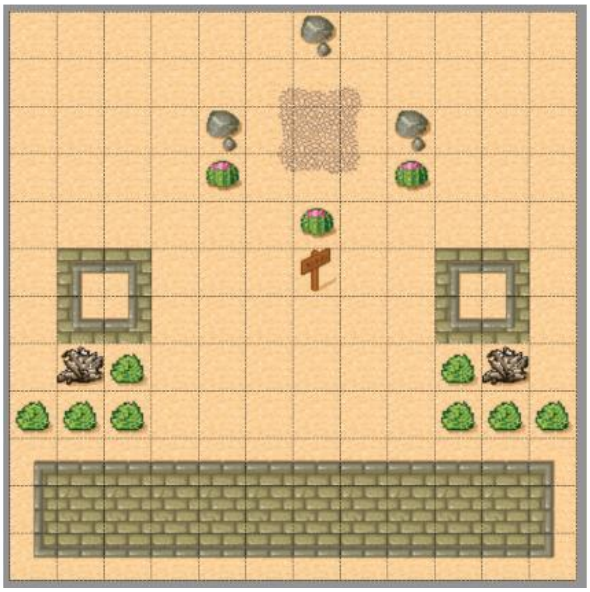
Hacemos clic en “Nuevo conjunto de Patrones...”:



Pulsando “Explorar...” y buscando (haciendo al encontrarlo doble clic sobre él) **tmw_desert.png** en nuestra carpeta tendremos la opción de hacer clic en “Guardar como...” y crear (de nuevo en nuestra carpeta) el fichero **tmw_desert.tsx** tras la pulsación del botón “Guardar”. Tiled nos crea una nueva pestaña representándolo, y en la correspondiente a mapa1.tmx nos aparece ya cargado:



Es entonces cuando creamos el mapa que vimos en el Tema 3.2. Mediante la herramienta de rellenado creamos inicialmente todas las baldosas lisas de color amarillo. Posteriormente, haciendo uso de la brocha de estampar, insertaremos los elementos que se indican, obteniendo el aspecto final:



Apéndice D

MISCELÁNEA

En este apéndice incluyo información sobre algunas clases, funciones y métodos de Python. Está pensado para explicar brevemente elementos que nos pueden ser muy útiles y que no son básicos, además de servir como consulta.

D.1 FUNCIONES LAMBDA, MAP() Y FILTER()

Las tres funciones forman parte del núcleo de Python. Originalmente también lo hacía la función reduce() que veremos más adelante, pero ahora se ha trasladado al módulo functools.

El uso de lambda, map(), filter() y reduce() está envuelto en polémica, ya que en algunos casos suele haber alternativas igual de potentes y más sencillas, como las listas de comprensión. Además, choca con uno de los lemas de Python, que insta a que haya solo una forma obvia de solucionar un problema. Incluso Guido van Rossum intentó eliminarlas en favor de otras alternativas, pero la enorme oposición con la que se encontró le hizo desistir. La función lambda fue añadida a Python a petición de los programadores con conocimientos de Lisp.

El operador (o la función) **lambda** es una forma de crear pequeñas funciones anónimas, es decir, funciones sin nombre. Se crean en el momento y lugar donde se necesitan. Se suele usar en combinación (como veremos) con map(), filter() y reduce().

La función lambda tiene la siguiente sintaxis:

```
lambda args : expr
```

En args tenemos los argumentos separados por comas, y expr es una expresión aritmética que hace uso de ellos. Podemos asignar la función a una variable para darle nombre. Dos sencillos ejemplos del uso de lambda son los siguientes:

```
>>> a = 10
>>> b = 7
>>> f1 = lambda x,y: x+y
>>> f1(a,b)
17
>>> f2 = lambda x,y: x**2 + y
>>> f2(a,b)
107
>>>
```

La función **map**() tiene la siguiente sintaxis:

```
map(f, sec)
```

Teniendo la función f y la secuencia sec, map() aplicará f a todos los elementos de sec. Antes de Python 3 se nos devolvía una lista con cada uno de sus componentes siendo la aplicación de f a los integrantes de sec. En Python 3 se nos devuelve un iterador, por lo que si necesitamos la salida en forma de lista deberemos usar list().

La función map() puede ser aplicada a más de una secuencia, teniendo en ese caso que ser éstas del mismo tamaño. La función f actuará sobre los elementos de las secuencias que tengan igual índice. Veamos ejemplos del uso de map():

```
>>> def f3(x):
...     return x+1
...
>>> map(f3, [4,8,2,9])
<map object at 0x0281A4B0>
>>> list(map(f3, [4,8,2,9]))
[5, 9, 3, 10]
>>> def f4(x,y):
...     return x+y
...
>>> list(map(f4, [4,8,2,9], [1,2,3,4]))
[5, 10, 5, 13]
>>> list(map(lambda x: x+1, [4,8,2,9]))
[5, 9, 3, 10]
>>> list(map(lambda x,y: x+y, [4,8,2,9], [1,2,3,4]))
[5, 10, 5, 13]
>>> c = [7,2,9]
>>> d = [2,8,3]
>>> e = [6,5,1]
>>> list(map(lambda x,y,z: x**3+3*y-z**2, c,d,e))
[313, 7, 737]
>>>
```

La función **filter**() tiene la siguiente sintaxis:

```
filter(f, sec)
```

Teniendo la función f y la secuencia sec, filter() nos devolverá un iterador con los elementos de sec que pasados a la función f devuelven True. Por lo tanto, f debe devolver un valor booleano. Veamos ejemplos:

```
>>> mi_secuencia = [5,23,54,12,7,27]
>>> list(filter(lambda x: x>20, mi_secuencia))
[23, 54, 27]
>>> list(filter(lambda x: x<20, mi_secuencia))
[5, 12, 7]
>>> list(filter(lambda x: x%2, mi_secuencia))
[5, 23, 7, 27]
>>> list(filter(lambda x: x%2-1, mi_secuencia))
[54, 12]
```

D.2 FUNCIONES REDUCE() Y PARTIAL()

Ambas están en el módulo **functools**, que trata sobre funciones de orden superior, es decir, funciones que actúan sobre (o devuelven) otras funciones.

La función **reduce**() tiene la siguiente sintaxis:

```
functools.reduce(f, sec)
```

El parámetro f es una función y sec una secuencia. Inicialmente se aplica la función f a los dos primeros elementos de sec, dando un resultado. Posteriormente se aplicará f con ese resultado y el siguiente valor de sec. Procederemos así consecutivamente hasta finalizar los elementos de la secuencia. Veamos ejemplos:

```
>>> from functools import reduce
>>> mi_secuencia = [2,5,1,3,7]
>>> reduce(lambda x,y: x+y, mi_secuencia)
18
>>> reduce(lambda x,y: x-y, mi_secuencia)
-14
>>> reduce(lambda x,y: x**2-2*y, mi_secuencia)
1322486
>>>
```

La función **partial**() tiene la siguiente sintaxis:

```
functools.partial(f, *args, **kwargs)
```

Nos devuelve un objeto que al ser llamado se comporta como si ejecutásemos la función f con los argumentos posicionales args y nombrados kargs. Si en la llamada a partial() aportamos más argumentos posicionales de los que tenemos configurados, éstos se añaden a args. Si aportamos más argumentos nombrados, éstos extienden (o anulan si el nombre coindice) los que ya tenemos. Un ejemplo de todo ello está en **ejemplo_funcion_partial.py**:

```
1
2 from functools import partial
3
4 def mi_funcion(x,y,z=10):
5     print(x+y+z)
6
7 def mi_funcion2(*args, **kwargs):
8     print(args, kwargs)
9
10 p1 = partial(mi_funcion, 1,5)
11 p2 = partial(mi_funcion, 1,5,20)
12 p3 = partial(mi_funcion, 2,10,z=100)
13 p1()
14 p2()
15 p3()
16
17 p4 = partial(mi_funcion2, 1,2,3,4,5,6, t=25)
18 p4()
19
```

Su salida es:

```
>>>
16
26
112
(1, 2, 3, 4, 5, 6) {'t': 25}
>>>
```

D.3 EVALUACIÓN Y EJECUCIÓN DE CÓDIGO. FUNCIONES EVAL() Y EXEC()

Python puede evaluar o ejecutar código proporcionado en forma de cadena. Para lo primero usaremos la función eval(), y para lo segundo la función exec().

La sintaxis de **eval**() es la siguiente:

```
eval(expr, globals=None, locals=None)
```

Mediante ella convertimos expresiones (pasadas en forma de cadena[173], representada por expr) en código objeto, lo ejecutamos y se nos devuelve el resultado. Recordemos que una expresión es una combinación de variables, funciones, operadores y valores que tras computarla se llega a un valor.

Los parámetros opcionales globals y locals son diccionarios de variables globales y locales que usaremos para la evaluación de la expresión. Si no se suministran, ésta se hará en el entorno que tengamos al llamar a eval().

La función eval() es una herramienta poderosa, pero también muy peligrosa ya que podría ejecutar código erróneo o malicioso, por lo que su uso a veces es cuestionado, sobre todo si tenemos alternativas más convenientes. Por ejemplo, las funciones int() o float() sólo convierten a números (enteros y reales, respectivamente), pero eso las hace más rápidas y seguras.

Ejemplos sencillos del uso de eval() son:

```
>>> a = 7
>>> b = 9
>>> eval('a+b')
16
>>> eval('(a*b)**2+25')
3994
>>> eval("print('La suma de {} y {} es {}'.format(a,b,a+b))")
La suma de 7 y 9 es 16
>>>
```

En Python 3 **exec**() es una función que nos va a permitir la ejecución dinámica de código, ya que compila[174] una cadena que lo contiene y la pasa al intérprete para ser ejecutada. Por lo general se ejecuta en el ámbito en el que estemos, pero le podemos pasar diccionarios personalizados de variables para que trabaje con ellas.

El uso de exec() tampoco está libre de polémica, ya que en muchos casos su uso puede ser reemplazado por soluciones más elegantes y menos peligrosas.

---

173 La función eval() también puede trabajar con código objeto, pero no consideraremos esa opción en el libro.

174 En Python el compilador bytecode es accesible en tiempo de ejecución.

La sintaxis de exec() es la siguiente:

```
exec(object[, globals[, locals]])
```

En ella tendremos los siguientes elementos:

- object: debe ser un código objeto o una cadena, siendo esta última opción la que hemos usado en el libro.
- globals: indica las variables globales que podrá manejar la función exec().
- locals: indica las variables locales que podrá manejar la función exec().

Si sólo adjuntamos globals, éste debe ser obligatoriamente un diccionario que valdrá tanto para las variables globales como para las locales.

Si sólo adjuntamos locals, podemos usar para ello cualquier objeto mapeado[175] (mapped).

Si se adjuntan tanto globals como locals, se usarán para las variables globales y locales, respectivamente.

Si ni globals ni locals son adjuntados en la ejecución de exec(), ésta se ejecuta en el ámbito de variables (globales y locales) donde nos encontremos. No obstante, si estamos dentro de una función el ámbito será un diccionario con una copia de las variables locales que tengamos en ese momento. Una cosa importantísima a resaltar es que, si en el código que ejecutamos en exec() se produce alguna modificación de estas variables locales, el cambio no quedará reflejado. Por ejemplo:

```
>>> def mi_funcion():
...     a = 10
...     exec("a += 5")
...     print(a)
...
>>> mi_funcion()
10
>>>
```

¿Como solucionaríamos esto si nos interesa el valor de esas modificaciones? Tendremos dos opciones:

1. Mediante el uso de la función locals():
2. Creando nuestro propio diccionario y pasándoselo a exec().

175 Un tipo de objeto especial en Python que incluye (aparte de otros) a los diccionarios.

En el primer caso, la función **locals**() nos devuelve un diccionario con las variables locales. Los cambios en este diccionario no afectan a las variables locales usadas en el intérprete. Podríamos por tanto conseguir mediante locals() este diccionario antes de la llamada a exec(), y posteriormente extraer de él los valores modificados:

```
>>> def mi_funcion2():
...     a = 10
...     var_loc = locals()
...     exec("a += 5")
...     print(var_loc['a'])
...
>>> mi_funcion2()
15
>>>
```

Es importante que trabajemos con el diccionario y no con la variable local definida.

Debemos tener cuidado con la ejecución de locals(), ya que cada vez que lo hacemos toma los valores actuales de las variables locales y sobrescribe el diccionario:

```
>>> def mi_funcion3():
...     a = 10
...     var_loc = locals()
...     exec("a += 5")
...     print(var_loc['a'])
...     locals()
...     print(var_loc['a'])
...
>>> mi_funcion3()
15
10
>>>
```

La alternativa a usar locals() es crear nuestro propio diccionario y pasárselo como argumento a la función exec(). Al no haberlo conseguido mediante locals(), su posible ejecución no nos afectará:

```
>>> def mi_funcion4():
...     a = 10
...     mi_var_loc = {'a': a}
...     exec("a += 5", {}, mi_var_loc)
...     print(mi_var_loc['a'])
...     locals()
...     print(mi_var_loc['a'])
...
>>> mi_funcion4()
15
15
>>>
```

Hacer notar que en la ejecución de exec() hemos pasado un diccionario vacío como argumento de variables globales y nuestro diccionario para el caso de las locales. De no haber incluido estos argumentos, la salida de mi_funcion4() habría sido un doble 10, por los motivos explicados anteriormente.

De la misma forma que hemos conseguido las variables locales mediante la función locals(), la función **globals**() nos devolverá un diccionario con las variables globales que tengamos en ese momento en el módulo (dentro de una función o método, éste es el módulo donde se define, no el módulo desde el que se llama). En ciertas ocasiones podrá ser útil su uso.

Como resumen crearé un pequeño programa (**ejemplo_exec.py**) con tres funciones que hacen uso de lo comentado hasta ahora:

```
1
2 def f1():
3     a = 1
4     var_locales = locals()
5     print("Variables locales antes de ejecutar exec() y tras ejecutar locals(): ", var_locales)
6     exec("b = a + 4\na += 1")
7     print("Variables locales después de ejecutar exec(): ", var_locales)
8     locals()
9     print("Variables locales después de ejecutar de nuevo locals(): ", var_locales)
10
11 def f2():
12     a = 1
13     var_locales = locals()
14     print("Variables locales antes de ejecutar exec(): ", var_locales)
15     exec("b = a + 4\na += 1", {}, var_locales)
16     print("Variables locales después de ejecutar exec(): ", var_locales)
17     locals()
18     print("Variables locales después de ejecutar de nuevo locals(): ", var_locales)
19
20 def f3():
21     a = 1
22     mis_loc = {'a': a}
23     print("Mis variables locales antes de ejecutar exec(): ", mis_loc)
24     exec("b = a + 4\na += 1", {}, mis_loc)
25     print("Mis variables locales después de ejecutar exec(): ", mis_loc)
26     locals()
27     print("Mis variables locales después de ejecutar locals(): ", mis_loc)
28
29
30 print("f1()" + 92*'-')
31 f1()
32 print("\nf2()" + 92*'-')
33 f2()
34 print("\nf3()" + 92*'-')
35 f3()
36
```

La salida es la siguiente:

```
>>>
f1()-------------------------------------------------------------------------------------------
Variables locales antes de ejecutar exec() y tras ejecutar locals():  {'a': 1}
Variables locales después de ejecutar exec():  {'b': 5, 'a': 2, 'var_locales': {...}}
Variables locales después de ejecutar de nuevo locals():  {'b': 5, 'a': 1, 'var_locales': {...}}

f2()-------------------------------------------------------------------------------------------
Variables locales antes de ejecutar exec():  {'a': 1}
Variables locales después de ejecutar exec():  {'b': 5, 'a': 2}
Variables locales después de ejecutar de nuevo locals():  {'b': 5, 'a': 1, 'var_locales': {...}}

f3()-------------------------------------------------------------------------------------------
Mis variables locales antes de ejecutar exec():  {'a': 1}
Mis variables locales después de ejecutar exec():  {'b': 5, 'a': 2}
Mis variables locales después de ejecutar locals():  {'b': 5, 'a': 2}
>>>
```

En f1() consigo mediante locals() las variables locales, ejecuto exec() sin argumentos y compruebo los efectos de una segunda llamada a locals(). Además podemos ver cómo analiza la cadena pasada como argumento a exec(), siendo un código de dos líneas donde creo una nueva variable b que se añade sin problemas al diccionario, desde el cual podemos acceder a ella. Recordar que intentarlo directamente por su nombre nos llevará a un error.

En f2() paso a exec() el diccionario conseguido mediante locals(), simplemente para comprobar que de cara a nuestras variables el resultado es el mismo que en f1().

En f3() creo mi propio diccionario, se lo paso a exec() en su ejecución y compruebo que la posterior ejecución de locals(), como es lógico, no le afecta para nada.

El uso de locals(), globals(), o de diccionarios personalizados para acceder a las modificaciones en variables al ejecutar exec() será cuestión de la situación concreta que se nos presente en el programa. En los casos que no importe una u otra elección, el lector elegirá con la que más cómodo se sienta.

D.4 MÉTODOS ESPECIALES O MÁGICOS

En Python podemos definir e implementar métodos (denominados **especiales** o **mágicos**) que serán llamados de forma automática cuando un operador sea aplicado a la instancia de una clase, permitiéndonos un uso más natural que llamando a los métodos por el nombre. Tenemos la posibilidad de **sobrescribir** el operador (redefiniendo el método que le corresponde) para adecuarlo al comportamiento que

queramos. Hay un método especial por cada operador, y su nombre está precedido y seguido de dos guiones bajos.

Si tenemos una expresión como a*b y a es una instancia de la clase C, Python chequeará la definición de clase de C y buscará el método __mul__()[176]. Si existe ejecutará a.__mul__(b), y si no generará un error.

Si tenemos a[2], Python buscará en la definición de la clase C el método __getitem__() y lo ejecutará, en caso de encontrarlo, con el argumento 2.

Hay muchos métodos especiales, con distintas funcionalidades. Mostraré una pequeña tabla con algunos de ellos:

Operadores	Métodos	Descripción
Operador índice		
a[key]	__getitem__(self, key)	Evalúa a[key].
a[key]=value	__setitem__(self, key, value)	Asigna valor a a[key].
del a[key]	__deltitem__(self, key)	Elimina a[key].
Binarios		
+	__add__(self, other)	Suma.
-	__sub__(self, other)	Resta.
*	__mul__(self, other)	Multiplicación.
//	__floordiv__(self, other)	División entera.
/	__truediv__(self, other)	División real.
%	__mod__(self, other)	Resto entero.
**	__pow__(self, other[, modulo])	Potenciación.
&	__and__(self, other)	Operador lógico Y .
^	__xor__(self, other)	Operador lógico O exclusiva.
|	__or__(self, other)	Operador lógico O.
De comparación		
<	__lt__(self, other)	Menor que.
<=	__le__(self, other)	Menor o igual que.
==	__eq__(self, other)	Igual que.
!=	__ne__(self, other)	Distinto de.
>=	__ge__(self, other)	Mayor o igual que.
>	__gt__(self, other)	Mayor que.

176 Método que corresponde con el operador *.

D.5 TIPOS FUNDAMENTALES EN PYTHON 3

A continuación presentaré dos tablas resumen, una sobre los tipos fundamentales de los que disponemos, y otra sobre los métodos de sus correspondientes clases:

		Cadenas	Listas	Tuplas	Conjuntos	Diccionarios
Operadores	[i]	Si	Si	Si	No	Si
	:	Si	Si	Si	No	No
	+	Si	Si	Si	No	No
	*	Si	Si	Si	No	No
	in/ not in	Si	Si	Si	Si	Si
	|	No	No	No	Si	No
	&	No	No	No	Si	No
	-	No	No	No	Si	No
	^	No	No	No	Si	No
	< <=	Si	Si	Si	Si	No
	> >=	Si	Si	Si	Si	No
	== !=	Si	Si	Si	Si	Si
Funciones	len	Si	Si	Si	Si	Si
	max	Si	Si	Si	Si	Si
	min	Si	Si	Si	Si	Si
	sum	No	Si	Si	Si	Si
Comandos especiales	del	No	No	No	No	Si
Recorrido mediante for		Si (directo y por índice)	Si (directo y por índice)	Si (directo y por índice)	Si (solo directo)	Si (solo mediante llave)

	Cadenas	Listas	Tuplas	Conjuntos	Diccionarios
Métodos	• capitalize() • center() • count() • endswith() • find() • format() • isalnum() • isalpha() • isdigit() • isidentifier() • islower() • isspace() • isupper() • ljust() • lower() • lstrip() • replace() • rfind() • rjust() • rstrip() • startswith() • strip() • swapcase() • tittle() • upper() • zfill()	• append() • clear() • copy() • count() • extend() • index() • insert() • pop() • remove() • reverse() • sort()	• count() • index()	• add() • clear() • copy() • difference() • difference_ update() • discard() • intersection() • intersection_ update() • isdisjoint() • issubset() • issuperset() • pop() • remove() • symmetric_ difference() • symmetric_ difference_ update() • union() • update()	• clear() • copy() • fromkeys() • get() • items() • keys() • pop() • popitem() • setdefault() • values()

D.5.1 Métodos de la clase str()

- **capitalize**():str

  Devuelve una cadena que es como la original pero con el primer carácter en mayúscula.

- **center**(ancho:int):str

  Devuelve una cadena igual a la original pero centrada en un ancho de caracteres que le indicamos mediante ancho.

- **count**(cadena:str):int

  Nos devuelve el número de apariciones de la cadena indicada, en el caso de que estuviese en nuestra cadena. En caso contrario devuelve 0.

- **endswith**(cadena:str):bool

  Nos devuelve True si nuestra cadena termina con la cadena que le indicamos y False en caso contrario.

- **find**(cadena:str):int

  Devuelve el índice[177] en el que aparece la cadena indicada. Si existen varias cadenas nos devuelve la primera que aparece (índice más bajo). Si no existe la cadena devuelve -1.

- **format**(*args,*kwargs)

  Formatea la cadena en base a unos determinados argumentos.

- **isalnum**():bool

  Devuelve True si todos los caracteres son números o alfabéticos, y False si no.

- **isalpha**():bool

  Devuelve True si todos los caracteres son alfabéticos (caracteres del alfabeto), y False si no.

- **isdigit**():bool

  Devuelve True si todos los caracteres son números, y False si no.

- **isidentifier**():bool

  Devuelve True si la cadena pudiera ser un identificador en Python, y False si no.

- **islower**():bool

  Devuelve True si la cadena tiene todos los caracteres (es necesario que haya al menos uno) en minúscula, y False en caso contrario. Los caracteres especiales (incluido el espacio en blanco no se consideran ni mayúscula ni minúscula).

- **isspace**():bool

  Devuelve True si la cadena está compuesta solo de espacios en blanco, y False en caso contrario.

- **isupper**():bool

  Devuelve True si la cadena tiene todos los caracteres (es necesario que haya al menos uno) en mayúscula, y False en caso contrario. Los caracteres especiales (incluido el espacio en blanco no se consideran ni mayúscula ni minúscula).

177 El índice recordemos que se inicia en el valor 0 para el primer carácter.

- **ljust**(ancho:int):str
  Devuelve una cadena igual a la original pero justificada a la izquierda en un ancho de caracteres que le indicamos mediante ancho.

- **lower**():str
  Devuelve una cadena que es como la original pero con todos los caracteres en minúscula.

- **lstrip**():str
  Devuelve una cadena igual a la original pero sin los posibles espacios en blanco[178] que pueda tener al comienzo.

- **replace**(a:str,d:str)**:**str
  Devuelve una cadena que es como la original pero con las posibles apariciones de la cadena a sustituidas por la cadena b.

- **rfind**(cadena:str):int
  Devuelve el índice en el que aparece la cadena indicada. Si existen varias cadenas nos devuelve la última que aparece (índice más alto). Si no existe la cadena devuelve -1.

- **rjust**(ancho:int):str
  Devuelve una cadena igual a la original pero justificada a la derecha en un ancho de caracteres que le indicamos mediante ancho.

- **rstrip**():str
  Devuelve una cadena igual a la original pero sin los posibles espacios en blanco que pueda tener al final.

- **startswith**(cadena:str):bool
  Nos devuelve True si nuestra cadena comienza con la cadena que le indicamos y False en caso contrario.

- **strip**():str
  Devuelve una cadena igual a la original pero sin los posibles espacios en blanco que pueda tener al comienzo o al final.

- **swapcase**():str
  Devuelve una cadena que es como la original pero con los caracteres en minúscula pasados a mayúscula y al revés.

178 Recordemos que lo que denominamos espacios en blanco incluyen no solo al carácter espacio en blanco sino también a los caracteres especiales tabulador('\t'), nueva línea ('\n') o retorno ('\r').

- **title**():str

  Devuelve una cadena que es como la original pero con el primer carácter de cada palabra en mayúscula.

- **upper**():str

  Devuelve una cadena que es como la original pero con todos los caracteres en mayúscula.

- **zfill**(ancho:int):str

  Devuelve una cadena numérica igual a la original pero de anchura de caracteres ancho, rellenando con ceros la parte de la izquierda que sobra. La cadena numérica nunca es truncada y puede incluir valores negativos.

D.5.2 Métodos de la clase list()

- **append**(x:object):None

  Añade el objeto x al final de la lista con la que estamos trabajando. Devuelve None.

- **clear**():None

  Elimina todos los elementos de la lista. Devuelve None.

- **copy**():list

  Devuelve una copia de la lista.

- **count**(x:object):int

  Devuelve el número de apariciones de un determinado valor en la lista. Si no está, nos devolverá 0.

- **extend**(l:iterable):None

  Añade un determinado iterable al final de la lista con la que estamos trabajando[179].

- **index**(x:object):int

  Devuelve el índice de la primera aparición del objeto x en la lista. Si no está en ella nos aparece un error de tipo ValueError.

179 El iterable a añadir podría perfectamente ser la misma lista con la que trabajamos, con lo que duplicaría su contenido.

- **insert**(i:int,x:object):None

  Inserta el objeto x en el índice i, desplazando los de índice posterior una unidad hacia la derecha. Si el valor de i supera el mayor índice de la lista, coge este último como valor de i.

- **pop**([i:int]):object

  Elimina y devuelve el objeto colocado en el índice i. Los elementos de índice superior pasan a tener un índice una unidad menor. Si no le pasásemos nada, por defecto elimina y devuelve el último objeto de la lista. En el caso de que el índice estuviese fuera de rango o la lista vacía, nos daría un error de tipo IndexError.

- **remove**(x:object):None

  Elimina la primera aparición del objeto x en la lista. Si no estuviese en ella obtendríamos un error de tipo IndexError. Devuelve None.

- **reverse**():None

  Invierte la lista, es decir, el primer valor pasa a ser el último, el segundo el antepenúltimo...y el último el primero. Devuelve None.

- **sort**([reverse=False]):None

  Ordena los elementos de la lista de forma ascendente por defecto. Si quisiésemos que fuese en orden descendente, colocaríamos reverse = True. Devuelve None.

D.5.3 Métodos de la clase tuple()

- **count**(x:object):int Devuelve el número de apariciones del objeto x en la tupla.

- **index**(x:object,[start,[stop]]):int Si el objeto x está en la tupla, devuelve el primer índice en el que aparece. Si no lo está, devuelve un error de tipo ValureError. Opcionalmente se pueden indicar tanto un índice de inicio como de final de la búsqueda.

D.5.4 Métodos de la clase set()[180]

- **add**(x:object):None

  Añade el objeto x al conjunto. Si ya está no tiene ningún efecto.

---

180 Consideraremos que los métodos se aplican a un objeto A de tipo conjunto.

- **clear**():None
  Elimina todos los objetos del conjunto.
- **copy**():set
  Devuelve una copia del conjunto A.
- **difference**(B:set):set
  Devuelve la diferencia de A y B.
- **difference_update**(B:set):set
  Actualiza el conjunto A con la diferencia de A y B.
- **discard**(x:object):None
  Si el objeto x está en el conjunto, lo elimina de él. Si no lo está, no hace nada.
- **intersection**(B:set):set
  Devuelve la intersección de A y B.
- **intersection_update**(B:set):set
  Actualiza el conjunto A con la intersección de A y B.
- **isdisjoint**(B:set):bool
  Devuelve True si los conjuntos tienen intersección nula. De lo contrario devuelve False.
- **issubset**(B:set):bool
  Devuelve True si A es un subconjunto[181] de B. De lo contrario devuelve False.
- **issuperset**(B:set):bool
  Devuelve True si A es un superconjunto[182] de B, de lo contrario devuelve False.
- **pop**():object
  Si el conjunto no está vacío, elimina de forma aleatoria uno de sus elemento, y lo devuelve. Si está vacío, devuelve un error de tipo KeyError.

181 Recordemos: Se dice que A es un subconjunto de B si todos los elementos de A están en B.

182 Recordemos: Se dice que A es un superconjunto de B si todos los elementos de B están en A.

- **remove**(x:object):None

  Si el objeto x está en el conjunto, lo elimina de él. Si no lo está, devuelve un error de tipo KeyError.

- **symmetric_difference**(B:set):set

  Devuelve la diferencia simétrica (u O exclusiva) de A y B.

- **symmetric_difference_update**(B:set):set

  Actualiza el conjunto A con la O exclusiva de A y B.

- **union**(B:set):set

  Devuelve la unión de A y B.

- **update**(B:set):set

  Actualiza el conjunto A con la unión de A y B.

D.5.5 Métodos de la clase dict()

- **clear**():None

  Elimina todas las entradas del diccionario.

- **copy**():dict

  Devuelve una copia de diccionario.

- **fromkeys**(i:object[,d:object]):dict

  Genera un nuevo diccionario con las llaves que tengamos en i. Los valores los rellena a None por defecto, salvo que tengamos el parámetro d, en cuyo caso los rellena con ese valor.

- **get**(i:object[,d:object]):object

  Devuelve el objeto diccionario[i] si el índice i está en diccionario. Si no está devuelve None, salvo que tengamos en parámetro opcional d, en cuyo cado devuelve d.

- **items**():set-like

  Devuelve un objeto parecido a un conjunto donde aparecen las entradas de diccionario en forma de tuplas (llave/dato).

- **keys**():set-like

  Devuelve un objeto parecido a un conjunto donde aparecen todas las llaves del diccionario.

- **pop**(i:object[,d:object]):object

  Si el índice i está en diccionario, elimina de él la entrada diccionario[i] y devuelve su valor. Similar al uso de del pero en este caso nos devuelve el valor eliminado. Si no lo está devuelve un error de tipo KeyError salvo que tengamos el parámetro opcional d, que es el que devuelve en ese caso.

- **popitem**():tuple

  Elimina de forma aleatoria una entrada del diccionario, y la devuelve en forma de tupla.

- **setdefault**(i:object[,d:object]):object

  Si i está en el diccionario, es equivalente a get. Si no lo está, asigna None a diccionario[i] si no tenemos d. Si lo tenemos, es d el valor asignado.

- **values**():set-like

  Devuelve un objeto parecido a un conjunto donde aparecen todas los valores del diccionario.

D.6 FUNCIONES INTERNAS DE PYTHON 3

Presentamos un subconjunto de las funciones incorporadas por defecto en Python, considerando las que pueden tener más utilidad directa para nuestros programas:

- **abs**(x)

  Calcula y devuelve el valor absoluto del valor numérico x.

- **bytes**([x])

  Crea y devuelve una nueva sencuencia de bytes a partir de un entero o secuencia.

- **chr**(x)

  Crea y devuelve una cadena conteniendo un carácter cuyo valor Unicode es el entero x.

- **dict**([cont])

  Crea y devuelve un nuevo diccionario. El parámetro cont (de contanier, contenedor) puede ser un diccionario o una secuencia de objetos inmutables.

- **float**([x])

  Convierte una cadena o un entero a un número de punto flotante, que nos devuelve.

- **hash**(obj)

  Crea y nos devuelve un valor hash para el objeto obj dado.

- **input**([prompt])

  Obtiene y devuelve una secuencia de caracteres dados por el usuario a través de la entrada estándar (el teclado).

- **int**(x)

  Convierte un número a un número entero, que nos devuelve.

- **isinstance**(obj,c)

  Nos devuelve booleano indicando si el objeto obj es de la clase de nombre (o tupla de nombres de clases) c.

- **issubclass**(c1,c2)

  Nos devuelve booleano indicando si la clase c1 es una subclase de la clase de nombre (o tupla de nombres de clases) c2.

- **len**(cont)

  Devuelve el número de elementos del contenedor cont.

- **list**([cont])

  Crea y nos devuelve una nueva lista. El parámetro cont es el contenedor cuyos elementos son usados para crear la nueva lista.

- **max**(a1[,a2, ...])

  Devuelve el mayor valor de una colección. Si solo damos un argumento (a1) debe ser un contenedor, y se devuelve el mayor elemento que contenga. Si hay varios argumentos se nos devuelve el mayor de ellos.

- **min**(a1[,a2, ...])

  Devuelve el menor valor de una colección. Si solo damos un argumento (a1) debe ser un contenedor, y se devuelve el menor elemento que contenga. Si hay varios argumentos se nos devuelve el menor de ellos.

- **open**(f,m)

  Abre un fichero de texto o binario de nombre f en el modo m.

- **ord**(c)
  Devuelve el valor Unicode para el carácter c.
- **print**(*a,**kwa)
  Imprime los argumentos en la salida (por defecto la salida estándar, la pantalla).
- **range**([i,]fin[,p])
  Crea y devuelve un contenedor de secuencias de valores enteros que puede ser usado en un for. La secuencia empieza en i hasta fin-1 con paso p.
- **round**(v[,n_d])
  Redondea un valor numérico dado por v al entero más cercano, o (si se proporciona n_d) a un valor con número de decimales dado por n_d. Se nos devuelve el redondeo.
- **set**([cont])
  Crea y nos devuelve un nuevo conjunto. El parámetro cont es el contenedor cuyos elementos son usados para crear el nuevo conjunto.
- **sorted**(cont)
  Crea y devuelve una lista ordenada (por defecto en orden ascendente) a partir del contenedor cont.
- **str**(obj)
  Convierte un objeto a una cadena, y nos la devuelve.
- **sum**(cont)
  Calcula (y nos devuelve) la suma de los elementos del contenedor de números cont.
- **super**()
  Nos devuelve un objeto que, al llamar a uno de sus métodos, se llama al método de su superclase.
- **tuple**([cont])
  Crea y devuelve una nueva tupla. El parámetro cont es el contenedor cuyos elementos son usados para crear la nueva tupla.

D.7 LIBRERÍA ESTÁNDAR DE PYTHON 3

Se listarán varias funciones de algunos de los módulos que componen la librería estándar de Python.

D.7.1 Módulo os

- os.**chdir**(path)
  Cambia el actual directorio de trabajo al indicado mediante path.
- os.**get_exec_path**()
  Devuelve la lista de directorios en los que se buscará al ejecutar.
- os.**getcwd**()
  Nos devuelve el directorio de trabajo actual (current work directory).
- os.**getpid**()
  Nos devuelve el identificador del proceso actual.
- os.**getppid**()
  Nos devuelve el identificador del padre del proceso actual.
- os.**listdir**(path='.')
  Nos devuelve una lista con los nombres de las entradas del direccorio indicado por path.
- os.**mkdir**(path,*a,**kwa)
  Crea un directorio. En path podemos indicar (en forma de cadena) solo el nombre (lo creará en el directorio activo) o la ruta completa.
- os.**name**
  Nos indica el sistema operativo. Como posibles valores tendremos 'posix', 'nt', 'mac','os2', 'ce' y 'java'.
- os.**remove**(path,*a,**kwa)
  Elimina un fichero indicado mediante path. En path podemos indicar (en forma de cadena) solo el nombre del fichero o su ruta completa, siempre incluyendo la extensión.
- os.**rename**(e1,e2,*a,**kwa)
  Renombra el fichero o directorio e1 como e2.
- os.**replace**(e1,e2,*a,**kwa)
  Renombra el fichero o directorio e1 como e2, sobrescribiendo e2.
- os.**rmdir**(path,*a,**kwa)
  Elimina el directorio vacío path.

- os.**startfile**(path[, operation])

  Arranca un fichero con la aplicación asociada. Si operation no se indica (o es 'open') se abre como si hiciésemos doble clic sobre él desde Windows. El parámetro operation puede tener un valor que indica qué hacer con el fichero o directorio. Puede ser 'print'(imprime) o 'edit' (edita) para ficheros, y 'explore'(explora) y 'find'(busca) para directorios.

- os.**strerror**(code)

  Nos devuelve el mensaje de error correspondiente al código de error code.

- os.**system**(command)

  Ejecuta el comando en forma de cadena command.

D.7.2 Módulo os.path[183]

- os.path.**exists**(path)

  Devuelve booleano indicando si el fichero o directorio indicado por path existe o no.

- os.path.**getsize**(path)

  Devuelve el tamaño en bytes del fichero indicado por path.

- os.path.**isdir**(path)

  Devuelve booleano indicando si la dirección indicada por path corresponde a un directorio.

- os.path.**isfile**(path)

  Devuelve booleano indicando si la dirección indicada por path corresponde a un fichero.

- os.path.**samefile**(p1, p2)

  Devuelve booleano indicando si p1 y p2 se refieren al mismo fichero o directorio.

- os.path.**sameopenfile**(f1,f2)

  Devuelve booleano indicado si los descriptores de fichero f1 y f2 se refieren al mismo fichero.

183 El parámetro path, p1, p2 deberá ser una cadena conteniendo la dirección absoluta o relativa del fichero o directorio.

- os.**path.split**(path)

  Nos devuelve una tupla con la dirección indicada por path dividida en dos. Por una parte el último elemento y por otra el resto.

D.7.3 Módulo sys

- sys.**argv**

  Variable que referencia la lista de los argumentos en línea de comandos pasados a un script de Python. arg[0] es el nombre (con dirección completa) del script.

- sys.**builtin_module_names**

  Variable que referencia una tupla de cadenas conlos nombres de todos los módulos compilados en nuestro intérprete de Python.

- sys.**exc_info**()

  Función que nos devuelve una tupla de tres valores que nos dan información sobre la excepción que está siendo manejada en ese momento.

- sys.**exec_prefix**

  Es una cadena que nos indica el directorio donde están los ficheros de nuestra instalación Python.

- sys.**executable**

  Una cadena que nos indica la dirección absoluta del ejecutable Python que estemos usando.

- sys.**exit**([arg])

  Salimos del intérprete Python. El parámetro arg puede ser una cadena o un entero que representa un código de salida. En ambos casos se representa por pantalla.

- sys.**getrefcount**(obj)

  Devuelve el número de referencias al objeto obj.

- sys.**getsizeof**(obj)

  Nos devuelve el tamaño del objeto obj en bytes.

- sys.**getwindowsversion**()

  Devuelve un objeto que nos describe la versión de Windows que estamos usando.

- sys.**implementation**
  Devuelve un objeto que contiene información sobre la implementación de Python que tengamos.

- sys.**modules**
  Es un diccionario con los módulos que han sido cargados en nuestro intérprete de Python.

- sys.**path**
  Una lista de cadenas especificando las rutas de búsqueda para módulos. Se inicializa desde la variable de entorno PYTHONPATH.

- sys.**platform**
  Nos indica en forma de cadena la plataforma en la que estamos.

- sys.**prefix**
  Una cadena que nos indica el nombre del directorio donde se ha instalado Python.

- sys.**version**
  Cadena que contiene los números de versión de intérprete Python y alguna información adicional.

- sys.**version_info**
  Tupla que contiene los cinco números de versión del intérprete Python.

D.7.4 Módulo random

- random.**choice**(sec)
  Devuelve un elemento individual aleatorio incluido en la secuencia sec.

- random.**randint**(a,b)
  Devuelve un número entero aleatorio entre los números enteros a y b (ambos inclusive).

- random.**random**()
  Devuelve un número real aleatorio en el rango [0.0, 1.0),

- random.**randrange**([c,]f[,p])
  Devuelve un número entero aleatorio entre los números enteros c (de "comienzo") y f (de "final") excluyendo este último, con paso opcional p (valor por defecto 1). Si no se proporciona c se comienza en 0.

- random.**sample**(sec, k)
  Devuelve una lista de tamaño k de elementos únicos elegidos en la secuencia sec. Usado para muestras aleatorias sin reemplazo.

- random.**seed**(a=None,version=2)
  Inicializa el generador aleatorio de números.

- random.**shuffle**(sec)
  Desordena la secuencia sec.

- random.**uniform**(a,b)
  Devuelve un número real aleatorio entre los números reales a y b.

D.7.5 Módulo math

- math.**acos**(x)
  Devuelve el arcocoseno de x, en radianes.

- math.**acosh**(x)
  Devuelve el arcocoseno hiperbólico de x, en radianes.

- math.**asin**(x)
  Devuelve el arcoseno de x, en radianes.

- math.**asinh**(x)
  Devuelve el arcoseno hiperbólico de x, en radianes.

- math.**atan**(x)
  Devuelve la arcotangente de x, en radianes.

- math.**atan2**(y,x)
  Devuelve la arcotangente de y/x, en radianes. El resultado está entre -pi y pi.

- math.**atanh**(x)
  Devuelve la arcotangente hiperbólica de x, en radianes.

- math.**ceil**(x)
  Devuelve el techo de x, el menor entero mayor o igual que x.

- math.**cos**(x)
  Devuelve el coseno de x, en radianes.

- math.**cosh**(x)
  Devuelve el coseno hiperbólico de x, en radianes.
- math.**degrees**(x)
  Convierte el ángulo x de radianes a grados.
- math.**e**
  La constante matemática e = 2.718281..., con la precisión disponible.
- math.**exp**(x)
  Devuelve e**x.
- math.**fabs**(x)
  Devuelve el valor absoluto de x.
- math.**factorial**(x)
  Devuelve el factorial de x.
- math.**floor**(x)
  Devuelve el suelo de x, el mayor entero menor o igual que x.
- math.**fmod**(x,y)
  Devuelve fmod(x,y) como está definida en la librería de C.
- math.**frexp**(x)
  Devuelve en forma de tupla la mantisa y el exponente de x.
- math.**fsum**(iter)
  Devuelve una precisa suma de valores en punto flotante del iterable iter.
- math.**hypot**(x,y)
  Devuelve la distancia euclídea (sqrt(x*x + y*y)).
- math.**isfinite**(x)
  Devuelve True si x no es ni infinito ni NaN[184], y False en caso contrario.
- math.**isinf**(x)
  Devuelve True si x es infinito, y False si no.

184 No es un número (Not a Number).

- math.**isnan**(x)
  Devuelve True si x no es un número, y False en caso contrario.
- math.**ldexp**(x, i)
  Devuelve x*(2**i).
- math.**log**(x[,base])
  Devuelve el logaritmo de x en la base dada. Si no se proporciona la base devuelve el logaritmo natural (en base e).
- math.**log10**(x)
  Devuelve el logaritmo en base 10 de x.
- math.**modf**(x)
  Devuelve tupla con la parte entera y la decimal de x.
- math.**pi**
  La constante matemática pi = 3.141592..., con la precisión disponible.
- math.**pow**(x, y)
  Devuelve x elevado a y.
- math.**radians**(x) Convierte el ángulo x de grados a radianes.
- math.**sin**(x)
  Devuelve el seno de x, en radianes.
- math.**sinh**(x)
  Devuelve el seno hiperbólico de x, en radianes.
- math.**sqrt**(x)
  Devuelve la raíz cuadrada de x.
- math.**tan**(x)
  Devuelve la tangente de x, en radianes.
- math.**tanh**(x)
  Devuelve la tangente hiperbólica de x, en radianes.
- math.**trunc**(x)
  Devuelve el valor real x truncado a un entero.

D.7.6 Módulo time

- time.**localtime**([seg])

  Convierte un tiempo en segundos (seg) desde el epoch[185] en una tupla con información de la fecha y hora a la que equivalen. Si no pasamos seg se indica la fecha y hora actuales.

- time.**sleep**([seg])

  Suspende la ejecución durante un número de segundos (puede ser un número real) indicado por seg.

- time**.time**()

  Devuelve un número real con el número de segundos transcurridos desde epoch.

D.7.7 Módulo calendar

- calendar.**calendar**(a[,w,l,f,c])

  Nos devuelve en forma de cadena un calendario del año a. Con w y l damos la anchura y altura en caracteres para las columnas y filas. Con f y c indicamos las filas y columnas en las que se distribuirán los meses, por defecto 4 y 3 respectivamente.

- calendar.**firstweekday**()

  Nos devuelve el primer día de la semana que tengamos configurado para que aparezca el primero. Por defecto es el lunes (0).

- calendar.**isleap**(year)

  Nos devuelve booleano indicando si el año a es o no bisiesto.

- calendar.**leapdays**(a1,a2)

  Nos devuelve el número de años bisiestos entre los años a1 y a2.

- calenda**r.month**(a,m[,w,l])

  Nos devuelve en forma de cadena un calendario del mes m del año a. Con w y l damos la anchura y altura en aracteres para las columnas y filas.

185 El epoch es el punto donde empezamos a considerar el tiempo, el 1 de enero de 1970 a las 00:00 horas.

- calendar.**monthcalendar**(a,m)

  Nos devuelve una lista bidimensional con el calendario del mes. Los días fuera de él se representan como 0´s.

- calendar.**monthrange**(a,m)

  Nos devuelve tupla con el número del primer día y del número total de días que tiene el mes m del año a.

- calendar.**prcal**(a[,w,l,f,c])

  Imprime un calendario del año a. Con w y l damos la anchura y altura en caracteres para las columnas y filas. Con f y c indicamos las filas y columnas en las que se distribuirán los meses, por defecto 4 y 3 respectivamente.

- calendar.**prmonth**(a,m[,w,l])

  Imprime un calendario del mes m del año a. Con w y l damos la anchura y altura en caracteres para las columnas y filas.

- calendar.**setfirstweekday**(dia)

  Configura el primer día de la semana con el que se iniciará el calendario. Podrá ser de 0 (lunes) a 6 (domingo).

- calendar.**weekday**(a,m,d)

  Nos devuelve el día de la semana que es el correspondiente al año a, el mes m y el día d. Se devuelve entero de 0 (lunes) a 6 (domingo).

- calendar.**weekheade**r(n)

  Nos devuelve cadena con la cabecera de los días de la semana. Con n especificamos el número de caracteres que dedicamos al nombre de cada día.

BIBLIOGRAFÍA

- Python Game Programming By Example. Alejandro Rodas de Paz y Joseph Howse. Packt Publishing, 2015.
- Documentación oficial de cocos2d
- Documentación oficial de pyglet
- Documentación oficial de Tiled Map Editor.
- Python 3. Curso Práctico. Alberto Cuevas Álvarez. Ra-Ma, 2016.
- Aplicaciones gráficas con Python 3. Alberto Cuevas Álvarez. Ra-Ma, 2018.

MATERIAL ADICIONAL

El material adicional de este libro puede descargarlo en nuestro portal web: *http://www.ra-ma.es*.

Debe dirigirse a la ficha correspondiente a esta obra, dentro de la ficha encontrará el enlace para poder realizar la descarga. Dicha descarga consiste en un fichero ZIP con una contraseña de este tipo: XXX-XX-XXXX-XXX-X la cual se corresponde con el ISBN de este libro.

Podrá localizar el número de ISBN en la página IV (página de créditos). Para su correcta descompresión deberá introducir los dígitos y los guiones.

Cuando descomprima el fichero obtendrá los archivos que complementan al libro para que pueda continuar con su aprendizaje.

INFORMACIÓN ADICIONAL Y GARANTÍA

- RA-MA EDITORIAL garantiza que estos contenidos han sido sometidos a un riguroso control de calidad.
- Los archivos están libres de virus, para comprobarlo se han utilizado las últimas versiones de los antivirus líderes en el mercado.
- RA-MA EDITORIAL no se hace responsable de cualquier pérdida, daño o costes provocados por el uso incorrecto del contenido descargable.
- Este material es gratuito y se distribuye como contenido complementario al libro que ha adquirido, por lo que queda terminantemente prohibida su venta o distribución.

ÍNDICE ALFABÉTICO

Desarrollo de videojuegos 2D con Python

El desarrollo de videojuegos para PC, lejos de ser cosa de niños, puede llegar a unos niveles de complejidad y sofisticación muy elevados, máxime si trabajamos en 3D. El presente libro pretende acercar al lector el mundo de la programación de juegos 2D con **Python**. Para ello haremos uso de la librería **cocos2d** y del editor de mapas **Tiled**, con los cuales también podremos realizar presentaciones y aplicaciones gráficas interactivas. Apoyándonos en múltiples códigos de ejemplo desarrollaremos un juego de marcianos y otro de plataformas.

Desde **www.ra-ma.es** podrá descargar material adicional.

9 788499 647982

ra-ma.es

Ra-Ma®