



Taller de Objetos y Eventos con Pygame

2013

VídeoJuegos con Python

PyGroupUD

Proyecto Curricular de Ingeniería de Sistemas

Taller de Introducción a Pygame

Índice de contenido

Paradigma de programación orientada a objetos.....	3
Características de la Programación Orientada a Objetos.....	4
La abstracción	4
La encapsulación.....	5
La herencia	5
Polimorfismo.....	6
Otros Conceptos.....	7
¿Qué es un evento?	7
Pygame y los eventos.....	7
Ejercicio de objetos y eventos.....	9
Generando la ventana.....	9
Creando la primera clase.....	11
Trabajo a realizar.....	16
Bibliografía consultada:.....	16

Paradigma de programación orientada a objetos

Cuando se emprende la tarea de escribir un programa, tal vez el aspecto más importante de esta es la generación del modelo del problema que se pretende resolver. Se debe abstraer el problema con la finalidad de reducir sus detalles y trabajar con pocos elementos a la vez hasta lograr su representación completa.

La representación del problema y su solución hace uso de un marco general y un enfoque en particular que le brinda coherencia. Este marco general y enfoque particular lo define el paradigma de programación sobre el que se trabaje dicha representación.

Un paradigma de programación es una herramienta conceptual desde la que se puede analizar, representar y abordar los problemas con la finalidad de pasar del espacio del problema al espacio de la implementación de la solución.

Uno de los paradigmas de la rama imperativa es el paradigma de programación orientada a objetos el cual reúne las características de: **abstracción**, **encapsulación**, **herencia** y **polimorfismo**; y los conceptos básicos que las forman: **objetos**, **mensajes**, **clases**, **instancias** y **métodos**.

En el paradigma de programación orientada a objetos los programas se definen en términos de “clases de objetos” que se comunican entre sí mediante el envío de mensajes. Estas “clases de objetos” nos permiten crear nuevos tipos de datos que agrupan una serie de características y un conjunto de operaciones que los definen. El paradigma es una evolución de los paradigmas de la programación estructurada, procedural y modular, y se implementa en lenguajes como Java, Smalltalk, C++, PHP o Python.

Veamos algunos de los elementos conceptuales que se manejan dentro del paradigma:

- Los objetos son el eje conceptual sobre el cual se desarrolla el paradigma y se pueden definir como una encapsulación abstracta de información, junto con los métodos o procedimientos para manipularla. Un objeto contiene operaciones que definen su comportamiento y atributos que definen su estado entre las llamadas a las operaciones.
- Los objetos se definen en clases y estas equivalen a la generalización o abstracción de todas las características y comportamiento de un tipo específico de objetos.
- Cuando pasamos del plano de la abstracción donde se tienen a las clases al plano concreto donde se tienen los objetos estamos creando instancias de las clases de tal manera que una instancia se puede definir como la concreción de una clase.
- Los objetos tienen un mecanismo para interrelacionarse en tiempos de ejecución (comunicarse), este mecanismo es el paso de mensajes, donde un mensaje representa una acción a tomar por un determinado objeto.
- Cuando un objeto recibe un mensaje debe responder a esta acción de manera apropiada según las definiciones sobre las que este está construido, para esto las clases que los definen tienen implementados métodos, de tal manera podemos decir que un método consiste en la implementación en una clase de un protocolo de respuesta a los mensajes dirigidos a los objetos de este tipo. La respuesta a tales mensajes puede incluir el envío por el método de mensajes al propio objeto y aun a otros, también como el cambio del estado interno del objeto.

Características de la Programación Orientada a Objetos

La abstracción

Consiste en la generalización conceptual de un determinado conjunto de objetos y de sus atributos y propiedades, dejando en un segundo término los detalles concretos de cada objeto. ¿Qué se consigue con la abstracción? Bueno, básicamente pasar del plano material (cosas que se tocan) al plano mental (cosas que se piensan).

La encapsulación

Se refiere a la capacidad de agrupar y condensar en un entorno con límites bien-definidos distintos elementos. Cuando hablemos de encapsulación en general siempre nos referiremos, pues, a encapsulación abstracta. De manera informal, primero generalizamos (la abstracción) y luego decimos: la generalización está bien, pero dentro de un cierto orden: hay que poner límites (la encapsulación), y dentro de esos límites vamos a meter, a saco, todo lo relacionado con lo abstraído: no sólo datos, sino también métodos, comportamientos, etc.

Por un lado es en una abstracción, de acuerdo con la definición establecida anteriormente, donde se definen las propiedades y atributos genéricos de determinados objetos con características comunes. La Clase es, por otro lado, una encapsulación porque constituye una cápsula o saco que encierra y amalgama de forma clara tanto los datos de que constan los objetos como los procedimientos que permiten manipularlos. Las Clases se constituyen, así, en abstracciones encapsuladas.

La herencia

Se aplica sobre las clases. O sea, de alguna forma las clases pueden tener descendencia, y ésta heredará algunas características de las clases "padres". Si disponemos las clases con un formato de árbol genealógico, tendremos lo que se denomina una estructura jerarquizada de clases.

La OOP promueve en gran medida que las relaciones entre objetos se basen en construcciones jerárquicas. Esto es, las clases pueden heredar diferencialmente de otras clases (denominadas "superclases") determinadas características, mientras que, a la vez, pueden definir las suyas propias. Tales clases pasan, así, a denominarse "subclases" de aquéllas.

La herencia se implementa mediante un mecanismo que se denomina derivación de clases: las superclases pasan a llamarse **clases base**, mientras que las subclases se constituyen en **clases derivadas**. El mecanismo de herencia está fuertemente entroncado con la reutilización del código en OOP. Una clase derivada posibilita, el fácil uso de código ya creado en cualquiera de las clases base ya existentes.

El concepto de herencia constituye un estrato básico del paradigma de objetos, pero esto no significa que todas las relaciones entre clases en OOP deban ajustarse siempre a este modelo jerárquico. Es necesario establecer si la pretendida relación entre objetos es de pertenencia o de derivación. En una relación típica de pertenencia un objeto contiene al otro

Polimorfismo

Esta propiedad, como su mismo nombre sugiere múltiples formas, se refiere a la posibilidad de acceder a un variado rango de funciones distintas a través del mismo interfaz. O sea, que, en la práctica, un mismo identificador puede tener distintas formas (distintos cuerpos de función, distintos comportamientos) dependiendo, en general, del contexto en el que se halle inserto.

El polimorfismo se puede establecer mediante la sobrecarga de identificadores y operadores, la ligadura dinámica y las funciones virtuales. El término sobrecarga se refiere al uso del mismo identificador u operador en distintos contextos y con distintos significados.

La **sobrecarga de funciones** conduce a que un mismo nombre pueda representar distintas funciones con distinto tipo y número de argumentos. En el ámbito de la OOP, la sobrecarga de funciones equivale a que un mismo mensaje puede ser enviado a objetos de diferentes clases de forma que cada objeto respondería al mensaje apropiadamente.

La **sobrecarga de operadores** permite, por otro lado, el desarrollo de un código más coherente, como especialización de la sobrecarga de funciones, posibilitando la re-definición (para tipos de datos definidos-por-el-usuario) de las operaciones realizadas por éstos (+, -, *, >, etc.). Esto es, ocurre lo mismo que en la sobrecarga de funciones, pero aquí, en vez de identificadores de funciones, tenemos operadores.

Gracias a la **ligadura dinámica**, pueden invocarse operaciones en objetos obviando el tipo actual del éstos hasta el momento de la ejecución del código.

Otros Conceptos

- **Agregación:** Composición de un objeto por otros. Es una relación más débil que la que existe entre el atributo y el objeto al cual pertenece, y más fuerte que una asociación.
- **Concurrencia:** Propiedad que distingue un objeto *activo* de otro *inactivo*.
- **Persistencia:** Es la propiedad de un objeto cuya existencia trasciende el tiempo y/o el espacio (ej. el objeto continua existiendo luego de que su creador deja de existir / la ubicación de un objeto se mueve a un espacio de direcciones diferente de aquella donde fue creada).
- **Visibilidad:** capacidad de restringir el acceso a atributos y servicios de un objeto. Particularmente importante en el diseño e implementación. (ej.: público / protegido / privado)


¿Qué es un evento?

Cuando trabajamos con programación interactiva un evento es una acción de un periférico de entrada sobre nuestra aplicación, es decir, que durante la ejecución de la misma el usuario realiza una acción con el ratón, el teclado, etc.

El manejo de estos eventos nos permite interactuar con el usuario de nuestra aplicación brindando interacción real usuario – aplicación. Con Python y Pygame se pueden construir aplicaciones realmente entretenidas haciendo uso de esta característica.

Pygame y los eventos

Pygame posee múltiples funciones para manejar eventos, todos los mensajes que generan estos eventos los maneja mediante una cola de eventos mediante la librería `pygame.event`, las rutinas en este módulo ayudan a manejar esta cola de eventos.



Los eventos en Pygame son altamente dependientes del modulo *display* de pygame y si este no es inicializado la cola de eventos no podrá manejarse. Una forma simple de trabajar con la cola de eventos es verificar la existencia de los mismos.

Todos los eventos tienen un identificador de tipo, a continuación una lista de los tipos de eventos posibles:

- QUIT none
- ACTIVEEVENT gain, state
- KEYDOWN unicode, key, mod
- KEYUP key, mod
- MOUSEMOTION pos, rel, buttons
- MOUSEBUTTONUP pos, button
- MOUSEBUTTONDOWN pos, button
- JOYAXISMOTION joy, axis, value
- JOYBALLMOTION joy, ball, rel
- JOYHATMOTION joy, hat, value
- JOYBUTTONUP joy, button
- JOYBUTTONDOWN joy, button
- VIDEORESIZE size, w, h
- VIDEOEXPOSE none
- USEREVENT code

Y alguna rutinas para trabajar con estos:

- `pygame.event.pump` internally process pygame event handlers
- `pygame.event.get` get events from the queue
- `pygame.event.poll` get a single event from the queue
- `pygame.event.wait` wait for a single event from the queue
- `pygame.event.peek` test if event types are waiting on the queue

• <code>pygame.event.clear</code>	remove all events from the queue
• <code>pygame.event.event_name</code>	get the string name from an event id
• <code>pygame.event.set_blocked</code> queue	control which events are allowed on the queue
• <code>pygame.event.set_allowed</code> queue	control which events are allowed on the queue
• <code>pygame.event.get_blocked</code> queue	test if a type of event is blocked from the queue
• <code>pygame.event.set_grab</code> other applications	control the sharing of input devices with other applications
• <code>pygame.event.get_grab</code>	test if the program is sharing input devices
• <code>pygame.event.post</code>	place a new event on the queue
• <code>pygame.event.Event</code>	create a new event object

Una documentación completa sobre el modulo puede ser consultada en:
<http://www.pygame.org/docs/ref/event.html>

Ejercicio de objetos y eventos

Generando la ventana

Para nuestro ejercicio practico de esta semana vamos a crear un clon del juego de “invaders”, lo primero que haremos será crear una ventana con dimensiones de 800 x 600, nuestro script al igual que el del ejercicio anterior queda de la siguiente forma:

```
import sys, pygame
from pygame.locals import *

size = width, height = 800, 600
```

```
screen = pygame.display.set_mode(size)

def main():
    pygame.init()

    background_image = pygame.image.load("imagenes/space.png")
    background_rect = background_image.get_rect()

    pygame.display.set_caption( "invaders" )

    while 1:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                sys.exit()

        screen.blit(background_image, background_rect)

        pygame.display.update()
        pygame.time.delay(10)

if __name__ == '__main__':
    main()
```

Los recursos usados para este ejercicio son tres imagenes una de un fondo, una nave y un ufo, se sugieren los siguientes:

- nave: <https://www.dropbox.com/s/fg9ou6xal0i2a7a/ship.png>
- fondo: <https://www.dropbox.com/s/3klqyg45q1v9oqj/space.png>
- ufo: <https://www.dropbox.com/s/7u9daz7ylvnx745/ufo.png>

De este scrip no hay mucho que explicar ya que replica en parte lo trabajado en el primer taller y el resultado obtenido es el siguiente:

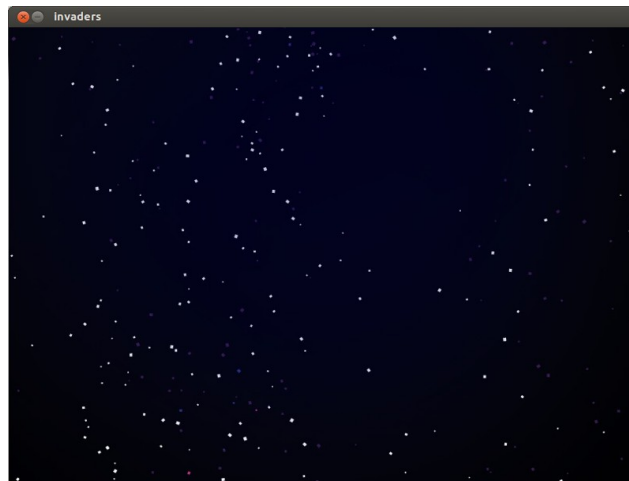


Ilustración 1: Ventana Inicial

Creando la primera clase

Con el script inicial funcionando ahora creemos nuestra primera clase, esta encapsulará la información y el comportamiento de nuestro héroe del juego y será la nave que se encargue de evitar que los invasores alcancen la parte inferior de la pantalla. Sin más que decir veamos cómo queda el código de nuestra clase “Ship”.

```
1. import pygame
2. from pygame.sprite import Sprite
3. from pygame.locals import *
4.
5. class Ship(Sprite):
6.     def __init__(self, cont_size):
7.         Sprite.__init__(self)
8.         self.puntos = 0
9.         self.vida = 100
10.        self.cont_size = cont_size
11.        self.image = pygame.image.load("imagenes/ship.png")
12.        self.rect = self.image.get_rect()
```

```
13.         self.rect.move_ip(cont_size[0]/2, cont_size[1]-35)
14.
15.     def update(self):
16.         teclas = pygame.key.get_pressed()
17.         if teclas[K_LEFT] and self.rect.left > 0:
18.             self.rect.x -= 10
19.         elif teclas[K_RIGHT] and self.rect.right < self.cont_size[0]:
20.             self.rect.x += 10
21.         elif teclas[K_UP]:
22.             pass
23.         elif teclas[K_DOWN]:
24.             pass
```

La definición de nuestra clase ocurre en la línea 5 en esta damos nombre a la clase y le indicamos que esta hereda todas las propiedades y comportamiento que tiene la clase “Sprite” de pygame.

```
class Ship(Sprite):
```

Esta clase “Sprite” nos permitirá un fácil manejo de los elementos que componen nuestro juego. En la programación orientada a objetos, implementada usando python, cuando creamos un objeto en base a una clase definida ocurren dos cosas: primero un llamado a la función `__new__` de la clase y luego un llamado a la función `__init__` de la misma, en nuestro script de la clase “Ship” creamos una función `__init__` para nuestros objetos de tipo Ship de tal manera que aseguremos la inicialización de los atributos requeridos para la funcionalidad de la nave en el juego y creamos una función `update` para actualizar los parámetros que modifican el estado de nuestra nave asociados a un evento, en este caso, a eventos del teclado.

Todas las funciones asociadas al comportamiento de un objeto deben trabajar por defecto con una referencia a este objeto, esta referencia en python se hace mediante el uso del parámetro `self` que aparece en nuestro script tanto en el método `__init__` como en el `update`, ya veremos que el llamado de esta función desde una referencia no es necesario pasar ningún valor para `self` ya que es simplemente una referencia al objeto que contiene la función.

En el método `__init__` inicialmente se hace un llamado al método `__init__` del Sprite, luego asignamos valores a los atributos puntos, vida, `cont_size` (que se refiere al tamaño del contenedor que dibujará nuestra nave), `image` y `rect` que como vimos en el taller anterior permiten manejar la imagen de nuestra nave. Luego mediante el método `move_ip` del `rect` de nuestra nave ubicamos la imagen en la parte inferior centrada del contenedor que dibujará nuestra nave.

En el método `update` lo primero que hacemos es capturar la tecla que generó el evento del teclado haciendo uso del método `get_pressed()` del módulo `key` de `pygame`, una vez capturado este evento preguntamos por la teclas asociadas y dependiendo de la flecha presionada moveremos a izquierda o derecha de la ventana nuestra nave. Las teclas las podemos identificar mediante las constantes `K_LEFT` para la flecha izquierda y `K_RIGHT` para la flecha derecha, en este movimiento controlamos que no se salga de la ventana la nave.

Ahora veamos las modificaciones realizadas al script inicial para que trabaje con un objeto del tipo `Ship`:

```
1. import sys, pygame
2. from pygame.locals import *
3. from ship import Ship
4.
5. size = width, height = 800, 600
6. screen = pygame.display.set_mode(size)
7.
8. def main():
9.     pygame.init()
10.
11.     background_image = pygame.image.load("imagenes/space.png")
12.     background_rect = background_image.get_rect()
13.
14.     pygame.display.set_caption( "asteroids" )
15.
16.     ship = Ship(size)
17.
18.     while 1:
19.         for event in pygame.event.get():
20.             if event.type == pygame.QUIT:
21.                 sys.exit()
```

```
22.  
23.     ship.update()  
24.     screen.blit(background_image, background_rect)  
25.     screen.blit(ship.image, ship.rect)  
26.  
27.     pygame.display.update()  
28.     pygame.time.delay(10)  
29.  
30. if __name__ == '__main__':  
31.     main()
```

En esta modificación creamos una instancia de Ship en la línea 16:

```
ship = Ship(size)
```

En este proceso de instanciación es llamado el método `__init__` que escribimos en nuestra clase Ship.

Luego en la línea 23 hacemos llamado al método `update` de nuestra nave:

```
ship.update()
```

En el que se capturan los eventos del teclado y se modifican los parámetros del área rectangular asociada a la imagen de la nave.

Y por último indicamos a nuestro script que pinte la imagen asociada a nuestra nave con el método `blit` del `screen`, esto se ve en la línea 25:

```
screen.blit(ship.image, ship.rect)
```

El resultado de estas modificaciones se ve en las siguientes imagenes.

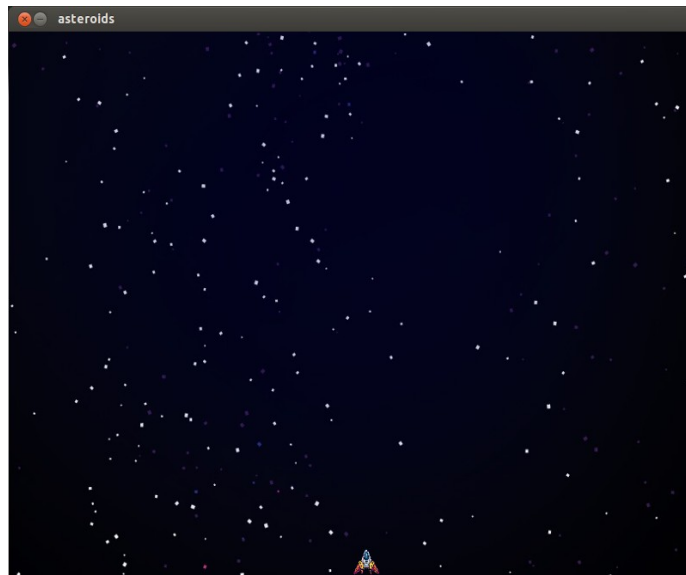


Ilustración 2: Ventana con la nave cargada

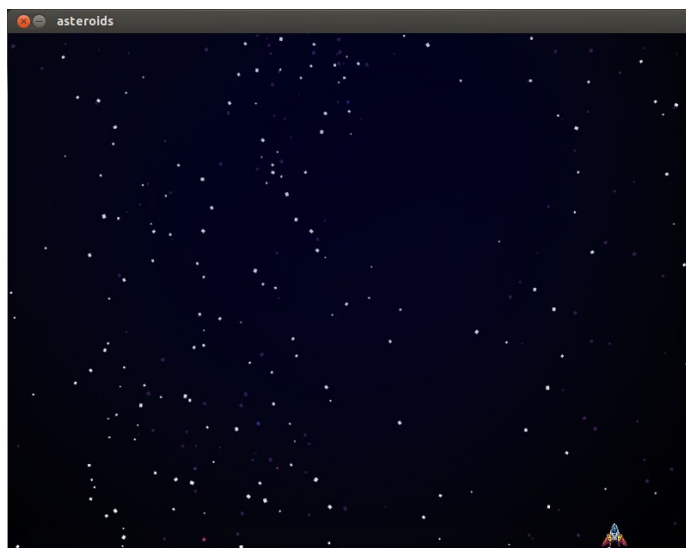


Ilustración 3: Ventana con nave desplazada

Trabajo a realizar

Sigue las instrucciones de modificación del ejercicio indicadas en cada uno de los puntos siguientes y genera un zip que contenga los archivos solicitados (scripts de python) y los recursos (imágenes, sonidos, archivos de texto, etc) necesarios para su ejecución.

1. Crea una clase ufo que maneje la imagen ufo.png y modifica el script principal para que lo dibuje en la parte superior centrada de la ventana. Guarda los scripts como taller02_01.py y ufo.py respectivamente.
2. Ahora modifica el script principal para que en lugar de interactuar con un único ufo genere usando una lista 10 ufos y los pinte en la parte superior de la pantalla. Guarda el script como taller02_02.py

Bibliografía consultada:

"Pygame documentación".

<<http://www.pygame.org/docs/index.html>>

(01 Sep 2013)

"Python documentation".

<<http://www.python.org/doc/>>

(01 Sep 2013)