
PYTHON
&
ALGORITHMIQUE
Bastien Gorissen & Thomas Stassin

Ready ?

Press start!

"Jeu" du jour

GDC Experimental Gameplay Workshop


18

Au menu...

Aujourd'hui...

On va voir une foule de concept:

- Le concept de frame
- Les événements clavier
- Les collisions
- La notion de vitesse et d'accélération appliquée au jeu vidéo.

Level 4-1

Compréhension de liste

Mais c’est quoi ce mot barbare…!?

Compréhension de liste

Les listes de compréhension sont un moyen de construire de liste à partir une d'autre liste.

Imaginons la liste suivante:

a_list = [1, 2, 5, 8, 10]

Imaginons que je ne veuille que les chiffre de cette liste compris entre 2 et 9:

[x for x in a_list if 2 <= x <= 9]

Compréhension de liste

Vous l'aurez compris, cette méthode permet de créer une liste sur base d'une condition.

Autre application:

Si j'ai une liste de ce type:

```
awesome_list = [(1, 2, 4), (5, 6), (1, 2, 3, 4)]
```

Je pourrais en extraire une liste de longueur:

```
[len(a) for a in awesom_list]
```

Compréhension de liste: Exercices

Exercice_4_1.py

```
animals = ['Giraffe', 'Dauphin', 'Ecureuil', 'Dragon',
'Zebre']
```

=> Afficher une liste des animaux dont le nom commence par C, D ou E

Compréhension de liste: Exercices

Exercice_4_2.py

Faire un classe Monster avec un attribut hp et un attribut attack, qui sont donnés comme paramètres au constructeur.

Faire une liste avec 5 monstres différents.

=> Afficher la liste des monstres avec au moins 25 hp et une attaque de 2.

=> Afficher la liste de toutes les valeurs d'attaque des monstres, triée par ordre croissant.

Level 4-2

On rajoute une couche

Game_Engine

Pour la suite du cours nous allons utiliser le package `game_engine`

Le package `game_engine` est une couche au-dessus de `cocos`.

Qu’est-ce que ça veut dire?

En clair, c’est du code qui tourne à l’aide de `cocos` et qui est là pour simplifier la vie.

Level 4-3

Premiers pas.

Ecran noir

Tout jeu à besoin de deux choses au minimun:

Jouer la fonction `init` de `game_engine` et instancier de la classe Game.

```
from game_engine import init, Game

resolution = (800,600)
init(resolution)
game = Game()
game.run()
```

Ecran noir

Normalement lors de l'exécution de ce script vous devriez obtenir un écran noir.

Petite note sur la fonction `init`: en premier argument, elle prend la résolution sous forme de *tuple*, mais en option vous pouvez passez le titre (c'est un argument optionnel). Essayez donc de remplacer l'appel de `init` par `init(resolution, "Awesome game")`

Level 4-4

Layer et Sprite

Encore une couche

Pour pouvoir ajouter des objets à notre jeu, il faut au moins une instance de *Layer*. Cette classe va servir de “récipient” à nos objets. Pour l’utiliser, nous devons créer une instance et l'ajouter au jeu (game).

```
from game_engine import init, Game, Layer

resolution = (800,600)
init(resolution)
game = Game()
layer = Layer()
game.add(layer)
game.run()
```

Première images

Enfin, on peut ajouter une instance de Sprite à notre jeu pour avoir la première image du jeu.
Un sprite a besoin au minimum du chemin de l'image (c'est à dire là où l'image se trouve) et cette fois encore il faut ajouter le Sprite, mais cette fois-ci au layer.

```
from game_engine import init, Game, Layer, Sprite
...
layer = Layer()
game.add(layer)
sprite = Sprite(r'assets/bullet.png')
layer.add(sprite)
game.run()
```

Première images

On constate que l’image apparaît à la position 0,0.
On peut évidemment choisir la position de départ d’un sprite en lui passant celle-ci en deuxième argument.

```
sprite = Sprite(r’assets/bullet.png’, (100, 200))
```

Level 4-5

Exercices :D

Constellation :

Avec ce que vous savez, faites en sorte de faire apparaître 10 sprites avec l’image “bullet.png” sur l’écran. La position de ces sprites doit être déterminée aléatoirement, mais rester dans les bornes de l’écran (donc de la résolution).

Level 4-6

On bouge!

Frame

Pour bien comprendre comment les choses s'actionne dans un jeu vidéo, il faut bien comprendre la notion de frame.

Un jeu va, comme un film, afficher plusieurs fois par seconde une version mise à jour de tous ses éléments graphiques. On appelle ceci une frame.

Entre 2 frames, le jeu va calculer toutes les mises à jour à appliquer aux différents objets. Entre autres, ça va impliquer d'appeler la fonction update de chaque instance de Sprite.

Avant Tout

Avant tout, nous allons créer une nouvelle classe.

Déclarons la classe Bullet (qui héritera de Sprite)

```
class Bullet(Sprite):
    def __init__(self, position):
        image_path = r‘assets/bullet.png’
        super().__init__(image_path)
```

A ce stade, nous avons une classe qui fera apparaître un sprite avec l’image “bullet.png”.

Vitesse :

Donnons la notion de vitesse à cette classe.

```
class Bullet(Sprite):
    def __init__(self, position, speed=(0,0)):
        image_path = r‘assets/bullet.png’
        super().__init__(image_path)
        self.speed = speed
```

Nous avons simplement ajouté un argument au constructeur, nommé `speed`. Cet argument est stocké dans un attribut qui porte le même nom.

A ce stade là, ça ne bouge toujours pas :(

E = V * T:

Revenons à la méthode `update`.

Cette méthode est jouée à chaque frame.

Elle se présente comme ceci:

```
def update(self, dt):
    pass
```

Dans le paramètre `dt` je retrouve le temps qu'il s'est produit depuis la dernière frame. Pourquoi dt ? Parce que "Delta Time".

E = V * T:

Donc si nous voulons écrire le comportement dû à la vitesse, nous avons tout ce qu'il nous faut: une vitesse (`self.speed`) et un temps (`dt`). Nous pouvons donc établir la distance parcourue par notre objet entre deux frames.

```
def update(self, dt):
    d_pos = self.speed[0] * dt, self.speed[1] * dt
    pos = self.position
    pos = pos[0] + delta_pos[0], pos[1] + delta_pos[1]
    self.position = pos
    super().update(dt)
```

Super:

Vous avez remarqué l'appel à l'ancêtre en fin de méthode:

```
super().update(dt)
```

Cela permet de garder le comportement pour la méthode update qui se trouve dans la classe `Sprite`

On tire à vue:

Maintenant, notre script principal, nous allons ajouter un Bullet au jeu:

```
from game_engine import init, Game, Layer
from custom_items import Bullet

init((800, 600))
game = Game()
layer = Layer()
game.add(layer)
bullet = Bullet((10,10),(100,100))
layer.add(bullet)
game.run()
```

LEVEL 4-7

Exercices ^^

ReBond:

Faites en sorte que le Bullet qui est généré ricoche contre les bords du jeu.

Pour ricocher, il faut inverser l’axe correspondant au bord touché.

Si vous touchez les bords de gauche ou de droite, inversez la vitesse horizontale (l’axe des X), si vous touchez les bords du haut ou du bas, faites de même pour la vitesse verticale (l’axe des Y).

LEVEL 4-8

On se rentre dedans

On tire à vue:

De la même manière que la méthode `update` est jouée à chaque frame, la méthode `on_collision` est jouée à chaque collision entre deux Sprites d'un même Layer.

Cette méthode se présente comme ceci:

```
def on_collision(self, other):
    pass
```

L'argument other contient le sprite rentrer en collision avec votre Sprite.

On tire à vue:

Donc si on voulait faire en sorte que chaque Bullet qui rentre en interaction avec un Sprite le détruise on fera comme suit.

```
def on_collision(self, other):
    other.destroy()
```

Il ne nous reste plus qu’a tester en ajoutant un deuxième Bullet à notre script et en s’arrangeant pour qu’ils se rencontrent.