Sujet 12 : Pyglet: display, sprites, events et event loop

Auteur : Térence Chevroulet
Date : 2019/05/31

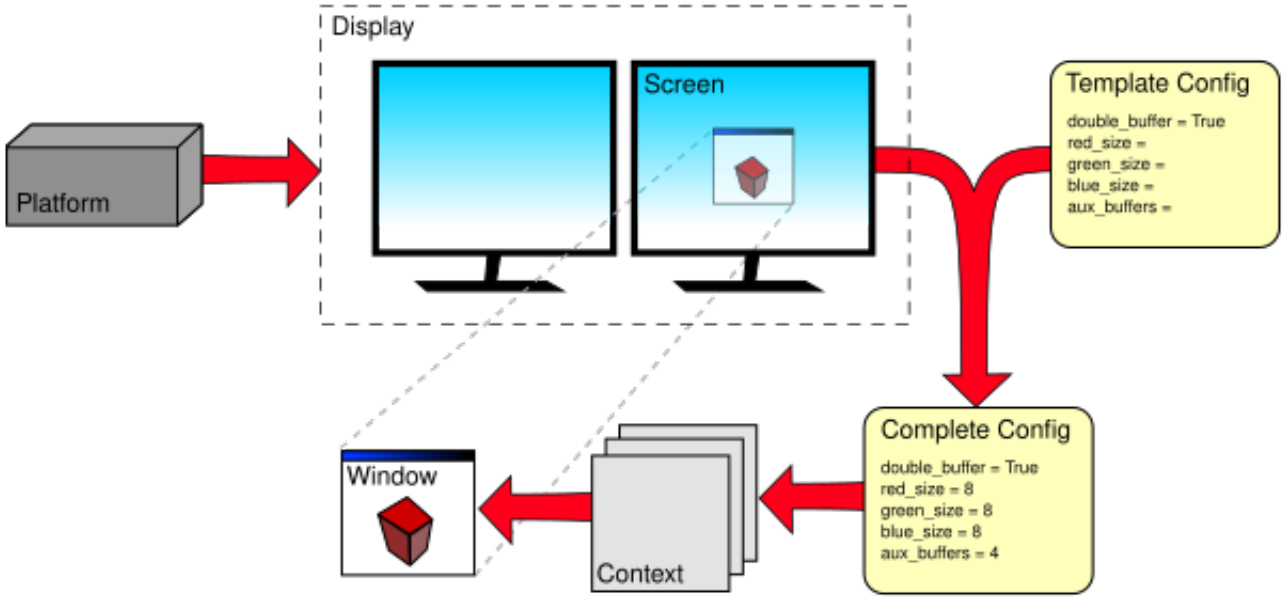
Sommaire

Introduction

En raison de limitations de *Jupyter Notebook*, il n'est malheureusement pas possible de faire entièrement fonctionner tous les exemples de *pyglet*. C'est pourquoi certaines fonctions ne sont pas exécutables, mais se retrouvent en format de code *Python* tout de même.

De plus, une dépendance est nécessaire afin de jouer des vidéos, ou de jouer des sons qui ne seraient pas supportés par *OpenAL*.

Display



Un *display* est, pour Pyglet, l'ensemble d'*écrans*, *screens* en anglais, d'un système sur lesquels il serait possible d'afficher des éléments. Un *écran* peut être local, où connecté à distance par réseau.

Il est possible d'obtenir le *display* utilisé avec `get_display()` :

In [1]:
```
import pyglet
```

In [2]:
```
display = pyglet.canvas.get_display()

print(display)
```

```
<pyglet.canvas.cocoa.CocoaDisplay object at 0x1104958d0>
```

La liste des *écrans* disponibles s'obtient avec `get_screens()`, et l'*écran* utilisé par défaut avec `get_default_screen()`. Une liste de *fenêtres*, *window* en anglais, attachées à un *display* spécifique s'obtient avec `get_windows()` :

In [3]:
```
screens = display.get_screens()
default_screen = display.get_default_screen()
windows = display.get_windows()

print(screens)
print(default_screen)
print(windows)

pyglet.app.exit()
```

```
[CocoaScreen(x=0, y=0, width=1680, height=1050)]
CocoaScreen(x=0, y=0, width=1680, height=1050)
[]
```

Window

Pyglet utilise des *fenêtres*, afin d'afficher du contenu à l'écran. Le contenu d'une fenêtre étant dessiné avec *OpenGL*, un *contexte OpenGL* doit exister, ce qui implique une *configuration* dudit *contexte*.

Heureusement, toute nouvelle *fenêtre* vient avec une configuration par défaut qui convient parfaitement à la plupart des usages. Ainsi, il n'y a pas besoin de modifier les paramètres du *contexte*. Ainsi, pour initialiser une *fenêtre*, il suffit de faire `window = pyglet.window.Window()`

La classe `Window` a une multitude de paramètres, dont les plus importants sont :

- `width = int` : sa largeur
- `height = int` : sa taille
- `resizable = bool` : si l'utilisateur peut en changer la taille
- `fullscreen = bool` : si elle est en plein écran
- `visible = bool` : si elle est visible

Certains de ces paramètres peuvent être changés après initialisation :

- `set_fullscreen(bool)`
- `set_size(width, height)`
- `set_visible(int)`

Il est possible d'obtenir la taille d'une fenêtre avec `window.get_size()` , ou de changer la taille de la fenêtre avec `window.set_size(x, y)` .

Utiliser `window.clear()` permet d'effacer le contenu d'une fenêtre, et donc de s'assurer par exemple que tous les objets ont bien été effacés de l'affichage.

Draw

Afin d'afficher des éléments graphiques *OpenGL*, il suffit d'appeler la fonction `draw()` avec un élément, qui sera alors dessiné dans la *fenêtre*.

Objects primitifs OpenGL

Il est possible d'afficher directement des objets simples avec la fonction `pyglet.graphics.draw()` :

```
pyglet.graphics.draw(2, pyglet.gl.GL_POINTS,
    ('v2i', (10, 15, 30, 35))
                    )
```

Cependant, cette fonction est très peu efficace lorsqu'il s'agit d'avoir du contenu rafraichi en haute fréquence. Il est alors mieux de conserver la référence sous forme de *vertex_list*, c'est à dire une liste de *vertices*, grosso modo des coordonnées, avec par exemple un élément entre les points (0;10) et (10;10) :

In [4]:
```
vertex_list = pyglet.graphics.vertex_list(
    2,
    ('v2f', (0, 10, 10, 10)),
    )

print(vertex_list)
print(vertex_list.vertices[:])
```

```
<pyglet.graphics.vertexdomain.VertexList object at 0x1134e5588>
[0.0, 10.0, 10.0, 10.0]
```

Il est ensuite possible de mettre la *vertex_list* à jour, plutôt que l'effacer et la redessiner systématiquement. Celle-ci peut être dessinée en la déplaçant vers un *batch* qui contiendra d'autres éléments à afficher avec la fonction `pyglet.graphics.Batch.migrate(list, mode, group, batch)` . Le `mode` fait référence à la fonction *OpenGL* utilisée pour dessiner le lien entre les deux *vertices*. Par exemple, `GL_LINES` dessinera une ligne.

In [5]:
```
batch = pyglet.graphics.Batch()

pyglet.graphics.Batch.migrate(vertex_list, vertex_list, pyglet.gl.GL_LINES, None, batch)
```

Pour mettre à jour la liste, rien de plus simple :

In [6]:
```
vertex_list.vertices[:] = [0.0, 0.0, 20.0, 20.0] # Nouvelles valeurs pour les vertices

print(vertex_list.vertices[:])
```

```
[0.0, 0.0, 20.0, 20.0]
```

Pour l'effacer :

In [7]:
```
try:
    vertex_list.delete()
except:
    pass
```

Attention, si `vertex_list` n'existe plus, il y aura une erreur!

Batch

Nous venons de mettre un élément *OpenGL* à dessiner dans un *batch*. Mais qu'est-ce qu'un *batch* ? Un batch est une collection d'éléments destinés à être affichés. Regrouper les éléments dans un seul *batch* permet de tous les dessiner d'un coups pour de bien meilleures performances, mais aussi d'aisément échanger entre différents *batchs* à dessiner si besoin est.
Comme vu plus haut, un *batch* est initialisé avec `batch = pyglet.graphics.Batch()` . Ainsi, un *batch* sera dessiné de la manière suivante : `batch.draw()` .

Sprites

Cependant, un *batch* peut contenir plus que juste de simples formes géométriques : il peut contenir des *sprites*. Un *sprite* est une image *instancée*, qui peut avoir différentes valeurs de transparence, rotation, position, échelle, etc.

Afin d'initialiser un *sprite*, il faut premièrement importer une image dans pyglet, de préférence de format .png afin d'avoir de la transparence.

In [8]:
```
image = pyglet.image.load('logo.png')

print(image)
```

```
<ImageData 210x210>
```

Puis, il faut instancer le *sprite* en transmettant l'image à utiliser. Il est possible de directement assigner un emplacement x, y et un *batch*.

In [9]:
```
x, y = 5, 5
sprite = pyglet.sprite.Sprite(image, x, y, batch = batch)

print(sprite)
print('Position :', sprite.x, sprite.y)
print('Échelle :', sprite.scale)
print(batch, sprite.batch)
print(sprite.image)
```

```
<pyglet.sprite.Sprite object at 0x11047bda0>
Position : 5 5
Échelle : 1.0
<pyglet.graphics.Batch object at 0x114b2edd8> <pyglet.graphics.Bat
ch object at 0x114b2edd8>
<TextureRegion 210x210>
```

Un sprite a de nombreuses propriétés, qui sont modifiables avec `sprite.<propriété> = <valeur>` .
Les plus pratiques sont :

- `color = (r, g, b)` : couleur du sprite
- `opacity = 0-255` : opacité - 100% opaque à 255
- `position = (x, y)` : emplacement du sprite
- `rotation = 0-359` : rotation en degrés dans la direction des aiguilles d'une montre
- `scale = int` : échelle du sprite
- `visible = bool` : détermine si le sprite est dessiné
- `subpixel = bool` : si le sprite peut ne pas être aligné avec les pixels de l'écran, ce qui risque de le rendre moins net

In [10]:
```
sprite.scale = 0.5

print('Échelle :', sprite.scale)
```

```
Échelle : 0.5
```

Un *sprite* peut être supprimé avec la fonction `delete()` :

Events

Lorsqu'un programme *pyglet* tourne, il est possible de lui faire appeler des *fonctions* grâce à des *events*. Un *event* est un évènement, comme par exemple le déplacement du curseur. Les *events* sont liés à certains *modules* de *pyglet*, dont les plus importants sont :

- `pyglet.window`
- `pyglet.app`
- `pyglet.input`

Pyglet appelle parfois des *fonctions* par défaut lors de certains *events*. Par conséquent, plutôt que de les remplacer, il est conseillé de simplement ajouter sa *fonction* à celles qui seraient appellées lorsque l'*event* arrive en utilisant un *décorateur* au dessus de sa fonction de la manière suivante : `@.event

Par exemple, cette fonction afficherait les clics de souris dans la console :

```
@window.event
def on_mouse_press(x, y, button, modifiers):
    print((x, y), button, modifiers)
```

L'un des *events* les plus importants est le `on_draw()` , qui est appelé à chaque fois que la *fenêtre* doit être rafraichie. Il permet de dessiner le nouveau contenu avec `<contenu>.draw()` , en général `batch.draw()` .

Event Loop

Ce qui appelle les *events* et maintient le programme *pyglet* actif est un *event loop*, c'est à dire que le programme s'appelle lui-même et à chaque fois appelle les sous-*events* appropriés.

L'*event loop* débute lorsqu'il est appelé avec `pyglet.app.run()` et se termine avec `pyglet.app.exit()`.

Par défaut, l'*event loop* appelle de lui-même `pyglet.app.exit()` lorsque toutes les fenêtres ont été fermées, mais il est possible d'appeler soi-même la *fonction* plus tôt pour, par exemple, quitter un jeu lorsqu'une touche spécifique est appuyée.

Multimédia

Pyglet permet, en plus des images, d'émettre du son et d'afficher des vidéos.

Son

Pyglet utilise *OpenAL* pour jouer des sons. À moins d'installer des dépendances supplémentaires, uniquement les fichiers *WAV* sont supportés. Ces fichiers doivent être non-compressés et encodés avec un PCM linéaire. Il est possible d'en faire à partir de fichiers MP3 grâce à VLC.

Il existe un problème sous *Linux* avec *OpenAL* qui empêche l'utilisation de la stéréo et joue par conséquent les sons en mono.

Son Bref

Selon le type de programme, il est parfois nécessaire de jouer un son lorsqu'un évènement se produit, sans pour autant avoir à gérer une queue. Ainsi, *pyglet* donne accès à une fonction `play()`, qui est accessible depuis n'importe quelle *source* audio chargée précédemment. Charger une source audio se fait tout simplement avec `pyglet.media.load('<source>')` :

```
In [11]: sound = pyglet.media.load('Sound.wav', streaming = False)
         sound.play()
```

```
Out[11]: <pyglet.media.player.Player at 0x1159907f0>
```

Il est recommandé d'utiliser `streaming = False` pour les fichiers lus fréquemment, car ceci permet de les garder en mémoire plutôt que de les décoder à chaque écoute, ce qui améliore nettement la performance.

Son Long

Vous avez peut-être remarqué que la *fonction* précédente nous renvoie une *instance* de `Player`. Il se trouve que *pyglet* utilise des *instances* de la *classe* `Player` pour gérer tout ce qui est sonore. Un `Player` permet d'avoir une liste de sons à jouer, de régler son volume, et nombre d'autres options.

Afin d'ajouter un son au `Player`, il faut d'abord le charger, puis instancer le `Player`, et enfin y ajouter le son chargé.

In [12]:
```
player = pyglet.media.Player()
sound2 = pyglet.media.load('Sound.wav')

player.queue(sound2)
```

Il est ensuite possible de jouer ce son avec `player.play()`, ou alors de modifier certaines variables du `Player` :

- `volume = int` : détermine le volume auquel les sons sont joués
- `pitch = int` : permet de changer la hauteur du son.

Lorsqu'un son est assez long pour que cela soit utile, il est possible de le mettre en pause avec `player.pause()`.

In [13]:
```
player.pitch = 0.2
player.play()
```

L'utilisation d'un `Player` permet également, par exemple, de sauter le son en train d'être ouer pour passer à la source suivante avec `player.next_source()`.

Vidéo

Les vidéos fonctionnent tout comme les sons, c'est à dire avec le `Player` elles aussi. Cependant, celles-ci pouvant être volumineuses, il faut être particulièrement attitif aux questions d'optimisation et de performance!

Toutefois, la présence d'une dépendance supplémentaire est nécessaire pour pouvoir décoder les vidéos : *AVbin*.