



PROJET INDIVIDUEL

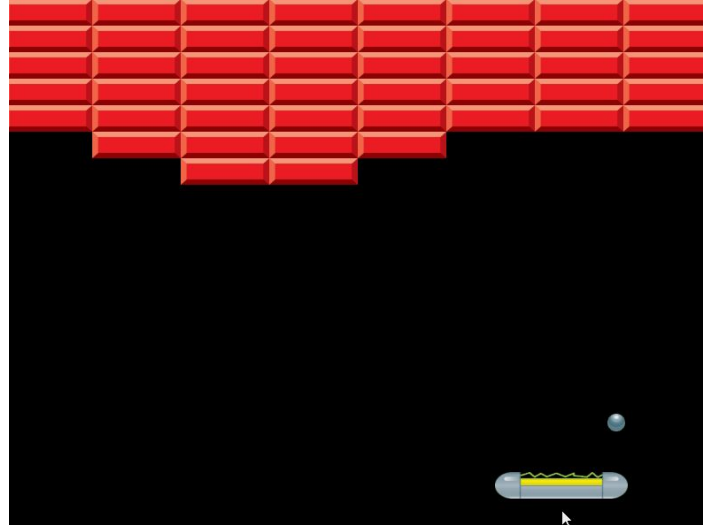
Bastien Gorissen & Thomas Stassin

BREAKOUT

"Casse-briques"

BUT DU JEU

Pour ce projet individuel, vous allez élaborer un petit jeu d'arcade, dans un genre nommé "casse-brique". L'objectif est de faire rebondir une balle sur une "raquette" en bas de l'écran pour l'envoyer détruire des briques visibles sur la partie supérieure de l'aire de jeu.



OUTILS

Nous allons avoir besoin d'un module Python qui n'est pas inclus dans la "bibliothèque standard". Contrairement à **random** que nous pouvons invoquer à l'aide de **import**, cette fois nous allons devoir installer Pygame Zero via un outil nommé pip.

Pour ce faire, ouvrez le terminal dans Visual Studio Code, et entrez la commande suivante, avant de valider par la touche Enter:

```
pip install pgzero
```

Après quelques instants, pip aura fini son travail et tout sera prêt !

PYGAME ZERO

Pygame Zero est une librairie destinée à la réalisation de jeux vidéos simples.

A l'aide de quelques lignes de codes et de quelques fonctions, vous pourrez afficher des sprites (images), détecter des collision, et répondre aux entrées clavier ou souris du joueur.

La documentation de la version que nous avons installée est ici:

<https://pygame-zero.readthedocs.io/en/stable/>

Mais nous allons voir les fondamentaux ensemble !

PREMIER "JEU"

"Un écran presque noir"

NOTRE PREMIER JEU: UN FICHIER VIDE

Pour fonctionner, Pygame Zero va vous demander de lancer vos programmes de façon un peu particulière.

En même temps que le code de la librairie, vous avez aussi installé un petit programme du nom de "**pgzrun**".

- Créez un script python vide (ex : **first_game.py**)
- Ouvrez un terminal, et écrivez **pgzrun first_game.py**
- Validez avec Enter et admirez une fenêtre noire !

Attention: utilisez "Open folder" dans VSCode pour ouvrir le dossier de votre jeu.

NOTRE PREMIER JEU, V2.0 !

Un écran noir, ce n'est pas encore très fun.

Pour contrôler votre jeu, vous allez devoir définir certaines variables et fonctions que Pygame Zero va reconnaître et prendre en compte.

Par exemple, vous pouvez déterminer la taille de la fenêtre du jeu (en pixels) en déclarant une variable **WIDTH** (largeur) et/ou **HEIGHT** (hauteur), en majuscules :

```
WIDTH = 800
```

```
HEIGHT = 600
```

Essayez aussi de changer les nombres pour voir ce qu'il se passe !

AFFICHER QUELQUE CHOSE

Essayons maintenant d'afficher quelque chose à l'écran ! Comme pour un dessin animé, l'illusion du mouvement dans un jeu vidéo vient du fait que l'ordinateur va afficher des images légèrement différentes très rapidement (ici, 60 fois par seconde).

Pygame Zero va donc nous fournir un moyen de dire à Python ce qu'il doit afficher dans notre fenêtre à chaque rafraîchissement de l'écran (vous entendrez aussi souvent le nom de "frame" pour ces images successives).

Voyons comment utiliser ce principe pour ajouter afficher un cercle vert !

LE CERCLE

Pour dire à Python ce qu'il doit afficher, Pygame Zero va vous demander de définir une fonction `draw()`, sans arguments, et en minuscules. Cette fonction sera appelée à chaque frame.

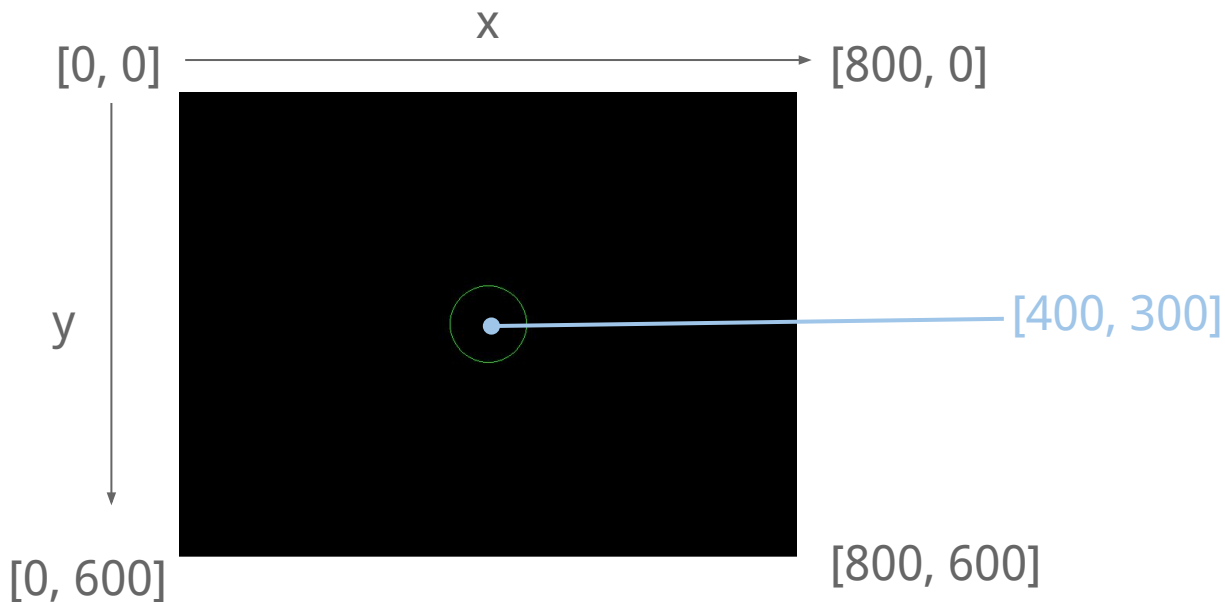
A l'intérieur, vous pourrez utiliser une série de fonctions, par exemple celle qui vous permet d'afficher un cercle :

```
def draw():  
    screen.draw.circle([400, 300], 50, "green")
```

Vous spécifiez donc le centre du cercle, son rayon, et sa couleur.

POUR SE COORDONNER

En Pygame Zero, l'unité de référence, c'est le "pixel". Chaque pixel de l'image correspond à une coordonnée en x et en y, en commençant par le coin supérieur gauche :



CASSE-BRIQUES

"Silence... Action !"

MISE EN MOUVEMENT

Notre jeu reste pour le moment très statique !

Pour introduire du mouvement, nous allons faire bouger le cercle. Pratiquement, ça revient à changer au cours du temps la position du centre du cercle. Si nous le faisons bouger un petit peu à chaque frame, l'impression sera que le cercle se déplace.

- Il faut donc stocker la position actuelle du centre du cercle dans une variable pour pouvoir s'y référer
- Il faut savoir où et quand mettre à jour cette position

MISE À JOUR

Nous pourrions modifier la position du cercle directement dans la fonction `draw()`, mais en général, un jeu va plutôt séparer la partie affichage de la partie mise à jour des différents objets, et c'est également vrai pour Pygame Zero.

De façon schématique, un jeu est une boucle infinie du type :

Tant qu'on ne quitte pas le jeu:

Gérer les entrées (clavier/souris/...)

Mettre à jour les données

Dessiner le résultat à l'écran

MISE EN PRATIQUE

Dans notre cas, Pygame Zero va proposer une fonction `update()` qui sera exécutée avant chaque nouvel affichage (nouveau code en **jaune**):

```
WIDTH = 800
```

```
HEIGHT = 600
```

```
center = [400, 300]
```

```
def draw():
```

```
    screen.draw.circle(center, 50, "green")
```

```
def update():
```

```
    center[0] = center[0] + 1
```

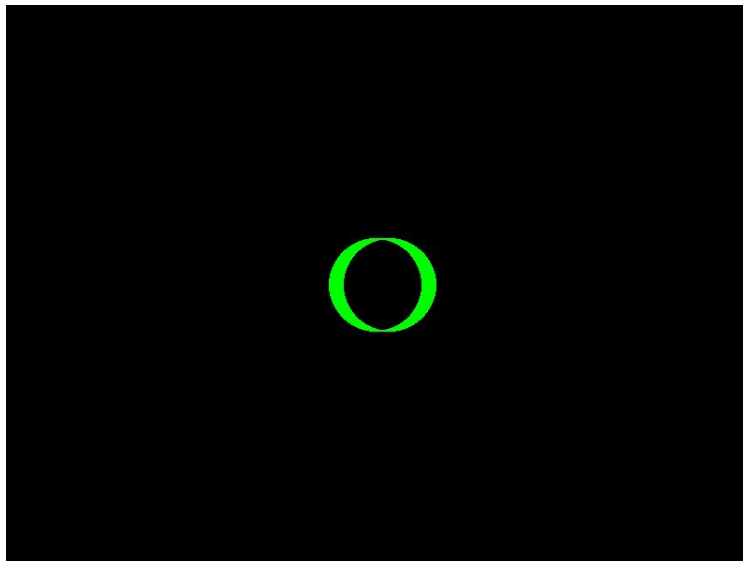
La coordonnée x du centre du cercle augmente de 1 pixel à chaque frame.

MISE AU VERT

Oups...

Chaque appel de **draw()** va simplement dessiner par dessus l'image précédente, ce qui donne cette effet de traînée verte...

Pour résoudre ceci, il suffit de rajouter une ligne au début de notre fonction **draw()** qui efface l'écran !

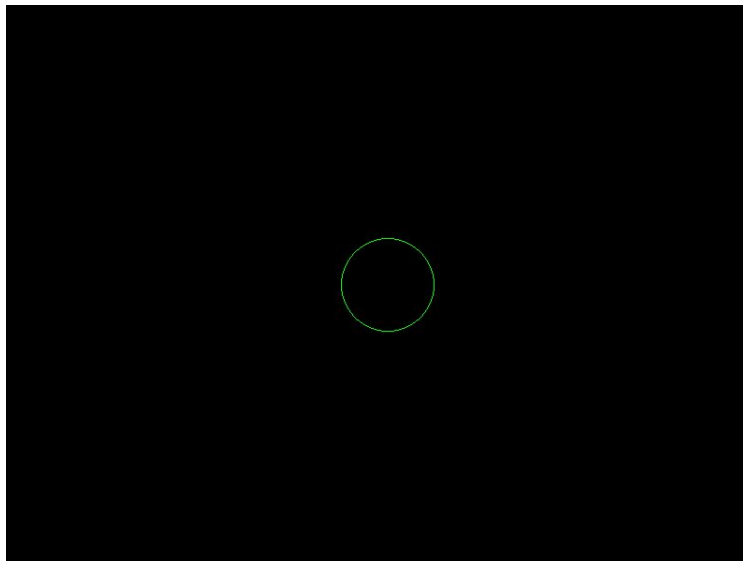


MISE AU PROPRE

Modifiez votre fonction `draw()` pour qu'elle commence par vider entièrement l'écran avant de dessiner le cercle à sa nouvelle position:

```
def draw():  
    screen.clear()  
    screen.draw.circle(center, 50, "green")
```

...much better!



CASSE-BRIQUES

"Soif de Sprite"

OBJETS INTERACTIFS

Au lieu d'un cercle, voyons maintenant comment ajouter au jeu des objets qui vont pouvoir interagir entre eux.

En Pygame Zero, le terme utilisé est "**Actor**", et peut représenter une plate-forme, un personnage, une brique de notre casse-brique, la balle, etc...

Chaque **Actor** va être créé à partir d'une image qui le représente, et va vous offrir plusieurs fonctions et variables que vous allez pouvoir modifier pour changer le comportement de l'élément de jeu.

UNE PREMIÈRE BRIQUE

Avant de pouvoir afficher une brique, nous devons placer un fichier image (ici, un fichier .png) dans un sous-dossier "images" dans notre projet.

Le nom du dossier doit être "images", en minuscules, sinon Pygame Zero ne va pas pouvoir retrouver les images :)

Une fois le fichier ajouté au projet (plus tard nous mettrons d'autres fichiers dans le sous-dossier images), nous allons pouvoir écrire le code qui l'affichera à l'écran !




brick.png

AFFICHAGE !

Pour afficher notre brique, il faut donc créer un Actor, et l'afficher dans la fonction `draw()`.

```
brick = Actor("brick")  
brick.pos = [0, 0]
```

Position sur l'écran où
l'Actor va être placé.



```
def draw():  
    screen.clear()  
    brick.draw()
```

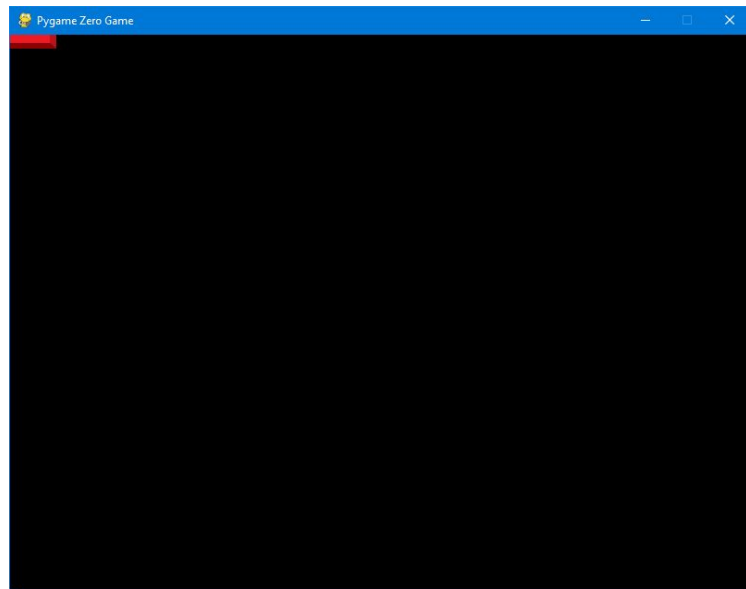
Chaque Actor a besoin du nom de l'image (sans extension) et doit être dessiné sur l'écran avec sa propre méthode `draw()` (attention à ne pas confondre :D).

HORS-JEU

Pour l'instant, notre brique dépasse de l'écran!

C'est parce que par défaut, Pygame Zero positionne le centre de l'Actor (et donc de l'image) là où vous le demandez.

Pour nous faciliter les choses, nous allons changer ce point de référence pour utiliser le coin supérieur gauche de l'image.

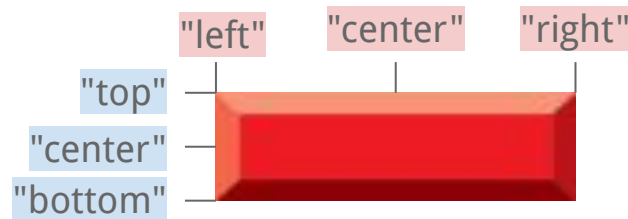


LEVER L'ANCRE

Le point de référence que Pygame Zero utilise est appelé "ancre". Vous pouvez, au moment de la création d'un Actor, changer sa position en spécifiant quelle partie de l'image utiliser comme référence (horizontalement, puis verticalement).

Exemple:

```
brick = Actor("brick", anchor=["left", "top"])
```



EXEMPLES D'ANCRES:

Prenons un écran de 300 x 100 avec notre image de brique (100 x 30)

Si je place l'Actor en 0, 0 avec une ancre de ["left", "top"]:



Si je place l'Actor en 20, 20 avec une ancre de ["left", "top"]:



Vous voyez que la brique se décale du coin de la fenêtre par rapport à son coin supérieur gauche.

EXEMPLES D'ANCRES:

Mais nous pouvons aussi changer l'ancre utilisé pour positionner l'Actor !

Si je place l'Actor en **150, 50** avec une ancre de `["right", "bottom"]`:



Si je place l'Actor en **150, 50** avec une ancre de `["center", "top"]`:



Le point jaune (l'ancre) est à la même position sur les deux images, mais Pygame Zero utilise un point de référence différent pour positionner l'Actor.

ANOTHER BRICK IN THE WALL

Pour que notre jeu soit intéressant, il nous faut plus d'une brique !

A la place d'en créer une seule, vous pouvez utiliser des boucles pour créer toute une liste de briques.

Si notre fenêtre fait 800 x 600, et que chaque brique fait 100 x 30, nous pouvons mettre 8 briques sur une ligne. Si l'ancre de nos briques est au coin supérieur gauche, et que nous mettons les briques en haut de l'écran, leurs position en y sera toujours 0 (le dessus de l'écran), et pour les coordonnées en x, elles augmenteront pour chaque brique :

0, 100, 200, 300, 400, 500, 600, 700

ANOTHER BRICK IN THE WALL

0, 100, 200, 300, 400, 500, 600, 700

Voilà qui fait penser à un `range()` !

En combinant `range()` et boucle `for`, vous pouvez mettre plusieurs briques dans une liste, et ensuite parcourir cette liste pour afficher chaque brique !

En guise d'exercice, nous allons vous fournir le code qui va créer la liste de briques, mais à vous de compléter les paramètres du `range` pour avoir une ligne de briques qui remplit correctement l'écran !

ANOTHER BRICK IN THE WALL

```
WIDTH = 800  
HEIGHT = 600
```

```
all_bricks = []  
for x in range(???, ???, ???):  
    brick = Actor("brick", anchor=["left", "top"])  
    brick.pos = [x, 0]  
    all_bricks.append(brick)
```

```
def draw():  
    screen.clear()  
    for brick in all_bricks:  
        brick.draw()
```

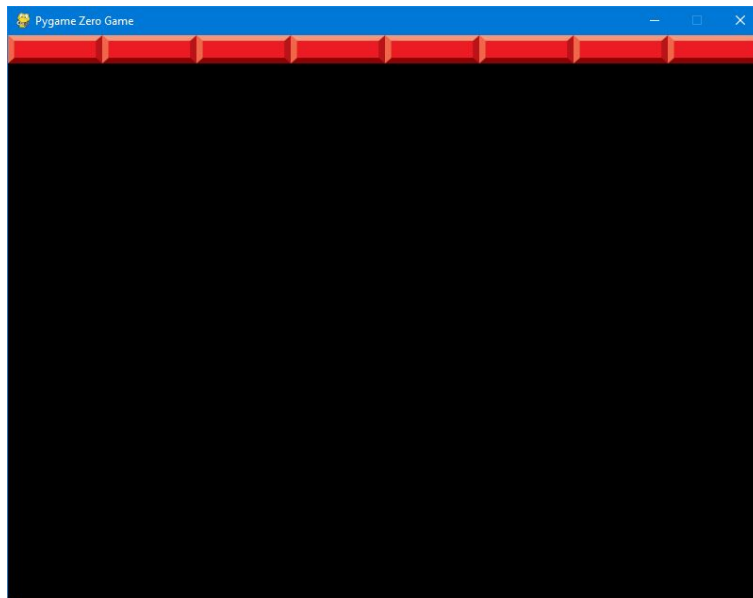
Ne pas oublier de bien positionner l'ancre des images !

La coordonnée x change, mais toutes les briques sont placées en y = 0.

On boucle sur toutes les briques pour les afficher.

PREMIÈRE LIGNE !

Une seule ligne, c'est bien, mais un peu pauvre ! Nous allons maintenant essayer d'en rajouter plusieurs !



PLUS DE LIGNES !

Pour rajouter des lignes, nous allons également utiliser une boucle. Mais cette fois, au lieu de faire varier la coordonnée en x, nous allons faire varier celle en y !

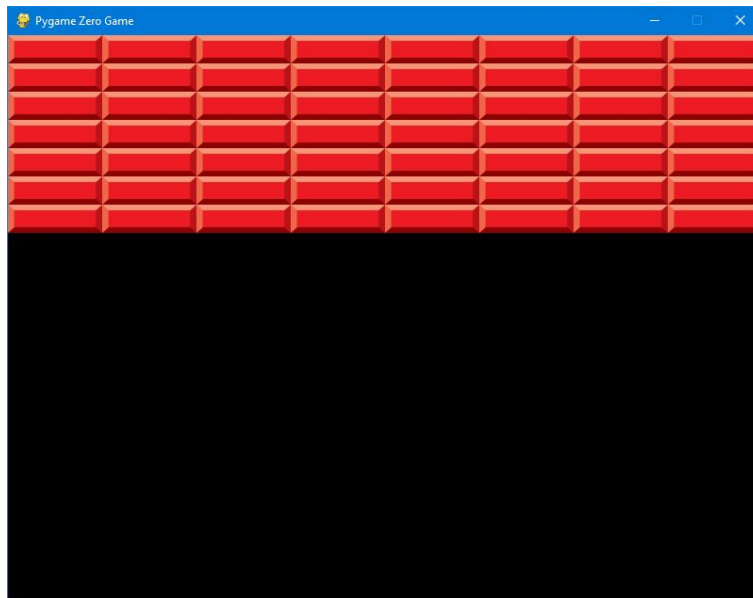
Exercice : Ajoutez une boucle autour de la boucle qui construit la première ligne pour obtenir au total 7 lignes.

Notes :

- Vous pouvez continuer à utiliser une simple liste, pas besoin de faire une liste de listes.
- En plus de votre nouvelle boucle, la seule chose qui devrait changer dans votre code est la coordonnée en y des briques.

DONE !

Voilà nos briques placées ! Nous n'avons pas encore tout à fait un jeu digne de ce nom, mais c'est un bon premier pas !



CASSE-BRIQUES

"Kid Paddle"

"PERSONNAGE" JOUEUR

Pour ajouter de l'interaction à notre jeu, nous devons mettre en place l'Actor qui va représenter le joueur. Dans le cas de notre casse-briques, c'est la "raquette" que le joueur va contrôler à l'aide de la souris.

Pour créer l'Actor, rien que nous n'ayons déjà vu :

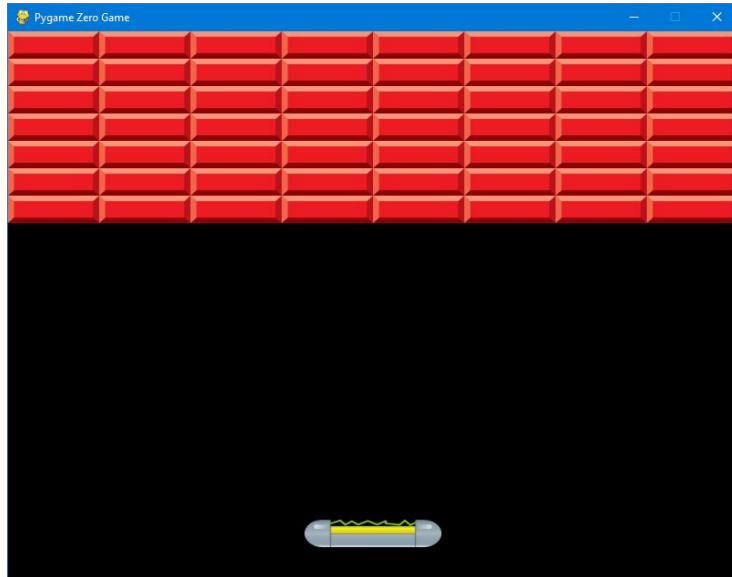
```
player = Actor("player")  
player.pos = [400, 550]
```

Et bien entendu, n'oubliez pas de l'afficher dans la fonction **draw()** :

```
player.draw()
```

"PERSONNAGE" JOUEUR

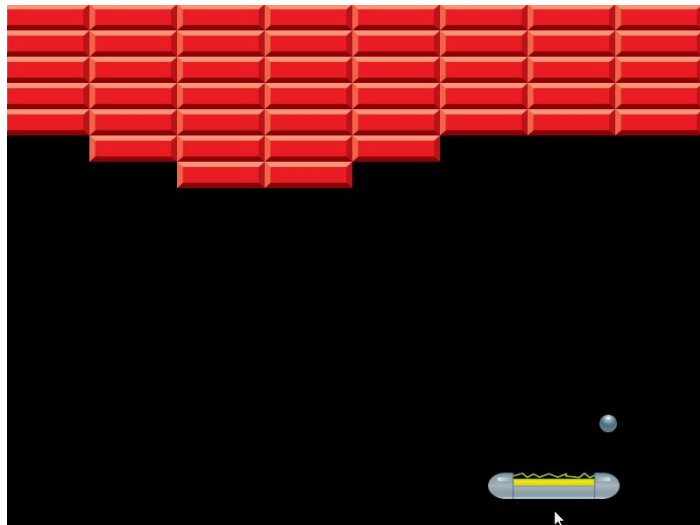
Pas encore très mobile, mais nous y venons !



COLLER AU POINTEUR

Le mouvement de notre joueur va être assez simple : quand il bouge la souris, nous allons simplement placer la raquette là où le pointeur se trouve, horizontalement.

Nous allons commencer par une version simple, où nous allons suivre le pointeur de la souris partout, avant de restreindre le mouvement de la raquette pour que le joueur ne puisse la bouger que de gauche à droite.



DÉTECTER LE MOUVEMENT DE LA SOURIS

Heureusement pour nous, Pygame Zero propose une série de fonctions pour détecter les mouvement de la souris, les touches du clavier, les clics, etc...

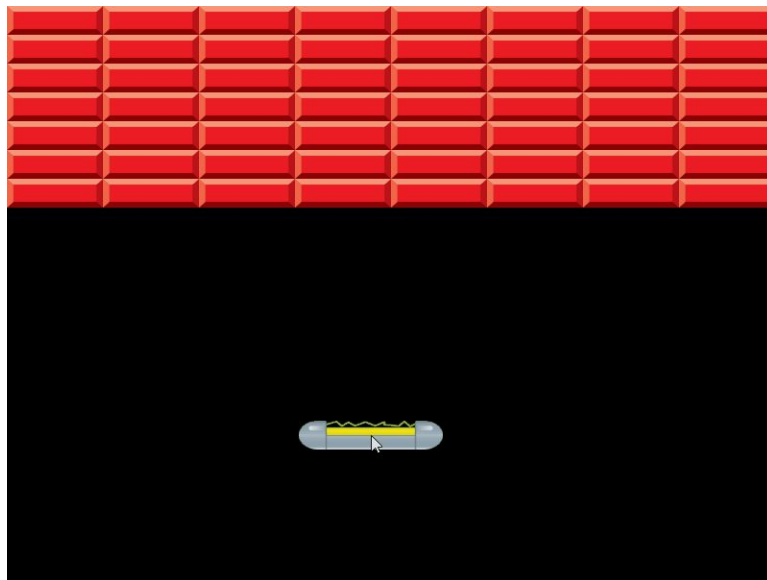
Celle que nous allons utiliser est appelée à chaque frame lorsqu'un mouvement de la souris est détecté par Pygame Zero. Elle reçoit comme argument la nouvelle position du pointeur. Ajoutons le code qui va bouger la position de notre **player** à la position du pointeur chaque fois qu'il bouge :

```
def on_mouse_move(pos):  
    player.pos = pos
```

Attention à la différence entre **pos** et **player.pos** !

MOUVEMENT LIBRE !

Pour l'instant, vous pouvez bouger la raquette comme vous voulez, partout sur l'écran. A nous de limiter un peu tout ça !



LIMITER LE MOUVEMENT

Pour limiter le mouvement, nous allons simplement éviter de modifier la coordonnée en y de la raquette.

Nous allons donc donner comme nouvelle position au joueur :

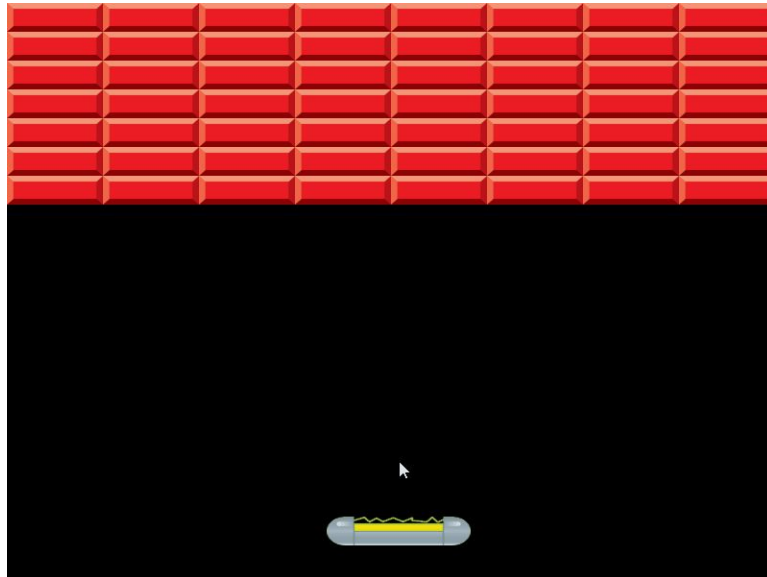
[position horizontale du curseur de la souris, position verticale du joueur]

Ce qui va donc changer sa position en x, mais conserver celle en y !

```
def on_mouse_move(pos):  
    player.pos = [pos[0], player.pos[1]]
```

SUCCESS !!!

Notre raquette reste bien dans le bas de l'écran, et vous pouvez la diriger horizontalement avec la souris.



CASSE-BRIQUES

"Collision Course"

PIÈCE DE RÉSISTANCE

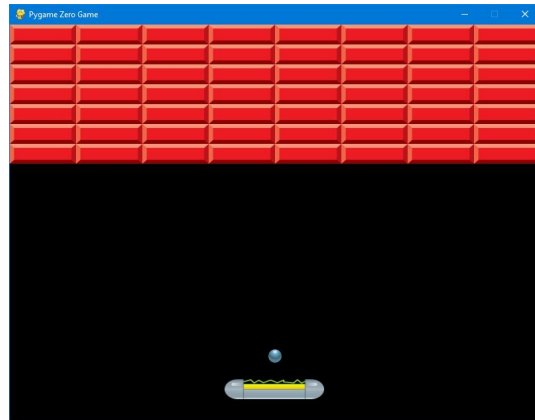
Le dernier élément manquant à notre casse-briques est ce qui va, justement, casser lesdites briques : une balle qui rebondit partout !

Pour créer la balle, rien de nouveau, il faut créer un Actor:

```
ball = Actor("ball")  
ball.pos = [400, 500]
```

et le dessiner à l'écran dans la fonction **draw()** :

```
ball.draw()
```



MOUVEMENT

Nous allons maintenant faire en sorte que la balle se déplace.

Tout comme pour le cercle vert au tout début de notre découverte de Pygame Zero, le "truc" va être de modifier la position de la balle au cours du temps via la fonction **update()** pour donner l'impression qu'elle se déplace.

Mais nous savons déjà que, selon les circonstances, la balle va devoir "rebondir", et changer de direction. Pour pouvoir gérer ceci plus tard, nous allons sauvegarder la "vitesse" (le nombre de pixels que la balle va parcourir horizontalement et verticalement chaque frame) dans une variable.

A VOS MARQUES...

Modifiez votre code pour rajouter la vitesse :

```
ball = Actor("ball")  
ball.pos = [400, 500]  
ball_speed = [3, -3]
```

La valeur signifie qu'à chaque update, la balle va avancer de 3 pixels vers la droite, et 3 vers le haut (n'oubliez pas, pour Pygame Zero l'origine est en haut à gauche de la fenêtre !)


Il nous reste à appliquer cette vitesse en utilisant la fonction **update()**.

MOUVEMENT

Dans la fonction `update()`, nous allons faire en sorte que la nouvelle position de la balle soit sa position actuelle + la vitesse, ce qui va avoir pour effet de la déplacer légèrement (dans notre cas, de 3 pixels vers la droite, et 3 vers le haut).

Pour simplifier l'écriture, nous pouvons calculer la nouvelle position via des variables intermédiaires :

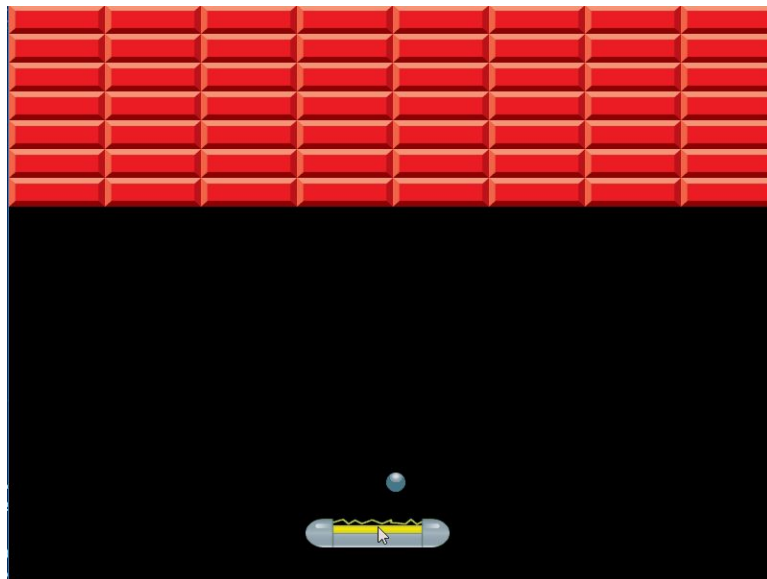
```
def update():  
    new_x = ball.pos[0] + ball_speed[0]  
    new_y = ball.pos[1] + ball_speed[1]  
    ball.pos = [new_x, new_y]
```



Notez la correspondance
entre les index !

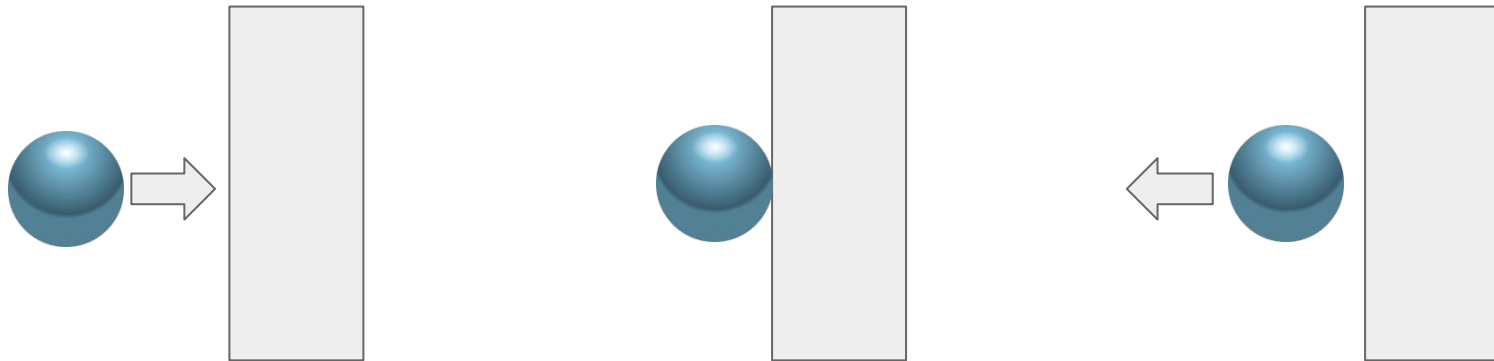
VERS L'INFINI ET AU-DELÀ !

Visiblement, notre balle n'est pas très intéressée par rester avec nous...



BOUNCE !

Pour simuler l'effet d'un mur, nous allons devoir détecter quand notre balle est sur le point de quitter l'aire de jeu, et modifier sa vitesse. Par exemple, si la balle va vers la droite, et touche le bord droit de l'écran, si elle "rebondit", elle va ensuite continuer à bouger vers la gauche.



CHANGEMENT DE VITESSE

En terme de nombres, si la vitesse horizontale était **3** (déplacement vers la droite), après l'impact, elle va devenir **-3**. Et inversement, si la vitesse était de **-3** (déplacement vers la gauche), elle deviendra **3** si la balle heurte le mur de gauche.

La même logique peut être appliquée pour la vitesse verticale.

Pour passer de 3 à -3 ou inversement, il suffit de multiplier la valeur actuelle de la vitesse, horizontale ou verticale, par -1 !

Mettons tout ça dans deux petites fonctions pour nous simplifier la vie.

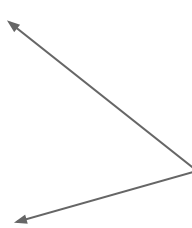
CHANGEMENT DE VITESSE

Cette fois-ci, nous allons écrire des fonctions qui ne sont pas spécifiques à Pygame Zero, mais qui sont simplement là pour nous simplifier la vie !

Mettez-les avant `update()` (nous aurons besoin de les utiliser à l'intérieur d'`update()` justement...) :

```
def invert_horizontal_speed():  
    ball_speed[0] = ball_speed[0] * -1
```

```
def invert_vertical_speed():  
    ball_speed[1] = ball_speed[1] * -1
```



Multiplier la vitesse par
-1 pour inverser la
direction !

RESTER SUR L'ÉCRAN

Maintenant que nous avons nos fonctions, nous allons pouvoir les utiliser !

En plus de la position de chaque Actor, Pygame Zero peut vous donner la position du bord gauche (left), droit (right), supérieur (top) ou inférieur (bottom) de votre Actor.

Dans update, ajoutons une condition pour faire rebondir la balle quand elle dépasse de l'écran à droite :

```
def update():
```

```
    ...
```

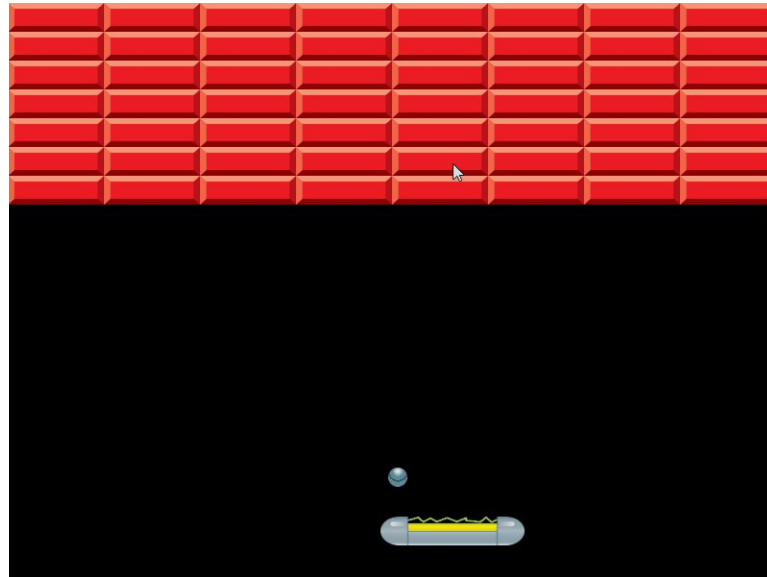
```
    if ball.right >= WIDTH:
```

```
        invert_horizontal_speed()
```

Code existant de la fonction update.

PREMIER REBOND

Pas mal, pas mal ! Mais la balle continue à sortir de l'écran !



A VOUS DE JOUER !

En vous basant sur ce que nous avons fait aux étapes précédentes, faites en sorte que la balle rebondisse sur au moins 3 des 4 bords de l'écran (vous pouvez la laisser filer par le bas, ce qui pourrait correspondre à un "Game Over" pour le joueur).

Faites attention au bord de la balle que vous considérer, ainsi que la coordonnée du bord de l'écran.

Pour vous simplifier la tâche, commencez par régler le problème du haut de l'écran, avant de régler celui de la gauche, et n'hésitez pas à changer la vitesse de la balle (pour voir directement les effets de vos changement !).

CASSE-BRIQUES

"Casser les briques..."

COLLISIONS ENTRE ACTORS

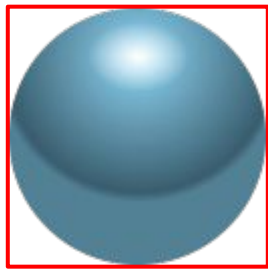
Le dernier élément qui nous manque pour terminer notre casse-briques est la collision entre Actors.

Entre la raquette et la balle, et entre les briques et la balle.

Pour détecter les collisions, Pygame Zero vous permet de tester si le rectangle d'un Actor est en train d'intersecter celui d'un autre Actor.

COLLIDERS

Pour simplifier les calculs, Pygame Zero considère chaque Actor comme un rectangle aux dimensions de l'image qui le représente.



C'est un peu moins précis que de vérifier chaque pixel de chaque image, mais ça permet au jeu de continuer à être performant !

COLLIDERS

Pour gérer la collision entre la raquette et la balle, il faut demander à Pygame Zero si le rectangle de la balle intersection celui de la raquette dans `update()`, et réagir de façon adéquate :

```
def update():
```

```
    ...
```

```
    if ball.colliderect(player):
```

```
        invert_vertical_speed()
```

Si la balle touche la raquette, la balle inverse sa vitesse verticale !

COLLISION AVEC LES BRIQUES

Votre dernier défi consiste à détruire les briques quand la balle les touche !

A la suite du code existant de la fonction **update()**, essayez de traduire les étapes suivantes en code :

- Pour chaque brique dans la liste de toutes les briques :
 - Si la balle est en collision avec la brique:
 - Retirer la brique de la liste des briques
 - Inverser la vitesse verticale de la balle

Après ça, vous pourriez améliorer plein d'aspects de votre jeu, mais la base est en place !