TP : un jeu avec Pygame

□ Objectifs

- Découvrir le module Pygame et utiliser ses fonctionnalités pour créer un mini-jeu
- Développer un projet en commun

□ Pré-requis

- Bases en Python
- Bases de Programmation Orientée Objet

Un jeu que nous ne programmerons pas tout de suite...

1 But du jeu

Le but de ce TP est de développer un jeu vidéo simple à partir de vos connaissances en Python et du module **Pygame**.

Peu original, le choix s'est porté sur **Pong**, célèbre jeu d'arcade édité par Atari en 1972. Malgré son apparente facilité, le développement d'un tel jeu est loin d'être évident, et le *gameplay* original peut être enrichi de nombreux ajouts, ce qui en fait un excellent choix d'expérimentation.

Inspiré du tennis de table en « vue de dessus », Pong se joue à 1 ou 2 joueurs, chacun déplaçant une raquette de haut en bas, dans le but de faire parvenir la balle de l'autre côté de l'écran, derrière la raquette adverse.
Si la balle touche les bords supérieur et inférieur de l'écran, elle rebondit, tandis que si elle touche les bords gauche et droite, un des joueurs marque un point et la partie recommence.

Prêt à se lancer ?

2 Par où commencer ?

Bien entendu, nous n'allons pas coder toute l'interface graphique à partir de zéro. Python étant un langage haut niveau, nous allons utiliser une librairie, *Pygame*, afin de créer la fenêtre de jeu et de représenter les éléments de base que sont la balle et les raquettes.

2.1 Installation

***Pygame* étant fourni de base avec EduPython, il n'est pas nécessaire de l'installer sur cet IDE.**
Pour les autres : `https://www.pygame.org/download.shtml`

Pour utiliser le module *Pygame*, il faut l'importer classiquement de la manière suivante :

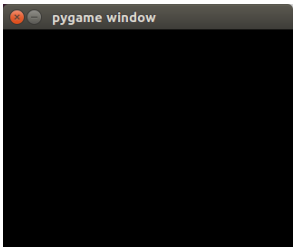
```
1 import pygame
```

De cette manière, chaque appel de fonction du module est de la forme `pygame.fonction(parametres)` : cela permet de bien séparer ce qui dépend du module *Pygame* du reste. Autre particularité, *Pygame* possède de nombreux **sous-modules**, dédiés à des actions différentes. Par exemple, le sous-module `display` gère l'affichage. On utilisera les fonctions de ce module en écrivant `pygame.display.fonction(parametres)`.

2.2 Création de l'écran de jeu

Pour créer la fenêtre de jeu, on peut utiliser le code suivant, commenté et détaillé page suivante :

```
1  import pygame
2
3  pygame.init() # Initialisation de Pygame et de ses sous-modules
4
5  # Définition des constantes
6  LARGEUR_ECRAN, HAUTEUR_ECRAN = 320, 240
7  TAILLE_ECRAN = (LARGEUR_ECRAN, HAUTEUR_ECRAN)
8  NOIR = (0,0,0)
9
10 # Création de l'écran de jeu
11 ecran = pygame.display.set_mode(TAILLE_ECRAN)
12
13 # Boucle d'affichage
14 continuer = True
15
16 while continuer:
17
18     # Événement de type "Terminer" : arrêt de la boucle
19     for event in pygame.event.get():
20         if event.type == pygame.QUIT:
21             continuer = False
22
23     ecran.fill(NOIR)     # On remplit l'écran en noir
24     pygame.display.flip() # On met à jour l'affichage
```



Quelques remarques

- Une **constante** est une variable dont la valeur ne varie pas au cours du programme. Bien sûr, c'est une règle que l'on se fixe (on ne modifiera pas la taille de l'écran au cours du programme) et la convention veut que l'on nomme ces constantes avec des **majuscules** uniquement.

- `NOIR` désigne une couleur, classiquement représentée par un triplet de valeurs `(R,V,B)` (pour Rouge, Vert et Bleu), chaque valeur étant comprise entre 0 et 255 (0 : niveau minimal, 255 : niveau maximal).

```
NOIR = (0,0,0)
```

- La fonction `pygame.display.set_mode` attend un **couple de valeurs** en paramètres.

```
ecran = pygame.display.set_mode(TAILLE_ECRAN)
```

  Elle retourne un **objet** que l'on récupère dans la variable `ecran`.

- Notre écran n'est pas figé mais s'actualise constamment dans la **boucle d'affichage**, qui est active tant que la variable `continuer` est égale à `True`.

```
while continuer:
```

- *Pygame* gère les **événements**. Un événement est une action enregistrée par le programme comme l'appui sur une touche du clavier, un clic sur une partie de l'écran, le déplacement de la souris... La fonction qui permet de récupérer la liste des événements enregistrés est `pygame.event.get()`. À chaque tour dans la boucle d'affichage, on **parcourt** cette **liste d'événements** pour voir si l'utilisateur a provoqué un événement :

```
for event in pygame.event.get():
```

  Si un événement de type « Terminer » survient, on arrête la boucle d'affichage en passant `continuer` à `False` :

```
for event in pygame.event.get():
    if event.type == pygame.QUIT:
        continuer = False
```

  `event.type` peut prendre plusieurs valeurs... Celle qui nous intéresse ici est `pygame.QUIT`, qui comme son nom l'indique, est une constante propre à *Pygame* désignant le type d'événement « Terminer ».

  Que vaut cette constante ? 0 ? 1234 ? Peu importe : elle signale qu'un événement de type « Terminer » est survenu, comme la demande de fermeture de la fenêtre.

- L'écran est colorié en noir avec la méthode `fill` (*fill* = *remplir*)

- La fenêtre est mise à jour avec l'appel de :

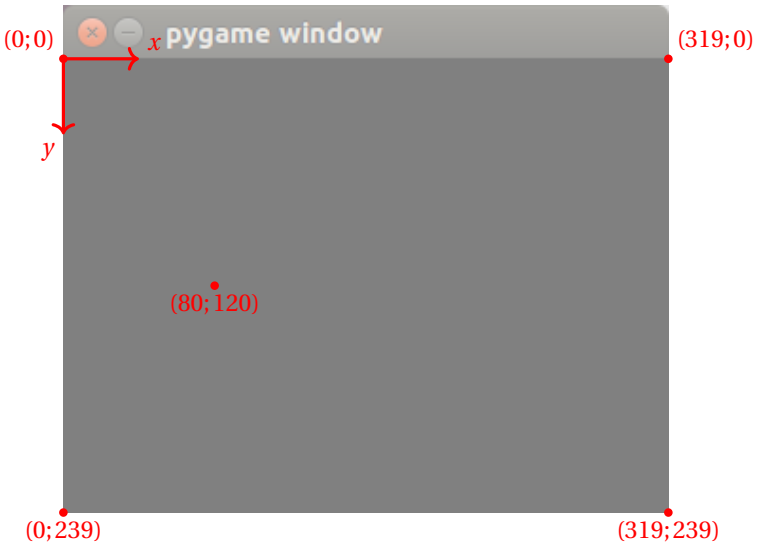
```
pygame.display.flip()
```

  Cette fonction est à appeler en dernier, après avoir colorié l'écran et affiché quoi que ce soit dessus (comme notre balle ou nos raquettes).

2.3 Fenêtre et coordonnées

Avant d'afficher des objets sur notre écran, il est nécessaire de comprendre le système de coordonnées utilisé par *Pygame* (et par la majorité des modules graphiques, tous langages confondus).

Chaque pixel de notre fenêtre est repéré par ses **coordonnées cartésiennes**, c'est à dire une abscisse x et une ordonnée y.
Le repère utilisé est **orthonormé**, mais l'axe des ordonnées est dirigé vers le bas, et l'origine du repère se situe en haut à gauche de l'écran.

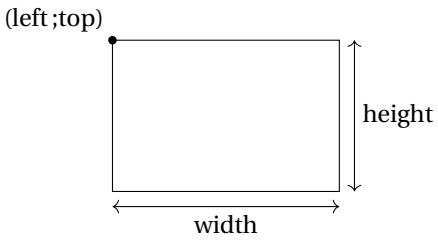


2.4 Création de la balle

Pygame permet de créer toutes sortes de formes géométriques, comme un rectangle, un cercle, un polygone... Pour simplifier la programmation du jeu, nous allons considérer un balle de forme **carrée**.
Pour créer un rectangle avec *Pygame* (un carré est un rectangle), on utilise `pygame.Rect()` qui permet de créer un objet de type `Rect` (remarquez la majuscule, signe d'un nom de classe). Le constructeur de `Rect` attend 4 paramètres :

```
Rect(left, top, width, height)
```

où `left` et `top` sont les coordonnées du point haut-gauche du rectangle, `width` sa largeur et `height` sa hauteur (le tout en pixels).



On peut par exemple définir notre balle comme un carré de 10 pixels de côté, disposée au point de coordonnées (100;50) :

```
balle = pygame.Rect(100, 50, 10, 10)
```

☞ *C'est le coin supérieur gauche de la balle qui est positionné au point* (100;50) *!*

Il faut ensuite afficher la balle sur l'écran, grâce à la fonction `pygame.draw.rect(surface, color, rect)`, où `surface` est la surface sur laquelle dessiner notre rectangle, `color` est la couleur utilisée, et `rect` est le rectangle lui-même. Par exemple :

```
pygame.draw.rect(ecran, BLANC, balle)
```

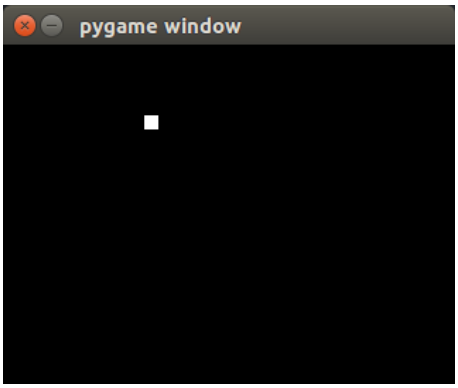
Notre programme devient :

```
import pygame

pygame.init()

LARGEUR_ECRAN, HAUTEUR_ECRAN = 320, 240
TAILLE_ECRAN = (LARGEUR_ECRAN, HAUTEUR_ECRAN)
NOIR = (0,0,0)
BLANC = (255,255,255) # Nouvelle couleur

ecran = pygame.display.set_mode(TAILLE_ECRAN)
balle = pygame.Rect(100,50,10,10) # Création de la balle (rectangle)

continuer = True
while continuer:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            continuer = False

    ecran.fill(NOIR)
    pygame.draw.rect(ecran, BLANC, balle) # Affichage de la balle sur l'écran
    pygame.display.flip()
```

Question 01 Modifier le programme précédent pour afficher la balle :

1. en haut à gauche de l'écran
2. en bas à droite de l'écran
3. au milieu de l'écran

2.5 Ça chauffe



Le programme précédent affiche ~~un gros amas de pixels blanc~~ une balle blanche sur un écran noir, et pourtant votre processeur est très sollicité (vous pouvez vérifier). Cela vient du fait que le nombre de tours de boucle effectués chaque seconde (ou le **nombre d'images par seconde**) est très important, et ce même si l'image affichée est toujours la même.
Vous pouvez le vérifier par vous même : créez une variable `c` égale à 0, incrémentée à chaque tour de boucle, et affichée à la fin du programme (pas dans la boucle !). Laissez le programme tourner 10s et regardez la valeur de `c` . Selon la puissance de l'ordinateur, on tourne entre 3000 et 8000 images par secondes !

Pour diminuer ce nombre, on peut inscrire une pause dans la boucle, grâce à la fonction `pygame.time.wait(ms)` où `ms` est le temps de pause en millisecondes. Avec une pause de 10ms, on limite le nombre d'images à 100 par seconde, ce qui est très raisonnable, et largement suffisant pour notre jeu. De plus, votre processeur sera beaucoup moins sollicité !

Rajoutez ainsi l'instruction suivante dans la boucle (en dernier) :

```
pygame.time.wait(10)
```

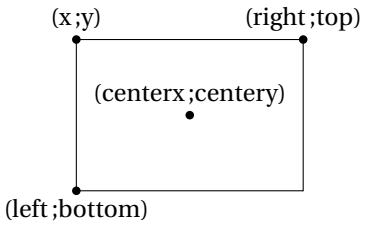
2.6 Et maintenant, elle bouge

Revenons à notre jeu.
Pour faire bouger notre balle, il faut mettre à jour ses coordonnées à chaque tour de boucle.
Un objet de type `Rect` possède plusieurs attributs auxquels il est possible d'accéder et qu'il est possible de modifier directement, en écrivant `objet.attribut` .

▷ ***L'encapsulation n'est pas respectée !***

En effet, il semble qu'une telle écriture ne respecte pas le principe d'encapsulation, mais en apparence seulement.
Par exemple, pour accéder aux coordonnées de notre balle, il suffit d'écrire `balle.x` ou `balle.y` .

Il est possible d'accéder à plusieurs attributs de notre balle, listés ci-dessous.
Par exemple, `balle.bottom` représente l'ordonnée du côté bas de la balle, `balle.right` l'abscisse du côté droit, ou encore `balle.centery` est l'ordonnée du centre de la balle.



Pour déplacer notre balle vers la droite, il nous suffit d'incrémenter son abscisse de 1 à chaque tour de boucle, en écrivant :

```
balle.x += 1
```

Cet incrément est à effectuer **dans la boucle** d'affichage, afin que la balle se déplace vers la droite en continu.
Notre programme devient :

```
1  import pygame
2
3  pygame.init()
4
5  LARGEUR_ECRAN, HAUTEUR_ECRAN = 320, 240
6  TAILLE_ECRAN = (LARGEUR_ECRAN, HAUTEUR_ECRAN)
7  NOIR = (0,0,0)
8  BLANC = (255,255,255)
9
10 ecran = pygame.display.set_mode(TAILLE_ECRAN)
11 balle = pygame.Rect(100,50,10,10)
12
13 continuer = True
14 while continuer:
15
16     for event in pygame.event.get():
17         if event.type == pygame.QUIT:
18             continuer = False
19
20     balle.x += 1 # <== ICI !
21
22     ecran.fill(NOIR)
23     pygame.draw.rect(ecran, BLANC, balle)
24     pygame.display.flip()
25
26     pygame.time.wait(10)
```

Pour représenter le déplacement de la balle, on peut également utiliser un **vecteur vitesse**. Il s'agit d'une simple liste `[vx, vy]` représentant le déplacement sur chaque axe de la balle à chaque tour de boucle. Par exemple, on définit **avant la boucle** :

```
vitesse = [2,1]
```

ce qui représente un déplacement de 2 pixels vers la droite et 1 pixel vers le bas. On écrit alors **dans la boucle** :

```
balle.x += vitesse[0]
balle.y += vitesse[1]
```

Notre programme devient :

```
import pygame

pygame.init()

LARGEUR_ECRAN, HAUTEUR_ECRAN = 320, 240
TAILLE_ECRAN = (LARGEUR_ECRAN, HAUTEUR_ECRAN)
NOIR = (0,0,0)
BLANC = (255,255,255)

ecran = pygame.display.set_mode(TAILLE_ECRAN)
balle = pygame.Rect(100,50,10,10)

vitesse = [2,1] # Définition du vecteur vitesse

continuer = True
while continuer:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            continuer = False

    balle.x += vitesse[0] # Mise à jour de la position
    balle.y += vitesse[1] # suivant le vecteur vitesse

    ecran.fill(NOIR)
    pygame.draw.rect(ecran, BLANC, balle)
    pygame.display.flip()

    pygame.time.wait(10)
```

Question 02 Modifier le programme précédent pour que la balle se déplace :

1. vers le bas de l'écran
2. vers le coin haut droite de l'écran, en partant du coin inférieur gauche

▷ *La balle traverse l'écran, elle ne rebondit pas !*

C'est normal ! Il va falloir coder le rebond, expliquer à Python que doit faire la balle lorsqu'elle arrive contre le bord de l'écran. Et c'est plus facile qu'il n'y paraît.

2.7 Rebonds dans tous les sens

Pour faire rebondir la balle, on utilise une simple condition.

```
if balle.right > LARGEUR_ECRAN:
    vitesse[0] = -vitesse[0]
```

Lorsque le coin droit de la balle dépasse la largeur de l'écran, on inverse la vitesse horizontale. De cette manière, lorsqu'on appelle l'instruction `balle.x += vitesse[0]`, le nombre ajouté est cette fois négatif, et la balle « recule » vers la droite.

Question 03 Placer ces instructions dans le programme, juste avant la mise à jour des coordonnées de la balle.

Question 04 Modifier le programme pour que la balle rebondisse sur tous les coins de l'écran.

☞ *Dans le jeu final, le jeu s'arrête si la balle rebondit sur le mur gauche ou droit... Nous prendrons ceci en compte plus tard.*

2.8 Une raquette qui bouge

Il est temps de créer les raquettes et de les faire bouger. Dans cette partie, on oublie quelques temps la balle pour se concentrer sur la création d'une raquette et la gestion de son déplacement.

Question 05 Modifier le programme précédent afin qu'il affiche une raquette sur l'écran.
On définira une raquette comme un rectangle « long et étroit », placé près du bord gauche de l'écran :

```
raquette = pygame.Rect(10,100,5,50)
```

☞ *Ne pas supprimer le code faisant référence à la balle, mais le mettre plutôt en commentaire.*

Afin que le joueur puisse déplacer la raquette, il faut d'abord définir un moyen de la déplacer. Par exemple, on peut utiliser les flèches directionnelles, ou les touches ZQSD.
Grâce à la fonction `pygame.key.get_pressed()`, on peut savoir quelles touches sont préssées lors de l'appel de la fonction.
La fonction retourne un n-uplet de booléens, représentés par des 0 et des 1 :

```
(0, 0, 0, 1, 0, ... , 1, 0)
```

Chaque 1 signale que la touche correspondante est enfoncée. Pour savoir quelle touche est associée à quel indice dans le n-uplet, on peut regarder dans la documentation de *Pygame* :

`https://www.pygame.org/docs/`

▷ ***Quelle horreur, c'est en anglais !***

Comme beaucoup de documentations en informatique ! L'important, c'est avant tout de se repérer : dans le menu en haut de la page, on peut y voir *Most useful stuff* (les « trucs les plus utilisés ») et le sous-module qui nous intéresse, `key`.

En suivant le lien, on tombe sur la documentation de `pygame.key`, *« pygame module to work with the keyboard »*. En descendant un peu sur la page, on tombe sur le Graal, la liste des constantes associées aux touches du clavier.

Celles qui nous intéressent sont `K_UP` et `K_DOWN`, respectivement associées aux touches *Haut* et *Bas* du clavier.
Ces valeurs sont égales aux indices des touches *Haut* et *Bas* dans le n-uplet renvoyé par `pygame.key.get_pressed()`.

Ainsi, pour savoir si la touche *Haut* est enfoncée, on écrira la condition suivante :

```
if pygame.key.get_pressed()[pygame.K_UP]:
```

Si la touche *Haut* est enfoncée, il faut déplacer la raquette vers le haut, en mettant à jour son ordonnée.
Notre programme affichant la raquette devient alors :

```
import pygame

pygame.init()

LARGEUR_ECRAN, HAUTEUR_ECRAN = 320, 240
TAILLE_ECRAN = (LARGEUR_ECRAN, HAUTEUR_ECRAN)
NOIR = (0,0,0)
BLANC = (255,255,255)

ecran = pygame.display.set_mode(TAILLE_ECRAN)

raquette = pygame.Rect(10,100,5,50)

continuer = True
while continuer:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            continuer = False

    if pygame.key.get_pressed()[pygame.K_UP]:
        raquette.y -= 1

    if pygame.key.get_pressed()[pygame.K_DOWN]:
        raquette.y += 1

    ecran.fill(NOIR)
    pygame.draw.rect(ecran, BLANC, raquette)
    pygame.display.flip()

    pygame.time.wait(10)
```

☞ *La gestion de la balle n'est pas implémentée dans ce programme*

Question 06 Modifier le programme précédent afin que la raquette ne puisse pas sortir de l'écran (!)

Notre raquette est en place... Si on replace la balle dans le jeu, le programme fonctionne mais la balle ne rebondit pas contre la raquette ! Il faut gérer la **collision** mais aussi le **rebond** de la balle.

Après avoir fait chauffer le processeur de votre ordinateur, il est temps de faire chauffer (un peu) votre cerveau !

2.9 Raquette et collisions

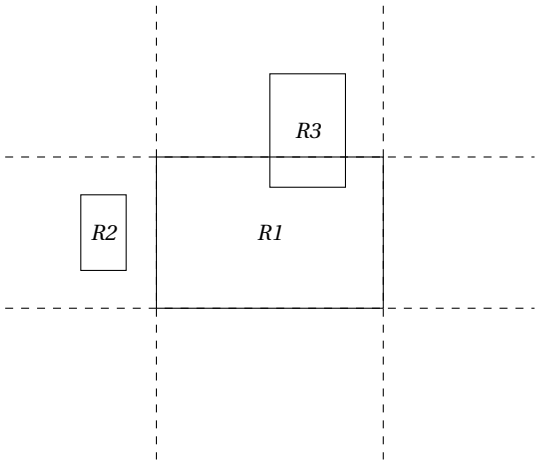
La gestion des collisions est un domaine compliqué en programmation. Elle se résume facilement en une question :

« À quelle condition deux objets se touchent-ils ? »

Elle est à la base de n'importe quel jeu. Par exemple, comment savoir si un personnage est sur le sol ? Est-ce que la boule de feu envoyée par cet ennemi a touché mon personnage ? Est-ce que ma souris pointe sur un objet particulier ?

Dans le cadre de ce TP, nous resterons en « mode facile », car nous allons gérer des collisions de rectangles « bien droits ».
Le but est de créer une fonction `collision(R1, R2)` qui renvoie `True` si deux rectangles `R1` et `R2` (comme notre balle et notre raquette) se touchent, et `False` sinon.

Comme souvent, un dessin est bien utile :



$R1$ et $R2$ ne sont pas en collision, mais $R1$ et $R3$ le sont

Grâce au dessin ci-dessus, on peut se convaincre que deux rectangles `R1` et `R2` ne sont **pas en collision** si et seulement si on se trouve dans un des cas suivants :

1. $R2$ est entièrement à gauche de $R1$
2. $R2$ est entièrement en dessous de $R1$
3. $R2$ est entièrement à droite de $R1$
4. $R2$ est entièrement au dessus de $R1$

Dans tous les autres cas, $R1$ et $R2$ sont nécessairement en collision.

☞ *On raisonne relativement à* $R1$*, mais le problème est symétrique.*

Chaque condition se traduit par une inégalité en Python. Par exemple, la condition 1 revient à `R2.right <= R1.left` (l'abscisse du côté droit de $R2$ est inférieure à l'abscisse du côté gauche de $R1$). Ainsi, notre fonction `collision` commence par :

```
def collision(R1, R2):
    if R2.right <= R1.left or ... :
        return False
    else:
        return True
```

Question 07 Compléter la fonction `collision` précédente.

Savoir si la balle et la raquette sont en collision n'est qu'un début, car il faut maintenant faire rebondir la balle.
Première chose : la balle rebondit lorsqu'elle est en collision avec la raquette. C'est évident, mais c'est la condition avant de calculer un éventuel rebond. La méthode est la suivante :

1. On évalue la position de la balle « juste avant » la collision :
   (a) Si la balle était entièrement à gauche ou entièrement à droite de la raquette, on inverse la vitesse horizontale.
   (b) Si la balle était entièrement en haut ou entièrement en bas de la raquette, on inverse la vitesse verticale.
2. Les cas (a) et (b) ne s'excluent pas : la balle peut revenir de là où elle est venue (on rebondit dans le coin)

Question 08 Compléter le code suivant, afin de gérer le rebond de la balle.

```
if collision(balle, raquette):
    if ......................... : # Cas (a)
        vitesse[0] = -vitesse[0]
    if ......................... : # Cas (b)
        vitesse[1] = -vitesse[1]
```

☞ *Pour cette question, ne pas hésiter à faire un dessin et/ou à découper des rectangles en papier !!*

La gestion des collisions ne se limite pas à la collision de rectangles « bien droits » (on parle de collision AABB, pour *Axis Aligned Bounding Box*). Elle inclut la collision de rectangles dont les côtés ne sont pas parallèles au repère, la collision de cercles, de formes plus complexes comme les polygones... et aussi la collision dite *pixel-perfect*, au pixel près !

Pour en savoir plus : `http://sdz.tdct.org/sdz/eorie-des-collisions.html`

2.10 Traversons les murs

Avant de terminer le jeu, découvrons un petit bug intéressant.

Faisons tourner le programme avec balle et raquette définies comme ci-dessous, dans une fenêtre 320x240 :

```
balle = pygame.Rect(200,100,10,10)
raquette = pygame.Rect(100,80,5,40)
vitesse = [15,0]
```

La balle se déplace très vite horizontalement, si vite qu'on semble distinguer deux balles, mais la balle rebondit correctement sur la raquette. Cependant, si on décale la raquette de 5 pixels vers la gauche, c'est le drame...

```
balle = pygame.Rect(200,100,10,10)
raquette = pygame.Rect(105,80,5,40)
vitesse = [15,0]
```

Question 09 D'où vient l'erreur ?

3 Pour aller plus loin : on se partage le travail !

Le jeu est presque terminé. En l'état actuel, vous êtes capable de mettre en place deux raquettes et de lancer une balle.

Question 10 Mettre en place la balle et deux raquettes, et jouer.

Cependant, il reste pas mal de choses à coder :

- gérer le début de partie : un joueur possède la balle et appuie sur *Espace* pour la lancer
- gérer la fin de partie : la partie se termine lorsque la balle atteint le côté gauche ou droit de l'écran
- la gestion et l'affichage des scores
- mettre un filet au milieu de la fenêtre

De plus, on peut implémenter beaucoup de fonctionnalités, comme :

- un mode 1 joueur : un mode « entrainement » dans lequel la balle rebondit sur le côté droit (ou gauche) de l'écran
- un mode 1 joueur contre l'ordinateur : la raquette de droite bouge seule pour intercepter la balle
- la disposition d'obstacles au milieu de la table
- une balle qui accélère à chaque rebond
- le déplacement horizontal des raquettes

Cette liste est loin d'être exhaustive ! Comme dans un vrai projet, l'idéal serait de se répartir les tâches, par groupes de 2 à 4 élèves. On pourrait imaginer plusieurs équipes :

1. GUI DESIGN : Menus et scores
2. LEVEL DESIGN : Imaginer des situations de jeux différentes (modifier angle balle, plusieurs balles, obstacles...)
3. GAME DESIGN : Programmation
4. IA DEVELOPPEMENT : Gestion de l'« intelligence artificielle » du jeu

Question 11 Développer Pong 2.0 !

4 Bonus : exporter son programme

Si vous souhaitez que des joueurs profitent de votre jeu, il serait malvenu de leur dire :

« Pour jouer, c'est facile, installe EduPython, copie le code, et ... »

Heureusement, il est très facile d'exporter un programme Python et de le rendre utilisable sur n'importe quelle machine, grâce à **cx_Freeze**, dont le rôle est de transformer vos fichiers Python en exécutables `.exe` !

Tout est très bien expliqué ci-dessous :

`https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python/235020-distribuer-facilement-nos-programmes-python-avec-cx-freeze`