

# Vincent B. TS1

Notre projet :



## Groupe:

Vincent B.

Emeric R.

Luka M.

## Sommaire:

- 2 - Description et création du projet
- 2 - Prise d'exemples
- 3 - Description et création du projet, la suite
- 5 - Ressources
- 5 - Mon impact dans le projet

L'avancement du Projets/ La confrontation aux problèmes:

- 7 - La création du menu
- 8 - La création des boutons
- 10 - Résolution de problèmes avec les balles
- 12 - Les classes et leur utilité
- 15 - Problèmes de lags
- 17 - Animation et compteur d'image
- 19 - Les missiles de GLaDOS
- 21 - Stockage des données et fichier JSON
- 23 - Les Noxines
  
- 31 - L'impact de ce projet sur mon avenir
- 32 - Annexes

## Description et création du projet:

Nous voulions faire un projet qui ait du style, qui soit très visuel et amusant. Nous avons donc choisi de faire un jeu, cela semblait la meilleure option. Nous avons en groupe choisi de faire un jeu de Pong, car cela permettait d'avoir des bases faciles, que tout le monde peut comprendre tout en laissant les améliorations et les ajouts possibles.

## Prise d'exemples:

Pour trouver de l'inspiration, nous avons regardé des exemples de Pong téléchargeables sur Github.

04 | 09

0 2



This is Pong!

Full Game  
game runner - Part 1  
bouncing ball - Part 2  
game loop - Part 3  
collision detection - Part 4  
computer AI - Part 5

This is a javascript version of Pong.

Press 1 for a single player game.  
Press 2 for a double player game.  
Press 0 to watch the computer play itself.

Player 1 moves using the Q and A keys.  
Player 2 moves using the P and L keys.

Esc can be used to abandon a game.

sound:   
stats:   
footprints:   
predictions:

frame: 44  
fps: 60  
update: 1ms  
draw: 2ms

Le dernier jeu est celui qui nous a motivé le plus à faire un pong, la possibilité de jouer contre un bot était vraiment intéressante.

Mais toutes ces versions sont très classiques, peu originales et très semblables ...

## Description et création du projet, la suite:

C'est alors là que nous avons décidé de nommer notre projet: L'Ultimate Pong Game, le mot "Ultimate" signifie que nous allons plus loin que de refaire un pong classique et que nous créons plusieurs ajouts.



Nous voulions dès le départ permettre plusieurs modes de jeu tel qu'un mode solitaire, ou un mode multijoueur.



Pour ne pas trop complexifier le jeu tout en l'améliorant et en le rendant plus original, nous avons décidé de créer des "modes", des ajouts qu'on peut activer et désactiver pour modifier le jeu. Nous avons eu besoin pour cela de créer un nouveau menu avec des boutons.

Emeric s'est ensuite occupé de la partie musique (qu'il vous détaillera dans son dossier) à l'aide du module minim. J'ai ensuite créé les boutons de musique et de son pour permettre à l'utilisateur d'écouter les musiques ou non.

**SCORE = 0**



Pour donner une touche un peu plus personnelle au jeu et pour avoir quelque chose d'amusant, nous avons décidé de faire des boss.

Nous avons réalisé chacun notre Boss de façon autonome.

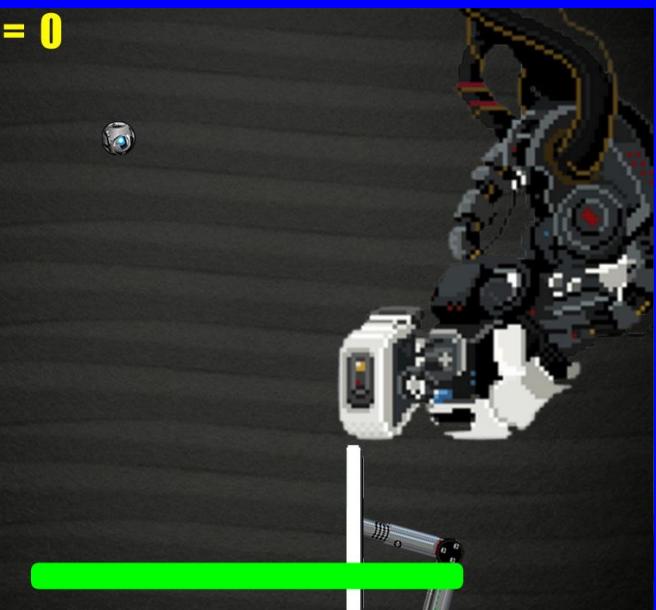
Le mien est celui sur la première capture d'écran à gauche.

**SCORE = 5**



(Boss de Luka)

**SCORE = 0**



(Boss d'Emeric)

## L'INFERNAL PROGRAMME D'ENTRAÎNEMENT CÉRÉBRAL™

du Dr Kawashima

Pouvez-vous rester CONCENTRÉ ?



Le boss de Luka est inspiré du jeu nintendo d'entraînement cérébrale. Ce boss vous donnera des multiplications à résoudre dans un temps très limité. Et attention, en cas de mauvaise réponse, il augmentera la vitesse des balles et réduira la taille de la raquette.

Les autres boss sont résumés plus tard.

## Ressources:

Nous avons codé ce projet en python.

Il a été réalisé dans l'environnement Processing-3, c'est un environnement de programmation qui permet de créer une fenêtre graphique.

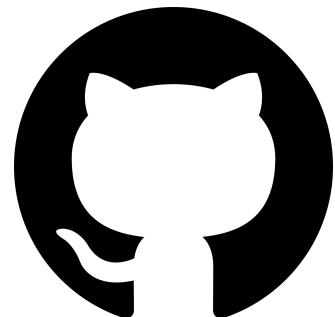


Nous avons également eu besoin d'importer certains modules, Emeric a eu besoin du module minim qui permet de jouer des sons ou musiques, et j'ai eu besoin du module json, qui permet d'écrire des dictionnaires python dans un fichier json que l'on pourra récupérer plus tard en ouvrant le programme.

J'ai personnellement utilisé Github pour permettre au groupe de télécharger plus simplement les nouvelles versions.

(Le projet est toujours en ligne et téléchargeable sur ce lien:

<https://github.com/Darkmauros/Ultimate-Pong-Game> )



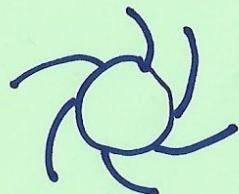
## Mon impact dans le projet:

J'ai créé les fondations du projet qui nous permettent de changer de menu grâce à la variable menuVar que j'explique dans la partie technique.

Je me suis aussi occupé de faire la majorité des modes: un nombre de balle plus grand, le champs de force, les portails et le modes hardcore.

Mais aussi et surtout le Trou noir, qui m'a permis d'utiliser de la physique, en voici un petit aperçu que je développerai en détail lors de l'oral. J'expliquerai aussi comment nous avons utilisé les vecteurs pour certains objets.

O balle



black hole

$$\text{masse} = 2,7 \times 10^{-3} \text{ kg}$$

$$= 2,7 \text{ g}$$

masse =  $7 \times 10^{15}$   
un réel trou noir pèse  
bien plus, on peut  
en réalité le  
comparer à une  
petite planète

comparatif:  
masse soleil =  $2,0 \times 10^{30}$  kg  
masse lune =  $7,3 \times 10^{22}$  kg  
masse Terre =  $6,0 \times 10^{24}$  kg  
 $G = 6,67 \times 10^{-11} \text{ N} \cdot \text{m}^2 \cdot \text{kg}^{-2}$   
(constante de gravitation)

formule de gravitation:

$$g = \frac{G \times m_1 \times m_2}{d^2}$$

On suppose que la balle ne subit aucune force.

référentiel: {point dans l'espace supposé galiléen}. objet: balle.

Bilan des forces: poids du trou noir.

Or après la 2<sup>e</sup> loi de Newton

$$\sum \vec{F}_{\text{ext}} = \frac{d\vec{P}}{dt} = \vec{P} \quad \text{la balle possède une masse constante.}$$

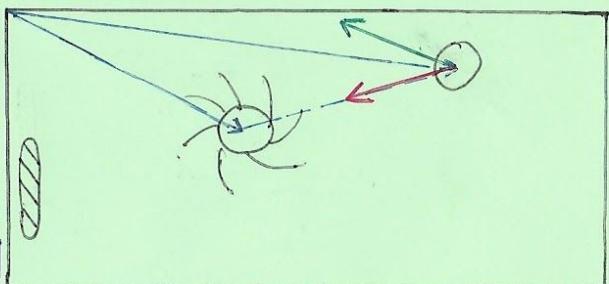
$$\text{Donc } \frac{d\vec{P}}{dt} = m\vec{a} \quad \Leftrightarrow \quad m\vec{a} = m\vec{g}$$

$$\Leftrightarrow \vec{a} = \vec{g}$$

L'accélération de la balle dépend du vecteur  $\vec{g}$  qui possède ces caractéristiques

- direction: centripète au trou noir
- sens: vers le trou noir
- point d'application: balle
- norme:  $\|\vec{g}\|$

Schéma:



→ vecteur position

--- distance balle - trou noir

→ vecteur vitesse

→ vecteur accélération } non à l'échelle

Un petit aperçu du travail sur le trou noir se trouve sur la page suivante

J'ai aussi complètement créer le Boss n°2 du jeu: Nox

J'en parle en détail lors de la partie technique.

Je me suis aussi occupé en grande partie de la partie graphisme et des images.

Mais tout cela paraît facile et rapide, or le projet nous a confronté à de nombreux problèmes. Je vais vous expliquer dans la prochaine partie comment on les a résolus de manière technique et précise.

## L'avancement du Projets/ La confrontation aux problèmes:

### Création du menu:

Au tout début j'ai pensé à créer plusieurs variables de menu (capture ci-dessus) et d'appeler la fonction correspondante à la variable qui est vraie.

(*exemple en langage naturel: si optionPong est vraie, alors jouer la fonction PongSolo()* )

```
menuOption = False  
menu = True  
menuPongSolo = False
```

Très vite, je me suis aperçu que cette idée n'était pas correcte car on ne peut être que dans un seul menu à la fois. On a donc besoin d'une unique variable.

J'ai alors créé la variable "menuVar" qui contient le nom du menu (*sous forme de string(chaîne de caractère)*) qui doit être présenté.

Voici une capture d'une ancienne version du programme montrant la fonction draw de processing, dans laquel je demande de présenter **le menu correspond à la variable**

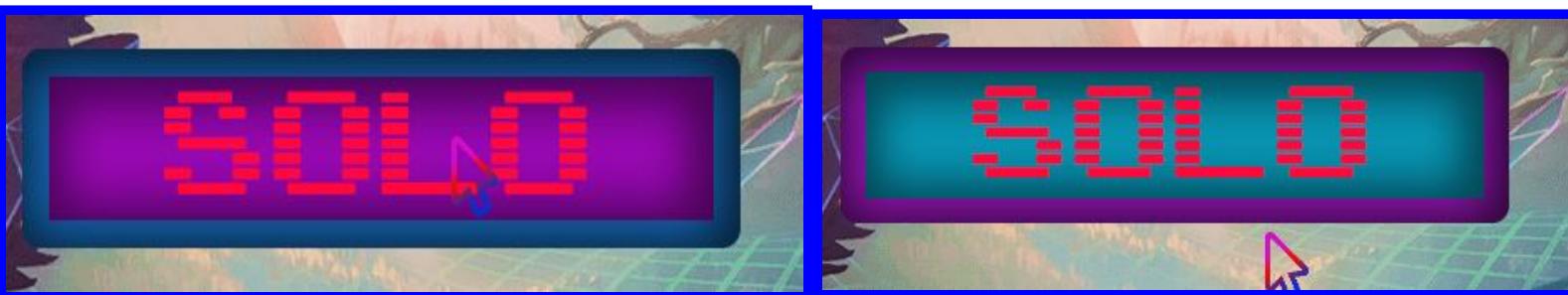
```
def draw():  
    noCursor()  
    frameRate(60)  
    print("fps=", int(frameRate))  
    if menuVar == "menue" :  
        menue()  
    if menuVar == "pongSolo" :  
        pongSolo()  
    if menuVar == "menuOption":  
        menuOption()
```

(*menue() est le menu principale  
pongSolo() est le jeu  
et menuOption() est le second  
menu permettant la modification des  
modes* )

## Création des boutons:

Après avoir créer différents menus, nous avons eu besoin de boutons pour permettre à l'utilisateur de lancer le jeu et de modifier des options.

Voici le programme de notre bouton "Solo" qui est animé quand on passe la souris dessus.



D'abord, nous avons correctement placé le bouton en chargeant une image (avec la fonction `image` intégré de Processing: `image(imgBoutonSolo...)` ).

Rappel: la fonction `image` possède ces paramètres:

`image(nomDeL'image, positionX, positionY, largeur, hauteur)`

Pour cela nous avons utilisé les mots clefs `width` et `height` qui renvoie la taille de la fenêtre du jeu. (*notre fenêtre fait 900x700 pixels, on a donc width = 900 et height = 700*)

**Explication:** `width/2` représente la moitié de la fenêtre et `-200` représente la moitié de la taille du bouton.

On a donc `width/2-200` qui permet de centrer le bouton sur l'axe des abscisses.

```
#----- Bouton Solo -----
if mouseX > width/2-200 and mouseX< width/2-200+400 and mouseY>height/2.2 and mouseY<height/2.2+100:
    #width/2-200 = gauche et width/2-200+400 = droite
    imgBoutonSolo = loadImage("boutonSolo2.png")
    if mousePressed:
        lancementDuJeu("solo")
else:
    imgBoutonSolo = loadImage("boutonSolo1.png")
image(imgBoutonSolo , width/2-200 ,height/2.2 , 400, 100)
#rect(width/2-200,height/3,400,100)
```

Nous avons établi d'une manière semblable l'ordonnée, la largeur et la hauteur du bouton.

Par la suite, nous avons eu besoin de détecter quand la souris est sur le bouton, nous avons pour cela un *if* qui réalise 4 tests.

*schéma simplifié à la page suivante*

**Explication:** Pour faire simple, nous testons si la souris ne se trouve pas dans chaque zone, si elle n'est dans aucune, elle sera forcément sur le bouton.



Cela nous permet aussi d'afficher différentes images lorsque la souris est sur le bouton ou ne l'est pas (*il y a une fonction loadImage dans le test, et une autre fonction en dehors*)

Enfin, pour lancer le jeu lors d'un clic, on utilise le test *if mousePressed*: et on lance alors la fonction LancementDuJeu() qui lance le jeu.

## Résolution de problèmes avec les balles:

Je vais maintenant vous présenter le premier problème inattendu auquel j'ai fait face.

*C'est difficile de montrer cela sur une capture, j'ai donc représenté le mouvement de la balle par des flèches noires.*



De temps en temps, il arrivait qu'une balle ne fasse que des aller-retour entre haut et bas et que cela devienne injouable. J'ai montré sur cette capture mon premier indice à la résolution de ce problème.

Les valeurs entre les crochets sont les valeurs du vecteur vitesse de la balle (x,y,z) ( $z = 0$  car nous sommes dans un plan). On peut donc voir que la vitesse dans l'axe des ordonnées de la balle est bien trop grande:

La balle se déplace de cette vitesse 60 fois par seconde, c'est à dire  $60 \times 164 = 9840$  pixels! Notre fenêtre en fait seulement 700 de hauteurs.  $9840/700 = 14$

La balle rebondit alors entre le haut et la bas de la fenêtre 14 fois par seconde.

Voici comment j'ai contourner le problème.

J'ai créer la fonction "antiBugY(self)" (*le mot clef "self" signifie que c'est une méthode de classe, je détaille cela dans la prochaine partie*) que j'exécute pour chaque balle lorsqu'elle touche un mur.

```
def antiBugY(self):
    if self.vecteurVitesse.y > 20: #pour
        self.vecteurAcceleration.y -= 3
    if self.vecteurVitesse.y <= -20:
        self.vecteurAcceleration.y += 3
```

**Explication:** Lorsque la vitesse en ordonnée de la balle est supérieur à 20 (la balle va trop vite vers le haut), je la réduit. Et je fais la même chose quand la

vitesse est inférieur à -20 (la balle va trop vite vers le bas). Cela permet de définir une limite à la vitesse en ordonnée de la balle et de résoudre le problème.

Pour reprendre le même calcul qu'au départ. On peut faire  $60 \times 20 = 1200$  pixels.  
 $1200 / 700 = 1.7 \approx 2$ . A sa vitesse maximale, la balle peut maintenant rebondir 2 fois par seconde. Ce qui reste jouable.

## Les classes d'objets et leur incroyable utilité:

Nous voulions faire un mod permettant de jouer avec plusieurs balles autonomes. Mais c'était impossible de copier-coller chaque variable de l'unique balle. J'ai alors décidé de faire une classe pour les balles, ainsi qu'une liste ("listeBalles") répertoriant chaque balle.

```
#----- Balle -----+
class Balle:           Nom de la classe
    def __init__(self):
        self.vecteurPosition = PVector(width/2,height/2)
        self.vecteurVitesse = PVector(0,0)
        self.vecteurAcceleration = PVector(0,0)
        self.R = random ((difficulty(50,200)),250) #couleurs
        self.V = random (50,(difficulty(255,-130)))
        self.B = random (50,(difficulty(255,-130)))
        self.rainbowR = int(random(-3,3))
        self.rainbowV = int(random(-3,3))
        self.rainbowB = int(random(-3,3))
        self.alive = True
        self.tpPossible = True #pour les portails
        self.angle = 0 #pour la balle weathley
        self.rand1 = 0 #pour utiliser la tp avec le boss Nox
        self.rand2 = 0
        self.diametre = int(random(19,21))
        self.initialised = False #pour permettre au boss Nox de pas totalement reset la balle
        self.trail = 0

    def display(self):           Une fonction de la balle
        if self.alive == True:
            noStroke()
            fill(self.R ,self.V ,self.B )
            ellipse(self.vecteurPosition.x,self.vecteurPosition.y,self.diametre,self.diametre)
```

Annotations sur le code Python:

- Un pointeur noir pointe vers "class Balle:" avec l'annotation "Nom de la classe".
- Un pointeur noir pointe vers "self.alive" avec l'annotation "Si elle est encore en jeu ou non".
- Deux accolades noires pointent vers "self.R", "self.V", "self.B" et "self.rainbowR", "self.rainbowV", "self.rainbowB" avec l'annotation "Sa couleur".
- Une accolade noire pointe vers "self.vecteurPosition", "self.vecteurVitesse", "self.vecteurAcceleration" avec l'annotation "Sa position".
- Deux accolades noires pointent vers "self.R", "self.V", "self.B" avec l'annotation "Sa vitesse et son accélération".
- Un pointeur noir pointe vers "self.diametre" avec l'annotation "Son diamètre".
- Un pointeur noir pointe vers "def display(self):" avec l'annotation "Une fonction de la balle".

Sur cette capture, on peut voir 14 variables, certaines servent à définir la balle elle-même, et d'autres comme `self.tpPossible` servent à l'association des balles avec les modes. Ce sont les attributs uniques à chaque balle.

Les balles ont ensuite des fonctions.

```
def display(self):
    if self.alive == True:
        noStroke()
        fill(self.R ,self.V ,self.B )
        ellipse(self.vecteurPosition.x,self.vecteurPosition.y,self.diametre,self.diametre)
```

Voici une fonction que peut exécuter chaque balle.

Cette fonction permet d'afficher la balle avec ses propres coordonnées, ses propres couleurs et sa propre taille et seulement si elle n'a pas quitté la zone de jeu.(grâce à la fonction `ellipse` qui dessine un disque et `fill` qui change la couleur du dessin)

```
def updateBalls():
    for Balle in listeBalles:
        Balle.display()
        Balle.rebondir()
        Balle.update()
```

Cette partie du programme se trouve dans le *def draw* de Processing et est donc exécutée 60 fois par seconde.

C'est à dire que chaque balle de la liste bouge (*balle.update()*) ,rebondit s'il y a un mur (*balle.rebondir()*) et se dessine (*balle.display()*) 60 fois par seconde.

```
def initialisationBalles():
    global listeBalles
    listeBalles = []
    for i in range(0,nombreBalle):
        listeBalles.append(Balle())
```

**Explication:** Pour créer des balles, on créer la liste “listeBalles” (vide) qui est utilisé dans la capture juste au dessus.

Le “global listeBalles” permet l’autorisation de modifier la liste “listeBalles” dans tout

le programme.

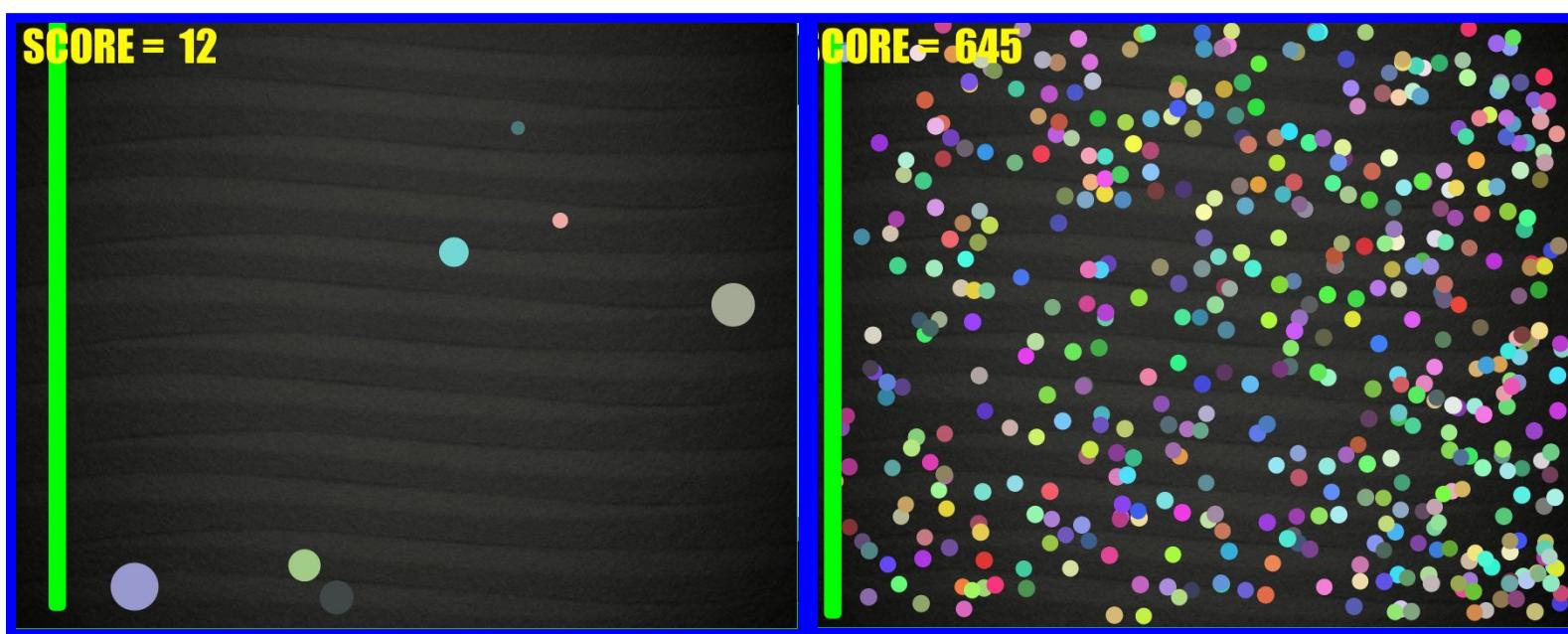
Les dernières lignes servent à ajouter autant de balle dans la liste que la variable “nombreBalle” a comme valeur.

Exemple: Si la variable “nombreBalle” est égal à 3, on ajoute 3 balles.

**SCORE = 3**



Tout cela nous permet d'augmenter le nombre de balle très facilement. On peut voir sur cette capture 5 balles aux couleurs uniques.



On peut ensuite s'amuser à modifier les paramètres (ici le diamètre et le nombre de balle) pour donner des choses amusantes.

Nous avons ensuite réutiliser le système de classe pour la plupart des modes du jeu ainsi que les boss.

## Problèmes de lags:

Par la suite, nous avons continué d'utiliser des images dans notre jeu, j'ai finis par remarquer que le nombre d'image par seconde (IPS) de notre programme diminuait. C'était problématique car la fonction draw de Processing s'exécute normalement 60 fois par seconde, si elle s'exécute seulement 20 fois, le programme entier (avec les déplacements des balles compris) est 3 fois plus lent.



Sur cette capture d'une précédente version, il n'y a que 6 images à charger et le nombre d'IPS chute à 25.

A ce moment, nous utilisions la même méthode que l'exemple de Processing qui consiste à charger l'image du dossier du jeu et de l'afficher juste après.

```
img = loadImage("moonwalk.jpg")      # Load the image into the program
image(img, 0, height / 2, img.width / 2, img.height / 2)
```

Pour contourner le problème, j'ai effectuer les chargements(*loadImage()*) de toutes les images seulement une fois au lancement du programme (avec la fonction *def setup*). J'ai globalisé les variables afin qu'elles soient récupérables au moment de leur affichage et on obtient donc ce "gros pavé" qui nous permet d'avoir un jeu stable.

```

Ultimate_Pong_Game

def setup():
    size(900,700)
    imgChangement = loadImage("loading.jpg")
    image(imgChangement,0,0,width,height)
    global imgTextePortail,Hardcore,backOption,back,fondPong,imgCase1, imgCase2,imgPortailBleu, imgPortailOrange, imgCurseur, imgForceField,imgBoutonNombre
    global imgLoseBoss1 , imgWinBoss1
    global texteForceField, imgBoutonOption, imgBlackHole, texteTrouNoir,imgGLaDOS1, imgGLaDOS2, imgMissile, imgPlateforme, imgWBalle,imgNox,imgNoxineG,imgNoxine
    global imgExplosionBleu, imgSpell, texteBoss1,imgRetry, imgReturn,imgTexteBoss2, imgNuage,imgBlueScreen,imgMusiqueOUI,imgMusiqueNON,imgSonOUI,imgSonNON,imgNe
    global imgKawashima,imgKawashima2,imgBonus,imgMalus
#----- Chargement des images -----
    imgTextePortail = loadImage("textePortail.png")#taille = 887x77
    Hardcore = loadImage("boutonHardcore.png")
    backOption = loadImage("ArcadeBackgroundVide.png")
    back = loadImage("ArcadeBackground.png")
    fondPong = loadImage("fondPong.png")
    imgCase1 = loadImage("case1.png")
    imgCase2 = loadImage("case2.png")
    imgBoutonNombre =loadImage("boutonNombre.png") #taille 100x 330 / taille d'une flèche 100x88
    imgPortailBleu = loadImage("Portailbleu.png") # 60x98
    imgPortailOrange = loadImage("PortailOrange.png")# 60x98
    imgCurseur = loadImage("curseur.png")
    imgBoutonOption = loadImage("boutonOption.png")
    imgBlackHole = loadImage("BlackHole.png") #250x250
    imgWinBoss1 = loadImage("GLaDOS/WIN.png")
    imgLoseBoss1 = loadImage("GLaDOS/Lose.png")

```

Eméric a ensuite utilisé le même procédé pour le chargement des sons et des bruitages.



On voit ici dans la console que les IPS sont presque toujours à leur maximum.

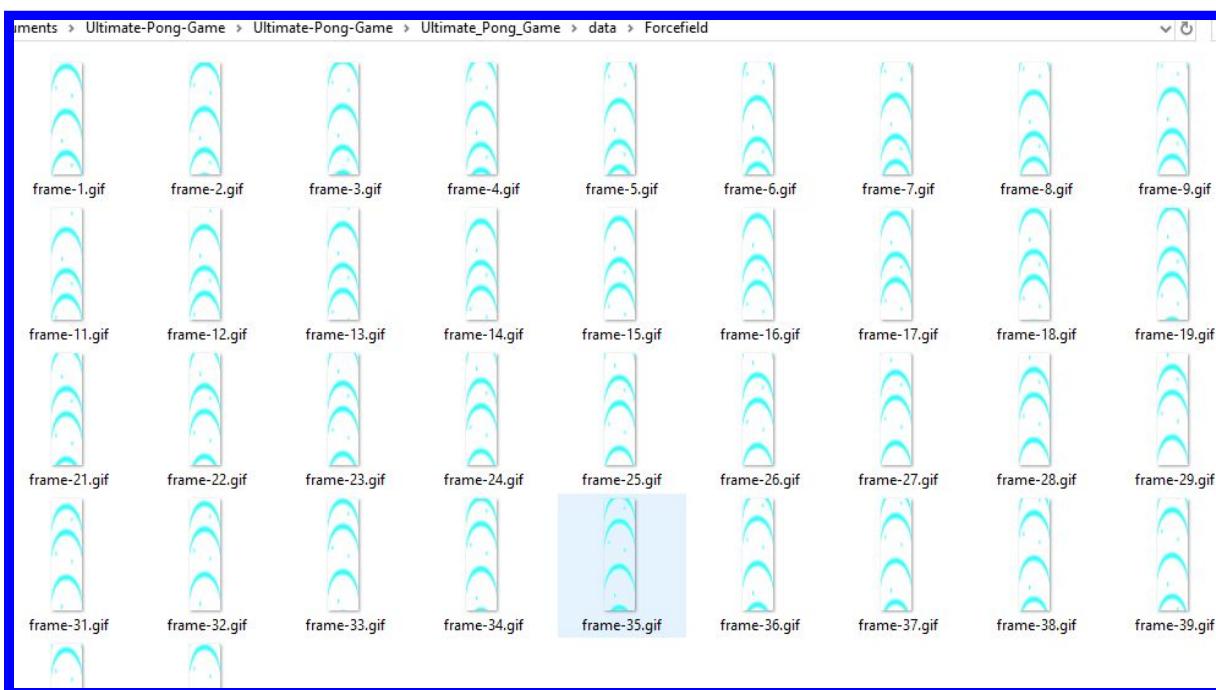
## Animation et compteurs d'image:

Lors de la création du Champs de Force, de Nox et surtout de ses Noxines, j'ai eu l'idée de faire des animations, qui changent complètement l'aperçu du jeu.



Pour en arriver là, j'ai utilisé les exemples de la bibliothèque de Processing et beaucoup d'images.

Tout d'abord, il nous faut une liste d'images qui en passant de l'une à l'autre rapidement, forment une animation. J'ai pris comme exemple le Champs de Force.



Remarque:  
les images sont ici  
en format gif mais  
fonctionnent  
comme des  
images normales.

Ensuite, je charge toutes les images (de la même manière que dans la précédente partie). Pour une meilleure praticité, on crée une liste que l'on remplit de 0 (pour avoir quelque chose à remplacer), et l'on remplace les 0 par les images chargés. Cela est possible car les images ont toutes le même nom, à l'exception de leur numéro.

```
for i in range(0,44):
    imgForceField.insert(0,0)
for i in range(1,43):
    imgForceField[i] = loadImage("Forcefield/frame-{}.gif".format(i))
```

On a donc une liste qui contient toutes les images (dans l'ordre).

```
def update(self):
    self.framecount += 1
    if self.framecount >= 42:
        self.framecount = 1
    self.display()
```

**Explication:** Pour que les images s'enchaînent. J'ai créé une variable `self.framecount` dans la classe du Champs de Force. Cette variable est incrémentée 60 fois par seconde. Le test du `framecount` permet de ne pas dépasser 42 (car on a que 42 images) et de relancer le comptage pour que cela se fasse en boucle.

**Explication:** Pour l'affichage des images, dans la fonction `display` de notre `Forcefield` (*qui est exécuté 60 fois par secondes*), on rend l'image semi-transparente avec la fonction `tint()` et on montre l'image ayant le même numéro que le compteur.

```
def display(self):
    tint(255,150)
    image((imgForceField[self.framecount]), self.positionX, 0, 200,height)
    noTint()
```

## Les missiles de GLaDOS:



GLaDOS est une intelligence artificiel créée à partir d'un esprit humain, elle est déterminé à finir ses tests. Et c'est vous qui en serez le cobaye. Dans ce jeu, le but est la détruire, ce ne sera pas simple car elle enverra une grosse dose de missile sur la plaque à chaque fois que vous la toucherez. En plus de cela, elle utilisera ses neurotoxines qui sont normalement inesquivables et qui nécessitent de bien viser la balle. GLaDOS ne l'a pas dit au joueur, mais ce test est en réalité impossible à résoudre.

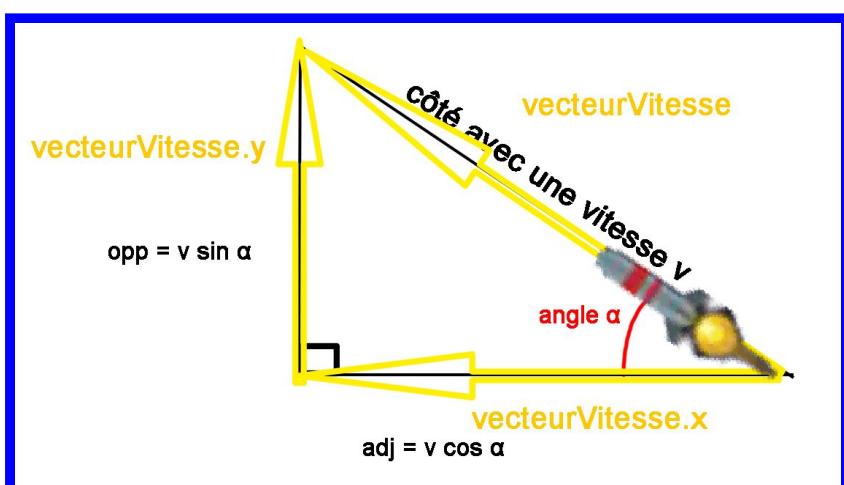
Lorsque Emeric a commencé la création de son Boss, on s'est mis d'accord sur le fait que le boss lancerait des missiles, et que cela serait un bon challenge à réaliser, et effectivement cela nous a demandé d'utiliser les cours de maths de trigonométrie et de géométrie.

**Remarque:** C'est la seule chose que j'ai co-réalisé envers le boss de Emeric, je ne connais pas le reste de sa partie.



Nous voulions que le missile puisse aller dans n'importe quel direction, avec toujours la même vitesse.

Comme nous utilisons des vecteurs (qui sont en fait des tuples) pour nos objets, nous avions besoin de connaître le vecteur de l'abscisse et le vecteur en ordonnée qui s'appelle respectivement "vecteurVitesse.x" et "vecteurVitesse.y".



Pour trouver les longueurs du triangle rectangle selon l'angle  $\alpha$  et la vitesse  $v$ , nous avons utilisé des formules de trigonométrie.

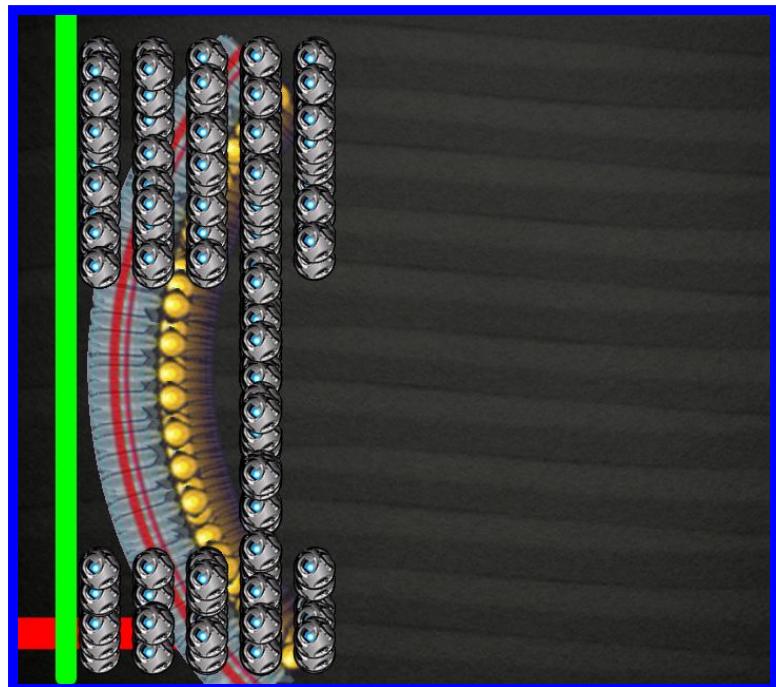
Voici une capture de la classe Missile de Emeric.

**Explication:** La fonction display(self) permet de les afficher et de tourner l'image sur elle-même pour qu'elle corresponde à l'angle.

```
class Missile:  
    def __init__(self):  
        self.degats = 1  
        self.alive = True  
        self.angle = random(135,225)  
        self.vecteurPosition = PVector(width/1.7 , 400)  
        self.vecteurVitesse = PVector(0, 0)  
        self.vecteurAcceleration = PVector(0, 0)  
  
    def display(self):#affichage  
        fill(255,0,0)  
        with pushMatrix():  
            translate (self.vecteurPosition.x, self.vecteurPosition.y)  
            rotate(radians(self.angle))  
            image(imgMissile,-120,-30,120,34)#120x34  
            #print(self.angle)  
  
    def reset(self):  
        v = 9|  
        x = v*cos(radians(self.angle))  
        y = sin(radians(self.angle))*v  
  
        self.vecteurVitesse = PVector(x , y)
```

La fonction reset(self) permet de calculer les deux valeurs qui composent le vecteur.

Dans cet exemple, on a une vitesse de 9 et un angle qui est aléatoire pour chaque missile. Ces valeurs sont modifiables facilement.



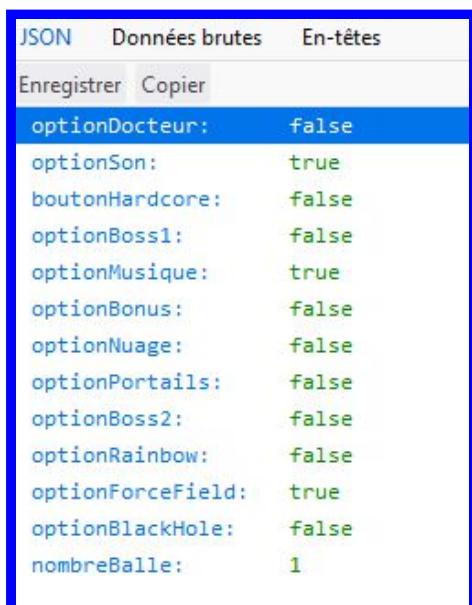
Sur cette capture réalisée avec un très grand nombre de missiles, on peut voir que tous les missiles ont effectivement la même vitesse (ils forment un arc-de-cercle et sont donc tous à la même distance de leur point de création).

## Stockage des données et fichier JSON:

A mesure que notre projet avançait, nous avons eu des idées plus ambitieuses les unes que les autres. Nous avons eu l'idée de faire un système de points récoltables et de créer une boutique pour que le joueur utilise ses points pour changer l'apparence de certains objets.

Pour cela j'ai eu besoin de m'intéresser à un système de stockage des données, pour que les points récoltés ne disparaissent pas à la fermeture du programme.

Malheureusement, par faute de temps, cette idée n'a pas encore vu le jour, mais nous avons gardé le système de sauvegarde pour sauvegarder les préférences de modes. Par exemple, si je coche la case Nox et que je ferme le programme puis que je le réouvre, la case sera cochée.



The screenshot shows a JSON configuration file with the following content:

```
JSON   Données brutes   En-têtes  
Enregistrer Copier  
optionDocteur: false  
optionSon: true  
boutonHardcore: false  
optionBoss1: false  
optionMusique: true  
optionBonus: false  
optionNuage: false  
optionPortails: false  
optionBoss2: false  
optionRainbow: false  
optionForceField: true  
optionBlackHole: false  
nombreBalle: 1
```

Le JSON est un type de fichier qui permet d'enregistrer des objets de type dictionnaire sous forme de texte dans un fichier.

(*un dictionnaire est une liste qui fonctionne comme un vrai dictionnaire, à un mot(une clef) est assigné un nombre, un autre mot, une variable ou ici une boolean*)

On peut voir sur cette capture que l'information peut-être directement lu, ici c'est par un navigateur web, mais la même chose est possible dans un programme.

C'est le module JSON qui permet de transformer les bibliothèques du programme en fichier JSON et inversement.

Dans la capture à la page suivante, je vous présente la fonction lectureDeFichier() qui est utilisée lors du lancement du programme.

**Explication:** Tout d'abord, le "try:" permet ici d'effectuer un essai sur les fonctions qui sont à l'intérieur. S'il n'y a aucun problème, c'est comme s'il n'existe pas, mais s'il y a un problème (une erreur de syntaxe, un fichier introuvable, une division par 0 ...), ce qui se trouve dans le "try" ne s'exécutera pas mais c'est à la place ce qui se trouve dans le "except" qui s'exécutera.

La ligne avec le "with..." permet d'ouvrir le fichier Json et de l'écrire dans la variable "texteLu".

Ensuite la ligne "dictionnaireInfo..." est une fonction spécifique au module json. Elle permet de transformer "texteLu" en un dictionnaire appelé "dictionnaireInfo".

En affichant ce dictionnaire à cet étape, on obtient la même chose que sur la capture précédente avec un navigateur internet.

Les variables du programme sont ensuite définies par leur variable correspondante dans le dictionnaire.

S'il y a une erreur ou que le fichier json n'est pas présent, pour ne pas créer de soucis de variables inconnues, on définit les variables par leur valeur par défaut. (*partie dans le "except:"*)

Les variables peuvent par la suite être utilisées et modifiées normalement par les boutons du menu option.

```
def lectureDeFichier():
    global texteLu
    #--- lecture du fichier
    try:
        with open("data/info.JSON","r") as f:
            texteLu = f.read()
            #print(texteLu)
    #--- écrasement du dictionnaire
    dictionnaireInfo = json.loads(texteLu)

    global optionSon,optionMusique,boutonHardcore,optionPortails,optionPortails,optionForceField,optionBlackHole,optionBoss1
    global optionBoss2, optionNuage, nombreBalle, optionBonus, optionRainbow, optionDocteur

    optionSon = dictionnaireInfo["optionSon"]
    optionMusique = dictionnaireInfo["optionMusique"]
    boutonHardcore = dictionnaireInfo["boutonHardcore"]
    optionPortails = dictionnaireInfo["optionPortails"]
    optionForceField = dictionnaireInfo["optionForceField"]
    optionBlackHole = dictionnaireInfo["optionBlackHole"]
    optionBoss1 = dictionnaireInfo["optionBoss1"]
    optionBoss2 = dictionnaireInfo["optionBoss2"]
    optionNuage = dictionnaireInfo["optionNuage"]
    nombreBalle = dictionnaireInfo["nombreBalle"]
    optionBonus = dictionnaireInfo["optionBonus"]
    optionRainbow = dictionnaireInfo["optionRainbow"]
    optionDocteur = dictionnaireInfo["optionDocteur"]
    except:
        optionSon = True
        optionMusique = True
        boutonHardcore = False
        optionPortails = False
        optionForceField = False
        optionBlackHole = False
        optionBoss1 = False
        optionBoss2 = False
        optionNuage = False
        nombreBalle = 1
        optionBonus = False
        optionRainbow = False
        optionDocteur = False
        print 'un probleme est survenu lors de la lecture des sauvegardes'
```

Lorsque le joueur est dans le menu des options, la fonction “ecritureDeFichier()” est exécutée en permanence.

**Explication:** D'abord, on crée un dictionnaire s'appelant “dictionnaireInfo” puis on rentre toutes les variables dedans.

A partir de la ligne 1790, on ouvre le fichier Json, on utilise la fonction .dumps() associé à la fonction .write() spécifique au module json qui permettent d'écrire le dictionnaire dans le fichier JSON.

```
1766 #----- écriture d'un fichier JSON -----
1767 def ecritureDeFichier():
1768     global dictionnaireInfo
1769     dictionnaireInfo = {}
1770     dictionnaireInfo["optionSon"] = optionSon
1771     dictionnaireInfo["optionMusique"] = optionMusique
1772     dictionnaireInfo["boutonHardcore"] = boutonHardcore
1773     dictionnaireInfo["optionPortails"] = optionPortails
1774     dictionnaireInfo["optionForceField"] = optionForceField
1775     dictionnaireInfo["optionBlackHole"] = optionBlackHole
1776     dictionnaireInfo["optionBoss1"] = optionBoss1
1777     dictionnaireInfo["optionBoss2"] = optionBoss2
1778     dictionnaireInfo["optionNuage"] = optionNuage
1779     dictionnaireInfo["nombreBalle"] = nombreBalle
1780     dictionnaireInfo["optionBonus"] = optionBonus
1781     dictionnaireInfo["optionRainbow"] = optionRainbow
1782     dictionnaireInfo["optionDocteur"] = optionDocteur
1783
1784 #le dictionnaire qui sera écrit en JSON (à n'utiliser que si le dictionnaireInfo est vide)
1785 """dictionnaireInfo = {"optionSon" : True, "optionMusique" : True,"boutonHardcore" : False, "optionPortails" : False,
1786             "optionForceField" : False,"optionBlackHole" : False , "optionBoss1" : False,"optionBoss2" : False,
1787             "optionNuage" : False,"nombreBalle" : 1}"""
1788
1789
1790 with open("data/info.JSON","w") as f:
1791     f.write(json.dumps(dictionnaireInfo, ensure_ascii=False))
1792
```

Voilà comment le système de sauvegarde fonctionne, il est extrêmement utile et nous resservira très probablement dans de futurs modes ou dans une peut-être futur boutique.

## Les Noxines:



Nox est l'antagoniste principale de l'animé Wakfu, et mon personnage préféré de part son histoire touchante et de sa puissance incontrôlable. J'ai décidé de recréer ce personnage sous forme de boss, et de recréer ses pouvoirs principaux, c'est à dire qu'il peut contrôler le temps, ralentir les balles, les accélérer, les arrêter, inverser leur trajectoire ou encore les téléporter.

Mais c'est aussi un inventeur qui a créé ses machines destructrices(les Noxines), il en invoquera de temps en temps, et celle-ci viendront exploser votre raquette tel des missiles téléguidés.



On peut voir sur cette capture l'explosion de Noxines sur la raquette.

Je vais principalement parler des Noxines dans cette partie et je vais détailler leur fonctionnement.

Je détaillerai aussi le fonctionnement des vecteurs et autres mécanismes que j'ai réutilisés à travers le programme.

```

----- Les Noxines ,les machines de Nox -----
class Noxine:
    def __init__(self,vecteurVitesse = PVector(15,0) ,vecteurAcceleration = PVector(-0.3,0), focus = "left"):
        self.alive = True
        self.vecteurPosition = PVector(width/1.8,height/2.5)
        self.vecteurVitesse = vecteurVitesse
        self.vecteurAcceleration = vecteurAcceleration
        self.framecount = 1
        self.framecountBis = 0
        self.angle = "droite"
        self.focus = focus #il ne peut être à droite qu'en multi
        self.vitesseMaxG = 7
        self.explosionFramecount = 1
        self.explosionFramecountBis = 0
        self.explosion = False

    def display(self):
        if self.angle == "gauche":
            a = imgNoxineG[self.framecount]
        else:
            a = imgNoxineD[self.framecount]
        image(a,self.vecteurPosition.x- 50 ,self.vecteurPosition.y-50, 100,100)

```

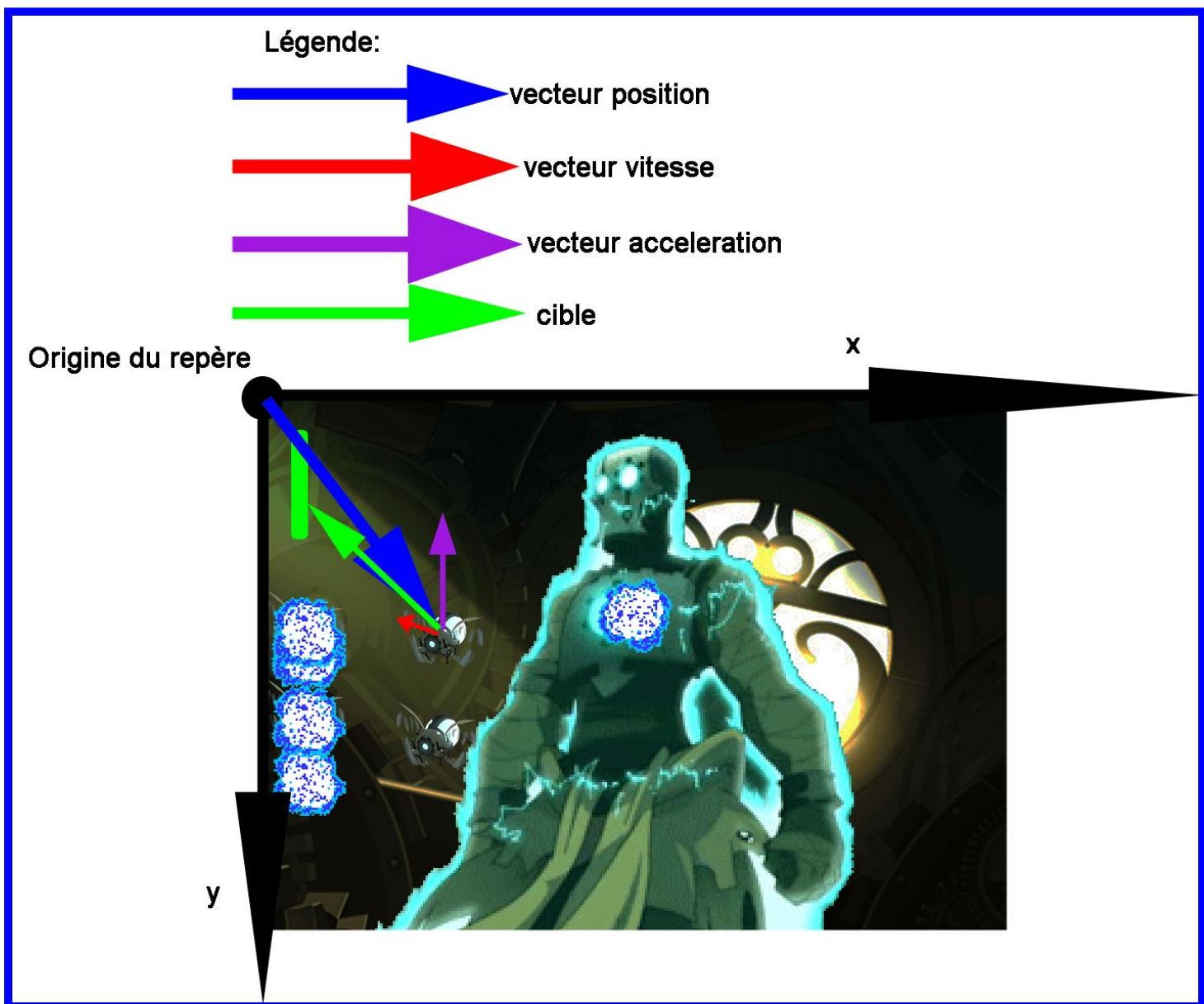
## Explication:

- la définition de la Noxine possède de nombreux paramètres que j'explique plus tard
- la variable “*self.alive*” permet savoir si la Noxine est encore fonctionnelle, cette variable est d'autant plus importante car j'ai besoin qu'après son explosion, la Noxine disparaisse.
- le “*self.vecteurPosition*” est un vecteur, les vecteurs sont en fait des tuples qui contiennent 2 variables: x et y.  
Le vecteur position permet de localiser la Noxine par rapport à l'origine qui est coin en haut à gauche de la fenêtre.
- le “*self.vecteurVitesse*” est celui qui donne la vitesse de la Noxine, il permet de modifier le vecteur position et donc de bouger la Noxine. On incrémente le vecteur position par le vecteur vitesse 60 fois par seconde.
- le “*self.vecteurAcceleration*” donne l'accélération de la Noxine. On incrémente le vecteur vitesse par le vecteur accélération 60 fois par seconde.
  
- “*self.framecount*” et “*self.framecountBis*” servent à animer la Noxine, le “*framecountBis*” incrémenté 60 fois par seconde, le “*framecount*” est incrémenté quand le “*framecountBis*” atteint environ la valeur de 4. Cela permet de ne pas incrémenter le “*framecount*” directement et de ralentir le défilement des images.
  
- le “*focus*” représente la direction dans laquelle la Noxine va se faire exploser. En jeu solo, il n'y a qu'une raquette à gauche, alors le “*focus*” sera sur la raquette de gauche. En multijoueur, il y aura une moitié de Noxine avec comme cible le joueur 1 et une autre moitié qui cible le joueur 2.

Voici une capture schématique représentant les différents vecteurs de la Noxine.

(la cible n'est pas un vecteur mais est lié à la raquette de gauche ou de droite)

(le vecteur acceleration est bien trop petit et n'est ici pas à l'échelle)



- "angle" cette variable est soit égal à "gauche", soit égal à "droite", elle change en fonction que le vecteur vitesse soit vers la gauche ou vers la droite. Cela permet d'afficher l'image de la noxine qui regarde à gauche ou celle qui regarde à droite.
- "self.vitesseMaxG" évite que le vecteur vitesse ne devienne trop grand et permet au joueur d'esquiver l'explosion.
- "explosionFramecount" et "explosionFramecountBis" font exactement la même chose que les "framecount" mais cela s'applique pour les images d'explosion lorsque la Noxine a atteint son objectif.
- "explosion" est une booléen qui permet de savoir quand la Noxine explose (cela permet de ne plus afficher l'image de la noxine et de lancer le "explosionFramecount" qui affiche des images d'explosion).

Voilà précisément l'utilité de toutes ces variables. Je vais maintenant vous présenter les fonctions de la Noxine.

La fonction “*display(self)*” présente sur la précédente capture du code permet à la Noxine d'afficher son image. On effectue un test pour savoir si son “angle” est “gauche” ou “droite” afin de choisir l'image correspondante à sa direction. “*self.framecount*” oscille entre 1 et 2. Si la Noxine va par exemple à gauche, on affichera alternativement les images: “noxine1G” et “noxine2G” pour créer l'animation de vol.

```
def update(self):
    ----- pour changer le sens de la noxine
    if self.vecteurVitesse.x < 0:
        self.angle = "gauche"
    else:
        self.angle = "droite"
    ----- pour la faire suivre la plaque
    if self.focus == "left":
        if P1.positionY > self.vecteurPosition.y:
            self.vecteurAcceleration.y = 0.2
        else:
            self.vecteurAcceleration.y = -0.2
    if self.focus == "right":
        if P2.positionY > self.vecteurPosition.y:
            self.vecteurAcceleration.y = 0.2
        else:
            self.vecteurAcceleration.y = -0.2
    ----- pour éviter que la noxine aille trop vite
    if self.vecteurVitesse.x < -self.vitesseMaxG:#gauche
        self.vecteurVitesse.x = -self.vitesseMaxG
    if self.vecteurVitesse.x > self.vitesseMaxG:#droit
        self.vecteurVitesse.x = self.vitesseMaxG
    ----- update des vecteurs
    self.vecteurPosition.add(self.vecteurVitesse)
    self.vecteurVitesse.add(self.vecteurAcceleration)
```

Voici la fonction “*update()*” de la Noxine, elle permet de viser la plaque et de bouger la Noxine.

Le premier test sert à savoir si le vecteur vitesse dans l'axe des abscisses est négatif, si oui, la noxine va à gauche et on change “*self.angle*”, si non, on fait l'inverse.

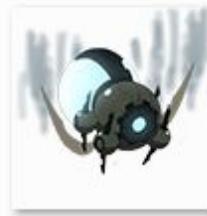
Le second test sert à savoir si la Noxine est orientée sur la raquette du joueur 1 ou la raquette du joueur 2.



noxine1D



noxine1G



noxine2D



noxine2G

Si elle est orientée sur le joueur 1, on fait un second test pour savoir si la Raquette du joueur 1 est plus haute ou plus basse en ordonnée que la Noxine, selon le cas, on ajuste l'accélération de la Noxine.

Le 3e test consiste à regarder si la vitesse de la Noxine est supérieur à sa vitesse maximum, si oui, on réduit la vitesse pour qu'elle soit égal à la vitesse maximum.

Les 2 dernières lignes servent à modifier les vecteurs selon leur vecteur dérivée.  
(La dérivée de la position est la vitesse, et sa dérivée est l'accélération)

Ensuite, la fonction “*frameUpdate(self)*” permet d'augmenter le “*framecount*” et donc d'animer les Noxines.

```
def frameUpdate(self):
    self.framecountBis += 1
    if self.framecountBis >= 5:
        self.framecount += 1
        self.framecountBis = 0
    if self.framecount >= 3:
        self.framecount = 1
```

On incrémente “*framecountBis*” 60 fois par secondes, et quand celui-ci est égal à 5, on le relance et on incrémente “*framecount*”, cela permet de faire défiler  $60/5 = 12$  images en une seconde au lieu de 60 images si “*framecountBis*” n'était pas là.  
La dernière fonction de la Noxine, “*explose(self)*” lui

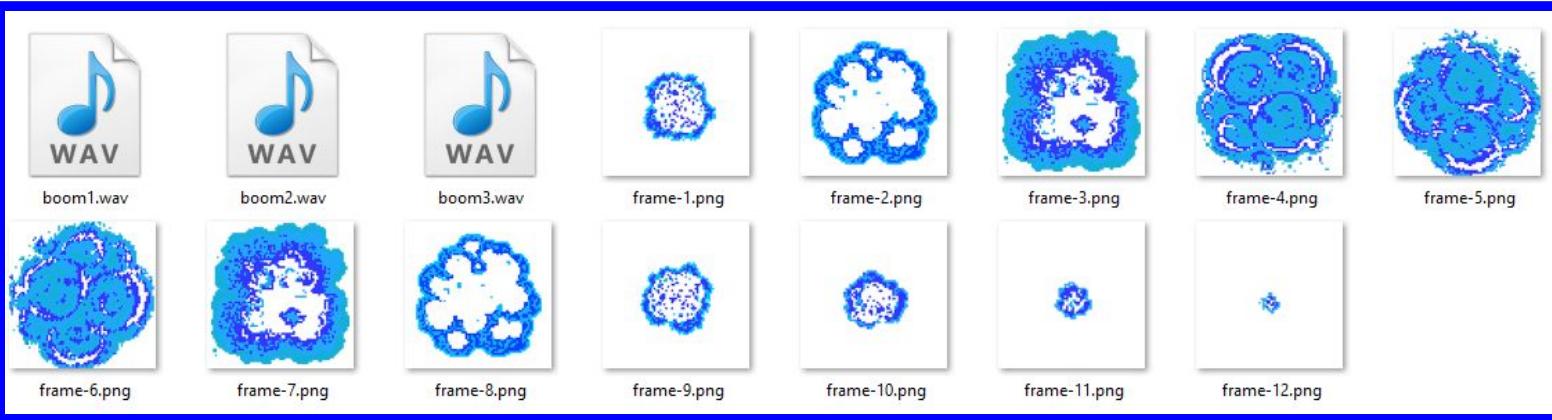
```
def explose(self):
    #---- boom
    if self.explosionFramecount == 1 and optionSon == True:
        nombreRandom = int(random(1,4)) # pour avoir un nombre entre 1 et 3
        if nombreRandom == 1:
            sonBoom1.play()
            sonBoom1.rewind()
        elif nombreRandom == 2:
            sonBoom2.play()
            sonBoom2.rewind()
        elif nombreRandom == 3:
            sonBoom3.play()
            sonBoom3.rewind()
        else:
            pass
    #---- framecount de l'animation
    self.explosionFramecountBis += 1
    if self.explosionFramecountBis >= 7:
        self.explosionFramecount += 1
        self.explosionFramecountBis = 0
    self.alive = False
    #--- explosion quand la noxine arrive à gauche
    """if self.vecteurPosition.y < P1.positionY + P1.hauteur+10 and self.vecteurPosition.y > P1.positionY - P1.hauteur-10:
        self.explosion = True"""
    #---destruction Plaque
    if self.explosionFramecount >= 3 and self.explosionFramecount <= 8:
        #calcul de norme : sqrt((Xb-Xa)**2+(Yb-Ya)**2)   ____(avec la plaque pour B)
        if sqrt((P1.positionX-self.vecteurPosition.x)**2+(P1.positionY-self.vecteurPosition.y)**2) < ((85 + P1.hauteur/2) or (85- P1.hauteur/2)):
            P1.hp -= 0.2
        if menuVar == "pongMulti":
            if sqrt((P2.positionX-self.vecteurPosition.x)**2+(P2.positionY-self.vecteurPosition.y)**2) < ((85 + P2.hauteur/2) or (85- P2.hauteur/2)):
                P2.hp -= 0.2
    if self.explosionFramecount >= 12:
        self.explosion = False
    image((imgExplosionBleu[self.explosionFramecount]),self.vecteurPosition.x-90 ,self.vecteurPosition.y-90, 180,180)
```

permet d'exploser au bon moment, d'afficher l'explosion et de faire des dégâts à la raquette des joueurs.

Les premières lignes servent à jouer un son d'explosion, j'ai essayé de faire jouer un son au hasard parmis 3 différents, mais je ne sais pas pourquoi, cela ne marche pas.

Il y a aussi un *framecount* de l'explosion qui fonctionne de la même manière que celui de la noxine, à la différence que l'explosion comporte 14 images.  
Le “*self.alive = False*” permet de “détruire” la Noxine, elle n'effectue plus ses autres fonctions, donc elle ne bouge plus et ne s'affiche plus.

Ensuite, on effectue un test pour savoir si le compteur d'image est entre 3 et 8 (le moment où l'explosion est la plus grosse)



Puis on effectue un calcul de norme pour savoir si la raquette est assez proche de la raquette pour lui infliger des dégâts. En multijoueur, on effectue la même chose pour la raquette.

Enfin, on affiche l'explosion, quand celle-ci est terminée, on modifie la variable "self.explosion" pour la rendre fausse et ainsi faire complètement disparaître la Noxine. Voilà comment se finit la triste vie d'une Noxine.



Mais lors de leur invocation, comment se fait-il qu'elles aient toutes une vitesse et une accélération si différente ? (ce n'est pas vraiment visible sur la capture)

```
----- Les Noxines ,les machines de Nox -----  
class Noxine:  
    def __init__(self,vecteurVitesse = PVector(15,0) ,vecteurAcceleration = PVector(-0.3,0) , focus = "left"):
```

Dans la création d'une Noxine, vous avez pu remarquer cette ligne. Celle-ci permet d'attribuer des paramètres à la Noxine. On peut voir ici que les 3 paramètres modifiables directement lors de la création sont: la vitesse, l'accélération et si elle attaque le joueur de gauche ou de droite.

Le “=” après le nom du paramètre permet d’attribuer une valeur par défaut si rien n’est spécifié à la création.

Lorsque Nox utilise son sort fétiche et envoie une nuée de ces bestioles, on donne un vecteur vitesse initiale très différent pour chaque Noxine, le vecteur accélération est aussi modifié et on rajoute le ciblage du joueur droit lorsque le jeu est en multijoueur.

```
listeNoxines.append(Noxine(PVector(16,6),PVector(-0.3,0)))
listeNoxines.append(Noxine(PVector(8,-8),PVector(-0.2,0)))
listeNoxines.append(Noxine(PVector(3,0),PVector(-0.1,0)))
listeNoxines.append(Noxine(PVector(25,-5),PVector(-0.3,0)))
listeNoxines.append(Noxine(PVector(29,5),PVector(-0.3,0)))
if menuVar == "pongMulti":
    listeNoxines.append(Noxine(PVector(-11,0),PVector(0.3,0),"right"))
    listeNoxines.append(Noxine(PVector(-16,6),PVector(0.3,0),"right"))
    listeNoxines.append(Noxine(PVector(-8,-8),PVector(0.2,0),"right"))
    listeNoxines.append(Noxine(PVector(-3,0),PVector(0.1,0),"right"))
    listeNoxines.append(Noxine(PVector(-25,-5),PVector(0.3,0),"right"))
```

## **L'impact de ce projet sur mon avenir:**

Ce projet m'a d'abord permis de bien comprendre ce qu'est la programmation, nous avons passé de très nombreuses heures à apprendre de nouvelles choses toujours plus difficiles pour les adapter selon nos envies. Le boss Nox (que je n'ai pas totalement expliqué en raison de sa grande composition) et le stockage JSON représentent en quelque sorte "l'apogée" de ce que j'ai appris.

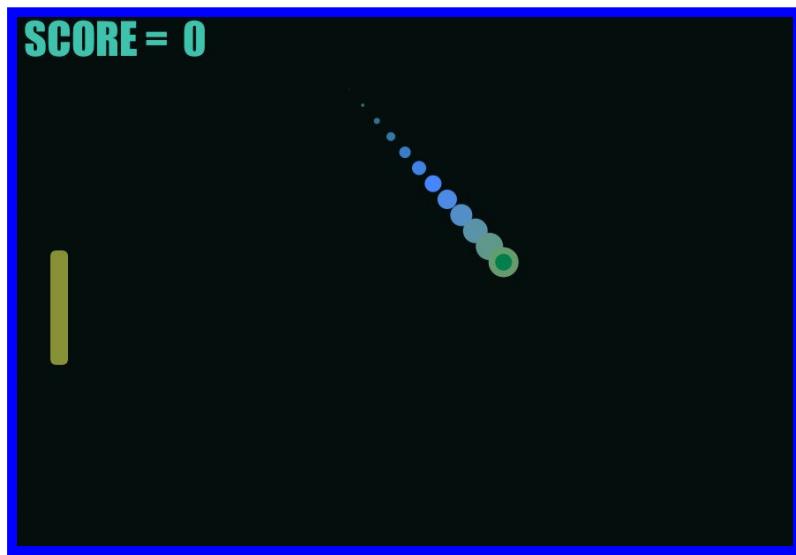
Faire ce Projet m'a montré que la programmation est un univers très agréable où presque tout est réalisable quand on en a la motivation. J'ai aussi compris que j'aime le fait d'avoir quelque chose de très visuel et de très direct quand je travaille sur quelque chose, contrairement aux cours théoriques par exemple.

J'ai aussi réalisé en travaillant avec Emeric et Luka que le travail en collaboration peut avoir de nombreux avantages, nous pouvions partager des idées, donner notre avis, s'entre-aider et avoir une plus large plage de connaissances (par exemple Emeric m'a aidé pour les musiques et le module minim et j'ai l'ai plutôt aidé pour les images et les animations)

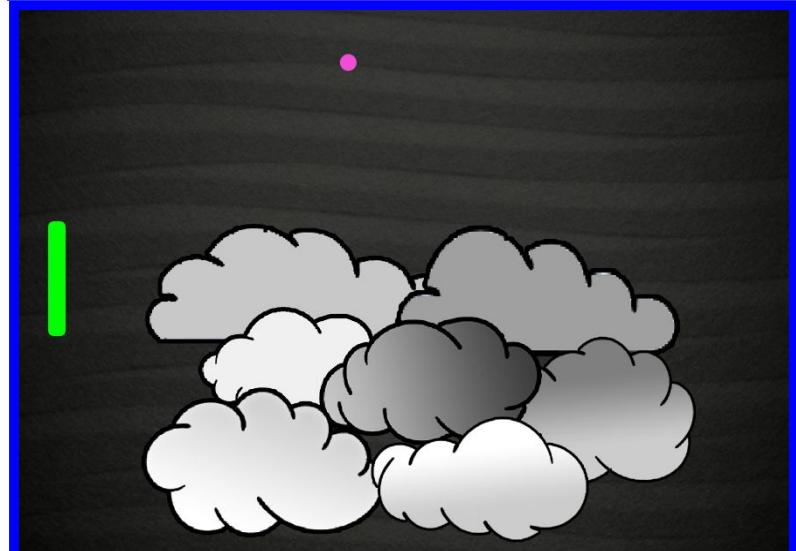
Cela a renforcé mon envie de réaliser des études en informatique (Pas seulement le projet mais aussi le reste des cours).

## Annexes:

Quelques captures d'écran de choses dont nous n'avons pas parler, ainsi que le code correspondant au trou noir que j'expliquerai lors de l'oral.



Le mode “disco” ou “arc-en-ciel” que Emeric et moi avons réalisé.

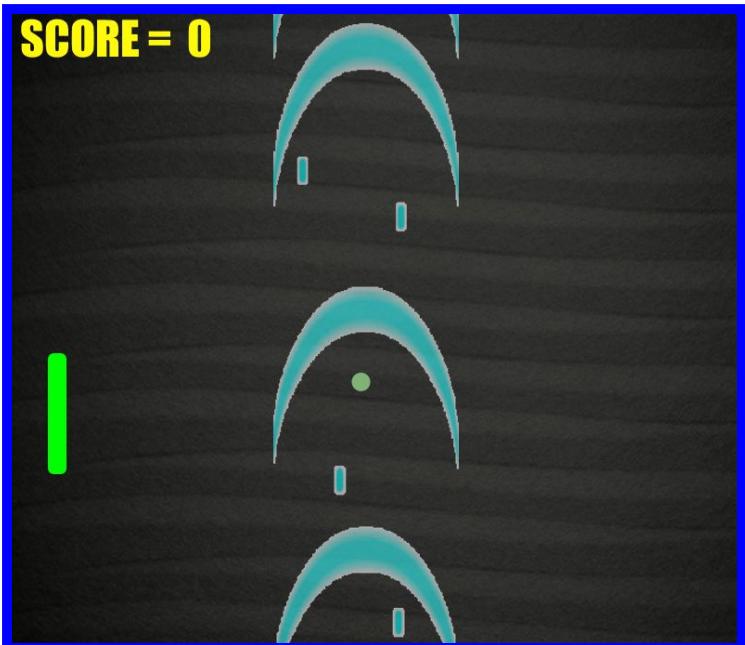


Le mode “nuage” que Luka a réalisé seul.

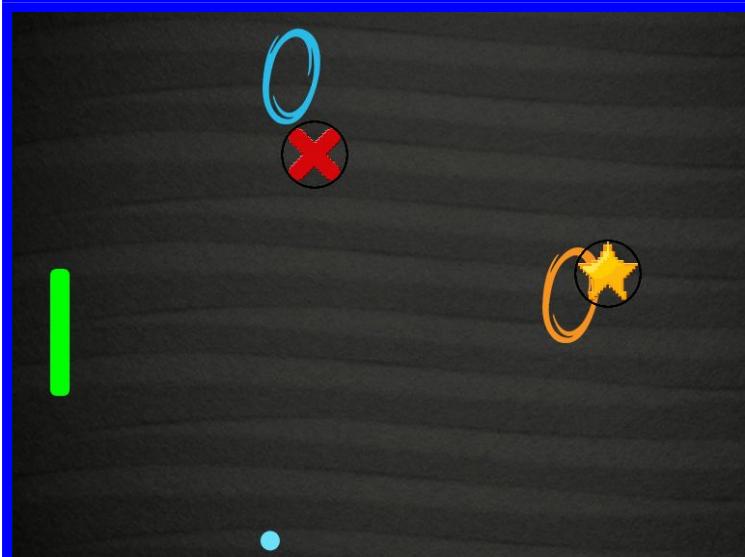


Le menu de fin de jeu que Luka a réalisé en grande partie.

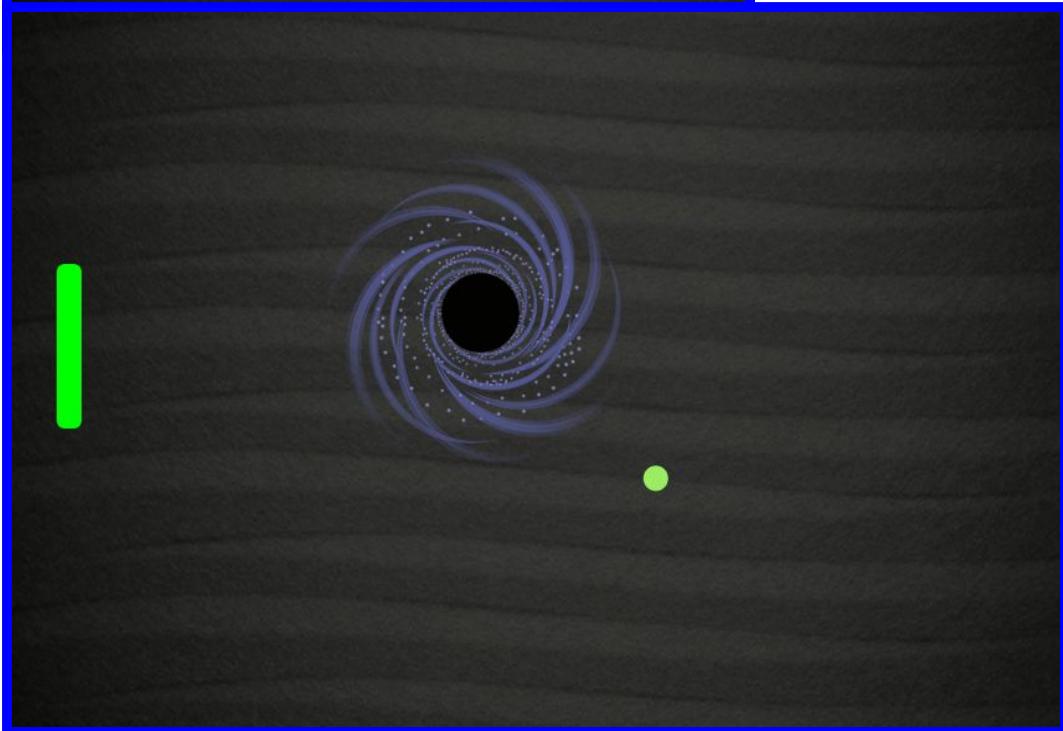
**SCORE = 0**



Le mode “champs de force” que j’ai réalisé.



Le mode “portails” que j’ai réalisé avec le mode “Bonus/Malus” que Luka a réalisé.



Le mode “trou noir” que j’ai réalisé et que je vais détailler pendant l’oral.

```

----- Calcul de la gravitation du trou noir -----
if optionBlackHole == True:
    global vecteurG, B1
    B1.display()
    for Balle in listeBalles:
        if Balle.alive == True:
            G = 6.67E-11 #constante gravitationnelle
            #distance entre 2 points = sqrt((xb-xa)2+(yb-ya)2) avec b pour Balle
            distance = sqrt((Balle.vecteurPosition.x - B1.vecteurPosition.x)**2 + (Balle.vecteurPosition.y- B1.vecteurPosition.y)**2)
            g = (G * 2.7E-3 * B1.masse) * 1/(distance**2)
            vecteurG = PVector(0,0)
            vecteurG.sub(Balle.vecteurPosition)
            vecteurG.add(B1.vecteurPosition)
            vecteurG.setMag(1) #pour une gravitation plus réaliste mais moins amusante
            vecteurG.mult(g)
            Balle.vecteurAcceleration.add(vecteurG)

```

Les captures relatives au trou noir que je réutilisera et expliquerai en détail lors de l'oral.

```

class BlackHole:
    def __init__(self):
        self.masse = 7E15 #(en kg) 7E15 :une valeur qui marche et ne bug pas
        #7E15 = 10 x la masse totale de carbone dans l'atmosphère source:wikipedia
        self.taille = 250
        self.vecteurPosition = PVector(random(100+self.taille,width-self.taille-150),random(self.taille, height-self.taille))
        self.angle = 0
        self.rotation = "left"
    def display(self):
        if self.rotation == "left":
            self.angle += 1
        if self.rotation == "right":
            self.angle -= 1
        if self.angle >= 360:
            self.angle = 0

        with pushMatrix():
            translate(self.vecteurPosition.x, self.vecteurPosition.y)
            rotate(radians(self.angle))
            image(imgBlackHole,-self.taille/2 , -self.taille/2, self.taille, self.taille)
    def reset(self):
        rand = int(random(0,2))
        if rand == 1:
            self.rotation = "right"
        else:

```