# La caverne aux Jeux

Bonjour, je m'appelle Gwendal Auphan, et avec mon camarade Dorian Gaspar, nous avons créé la « Caverne aux Jeux ».

Le principe de l'application est de réunir plusieurs mini-jeux, incluant un aspect compétitif.

Ils sont au nombre de huit au moment de la rédaction de ce dossier. Il est prévu néanmoins d'en ajouter d'autres, voire d'inclure les projets des autres groupes d'ISN.

Afin de ne pas avoir un dossier trop important et de faciliter la lecture de celui-ci, je ne détaillerai qu'un seul jeu parmi les huit présents dans l'application : Flappy Bird. J'ai choisi ce jeu car il est très complet et en outre il se démarque des autres par son aspect dynamique.

Ainsi, à travers ce dossier, je vais vous présenter une première partie qui explique l'aspect global de l'application et une seconde consacrée au jeu « Flappy Bird » dans laquelle j'expliquerai en détail l'algorithme du jeu. Au total, je présenterai quatre jeux du projet, les quatre autres seront présentés par Dorian (se référer à son dossier).

Je précise que la partie réseau a été réalisée en totalité par Dorian. Pour ma part, je me suis consacré à l'aspect graphique.

L'application est toujours en cours de développement pour des raisons de finition, cependant tous les jeux présentés sont fonctionnels et disposent d'un algorithme de score.

Sur ce, je vous souhaite une agréable lecture.

# Table des Matières

# ISN Projet Final

## 1. Projet principal

### I) Choix du projet

Lors de la mise en place des groupes, il me paraissait évident de me mettre avec Dorian Gaspar puisque nous faisons du codage ensemble.

Au début, nous avons commencé par chercher l'idée du projet, nous pensions faire un « puissance 4 » contre une intelligence artificielle, cependant les contraintes fixées par le professeur (recours aux librairies installées de base avec Python) nous empêchaient de le faire, j'ai donc eu l'idée de créer une application regroupant plusieurs mini-jeux. Au fur et à mesure, nous avons rajouté les fonctionnalités de la partie « réseau » qui n'étaient pas prévues au départ. Finalement nous avons réussi à créer une interface permettant de créer autant de jeux que nous le voudrons. Cette idée m'a été inspirée par le site « Jeu.fr » sur lequel j'allais souvent étant plus jeune.

Nous avons donc décidé de créer une application contenant des mini-jeux, le tout étant connecté en réseau afin de comparer les scores de chacun et de faire un classement.

### II) Objectifs

Les objectifs que nous nous étions donnés étaient, dans un premier temps, une certaine efficacité au niveau du code, nous devions en effet réaliser le projet dans un délai restreint dû aux concours. Ensuite il y avait la réalisation des différentes interfaces graphiques car il fallait aussi que l'application soit agréable, de plus l'utilisation des classes était aussi un objectif fixé en vue de l'optimisation du code. Enfin nous devions nous essayer à la partie réseau qui était nouvelle pour ma part, un peu moins pour Dorian.

### III) Répartition des tâches

Concernant la répartition des tâches, nous avons commencé à travailler sur le même jeu qui était « Tête chercheuse ». Nous ne savions pas vraiment par quoi commencer, puis au fil des jours, nous avons réussi à instaurer une certaine routine de codage avec de nouveaux objectifs chaque jour sur le programme. Il y avait des jours où nous codions sur le même jeu et d'autres où nous étions chacun sur notre tâche personnelle mais il y a toujours eu une entraide et une confiance entre nous importantes afin de s'aider mutuellement et au besoin de reprendre le dossier de l'autre.

## IV) Environnement

Le travail s'est fait sur nos ordinateurs personnels, et le projet a été réalisé essentiellement à la maison car une durée de 2 heures par semaine ne suffisait pas à nos objectifs.

Pour pouvoir travailler ensemble, nous étions 90% du temps en conversation vocale à l'aide de « Discord » (application similaire à « Skype »). Ensuite pour le codage, nous avons utilisé « Atom » et de temps en temps « Visual Studio Code ». Ces applications sont très intuitives et permettent une meilleure rapidité, de plus cela nous permettait de faire des « télétypes » (fonctionnalité permettant d'écrire sur un même fichier qu'une autre personne et en même temps). Enfin nous avons créé un dossier « GitHub » nous permettant une progression plus rapide et servant à suivre l'avancement du projet, enfin grâce à cela nous pouvons avoir accès au code source depuis n'importe où, seulement avec Internet et l'adresse : https://github.com/dogasp/ISN_Projet_Final.

## V) Organisation du projet

Pour l'organisation du code, nous avons décidé de créer un dossier pour chaque jeu et de mettre les ressources correspondantes dans ces derniers, nous avons aussi créé d'autres dossiers pour des fonctionnalités comme le menu principal ou encore le « Scoreboard ». Bien sur les fichiers des différents dossiers sont reliés entre eux par des « imports » au début de code si nécessaire. Cette organisation des dossiers nous permet d'avoir une meilleure répartition des tâches et une meilleure clarté dans le projet afin de pouvoir modifier des parties précises du code sans avoir à chercher trop longtemps.

Pour une meilleure organisation pendant la réalisation du projet, nous avons décidé de faire un journal de bord pour répertorier tout ce que l'on faisait pour le projet dans la journée. Ce document s'est avéré très utile à la réalisation de ce dossier et à la visualisation de l'avancement de chaque module.

# VI) Application principale

## A) Le réseau

La partie serveur du projet a été faite en totalité par Dorian puisqu'il dispose d'un **Raspberry PI** chez lui. Je me suis malgré tout intéressé à ce que Dorian avait fait et j'ai bien compris le code. Je vais donc reprendre ce qu'il a rédigé pour expliquer cette partie du projet. Le code se partage en deux parties : la partie client et la partie serveur. Pour faire les requêtes réseau, on a utilisé la librairie « socket » qui est une librairie installée par défaut avec Python et qui permet de faire du codage réseau en Python.

La partie client se compose de quatre fonctions servant à :

- Envoyer le score d'un joueur à la fin d'une session sur un jeu
- Récupérer le scoreboard des 10 meilleurs joueurs tout jeux confondus
- Récupérer le scoreboard des 10 meilleurs joueurs sur un jeu en particulier
- Récupérer les meilleurs scores d'un joueur en particulier

Dans chaque fonction, on établit une connexion vers l'IP 90.91.3.228, qui correspond à celle de chez Dorian, et on envoie un message contenant le type de valeur qu'on veut récupérer. Une fois la réponse reçue, on retourne le résultat.

La partie « client » sert aussi à détecter le fait que la connexion puisse être insuffisante afin d'effectuer malgré tous les transferts de données. Pour cela, on a défini le Timeout (durée d'attente d'une réponse) à 100 ms. Si ce temps est dépassé, une valeur par défaut est retournée afin d'éviter un temps de latence trop important.

La partie serveur est enregistrée sur un **Raspberry PI** installée chez Dorian.


Photo du **Raspberry PI**

Le Raspberry PI Il est allumé en permanence et il est à l'écoute des demandes de connexion et de données d'un éventuel client. Le serveur accepte toutes les requêtes entrantes et traite toutes les demandes une à une. On a un délai moyen de 10 millisecondes et un délai

maximum de 60 millisecondes entre l'appel de la fonction du client et la réponse du serveur, ce qui est très faible. Le serveur n'aura donc pas de mal à supporter une cinquantaine de personnes qui envoient des requêtes simultanément or les requêtes serveur ne sont pas effectuées en permanence, l'application pourrait donc supporter plusieurs centaines personnes en même temps.

Au niveau du traitement des données, le fonctionnement du serveur est similaire à une console système : suivant le message envoyé au serveur, il y a différentes actions qui sont faites. Par exemple pour ajouter le score d'un joueur à la fin d'une session, le code suivant est utilisé :

```
if command == "add":     #si la commande est add, on ajoute le score au joueur

    print("player {} scored {} in {}".format(list[1], list[3], list[2]))
    try:
        if players[list[1]][list[2]] < float(list[3]):
            players[list[1]][list[2]] = float(list[3])
    except:
        players[list[1]][list[2]] = float(list[3])
    save()
    return b"ok" #le retour n'est pas important
```

« Command » correspond au message envoyé par le client au serveur, il est accompagné d'arguments stockés dans la liste *list*. Le try est utilisé pour tenter de comparer la valeur de l'ancien meilleur score du joueur. Les données sont stockées dans un dictionnaire contenant les pseudos des joueurs et un autre dictionnaire contenant les meilleurs scores du joueur. Si le try ne renseigne pas d'erreur, on donne la valeur du score envoyé à la place de l'ancien si le nouveau est supérieur. Si le try aboutit à une erreur, c'est que le joueur n'a pas de meilleur score pour le jeu voulu, on l'initialise donc avec la valeur envoyée.

## B) La page principale

La page principale est l'endroit où on peut lancer les jeux. On peut comparer cette fenêtre au site « jeu.fr » qui est un site de mini-jeux en ligne. Le but premier de cette interface est de proposer des boutons afin de permettre de lancer les différents jeux et d'afficher le scoreboard principal.

Pour ce faire, nous avons créé une classe pour gérer l'exécution d'un jeu. Cette classe agit comme un bouton qui est en fait une image sur un Canvas, lui-même pouvant être défilé afin de permettre d'ajouter plus de jeux dans le futur.

Lorsque l'utilisateur clique sur une image, la fonction « command » est exécutée. Cette fonction permet de lancer le jeu sélectionné et de cacher la fenêtre principale. Une fois la session finie, le score réalisé par l'utilisateur est envoyé au serveur via la fonction « push

score » se trouvant dans le module réseau du projet, détaillé dans la partie précédente. Une fois que le score est envoyé, l'affichage du scoreboard est actualisé.

Justement, une autre fonctionnalité de cette page principale est l'affichage d'un scoreboard général, c'est à dire le classement des meilleurs joueurs par la somme des meilleurs scores dans chaque jeu.

Afin d'identifier le joueur, à chaque lancement de l'application, un pseudo est demandé à l'utilisateur. Ce pseudo est utilisé dans les fonctions clients et serveur afin d'identifier les joueurs et afficher le classement. Il y a aussi un bouton aide qui permet d'obtenir quelques renseignements sur le fonctionnement de l'application.

### C) Les jeux

## Structure :

Les jeux sont structurés de la même façon :

- Une fonction d'initialisation de la classe correspondant au jeu avec renvois du meilleur score de la session ;

- Une classe qui se charge de l'affichage et du fonctionnement du jeu.

À l'intérieur de la classe, l'application est structurée de la façon suivante :

- une première fenêtre avec l'affichage des règles et des commandes spécifiques au jeu lancé. Ainsi que l'affichage du scoreboard spécifique avec les meilleurs joueurs du jeu.

- une seconde fenêtre (optionnel) pour demander à l'utilisateur la difficulté de la partie.

- une fenêtre principale avec l'interface du jeu en lui-même.


Scoreboard est commun à tous les jeux.

## Présentation des jeux :

Sommaire des jeux :

- **Tête chercheuse**
- **Tom & Jerry**
- **Pendu**

# Tête chercheuse

**Principe du jeu :**

Le principe du jeu est que le robot atteigne le drapeau. A chaque fois que le robot rencontre un mur ou un obstacle il tourne à sa droite, ainsi avant le début de la partie, le joueur doit disposer des obstacles de manière à ce que le robot arrive jusqu'à la fin du parcours. Le score de la partie est calculé en fonction du temps, des déplacements et du nombre d'obstacles placés sur la grille. Le robot peut aussi passer sur des pièces qui rapporteront plus de points, au joueur d'avoir la bonne stratégie afin d'avoir le meilleur score. Pour l'instant il y a 5 niveaux différents et le score s'ajoute à chaque niveau réussi. Il s'agit d'un bon jeu de réflexion et de visualisation de l'espace.

Pour placer les obstacles, il suffit de cliquer à l'endroit qu'on souhaite. On peut les enlever en recliquant dessus.

Un bouton « Restart » a été placé afin de recommencer la partie si l'on s'est trompé ; de même la partie recommencera si le robot passe 4 fois sur la même case. A chaque « Restart », on soustrait 50 points au score total du joueur.

**Réalisation :**

Nous avons commencé le projet dès le premier jour. Ce premier jeu n'a pas été très bien structuré, il faut savoir que nous avions commencé à le faire sans les classes et que finalement pour des raisons d'optimisation, nous avons décidé de le remodifier entièrement en ajoutant les classes. Vu qu'il s'agit du premier jeu, il a été fait un peu par tâtonnement, nous n'avions pas vraiment de répartition des tâches.

Je me suis tout de même occupé de la création de la carte, des différentes images comme le robot ou le drapeau, ensuite de la création des messages d'interactions et de la détection du click. Dorian, lui, s'est occupé des déplacements du robot et des fonctions du jeu servant au fonctionnement.

**Fonctionnement :**

Après avoir mis les imports*, l'initialisation des variables, fait le chargement des règles, on lance la fonction « update » qui nous permet de créer la carte et l'interface, à chaque click, cette fonction est appelée pour mettre à jour les différents obstacles mis en place. Ici il s'agit de la partie avant le lancement du « START ».

Maintenant après avoir appuyé sur le bouton « START », on lance la fonction « start » qui nous permet de déplacer le robot toutes les 400 ms et effectue en même temps les vérifications de mur, de pièces, d'arrivée et de mort (si le robot passe 4 fois au même endroit).

La fonction « command_user » répertorie toutes les fonctions utiles lorsque la partie est finie. On appelle la fonction « command_user » avec un argument et en fonction de ce dernier, on lance la fonction correspondante. Cette méthode sert à regrouper les fonctions dont on avait besoin en fin de partie.

Si le joueur appuie sur « Restart » ou si le robot meurt alors la partie se relance avec l'initialisation des variables du début de partie, si il gagne, un menu s'affiche avec le choix entre : recommencer, passer au prochain niveau, ou retourner au menu principal.

# Tom & Jerry

**Principe du jeu:**

*Tom & Jerry* est un jeu très intéressant avec une réflexion importante. Le but du jeu est que Jerry (la souris) atteigne le fromage sans qu'il se fasse toucher par Tom (le chat). A chaque déplacement Tom se rapproche de Jerry le plus possible. Mais le jeu est plus compliqué que cela car Tom le chat peut se déplacer en diagonale et ainsi se rapprocher plus de Jerry. C'est là où tout est l'intérêt du jeu, la stratégie est donc de trouver le bon chemin afin de ne pas se faire rattraper par Tom. Le score du jeu est calculé en fonction du temps mis par le joueur pour faire le niveau et en fonction du nombre de déplacements de Jerry. Si le joueur se fait attrapé par Tom, il perd 50 points multiplié par le niveau. Il s'agit ici d'un bon jeu de réflexion et de visualisation de l'espace.

**Réalisation :**

*Tom & Jerry* comme *Flappy Bird* a été fait par moi-même, Dorian m'a tout de même aidé lorsque j'avais besoin de lui. Le fonctionnement de la création de la carte s'inspire énormément du premier jeu « Tête chercheuse » ce qui m'a permis d'aller plus vite sur cette partie. Ensuite une partie importante a été dédiée à l'algorithme pour déplacer Tom (le chat) car il peut y avoir 2 chats dans la même carte. Une partie assez longue a aussi été dédiée aux images (redimensionnement en fonction de la partie). Ainsi c'est un jeu assez court mais qui a tout de même nécessité une réflexion importante sur les déplacements des éléments du jeu.

**Fonctionnement :**

Après avoir mis les différents imports et organisé le design de la page, on lance la fonction « start» servant à créer l'affichage du jeu et initier les différentes variables (elle est appelée à chaque démarrage ou recommencement de la partie). Ensuite la fonction « move_Jerry » est appelée lorsqu'on appuie sur une des touches des flèches du clavier. Ensuite si Jerry peut se déplacer, on appelle la fonction « move_Tom » qui sert à déplacer Tom (le chat) vers Jerry. (C'est ici que le l'algorithme a été le plus complexe. Enfin à chaque déplacement de Tom, la fonction « verif » est appelée pour vérifier si Tom n'a pas rattrapé Jerry et ainsi la partie serait finie. Dans cette même fonction, on vérifie aussi si Jerry est arrivé jusqu'au fromage. Si oui, on lance un menu avec 3 choix (recommencer, prochain niveau, menu principal). Le jeu est pour l'instant composé de 5 niveaux.

# Pendu

**Principe :**

Le *Pendu* est un jeu consistant à trouver un mot en devinant quelles sont les lettres qui le composent. Le jeu se joue traditionnellement à deux, avec un papier et un crayon, selon un déroulement bien particulier. Ici le jeu est un peu différent puisqu'il ne se joue pas à deux mais tout seul contre l'ordinateur. Ainsi le mot à deviner est tiré aléatoirement dans une base de données de mots destinés au jeu.

**Réalisation :**

Le jeu du pendu a été réalisé par moi-même en très peu de temps puisque son algorithme est vraiment très basique, cependant la partie interface graphique n'a pas encore été réalisée.

**Fonctionnement :**

Pour ce jeu, nous avons décidé de créer 3 niveaux différents afin d'avoir des scores différents et d'avoir plus de contenu. Ainsi en fonction du choix du joueur, le mot aura une longueur plus ou moins grande et un nombre plus ou moins important de voyelles. Après avoir choisi la difficulté, on lance la fonction « start » qui nous sert à choisir le mot aléatoirement, elle sert aussi à initialiser des variables et créer l'interface du jeu, elle est appelée à chaque début de partie ou recommencement. Ensuite la fonction « check » est appelée lorsque le joueur rentre une lettre dans l'espace de saisie. Cette fonction est la partie principale de l'algorithme, elle sert à déterminer si la lettre saisie est correcte, si elle a déjà été saisie etc… Enfin elle vérifie si le joueur a gagné ou non et rappelle la fonction « start » en fonction de la réponse du joueur.

# Détail du jeu « Flappy Bird » :

## Principe du Jeu de Flappy Bird :

Le jeu repose sur l'agilité du joueur, qui doit faire avancer un oiseau dans un environnement à défilement horizontal en tapotant sur l'écran tactile, tout en évitant des tuyaux présents en haut et en bas de l'écran. Les règles de jeu sont très simples : lorsque l'oiseau touche un tuyau ou heurte le sol, la partie est terminée. Le joueur reçoit un point pour chaque tuyau que l'oiseau évite. Le jeu est assez connu chez les jeunes puisqu'il a connu un moment de gloire entre 2013 et 2014. Le jeu a même été retiré des plateformes de jeu fin 2014 dû à son succès.

## Réalisation de Flappy Bird :

*Flappy Bird* est le dernier jeu à avoir été réalisé, je le trouve assez complet et plaisant. C'est moi qui l'ai développé à 95%, Dorian m'a aidé pour les idées et pour l'algorithme du jeu. Nous avions convenu de coder des jeux chacun de notre côté mais toujours avec un suivi sur ce que l'autre faisait. *Flappy Bird* est un jeu différent des autres dans le sens où le jeu donne une impression de vitesse et de mouvement mais je reviendrai sur ce point dans les difficultés abordées. Je me suis tout d'abord intéressé à la création puis aux déplacements des tuyaux et ceux de l'oiseau. Je me suis ensuite occupé des images du jeu, il s'agit d'une partie très longue faite à l'aide de Photoshop. Enfin la dernière partie s'est concentrée sur la réalisation du recommencement de la partie et de l'animation du début de partie.

## Fonctionnement de Flappy Bird :

Après avoir mis les différents imports et initié les variables du jeu, on lance la fonction « build_game » servant à créer l'affichage du jeu (elle est appelée à chaque démarrage ou recommencement de la partie). Ensuite le jeu est divisé en 2 temps, le premier temps où le joueur n'a pas encore appuyé pour faire bondir l'oiseau et le deuxième temps qui consiste à la partie du jeu en cours lorsque le joueur doit passer entre les tuyaux.

Le premier temps est constitué de 2 fonctions, « wait_game » qui sert à bouger l'oiseau de haut en bas jusqu'à temps que le joueur appuie, alors la deuxième fonction « move_bird_begin » est appelée qui sert à déplacer l'oiseau jusqu'à la zone où l'oiseau se déplacera par les bonds du joueur.

Arrivé à cette zone, on passe alors au deuxième temps du jeu. La fonction « start » est lancée dès le début et crée les objets dont on aura besoin. La fonction « update » est ensuite lancée et elle s'appelle elle-même toutes les 50 ms. Cette fonction sert à lancer d'autres fonctions servant aux déplacements des tuyaux et de l'oiseau et à la vérification de la mort de l'oiseau. Si l'oiseau meurt, une animation est lancée par la fonction « wait_dead », ensuite le joueur décide soit de relancer la partie ou de revenir au menu principal.

**Contraintes :**

L'intérêt de ce jeu est qu'il est assez dynamique en comparaison aux autres. Ainsi le défi majeur de ce jeu était d'obtenir un rendu assez fluide et réaliste puisque l'oiseau vole.

Cependant les performances du processeur étaient limitées étant donné que le but de notre projet principal était de rendre l'application accessible à tous, et donc à tous les types de processeurs, des plus performants aux moins performants.

Enfin le jeu devait comporter une interface graphique agréable avec quelques fonctionnalités en plus du jeu pour que ce dernier ne lasse pas les joueurs.

**Objectifs et explication du code :**

Pour ce jeu, nous avions une liste d'objectifs conséquente :

- Intégrer la mécanique de base du jeu (assez complexe)
- Différentes animations en début et fin de partie du jeu
- Mouvement de tête de l'oiseau
- Mise en place du décor et des images

Le code est composé de deux classes, une servant au jeu principal et l'autre servant à gérer les tuyaux. Ainsi on va créer 4 objets (tuyaux) que l'on pourra gérer indépendamment, cela permet d'améliorer l'algorithme et d'avoir une meilleure lisibilité de code car sans cette méthode, cela aurait été plus compliqué.

**Mini sommaire :**

1) **Création de l'interface graphique et animations**
2) **Déplacement et orientation de l'oiseau**
3) **Création et déplacement des tuyaux**
4) **Vérification de mort de l'oiseau**

# I) Création de l'interface graphique et animations:

## A) Interface graphique

Après avoir vu les différentes règles et le scoreboard lié au jeu, on importe les photos et on appelle la fonction **« build_game ».**

C'est la fonction qui sert à générer tout ce qui se trouve sur le jeu, elle est appelée à chaque début et chaque recommencement de partie. Cette partie du programme n'est pas vraiment différente de celles des autres jeux puisqu'elle reprend les mêmes syntaxes en termes de structure du code. Dans cette même fonction, on initialise les variables dont on aura besoin dans le jeu et il y en a beaucoup…

Ensuite on crée le salon d'attente où l'oiseau va être en attente que le joueur clique sur le jeu. Dans le même temps on choisit au hasard un décor pour l'image de fond (servant aussi au divertissement pour le joueur).

Enfin on appelle la fonction **« wait_game »** qui servira à l'animation de départ.

## B) Animations de début et fin de partie

**Début de partie :**

L'animation du début de partie est composée de deux fonctions, la première fonction **« wait_game »** sert à déplacer de haut en bas l'oiseau tant que le joueur n'a pas appuyer pour lancer la partie. On utilise la condition self.play = True ou False.

Pour que l'oiseau redescende au moment souhaité, on lui fixe une limite, par exemple ici dès que la vitesse dépasse 7, on le fait descendre. Le résonnement est le même lorsque l'oiseau monte.

Enfin la fonction s'exécute toutes les 75 ms permettant une certaine fluidité. Le « else : » signifie que le joueur a appuyé pour commencer la partie, on appelle ainsi la fonction **« self.move_bird_begin ».**

La fonction **« self.move_bird_begin** » sert à déplacer l'oiseau jusqu'à sa zone de déplacement durant le jeu. On enlève bien sûr l'affichage de départ avec les flèches et la main, l'oiseau effectue alors le même déplacement vertical, on rajoute juste un déplacement horizontal. Pendant cette période, le joueur n'a encore aucun contrôle sur les déplacements

de l'oiseau. De plus, pour des questions de réalisme on déplace le sol en même temps que l'oiseau.

Pour savoir quand l'oiseau arrive à la zone déterminée, on regarde si son abscisse plus son déplacement horizontal dépassent une certaine valeur, ici 100 pixels.

```
if xl + vitesse > 100:#Tant que l'abscisse de l'oiseau
```

Sinon on appelle la fonction **« start »** pour lancer le jeu.

On arrive ainsi à la fin de l'animation du début de partie, voici alors un exemple de l'oiseau attendant le clique du joueur.

**Fin de partie :**

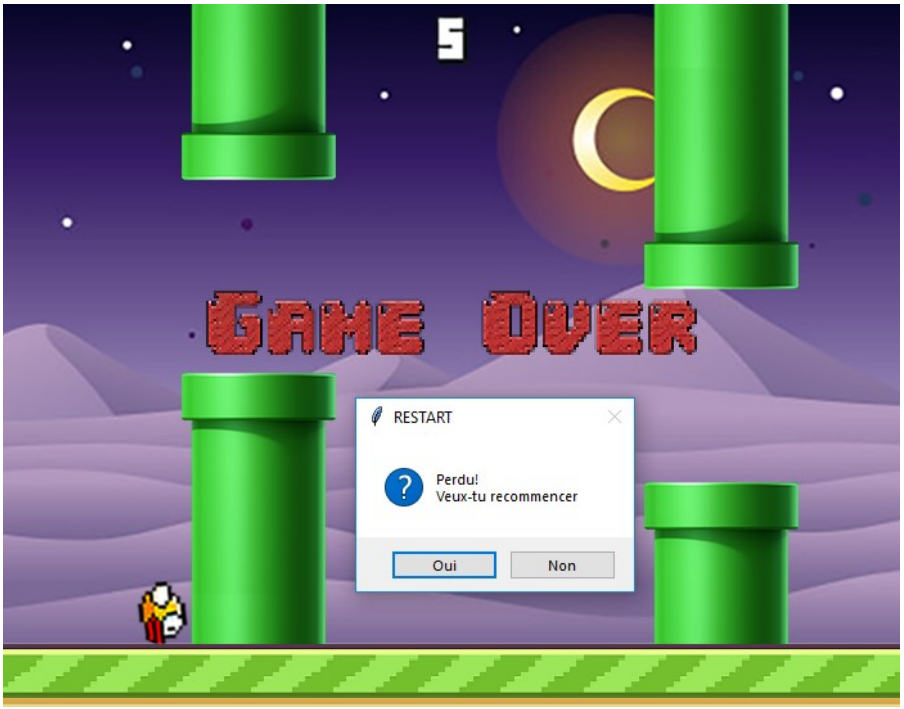
L'animation de fin est exécutée lorsque l'oiseau meurt, en effet il n'y a pas de victoire à proprement parler dans ce jeu. Ensuite il faut savoir que l'animation de fin de partie ne sera pas la même si l'oiseau meurt en chutant au sol ou s'il rencontre un tuyau.

Pour commencer, je vais prendre le cas où l'oiseau rencontre un tuyau, j'expliquerai plus tard comment la vérification se fait. On lance alors la fonction **« wait_dead ».** Cette fonction sert à faire chuter l'oiseau verticalement seulement lorsque ce dernier rentre en contact avec le tuyau. C'est pourquoi elle n'est pas appelée lorsqu'il touche le sol. Dès que l'oiseau touche le sol, la fonction s'arrête sinon elle tourne en boucle.

Dans le même temps, dès que la mort est vérifiée dans la fonction **« verif_bird »,** on affiche un **« Game Over »** avec une animation assez stylisée. L'image apparait puis grossit au fur et à mesure, puis au bout de 2 secondes, la fenêtre restart se lance.

La fonction **« ending »** ligne 343 sert à agrandir petit à petit l'image du « Game Over ».

Exemple de fin de partie avec mort contre un tuyau




Dans le second cas où l'oiseau ne meurt pas d'un contact avec le tuyau mais d'une chute au sol, la fonction **« wait_dead »** n'est pas appelée, il y a seulement l'affichage du « Game Over » et la question du relancement qui s'affiche.

## II) Déplacement de l'oiseau

Les déplacements de l'oiseau sont gérés par la fonction « bird_move », cette fonction est appelée grâce à la fonction « update » qui tourne en boucle toutes les 50 ms, celle-ci précédemment appelée dès le lancement de la partie.

La fonction « bird_move » sert en même temps à gérer l'inclinaison de l'oiseau et à détecter si l'oiseau a touché le sol. La fonction est donc décomposée en 2 parties, une partie où l'oiseau monte et une autre où l'oiseau descend. La distinction des 2 parties se fait grâce à la variable « self.press ».

Tout d'abord, on place toute la fonction dans une boucle qui se répète 6 fois afin de faire plus de calculs et d'être plus précis. L'oiseau va donc commencer à descendre puisque la dernière valeur de « self.press = False ».

```
for _ in range(6): #Début de la boucle pour plus de calculs et de précision
```

Pour gérer l'inclinaison de l'oiseau, on a créé 25 images inclinées du plus haut au plus bas. Cette manière nous sert à simuler une certaine réalité, quand l'oiseau monte, on affiche très rapidement différentes images inclinées du bas vers le haut et inversement quand l'oiseau va vers le bas.

**Pour la descente**, self.press = False. On sépare tout d'abord l'algorithme en deux parties, la 1ère où self.count_image < 25 et l'autre où self.count_image > 25. On sépare de cette manière afin de ne pas augmenter l'index de l'image lorsqu'on est à la dernière image pour éviter d'avoir l'erreur (IndexError: list index out of range). Pour simuler une chute de l'oiseau, on multiplie par l'index par 1,04 et après avoir dépassé l'index 5 par 1,1. Ceci est fait de telle manière à accélérer la chute pour rendre le jeu réaliste.

```
if self.count_image < 25:        #Si self.count_image < 27, pour éviter index out of range
    if self.count_image > 5:     #Si self.count_image > 5:
        self.count_image *=1.1   #On multiplie self.count_image par 1.1
    else:                        #Sinon
        self.count_image *=1.04 #On multiplie self.count_image par 1.4 pour faire accéler l'inclinaison
    self.vitesse +=0.06         #La vitesse est toujours augmentée
```

Après avoir géré l'inclinaison on passe au déplacement de l'oiseau, on commence alors par incrémenter la vitesse de +0.055 à chaque boucle.

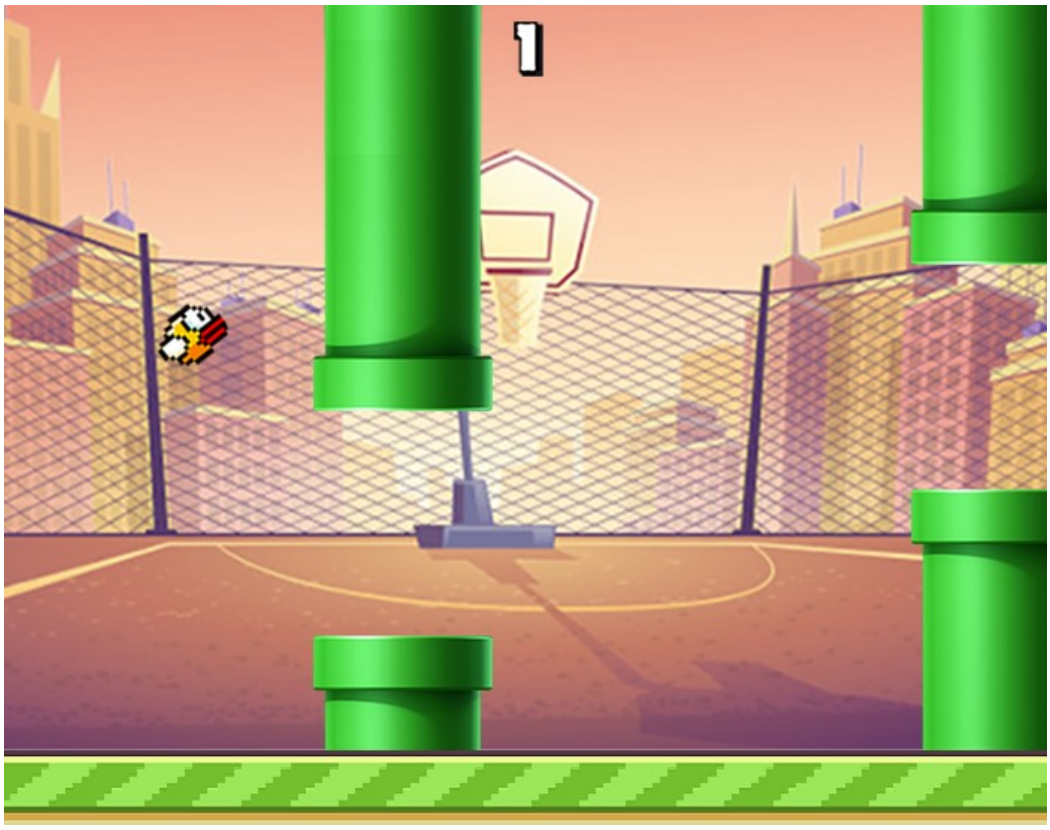
En même temps que déplacer l'oiseau, on vérifie aussi si l'oiseau ne touche pas le sol avec la condition : **if (self.y_center_bird +self.vitesse + 25) > 500**, ainsi si cette condition est vérifiée, on place l'oiseau à y = 475 afin d'être sûr qu'il touche bien sans dépasser le sol, on appelle la fonction **« verif_bird »** qui nous sert à afficher le Game Over et self.verite = False afin d'arrêter la boucle de la fonction update.

**Pour la montée**, self.press = True grâce au click du joueur, l'oiseau va donc monter de 2.9 pixels multiplié par 6 toutes les 50 ms. Pour éviter que l'oiseau ne s'envole vers l'infini, on pose i = 0 et à chaque tour on l'incrémente de 1, ainsi au bout de 30 calculs la fonction s'arrête et l'oiseau aura monté de 87 pixels en 250 ms.

En ce qui concerne l'inclinaison de l'oiseau, on connait la dernière valeur de l'index de l'image de la descente avec **« self.count_image »,** on va lui soustraire à chaque boucle -1,2. La liste_image contient 25 images de l'oiseau ce qui permet d'avoir une bonne précision sur l'inclinaison de l'oiseau. A la fin, quand **i == 30**, self.press = False et l'oiseau va alors descendre.

```
if self.i >= 30:    #Grace à ctte condition, on effectue 30 calculs et on recommence
    self.press = False
    self.count_image = 0.5 #On réinitialise self.count_image = 0.5 pour la descente
else:                      ################################
    if self.count_image > 0:    # Sinon:
        self.count_image -= 1.2 # On baisse self.count_image à chaque boucle
    else:                       # jusqu'à ce self.count_image = 0
        self.count_image = 0    ################################
```

Exemple de vol de l'oiseau, nous pouvons maintenant passer au déplacement des tuyaux.



## III) Création et déplacement des tuyaux :

Pour gérer les tuyaux, j'ai décidé de créer une classe qui serait appropriée à ça. J'ai fait ce choix car je voulais absolument créer plusieurs tuyaux qui se voient sur l'écran, je n'ai trouvé que cette solution, bien sûr il en existe sûrement d'autres. Le principe global est de 4 objets (tuyaux) et de les gérer indépendamment. Ainsi je les crée en même temps à des endroits différents et en fonction des conditions vérifiées, ils seront régénérés au début ou encore d'autres fonctions seront appelées.

### Création des tuyaux

La création des tuyaux se fait au lancement de la fonction **« start »** appelée au moment où le joueur clique pour commencer la partie. On crée d'abord les 4 objets puis on lance pour les 4 la fonction **« create_pipe »** servant à créer le tuyau. Les arguments envoyés servent à l'endroit où l'on veut les placer et la hauteur du milieu des tuyaux.

```
def start(self):    #Fonction servant à créer le nouvel oiseau qui pourra voler et les 4 tuyaux
```

### Création des objets

```
self.tuyau0 = Pipe(self)                    ####################
self.tuyau1 = Pipe(self)                    # création des 4 objets
self.tuyau2 = Pipe(self)                    #
self.tuyau3 = Pipe(self)                    ####################
```

### Appel de la fonction « create_pipe »

```
self.tuyau0.create_pipe(1100,350)                  ################################
self.tuyau1.create_pipe(1500, randint(200,400))    # création des 4 tuyaux à l'aide
self.tuyau2.create_pipe(1900, randint(100,300))    # appartenant à la classe Pipe
self.tuyau3.create_pipe(2300, randint(100,400))    ################################
```

Après avoir créé les tuyaux, il faut les **déplacer**. C'est pourquoi il y a la fonction **« move_pipe »** qui sert tout simplement à déplacer les tuyaux horizontalement de gauche à droite les tuyaux. Cette fonction est appelée par la fonction **« bird_move »,** c'est-à-dire 6 fois toutes les 50 ms et elle déplace le tuyau de 2 pixels à chaque tour. Ces valeurs ont été définies de telle sorte à rendre le jeu réaliste.

A chaque déplacement des tuyaux, c'est-à-dire à chaque appel de la fonction « move_pipe », on lance la fonction « verif_pipe » qui contient 3 conditions. Ces 3 conditions servent à détecter :

- Si le tuyau à disparu du jeu, à ce moment là on le supprime et on le recrée tout à droite de l'écran et caché.
- Si le tuyau dépasse l'oiseau, on incrémente le score de 1.
- Si le tuyau se trouve dans la zone de l'oiseau, c'est-à-dire si le centre du tuyau se trouve entre 40 et 210 pixels, on appelle la fonction « verif_bird » qui sert à détecter s'il y a contact.

## IV) Vérification de mort de l'oiseau

La vérification de la mort de l'oiseau se fait à plusieurs endroits du code, on a pu voir qu'on détectait si l'oiseau chutait au sol dans la fonction « bird_move ». Nous allons maintenant voir comment la détection de contact entre le tuyau et l'oiseau s'effectue. Ainsi cette partie s'effectue grâce à la fonction « verif_bird ». Cette fonction est appelée dès que le tuyau passe à proximité de l'oiseau.
La fonction se décompose en plusieurs conditions. La première sert à exécuter le programme seulement si l'oiseau est toujours en vie car dès que l'oiseau meurt, « self.verite = False ».
Ensuite la seconde condition vérifie si l'oiseau est entré en contact avec le tuyau.

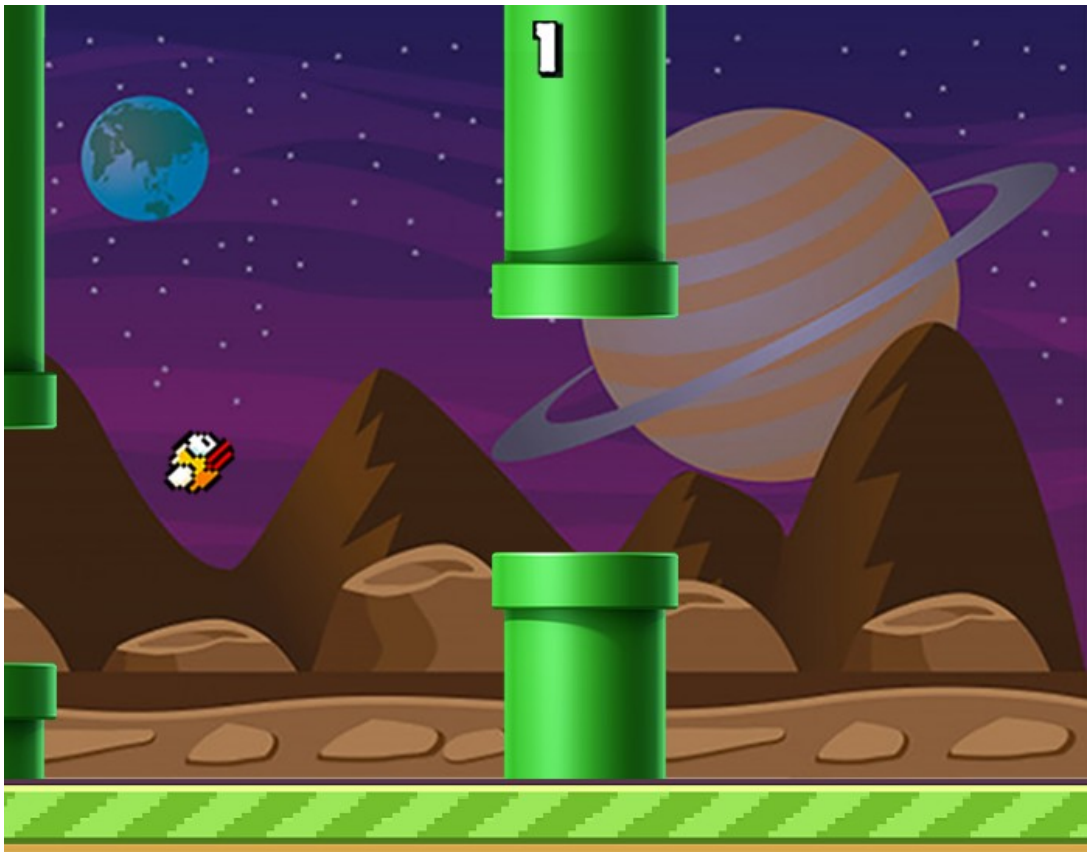
```
if self.y_pipe_down<self.y_center_bird + 24 or self.y_center_bird - 24< self.y_pipe_top
```

Ou bien si l'oiseau a touché le sol.

La variable « b » sert à faire le lien entre le `or b == True:` contact au sol de l'oiseau et la fonction « verif_bird ».

Après avoir vérifié que l'oiseau est mort, il reste une dernière condition servant à déterminer quelle animation doit être lancée. Cette partie a été précisée dans la partie « Animations de début et fin de partie ».

Ci-dessous un exemple où le tuyau a dépassé l'oiseau, le score est alors de 1, ce tuyau va être bientôt détruit et régénéré tout à droite de l'écran (caché) entre 1400 et 1500 pixels. Le jeu tourne en boucle ainsi de suite.



**Difficultés rencontrées lors du développement du Flappy Bird :**

Comme je le dis au début de la présentation du jeu, le défi majeur du projet était de réaliser un jeu qui puisse à la fois être jouable et agréable pour l'utilisateur. J'ai donc passé le plus de temps sur la fonction « bird_move » qui sert à la fois aux déplacements et à l'inclinaison de l'oiseau. Elle est complexe car les variables utilisées ont été trouvées par tâtonnement, il a donc fallu trouver le bon ajustement entre la répétition de la fonction toutes les 50 ms et le déplacement des différents objets sur l'écran. J'ai également consacré beaucoup de temps à la découpe des images et à la création de toutes les images de l'oiseau. Je me suis aussi beaucoup attardé sur la création des tuyaux car j'avais du mal à faire ce que je voulais exactement. Il m'a donc fallu un petit temps de réflexion avant de commencer réellement le jeu.

**Améliorations possibles :**

Etant donné que je me suis beaucoup occupé de la partie des déplacements et de l'inclinaison de l'oiseau, je pense que j'aurais pu trouver une méthode plus précise et plus intuitive. En effet avec notre algorithme, il y a 30 calculs de vérification toutes les 50 ms, avec une autre méthode on aurait peut-être pu augmenter le nombre de vérifications et ainsi diminuer le risque de ne pas détecter l'oiseau même si la méthode présentée est déjà assez précise.

Un autre point à améliorer serait de changer la méthode d'inclinaison de l'oiseau. Dans notre cas, nous avons créé 25 images que l'on affiche à la suite. Une autre méthode serait de ne créer seulement qu'une image et de faire pivoter celle-ci en fonction du cours du jeu. Cette méthode donnerait une meilleure précision et un rendu plus fluide et rendrait le jeu plus agréable à l'utilisateur.

**Bilan Personnel :**

En conclusion, je suis fier du projet que j'ai pu mener avec Dorian au cours de cette fin d'année. Il m'a été très bénéfique au niveau de l'apprentissage du « Python » ainsi que dans la confortation de mon avenir à l'EISTI. Le travail en binôme a aussi été fondamental pour la bonne et joyeuse réalisation du projet. En effet, nous avons passé énormément de temps à travailler ensemble ce qui a créé une forte cohésion et complicité et nous a permis d'avancer très vite. Le temps passé au développement du projet peut être estimé à plus de 200 heures par personne. Finalement il s'agit ici d'un détail en comparaison au plaisir et à l'engouement que ce projet nous a procurés.