

# NEFT

Bibliothèque de neuro-évolution à topologie fixée

Projet informatique réalisé dans le cadre d'une évaluation de fin d'année  
en PeiP2 à Polytech Orléans

Polytech Orléans - Université d'Orléans



28 mai 2021

Léo Poinet

Encadré par  
M. Rémy Leconge

# Sommaire

<b>Sources et recommandations</b>	<b>1</b>
<b>Introduction</b>	<b>2</b>
<b>Mots clés</b>	<b>3</b>
<b>Mise en place du projet</b>	<b>4</b>
<b>Organisation du projet</b>	<b>5</b>
<b>Construction du code</b>	<b>6</b>
La théorie	6
Réseau de neurones	6
Le perceptron	6
Réseau de neurones	7
Algorithme génétique	9
Structure du code	12
<b>Analyse, performance et stabilité</b>	<b>16</b>
<b>Historique des problèmes rencontrés</b>	<b>18</b>
<b>Etat final</b>	<b>20</b>
Points forts et points faibles	20
Aperçu en image	21
<b>Cahier des charges</b>	<b>22</b>
<b>Conclusion</b>	<b>23</b>
<b>Annexes</b>	<b>24</b>
Compilation	24
Code source	25

## Sources et recommandations

- Image Extrait Flappy Bird.  
<https://journalmetro.com/techno/440585/flappy-bird-ce-petit-jeu-quon-aime-detester/>
- [https://fr.wikipedia.org/wiki/Intelligence\\_artificielle](https://fr.wikipedia.org/wiki/Intelligence_artificielle)
- <https://fr.wikipedia.org/wiki/Perceptron>
- Chapitre 9 et 10. Daniel Shiffman. The Nature of Code. 2012.  
<https://natureofcode.com/book/>

# Introduction

*“Open the pod bay doors, HAL”*

- 2001: A Space Odyssey. Stanley Kubrick

Cela fait maintenant plusieurs années que l’intelligence artificielle (IA) connaît un succès croissant. On en entend parler de plus en plus et les interactions sont désormais omniprésentes : moteurs de recherches, réseaux sociaux, jeux vidéo et bientôt voitures autonomes. C’est d’ailleurs vers ces deux dernières catégories que nous allons nous orienter à travers ce projet. En effet, les réseaux de neurones et les algorithmes génétiques sont deux domaines fortement inspirés de la biologie qui, une fois associés, offrent la possibilité de résoudre des tâches complexes.

Voici un petit point récapitulatif quant à l’IA avant de se lancer. C’est Alan Turing lui-même qui semble être le premier à développer cette idée en se demandant si un ordinateur pourrait penser au sens Cartésien. Nous sommes alors en 1950 et six ans après, le terme “Artificial intelligence” apparaît. Avec l’évolution exponentielle de la puissance de calcul des ordinateurs dans le monde, il est devenu envisageable de renouveler les techniques liées à l’IA. Même si nous sommes encore très loin d’atteindre un niveau de pensée comme on l’imagine, les applications ont fait leurs preuves et le futur est très prometteur. L’IA est aujourd’hui un domaine très large qui comporte notamment le *machine learning* puis le *deep learning*. C’est dans ce dernier que nous allons nous situer à travers ce projet.

Il s’agira de développer de A à Z un ensemble de classes alliant réseaux de neurones et algorithme génétique de manière à réaliser une bibliothèque C++ facile d’utilisation que l’on peut intégrer à n’importe quel projet. On prouvera l’apprentissage et la réussite de cette dernière en réalisant un clone d’un jeu bien connu pour sa difficulté malgré sa simplicité : *Flappy Bird*.

# Mots clés

Bibliothèque → Ensembles de classes qui peuvent être réutilisées dans un programme.

Topologie → Nom donné à la structure d'un réseau de neurones.

Neuro évolution → Traduction personnelle non officielle pour “neuroevolution”. Méthode qui consiste à utiliser les algorithmes génétiques pour entraîner les réseaux de neurones.

IA → Intelligence Artificielle.

NN → Neural Network / Réseau de neurones.

Population → Nom de la classe au cœur de l'algorithme génétique.

Organism → Nom de la classe donnée à un individu de la classe population.

Flappy Bird → Référence au vrai jeu ou au clone du projet.

Makefile → Fichier qui lance un ensemble de commandes selon des règles de dépendances et qui réduit ainsi les temps de compilation.

CMake → Programme qui génère les Makefile de manière automatique.

Framerate → Nombre d'images par seconde.

IHM → interface homme-machine.

# Mise en place du projet

Voici un cahier des charges global du projet pour mettre en place les différents objectifs.

Nous devons réaliser une bibliothèque de neuro évolution. Cette étape est longue et peut s'avérer risquée mais heureusement nous disposons déjà des classes toutes faites en Java que nous devons porter en C++. En effet j'ai réalisé ces classes sur un projet expérimental antérieur qui permettait de résoudre le problème classique XOR avec la neuro évolution. Ce problème est normalement réalisé avec un réseau de neurones en utilisant d'autres méthodes sans algorithme génétique.

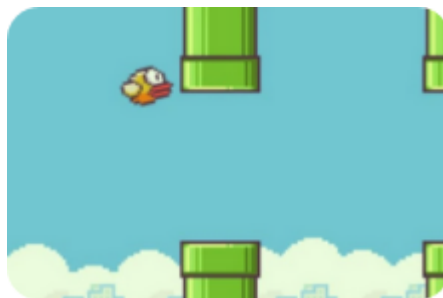
L'utilisation d'un tel algorithme propose des avantages qui seront profitables au test que nous allons imposer à la bibliothèque : Il faut que l'IA soit capable de gagner à un jeu.

*"The Matrix has its roots in primitive arcade games"*

- Neuromancer. William Gibson

Pourquoi *Flappy Bird* ? C'est un jeu tout à fait adapté au projet. Le temps de développement pour cette partie est très court (une demie-journée). Il nous faut donc un jeu simple mais qui permet de tester efficacement notre IA. Au-delà de la difficulté que présente le jeu même pour un être humain, la position aléatoire des murs en fait un candidat idéal pour vérifier que notre IA n'apprend pas par cœur un chemin prédéfini mais comprend le mécanisme et la physique derrière le jeu.

Pour réaliser ce dernier, on utilisera la bibliothèque open source SFML. En effet c'est une bibliothèque qui dispose de toutes les fonctionnalités dont nous avons besoin est qui n'est pas trop bas-niveau.



Extrait du jeu développé par Nguyễn Hà Đông en 2013

# Organisation du projet

Voici le calendrier du projet. En noir, les croix prévisionnels faites le 17 mai. En bleu, les étapes validées au fur-et-à-mesure. En rouge, les retards.

Tâches	17-mai	18-mai	19-mai	20-mai	21-mai	22-mai	23-mai	24-mai	25-mai	26-mai	27-mai	28-mai
Réalisation du jeu												
Classe Bird	X X											
Classe Wall	X X											
Classe WallManager	X X											
Gérer collisions Wall/Bird	X X											
Réalisation de l'IA												
Porter les classes Matrix2, Layer, NN et DNA de Java vers C++		X X										
Classe abstraite Organism		X X										
Porter la classe Population de Java vers C++. L'adapter avec la classe Bird (qui hérite de Organism)		~	X X									
Utilisation de l'IA dans le jeu et validation												
Harmoniser les classes Bird, Organism et Population entre elles (càd généraliser Population sur Organism et non pas Bird)			X	X								
Debugger, analyser puis valider l'apprentissage de l'IA				X	X X	X X		X X	X			
(Optionnel) Interface utilisateur												
Intégrer et compiler la bibliothèque Dear ImGui au projet						X X		X X				
Ajouter les composants d'interfaces utilisateurs : Jeu, Environnement, Données								X X	X X			
Ecrire le rapport		X X	X X	X X	X X	X X		X X	X X	X X	X X	
Préparer la soutenance										X X	X X	X

Il peut être intéressant de constater que nous avons donné beaucoup de temps et d'importance à l'étape de débogage de l'IA. En effet c'est une étape critique car il ne s'agit pas seulement d'avoir un code qui passe la compilation. La moindre erreur demande de repasser sur l'intégralité du code.

# Construction du code

## La théorie

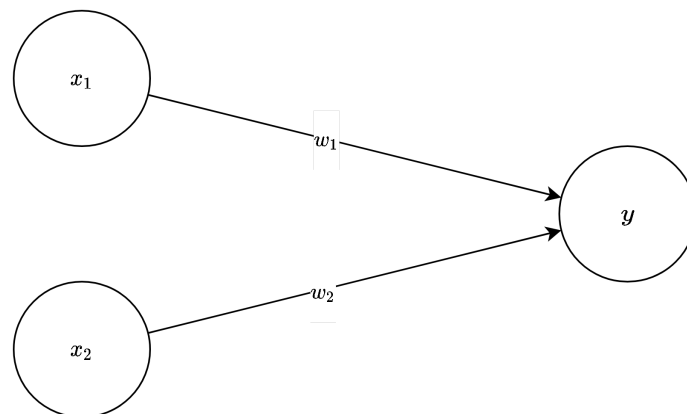
Avant de se lancer, il est nécessaire d'avoir une base théorique minimale pour bien définir à l'avance la structure du programme. Il faut réussir à relier deux notions qui peuvent être employées séparément. Le réseau de neurones sera le cerveau de nos instances. C'est-à-dire, le preneur de décisions. Tandis que l'algorithme génétique sera l'entraîneur de ce dernier.

Nous allons donc voir brièvement ces deux notions fondamentales dans cette partie.

## Réseau de neurones

### Le perceptron

Inventé en 1957 par Frank Rosenblatt au laboratoire d'aéronautique de l'université de Cornell, c'est le réseau de neurones le plus simple que l'on peut obtenir. Son défaut est qu'il ne peut résoudre que des problèmes **linéaires**. Nous reviendrons sur cette notion de linéarité plus tard et pourquoi c'est un défaut dans notre cas.

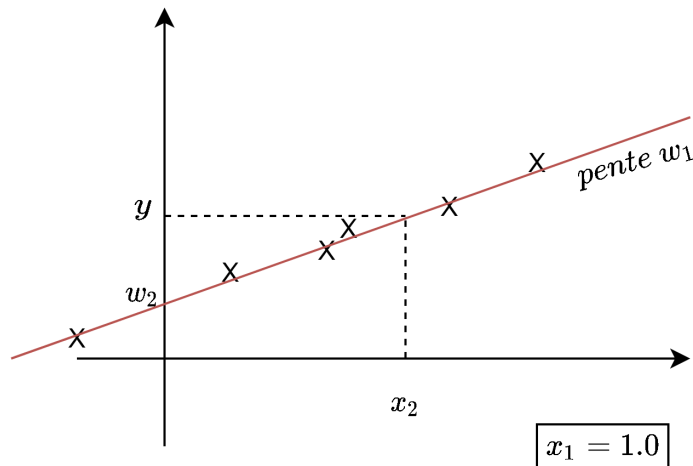


Le perceptron

Avec en entrée  $x_1$  et  $x_2$ , on obtient en sortie la somme des poids avec l'entrée :

$$y = w_1.x_1 + w_2.x_2$$

En entraînant le perceptron sur un certain nombre de données, on peut donc trouver la combinaison de poids qui donnent un modèle représentant correctement l'ensemble des données. On peut alors faire des **prédictions** pour des valeurs intermédiaires aux échantillons (cf figure suivante).



Graphique du modèle (perceptron entraîné)

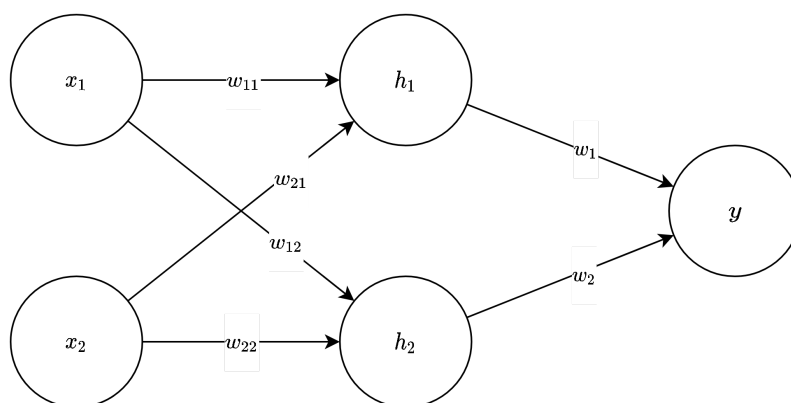
A noter ici que  $x_1$  est une entrée constante égale à 1. C'est ce que l'on appelle le biais (**bias**).

On remarque que si les échantillons ne forment plus une droite mais une courbe, on ne pourra jamais trouver de solution générale qui en définisse correctement l'ensemble. En effet, quel que soit le nombre de poids que l'on ajoute, le modèle restera toujours linéaire.

Il nous faut un moyen de passer au-delà de cette linéarité pour notre jeu.

### Réseau de neurones

Le réseau de neurones ressemble fortement au perceptron mais a l'avantage de pouvoir résoudre des problèmes **non linéaires**. Voici une structure possible.



Réseau de neurones

Pour chaque **couche**, on peut associer une **matrice** de poids avec en entrée le **nombre de lignes** et en sortie le **nombre de colonnes**.



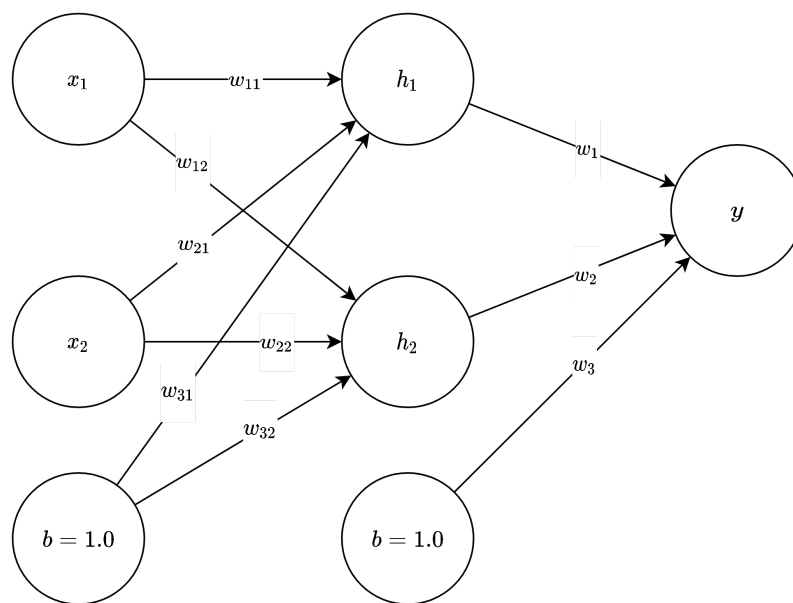
Avec en entrée  $x_1$  et  $x_2$ , on obtient en sortie la somme des poids avec l'entrée en passant par une couche cachée (hidden layer) :

$$\begin{bmatrix} h_1 & h_2 \end{bmatrix} = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = [w_{11} \cdot x_1 + w_{21} \cdot x_2 \quad w_{12} \cdot x_1 + w_{22} \cdot x_2]$$

$$\begin{bmatrix} y \end{bmatrix} = \begin{bmatrix} h_1 & h_2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = [w_1 \cdot h_1 + w_2 \cdot h_2]$$

A noter que l'entrée est un vecteur et la sortie aussi. On aurait très bien pu avoir deux sorties ce qui aurait transformée la deuxième couche en une matrice  $2 \times 2$ . L'utilisation de matrices ici est très puissante car elle ouvre la possibilité d'avoir un nombre d'entrées, de sorties et de couches cachées flexible tout en gardant la même méthode de calcul.

On utilisera le biais de la manière suivante. On ajoute une ligne à toutes les matrices de chaque couche et les étapes intermédiaires se verront ajouter un réel égal à 1,0.



Réseau de neurones avec bias

Plusieurs méthodes telle que la *backpropagation* permettent de faire apprendre le réseau de neurones, c'est-à-dire trouver la bonne combinaison de poids qui donne le bon résultat sur un ensemble de données.

Nous allons utiliser une méthode qui consiste à tester un grand nombre de possibilités. Bien que peu performante, elle permet de découvrir des solutions adaptatives et n'a pas besoin de données d'entraînement au préalable.

## Algorithme génétique

*“Ape alone... weak. Apes together strong.”*

- Rise of the planet of the apes. Rupert Wayatt

L'algorithme génétique est un ensemble d'étapes inspirées de la sélection naturelle : Seules les individus d'une population qui survivent et qui s'adaptent à leur environnement restent ou dominent leur population.

Les individus possèdent un génotype : l'ADN. Celui-ci s'exprime sous forme d'un phénotype (caractéristiques visibles).

Dans le cas de l'algorithme, l'ADN est un **tableau de réels** et le phénotype est la caractéristique que l'on tire de cet ADN. Cela peut être une lettre, une forme, une couleur, un vecteur, une vitesse, une décision, etc... C'est ce qui est modifié et pris en compte directement par l'algorithme.

Il est très important et délicat de définir correctement le phénotype par rapport au résultat attendu car il influence le génotype et inversement.

Exemple :

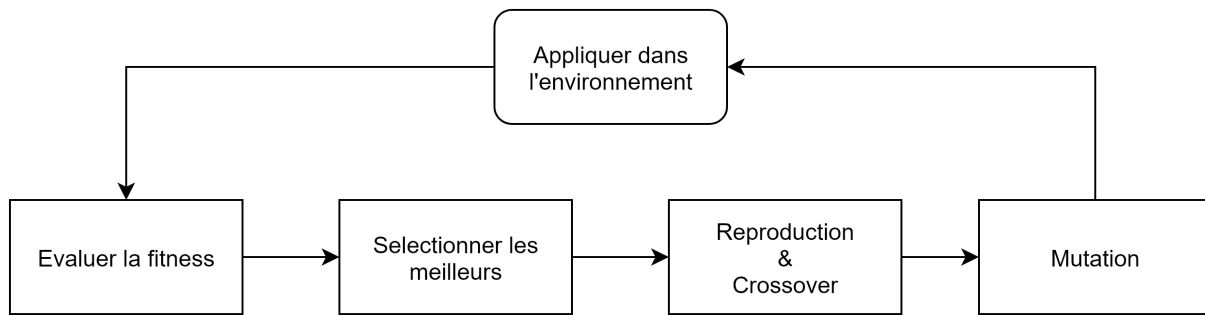


On peut par exemple imaginer un canon qui doit tirer sur une cible. Le génotype pourra comporter un tableau de deux réels :

- Un pour l'angle de tir
- Un pour la force de tir

Il serait donc inutile par rapport au résultat souhaité d'exprimer la forme ou la couleur du canon. A moins que cela ait une influence sur le tir.

L'ADN est généré de manière aléatoire. Une fois que nous avons bien défini le phénotype, il faut que la population converge vers la solution.



### Cycle de la population sur une génération

#### Evaluer la fitness :

La fitness est la mesure de **l'adaptation** d'un individu à son environnement. Une forte fitness signifie que l'individu correspond bien à son environnement. On peut voir cela comme un score.

Pour l'exemple du canon on pourrait la définir comme étant  $\frac{1}{\text{distance à la cible}}$

#### Sélection :

On sélectionne les individus de la population selon leur fitness. En effet, on veut augmenter les chances qu'un individu à forte fitness soit reproduit. Les fitness de chaque individu sont normalisées (la somme doit être égale à 1) pour représenter un pourcentage. C'est pour cette raison qu'il est intéressant de multiplier (ex: mettre au carré) en amont le score des individus de manière à démarquer ceux ayant particulièrement bien réussi.

On construit ensuite une **mating pool** qui contiendra les parents en proportion selon leur fitness.

A noter qu'il est essentiel de garder les individus qui n'ont pas forcément bien réussi car cela augmente la diversité et peut potentiellement faire émerger un meilleur individu grâce au Crossover.

#### Reproduction & Crossover :

Il s'agit de prendre l'information génétique d'un parent A et de la mélanger avec celle d'un parent B, tous les deux sélectionnés **aléatoirement** dans la mating pool définie précédemment. On procède de cette manière :

Il existe plusieurs méthodes pour réaliser le Crossover mais nous choisissons la plus simple qui consiste à découper l'information en deux.

$\mu$  est un nombre aléatoire qui décide du point de séparation. Il est compris entre 0 et la taille de l'information génétique. Dans l'exemple suivant,  $\mu = 3$  et  $\mu \in [0, 7]$

Parent A

0.4	-0.5	0.2	0.1	-0.4	-0.4	0.9
-----	------	-----	-----	------	------	-----

Parent B

-0.8	0.1	0.4	1.0	0.2	-0.5	0.7
------	-----	-----	-----	-----	------	-----

Enfant résultant

0.4	-0.5	0.2	1.0	0.2	-0.5	0.7
-----	------	-----	-----	-----	------	-----

### Mutation

Pour chaque gène, on applique une mutation aléatoire contrôlée par le taux de mutation (**mutation rate**).

$\varepsilon$  est un nombre aléatoire compris entre 0 et 1. Dans l'exemple suivant,  $\varepsilon < \text{mutation rate}$  pour la case d'indice 0 et 4 :





Enfant

0.4	-0.5	0.2	1.0	0.2	-0.5	0.7
-----	------	-----	-----	-----	------	-----

Enfant après mutation

0.4	-0.5	0.2	1.0	-0.9	-0.5	0.7
-----	------	-----	-----	------	------	-----

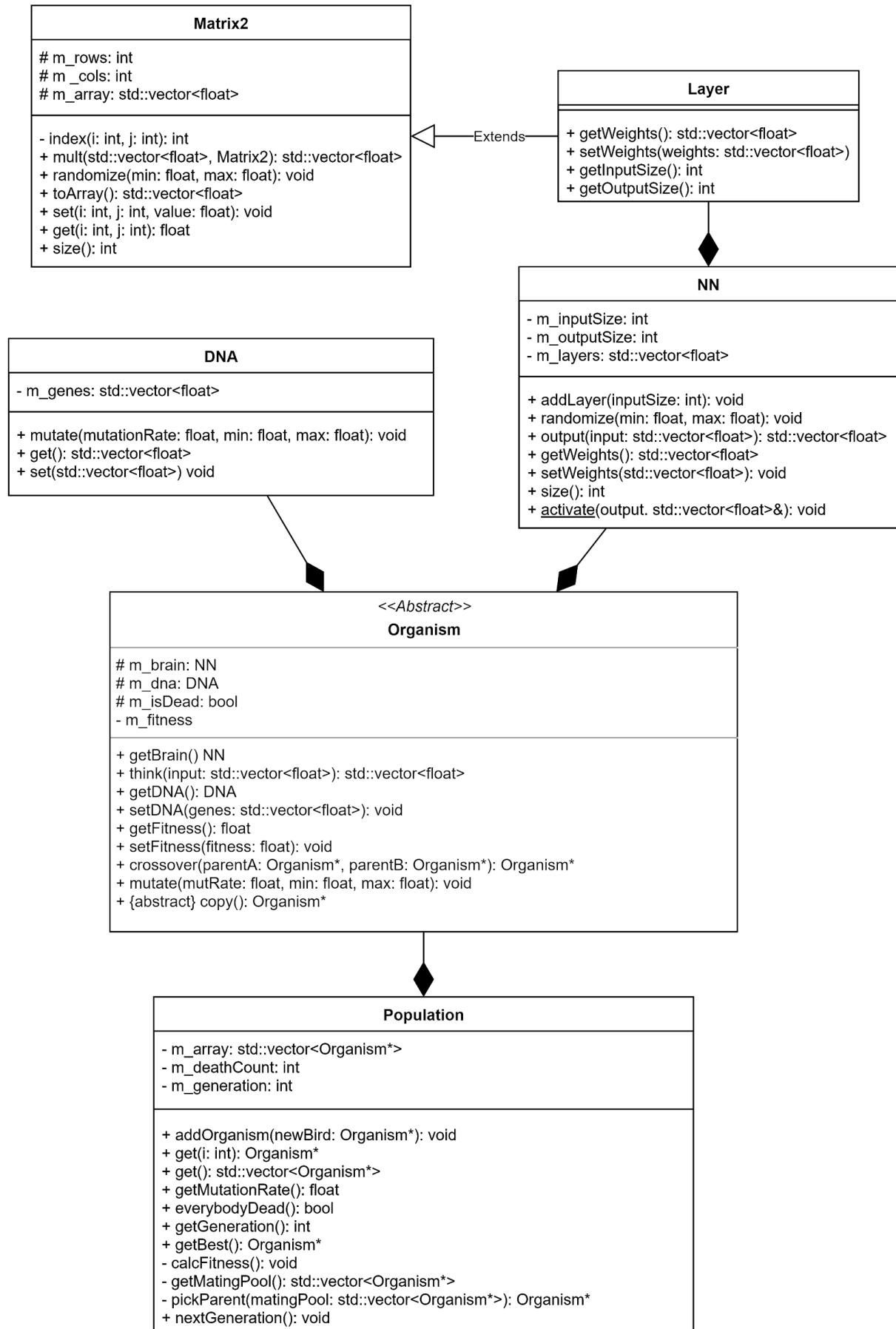
Les mutations

-  Augmentent les chances de tomber sur la bonne solution rapidement
-  Empêchent la population de s'uniformiser et de bloquer sur le même individu
-  Augmentent les capacités d'adaptation à un nouvel environnement **pendant ou après l'apprentissage**
-  Empêchent la convergence si le taux est trop élevé. (Les valeurs deviennent aléatoires et il n'y a plus d'apprentissage)

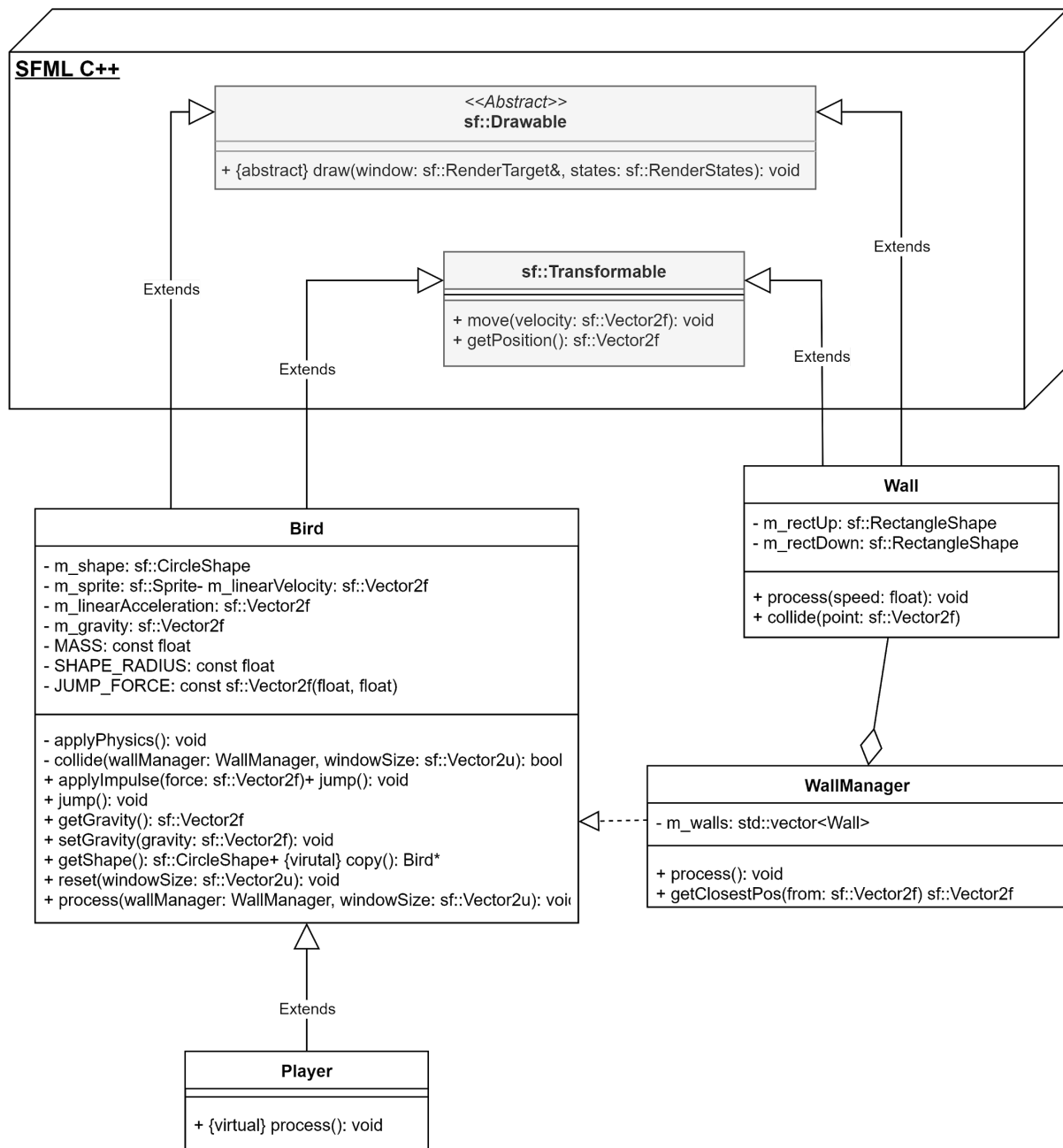
Après ces étapes, on peut réitérer le cycle vers une nouvelle génération jusqu'à obtention d'un phénotype satisfaisant. Le critère étant fixé par la définition de la fitness.

# Structure du code

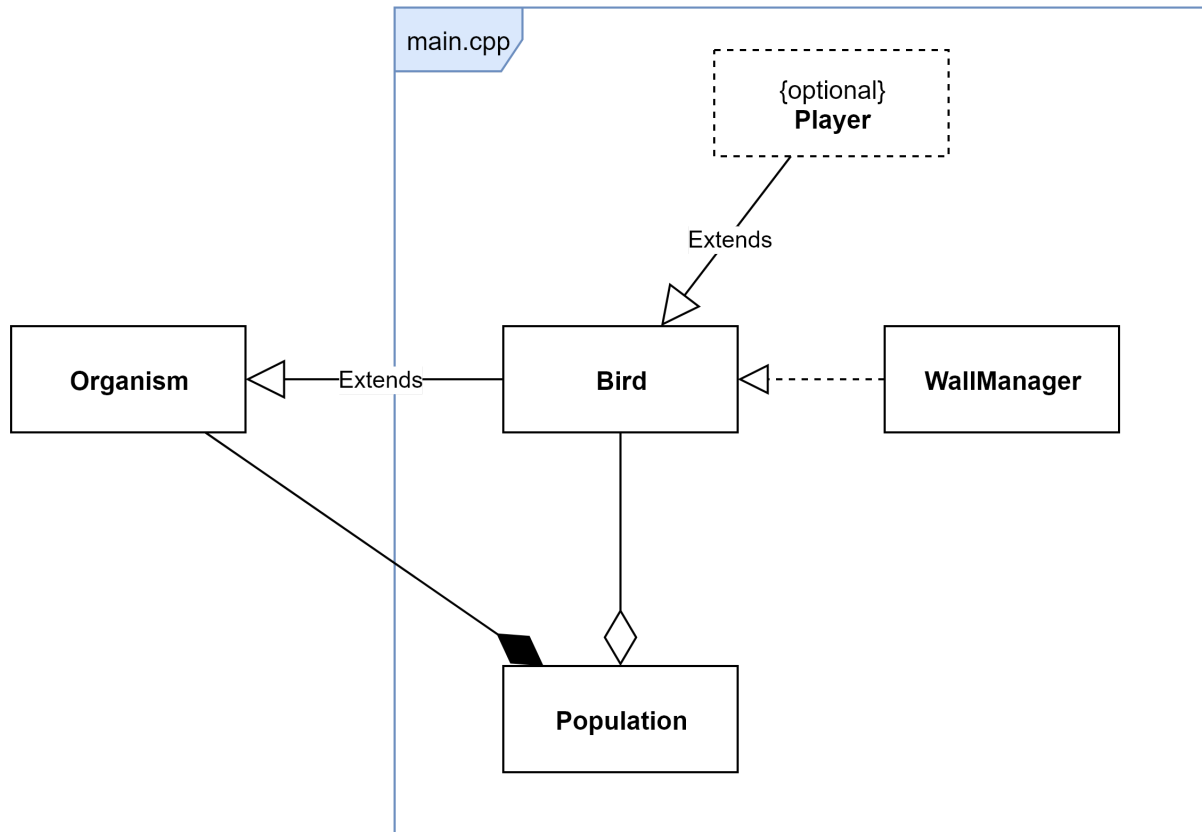
## La librairie NEFT



## Clone Flappy Bird



### Structure générale du main.cpp



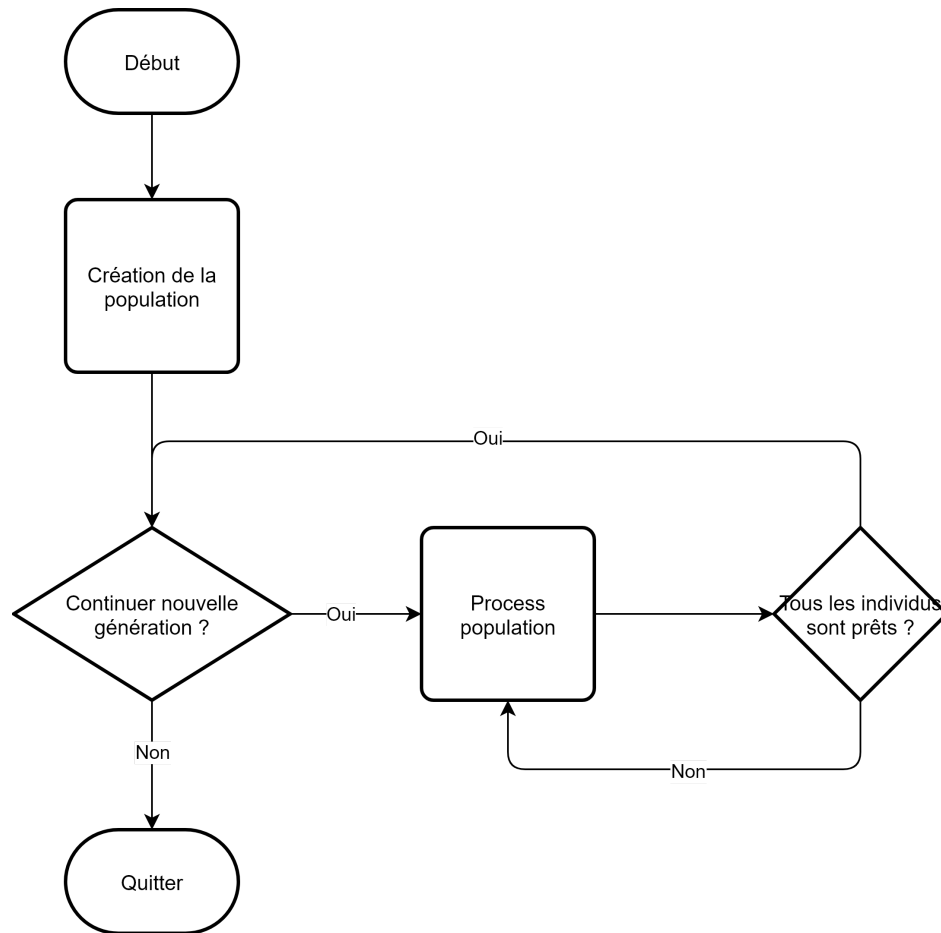
La structure du main.cpp ci-dessus est la base que l'on pourra utiliser pour tout autre projet utilisant NEFT. En effet le choix à été de laisser la possibilité de mettre n'importe quoi dans la population qui est un *Organism\**. D'où l'utilisation d'une classe abstraite et de pointeurs. On garde ainsi la modularité.

Dans notre cas, on met dans la population des *Bird\**. On peut ainsi appeler les méthodes propres à *Bird* de la manière suivante.

```
for (Organism* organism: population)
{
    Bird* bird = (Bird*) organism; // cast sur Bird*
    bird->method();
}
```

Et la population pourra travailler sur les *Bird\** avec les méthodes de *Organism*.

### Diagramme minimal d'utilisation de NEFT



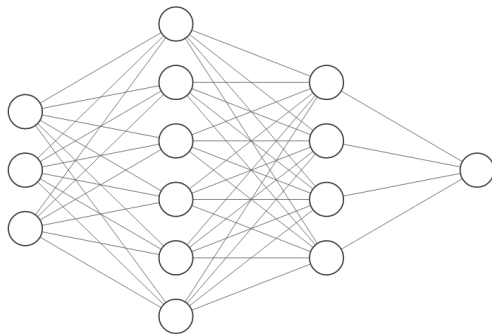
Voici le diagramme général nécessaire au bon fonctionnement de la bibliothèque. Dans le cas de Flappy Bird, on crée la population avec des *Bird\**. A chaque nouvelle image, on fait jouer tous les Birds. S'ils sont tous morts dans la population, on appelle la méthode *nextGeneration()* sur celle-ci et on recharge le jeu. La particularité est que l'on entraîne pas la population sur un nombre fini d'itérations. En effet, c'est lorsque l'utilisateur ferme la fenêtre que l'entraînement s'arrête vraiment. Ainsi, il peut laisser travailler la population jusqu'à satisfaction puis enregistrer les données grâce à l'interface utilisateur.

A noter que c'est à la responsabilité du programmeur de la classe *Bird* d'appeler *setDead()* lorsqu'il y a collision avec un mur mais aussi d'évaluer correctement la *fitness*. C'est le compromis nécessaire à la conservation de la modularité. En effet, s'il s'agissait de faire un pilote automatique par exemple, on pourrait définir la *fitness* avec la distance à la cible et le temps de parcours, qui sont deux paramètres complètement différents du simple score que l'on applique dans *Flappy Bird*.



# Analyse, performance et stabilité

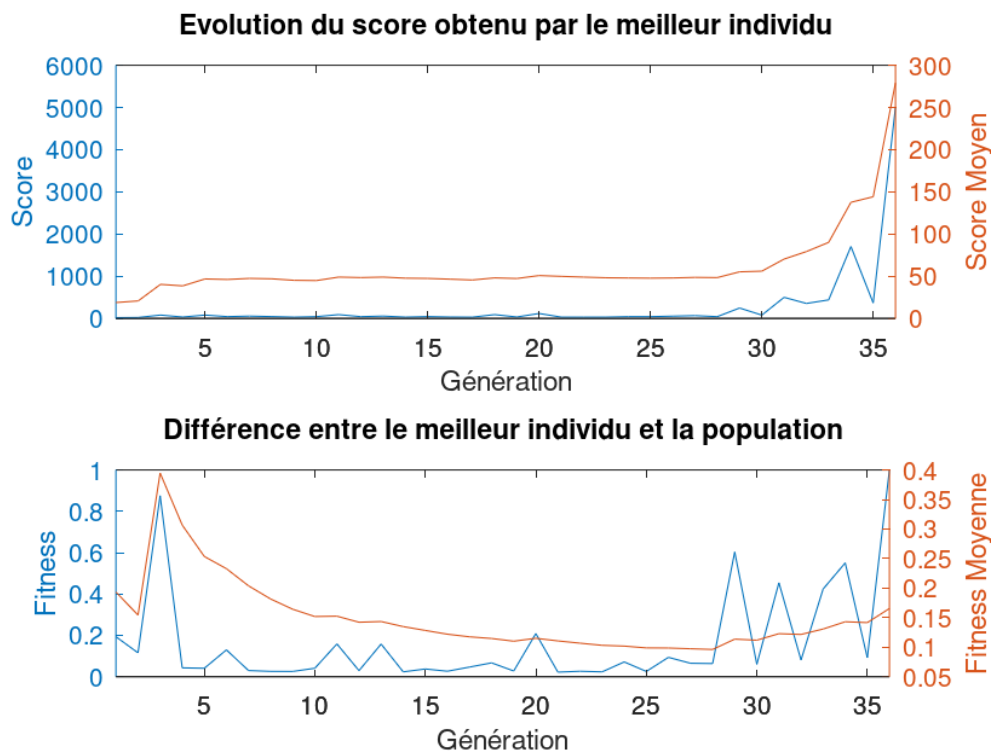
Voici un exemple de données obtenues avec une population de **100** individus, un taux de mutation de **1%**, et les paramètres d'environnement **par défaut**. On donne en entrée : la vitesse verticale, et un vecteur (x et y) vers la sortie du prochain mur.



la configuration du NN de Bird

**NN(3, {6, 4}, 1)**

Input Layer  $\in \mathbb{R}^3$     Hidden Layer  $\in \mathbb{R}^6$     Hidden Layer  $\in \mathbb{R}^4$     Output Layer  $\in \mathbb{R}^1$



Chaque graphique sera différent car la population est initialisée aléatoirement. Bien que peu probable, on peut obtenir le meilleur individu possible à la toute première génération.

Cependant on remarquera des points communs entre tous les graphiques qui sont bien visibles sur celui présenté ci-dessus.

### Evolution du score

Après avoir fait plusieurs tests, on considère que l'IA a battu le jeu lorsqu'elle dépasse les 5000 de score. On pourrait éventuellement démontrer cela en prenant le nombre de murs passés en fonction du score sachant que leurs position est donnée par une loi uniforme.

On remarque que jusqu'à la génération 29, l'IA n'avance pas. Aucun individu n'a passé le premier mur et on attend une/des mutation(s) au bon endroit. A partir de la génération 30, le premier individu passe le mur et va être reproduit d'avantage. Le score moyen augmente très vite car c'est maintenant une partie de la population qui va de plus en plus loin. On attend cette fois ci des mutations ou le bon crossover qui améliorent ceux qui réussissent.

Finalement, on converge jusqu'à obtenir un individu parfait qui ne meurt jamais. L'IA s'est adaptée à l'ensemble des configurations de murs possibles.

### Différence entre le meilleur individu et la population

Ce graphique représente les moments où il y a eu un ou des individus qui sont sortis du lot. Il est intéressant de constater que les plus grosses démarcations sont en lien direct avec l'augmentation du score.

Il y a parfois des améliorations mais pas suffisamment conséquentes pour être retenues. Cela vient du fait que l'on incrémente la *fitness* linéairement. On pourrait l'incrémenter au carré pour augmenter la probabilité de reproduire un individu ayant plus réussi.

A noter que plus on augmente la complexité des couches cachées, plus on est susceptible de voir apparaître des comportements "intelligents". En revanche, cela augmente considérablement le nombre de générations nécessaires pour résoudre le problème.

### Performance et stabilité

Côté performance, on tire un avantage du C++. Sur mon PC, le framerate commence à ralentir à partir d'une population de 1100 *Bird* et le temps de désallocation / réallocation mémoire pour une nouvelle génération se fait sentir (~1s).

Côté stabilité, le framerate est constant et je n'ai jamais eu affaire à un crash soudain. L'utilisation de la mémoire est constante. (Plus d'informations dans la partie historique des problèmes : fuite de mémoire)

# Historique des problèmes rencontrés

Voici une liste des différents problèmes rencontrés lors du développement. Elle pourra permettre à d'éventuels successeurs de comprendre certains choix de programmation ou de trouver des solutions à d'autres problèmes.

## Intégration de SFML

Il était difficile de compiler la bibliothèque SFML avec le projet. C'est une étape à laquelle on pense peu et où il aurait été judicieux de prévoir plus de temps. La seule version qui fonctionna fut la suivante : SFML 2.5.1 GCC 7.3.0 MinGW (SEH) - 64-bit. La version 32-bit présentant des erreurs de linking probablement à cause de la version de MinGW précédemment installée (64-bit). L'utilisation massive de Google.com ont permis de résoudre ce problème.

## Intégration de ImGui

De même, cette bibliothèque était difficile à intégrer car nous utilisons ce dépôt github <https://github.com/eliasdaler/imgui-sfml> pour la compatibilité avec SFML. De plus, la bibliothèque ImGui ne propose aucune documentation. Les fonctionnalités de l'IDE et encore l'utilisation de Google.com ont permis de résoudre ce dernier point.

## Makefile ou CMake

Pour limiter les temps de compilation et rendre le projet crossplatform il s'agissait au début d'utiliser un Makefile. Or, j'ai découvert CMake lors des problèmes de compilations avec ImGui. L'utilisation de ce dernier s'est avérée bien plus optimisée et efficace que la création manuelle des Makefile bien que je ne possède que les bases.

## Difficulté d'accès à la taille de la fenêtre

Pour placer les murs dans le jeu et faire d'autres opérations, il est souvent nécessaire d'obtenir la taille de la fenêtre. On aurait pu utiliser une classe statique ou une variable globale mais pour éviter les confusions et les bugs introuvables, j'ai décidé de faire des passages par paramètre.

## La classe Utils.cpp

Le projet utilise souvent les mêmes fonctions telles que *randrange(float, float)*. Je me suis inspiré de plusieurs autres projets pour créer une classe *Utils* avec des méthodes statiques qui sont partagées sur l'ensemble du projet.

## La classe Matrix et pointeurs

J'ai rencontré beaucoup de problèmes avec l'utilisation de tableaux de pointeurs à deux dimensions. Puis je suis tombé sur une technique très efficace qui permet de faire une matrice avec un simple tableau en définissant le nombre de lignes et de colonnes. On accède à un élément avec la méthode *index(i, j)*.

Pour rendre la bibliothèque plus solide et pour limiter les pertes de temps dues aux pointeurs, j'ai décidé d'en utiliser le moins possible et de remplacer les tableaux par *std::vector*.

### Organism, Population et Modularité

J'étais bloqué après avoir fait la classe population car il fallait appeler des méthodes de *Bird* depuis le main mais la population renvoie un tableau de *Organism\**. La première solution était d'ajouter des méthodes abstraites de *Bird* dans la classe *Organism*. L'inconvénient de cette dernière est que pour chaque projet utilisant la bibliothèque NEFT, il faut adapter *Organism*.

Finalement la solution à ce problème fut de faire un cast sur *Bird\** dans le main ce qui permet non seulement d'appeler les méthodes propres à *Bird* mais aussi de laisser travailler la *Population* sur les *Bird* qui sont des *Organism*.

Je tiens à remercier M. Leconge pour cette technique.

### Fuite de mémoire

Pour vérifier qu'il n'y avait pas de fuite de mémoire, j'ai passé le programme dans le logiciel Valgrind. Premièrement, une grosse fuite était détectée. Celle-ci venait d'une mauvaise gestion dans la méthode *nextGeneration()* de la classe *Population*.

Or il y avait toujours une fuite même sans création de la *Population* dans le main (~64 à ~1024 bytes). Pour aller plus loin, j'ai utilisé l'outil inspecteur de mémoire de Visual Studio (qui m'a par ailleurs confirmé que *nextGeneration()* été corrigé car la mémoire était désormais constante tout le long de l'exécution malgré les conditions propices). Aucun problème n'était reporté.

Je suis ensuite tombé sur plusieurs forums qui ont fait référence au même problème avec Valgrind et SFML. Il s'agirait donc de mémoire vidéo qui est gérée après que Valgrind finisse son analyse. J'ai donc testé le programme le plus simple que l'on puisse obtenir avec SFML (ouvrir une fenêtre). Et celui-ci a bien trouvé une fuite.

Après avoir vérifié l'intégralité du programme à la recherche de pointeurs mal gérés, je me suis conformé à cette idée et ai considéré le problème comme résolu.

### L'IA va loin mais n'atteint pas l'infinie

Au bout d'un grand nombre d'itération, on doit voir l'IA résoudre le jeu. C'est-à-dire qu'elle ne se prend jamais de murs. Or je n'arrivais pas à obtenir ce résultat. J'ai tenté de faire varier les paramètres de l'IA (nombre de génération, nombre de neurones, changer les entrées, nombre d'individus...) mais ce problème était récurrent. Après avoir observé plus en détail le comportement des *Bird*, je me suis rendu compte qu'ils avaient tendance à sauter trop tôt dans les cas "extrêmes" (un mur tout en bas puis un mur tout en haut et inversement).

La solution a été de changer le mur visée **après** que le *Bird* ait complètement dépassé le mur. Se référer à la fonction *getTargetPos()* dans *WallManager*.

# Etat final

## Points forts et points faibles

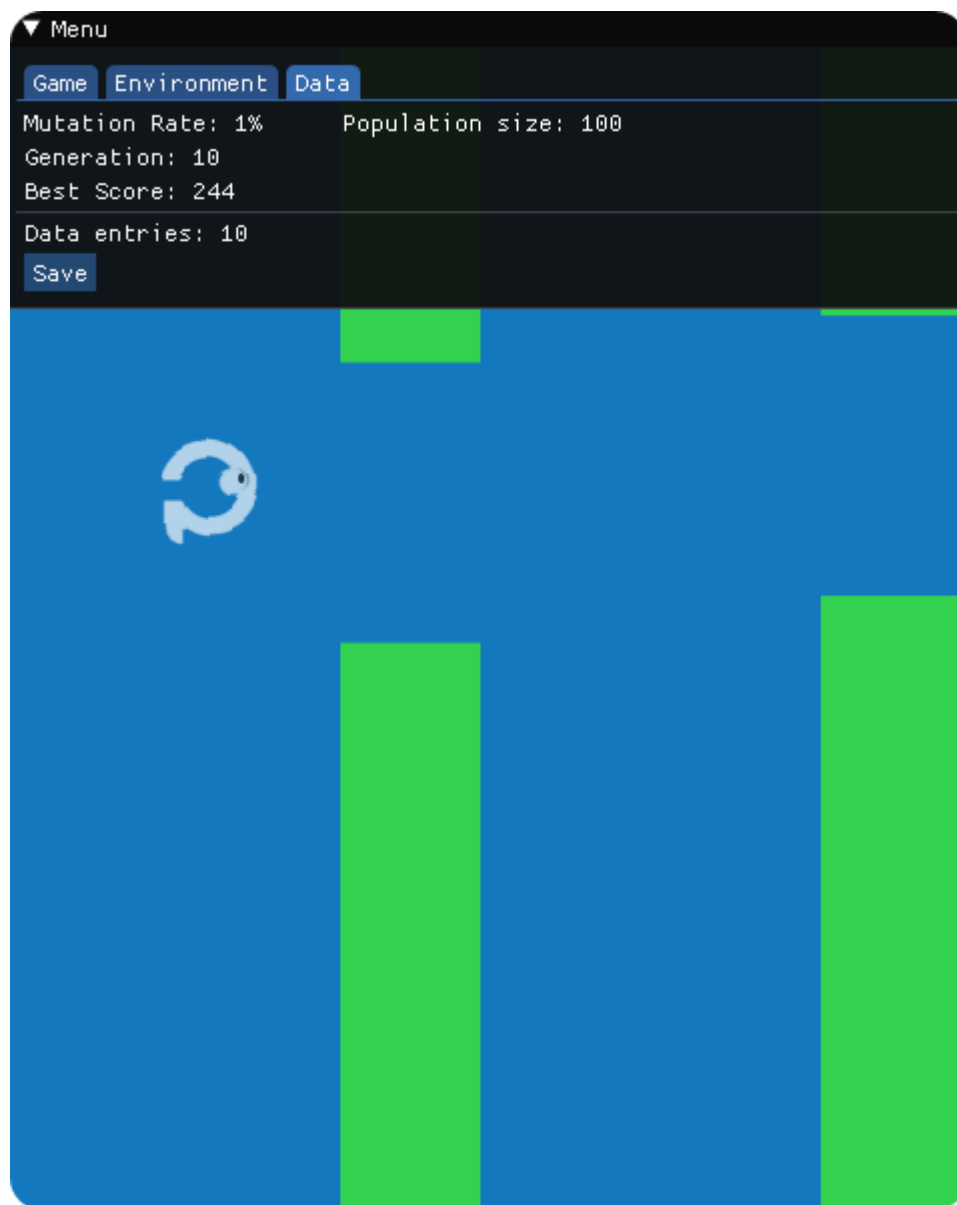
Points forts :

- Modularité. Il est facile d'utiliser la bibliothèque dans n'importe quel autre projet. Il suffit de faire hériter une classe de *Organism*, de définir la fitness puis d'utiliser la *Population*).
- Performances. La *Population* peut accueillir un grand nombre d'individus.
- Expérimentation temps réel par l'utilisateur grâce à l'IHM. Côté *Flappy Bird*, il est possible d'expérimenter pour voir les performances de la bibliothèque.
- Nombre de couches du NN au choix. Cela n'était pas prévu mais il est tout à fait possible de faire un perceptron. Il suffit de construire un *Organism* avec (nombre d'entrées, {0}, nombre de sorties). De même, on peut ajouter autant de couches que l'on souhaite. Ex: (nombre d'entrées, {3, 14, 6, 2}, nombre de sorties) → 4 couches cachées.

Points faibles :

- Compilation du *Flappy Bird* uniquement sur Windows. Après l'avoir passé sous CMake, je n'ai pas réussi à compiler sur une autre plateforme que Windows (des erreurs avec le cmake de imgui-sfml qui ne détecte pas SFML).
- Pas de réglages de la taille de la population pendant l'apprentissage. J'ai décidé de ne pas ajouter cette fonctionnalité car cela me semblait trop complexe et risqué. En revanche, il serait intéressant de faire une taille de population adaptative en temps réel pour améliorer les performances.
- Topologie fixée du réseau de neurones. Il pourrait être fortement profitable de faire varier la topologie des réseaux de neurones des individus pour exploiter pleinement le potentiel de l'algorithme génétique.
- Méthode de crossover. On sépare l'information des deux parents en deux pour faire un enfant. Il existe de meilleures méthodes comme faire un mélange gène par gène.
- Fonction d'activation : tangente hyperbolique. Le réseau de neurones est bloqué sur des valeurs entre -1 et 1 à cause de cela. On pourrait ajouter à l'utilisateur la possibilité de donner sa propre fonction d'activation. Il faudra alors adapter les valeurs aléatoires assignées (ex: entre 0 et 1 si la fonction utilisée est sigmoid).
- main.cpp trop long. Il aurait été judicieux de découper le main avec des classes.

## Aperçu en image



Extrait du programme final

# Cahier des charges

Voici le cahier des charges qui permet de réaliser le projet

- Une connaissance préalable des réseaux de neurones et des algorithmes génétiques est nécessaire.
- Une connaissance approfondie des mécanismes de programmation pour réaliser des jeux ou des environnements graphiques est nécessaire (le temps de développement de cette partie ne doit pas prendre plus d'une journée).
- Le réseau de neurones doit être flexible (nombre et taille de couches cachées indéterminés).
- Le jeu ou environnement qui permet de démontrer l'efficacité doit être de préférence non résolu avec un simple algorithme.
- Le jeu doit ressembler à l'original si c'est un clone (vitesse, mouvements, taille des murs, contrôles, etc...)
- La stabilité logicielle est une priorité majeure.
- La bibliothèque s'adresse à des programmeurs C++ (héritage, classe abstraite, pointeurs...) et doit rester modulable.
- Le temps de développement total ne doit pas prendre plus de 7 jours.
- La bibliothèque finale doit absolument fonctionner (le score moyen du meilleur individu ne doit qu'augmenter et surtout ne jamais redescendre à l'état d'origine. Il faut *visualiser* une amélioration nette ou plus simplement résoudre le jeu si possible)

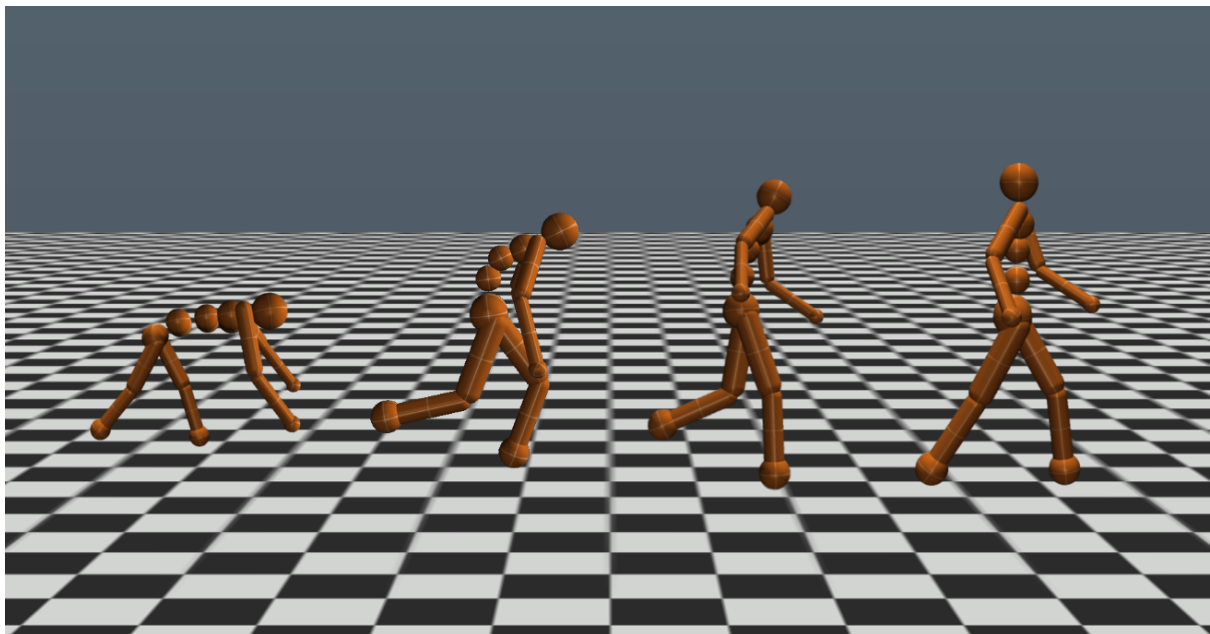
# Conclusion

Ce projet fut très intéressant à réaliser bien que risqué. En effet, le test de validation final intervient très tard dans le processus de développement et aurait pu s'avérer problématique si aucune solution évidente n'était survenue. D'où l'organisation prévisionnelle nécessaire au bon développement du projet : cahier des charges, calendrier et dates butoires. Il n'est pas étonnant que ces processus soient utilisés en entreprises. Ils permettent de rester focalisé et de ne pas s'égarer. Cela pour optimiser le travail car l'on sait en permanence qu'il n'y a pas de temps à perdre.

On peut aussi remarquer que la programmation orientée objet est au cœur du projet et qu'elle est, selon moi, absolument indispensable. Elle permet de facilement passer d'une vue de l'esprit à des lignes de codes. On exploite d'ailleurs toute la puissance de ce paradigme avec l'héritage, le polymorphisme et les classes abstraites.

Cependant le C++ n'aurait pas été mon premier langage de choix pour ce projet. Utiliser un moteur de jeu tel que *Unity* ou *Godot* aurait considérablement réduit le temps de développement car ils proposent des langages plus haut niveau comme le C# ou le GDScript ainsi qu'une API et un grand nombre d'objets tout fait. Je pense notamment aux RayCast2D qui auraient permis d'explorer de nouvelles possibilités d'entrées pour le réseau de neurones.

Pour finir, j'ai été ravi d'avoir eu la possibilité de réaliser ce projet que je réutiliserai certainement dans le futur. C'était une expérience complète et enrichissante.



Deep neuroevolution

<https://eng.uber.com/deep-neuroevolution/>



# Annexes

## Compilation

Voici les étapes à suivre pour compiler le projet Bird.

- Ne pas avoir peur des maux de tête.
- Télécharger et installer CMake. <https://cmake.org/download/>
- Télécharger et installer mingw-w<sup>64</sup> (compilateur gcc, g++ pour windows).
- Créer un répertoire pour placer les 3 dossiers qui vont suivre.
- 1) Télécharger SFML v<sup>2.5.1</sup> pour mingw64. GCC 7.3.0 MinGW (SEH) - **64-bit**.
- 2) Télécharger et extraire le code source .zip de ImGui v<sup>1.82</sup>.  
<https://github.com/ocornut/imgui/releases/tag/v1.82>
- 3) Télécharger et extraire le code source .zip de ImGui-SFML v<sup>2.3</sup>.  
<https://github.com/eliasdaler/imgui-sfml/releases/tag/v2.3>
- Naviguer vers le dossier où vous voulez compiler ImGui-SFML.
- Exécuter la commande suivante dans le répertoire où vous voulez compiler imgui et imgui-sfml sachant que le dossier de compilation de SFML est le suivant :  
SFML-2.5.1/lib/cmake/SFML

```
cmake <dossier ImGui-SFML> -DIMGUI_DIR=<dossier ImGui> -DSFML_DIR=<dossier où SFML est compilé>
```

- Installer la bibliothèque imgui-sfml.

```
cmake --build . --target install
```

- Si la commande cmake n'est pas reconnue, vérifier qu'elle fait partie du PATH. Si c'est le cas, redémarrer ou se reconnecter sur la session pour mettre à jour.
- Ouvrir et modifier les lignes suivantes du fichier CMakeLists.txt pour correspondre avec votre répertoire SFML et celui où imgui-sfml a été installé. Exemple :

```
[...]
set(SFML_DIR "C:/SFML-2.5.1/lib/cmake/SFML")
[...]
set(ImGui-SFML_DIR "C:/Program Files (x86)/imgui_sfml/lib/cmake/ImGui-SFML")
```

- Créer un répertoire build au sein du répertoire de projet.
- Depuis ce répertoire, exécuter la commande suivante pour générer les fichiers de compilation pour le projet.

```
cmake -G "MinGW Makefiles" -S .. -B .
```

- Compiler le projet depuis le répertoire racine de ce dernier avec :

```
cmake --build ./build
```

- Copier les fichiers sfml-graphics-d-2.dll sfml-system-d-2.dll sfml-window-d-2.dll depuis SFML-2.5.1/bin vers le répertoire build du projet (à côté du fichier birdAI.exe).

## Code source

### utils.hpp

```
#ifndef UTILS_HPP
#define UTILS_HPP

#include <iostream>
#include <iomanip> // for std::cout formatting
#include <vector>

class Utils
{
public:
    static float random();
    static float random(const float& max);
    static float randrange(const float& min, const float& max);

    static void printArray(const std::vector<float>& array);

    static std::vector<float> withBias(const std::vector<float> array);

    static std::vector<float> slice(
        const std::vector<float>& array, const int& start, const int& end);
};

#endif
```

### utils.cpp

```
#include "NEFT/utils.hpp"

float Utils::random()
{
    return randrange(0.f, 1.f);
}

float Utils::random(const float& max)
{
    return randrange(0.f, max);
}

float Utils::randrange(const float& min, const float& max)
{
    float r = float(std::rand()) / RAND_MAX;
    return min + r * (max - min);
}

void Utils::printArray(const std::vector<float>& array)
{
    std::cout << std::fixed << std::setprecision(2);

    std::cout << "[";
    for(const float& elt: array)
    {
```

```

        std::cout << std::setw(5);
        std::cout << elt << ", ";
    }
    std::cout << "\b\b]" << std::endl;
}

std::vector<float> Utils::withBias(std::vector<float> array)
{
    // [4.5, -1.2] --> [4.5, -1.2, 1.0]
    array.push_back(1.f);
    return array;
}

std::vector<float> Utils::slice(
    const std::vector<float>& array, const int& start, const int& end)
{
    // returns the subarray from start to end

    auto first = array.begin() + start;
    auto last = array.begin() + end;
    std::vector<float> subArr(first, last);

    return subArr;
}

```

## matrix2.hpp

```

#include <iostream>
#include <iomanip> // for std::cout formatting
#include <vector>
#include "utils.hpp"

class Matrix2
{
private:
    int index(const int& i, const int& j) const;

protected:
    int m_rows;
    int m_cols;
    std::vector<float> m_array;

public:
    Matrix2(const int& rows, const int& cols);
    ~Matrix2();

    void randomize(const float& min, const float& max);

    void set(const int& i, const int& j, const float& value);
    float get(const int& i, const int& j) const;

    int size() const;
    std::vector<float> toArray() const;

    static std::vector<float> mult(

```

```

        const std::vector<float>& A, const Matrix2& B);

    friend std::ostream& operator<<(std::ostream& out, const Matrix2& self);
};

#endif

```

## matrix2.cpp

```

#include "NEFT/matrix2.hpp"

Matrix2::Matrix2(const int& rows, const int& cols): m_rows(rows), m_cols(cols)
{
    m_array = std::vector<float>(size());
}

Matrix2::~Matrix2()
{
}

int Matrix2::index(const int& i, const int& j) const
{
    // translates 2d array indexes into 1d array index

    //      j
    // |  0  1  2  3 |
    // i|  4  5  6  7 | <--> [0, 1, 2, 3; 4, 5, 6, 7; 8, 9, 10, 11]
    // |  8  9 10 11 |
    //                               ^
    //                          i * m_cols + j

    //
    // This allows for easy access to the matrix while keeping 1d array
    // for performances
    // ==> m_array[index(i, j)]

    return i * m_cols + j;
}

void Matrix2::randomize(const float& min, const float& max)
{
    for(float& elt: m_array)
    {
        elt = Utils::randrange(min, max);
    }
}

void Matrix2::set(const int& i, const int& j, const float& value)
{
    m_array[index(i, j)] = value;
}

float Matrix2::get(const int& i, const int& j) const
{
    return m_array[index(i, j)];
}

```

```

int Matrix2::size() const
{
    return m_rows * m_cols;
}

std::vector<float> Matrix2::toArray() const
{
    return m_array;
}

std::vector<float> Matrix2::mult(const std::vector<float>& A, const Matrix2& B)
{
    if((int) A.size() != B.m_rows)
    {
        std::cout << "Matrix2::Warning: Incompatible multiplication ";
        std::cout << A.size() << "x" << B.size() << std::endl;
        return std::vector<float>();
    }

    std::vector<float> C = std::vector<float>(B.m_cols);
    for(int j = 0; j < B.m_cols; j++)
    {
        float sum = 0;
        for(int k = 0; k < B.m_rows; k++)
        {
            sum += A[k] * B.get(k, j);
        }
        C[j] = sum;
    }

    return C;
}

std::ostream& operator<<(std::ostream& out, const Matrix2& self)
{
    out << std::fixed << std::setprecision(2);

    out << "[" << self.m_rows << "x" << self.m_cols << "]";
    for(int i = 0; i < self.m_rows; i++)
    {
        out << std::endl << "| ";
        for(int j = 0; j < self.m_cols; j++)
        {
            out << std::setw(5);
            out << self.get(i, j) << " ";
        }
        out << "|";
    }

    return out;
}

```

## layer.hpp

```
#ifndef LAYER_HPP
#define LAYER_HPP

#include "matrix2.hpp"

class Layer : public Matrix2
{
public:
    Layer(const int& inSize, const int& outSize);
    ~Layer();

    std::vector<float> getWeights() const;
    void setWeights(const std::vector<float>& weights);

    int getInputSize() const;
    int getOutputSize() const;
};

#endif
```

## layer.cpp

```
#include "NEFT/layer.hpp"

Layer::Layer(const int& inSize, const int& outSize): Matrix2(inSize, outSize)
{
}

Layer::~~Layer()
{
}

std::vector<float> Layer::getWeights() const
{
    return toArray();
}

void Layer::setWeights(const std::vector<float>& weights)
{
    if(size() != int(weights.size()))
    {
        std::cout << "Layer::Warning: set weights with different sizes";
        std::cout << std::endl;
        return;
    }

    for(int i = 0; i < size(); i++)
    {
        m_array[i] = weights[i];
    }
}
```

```

int Layer::getInputSize() const
{
    return m_rows;
}

int Layer::getOutputSize() const
{
    return m_cols;
}

```

#### dna.hpp

```

#ifndef DNA_HPP
#define DNA_HPP

#include <vector>
#include "utils.hpp"

class DNA
{
private:
    std::vector<float> m_genes;
public:
    DNA();
    ~DNA();

    void mutate(const float& mutationRate, const float& min, const float& max);
    std::vector<float> get() const;
    void set(const std::vector<float>& array);
};

#endif

```

#### dna.cpp

```

#include "NEFT/dna.hpp"

DNA::DNA()
{
    m_genes = std::vector<float>();
}

DNA::~DNA()
{
}

void DNA::mutate(const float& mutationRate, const float& min, const float& max)
{
    // pick a gene and randomize it with a probability of mutationRate

    for(float& elt: m_genes)
    {
        float r = Utils::random();
        if(r < mutationRate) elt = Utils::randrange(min, max);
    }
}

```

```

    }
}

std::vector<float> DNA::get() const
{
    return m_genes;
}

void DNA::set(const std::vector<float>& array)
{
    m_genes.clear();
    for(const float& gene: array)
    {
        m_genes.push_back(gene);
    }
}

```

### nn.hpp

```

#ifndef NN_HPP
#define NN_HPP

#include <cmath>
#include "layer.hpp"

class NN
{
private:
    int m_inputSize;
    int m_outputSize;

    std::vector<Layer> m_layers;

public:
    NN(const int& inputSize, const int& outputSize);
    ~NN();

    void addLayer(const int& inputSize);
    void randomize(const float& min, const float& max);

    std::vector<float> output(const std::vector<float>& input) const;

    std::vector<float> getWeights() const;
    void setWeights(const std::vector<float>& weights);

    int size() const;

    static void activate(std::vector<float>& arr);

    friend std::ostream& operator<<(std::ostream& out, const NN& self);
};

#endif

```



## nn.cpp

```
#include "NEFT/nn.hpp"

NN::NN(const int& inputSize, const int& outputSize):
    m_inputSize(inputSize), m_outputSize(outputSize)
{
    m_layers = std::vector<Layer>();
    m_layers.push_back(Layer(m_inputSize + 1, m_outputSize)); // set first layer
                                                         // + 1 for bias

    randomize(-1, 1);
}

NN::~~NN()
{
}

void NN::addLayer(const int& inputSize)
{
    // modify last layer to match with the next new layer
    int lastLayerIndex = m_layers.size() - 1;
    int lastInputSize = m_layers[lastLayerIndex].getInputSize();
    m_layers[lastLayerIndex] = Layer(lastInputSize, inputSize);

    // add new layer
    Layer newLayer(inputSize + 1, m_outputSize); // + 1 for the bias
    m_layers.push_back(newLayer);

    randomize(-1, 1);
}

void NN::randomize(const float& min, const float& max)
{
    for(Layer& layer: m_layers)
    {
        layer.randomize(min, max);
    }
}

std::vector<float> NN::output(const std::vector<float>& input) const
{
    // the array that will go through the NN
    std::vector<float> signal = input;

    for(const Layer& layer: m_layers)
    {
        // prepare next input with bias
        std::vector<float> biasedSignal = Utils::withBias(signal);

        // feed forward with the current layer
        signal = Matrix2::mult(biasedSignal, layer);
        // pass the output through the activation function
        NN::activate(signal);
    }

    return signal;
}
```

```

std::vector<float> NN::getWeights() const
{
    std::vector<float> weights = std::vector<float>();
    for(const Layer& layer: m_layers)
    {
        for(const float& weight: layer.getWeights())
        {
            weights.push_back(weight);
        }
    }

    return weights;
}

void NN::setWeights(const std::vector<float>& weights)
{
    if(size() != int(weights.size()))
    {
        std::cout << "NN::Warning: set weights with different sizes";
        std::cout << std::endl;
        return;
    }

    int blockIndex = 0;
    for(Layer& layer: m_layers)
    {
        // extract layer weights from full NN weights
        std::vector<float> layerBlock = Utils::slice(
            weights, blockIndex, blockIndex + layer.size());

        // then set the layer with the extracted sub array corresponding
        // to this layer
        layer.setWeights(layerBlock);
        blockIndex += layer.size();
    }
}

int NN::size() const
{
    int size = 0;
    for(const Layer& layer: m_layers)
    {
        size += layer.size();
    }

    return size;
}

void NN::activate(std::vector<float>& arr)
{
    // keep values normalized and coherent

    for(float& x: arr)
    {
        x = tanh(x); // tanh returns values between [-1, 1]
    }
}

```

```

    }
}

std::ostream& operator<<(std::ostream& out, const NN& self)
{
    out << "NN[in: " << self.m_inputSize << " +1 ";
    out << "==" << out: " << self.m_outputSize;
    out << "]" << std::endl;
    out << "|< in" << std::endl << "v" << std::endl;
    for(const Layer& layer: self.m_layers)
    {
        out << layer << std::endl;
        out << "v" << std::endl;
    }
    out << "|> out";

    return out;
}

```

### organism.hpp

```

#ifndef ORGANISM_HPP
#define ORGANISM_HPP

#include "nn.hpp"
#include "dna.hpp"

class Organism
{
private:
    float m_fitness;

protected:
    NN m_brain;
    DNA m_dna;

    bool m_isDead;

public:
    Organism(
        const int& brainInputSize,
        const std::vector<int>& hiddenLayersSizes,
        const int& braineOutputSize);
    virtual ~Organism();

    NN getBrain() const;
    std::vector<float> think(const std::vector<float>& input) const;

    DNA getDna() const;
    void setDna(const std::vector<float>& genes);

    float getFitness() const;
    void setFitness(const float& fitness);

    bool isDead() const;
}

```

```

void setDead(const bool& isDead);

static Organism* crossover(Organism* parentA, Organism* parentB);
void mutate(const float& mutRate, const float& min, const float& max);

friend std::ostream& operator<<(std::ostream& out, Organism* self);

/* usefull abstract functions */
virtual Organism* copy() const = 0;
};

#endif

```

### organism.cpp

```

#include "NEFT/organism.hpp"

Organism::Organism(
    const int& brainInputSize,
    const std::vector<int>& hiddenLayersSizes,
    const int& brainOutputSize):
    m_fitness(0.f),
    m_brain(brainInputSize, brainOutputSize),
    m_isDead(false)
{
    // add hidden layers
    for(const int& layerSize: hiddenLayersSizes)
    {
        m_brain.addLayer(layerSize);
    }
    m_dna.set(m_brain.getWeights()); // It is very important to update DNA
                                    // each time there is a modification
                                    // on the neural net and vice-versa.
                                    // It is on the responsibility of this
                                    // class to give appropriate set/get
                                    // and handle it correctly
}

Organism::~Organism()
{
}

NN Organism::getBrain() const
{
    return m_brain;
}

std::vector<float> Organism::think(const std::vector<float>& input) const
{
    return m_brain.output(input);
}

DNA Organism::getDna() const

```

```

{
    return m_dna;
}

void Organism::setDna(const std::vector<float>& genes)
{
    m_dna.set(genes);
    m_brain.setWeights(m_dna.get());
}

float Organism::getFitness() const
{
    return m_fitness;
}

void Organism::setFitness(const float& fitness)
{
    m_fitness = fitness;
}

bool Organism::isDead() const
{
    return m_isDead;
}

void Organism::setDead(const bool& isDead)
{
    m_isDead = isDead;
}

Organism* Organism::crossover(Organism* parentA, Organism* parentB)
{
    // use parentA as an exemple for the child.

    int dnaSize = parentA->m_dna.get().size();
    int separator = int(Utils::random(dnaSize));

    std::vector<float> crossedGenes = std::vector<float>();
    for(int i = 0; i < separator; i++)
    {
        // add parentA genes
        crossedGenes.push_back(parentA->m_dna.get()[i]);
    }
    for(int i = separator; i < dnaSize; i++)
    {
        // add parentB genes
        crossedGenes.push_back(parentB->getDna().get()[i]);
    }

    Organism* child(parentA);
    child->setDna(crossedGenes);
    // Set the new crossover genes: the only thing that changes between children
    // and parents.
    // We keep the topology but the neural net weights are updated in setDna()

    return child;
}

```

```

}

void Organism::mutate(const float& mutRate, const float& min, const float& max)
{
    m_dna.mutate(mutRate, min, max);
    m_brain.setWeights(m_dna.get());
}

std::ostream& operator<<(std::ostream& out, Organism* self)
{
    out << self->m_brain << std::endl;
    out << "\\DNA: ";
    Utils::printArray(self->m_dna.get());

    return out;
}

```

### population.hpp

```

#ifndef POPULATION_HPP
#define POPULATION_HPP

#include "organism.hpp"

class Population
{
private:
    std::vector<Organism*> m_array;
    int m_generation;
    float m_mutationRate;

    void calcFitness();
    std::vector<Organism*> getMatingPool() const;
    Organism* pickParent(const std::vector<Organism*>& matingPool) const;

public:
    Population();
    Population(const float& mutationRate);
    ~Population();

    void addOrganism(Organism* newOrganism);

    Organism* get(const int& i) const;
    std::vector<Organism*> get() const;

    float getMutationRate() const;

    bool everybodyDead() const;

    int getGeneration() const;

    Organism* getBest() const;

    void nextGeneration();
}

```

```
};
```

```
#endif
```

### population.cpp

```
#include "NEFT/population.hpp"
```

```
Population::Population(): m_generation(0), m_mutationRate(0.01f)
{
    m_array = std::vector<Organism*>();

    std::cout << "Population::>mutationRate set to " << m_mutationRate;
    std::cout << std::endl;
}
```

```
Population::Population(const float& mutationRate):
    m_generation(0), m_mutationRate(mutationRate)
{
    m_array = std::vector<Organism*>();

    std::cout << "Population::>mutationRate set to " << m_mutationRate;
    std::cout << std::endl;
}
```

```
Population::~~Population()
{
    for(Organism* organism: m_array)
    {
        delete organism;
        organism = nullptr;
    }
}
```

```
void Population::addOrganism(Organism* newOrganism)
{
    m_array.push_back(newOrganism);
}
```

```
Organism* Population::get(const int& i) const
{
    return m_array[i];
}
```

```
std::vector<Organism*> Population::get() const
{
    return m_array;
}
```

```
float Population::getMutationRate() const
{
    return m_mutationRate;
}
```

```

bool Population::everybodyDead() const
{
    for(Organism* organism: m_array)
    {
        if(!organism->isDead()) return false;
    }

    return true;
}

void Population::nextGeneration()
{
    calcFitness();
    std::vector<Organism*> matingPool = getMatingPool();

    /* begin reproduction */
    int size = (int) m_array.size();

    // delete each organism before new allocation
    for(Organism* organism: m_array)
    {
        delete organism;
        organism = nullptr;
    }

    for(int i = 0; i < size; i++)
    {
        Organism* parentA(pickParent(matingPool));
        Organism* parentB(pickParent(matingPool));

        Organism* child = Organism::crossover(parentA, parentB)->copy();
        m_array[i] = child;
        m_array[i]->mutate(m_mutationRate, -1.f, 1.f);

        delete parentA;
        parentA = nullptr;
        delete parentB;
        parentB = nullptr;
    }

    /* end of reproduction */

    // free mating pool
    for(Organism* organism: matingPool)
    {
        delete organism;
        organism = nullptr;
    }

    m_generation++;
}

void Population::calcFitness()
{
    // normalize the fitness according to the population

```



```

float fitSum = 0;
for(Organism* organism: m_array)
{
    organism->setFitness(0.01 * pow(organism->getFitness(), 4));
    fitSum += organism->getFitness();
}
for(Organism* organism: m_array)
{
    organism->setFitness(organism->getFitness() / (fitSum + 1));
    // avoid zero-division errors by adding 1
}
}

int Population::getGeneration() const
{
    return m_generation;
}

Organism* Population::getBest() const
{
    float bestFit = m_array[0]->getFitness();
    int bestIndex = 0;
    for(size_t i = 1; i < m_array.size(); i++)
    {
        if(m_array[i]->getFitness() > bestFit)
        {
            bestFit = m_array[i]->getFitness();
            bestIndex = i;
        }
    }

    return m_array[bestIndex];
}

std::vector<Organism*> Population::getMatingPool() const
{
    std::vector<Organism*> matingPool = std::vector<Organism*>();
    for(Organism* organism: m_array)
    {
        int orgPercent = int(organism->getFitness() * 100.f + 1);
        for(int i = 0; i < orgPercent; i++)
        {
            matingPool.push_back(organism->copy());
        }
    }

    return matingPool;
}

Organism* Population::pickParent(const std::vector<Organism*>& matingPool) const
{
    int r = int(Utils::random((float) matingPool.size() - 1));
    return matingPool[r]->copy();
}

```

### --- Flappy Bird Clone ---

#### bird.hpp

```
#ifndef BIRD_HPP
#define BIRD_HPP

#include <SFML/Graphics.hpp>
#include "NEFT/organism.hpp"
#include "wall_manager.hpp"

class Bird : public Organism, public sf::Transformable, public sf::Drawable
{
protected:
    sf::CircleShape m_shape;

    sf::Sprite m_sprite;

    sf::Vector2f m_linearVelocity;
    sf::Vector2f m_linearAcceleration;
    sf::Vector2f m_gravity;

    const float MASS = 1.f;
    const float SHAPE_RADIUS = 20.f;
    const sf::Color DEBUG_COLOR = sf::Color(255, 0, 0, 50);
    const sf::Color SPRITE_MODULATE = sf::Color(255, 255, 255, 170);
    const sf::Vector2f JUMP_FORCE = sf::Vector2f(0.f, -14.f);

    void applyPhysics();
    bool collide(
        const WallManager& wallManager, const sf::Vector2u& windowSize) const;

public:
    Bird(const sf::Vector2u& windowSize, const sf::Texture& texture);
    ~Bird();

    sf::Vector2f getGravity() const;
    void setGravity(const sf::Vector2f& gravity);

    void applyImpulse(const sf::Vector2f& force);
    void jump();

    sf::CircleShape getShape() const;

    void reset(const sf::Vector2u& windowSize);
    void process(
        const WallManager& wallManager, const sf::Vector2u& windowSize);

    /* methods relative to abstract classes that this object inherit from */
    virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
    virtual Bird* copy() const;
};

#endif
```

## bird.cpp

```
#include "bird.hpp"

Bird::Bird(const sf::Vector2u& windowSize, const sf::Texture& texture):
    Organism(3, {6, 4}, 1), m_gravity(sf::Vector2f(0.0f, 0.4f))
{
    reset(windowSize);
    m_shape.setRadius(SHAPE_RADIUS);
    // center the shape on bird's position
    m_shape.setOrigin(sf::Vector2f(m_shape.getRadius(), m_shape.getRadius()));
    m_shape.setFillColor(DEBUG_COLOR);

    m_sprite.setTexture(texture);
    m_sprite.setOrigin(32, 32);
    m_sprite.setColor(SPRITE_MODULATE);
}

Bird::~Bird()
{
}

sf::Vector2f Bird::getGravity() const
{
    return m_gravity;
}

void Bird::setGravity(const sf::Vector2f& gravity)
{
    m_gravity = gravity;
}

void Bird::applyImpulse(const sf::Vector2f& force)
{
    m_linearAcceleration += MASS * force;
}

void Bird::jump()
{
    applyImpulse(JUMP_FORCE);
}

bool Bird::collide(
    const WallManager& wallManager, const sf::Vector2u& windowSize) const
{
    // collision with walls
    for(Wall& wall: wallManager.getWalls())
    {
        if(wall.collide(getPosition(), m_shape.getRadius()))
        {
            return true;
        }
    }

    // collision with window boundaries
    if(getPosition().y < 0 || getPosition().y > windowSize.y)
```

```

    {
        return true;
    }

    return false;
}

sf::CircleShape Bird::getShape() const
{
    return m_shape;
}

void Bird::reset(const sf::Vector2u& windowSize)
{
    m_isDead = false;
    setPosition(sf::Vector2f(100.f, float(windowSize.y) / 2.f));
    m_linearAcceleration *= 0.f;
    m_linearVelocity *= 0.f;
    setFitness(0);
}

void Bird::process(
    const WallManager& wallManager, const sf::Vector2u& windowSize)
{
    if(!isDead())
    {
        if(collide(wallManager, windowSize))
        {
            setDead(true);
            //std::cout << "Bird::>bird died with fitness: " << getFitness();
            //std::cout << std::endl;
        }

        sf::Vector2f targetPos = wallManager.getTargetPos(
            getPosition(), SHAPE_RADIUS);
        // wait that bird has passed the wall:
        targetPos += sf::Vector2f(SHAPE_RADIUS, 0.f);

        std::vector<float> input = std::vector<float>(3);
        input[0] = targetPos.x / float(windowSize.x); // normalize
        input[1] = targetPos.y / float(windowSize.y);
        input[2] = 0.1f * m_linearVelocity.y;

        //Utils::printArray(input);
        bool shouldJump = think(input)[0] > 0.f;
        if(shouldJump) jump();

        applyPhysics();

        // rotation movement with falling speed
        bool heaviside = m_linearVelocity.y > 1.f;
        float angle = heaviside * pow(m_linearVelocity.y - 1.f, 2);
        if (angle > 90.f) angle = 90.f;
        setRotation(angle - 5.f);

        setFitness(getFitness() + 0.1f);
    }
}

```

```

    }
}

void Bird::applyPhysics()
{
    applyImpulse(m_gravity); // gravity
    m_linearVelocity += m_linearAcceleration;
    move(m_linearVelocity);
    m_linearAcceleration *= 0.f; // reset acceleration to keep impulse effect
}

void Bird::draw(sf::RenderTarget& target, sf::RenderStates states) const
{
    states.transform *= getTransform(); // make the shape follow bird's position

    // draw collision shape
    //target.draw(m_shape, states);

    target.draw(m_sprite, states);
}

Bird* Bird::copy() const
{
    // return a pointer to a copy of the current bird

    Bird* copyBird = new Bird(*this);
    return copyBird;
}

```

#### player.hpp

```

#include <SFML/Graphics.hpp>
#include "bird.hpp"

class Player : public Bird
{
public:
    Player(const sf::Vector2u& windowSize, const sf::Texture& texture);
    ~Player();

    virtual void process(
        const WallManager& wallManager, const sf::Vector2u& windowSize);

    virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
};

#endif

```

## player.cpp

```
#include "player.hpp"

Player::Player(const sf::Vector2u& windowSize, const sf::Texture& texture):
    Bird(windowSize, texture)
{
}

Player::~Player()
{
}

void Player::process(
    const WallManager& wallManager, const sf::Vector2u& windowSize)
{
    if(collide(wallManager, windowSize))
    {
        m_isDead = true;
    }

    sf::Vector2f target = wallManager.getTargetPos(getPosition(), SHAPE_RADIUS);

    applyPhysics();
}

void Player::draw(sf::RenderTarget& target, sf::RenderStates states) const
{
    states.transform *= getTransform(); // make the shape follow bird's position

    target.draw(m_shape, states);
}
```

## wall.hpp

```
#ifndef WALL_HPP
#define WALL_HPP

#include <SFML/Graphics.hpp>

class Wall : public sf::Transformable, public sf::Drawable
{
private:
    sf::Vector2f m_gapSize;

    sf::RectangleShape m_rectUp;
    sf::RectangleShape m_rectDown;

    void autoSize(const sf::Vector2u& windowSize);

    const sf::Color ACTIVE_COLOR = sf::Color(50, 210, 80);

public:
    Wall(const sf::Vector2u& windowSize);
    ~Wall();
}
```

```

sf::Vector2f getSize() const;
void setSize(const sf::Vector2f &size, const sf::Vector2u &windowSize);

void process(const float& speed);
bool collide(const sf::Vector2f& point, const float& radius) const;

virtual void draw(sf::RenderTarget& target, sf::RenderStates states) const;
};

#endif

```

## wall.cpp

```

#include "wall.hpp"

Wall::Wall(const sf::Vector2u& windowSize): m_gapSize(sf::Vector2f(70.f, 140.f))
{
    autoSize(windowSize);
}

Wall::~Wall()
{
}

void Wall::autoSize(const sf::Vector2u& windowSize)
{
    m_rectDown.setPosition(0.f, m_gapSize.y);
    m_rectDown.setSize(sf::Vector2f(m_gapSize.x, float(windowSize.y)));
    m_rectDown.setFillColor(ACTIVE_COLOR);
    m_rectUp.setPosition(0.f, 0.f);
    m_rectUp.setSize(sf::Vector2f(m_gapSize.x, -float(windowSize.y)));
    m_rectUp.setFillColor(ACTIVE_COLOR);
}

sf::Vector2f Wall::getSize() const
{
    return m_gapSize;
}

void Wall::setSize(const sf::Vector2f &size, const sf::Vector2u& windowSize)
{
    m_gapSize = size;
    autoSize(windowSize);
}

void Wall::process(const float& speed)
{
    move(-speed, 0.f);
}

bool Wall::collide(const sf::Vector2f& point, const float& radius) const
{
    // returns true if there is an intersection with the UP or the DOWN wall

    // check inside rect gap on X axis

```

```

    bool isInLeft = point.x + radius > getPosition().x;
    bool isInRight = point.x - radius < getPosition().x + m_gapSize.x;
    bool isInSideX = isInLeft && isInRight;

    // check inside rect gap on Y axis
    bool isInUp = point.y - radius > getPosition().y;
    bool isInDown = point.y + radius < getPosition().y + m_gapSize.y;
    bool isInSideY = isInUp && isInDown;

    return isInSideX && !isInSideY;
}

void Wall::draw(sf::RenderTarget& target, sf::RenderStates states) const
{
    states.transform *= getTransform(); // make shapes members follow self

    target.draw(m_rectUp, states);
    target.draw(m_rectDown, states);
}

```

#### wall\_manager.hpp

```

#ifndef WALL_MANAGER_HPP
#define WALL_MANAGER_HPP

#include <SFML/Graphics.hpp>
#include "NEFT/utils.hpp"
#include "wall.hpp"

class WallManager
{
private:
    float m_gapLength; // between each wall
    std::vector<Wall> m_walls;

    float m_wallSpeed;

public:
    WallManager(const int& size, const sf::Vector2u& windowSize);
    ~WallManager();

    float getWallSpeed() const;
    void setWallSpeed(const float& wallSpeed);

    void process(const sf::Vector2u& windowSize);
    std::vector<Wall> getWalls() const;
    void setWallSize(const sf::Vector2f& size, const sf::Vector2u& windowSize);
    void reset(const sf::Vector2u& windowSize);
    sf::Vector2f getTargetPos(
        const sf::Vector2f& from, const float& shapeRadius) const;
};

#endif

```



## wall\_manager.cpp

```
#include "wall_manager.hpp"

WallManager::WallManager(
    const int& size, const sf::Vector2u& windowSize): m_wallSpeed(2.f)
{
    m_walls = std::vector<Wall>();
    m_gapLength = float(windowSize.x) / float(size);

    // size + 1 to keep constant gap between walls sequences
    for(int i = 0; i < size + 1; i++)
    {
        m_walls.push_back(Wall(windowSize));
    }
    reset(windowSize);
}

WallManager::~WallManager()
{
}

float WallManager::getWallSpeed() const
{
    return m_wallSpeed;
}

void WallManager::setWallSpeed(const float& wallSpeed)
{
    m_wallSpeed = wallSpeed;
}

void WallManager::process(const sf::Vector2u& windowSize)
{
    for(Wall& wall: m_walls)
    {
        wall.process(m_wallSpeed);

        // reset new wall position when it goes outside of window
        if(wall.getPosition().x < -m_gapLength)
        {
            wall.setPosition(
                float(windowSize.x),
                Utils::randrange(0.2f * windowSize.y, 0.5f * windowSize.y));
        }
    }
}

std::vector<Wall> WallManager::getWalls() const
{
    return m_walls;
}

void WallManager::setWallSize(
    const sf::Vector2f& size, const sf::Vector2u& windowSize)
{
}
```

```

    for(Wall& wall: m_walls)
    {
        wall.setSize(size, windowSize);
    }
}

void WallManager::reset(const sf::Vector2u& windowSize)
{
    for(size_t i = 0; i < m_walls.size(); i++)
    {
        m_walls[i].setPosition(
            float(i + 2) * m_gapLength,
            Utils::randrange(0.2f * windowSize.y, 0.5f * windowSize.y));
    }
}

sf::Vector2f WallManager::getTargetPos(
    const sf::Vector2f& from, const float& shapeRadius) const
{
    // return the position of the closest wall to "from"

    sf::Vector2f closestPos = sf::Vector2f(4000.f, 0.f); // get an overshoot
    for(const Wall& wall: m_walls)
    {
        float dist = wall.getPosition().x - from.x; // to beginning of wall
        dist += wall.getSize().x; // aim at end of wall
        dist += shapeRadius; // wait for the shape to have passed the wall

        if(dist > 0 && dist < closestPos.x)
        {
            // aim at middle right
            closestPos = wall.getPosition() + sf::Vector2f(
                wall.getSize().x + shapeRadius, 0.5f * wall.getSize().y);
        }
    }

    return closestPos - from;
}

```

## main.cpp

```

/* Flappy bird clone */
/* Interactive demo for the NEFT Library */

#include <iostream>
#include <fstream> // to save data

#include <SFML/Graphics.hpp> // graphics
#include <SFML/Window.hpp>
#include <SFML/System.hpp>

#include "imgui.h" // graphical user interface
#include "imgui-SFML.h"

#include "NEFT/population.hpp" // neuroevolution

```

```

#include "wall_manager.hpp" // game
#include "player.hpp"

void handleEvents(
    sf::RenderWindow &window, Player &player, bool &paused, int &gameSpeed);

void process(
    sf::RenderWindow &window, WallManager &wallManager, Player &player,
    Population &pop, const int &gameSpeed, const bool &paused,
    const bool &playerMode, std::vector<std::vector<float>> &data);

void render(
    sf::RenderWindow &window, const WallManager &wallManager,
    const Player &player, const Population &pop, const bool &playerMode);

void addToFile(std::ofstream &f, const std::vector<float> &dataArr);

void applyGui(sf::RenderWindow & window, bool & paused, bool & playerMode,
    int & gameSpeed, WallManager & wallManager, Population & pop, Player & player,
    std::vector<std::vector<float>> & data);

int main(int argc, char **argv)
{
    /* SFML SETTINGS */
    sf::ContextSettings settings;
    settings.antiAliasingLevel = 8;
    sf::RenderWindow window(
        sf::VideoMode(480, 600), "Bird AI", sf::Style::Close, settings);
    window.setVerticalSyncEnabled(true);
    window.setKeyRepeatEnabled(false); // avoid jumping too fast with key echos

    // initialize imgui
    ImGui::SFML::Init(window);

    /* SETUP */
    srand((unsigned int)time(NULL)); // random seed

    bool paused = true;
    int gameSpeed = 1;
    const float maxAiScore = 5000.f;
    bool aiWon = false; // to pause when ai reached the max score

    // create a wallManager with 2 walls
    WallManager wallManager(2, window.getSize());

    // declare and load a unique texture for every Birds
    sf::Texture spriteTexture;
    spriteTexture.loadFromFile("res/polybird.png");

    Player player(window.getSize(), spriteTexture);
    bool playerMode = false; // player cycle or ai cycle

    // create the population and add custom instances of Organism inside it
    Population pop(0.01f); // mutation rate of 1%
    for (int i = 0; i < 100; i++)

```

```

{
    Bird *bird = new Bird(window.getSize(), spriteTexture);
    pop.addOrganism(bird);
}

// create a file to save data about AI Learning performances
bool saveData = false;
std::vector<std::vector<float>> data = std::vector<std::vector<float>>();

/* FRAME LOOP */
sf::Clock deltaClock; // needed by imgui
while (window.isOpen())
{
    handleEvents(window, player, paused, gameSpeed);

    ImGui::SFML::Update(window, deltaClock.restart());
    applyGui(window, paused, playerMode, gameSpeed, wallManager, pop,
            player, data);

    // pause the game when ai has won
    if (pop.getBest()->getFitness() > maxAiScore && !aiWon)
    {
        paused = true;
        aiWon = true;
        std::cout << "AI beat the game!" << std::endl;
    }

    process(
        window, wallManager, player, pop, gameSpeed, paused, playerMode,
        data);

    render(window, wallManager, player, pop, playerMode);
}

ImGui::SFML::Shutdown();
return 0;
}

void handleEvents(
    sf::RenderWindow &window, Player &player, bool &paused, int &gameSpeed)
{
    sf::Event event;
    while (window.pollEvent(event))
    {
        ImGui::SFML::ProcessEvent(event);

        switch (event.type)
        {
            // exit programm gracefully on window close event
            case sf::Event::Closed:
                window.close();
                break;

            // keyboard inputs
            case sf::Event::KeyPressed:
                switch (event.key.code)

```

```

        {
            case sf::Keyboard::Space:
                player.jump();
                break;

            case sf::Keyboard::P:
                paused = !paused;
                break;

            case sf::Keyboard::Up:
                gameSpeed++;
                break;

            case sf::Keyboard::Down:
                if (gameSpeed - 1 > 0)
                    gameSpeed--;
                break;

            default:
                break;
        }

    default:
        break;
    }
}

void applyGui()
{
}

void process(
    sf::RenderWindow &window, WallManager &wallManager, Player &player,
    Population &pop, const int &gameSpeed, const bool &paused,
    const bool &playerMode, std::vector<std::vector<float>> &data)
{
    if (!paused)
    {
        // process more at each frame:
        // with gameSpeed = 2, the game will run twice a frame.
        // This allows for accelerating the game and so the learning process
        for (int i = 0; i < gameSpeed; i++)
        {
            // process walls
            wallManager.process(window.getSize());

            // process player
            if (playerMode)
            {
                player.process(wallManager, window.getSize());

                // handle death
                if (player.isDead())
                {
                    // reset game

```

```

        player.reset(window.getSize());
        wallManager.reset(window.getSize());
    }
}

// process population
else
{
    // process birds
    for (Organism *organism : pop.get())
    {
        Bird *pBird = (Bird *)organism;
        pBird->process(wallManager, window.getSize());
    }

    // handle death
    if (pop.everybodyDead())
    {
        std::vector<float> dataRow = std::vector<float>();
        // add score before nextGeneration
        dataRow.push_back(pop.getBest()->getFitness());

        pop.nextGeneration();

        // add the fitness after normalization of fitnesses
        dataRow.push_back(pop.getBest()->getFitness());
        data.push_back(dataRow);

        // reset game
        wallManager.reset(window.getSize());
        for (Organism *organism : pop.get())
        {
            ((Bird *)organism)->reset(window.getSize());
        }
    }
}
}

}

void render(
    sf::RenderWindow &window, const WallManager &wallManager,
    const Player &player, const Population &pop, const bool &playerMode)
{
    window.clear(sf::Color(20, 120, 190));

    // draw walls
    for (Wall &wall : wallManager.getWalls())
    {
        window.draw(wall);
    }

    // draw player
    if (playerMode)
    {
        window.draw(player);
    }
}

```

```

    }

    // draw population instances
    else
    {
        for (Organism *organism : pop.get())
        {
            Bird *pBird = (Bird *)organism;
            if (!pBird->isDead())
                window.draw(*pBird);
        }
    }

    ImGui::SFML::Render(window);

    window.display();
}

void addToFile(std::ofstream &f, const std::vector<float> &dataArr)
{
    for (const float &data : dataArr)
    {
        f << std::to_string(data) << ",";
    }
    f << "\n";
}

void applyGui(sf::RenderWindow& window, bool& paused, bool& playerMode,
int& gameSpeed, WallManager& wallManager, Population& pop, Player& player,
std::vector<std::vector<float>>& data)
{
    ImGui::SetNextWindowPos(ImVec2(0, 0));
    ImVec2 guiSize = ImVec2(
        window.getSize().x,
        window.getSize().y / 4);
    ImGui::SetNextWindowSize(guiSize);
    ImGui::SetNextWindowCollapsed(true, ImGuiCond_Once);

    ImGui::Begin("Menu", NULL,
        ImGuiWindowFlags_NoResize |
        ImGuiWindowFlags_NoMove |
        ImGuiWindowFlags_NoSavedSettings);
    ImGui::BeginTabBar("Tabbar");

    if (ImGui::BeginTabItem("Game"))
    {
        if (ImGui::Button(paused ? "Run" : "Pause"))
        {
            paused = !paused;
        }
        ImGui::SameLine();
        if (ImGui::Button(
            playerMode ? "Train AI" : "Take commands"))
        {
            playerMode = !playerMode;
        }
    }
}

```

```

    ImGui::EndTabItem();
    ImGui::SliderInt("Game Speed", &gameSpeed, 1, 30);
}

if (ImGui::BeginTabItem("Environment"))
{
    sf::Vector2f wallSize = wallManager.getWalls()[0].getSize();
    if (ImGui::SliderFloat("Wall Height", &wallSize.y, 80.f, 160.f))
    {
        // update wall sizes
        wallManager.setWallSize(wallSize, window.getSize());
    }
    if (ImGui::SliderFloat("Wall Width", &wallSize.x, 10.f, 70.f))
    {
        wallManager.setWallSize(wallSize, window.getSize());
    }

    sf::Vector2f gravity = ((Bird *)pop.get()[0])->getGravity();
    if (ImGui::SliderFloat("Gravity", &gravity.y, 0.1f, 0.8f))
    {
        player.setGravity(gravity);
        for (Organism *bird : pop.get())
        {
            ((Bird *)bird)->setGravity(gravity);
        }
    }

    float wallSpeed = wallManager.getWallSpeed();
    if (ImGui::SliderFloat("Bird Speed", &wallSpeed, 0.5f, 5.f))
    {
        wallManager.setWallSpeed(wallSpeed);
    }

    ImGui::EndTabItem();
}

if (ImGui::BeginTabItem("Data"))
{
    std::string text = "Mutation Rate: ";
    text += std::to_string(int(100 * pop.getMutationRate()));
    text += "%";
    ImGui::Text(text.c_str());

    ImGui::SameLine(0.f, 40.f);
    text = "Population size: ";
    text += std::to_string(pop.get().size());
    ImGui::Text(text.c_str());

    text = "Generation: ";
    text += std::to_string(pop.getGeneration());
    ImGui::Text(text.c_str());

    text = "Best Score: ";
    text += std::to_string(int(pop.getBest()->getFitness()));
    ImGui::Text(text.c_str());
}

```



```

ImGui::Separator();

text = "Data entries: ";
text += std::to_string(data.size());
ImGui::Text(text.c_str());

if (ImGui::Button((data.size() == 0) ? "Need more data" : "Save"))
{
    if (data.size() > 0)
    {
        // add current data
        data.push_back({pop.getBest()->getFitness(), 1});

        // get date for file name
        time_t t = time(0); // get time now
        struct tm *now = localtime(&t);
        char buffer[80];
        strftime(buffer, 80, "%Y-%m-%d_%H%M%S.csv", now);

        // save data to file
        std::ofstream f;
        f.open(buffer);
        for (const std::vector<float> &row : data)
        {
            addToFile(f, row);
        }
        f.close();

        // clear data
        data.clear();
    }
}
ImGui::EndTabItem();
}

ImGui::EndTabBar();
ImGui::End();
}

```