PARTIE 1

Q1.1 : Pour éviter les duplications de code, il faut utiliser les méthodes déjà implémentées isColliding et isPointInside.

Q1.2 : Nous avons choisi des surcharges externes car nous pouvons le faire sans utiliser « friend », et les méthodes étant publiques il n'est pas nécessaire d'avoir directement accès aux attributs (auquel cas une méthode interne aurait été préférable).

Q1.3 : Toutes les méthodes publiques sont des « getters » et ne nécessitent donc pas d'arguments donc aucun à passer par référence constante (présence d'un seul size_t dans la méthode getImmuneGenes() qui n'a pas besoin d'être passé par référence constante car c'est une variable peu volumineuse).

Q1.3 : Tous les « getters » sont à déclarer comme const car ils ne modifient pas les attributs. Tandis que les méthodes privées (randomSex() et randomColor()) sont utilisées par le constructeur qui a pour but d'initialiser les attributs du génome et donc de les modifier).

PARTIE 2

Q2.1 : Seule la méthode drawOn est déclarée comme const car c'est la seule qui ne modifie pas les attributs de la classe Environnement, (addAnimal ajoute un animal au tableau, de même pour addTarget avec les cibles, reset vide les listes)

Q2.2 : Pour ne pas permettre la copie d'un Environnement, on peut aussi interdire la copie en supprimant explicitement le constructeur de copie comme « = delete ». Il est aussi judicieux de redéfinir l'opérateur =, pour empêcher la copie.

Q2.3 : Comme la vie des animaux est liée à l'environnement, la destruction de l'environnement entraîne aussi la destruction des animaux y vivant.

Q2.4 : On propose les prototypes :
Pour le calcul de la force lié à l'attraction : Vec2d calculForce() const;
La force est un vecteur d'où le retour Vec2d, la méthode calculForce() ne prend pas de paramètres puisqu'il a directement accès aux variables dont il a besoin (qui sont des attributs de la classe), et calculForce() est const car elle ne modifie pas les attributs de la classe.
Pour la mise à jour des données de déplacement : void updateDonnees(sf::Time dt);
(Par la suite on a rajouté la force en attribut car elle est différente en fonction de la présence de cible ou non).

Q2.5 : On remarque que le rapport entre les 3 types de décélérations est 0.3.
On déclare : enum Deceleration {forte=1, moyenne, faible} ;
et on ajoute une **constante DECELERATION** (qui vaut 0.3) dans le fichier Constants.hpp
Ainsi pour retrouver les valeurs 0.3, 0.6, 0.9 il suffit de multiplier la décélération par la constante : on le fait à travers la **méthode setDeceleration()** qui met à jour la valeur de l'**attribut deceleration** (que nous avons créé).
Comme la méthode update() qui fait appel à la méthode updateDonnees() qui fait appel à la méthode calcul_force() qui utilise une deceleration, n'utilise que le paramètre dt : pour éviter de mettre une deceleration en paramètre de toutes ses méthodes, on a décidé de créer un attribut que l'on peut modifier.

Q2.7 : On a rajouté dans la méthode drawOn de Environnement une boucle for sur la liste lesAnimaux qui fait appel à la méthode drawOn de la classe Animal.
On a aussi enlevé les instructions dans la méthode drawOn de la classe Animal qui permettait d'afficher les cibles (ce n'est pas à l'animal de dessiner les cibles, mais à l'environnement).

Q2.9 : vector<Vec2d> est le type de retour choisi : on y stocke les cibles de l'environnement visibles par l'animal. On a choisi vector au lieu de list afin de pouvoir choisir une cible grâce à son indice.

Q2.10 : On invoque dans la méthode update() de l'environnement la méthode update() d'Animal pour chaque animal de la liste lesAnimaux.

PARTIE 3

Q3.1 : Toutes les méthodes suivantes sont à déclarer comme virtuelles pures dans Animal tandis qu'il faut ajouter override à celles-ci dans Wolf/Sheep car il est nécessaire de les définir pour chaque animal.

```
double getStandardMaxSpeed()  const ;
double getMass()  const;
double getRadius()  const;
double getViewRange()  const;
double getViewDistance()  const;
double getRandomWalkRadius()  const;
double getRandomWalkDistance()  const;
double getRandomWalkJitter()  const;
```

Q3.2 : On doit retoucher dans le fichier app.json la valeur de l'énergie initiale qui se trouve à cet endroit :

```
"energy":{
        "initial": « valeur »,
```

Pour que cette nouvelle valeur soit transmise au programme il est nécessaire de recharger ce fichier par le biais de la touche 'L' qui va utiliser la nouvelle valeur entrée.

Q.getEntitiesInSightForAnimal : Le tableau retourné par la méthode contient des pointeurs sur des entités que l'on pourrait modifier à l'endroit qui fait appel à cette méthode (qui récupère ce tableau de pointeurs).

Q3.3 : on doit introduire la méthode update() qui va changer la taille de l'herbe, et il faut aussi penser à définir la méthode drawOn de façon polymorphique pour que la croissance soit visible graphiquement.

Q3.4 : Tester le type des objets peut être potentiellement nocif car ce mode de fonctionnement n'est pas du tout robuste face au changement et à la mise a jour de code. Ce qui n'en fait pas un bon mode de fonctionnement

Q3.5 : Les informations de debugging que nous voulons afficher concernent les animaux, c'est donc au niveau de Animal que nous plaçons l'affichage.
Si les massifs d'herbes nécessitaient aussi un affichage de debugging, nous aurions défini une méthode virtuelle pure dans LivingEntity que nous aurions redéfini dans Animal et dans Grass.

Q3.6 : On fait appel à la méthode update() de LivingEntity dans les méthodes update() de Grass et Animal (en réalité dans updateDonnees() dans Animal) afin de faire vieillir les entités.
On définit la méthode getLongevity() en virtuelle pure dans LivingEntity que l'on redéfinit pour les sous classes Sheep, Wolf, et Grass. Si on veut changer la longévité, on a juste à modifier les valeurs du fichier.json .

Q3.7 : Lorsque qu'une entité est morte on fait pointer le pointeur qui pointait vers celle ci vers « nullptr », puis on parcourt la liste de pointeurs dans Environnement et on supprime ceux qui pointent vers « nullptr ».

Q3.8 : Nous avons rajouté une condition avant le switch, si le niveau de vie est inférieur à un certain seuil, la valeur de vitesse maximum retournée sera plus faible.

Q3.9 : L'attribut position des entités organiques se trouve au niveau de LivingEntity, on redéfinit donc la méthode getCenter() (de Obstacle) au niveau de LivingEntity : on retourne getPosition().

Q3.10 : On pourrait aussi considérer qu'une LivingEntity n'est pas un Obstacle mais a un obstacle. L'avantage serait que cela pourrait permettre une représentation plus fidèle des animaux en pouvant adapter la forme en fonction du type de l'entité.
L'inconvénient serait qu'il serait plus difficile à mettre en place et nécessiterait de créer plusieurs classes obstacles avec différentes formes.

Q3.11 :
On utilise le double dispatch de la même manière que pour la méthode eatable et eatableBy :
3 méthodes canMate :
bool canMate(Wolf const* wolf) const ;
bool canMate(Sheep const* sheep) const ;
bool canMate(Grass const* grass) const ;
qui sont appelées par la méthode matable(LivingEntity const*)
Dans la méthode meet ; on vérifie la compatibilité de 2 individus grâce à la méthode matable qu'il faut utiliser dans les 2 sens (entité courante → cible / cible → entité courante)
(Plutôt que de le faire dans meet, nous l'avons fait en amont dans notre méthode laPlusProche qui sélectionne une entité potentiellement partenaire la plus proche, sélectionnée avec ces tests de matable).

Q3.12 : On a créé un attribut (classe Animal) : sf::Time gestationTime
Il est initialisé lors de la rencontre et décrémenté tant que la femelle est enceinte. Dès qu'il est inférieur ou égal à 0, on déclenche la naissance des bébés (passage en GIVING_BIRTH) puis on initialise le temps de pause après l'accouchement.

Q3.13 : Nous avons défini une méthode giveBirth() en virtuelle pure dans Animal que nous avons redéfinie dans Wolf et Sheep de façon polymorphique.

Q3.14 : Le nombre de bébés est enregistré sous la forme d'un attribut (de type unsigned int).

Q3.15 : Nous avons choisi de créer un attribut pour mémoriser les prédateurs potentiels. Si il y a des prédateurs, on enregistre ce tableau de prédateurs potentiels en tant qu'attribut pour ne pas avoir à refaire appel à la méthode lesPlusProches pour les ennemis dans le calcul de la force en présence de prédateurs.
Nous avons hésité car nous trouvions que les deux solutions étaient quasiment équivalentes.

Q4.1 : Un tableau associatif est préférable car l'ensemble des graphes et l'ensemble des libellés sont liés, quand on retourne un graphe il faut aussi retourner le libellé qui lui est associé, ce que permet l'utilisation d'un tableau associatif.

Q4.2 : Nous avons mis en place un attribut de classe (un unordered_map avec comme clé un string (grass, sheep ou wolf) lié a un entier qui représente le nombre d'entité de ce type dans la classe LivingEntity qui décompte le nombre de d'entité de chaque sous classe de LivingEntity.
Nous avons également ajouté une méthode getNbLivingEntities() qui prends en argument un string et qui retourne le nombre d'entité de ce type.
Nous y faisons appelle dans une méthode fetchData() dans Environnement qui va

retourner les nouvelles données que l'on va utiliser dans Stats (update() plus précisement).

Q4.3 : Nous avons répertorié la correspondance indice-chef de troupeau dans un nouvel attribut d'environnement puisque c'est lui qui contient tous les animaux:

```
std::unordered_map<int, LivingEntity*> herdsLeaders;
```

A chauqe numéro de troupeau correspond un leader.
On a choisi une unordered_map car l'ordre des troupeaux n'est pas important puisque ils sont identifiés par numHerd.

Q4.4 : Nous avons implémenté une fonction getLeader qui retourne le chef de troupeau du troupeau associé au numéro entré en paramètre.
Cette même fonction retourne « nullptr » si le numero de troupeau n'a pas de leader, elle permet donc aussi de savoir si un animal à un chef.
Pour savoir si un animal est un chef nous avons mis en place un attribut supplémentaire dans la classe Sheep « leader » qui est un booléen qui vaut true si l'animal est un chef et false si il n'est pas un chef.
Nous avons également codé une méthode isFree() dans Sheep qui retourne true si l'animal est libre de ses déplacements et false sinon.

Q4.5 : Nous avons créé une fonction setNewLeader qui est appelé dans le destructeur de Sheep (→ quand il meurt), qui change le leader actuel qui est en train de mourir en un nouveau (nous avons fait le choix du plus vieux).

Q4.6 : Il faudrait ajouter une touche qui permettrait de masquer l'espace dédié à l'affichage de graph afin de passer en mode non-graphique.
Grâce aux compteurs statiques d'animaux qu'on a déjà créés, à un instant, on pourrait enregistrer le nombre de moutons et de loups dans un attribut.
Grâce à un compteur mis dans dt on pourrait compter le nombre de cycles de simulation, une fois le compteur atteint le nombre de cycle voulu, on afficherait sur le terminal "Avec x moutons et y loups au départ (valeurs de l'attribut de départ), on obtient au bout de n cycles de simulation ... population de loup et .. population de moutons (avec les valeurs à l'instant où le compteur à atteint n)".
On pourrait mettre un place un fichier avec tous les loups, les moutons et les herbes à placer sur le monde que lirait une méthode de environnement et les placerait suivie de la simulation (avec un temps accéléré pour ne pas devoir attendre)

Q5.1 : Nous avons rajouté un argument à FetchData qui indique si le graph dont il doit donner les datas est actif ou non.
Si le graph est actif fetchData retourne des données, sinon il retourne une map vide.

Q5.2 :
on a introduit des méthodes getters dans ImmuneSystem :
getAdaScore() -> pour score adaptatif

getGlobalScore() -> pour score global
getActivationLevel()
getQuantityInfectiousAgents() dans virus qui est utilisé par ImmuneSystem :
getQuantityVirus()
on a également introduit une méthode getAnimalData() dans animal qui regroupe toutes ces données dans un array<double,6>
→ c'est cette méthode qui est utilisée dans fetchData()

Q5.3 : Nous avons rajouté un attribut de classe à la classe ImmuneSystem :

```
static unsigned int nbInfected;
```

Ainsi qu'un getter qui retourne la valeur de cet attribut.
Nous avons aussi mis aux endroits nécessaires, des incrémentations ou des décrémentations de cet attribut (lors de l'infection, du combat réussi contre cette infection ou à la mort de l'animal).

Q5.4 :
On a introduit des méthodes infect(...) en virtuelle pure dans LivingEntity prenant soit un loup, un mouton ou une grass en argument (double dispatch) permettant de tester si les animaux concernés sont du même type.
On a introduit une méthode virtuelle pure isInfectedBy(LivingEntity other*) dans LivingEntity qui permet la transmission du virus.
On a introduit une méthode canInfect(Sheep*/Wolf*) dans Sheep et Wolf respectivement, permettant de tester si toutes les conditions nécessaires pour qu'un animal transmette son virus sont réunies. (Cette méthode est réutilisée par infect(..) quand cela est nécessaire, exemple interaction de 2 moutons) Cette méthode permet juste de modulariser le code.

On a remanié notre conception de nos méthodes de mise à jour dans Animal, désormais :
la méthode updateClosest() parcourt une seule fois l'ensemble de toutes les entités de Environnement et enregistre dans les attributs correspondants :
- les ennemis les plus proches
- le partenaire le plus proche
- la nourriture la plus proche
- les voisins, afin de pourvoir calculer la force d'évitement

Pour l'infection, on regarde les entités qui sont dans "infection range" et on les infecte.