Groupe 44 : Elaissaoui Souhail et Honsali Fatim-Zahra.

REPONSES :

PARTIE 1 :

Q1.1 : Pour réutiliser l'algorithme de « clamping » dans la classe « Collider » sans duplication du code il faut créer une méthode « clamping () » dans la classe « Collider ».
Q1.2 : Pour éviter d'avoir un code trop répétitif dans la mise en oeuvre de l'algorithme « clamping » il faut imbriquer deux boucles qui font passer toutes les 9 possibilités du vecteur « to ».
Q1.3 : Les arguments de méthodes passés par références constantes sont :

- Le Vec2d de clamping.
- Le Vec2d de distanceTo(), et le Collider pris en argument par la surcharge de distanceTo ; même chose pour directionTo.
- Le Vec2d de move et donc de l'operateur +=.
- Le Collider pris en argument par le constructeur de copie.
- Le Collider pris en argument par le constructeur de recopie.

Q1.4 : Les méthodes déclarés comme const sont :

- directionTo() et sa surcharge.
- getRadius().
- getPosition().
- distanceTo() et sa surcharge.

Q1.5 : Pour coder les trois opérateurs précédemment décrits en évitant toute duplication de code dans le fichier Collider.cpp il faut appeler les fonctions correspondantes dans les définitions de ces operateurs.
Q1.6 : Surcharge des operateurs :

- Les opérateurs comparant l'argument a l'instance courante doivent être des surcharge interne : > et | .
- L'opérateur d'affichage doit être une surcharge externe, mais être passé en friend de Collider.

Q1.7 : Les arguments passés en référence constante sont :

- Le Collider de isColliderInside et de IsColliding.

Þ (pareil pour les opérateurs > et |)

- Le Vec2d de isPointInside et de l'opérateur >.
- Le Collider pris par l'operateur d'affichage.

Q1.8 : Les méthodes déclarées comme const sont :
isColliderInside, Iscolliding, isPointInside, opérateurs > et |

PARTIE 2 :

Q2.1 : La taille de cells_ doit être nbcells_ * nbcells_ car la coordonnée maximale du tableau doit être x+y*nbcells_ avec x et y = nbcells_ -1 .

Q2.2 : Pour faciliter l'accès aux paramètres du fichier de configuration on peut utiliser une variable déclarée comme suit :
auto const& cfg =getAppConfig()["simulation"]["world"]

Q2.3 : Les méthodes invoquées pour réaliser ces initialisations on appelle reloadConfig() puis reloadCacheStructure() puis updateCache().

Q2.4 : L'avantage d'utiliser ces méthodes plutôt que d'associer au flot de lecture est de garder une certaine flexibilité : on peut changer le fichier duquel on charge le monde, directement dans le fichier de configuration appX.json.

Q2.5: La méthode loadFromFile pour mettre à jour les attributs nécessaires au rendu graphique du terrain après l'initialisation de cells_ depuis un fichier doit appeler les fonctions ReloadCacheStructure et UpdateCache afin de mettre à jour les attributs nécessaires au rendu graphique.

Q2.6: Pour que le tableau de seeds_ soit modifiable, on choisit un vector pour le modéliser.

PARTIE 3 :

Q3.1: Pour coder le corps des méthodes Env::drawOn, Env::update et Env::reset on appelle les fonctions correspondantes de son attribut world_.

Q3.2: Codage : la fonction loadWorldFromFile() va appeler la fonction loadFromFile de son attribut world_. Nous allons l'utiliser dans le constructeur de la classe Env.
Le fichier qui sera utilisé concrètement est le fichier étiqueté ["simulation"]["world"]["file"] dans le fichier de configuration, il figure dans le dossier « res ».

Q3.3: Nous avons crée une méthode de world : « humidify » qui prend en argument les coordonnées de la cellule qui va humidifier son entourage en appliquant l'algorithme proposé.
La méthode « humidify » va être appelée dans la méthode step de World, à chaque fois qu'une graine d'eau transmet son type a une autre cellule.

Q3.4:
Pour éviter un nouveau parcours de toutes les cellules (x,y) de humidityVertexes_ :
Dans updateCache, on commence par calculer les valeurs d'humidité en dehors de la boucle « for », itérant sur toutes les cellules.
Ensuite, dans cette même boucle for qui définit la transparence des sf::Quad des waterVertexes_, rockVertexes_ etc , on ajoute le code nécessaire pour définir les niveaux
de bleu de humidityVertexe.

Q3.5: Pour faire des tests de collision avec la classe Flower, celle-ci devra hériter de la classe Collider.

Q3.6: Pour s'assurer que le choix de la texture se fasse une fois pour toute lors de la création de la fleur on ajoute un attribut 'indice_' a la classe Flower qui sera initialisé dans le constructeur de Flower a l'aide de la fonction uniform().

Q3.7: La collection de fleur sera une collection hétérogène (tableau de pointeur sur des Flower) afin de pouvoir le remplir avec des objet de classe Flower ou des objets qui héritent de cette dernière classe.

Q3.8: Le destructeur de Env doit appeler le destructeur de chacune de ses fleurs car c'est dans la classe Env qu'ils ont été alloue dynamiquement.

Q3.9: Pour repartir d'un nouveau terrain et de supprimer les fleurs on vide le tableau de fleurs en appelant un 'Flowers_.clear(); dans la fonction « reset ()».

Q3.10: La méthode existante qui doit être modifiée pour permettre le dessin des sources de fleurs nouvellement ajoutées est la méthode DrawOn de Env :
On y itère sur la collection de fleur en appelant, pour chacune, sa fonction DrawOn (de la classe Flower).

Q3.11: Nous avons placé la méthode getHumidity dans la classe Env, celle ci fait appelle à une autre fonction getHumidity définie dans la classe World.

Q3.12: Pour que l'évolution des fleurs devient visible lors de l'exécution du test graphique : dans la fonction update de Env on itere sur le tableau de fleurs, en appelant, pour chacune, sa fonction update. (Comme c'est un tableau de vecteur , on utilise la syntaxe : Flowers_[i]->update();).
Pour s'assurer de n'iterer que sur l'ensemble initial de fleurs, on crée au debut de update une copie du tableau de fleurs. On iterera sur ce dernier.

Q3.13: On modifie la méthode update de la classe Env pour faire en sorte que les fleurs de notre simulation disparaissent si leur quantité de pollen devient nulle
: pendant le parcourt de la collection de fleurs, on vérifie si celle ci a une quantité de pollen est nulle; le cas échéant, on supprime cette fleur avec un delete. A la fin de la boucle d'itération, on vide le tableau de fleurs, de tous ses pointeurs nuls.

Q3.14 : On pourrait faire hériter de Drawable et Updatable la classe Flower, car elle contient les méthodes DrawOn et update, et de plus elle n’est utilisée qu’a travers des pointeurs dans notre programme. (Ceci est important car les classes Drawable et Updatable sont de type Abstract Class car contiennent des fonctions virtuelles pure.

Q3.15 : On supprime le constructeur de copie par défaut de World et de Env pour éviter la copie d’élément trop volumineux.

Q3.16: Dans la fonction update de la classe Env on appelle la fonction update de son

PARTIE 4 :

Q4.1 : Les abeilles seront de diffèrent type (butineuse, éclaireuses..) donc il est important de travailler avec une collection hétérogène pour pouvoir utiliser des sous classes.

Q4.2 : La classe Hive étant updatable et drawable, elle hérite alors des classes Updatable et Drawable. Etant un objet dans le monde torique qui pourra faire des collisions, elle héritera également de Collider.

Q4.3: On supprime le constructeur de copie et l'opérateur =, afin d'éviter une copie inutile d'élément trop volumineux.

Q4.4: Il faut supprimer tous les pointeurs sur les abeilles de cette ruche lors de sa destruction

Q4.5: Non, on ne peut pas permettre aux ruches de survivre à la destruction de l'environnement, il faut les supprimer dans le destructeur.

Q4.6 : Pour empêcher que les fleurs puissent pousser ou être plantées dans un endroit occupé par une ruche, on ajoute une fonction bool isGrowable(Vec2d) et on vérifie que le collider de type flower qui va être crée n'est pas en collision avec une des ruches lors de l'ajout a l'environnement (dans addFlowerAt)

Q4.7 : On appelle la méthode DrawOn de chaque ruche du tableau de ruches dans les update de Env. De plus, on delete chaque ruche dans le destructeur de Env.

Q4.8 : La classe Bee va hériter de Drawable, Updatable et Collider.
Elle a comme attribut un pointeur sur une ruche, un vec2d pour sa vitesse, un double pour son niveau d'énergie. Quant aux méthodes elle aura drawOn, update, un constructeur, destructeur, des fonctions « bool isDead() », et « void move(sf :Time dt) »

Q4.9: La méthode getConfig doit être virtuelle pour pouvoir être redéfinie dans les sous classes de Bee de façon polymorphique.

Q4.10: Pour retrouver la texture de l'abeille de façon polymorphique dans les sous classes de Bee, on dessine la texture à partir de la méthode virtuelle getConfig().

Q4.11: Dans le update de Hive, on appelle le update des abeilles de cette ruche pour vérifier si elles sont mortes ou non.

PARTIE 5 :

Q5.1 : Les fichiers de texture associés à chacune des abeilles dans les appX.json correspondent à la forme avec laquelle on modelise chaque abeille.

Q5.2 : On met la méthode getConfig() de Bee en virtuelle pure pour que Bee ne soit plus instanciable en tant que telle. Dans le drawOn de Bee, on dessine la texture associée au getConfig() pour déssiner l'abeille de façon polymorphique, sans redéfinir la méthode drawOn dans les sous classes de Bee.

Q5.3 : Connaissant les états possibles de chaque abeille, nous allons appeler le constructeur de CFSM dans le constructeur de Bee, en l'initialisant avec l'ensemble des états possible de notre abeille, qu'on passera en argument au premier constructeur.

Q5.4 : Pour que l'abeille puisse oublier une fleur, on ajoute un attribut de type pointeur sur une fleur, qui prendra comme valeur par défaut nullptr afin de ne pas avoir à changer le constructeur.

Q5.5 : On doit absolument redéfinir getConfig et drawon dans les sous classes de Bee pour que celles-ci soient instanciables

Q5.6 : La cible courante de l'abeille sera modélisée par un pointeur sur un vec2d.
Les modes de déplacement seront modélisés par un enum, et le mode courant de l'abeille sera un attribut du type de ce enum.

Q5.7 Les traitements polymorphiques peuvent être directement mis en œuvre dans Bee car on utilise la méthode polymorphique getConfig().

Q5.8 : Les états utilisés pour modéliser le comportement des éclaireuses sont maintenant complètement spécifiés, on initialise donc le constructeur avec l'ensemble de ces états que le CFSM va utiliser.

Q5.9 : L'éclaireuse doit "vider" sa mémoire lors de la transition de l'état « dans la ruche » à l'état « recherche de fleur ». Elle doit fixer sa ruche comme cible lors de la transition de l'état « recherche de fleur » a l'état « retour a la ruche ».

Q5.10 : Afin de mettre en oeuvre la sélection d'une fleur à mémoriser par l'éclaireuse, on test, durant son vol aléatoire, si un collider fictif (de la taille de l'abeille + son rayon de

vision) est en collision avec une fleur de l'environnement avec la méthode Env::getCollidingFlower.

Q5.11 : Dans la méthode onEnterState, la butineuse doit

- oublier qu'elle cible une fleur lorsqu'elle passe dans l'état « retour a la ruche » ;
- se fixer pour cible la fleur qu'elle a en mémoire lorsqu'elle passe dans l'état « vers une fleur » ;
- se fixer pour cible sa propre ruche lorsqu'elle passe dans l'état « retour a la ruche » ;
- passer en mode de déplacement aléatoire : a aucun transition d'état ;
- passer en mode de déplacement ciblé lorsqu'elle passe dans l'état « retour a la ruche » et dans l'état « vers une fleur ».

Question (ICC) : On commence par répertorier les abeilles à l'intérieur de la ruche afin de faire interagir seulement les abeilles qui y sont. Cela nous évite de faire interagir toutes les abeilles de la ruche entre elle et de vérifier à chaque fois si elles sont bien toutes deux dans la ruche. On réduit ainsi la complexité plutôt en moyenne, car ce traitement est avantageux dans tous les cas, pas seulement au pire cas.

Q5.12 : les tests de type ont mauvaise réputation car ils impliquent une désencapsulation ;

Q5.13 : Le traitement de l'ajout des abeilles se fera dans le update de Hive.