

REPONSES

Semaine 4 (11/03 - 17/03)

[Question Q1.1] L'algorithme de «clamping» de la position devra être appliqué dans d'autres contextes que celui de la construction. Comment le coder pour que sa réutilisation dans une autre méthode de la classe `CircularCollider` n'implique pas de duplication de code ?

On pourrait faire que l'algorithme de "clamping" soit une méthode à part entière (dans la classe `CircularCollider` mais en dehors du constructeur) et l'appeler dans le constructeur afin que les valeurs fournies en paramètre d'une instance `CircularCollider` soient interprétées dans le monde torique.

Cette méthode sera privée car l'utilisateur ne doit pas la manipuler et n'a pas besoin de savoir comment le programmeur s'y prend pour que les `CircularCollider` restent dans le monde torique et ne disparaissent pas de l'écran.

[Question Q1.2] Si les définitions par défaut des constructeurs de copie et de l'opérateur de (re)copie nous conviennent pourquoi est-il préférable de les coder explicitement ?

Il est préférable de coder explicitement les constructeurs afin d'être certain que les bons attributs aient été copiés aux bons endroits pour éviter qu'il y ait des confusions lorsque le programme copierait automatiquement les valeurs des attributs.

[Question Q1.3] Comment utiliser des boucles pour éviter d'avoir un code trop répétitif dans la mise en oeuvre de cet algorithme ?

Nous avons décidé d'abord de ne seulement changer le `Vec2d to` neuf fois pour pouvoir calculer les différentes distances possibles qu'il y aurait avec `to` et `from` (position du `CircularCollider`).

Pour modifier `to`, nous avons imbriqué deux tableaux l'un dans l'autre: `tabX` et `tabY`.

Ensuite nous comparons la distance du vecteur `direction` et du vecteur `tmp_vec` pour pouvoir connaître la plus courte distance (grâce à la méthode de `Vec2d length()`) parcouru par le vecteur entre le point `from` et les points `to`. Au lieu d'écrire cette comparaison neuf fois dans notre code, ce qui serait redondant nous avons créé une fonction `checkShortestDistance`.

[Question Q1.4] Quels arguments de méthodes, parmi celles qu'il vous a été demandé de coder ci-dessus, vous semble-t-il judicieux de passer par référence constante ?

Tous les arguments des méthodes: `directionTo`, `distanceTo`, `move` et de `l'operator+=` doivent être passés par référence constante car dans nos méthodes ces valeurs ne changent pas. Seulement dans la méthode de `checkShortestDistance`, `tmp_dist` et `tmp_vec` sont passés par référence constante et `distance` et `direction` sont passés par référence (et non pas par valeur).

[Question Q1.5] Quelles méthodes parmi celles que l'on vous a demandé de coder ci-dessus vous semble-t-il judicieux de déclarer comme `const` ?

Toutes les méthodes peuvent être déclarées comme `const` sauf pour la méthode de `move` et `l'operator+=` donc au final toutes les méthodes `directionTo` et `distanceTo`.

[Question Q1.6] Comment coder les trois opérateurs précédemment décrits en évitant toute duplication de code dans le fichier `CircularCollider.cpp` ?

Pour éviter toute duplication, il suffit d'utiliser les trois méthodes précédemment programmées (`isCircularColliderInside` pour `l'operator>`, `isColliding` pour `l'operator|`, et `isPointInside` pour `l'operator>`) que l'on a mis en privé car elles doivent seulement être accessibles au programmeur afin de programmer les trois opérateurs.

[Question Q1.7] Quelle surcharge choisissez-vous pour les opérateurs qu'il vous a été demandé de coder (interne ou externe) et comment justifiez-vous ce choix ?

Les opérateurs `operator>`, `operator|`, `operator>`, sont codés en surcharge interne puisque chacun a pour but de connaître d'une manière la proximité d'un autre `CircularCollider` ou un point quelconque a une instance `CircularCollider` courante.

L'opérateur `operator<<` est aussi codé en interne puisqu'il affiche la position et le rayon d'une instance courante `CircularCollider` sur le terminal (passé par référence en paramètre, car modifié).

[Question Q1.8] Quels arguments de méthodes, parmi celles qu'il vous a été demandé de coder ci-dessus, vous semble-t-il judicieux de passer par référence constante ?

Les `Vec2d` et les `CircularCollider` non modifiés dans les méthodes sont passés par référence constante puisqu'ils sont plutôt lourds au niveau de la mémoire. Ceci évite de les

modifier et de faire leur copie. C'est le cas des deux `operator<` ainsi que de l'`operator|` et les trois méthodes qui les utilisent (`isCircularColliderInside`, `isPointInside`, `isColliding`).

[Question Q1.9] Quelles méthodes parmi celles que l'on vous a demandé de coder ci-dessus vous semble-t-il judicieux de déclarer comme `const` ?

Toutes les méthodes peuvent être déclarées comme `const` car elles ne changent pas l'objet courant en tant que tel, à part l'`operator<<` car cela change le ostream.

Semaines 5-6(18/03-30/03)

[Question Q2.1] Quelles méthodes vous semble t-il judicieux de déclarer comme `const` ?

La méthode `draw` est déclarée comme `const` puisqu'elles ne modifient pas l'environnement courant mais dessinent des cibles dans la fenêtre (déjà incluses dans l'environnement courant lors de sa construction).

[Question Q2.2] On souhaite ne pas permettre la copie d'un `Environment`. Quelle solution proposez-vous pour satisfaire cette contrainte ?

Il suffit de supprimer le constructeur `Environment(Environment const& autre)` de copie par défaut en utilisant la commande `"=delete"`.

[Question Q2.3] Un `Environment` peut donc être considéré comme responsable de la durée de vie des animaux amenés à être créés dans la simulation. Quelle incidence cela a-t-il sur la destruction d'un environnement ?

Lorsqu'un environnement est détruit, il serait intelligent de directement appeler la méthode `clean` afin d'effacer la faune et les cibles qui le composait.

[Question Q2.4] Dans quel(s) fichier(s) est définie l'utilisation de ces touches ?

L'utilisation de ces touches sont définis dans `EnvTest.cpp` grâce à l'utilisation des méthodes d'`Application`.

Par exemple pour la touche T, grace au `switch(event.key.code)`:

```
case sf::Keyboard::T:
{
    Vec2d const coord = getCursorPositionInView();
    getAppEnv().addTarget(coord);
}
break;
```

[Question Q2.5] Quel prototype proposez-vous pour les deux méthodes distinctes suggérées ?

Les deux méthodes distinctes suggérées sont:

- `void update(sf::Time dt)`
Elle n'est pas déclarée comme `const` car elle modifie trois attributs définissant un `ChasingAutomaton` (sa position (héritée du `CircularCollider`), sa direction et la norme de sa vitesse).
- `Vec2d forceAttraction() const`
Elle est déclarée comme `const` car elle fait que retourner la force exercée par une cible sur un `ChasingAutomaton` et ne modifie pas l'instance courante.

[Question Q2.6] Comment proposeriez-vous d'utiliser un type énuméré pour faire en sorte que la décélération puisse valoir à choix : 0.9 (forte décélération/faible vitesse), 0.6 (décélération et vitesse moyennes) ou 0.3 (décélération faible/vitesse forte) ? Et comment proposez-vous d'intégrer cet élément de choix dans votre code si l'on souhaite qu'il soit dicté par l'extérieur (de sorte à pouvoir décider à la demande quelle décélération l'on veut utiliser selon la situation) ?

Nous pouvons créer un type enum (que nous plaçons en dehors de la classe):

```
enum Deceleration {fort , moyen, faible};
Deceleration d = moyen; //choisie par défaut
double decelere;
switch(d)
{
    case fort: decelere = 0.9;
        break;
    case moyen: decelere = 0.6;
        break;
    case faible: decelere = 0.3;
        break;
}
```

Afin de pouvoir décider à la demande quelle décélération l'on veut utiliser selon la situation, on peut créer un paramètre `deceleration` dans les fichiers json qui sont ensuite facilement modifiables. Mais puisque nous ne savons pas comment intégrer des Enum dans le fichier json, nous avons choisi une valeur par défaut.

[Question Q2.7] Pourquoi selon vous est-il préférable de déclarer la méthode `setRotation` (et par le même raisonnement une éventuelle méthode `setPosition` de `CircularCollider`) comme protégée ?

Il est préférable de déclarer les méthodes `setRotation` et `setPosition` comme protégées car cela permet seulement les sous classe de manipuler la position et la rotation des objets qui héritent de `CircularCollider` et empêche tout autre utilisateur de les manipuler.

[Question Q2.8] Que devez-vous modifier pour que le dessin de l'environnement tienne compte de la présence de l'animal (et donc l'affiche aussi) ?

Afin que le dessin de l'environnement tienne compte de la présence des animaux, il faut invoquer le dessin de chaque animal de la faune soit appelé avec la fonction `Animal::draw`.

[Question Q2.9] (avancée) Pour pouvoir tester différentes configurations (cibles dans le champ de vision et hors de celui-ci), il est nécessaire ici de pouvoir faire varier à volonté le vecteur direction de l'animal. Pour se donner cette liberté, le test `TargetInSightTest` redéfinit une sous-classe de `Animal` appelée `DummyAnimal` (juste utilisée pour les besoins du test). Sauriez-vous expliquer pourquoi ?

Nous pensons que le test `TargetInSightTest` redéfinit une sous-classe d'`Animal`, `DummyAnimal`, dans laquelle des fonctions sont redéfinies pour seulement tester notre animal (s'il a le `DummyAnimal` in sight).

[Question Q2.10] Quel type de retour proposez-vous pour cette méthode ?

Le retour de la fonction `getTargetsInSightForAnimal` sera de type `vector<Vec2d>` afin de retourner directement toutes les cibles dans le champ de vision de l'animal et de pouvoir y accéder facilement.

[Question Q2.11] Que devez-vous modifier pour que la mise à jour de l'environnement tienne compte de la présence de l'animal (et donc invoque les mises-à-jour nécessaires sur les animaux au bout de l'écoulement d'un pas de temps `dt`) ?

Il faut mettre à jour les instances des animaux a chaque pas de temps dans la méthode `update` d'`Environment` avec une boucle for qui regarde à travers la liste faune d'animaux avec la commande: `animal -> Animal::update(dt);`

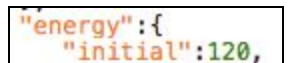
Semaines 7-8(01/04-14/04)

[Question Q3.1] Quelles méthodes avez-vous décidé de déclarer comme virtuelles pures dans la classe `Animal`? A quels endroits l'utilisation du mot clé `override` vous semble t-il pertinent ?

Toutes les méthodes que nous devons specifier dans les classes `Scorpion` et `Gerbil` doivent etre declarees comme virtuelles pures dans `Animal`. Ces méthodes dans les classes `Scorpions` et `Gerbil` doivent avoir le mot clé `override`.

[Question Q3.2] : que doit-on retoucher dans le fichier `.json` si l'on souhaite par exemple modifier en cours de simulation le niveau d'énergie initial par défaut des gerbilles?

Il faut, dans le fichier `app3.json` retoucher la valeur entrée. Par exemple, dans la capture d'écran ci-contre, il faudrait changer "120".



```
"energy":{  
  "initial":120,  
}
```

Pour le test nous avons par contre copiee `app3.json` dans `appTest.json` afin de ne pas modifier `app3.json`.

[Question] Telle que suggérée, la méthode `getEntitiesInSightForAnimal` est une solution simple mais pas idéale du point de vue de l'encapsulation. Expliquez pourquoi.

Le problème d'encapsulation de `getEntitiesInSightForAnimal` est qu'il est obligé de prendre un animal dans ces arguments afin de trouver les `OrganicEntities` qui sont dans son champ de vision. Cela rend la tâche plus difficile de pouvoir utiliser les méthodes d'animal dans l'environnement.

[Question Q3.3] Comment proposez-vous d'utiliser les classes Updatable et Drawable dans votre conception (on peut considérer que toute entité organique évolue au cours du temps, par exemple la nourriture évolue en se périssant/fanant).

Toutes les classes avec une méthode `void draw(sf::RenderTarget& target) const` héritent de la classe `Drawable` et toutes celles avec une fonction `void update(sf::Time dt)` héritent de la classe `Updatable`.

Ainsi, on peut inclure `Updatable` et `Drawable` dans `OrganicEntity.hpp` puisque toutes les entités organiques évoluent au cours du temps et sont représentables. `Environment` est aussi `Updatable` et `Drawable` mais `FoodGenerator` est seulement `Updatable`.

[Question Q3.4] Pourquoi tester le type des objets à l'exécution est-il potentiellement nocif ?

Tester le type des objets à l'exécution est potentiellement nocif, si dans le test on change les types des objets testés, alors cela pourrait poser des problèmes.

Semaine 9 (14/04-21/04)

[Question Q3.5] A quel niveau de la hiérarchie est-il intéressant de placer l'affichage des informations de debugging ? Le fait qu'un `CircularCollider` soit maintenant dessinable rediscute-t-il vos héritages de `Drawable` ?

L'affichage des informations de debugging, mis dans la méthode `DrawDebug`, peut être directement placé dans `Animal` car seulement les animaux ont un État qui peut donc changer et être dessiné dans cette méthode.

Le fait qu'un `CircularCollider` soit dessinable fait qu'on le fait directement hériter de `Drawable` (au lieu de faire hériter seulement les `OrganicEntity` de cette classe, du coup on efface l'héritage au niveau de `OrganicEntity` mais laissons `Updatable`).

[Question Q3.6] Comment proposez-vous de procéder pour faire en sorte que les sous-classes d'entités organiques existantes vieillissent et puissent avoir soit la longévité par défaut soit une longévité spécifique?

Nous avons mis une méthode virtuelle `getLongevity()` dans la classe `OrganicEntity` retournant la longévité "infinie" par défaut. Celle-ci est surchargée dans les classes `Scorpion`

et `Gerbil` qui retournent leur propre longévité alors que la classe `Food` n'a pas de surcharge mais utilise la valeur par défaut.

[Question Q3.7] Comment proposez-vous de procéder pour qu'une entité morte disparaisse de l'environnement ? Quelle précaution doit alors être prise pour une gestion correcte de la mémoire ?

Dans la méthode `Update` dans `Environnement`, lorsque nous parcourons le tableau `faune`, nous faisons appelle à `isDead()` dans `OrganicEntity` pour chaque pointeur et si la méthode retourne vrai, alors nous effaçons la zone pointée puis affectons à l'`OrganicEntity*` un `nullptr`. Il faut faire attention à toujours tester si la cible n'est pas un `nullptr` sinon cela créera un segmentation fault.

Ensuite, nous avons ajouté la commande:

```
faune.erase(std::remove(faune.begin(), faune.end(), nullptr), faune.end());
```

Cette commande nous permet de supprimer tous les éléments valant `nullptr` dans le tableau `faune` (qui est notre collection de pointeurs).

[Question Q3.8] Comment proposez-vous de modifier la méthode `getMaxSpeed` de sorte à ce qu'en dessous d'un seuil critique d'énergie l'animal se déplace plus lentement ?

Dans `getMaxSpeed`, on teste d'abord si son niveau d'énergie est en dessous du seuil critique (valeur que l'on a introduite dans les fichiers `.json` en tant que `seuilEnergie`, mis à 30). Si c'est le cas, l'animal aura directement une vitesse maximale de $0.5 * \text{getStandardMaxSpeed}$, indépendamment de son état.

[Question Q3.9] Pour réaliser les tests de collision, notre conception «voit» les entités organiques qui peuplent le monde simulé comme étant aussi des `CircularCollider`. Quelle autre conception est-il possible de mettre en place pour réaliser ces traitements? Quel avantage/inconvénient y voyez-vous ?

Au lieu de faire les tests de collisions grâce aux méthodes de `CircularCollider`, nous aurions faire un teste qui retournerait vrai si les positions des animaux observés sont les mêmes.

Le problème de cette conception est que nous aurions créé une classe exprès afin de faire ces tests et nous ne l'utiliserions pas sinon. De plus, cette vision de conception n'est pas idéale car les animaux seraient les uns sur les autres.

[Question Q3.10] Comment proposez-vous d'utiliser le «double dispatch» pour coder la méthode `meet` sans faire de test de type ?

Nous avons fait une méthode virtuelle pure `meet()` dans la classe `Animal` que nous avons redéfini dans les classes `Gerbils` et `Scorpions`. Ainsi, nous utilisons leurs getters afin d'obtenir la bonne perte d'énergie et le nombre minimum/maximum de bébés. De cette manière, il est également possible d'appeler cette méthode pour les deux partenaires dans `updateState()` dans la classe `Animal` lorsque les deux animaux entrent en collision.

[Question Q3.11] Comment proposez-vous de mettre en place le temps de gestation.

Nous avons ajouté un attribut `tempsEnceinte` qui est initialisée à `sf::Time::Zero`. Lorsqu'une femelle est dans l'état `MATING`, la femelle commence son temps de gestation qui sera transmis avec des getters dans les classes `Scorpion` et `Gerbil`. Ce temps sera décrémenté dans le `updateState` de `dt`.

Puisque les temps de gestations dans le fichier '`config`' étaient des doubles, nous avons dû mettre le double en `sf::seconds` pour mettre `tempsEnceinte` en `sf::Time` et pouvoir le décrémenter de `dt` après ce pas de temps.

[Question Q3.12] Un animal ne peut donner naissance qu'à des animaux de son type, comment proposez-vous de mettre en oeuvre la méthode `give_birth` (ou équivalent) dans la hiérarchie d'animaux.

On fait une méthode virtuelle pure dans la classe `Animal` que l'on définit dans les classes `Scorpion` et `Gerbilles`. Ainsi, chaque type d'animal ajoute une entité de son type dans l'Environnement (avec la commande : `getAppEnv().addEntity(new Gerbil(position));`).

[Question Q3.13] Comment stockez-vous le nombre de bébés attendus (utile pour mettre en place la naissance lorsque le temps de gestation est écoulé).

Nous avons fait un attribut `nbBabies`, un getter et un setter pour pouvoir gérer les bébés attendus.

[Question Q3.14] Vous semble-t-il utile d'avoir un attribut pour mémoriser les prédateurs potentiels d'un animal.

Oui, nous les avons stockés dans une liste, enemies, dans `Animal`.

[Question Q4.1] Quel liens d'héritage vous semble t-il pertinent d'établir lors du codage de la classe `Wave` ?

Nous avons programmé la classe `Wave` en tant qu'un `OrganicEntity` pour que les ondes puissent entrer en collision avec les diverses obstacles. Donc `Wave` herite de `OrganicEntity`.

[Question Q4.2] Comment comptabilisez-vous le temps écoulé depuis le début de la propagation de l'onde pour faire évoluer le rayon de celle-ci~?

Le temps écoulé est stocké dans l'attribut `tempsEcoule` que nous mettons à jour avec `update()` qui appelle d'abord `updateTemps()`, puis `updateRayon()` (qui est en fonction du temps écoulé), puis `updateEnergie()` et enfin `updateIntensite()` (qui sont les deux en fonction du rayon) .

[Question Q4.3] Quelles retouches devez vous faire à la `Environment` pour prendre en compte la présence des ondes et les voir se propager et se dessiner ?

Nous avons besoin de faire une liste de pointeurs de waves afin de pouvoir les dessiner grâce à une itération dans le draw de l'environnement.

[Question Q4.4] Quelle(s) autre(s) modification(s) devez vous apporter à la classe `Environment` suite à l'ajout de la méthode `newObstacle`?

Nous avons créé une liste de `CircularCollider`, nommée obstacles, qui va contenir les rochers. Ainsi, `newObstacle` appelle la méthode `addObstacle` en passant en paramètre un pointeur sur un `CircularCollider` qui est essentiellement un pointeur sur un rocher.

[Question Q4.5] Comment proposez-vous de représenter, l'ensemble des senseurs et leurs angles dans `NeuronalScorpion` ?

Nous avons fait un tableau de paires, avec un double caractérisant l'angle du sensor et un pointeur sur le sensor correspondant.

[Question Q4.6] Quel prototype proposez-vous pour la méthode `getPositionOfSensor()` ?

Prototype: `Vec2d getPositionOfSensor(size_t i) const;`

La méthode retourne donc un `Vec2d` en fonction du sensor a la position `i` dans le tableau.

[Question Q4.7] Quelle(s) méthode(s) ajoutez-vous et à quelle(s) classe(s) pour qu'un senseur puisse connaître l'intensité cumulée des ondes qui le touchent; sans pour autant coder de getters trop intrusifs donnant accès à l'ensemble des ondes de l'environnement?

Pour ce faire, dans `Environment` (puisque celui-ci a des pointeurs sur tous les objets qui le peuplent), nous avons définis une méthode `getIntensityForSensor(scorpion, index)` avec comme paramètres l'index du senseur et un pointeur sur le scorpion auquel il appartient. Cette méthode retourne l'intensité cumulée des ondes qui le touchent sans que le senseur ait accès à l'ensemble des ondes, simplement ce double. Des getters sont utilisés dans cette méthode afin de pouvoir accéder aux attributs du senseur.

[Question Q4.8] Le senseur, pour connaître sa position à donc besoin de connaître le scorpion auquel il appartient. Qu'est-ce que cela implique au niveau des dépendances entre les classes `NeuronalScorpion` et `Sensor` ?

Le `NeuronalScorpion` hérite de la classe `Sensor`. Pour que le sensor sache a quel scorpion il appartient, on lui a mis un attribut `scorpionHote` qui pointe sur le `NeuronalScorpion` correspondant.

[Question Q4.9] Cela a t-il une incidence sur la façon de construire un senseur?

Il faut alors ajouter au constructeur du sensor le scorpion auquel il "appartient".

[Question Q4.10] Comment et où proposez-vous de modéliser les états possibles?

Les Etats peuvent être modélisés en tant qu'enum dans `NeuronalScorpion` puisqu'ils sont différents de ceux dans `Animal` (`WANDERING`, `IDLE`, `MOVING`, `TARGET_IN_SIGHT`). On redéfinit aussi un `Etat`, `state`, qui masque celui hérité de `Animal`. Nous avons dû créer une classe Enum `Etat` dans `Animal` et `NeuronalScorpion`.

[Question Q4.11] Comment proposez-vous de modéliser les horloges associées aux états?

On a fait un attribut horloge de type `sf::Time` qu'on met à 3 ou 5 dès que le `NeuronalScorpion` passe aux états respectifs MOVING et IDLE. Il va se faire décrémenter dans la méthode `update()` et lorsqu'il atteint `sf::Time::Zero` (ou moins), on gère son changement d'état dans `updateState()`.

[Question Q4.12] Quelles actions liées à la gestion des différentes horloges impliquées devez-vous entreprendre et dans lesquels des états du scorpion?

Nous avons fait alors un `updateHorloge()` qui décrémente d'un temps dt l'horloge du scorpion. Cette méthode est appelée au début de `update()`.

[Question Q5.1] : Lorsqu'il n'y a aucun graphe actif dans un programme (dérivé de Application), l'identificateur actif est à -1. Cela est décidé par la classe Application qui gère la numérotation des graphes au travers de son attribut `mCurrentGraphId` (jetez un oeil au corps de `Application::addGraph`). Pourquoi une table associative est-elle plus appropriée qu'un vector pour modéliser l'ensemble des graphes et l'ensemble des libellés de la classe Stats?

C'est plus intéressant car cela permet d'associer chaque graphe avec son titre. Ainsi, on peut accéder à chaque graphe en connaissant seulement son titre, sans connaître sa position dans la table associative.

[Question Q5.2] Quelle méthodes prévoyez-vous d'ajouter/modifier et dans quelles classes pour réaliser les décomptes souhaités et construire les ensembles `new_data`?

Afin de connaître le nombre de Scorpions, Gerbils et Food Sources qui sont créés, nous avons décidé de mettre des compteurs statics dans chacune de ces classes, initialisés à l'extérieur des classes et incrémentés à chaque fois que le constructeur de chacun est appelé (donc quand un nouveau Scorpion/Gerbil/Food est créé). Ensuite, puisque nous avons mis `compteurScorpion()`, `compteurGerbil()` et `compteurFood()` en private, nous avons fait des static getters afin d'accéder à ces attributs dans la class Environment. Nous ajoutons alors ces données dans un `unordered_map` `newData` de `string` (titre de la suite des points) et `double` (le nombre d'entités de ce titre à au moment donné).

Attention: nous avons aussi du décrémenter chaque compteur donc avons fait un destructeur explicite pour chacune des entités organiques, dans lequel nous décrémentation de 1 le compteur associé.

[Question Q5.3] Quelles modifications apportez-vous et dans quelles classes pour ajouter les traitements souhaités ?

Dans le fichier `Application`, quand on appuie sur la touche Tab, le programme appelle la fonction `toggleStats()` pour qu'il se mette en mode Neuronal s'il est dans PPS simulation et vise-versa.

Nous avons alors modifié le draw de `Stats` afin qu'il n'affiche seulement le graph qui a l'identifiant actif ainsi que la méthode `fetchData()` d'`Environment`. Celle-ci itère sur la liste de Waves de l'environnement qui est dans ses attributs privés. FetchData doit d'abord savoir quel est le graph qui doit être utilisé en comparant le titre en paramètre aux titres des deux graphes possibles, General et Waves. Quand c'est General, il regarde les compteurs de Scorpion, Gerbil et Food, mais quand c'est Waves, nous avons créé un compteur qui est incrémenté au moyen d'une boucle "for" implicite, à chaque fois qu'elle accède à un élément (non nullptr) de la liste Ondes. De même que pour General, on place ensuite cette donnée à `newData`.