TP 5 : Modélisation du comportement

Dans le cadre de ce TP, nous allons nous intéresser à la modélisation du comportement de moutons et de loups. Les moutons vivront leur vie et se déplaceront en troupeaux alors que les loups, affamés, viendront mettre la pagaille et attaquer les moutons.

Dans ce TP, nous verrons :

- L'utilisation d'un modèle type Reynolds pour la gestion du déplacement des entités.
- La modélisation du comportement via un modèle proche des « Behavior Trees ».
- L'utilisation de triggers pour gérer la perception.

Préambule

Le TP est construit autours d'un simulateur de système multi-agent se trouvant dans la classe *Crowds::Simulator* qui a pour rôle de simuler des agents situés dans un monde 2D modélisés par héritage de la classe *Crowds::Agent*. Normalement, durant le TP, vous n'aurez pas à utiliser directement la classe *Crowds::Simulator* fournie mais simplement les méthodes de la classe *Crowds::Agent*.

Chaque agent possède une position 2D (méthodes *getPosition* / *setPosition*), un rayon (méthode *getRadius*), une vitesse courante (méthodes *getSpeed, setSpeed*), une orientation (méthodes *getOrientation, getFrontVector*) et une masse (méthode *getMass*). Cet agent peut aussi percevoir les agents qui sont autour de lui :

```
template <typename EntityType> std::vector<std::shared_ptr<EntityType>> perceive(float radius, float openingAngle = Math::pi)
```

Cette méthode retourne une table contenant tous les voisins perçus dans un rayon *radius* autour de l'agent (la distance est calculée à partir des positions des agents) et dont l'angle avec le vecteur direction (*getFrontVector*) est inférieur ou égal à *openingAngle*.

Le comportement de l'agent est décrit via la méthode *void update(double dt)*, qui est abstraite dans la classe *Agent.* Cette méthode est appelée à chaque pas de temps avec en paramètre le temps écoulé depuis le dernier appel.

D'autre part, la classe *Crowds*::*Boid,* héritant de la classe *Crowds::Agent*, met à disposition différentes méthodes permettant de gérer un déplacement des agents à la manière de Reynolds. Cette classe met à votre disposition des méthodes permettant de gérer les forces de steering (voir les méthodes *getSteeringForce , setSteeringForce, addSteeringForce*). Les comportements basiques de Reynolds sont déjà programmés : *seek, flee, pursuit, evasion, arrival, wander, cohesion, alignment, separation, stop*. D'autre part, pour permettre une gestion correcte de l'évitement de collisions entre agents, les méthodes *mayCollide* (permettant de savoir si une collision est anticipée) et *avoidCollisions* (qui réalise l'évitement de collision, demandez moi pour une description de l'algorithme utilisé) sont à votre disposition. Toutes ces méthodes retournent une force de steering que vous pouvez ensuite appliquer (méthode *addSteeringForce* ou *setSteeringForce*). Il est à noter que la classe *Crowds::Boid* implémente la méthode *void update(double dt)* de la classe *Crowds::Agent* et effectue l'intégration de la force de steering pour déterminer la nouvelle position et la nouvelle vitesse de l'agent.

Pour modéliser le comportement des loups et des moutons, vous disposez des classes *Crowds::Prey* (les moutons) et *Crowds::Predator* (les loups) qui héritent toutes deux de la classe *Crowds::Boid*.

Au fur et à mesure de ce TP, vous découvrirez d'autres fonctionnalités mises à votre disposition pour la description du comportement des agents et leur communication.

Première partie : un peu de déplacement

Dans un premier temps, nous allons programmer un comportement simple de déplacement de moutons décrit dans la classe *Crowds::Prey*.

Question 1 : En vous aidant de la méthode *wander* fournie par la classe *Crowds::Boid*, dotez le mouton d'un comportement de navigation consistant à se mouvoir aléatoirement dans l'environnement. Un bon paramétrage de wander est de fournir 2.0 (paramètre 1) pour la distance du cercle, 3.0 pour le rayon (paramètre 2) et dt*5.0 pour le pourcentage de modification (paramètre 4). Attention, cette méthode, en troisième paramètre, prend une référence sur un flottant modifiable (*previousAngle*) qui permet de gérer la continuité du comportement au cours du temps. Ce flottant doit être un attribut de votre classe. Visualisez le résultat de ce premier comportement.

Vous pouvez constater que les moutons ont une légère tendance à se marcher dessus. Nous allons maintenant nous occuper de la gestion des collisions.

Question 2 : Modifiez le comportement du mouton de la manière suivante (système à base de règles basique) :

- Si le mouton détecte qu'il peut entrer en collision avec un de ses voisins (méthode *mayCollide*), il évite cette collision (méthode *avoidCollisions*). Pour gérer la perception du mouton, vous utiliserez la méthode *perceive* de la classe Crowds::Agent. Pour la perception, utilisez une distance de pecetion de 10.0 et une ouverture de Math::pi.
- Si le mouton ne détecte pas de collision, il se promène aléatoirement (ce que vous avez testé à la question 1).

Une fois ce comportement programmé, testez votre simulation en mettant une bonne quantité de moutons (de 500 à 1500 en fonction de la puissance de votre processeur).

Vous venez d'encoder un comportement très simple de mouton se promenant aléatoirement dans un champ mais ayant la capacité de s'éviter.

Question 3 : Nous allons maintenant donner aux moutons la faculté de se déplacer en groupe. Pour ce faire, nous allons ajouter une règle au comportement de la question 2. Si le mouton n'anticipe pas de de collision avec ses voisins et qu'il perçoit un mouton plus lourd que lui (méthode *getMass*). Il va suivre le mouton le plus lourd. Pour ce faire, il utilisera un comportement d'arrivée (méthode *arrival*) sur le mouton le plus lourd et un comportement de séparation de ses voisins (méthode *separation*) avec une vitesse d'adaptation de *dt* (dernier paramètre de ces deux méthodes). Dans le cas contraire, il appliquera le comportement de la question 2. Une fois le comportement programmé, testez votre implémentation. Vous devriez voir quelques groupes se former, se croiser et se modifier.

La communication par messages

Les classe *Agent* possède plusieurs méthodes permettant d'effectuer une communication par message :

```
template <typename MessageType> void broadcast(const MessageType & message)
```
Permet d'envoyer un message à tous les agents de la simulation.

```
template <typename MessageType> void post(const MessageType & message, Agent * agent)
```
Permet d'envoyer un message à un agent particulier.

```
template <typename MessageType> void post(const MessageType & message, float radius)
```
Permet d'envoyer un message à tous les agents se trouvant autour de l'agent.

Pour qu'un agent puisse recevoir un message, il faut qu'il y soit abonné. Pour ce faire, vous disposez de la méthode suivante de la classe Agent :

```
template <typename MessageType, typename Callback>
stdext::message_handler::receiver_callback<MessageType> * createReceiver(const Callback
& callback)
```

Cette méthode crée un récepteur de message pour l'agent courant qui consiste à appeler la fonction callback (signature : *void function(const MessageType &)*) fournie lorsque qu'un message de type *MessageType* est reçu.

Le fichier « Crowds/Messages.h » contient la déclaration de quelques messages utiles à la simulation.

Un trigger

Nous souhaitons désormais doter nos moutons de la capacité à boire. Pour ce faire, nous disposons de la classe *Crowds::Water* qui est un agent particulier. En effet, cet agent est simulé comme tout autre agent mais si vous regardez dans l'initialisation de l'application, il s'agit d'un agent qui n'est pas perceptible par les autres agents. La raison est la suivante, une étendue d'eau peut être grande et impliquerait pour les agents simulés d'avoir un grand rayon de perception pour pouvoir en percevoir sa position ; ce qui aurait aussi un énorme impact sur la complexité globale de la simulation.

Pour palier au problème évoqué ci-dessus, nous allons programmer l'agent *Crowds::Water* de manière à déporter la perception sur ce dernier et utiliser le système de communication par messages pour informer les agents de la présence d'eau.

Question 4 : Programmez dans la méthode *update(double dt)* de l'agent *Crowds::Water* l'envoi systématique d'un message de type *WaterMessage* à tous les agents se trouvant à une distance inférieure ou égale à son rayon plus 10m (voir le paragraphe « Communication par messages » pour les méthodes mises à votre disposition).

Question 5 : Du côté du mouton (Crowds::Prey), dans le constructeur, référencez un récepteur de messages (méthode *createReceiver*, voir le paragraphe « Communication par messages ») réagissant aux messages de type *WaterMessage*. Ce récepteur ajoutera les points d'eau perçu dans un attribut de la classe Prey contenant tous les points d'eau perçus (de type *std ::vector* par exemple). Comme le mouton a la mémoire courte, l'attribut sera vidé à la fin de la méthode *update* à chaque pas de temps. Pour tester votre implémentation, affichez, à la fin de la méthode update le nombre de points d'eau perçus par le mouton. Après le test, vous supprimerez cet affichage.

Notre mouton a soif.

Question 6 : Nous allons maintenant prendre en compte la soif du mouton. Ajoutez un attribut flottant correspondant à la soif dans la classe du mouton. Cet attribut représentera le pourcentage d'eau que le mouton a en réserve. Au départ, ce pourcentage sera tiré aléatoirement dans l'intervalle [0 ;100] (utilisez *Math::Interval<float>(0, 100.0).random()* pour obtenir un nombre aléatoire entre 0 et 100). Modifiez le comportement du mouton de manière à ce que la quantité d'eau diminuera de 1% en 1 seconde de simulation.

Question 7 : Lorsque le mouton tombe en dessous de 10% d'eau en réserve, il a soif. Il doit donc partir en recherche d'eau. Le comportement devrait être le suivant :

1. Si le mouton n'a pas soif ou ne perçoit pas d'eau, il utilise son comportement normal (programmé en question 3).
2. Si le mouton a soif et perçoit de l'eau, il sélectionne une source d'eau à laquelle boire, s'en souvient (i.e. entre deux appels à update, la source d'eau ne doit pas changer) et dirige vers cette dernière (comportement *seek*). Lorsque le mouton est à une distance du centre de la source d'eau inférieure au rayon de cette même source, il peut boire. Le mouton boit 5% d'eau par seconde et ce jusqu'à ce que la réserve d'eau soit remontée à 100%.

Comme vous pouvez le remarquer, le point (2) du comportement nécessite que le mouton ait une mémoire de son choix (un attribut suffit pour cela) mais nécessite aussi que le mouton reste dans l'état boire de l'eau jusqu'à ce que la jauge soit à 100%. Il s'agit donc de l'utilisation d'un automate avec une notion d'état courant : (1) ou (2) et de transition de (1) vers (2) quand le mouton a soif et perçoit de l'eau et de transition de l'état (2) vers (1) quand la jauge d'eau est à 100%.

Programmez le comportement du mouton.

Les « behavior trees »

Une implémentation un peu évoluée de « behavior trees » est disponible dans l'espace de nommage *AI::Tasks*. Il repose sur des notions de tâches et d'opérateurs permettant de combiner ces tâches.

Une tâche est décrite par héritage de la classe *AI::Tasks::Task* et modélise un comportement atomique. Une tâche lors de son exécution peut être dans 4 états :

- *AI::Tasks::Task::Status::waiting* : la tâche est initialisée, en attente de démarrage.
- *AI::Tasks::Task::Status::running* : la tâche est en cours d'exécution.
- *AI::Tasks::Task::Status::success* : la tâche a terminé son exécution par un succès.
- *AI::Tasks::Task::Status::failure* : la tâche a terminé son exécution par un échec.

La classe de tâche (*AI::Tasks::Task*) possède 5 méthodes importantes :

```
Status getStatus() const
```

Retourne l'état d'exécution de la tâche (Cf. ci-dessus).

```
virtual void initialize()
```

Méthode appelée pour initialiser la tâche. Méthode à redéfinir par héritage.

```
virtual Status activity()
```

Méthode appelée tant que la tâche est en cours d'exécution. Elle retourne son état d'exécution (Cf. ci-dessus, running, success, failure). Méthode à redéfinir par héritage.

```
virtual void finalize()
```

Cette méthode est appelée lorsque l'exécution de la tâche est terminée. Méthode à redéfinir par héritage.

```
Status execute()
```

Cette méthode est appelée pour exécuter la tâche. Elle retourne l'état de la tâche (waiting, running, success, failure).

Le fichier « AI/Tasks/Task.h » contient la définition de cette classe et deux fonctions permettant de rapidement créer des tâches via des lambdas fonctions :

```
template <typename InitializationType, typename ActivityType, typename FinalizationType>
std::shared_ptr<Task> makeTask(const InitializationType & init, const ActivityType & activity, const FinalizationType & finalization)
```

Cette function crée une tâche avec init comme implémentation de la méthode *initialize()*, activity comme implémentation de la méthode *activity()*, et finalization comme implémentation de la méthode *finalize()*.

```
template <typename ActivityType>
std::shared_ptr<Task> makeTask(const ActivityType & activity)
```

Cette fonction crée une tâche ne possédant qu'une implémentation de la méthode *activity()* dont l'implémentation est fournie via le paramètre activity de la fonction.

Remarque : comme vous pouvez le remarquer, ces fonctions renvoient des shared_ptr<Task> qui sont utilisés par tout le système. Cela nous permet de ne pas avoir à gérer manuellement la mémoire.

Pour vous faciliter la création de tâches complexes, vous disposez d'un ensemble de fonctionnalités décrites dans le fichier « AI/Tasks/Operators.h ». Ces opérateurs sont eux-mêmes implémentés sous la forme de tâches pour des raisons de généralisation et de facilité d'utilisation du système. D'autre part, des opérateurs prenant en compte le temps sont fournis dans le fichier « AI/Tasks/TimeOperators.h ».

Ajoutons un peu de vie… de chasse… de moutons apeurés…

Nous allons maintenant programmer le comportement du loup. Notre loup a principalement deux objectifs dans la vie : boire et manger du mouton. De son coté, le mouton, moins bête qu'il n'y parait, a peur du loup et souhaite donc ne pas être mangé. De plus notre mouton peut bêler pour prévenir ses compatriotes d'un danger.

Dans un premier temps, nous allons programmer le comportement du loup dans la classe *Crowds::Predator*. Le loup perçoit à 15m (contrairement au mouton qui perçoit à 10m), cela lui donne un petit avantage.

Le loup a le comportement suivant

- Il se promène en évitant les collisions (comportement du mouton de la question 2).
- Lorsqu'il voit un mouton alors qu'il se promène, il le chasse (comportement de poursuite : voir méthode *pursuit*). Si au bout de 60.0s le loup n'a pas attrapé sa proie, il arrête de chasser pour 20s en continuant à se promener. Si le loup attrape sa proie, il s'arrête sur place et mange pendant 30s.

Question 8 : Programmez le comportement de promenade du loup sous la forme d'une tâche (Cf. section précédente sur les « behavior trees ») ainsi qu'une méthode de création de cette tâche. Vous pourrez vous aider d'une lambda fonction et de la fonction *makeTask* avec un seul paramètre pour créer la tâche. Ce comportement, quoi qu'il arrive est dans l'état running. Testez votre implémentation en appelant la méthode *execute()* de la tâche dans la méthode update du loup. Le loup devrait avoir le même comportement que le mouton à la question 2.

Question 9 : Programmez une tâche de poursuite de Mouton ainsi qu'une méthode de création de cette tâche. Cette tâche consiste à utiliser un comportement de type *pursuit* sur un mouton sélectionné parmi les moutons perçus par le loup. Cette tâche se termine par un succès si le loup a attrapé le mouton, un échec si le loup ne perçoit plus le mouton (i.e. si la distance entre le mouton et le loup devient supérieure à 20m). Lorsqu'un loup attrape un mouton, ce dernier meurt. Pour ce faire, il appelle la méthode *killed()* du mouton. Pour tester votre implémentation, vous pouvez programmer, dans la méthode update l'exécution de la tâche de la question 8 quand un loup ne voit pas un mouton et l'exécution de la tâche décrite dans cette question sinon.

Question 10 : La tâche consistant à manger est simplement une tâche d'arrêt de déplacement du loup suivie d'une tâche d'attente de 20s (crée via la fonction *waitFor* du fichier « *TimeOperators.h* »). Le séquencement de tâches peut être effectué via l'opérateur && défini dans le fichier « Operators.h ». Créez une méthode de création de cette tâche. Cette tâche se termine toujours par un succès.

Question 11 : Programmez le comportement du loup (sous la forme d'une tâche) en créant une tâche combinant les tâches crées aux questions 8, 9 et 10 (vous créerez une méthode pour la création de cette tâche). Pour ce faire vous pouvez vous aider des opérateurs supplémentaires suivants (fichiers *Operators.h* et *TimeOperators.h*) :

- *succeedsWhen* : opérateur prenant une condition et une tâche en paramètre. La tâche passe dans un état *success* quand la condition est satisfaite ou continue son exécution jusqu'à un état échec.
- *ifSuccessOrFailure* : opérateur prenant 3 tâches en paramètres. Cet opérateur exécute la première tâche. Si la première tâche se termine par un succès, la seconde tâche est exécutée et le résultat est l'état de la seconde tâche. Si la première tâche est un échec alors la troisième tâche est exécutée. Le résultat de cet opérateur est alors le résultat de la troisième tâche.
- *tryDuring* : opérateur essayant de réaliser la tâche fournie en paramètre pendant un certain temps. Si la tâche se termine durant le temps imparti, le résultat de l'opérateur est le résultat de la tâche. Si la tâche ne se termine pas durant le temps imparti, le résultat de l'opérateur est le résultat fourni en second paramètre de *tryDuring*.
- *loop* : opérateur consistant à répéter indéfiniment la tâche fournie en paramètre, quel que soit son résultat. Cet opérateur devrait être utiliser pour boucler indéfiniment sur le comportement du loup actuellement décrit.

Comme vous pouvez le constater, les moutons ne sont pas très réactifs à l'approche du loup. Nous allons corriger cela.

Lorsqu'un mouton perçoit un loup, il va envoyer un message (de type *PanicMessage*) à tous les moutons dans un rayon de 20m, cela simule le fait que le mouton se mette à bêler à la perception du danger. Les moutons en réaction, arrêteront ce qu'ils sont en train de faire et se mettrons à fuir le prédateur en question pendant au moins 5s (et oui… le mouton a vraiment la mémoire courte…).

Question 12 : Programmez une méthode permettant au mouton d'émettre le message de panique (type PanicMessage) lorsqu'un prédateur est perçu (vous pouvez utiliser la méthode *filterByType*

fournie par la classe Agent pour filtrer les agents par leur type réel). Lorsqu'un message de type PanicMessage est reçu par un agent, ce dernier ajoute le prédateur dans une table de correspondance (type *std::map* ou *std::unordered_map*) qui fait correspondre un prédateur avec le temps depuis lequel il a été perçu. Quand le message arrive, ce temps est donc mis à 0. Dans la méthode update, vous mettrez à jour la mémoire comme suit :

- Le temps depuis lequel le prédateur a été perçu doit être incrémenté de dt.
- Si le temps dépasse 3.0s, l'entrée est supprimée.

Donc notre mouton possède une belle mémoire de 3s. Heureusement qu'elle sera rafraichie régulièrement en cas de panique généralisée…

Question 13 : Passez le comportement complet du mouton sous la forme d'une seule tâche (comme ce que vous avez fait pour le loup) et fournissez une méthode de création de cette tâche. La tâche créée sera toujours dans l'état running. Testez votre résultat pour bien vérifier que le mouton se comporte toujours de la même manière.

Question 14 : Créez une nouvelle tâche qui aura le comportement suivant : l'exécution du comportement de la question 13 est préconditionné par l'absence de prédateurs et s'arrête son exécution si un prédateur est perçu. Cette tâche sera encapsulée par un opérateur *loop* pour que son exécution ne s'arrête jamais. Vous pouvez utiliser les opérateurs suivants pour créer cette tâche :

- *precondition* : cet opérateur prend une précondition et une tâche. La tâche n'est lancée que lorsque la précondition devient vraie. Tant que la précondition n'est pas vraie, l'opérateur reste dans un état waiting. A partir du moment où la précondition est devenue vraie, l'état de l'opérateur est l'état de la tâche fournie en paramètre.
- *succeedsWhen* : opérateur prenant une condition et une tâche en paramètre. La tâche passe dans un état *success* quand la condition est satisfaite ou continue son exécution jusqu'à un état échec.

Testez votre résultat en exécutant ce comportement comme comportement du mouton. Normalement, le mouton devrait se comporter comme dans la question précédente… Normal, personne ne crie au loup !

Question 15 : Créez une tache qui répète indéfiniment (opérateur *loop*) une séquence (opérateur &&) consistant à attendre pendant 10s +/- 1s (opérateur *waitFor*) et une tâche consistant à tester si un prédateur est perçu et à envoyer un message de type *PanicMessage* si c'est le cas (méthode de la question 12).

Question 16 : Le comportement global du mouton peut alors être décrit comme exécutant en parallèle (opérateur *parallelAnd*) le comportement de la question 14 et celui de la question 15. Testez ce nouveau comportement. Le résultat devrait être le suivant : lorsqu'un est au courant de la présence d'un prédateur il se met à évoluer en ligne droite sans jamais changer son cap, sinon, il reprend son comportement précédent. Cela est normal puisqu'aucune réaction aux prédateurs n'a été programmée.

Question 17 : Sur le modèle de la question 14, créez une tâche bouclant indéfiniment permettant au mouton de fuir le prédateur le plus proche (méthode *flee* de la classe *Agent*) lorsqu'au moins un prédateur est présent dans la mémoire. Pour tester votre résultat, ajoutez ce comportement au comportement de la question 16 en l'ajoutant aux comportements fournis en paramètre de l'opérateur *parallelAnd*. Normalement, les moutons devraient se mettre à fuir le prédateur.

Question 18 (subsidiaire) : Nous souhaiterions que le loup ait un comportement similaire à celui du mouton par rapport à la soif (excepté qu'il perd 1% d'eau toutes les 4 secondes). De plus, quand le loup a soif et qu'il perçoit de l'eau, il laisse tout tomber et cherche à boire, tant pis pour la nourriture. Proposez une nouvelle modélisation du comportement du loup.