TP 12 : modélisation de fourmilières

Dans le cadre de ce TP, nous allons programmer un simulateur de fourmilières. Ce simulateur devra à terme permettre de simuler plusieurs colonies de fourmis évoluant dans le même environnement. Il s'agira de programmer le comportement de fourmis (des agents) évoluant dans un environnement dans lequel se trouvent des fourmilières ainsi que de la nourriture. Les fourmis, lors de leur déplacement, laisseront des traces de phéromones dont la quantité dépendra du fait qu'elles transportent ou non de la nourriture. Au cours du TP, vous pourrez observer que ce simple dépôt de phéromones permet à la fourmilière de s'organiser efficacement pour la collecte de nourriture alors que chaque fourmi, prise individuellement, n'exhibe aucun comportement « intelligent ».

L'objectif principal de ce TP est de vous faire travailler sur la modélisation objet en C++ (héritage, polymorphisme, abstraction…). Il n'en reste pas moins que ce TP constitue une rapide introduction aux systèmes multi-agents et à la programmation de leur comportement via un système à base de règles.

1. Préparation du TP

Ouvrez éclipse et créez un nouveau projet C++ nommé *Ants* en activant les modes *debug* et *release*.

Allez dans le répertoire */share/esir1/prog/tp12* et recopiez le contenu du répertoire *src* dans votre projet.

Nous allons maintenant configurer le projet. Ouvrez les propriétés de votre projet et suivez les instructions suivantes :

- Ouvrez la fenêtre C/C++ Build / Settings.
- En haut de la partie droite de la fenêtre, pour configuration, sélectionnez « All configurations ».
- Dans GCC C++ Compiler/Includes, ajoutez le répertoire contenant vos fichiers sources afin que le compilateur sache où recherche les fichier « .h »
- Dans GCC / C++ Linker, ajoutez :
    - pthread, dl, SDL2, SDL2_gfx dans la fenêtre *libraries*. Il s'agit de la liste des librairies dont le programme aura besoin.

2. Description des classes fournies

La documentation des classes qui vous sont fournies se trouve dans le répertoire /share/esir1/prog/tp12/doc. Il s'agit d'une documentation html générée automatiquement en utilisant l'outil doxygen. Il faudra vous y référer durant le déroulement de ce TP pour obtenir une description des fonctionnalités proposées par les classes fournies.

a. Gestion du temps

Pour pouvoir réaliser la simulation, nous avons besoin de gérer le temps. Ce dernier est géré par l'intermédiaire de la classe *Timer* (fichier « Timer.h »). Dans la simulation, le temps est exprimé en secondes. La classe *Timer* propose les méthodes de classe (indépendantes de l'instance) suivantes :

- `static float time()` : récupération du temps absolu depuis le lancement de l'application (en secondes).

- `static void update(float dt)` : mise à jour de la date courante en fournissant le temps écoulé depuis le dernier appel à update (appelé depuis le programme principal). Vous n'aurez normalement jamais à appeler cette méthode, cela est fait dans le programme principal fourni.
- `static float dt()` : récupération du temps écoulé depuis le dernier appel à update

b. Les vecteurs

Pour positionner et déplacer des entités ou encore calculer des directions, nous utilisons la classe suivante :

```
template <class Scalar> class Vector2
```

Il s'agit d'une classe décrivant un vecteur à 2 dimensions, générique du type de scalaire. Cette classe est définie dans le fichier « Vecteur2.h ». Elle propose les opérateurs classiques tels que ceux que vous avez pu programmer dans le TP vecteurs (+, -, *) ainsi que quelques méthodes (rotation, calcul de norme etc…). Référez-vous à la documentation html fournie.

c. L'environnement et les entités situées

La classe *Environment* (fichier « Environment.h ») permet de représenter l'environnement dans lequel les fourmis vont évoluer. Cette classe possède une classe interne nommée *LocalizedEntity* représentant une entité située (i.e. localisée) dans l'environnement. Il s'agit donc de la classe dont chaque entité pouvant être placée dans l'environnement doit **hériter**. Elle possède les fonctionnalités suivantes :

- Construction à partir d'un pointeur sur l'environnement, de la position de l'entité et du rayon de l'entité (pour des raisons de simplicité, toutes les entités sont représentées par des cercles).
- Consultation et modification de la position de l'entité (méthodes *getPosition*, *setPosition*).
- Translation de l'entité (méthode translate).
- Consultation / modification du rayon de l'entité (méthodes *getRadius*, *setRadius*).

D'autre part, la classe *LocalizedEntity* permet de gérer la perception de l'entité via la méthode suivante :

```
template <class T>

std::vector<T*> LocalizedEntity::perceive(Vector2<float> const & direction, float openingAngle, float extent, float minimumDistance=0.01) const
```

Il s'agit d'une méthode permettant de percevoir des entités suivant leur type. Le paramètre *direction* correspond à la direction vers laquelle l'entité fait face, *openingAngle* est l'angle d'ouverture de la perception en radians de part et d'autre de la direction, *extent* est la distance maximale de perception et *minimumDistance* est la distance minimale de perception. Le schéma de la Figure 1 : Perception d'une entité, explicite ces paramètres et fournit la forme de l'espace de perception en jaune. Le paramètre générique T correspond à la classe des éléments à percevoir qui doivent hériter de *LocalizedEntity*. Supposons que la classe *Food* hérite de *LocalizedEntity* et que nous souhaitions percevoir les instances de cette classe à partir d'une classe héritant aussi de *LocalizedEntity*, il faut alors faire l'appel suivant :

```
std::vector<Food*> perceivedFood = perceive<Food>(direction, perceptionAngle, extent);
```

Le tableau *perceivedFood* contiendra alors les pointeurs sur les instances de la classe *Food* perçues par l'entité.

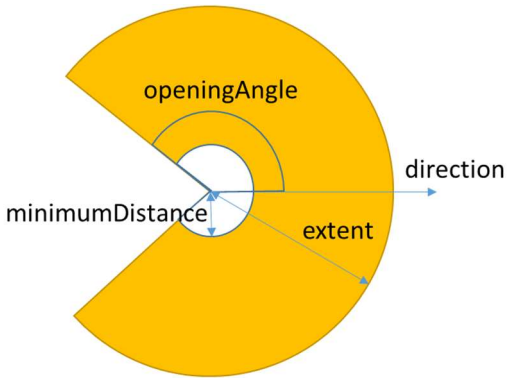

Figure 1 : Perception d'une entité : la zone de perception est peinte en jaune.

d. Le rendu graphique

La classe *Renderer* permet de réaliser des affichages graphiques dans une fenêtre de rendu. Cette classe utilise un patron de conception de type singleton, autrement dit, il n'existe qu'une seule instance de cette classe au sein de l'application. Cette instance peut être récupérée via la méthode de classe `static Renderer * getInstance()`.

Cette classe propose une classe interne *Color* permettant de gérer des couleurs au format RGBA (rouge, vert, bleu, alpha) dans lequel la composante alpha correspond à la transparence. Chaque composante de couleur est dans l'intervalle [0 ;255]. Pour la composante alpha, 0 correspond à transparent, 255 à opaque.

Cette classe met à disposition les méthodes suivantes pour réaliser un affichage graphique :

- `void drawPixel(Vector2<float> const & coordinates, const Color & color)` : méthode permettant d'afficher un pixel à la position coordinates et de la couleur color.
- `void drawCircle(Vector2<float> const & center, float radius, const Color & color)` : méthode permettant d'afficher un cercle dont le centre est à la position center et dont le rayon est radius.
- `void drawString(Vector2<float> const & position, const ::std::string & str, const Color & color)` : affiche une chaine de caractères à la position fournie.

e. Les classes de la STL

Dans ce TP, nous allons utiliser deux classes de la STL : *std::set* qui permet de gérer des ensembles d'éléments et *std::vector* qui représente un tableau de taille variable. Vous pouvez accéder à la description de ces classes via le lien suivant : http://en.cppreference.com/w/. La description des classes sera alors accessible via le lien *Container Library*.

3. Première étape : création d'un agent

Dans cette partie, nous allons créer une classe correspondant à un agent. Un agent correspond à une entité localisée dans l'environnement, dotée d'un comportement.

Question 3.1 : Créez la classe *Agent* qui devra hériter de la classe *Environment::LocalizedEntity*. Cet agent disposera d'un constructeur prenant en paramètre un pointeur sur une instance de la classe

Environnement, sa position initiale ainsi que son rayon qui par défaut prendra la valeur *Environment::LocalizedEntity::defaultRadius()*.

Question 3.2 : Cette classe devra posséder une méthode `void update()` qui sera utilisée pour implémenter, par héritage, le comportement de l'agent. Cette méthode sera appelée à intervalles de temps réguliers afin de permettre à l'agent de réaliser des actions telles qu'un déplacement, ramasser de la nourriture etc… Quelle est la propriété de cette méthode ? Déclarez cette méthode.

Nous allons considérer qu'un agent peut se trouver dans deux états :

- *running* : l'agent est en cours d'exécution.
- *destroy* : l'agent a terminé son exécution et peut être détruit.

Pour décrire ces états nous allons utiliser un type énuméré définit comme suit à l'intérieur de la classe *Agent* : `typedef enum { running, destroy } Status.`

Question 3.3 : Modifiez la classe *Agent*, de manière à intégrer la définition du type enuméré comme ci-dessus. D'autre part, programmez les méthodes *getStatus* et *setStatus* permettant respectivement de consulter et modifier l'état de l'agent. Par défaut un agent, dès sa création doit être dans l'état *running*.

Pour permettre de facilement simuler les agents, nous allons ajouter un attribut et une méthode de classe dans la classe Agent.

Question 3.4 : Ajoutez un attribut (privé) de classe (*static*) de type *std::set<Agent*>* dont le rôle est de contenir un pointeur vers tous les agents créés au cours de la simulation. Modifiez le code de l'agent afin que chaque agent créé soit automatiquement référencé dans cette structure de données.

Question 3.5 : Ajoutez une méthode (publique) de classe `static void simulate()` dont le rôle est le suivant. Cette méthode parcourt tous les agents. Lorsqu'un agent est dans l'état *running*, sa méthode *update()* est appelée. Lorsqu'un agent est dans l'état *destroy*, il doit être détruit par un appel à *delete* sur son pointeur et sortir de l'ensemble des agents actifs. **Attention** à la modification de l'ensemble durant le parcours, cela risque d'invalider votre itérateur si vous ne prenez pas de précaution. D'autre part, ajoutez une méthode `static void finalize()` dont le rôle est de détruire tous les agents encore actifs en fin de la simulation.

Pour tester notre classe nous allons programmer un premier agent très simple, il s'agit d'un agent qui correspond à de la nourriture.

Question 3.6 : Créez une classe *Food* héritant de la classe *Agent*. Cette classe devra posséder un constructeur prenant les informations nécessaires à la construction d'un agent (environnement, position) ainsi que la quantité de nourriture. Le rayon de l'agent est dépendant de la quantité de nourriture et est calculé en utilisant la fonction *`float circleRadius(float quantity)`* (définie dans l'espace de nommage *MathUtils* et dans le fichier *MathUtils.h*).

Question 3.7 : La classe *Food* devra permettre de consulter la quantité de nourriture restante (*`float getFoodQuantity() const`*) et permettre de récolter de la nourriture (*`float collectFood(float quantity)`*). Le paramètre *quantity* correspond à la quantité de nourriture demandée, la valeur de retour est la quantité de nourriture accordée. Attention la méthode *collectFood* ne pourra jamais renvoyer plus de nourriture qu'il n'en reste à disposition. Programmez ces fonctionnalités.

Question 3.8 : Le comportement de l'agent *Food* est le suivant. A chaque appel à la méthode *update*, l'agent doit vérifier si la quantité de nourriture a changé. Si c'est le cas, le rayon de l'agent doit être

mis à jour (Cf. classe *LocalizedEntity*). De plus, l'agent *Food* doit s'afficher sous la forme d'un cercle (Cf. classe *Renderer*) de couleur R : 154, G : 235, B : 38, A : 255 ; le rayon de ce cercle est le rayon de l'agent. Enfin, si la quantité de nourriture restante est égale à 0, l'agent doit passer en état *destroy* de manière à être détruit.

Question 3.9 : Pour tester la simulation, ouvrez le fichier main.cpp et complétez le fonction `void onSimulate()` (fonction appelée à chaque affichage) par un appel à la méthode de classe *simulate* de la classe *Agent*. Ajoutez aussi un appel (en fin de la fonction main) à la méthode *finalize* de la classe Agent afin de libérer la mémoire allouée pour les agents en fin de simulation. D'autre part, complétez la fonction `void onKeyPressed(char key, Environment * environment)` (fonction appelée à chaque appui sur une touche) de manière à créer une instance de la classe *Food* lorsque la touche 'f' est enfoncée. La quantité de nourriture associée à chaque instance la classe *Food* sera tirée au hasard et la quantité sera dans l'intervalle [200, 2000] (la fonction *MathUtils::random(min, max)* pourra être utilisée pour effectuer un tirage aléatoire). D'autre part, la méthode *randomPosition()* de la classe *Environment* vous permet de tirer une position aléatoire. Attention, cette instance de classe doit être créée via une allocation dynamique (comme tous les agents de la simulation). Enfin, ajoutez le fait qu'en appuyant sur la touche 'd' une instance de nourriture disparaisse (vous pourrez utiliser la méthode *Environment::getAllInstancesOf<T>()* pour récupérer toutes les instances de la classe Food référencées dans l'environnement).

Vous pouvez désormais lancer la simulation. Lorsque vous appuyez sur la touche 'f', des cercles verts devraient apparaitre à l'écran. Ces cercles correspondent à de la nourriture présente dans l'environnement. De même, lorsque vous appuyez sur la touche 'd', un cercle vert devrait disparaitre.

Nous allons maintenant ajouter une classe représentant une fourmilière. Créez une classe nommée *Anthill* héritant de la classe *Agent*. La fourmilière est un agent localisé ayant un rayon de 10. A chaque appel à la méthode *update()*, un cercle de rayon 10 et de couleur R : 0, G : 0, B : 255, A : 255 doit être affiché à l'écran. La fourmilière est aussi un dépôt de nourriture. Elle doit posséder la méthode *void depositFood(float quantity)* permettant aux fourmis de déposer la nourriture qu'elles transportent dans la fourmilière.

Question 3.10 : Ecrivez la classe *Anthill* et complétez le programme principal pour créer une fourmilière lorsque l'on appuie sur la touche 'a'.

4. Nos premières fourmis

Nous allons maintenant créer notre première classe de fourmi. Cette classe nommée *AntBase* devra hériter de la classe *Agent*. Toutes les fourmis ont différentes choses en commun.

- Elles sont des agents.
- Elles sont membres d'une fourmilière.
- Elles ont une vitesse de déplacement dont la valeur par défaut est de 1 $cm.s^{-1}$
- Elles ont une direction de déplacement représentée par un vecteur en 2 dimensions dont la norme 1.
- Elles ont une durée de vie dans l'intervalle [1000 ; 2500] secondes.
- Elles peuvent transporter une certaine quantité de nourriture.
- Une fourmi peut avancer suivant sa direction de déplacement. Dans ce cas, la fourmi se translate d'un vecteur d*v*Timer::dt() où d est le vecteur direction de déplacement, v la vitesse de déplacement et Timer::dt() fournit le temps écoulé depuis le dernier appel à update().

- Une fourmi peut tourner d'un certain angle (exprimé en radians). Cela correspond à appliquer une rotation sur le vecteur fournissant la direction de déplacement.
- Une fourmi peut faire demi-tour ; cela correspond à inverser la direction de déplacement.
- Une fourmi peut s'orienter vers une cible dont la position est fournie.
- Une fourmi peut déposer la nourriture qu'elle transporte.
- Une fourmi peut récolter de la nourriture lorsqu'elle se trouve dessus et peut transporter un maximum de 5 unités de nourriture.
- Une fourmi peut percevoir avec un angle d'ouverture de pi/2 radians de la nourriture à une distance maximale de 3 cm.
- Lorsque la fourmi a atteint sa durée de vie, elle meurt.
- Une fourmi s'affiche sous la forme d'un point blanc (R :255, V :255, B :255, A :255) tracé à sa position ou d'un point vert (128,255,128,255) si elle transporte de la nourriture.

Question 4.1 : Programmez la classe *AntBase*.

Nous allons programmer une classe de fourmi « idiote » dont le comportement consiste à explorer l'environnement au hasard. Si la fourmi rencontre de la nourriture, elle en ramasse une partie (5 unités maximum) puis rentre en ligne droite à la fourmilière pour la déposer.

Question 4.2 : Créez une classe *SillyAnt* héritant de la classe *AntBase* et définissant le comportement décrit ci-dessus. Cette fourmi, lors de sa création, tirera au hasard une direction de déplacement (voir méthode *Vector2::random()*). Pour la partie exploration au hasard, vous pourrez considérer que la fourmi peut, à chaque appel à la méthode update tourner au hasard d'un angle compris dans l'intervalle [-pi/10* Timer::dt(); pi/10* Timer::dt()] puis avancer. Pour le retour au nid, la fourmi se tourne vers le nid puis avance jusqu'à le trouver. Pour vérifier que la fourmi se trouve sur de la nourriture ou sur le nid, vous pouvez utiliser la méthode *LocalizedEntity::perceive<T>()* permettant de percevoir ce qui se trouve sur la portion de terrain sur laquelle la fourni se tient.

Question 4.3 : Complétez votre programme principal pour créer une cinquantaine de fourmis de type *SillyAnt* en même temps que la création du nid puis lancez la simulation en créant de la nourriture et un nid pour tester votre fourmi et valider son comportement.

5. Vers des fourmis au comportement… plus fourmi…

Comme vous le savez certainement, les fourmis, lorsqu'elles se déplacent, déposent des phéromones. Ces phéromones leur permettent d'une part de connaitre le chemin vers leur nid (bien qu'elles n'utilisent pas toujours cette propriété) mais aussi et surtout de marquer le chemin vers la nourriture. De manière à modéliser ce comportement, nous allons programmer la classe de phéromones. Les phéromones ont la propriété de s'évaporer au cours du temps.

Question 5.1 : Créez une classe *Pheromone* qui hérite de la classe *Agent*. Cette classe possèdera un constructeur prenant en paramètre la quantité de phéromones déposée, possèdera des méthodes *getQuantity()* et *addQuantity(float q)* permettant respectivement de consulter la quantité de phéromones et d'ajouter une certaine quantité de phéromones. Le comportement par défaut de la phéromone est le suivant : à chaque appel à *update()* elle s'évapore et la quantité (q) diminue de 0.01*q*Timer::dt(). Lorsque la quantité devient inférieure à 0.01, la phéromone disparait. De plus, elle s'affiche sous la forme d'un point de couleur (R : 0, G : 128, B : 128), la composante A prend la valeur minimale entre la quantité de phéromones et 255 (voir std::min).

Question 5.2 : Créez la classe *AntBasePheromone* héritant de *AntBase*. Cette classe ajoute une méthode *void putPheromone(float q)* qui permet à la fourmi de déposer des phéromones à chaque

déplacement. Cette méthode est un peu particulière. Si la fourmi se trouve sur une partie de l'environnement sur laquelle des phéromones sont déjà présentes (voir *LocalizedEntity::perceive<T>()*), elle ajoute la quantité de phéromones aux phéromones déjà présentes. Dans le cas contraire, elle crée une instance de *Pheromone* pour déposer les phéromones à sa position courante.

Question 5.3 : Complétez la classe *AntBasePheromone* par la méthode *Pheromone * choosePheromone() const* dont le rôle est (1) de percevoir les phéromones à une distance maximale de 8 cm, (2) de choisir au hasard une phéromone à suivre en fonction de sa concentration. Pour ce faire vous pourrez vous aider la fonction *int MathUtils::randomChoose(const std::vector<float> & weights)* permettant de tirer au hasard un indice dans la table avec une probabilité dépendant du poids associé à l'indice. La fonction *choosePheromone()* renvoie *nullptr* si aucune phéromone n'est perçue.

Question 5.4 : Créez une nouvelle classe *Ant* dont le comportement est le suivant :

1. Si la fourmi ne transporte pas de nourriture et se trouve sur de la nourriture, elle en ramasse.
2. Si la fourmi ne transporte pas de nourriture mais perçoit la nourriture, elle se déplace vers la nourriture.
3. Si la fourmi ne transporte pas de nourriture et perçoit des phéromones, elle se déplace dans la direction de la phéromone perçue (utilisation de *choosePheromone*)
4. Si la fourmi ne transporte pas de nourriture et ne perçoit pas de phéromones, elle se déplace au hasard suivant les conditions de la question 4.2.
5. Si la fourmi transporte de la nourriture et se trouve sur son nid, elle dépose la nourriture et fait demi-tour.
6. Si la fourmi transporte de la nourriture et ne se trouve pas sur son nid, elle retourne au nid. Pour ce faire (1) elle se tourne vers le nid, (2) si elle perçoit des phéromones dans la direction du nid, elle les suit (*choosePheromone*), si elle ne perçoit pas de phéromones, elle se dirige vers le nid en ligne droite.

Dans tous les cas :

- Lorsque la fourmi se déplace en transportant de la nourriture, elle dépose 100 unités de phéromones sur son chemin.
- Lorsque la fourmi se déplace sans transporter de nourriture, elle dépose 10 unités de phéromones.

Question 5.5 : modifiez votre programme principal afin de créer des instances de *Ant* au lieu de *SillyAnt* et observez et validez le comportement produit.

6. Retour sur le comportement

Le comportement de la fourmi défini dans la question 5.4 correspond à un système à base de règles de type *si condition alors action*. Nous allons travailler sur un modèle permettant de décrire ces comportements de façon plus modulaire.

Question 6.1 : Créez une classe abstraite *AbstractRule* comportant deux méthodes abstraites : *bool condition()* et *void action()*. Cette classe sera la classe mère de toutes les règles.

Question 6.2 : Créez une classe *OrRule* qui regroupe plusieurs règles fournies à la création de l'instance dans un ordre décidé par l'utilisateur. Sa condition est vraie si l'une des conditions des règles est vraie. L'action associée est alors l'action associée à la première règle dont la condition est vraie.

Question 6.3 : Créez une classe *AbstractAntRule* représentant une règle qui possède la propriété de conserver un pointeur vers la fourmi sur laquelle la règle sera appliquée.

Question 6.4 : Créez une classe *AntWithRules* héritant de *AntBasePheromone* possédant des classes internes (héritant de *AbstractAntRule*) correspondant aux règles de la question 5.4.

Question 6.5 : Sur la base des règles que vous venez de programmer, reprogrammez le comportement de la fourmi *AntWithRules* tel que décrit dans la question 5.4 mais en utilisant le système que vous venez de mettre en place.

7. Extensions possibles

Vous pouvez programmer de nombreuses extensions de ce TP. Voici quelques propositions :

- La fourmilière actuellement utilisée ne gère pas la naissance des fourmis. Vous pouvez programmer une nouvelle fourmilière de manière à ce que cette dernière, lorsque la quantité de nourriture le permet crée, une nouvelle fourmi. Afin de créer cette fourmi, cette nouvelle fourmilière pourra prendre en paramètre une instance d'une classe dont le rôle sera de créer des instances de fourmis pour la fourmilière (un patron de conception de type *Factory*).
- Actuellement, les fourmis ne combattent pas pour leur territoire. Vous pouvez ajouter un système de combat de manière à ce que lorsqu'une fourmi rencontre une fourmi d'une autre fourmilière, elle la combatte en émettant des phéromones d'alerte attirant les fourmis alentours.
- Et bien d'autres… discutez avec votre encadrant de TP ☺