Le Système Multi-Agents
et
La simulation de ville OpenCity

Duong-Khang NGUYEN
Victor STINNER

A l'attention de:
M. Vincent HILAIRE
Le 10 janvier, 2006

Sommaire

Description du projet

Dans le cadre du jeu de simulation de ville OpenCity, nous avons eu l'idée d'utiliser un Système Multi-Agents (SMA) afin de créer une intelligence artificielle entre les différents véhicules présents dans le jeu. Ce système est indépendant du jeu de simulation. L'intégration dans la simulation sera facilitée par l'utilisation des techniques de spécialisation dans le développement de logiciel orienté objet.

Le jeu de simulation OpenCity est un jeu de gestion de ville en 3 dimensions et en temps réel. C'est pourquoi la simulation est relativement gourmande en temps processeur. Une ville dans OpenCity est composée de plusieurs *zones*. Il y a trois types de zones possibles: résidentielle, commerciale, et industrielle. Et finalement, ces zones sont desservies par des routes.

Les agents policiers et les agents voleurs évoluent sur des routes construites par les joueurs et ne peuvent pas sortir de ces routes.

Objectifs du projet

Étant donné l'objectif de l'UV IA54, nous avons développé un noyau relativement simple afin de pouvoir concentrer nos efforts sur la conception et le développement du comportement des agents ainsi que leur interactions avec l'environnement.

Noyau

Le SMA a les caractéristiques suivantes: ouvert, hétérogène et composé d'agents réactifs. L'ensemble du SMA sera géré par un seul thread. L'ordonnancement des agents consiste à exécuter le traitement propre à chaque agent dans une boucle et dans un l'ordre de création des agents.

Environnement

L'environnement a la connaissance des positions courantes des agents disposés sur la carte. Il est chargé de gérer les déplacements des agents et faire le lien avec les données du simulateur. Le calcul du plus court chemin est effectué via l'environnement qui est basé sur l'algorithme de Dijkstra amélioré avec une heuristique. Nous n'allons pas détailler cet algorithme car il ne fait pas partie du projet.

Agent

Une classe « *Agent* » générique (cf. Diagramme de classe ci-dessous) a été implémentée. Chaque agent a un rôle précis qui conditionne son comportement. Un rôle est donc assimilé à un type d'agent. Dans notre cas, nous avons deux types (rôles) d'agent précis: policier et voleur. La perception d'un agent est limitée dans l'environnement. Lorsqu'un agent envoie un message, ce message n'est pris en compte que par des agents qui lui sont proches.

Agent policier

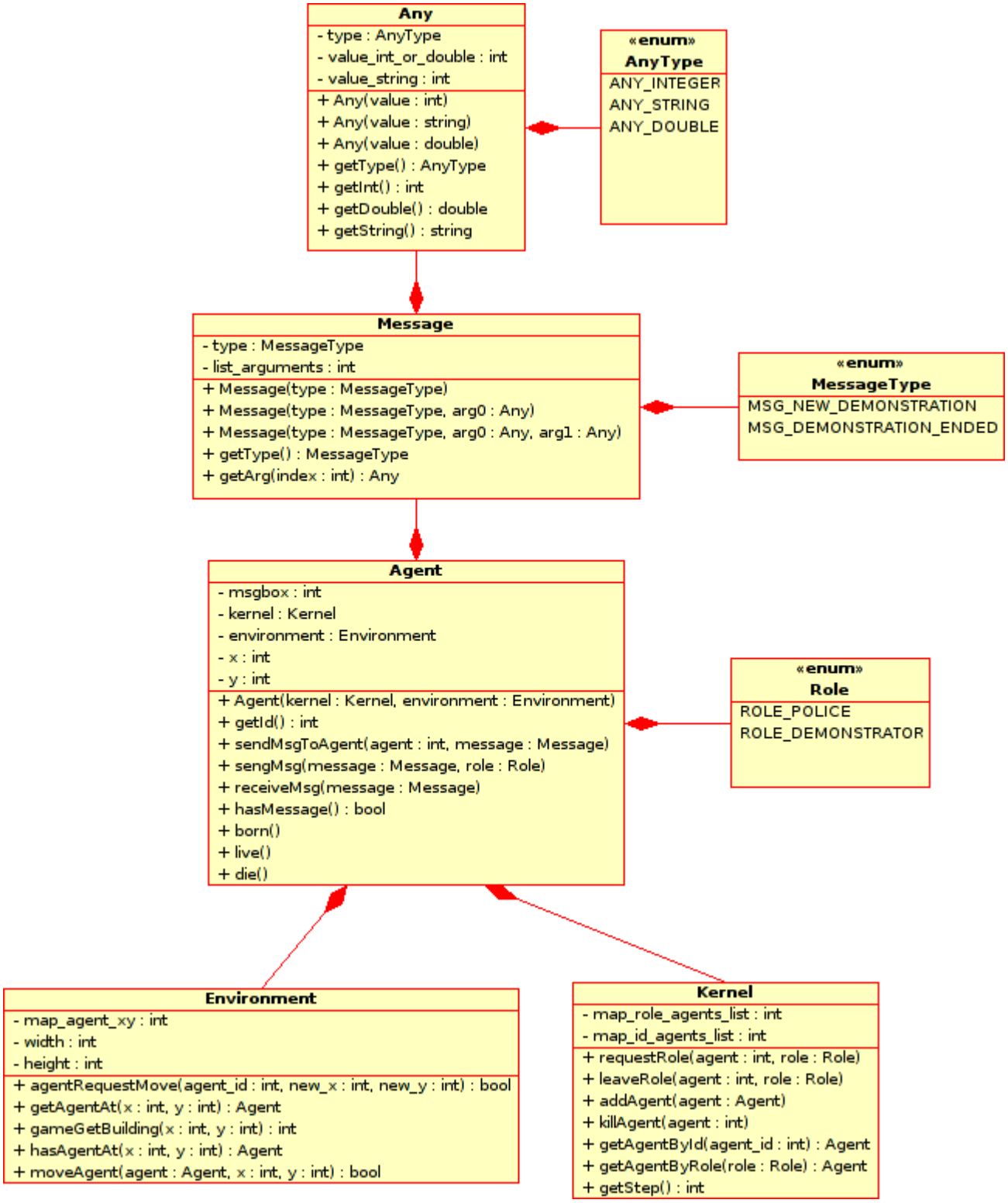
Par défaut, l'agent policier *patrouille*. Lorsqu'il rencontre un agent *voleur*, il demande du renfort afin d'*interpeller* cet agent voleur. Si l'agent voleur est arrêté et enlevé de l'environnement quand il est bloqué par au moins un agent policier et il n'arrive plus à se déplacement.

Agent voleur

Par défaut, un agent voleur *fuit* d'un point à un autre le plus rapidement possible en évitant les agents policiers. Il n'y a pas de collaboration entre les agents voleurs.

Image 1: Arrestation d'un agent voleur par les policiers

Diagramme de classe



Implémentation du SMA

L'implémentation du système multi-agents est représentée ci-dessous. Les *agents* interagissent avec le *noyau* en lui envoyant des messages et vice versa. Cependant, par souci de performance, nous n'avons pas implémenté le même système de communication entre les *agents* et *l'environnement*. En effet les agents appellent directement des fonctions implémentées dans l'environnement.

Nous avons prévu que le SMA puisse fonctionner en concurrence avec le jeu, c'est-à-dire que chacun tourne dans un thread séparé. Cependant, nous n'avons pas eu le temps nécessaire pour activer cette fonctionnalité.

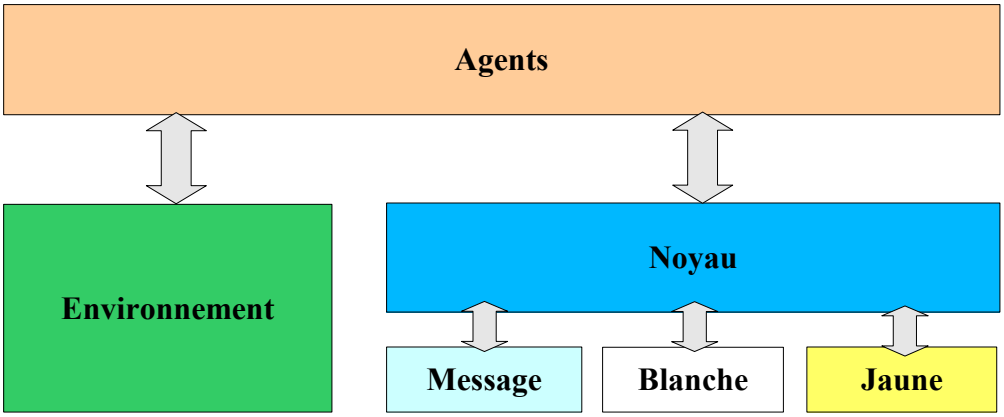


Illustration 1: Les composants du SMA

Noyau

Pour le noyau du SMA, nous avons choisi un modèle volontairement très simple. Le problème était de ne pas consommer trop de ressources car le jeu OpenCity en consomme déjà beaucoup, et le jeu doit répondre très rapidement aux actions de l'utilisateur. Nous avons donc à chaque fois réfléchi aux algorithmes les plus efficaces pour éviter les goulots d'étranglements.

Le noyau a les tâches suivantes :

- Annuaire « pages blanches » (*white pages*) qui référence les agents selon leur identifiant (nombre entier)
- Annuaire « pages jaunes » (*yellow pages*) qui référence les agents selon les rôles auxquels ils sont enregistrés
- Enregistrer un agent dans les deux annuaires
- Tuer un agent (le supprimer des annuaires)
- Envoi d'un message à un agent (selon son identifiant)

- Envoi d'un message à un rôle
- Gestion du temps de simulation
- Ordonnancement des agents
- Appel de la fonction live() de chaque agent

Pour des questions de performances, nous avons utilisés des tables de hachage (*std::map* en C++) pour les annuaires. Les pages blanches associent un identifiant à un pointeur sur un agent, et les pages jaunes associent un rôle à une liste chaînée (*std::list* en C++) de pointeurs sur des agents. Il aurait été plus propre que les pages jaunes contiennent des identifiants plutôt que des pointeurs, mais nous avions toujours en tête le problème des performances.

Pour tuer un agent, nous avons créé une liste chaînée des agents qui doivent être détruits à la fin du pas de simulation. Les agents ne doivent pas être détruits directement durant un pas de la simulation, sinon on risque d'avoir des incohérences dans le système.

Lorsqu'un agent souhaite disparaître du système, il demande au noyau de le tuer en lui envoyant un message. Le noyau enregistre sa demande et stocke son adresse dans la liste des agents mourants. Au pas de simulation suivant, le noyau examine la liste des agents mourants et leur envoie un message de confirmation de la demande de disparition avant d'exécuter les méthodes *die()* des agents contenus dans cette liste. C'est seulement après ces traitements que les agents mourants sont enlevés du système par le noyau.

L'ordonnancement est relativement simple: les agents sont exécutés dans l'ordre de leur enregistrement auprès du noyau via la méthode *live()*.

Environnement

La communication entre le système multi-agents et le jeu est assurée par la classe *environnement*. Cette dernière utilise des fonctionnalités fournies par la classe *PathFinder* du jeu afin de déterminer le chemin le plus court entre deux points de la carte. De ce fait, nous pouvons dire que tous agents connaissent l'environnement dans lequel ils évoluent et ce ne sont plus des agents purement réactifs.

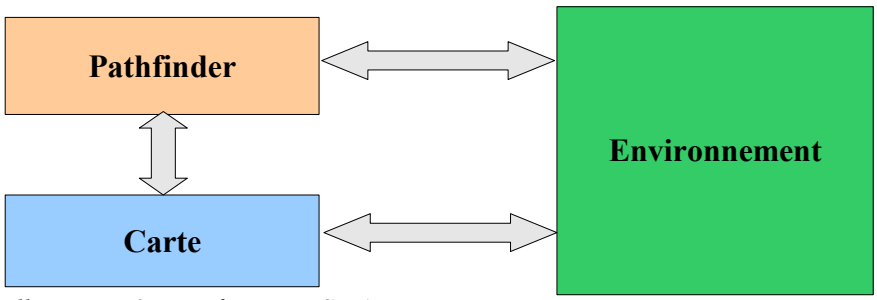


Illustration 2: Interface jeu - SMA

L'environnement gère à la fois la carte sur laquelle évoluent les agents, et les interactions avec le jeu OpenCity. L'environnement est affiché à chaque itération dans le jeu. Lors de l'affichage de l'environnement, la position de chaque agent est interpolée afin d'obtenir une meilleure fluidité et donne une illusion des agents animés. La classe *environnement* permet d'accéder à l'état d'une case du jeu et de savoir si elle contient une route, un bâtiment, ou n'est pas déjà occupée un autre agent.

Étant donné que l'environnement du jeu évolue en temps réel (une route peut être détruite, remplacée par un bâtiment, ou une autre route peut être construite), nous avons choisi que les actions des agents sur l'environnement étaient effectuées dès que l'agent en fait la demande (et non pas une fois que tous les agents ont été exécutés).

Dans notre environnement du système multi-agents, il ne peut y avoir qu'un seul agent par case. Par contre, la taille de cet environnement est la même que celui du jeu (et il y a un correspondance au niveau des coordonnées). L'environnement du SMA peut être vue comme une couche (*layer*) au-dessus de la carte du jeu. Lorsqu'un agent veut se déplacer, il doit en faire la demande à l'environnement qui peut lui refuser son déplacement, s'il y a déjà un autre agent aux coordonnées demandées par exemple.

Étant son rôle central dans le système, l'environnement devient vite un goulot d'étranglement lorsque le nombre d'agents devient important.

Agents et comportements

Comme c'est décrit dans le schéma UML précédent, les agents dans notre SMA ont les propriétés suivantes :

- Il peut envoyer et recevoir des messages
- Il a un identifiant unique (nombre entier)
- Il a trois méthodes principales :
  - *born()* : Appelée lors de la création de l'agent
  - *live()* : Exécution du comportement de l'agent
  - *die()* : Appelée lors de la destruction de l'agent
- Il stocke les messages reçus car ils ne sont traités qu'une fois la fonction *live()* appelée

Lors de la création d'un agent, la méthode *born()* s'occupe d'enregistrer l'agent auprès du noyau pour chacun de ses rôles. Elle va également placer l'agent dans l'environnement.

Agent policier

Le comportement du policier est celui qui est le plus évolué dans le système. Un policier a plusieurs états et est capable de collaborer avec d'autres policiers. Lorsqu'un policier est créé, il est dans l'état *Patrouille*. Sa vision est limité à une longueur de 10 cases.

État : Patrouille

Cet état est l'état par défaut. Le policier va se déplacer aléatoirement (essaye de tourner à droite ou à gauche 60% du temps) dans l'environnement à la recherche de voleurs. La recherche se fait dans toutes les directions : devant, à droite, à gauche, ou derrière.

Lorsqu'un policier aperçoit un voleur, il va le prendre en chasse, c'est-à-dire qu'il passe dans l'état *Poursuite* et émet une alerte à destination des autres policiers à proximité.

État : Poursuite

Dans cet état, un policier se dirige dans la direction d'un voleur en avançant toujours tout droit. Le message d'alerte est répété toutes les 5 itérations pour attirer d'autres policiers qui arrivent dans le secteur. Nous n'avons pas voulu le faire émettre un message d'alerte à chaque itération afin d'éviter d'inonder les système de messages inutiles.

Si le voleur n'est plus visible, c'est-à-dire qu'il n'est ni devant, ni à droite, ni à gauche, le policier considère qu'il l'a perdu de vue et passe dans l'état *Poursuite (perdue)*.

État : Poursuite (perdue)

Lorsqu'un policier a perdu de vue un voleur, il va tenter de le retrouver en se déplaçant plutôt vers l'avant. Il a un quand même un déplacement aléatoire, mais tente de tourner seulement 30% du temps.

Le policier va regarder dans toutes les directions (devant, droite, gauche, derrière). S'il aperçoit un voleur, il va le prendre en chasse, avancer dans sa direction et alerter les autres policiers à proximité.

Sinon, il va avancer d'une case et à nouveau regarder autour de lui. À ce moment là, s'il aperçoit un voleur, il va le prendre en chasse et alerter les autres policiers à proximité.

État : Collaboration

La première chose que fait un policier dans sa méthode *live()* est de traiter les messages reçus. Seuls les messages émis par des agents à proximité sont pris en compte. Si le message reçu est « voleur aperçu en (x,y) », alors qu'il n'est pas déjà dans l'état *Poursuite* ou *Collaboration*, il va se diriger vers ce point en suivant un chemin optimal.

L'agent va alors demander à l'environnement de calculer le plus court chemin pour aller en ce point en utilisant un algorithme de Dijkstra. Toujours pour des questions de performance, nous limitons la recherche à une longueur maximale de 15 cases à vol d'oiseau. Il se peut donc que le chemin retourné par l'environnement ne soit pas le plus court chemin car l'algorithme n'est pas exécuté jusqu'au bout. En

effet il s'agit d'un algorithme A*, c'est-à-dire le programme essaie de construire un chemin optimal en choisissant à chaque itération un minimum local.

Cet algorithme de recherché était déjà programmé dans le jeu (dans le cadre de l'UV *AG41*), nous avons choisi de le réutiliser. Pour cette raison, nous ne pouvons pas considérer les agents comme purement réactifs. L'agent accède directement à l'environnement sans passer par des messages.

Durant le parcours du chemin optimal, si l'agent se trouve juste à côté d'un voleur, il le prend en chasse (passe en mode *Poursuite* et envoie du message d'alerte). Si le chemin a été modifié (suppression d'une portion de route dans le jeu), le policier va repasser en mode *Patrouille*.

À la fin du parcours, s'il voit un voleur en face de lui : il le prend en chasse (envoie de l'alerte, etc.). Sinon, il repasse simplement en mode patrouille

Propriétés :

- Traitement prioritaire les messages reçu (sont susceptibles de le faire passer dans l'état *Collaboration*)
- Le passage à l'état *Poursuite* envoie toujours un message d'alerte aux autres policiers à proximité
- Durant la *Poursuite*, un nouveau message d'alerte est émise toutes les 5 itérations
- Avance toujours d'une case, sauf si le policer est en train de bloquer un voleur

Agent voleur

Le comportement des agents de type voleur est moins élaboré. Le voleur se contente de se déplacer « aléatoirement » (tourne 60% du temps) dans l'environnement. S'il est à proximité d'un policier, il va rebrousser chemin.

Par contre, les voleurs ne collaborent pas entre eux. Il aurait été intéressant d'écrire un comportement de dispersion par exemple.

Conclusion

Nous avons testé le jeu dans différentes configurations : peu/beaucoup de policiers, peu/beaucoup de voleurs, petit/moyenne/grande carte. L'algorithme des policiers fonctionne bien car les voleurs sont toujours attrapés rapidement. Par contre, quand il y une cinquantaine ou une centaine de voleurs contre moins d'une dizaine de policiers, cela prend forcément plus de temps bien que les policiers arrivent à arrêter plusieurs voleurs à la fois.

Le jeu fonctionne donc très bien dans l'ensemble, mais nous avons tout de même noté une faiblesse dans le comportement des policiers. En effet, lorsque deux policiers poursuivent chacun un voleur, aucun des deux n'est capable d'abandonner sa poursuite pour aider son coéquipier. Il faut donc qu'ils se croisent par hasard ou encore qu'un des deux policiers perdent de vue le voleur qu'il poursuivait.

Beaucoup de points auraient été intéressants à étudier, mais par manque de temps nous n'avez pu le faire. Il serait intéressant de mesurer, par exemple, l'impact d'un meilleur algorithme d'ordonnancement. Il faut avouer que le développement du SMA en lui-même ainsi que son intégration dans le jeu nous ont tout de même pris une part importante dans la durée totale du projet.