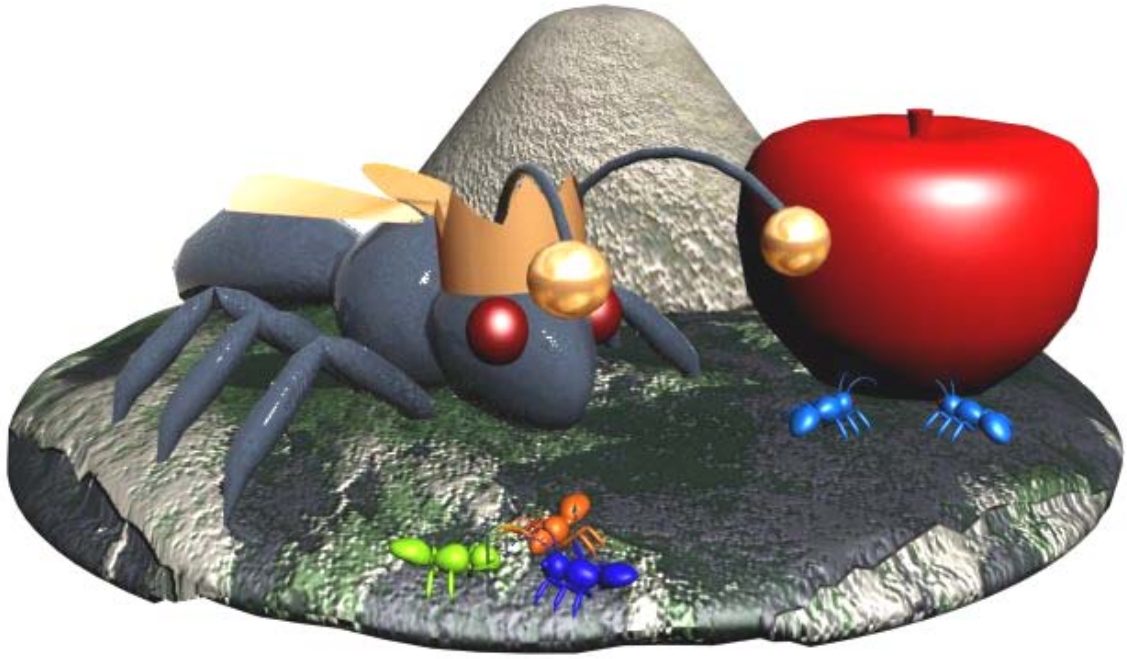
Béatrice Frey et Cyril Crassin **Printemps 2004**

Projet Fourmis



Sommaire

Introduction

L'objectif de ce projet LO43 est de simuler le comportement d'un ensemble de fourmis dans un contexte compétitif sur un terrain parsemé de nourriture et d'obstacles. Nous sommes partis d'un projet de base écris en Java et fourni en TP que nous avons tout d'abord analysé puis nous avons modélisé un certain nombre d'améliorations que nous avons ensuite implémentées.

Notre simulation comporte des fourmis qui peuvent êtres de trois types : Ouvrières, Guerrières et Reine, chaque fourmi appartient à une fourmilière et possède un comportement qui lui est propre.

Notre objectif a été tout au long de ce projet de réaliser un développement générique et réutilisable basé sur une analyse claire de l'existant ainsi qu'une conception UML avancée. Notre problématique principale a également été d'optimiser au maximum l'application pour permettre la simulation d'un très grand nombre d'individus.

1. Analyse de l'existant

1.1 Description générale

L'application fournie proposait une simulation composée de fourmis, de fourmilières, d'obstacles et de nourriture. Les fourmis possédaient un comportement très élémentaire puisqu'elles se contentaient de se déplacer aléatoirement sur le terrain et d'éviter les obstacles.

1.2 Analyse des Classes

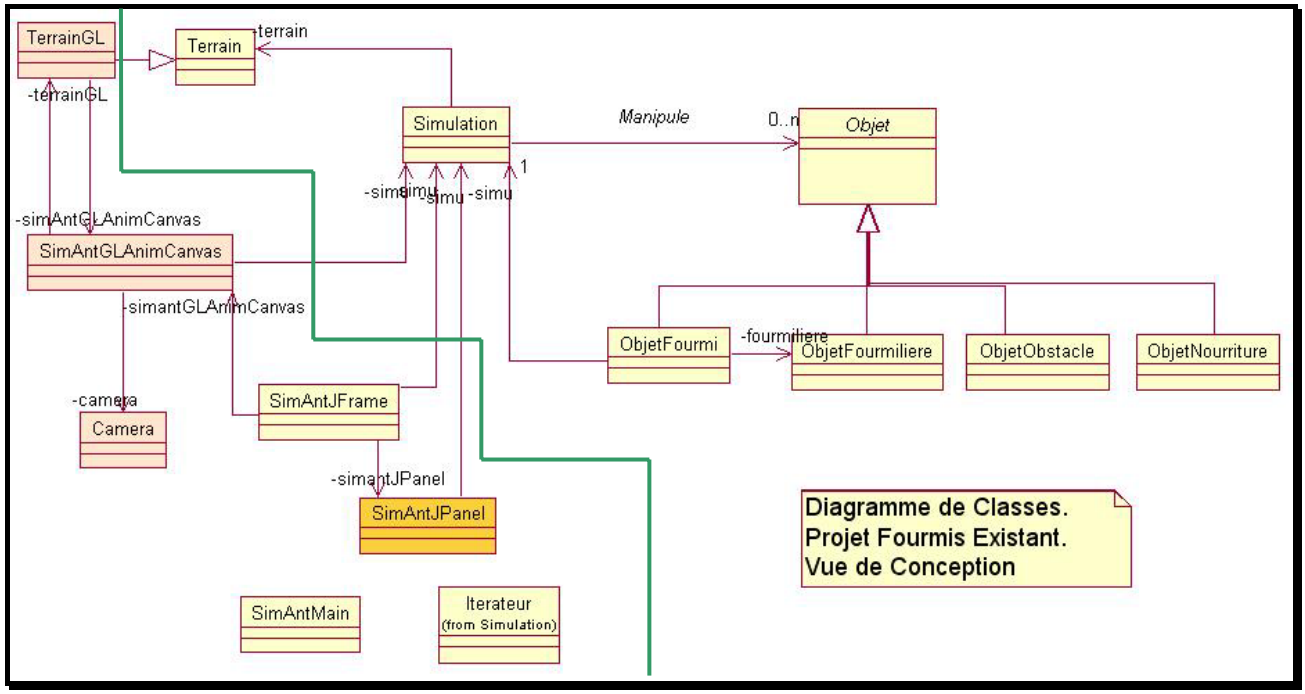


Diagramme de classes : Point de vue conceptuel

Une étude détaillée du programme existant nous a permis de découvrir les différentes classes mises en œuvre ainsi que leurs relations.

La classe centrale du programme se nomme *Simulation* et orchestre le déroulement de la simulation. C'est elle qui crée les différentes entités mises en œuvre (Fourmis, Fourmilières, Obstacles, Nourriture) et déclenche leur comportement sur le terrain.

Toutes les entités intervenant dans la simulation héritent d'une classe commune Objet. Chaque instance d'Objet sera localisé sur le terrain grâce à ses coordonnées, possédera une taille et une orientation.

Les autres classes mises en oeuvre permettent d'effectuer l'affichage (séparés par la ligne verte sur le diagramme). Deux types d'affichage sont disponibles : L'affichage standard en 2D géré par la classe *SimAntJPanel* et l'affichage 3D géré par les classes en rosé sur le diagramme.

1.3 Déroulement global de l'application

Le programme s'articule autour de deux *threads*, l'un gérant l'affichage et l'autre la simulation. L'affichage est ainsi désynchronisé de la simulation ce qui permet son rafraîchissement régulier. L'affichage peut être 2D ou 3D suivant le mode choisi au lancement. L'affichage 3D a été réalisé via un *wrapper* OpenGL pour java appelé gl4java ce qui assure un affichage 3D rapide (car accéléré matériellement) et simple à programmer.

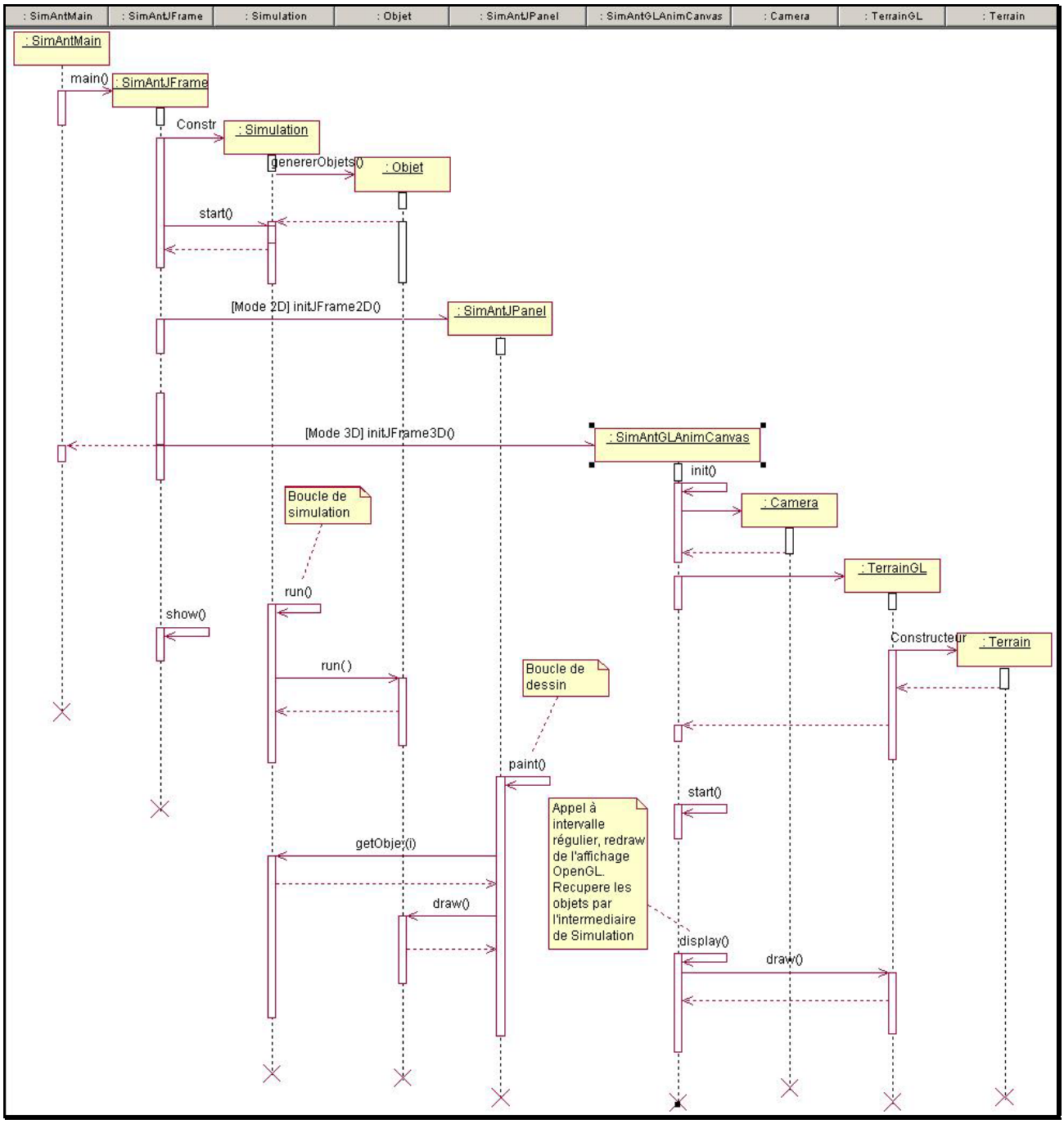


Diagramme de séquence : point de vue de l'implémentation

La première étape du programme consiste à créer une instance de *SimAntJFrame*. Cette classe crée une instance de *Simulation* et lance son exécution dans un nouveau Thread par l'appel de la méthode *start()* de *Simulation*. *SimAntJFrame* se charge ensuite d'initialiser l'affichage demandé par l'utilisateur, il peut s'agir d'un mode 2D ou 3D (OpenGL).

L'objet simulation lors de sa création génère tous les éléments intervenant dans la simulation et les place dans son vecteur d'*Objet*. Chaque objet (descendant d'*Objet*) reçoit les informations qui lui sont nécessaires par l'intermédiaire de son constructeur et les utilise pour initialiser ses paramètres propres.

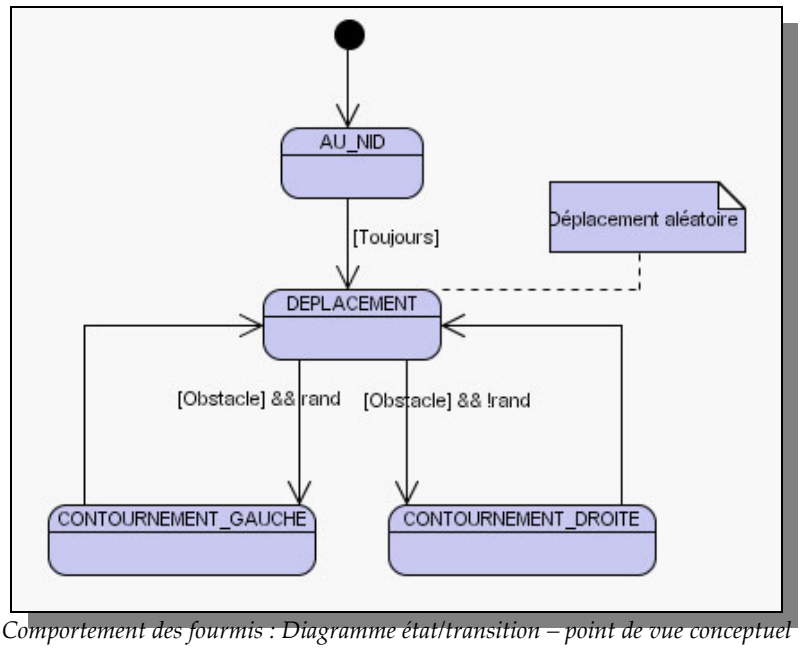
Le nouveau thread de Simulation appel à intervalle régulier la simulation de chacun de ses *Objet* par l'intermédiaire de leur méthode *run()*.

Au niveau de l'affichage 2D, on instancie dans *SimAntJFrame* un *SimAntJPanel* et lance son exécution dans un nouveau thread qui dessine chaque Objet à intervalle régulier par l'intermédiaire de leur méthode *draw()*.

Au niveau de l'affichage 3D, un nouveau thread est créé automatiquement par l'API GL4Java et la méthode *display()* de *SimAntGLAnimCanvas* est appelée à intervalle régulier pour rafraîchir l'affichage (chaque objet est récupérée par l'intermédiaire de *Simulation*).

1.4 Comportement des fourmis

Un mécanisme d'état avait été prévu pour assurer un comportement cohérent des fourmis au cours des itérations de simulation. Ces états étaient de deux types : l'état primaire et l'état secondaire. Les fourmis étaient crées autour de la fourmilière dans un état primaire initial *AU_NID* et un seul comportement avait été implémenté : le déplacement aléatoire de la fourmi sur le terrain. Pour cela, deux états secondaires intervenaient pour le contournement des obstacles.



Comportement des fourmis : Diagramme état/transition – point de vue conceptuel

1.5 Interactions en mode 3D

Le mode d'affichage 3D propose un nombre important d'interactions avec l'utilisateur ce qui nous donne l'occasion de présenter un diagramme de cas d'utilisation. Il est en effet possible de naviguer dans l'environnement au clavier et à la souris, d'obtenir un affichage fil de fer, d'activer ou de désactiver les textures et l'éclairage de la scène, de passer en mode vol, de réinitialiser la simulation et enfin de quitter l'application.

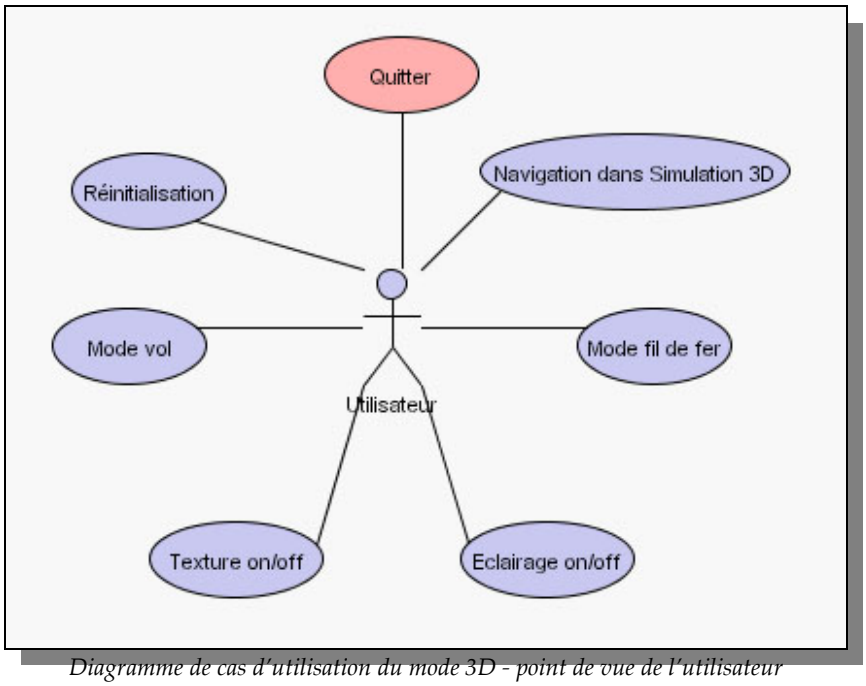


Diagramme de cas d'utilisation du mode 3D - point de vue de l'utilisateur

2. Améliorations

2.1 Description générale

Nous avons conçu trois types de fourmis, les ouvrières, les guerrières et les reines avec des comportements spécifiques. Nous avons également mis en place un système de phéromones permettant aux fourmis de laisser des traces de leur passage sur le terrain afin de guider les autres fourmis vers une cible déterminée.

Notre objectif principal était, comme nous l'avons indiqué dans l'introduction, de créer une application capable de gérer un nombre important de fourmis et nous avons donc conçu les structures de données adaptées permettant un fonctionnement optimal dans ces conditions.

Nous sommes également intervenus au niveau de l'affichage 3D en ajoutant un décor à la scène, en modifiant les textures, les couleurs des objets, les modèles et en en créant de nouveaux (Reine, Guerrière). Nous avons également géré l'affichage des phéromones et leur disparition progressive, la diminution de la taille de la nourriture et la représentation des cadavres de fourmis mortes au combat.

Nous avons également fait quelques modifications au niveau de diverses autres classes existantes qui posaient problème ou pouvaient être améliorés.

2.2 Architecture globale

Nous avons conservé l'architecture générale de l'application proposée en y ajoutant les classes nécessaires aux différentes améliorations et en modifiant les classes existantes affectées ou repensées.

Notre première modification a été l'abstraction de la localisation des objets en définissant la notion d'objets localisés. Pour cela nous avons créé une interface appelée *Localized* qui fixe les méthodes de récupération de coordonnées et de type pour les objets l'implémentant. Ceci permettra de pouvoir manipuler un grand nombre d'objets localisés de différentes natures dans des structures de données communes. Les classes *Objet*, *Pheromone* et *Vecteur* (Vecteur 3D qui permet de passer simplement des coordonnées aux méthodes) implémentent cette interface.

L'amélioration suivante a consisté à créer une classe de gestion optimalisée des objets présents sur le terrain. Cette classe appelée *TerrainManager* permet de stocker des objets localisés dans un monde discrétisé (quadrillage) autorisant l'accès rapide de chaque objet aux objets adjacents. *TerrainManager* fournis une méthode *radarScan* qui renvoi une liste de références sur les objets localisés détectés dans un périmètre, une direction et un angle donné à partir d'un point sur le terrain. Des instances de *TerrainManager* sont créées dans simulation pour chaque type d'objet indépendant (Phéromones, Obstacles et Fourmis) ce qui permet d'accélérer encore davantage les recherches en ne sélectionnant que la grille concernée.

Une fois ces outils définis, nous avons pu créer les trois types de fourmis intervenant dans la simulation (*FourmiOuvriere*, *FourmiGuerriere*, *FourmiReine*) sur la base d'une fourmi générique implémentée par la classe *ObjetFourmi* que nous avons adaptée. Les trois types de fourmis partagent en effet un certain nombre de comportements et de caractéristiques communes fournis par *ObjetFourmi* qu'elles spécialisent en fonction de leur rôle propre.

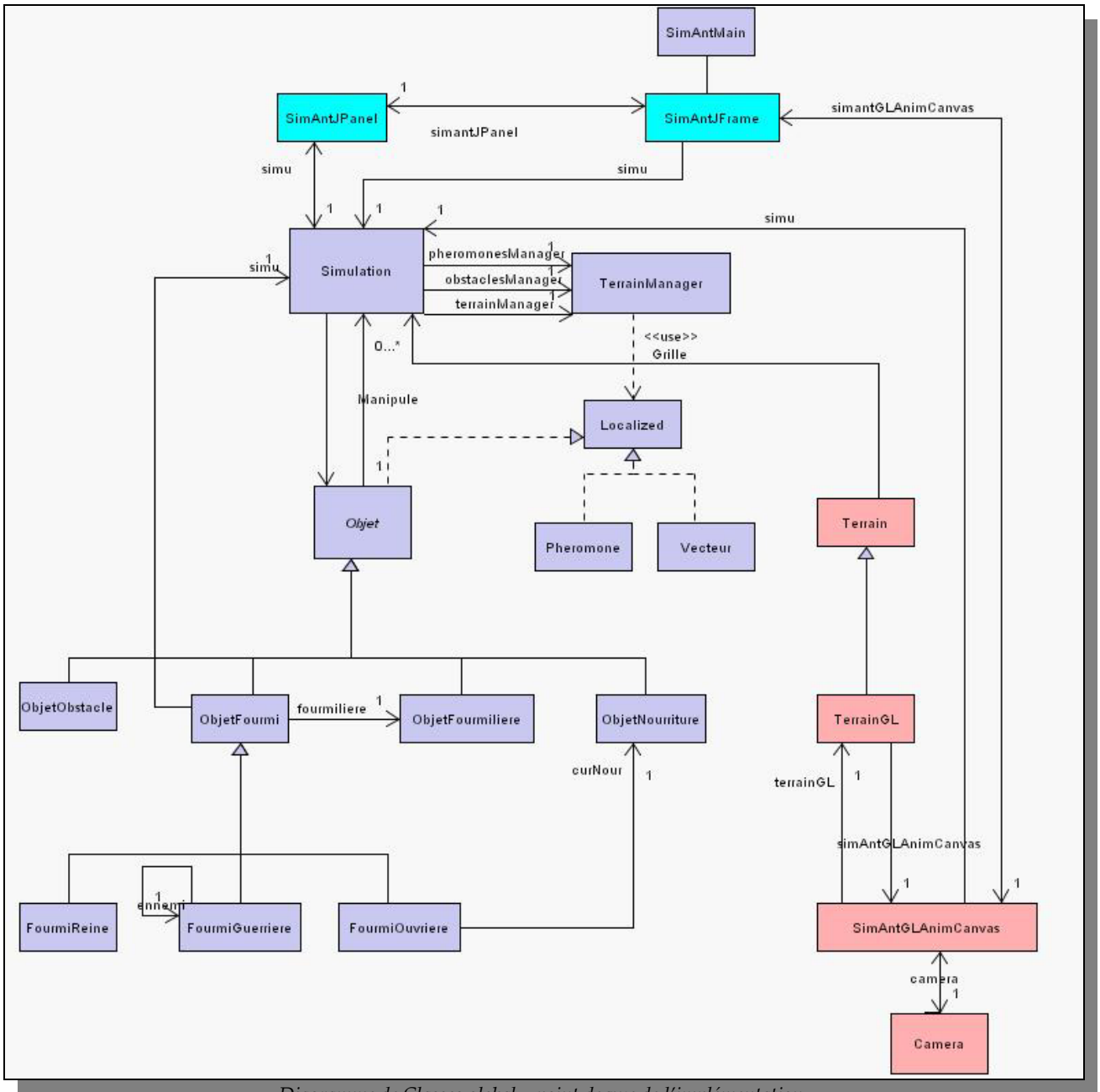
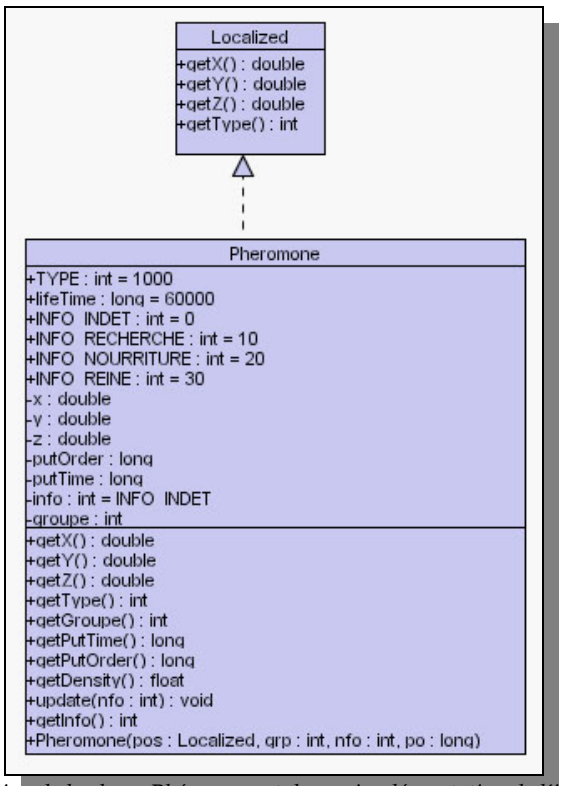


Diagramme de Classes global – point de vue de l'implémentation.
Les classes en rose sont spécifiques à l'affichage 3D et les bleus à l'affichage2D.

2.3 Les phéromones

Les fourmis que nous avons implémentées disposent de la capacité de déposer et de suivre des phéromones sur le terrain. Celles-ci permettent d'indiquer aux autres fourmis la direction de nourritures ou de fourmilières ennemis. Les phéromones sont des objets localisés et identifiés sur le terrain et implémentent de ce fait l'interface *Localized* qui fixe des méthodes de localisation et d'identification de type. Elles n'héritent pas d'objet car un grand nombre de ses caractéristiques ne lui sont pas utiles. Chaque phéromone appartient à un groupe qui correspond au groupe (fourmilière) de la fourmi qui l'a déposé. Ceci permet de réserver l'utilisation des phéromones seulement aux fourmis appartenant à la même fourmilière. Les phéromones disposent de trois caractéristiques principales : Une densité qui permet de les faire évaporer, une information qui permet de préciser ce qu'indique la phéromones, dans le cas présent il peut s'agir de nourriture ou d'un fourmilière ennemie et enfin d'un ordre de pose qui correspond en quelque sorte à la distance à l'objets vers lequel mènent les phéromones.

Au niveau de l'implémentation, nous avons fixé une durée de vie constante et la densité est calculée dynamiquement à partir de la date courante, la date de pose et la durée de vie, ceci permet de faire évaporer automatiquement les phéromones sans avoir à les mettre à jour constamment. Lorsqu'elles sont en fin de vie, la simulation se charge de les détruire pour réduire l'occupation mémoire. On instancie une *Phéromone* à partir d'une référence sur *Localized* indiquant sa position sur le terrain, d'un entier représentant le numéro du groupe auquel il appartient, d'une constante information ainsi que d'un long représentant la distance à la « cible ». On voit bien ici l'intérêt d'avoir spécifier une interface *Localized* implémentée par tous les objets localisés car cela permet ici de passer facilement les coordonnées de la fourmis dépositaire puisqu'il suffit dans la méthode de fourmis d'instancier une phéromone en passant *this* comme 1er paramètre.



Vue d'implémentation de la classe Phéromone et de son implémentation de l'interface Localized

2.4 Les fourmis

2.4.1 Comportements

Le principe global de notre simulation au niveau des fourmis est le suivant :

Les fourmis ouvrières naissent dans leurs fourmilières et partent à la recherche de nourriture en se déplaçant aléatoirement sur le terrain, si elles rencontrent un obstacle elles le contournent et dés qu'elles trouvent de la nourriture, elles la ramasse puis retournent à leur fourmilière pour la déposer en laissant une traînée de phéromone indiquant le présence de nourriture derrière elles. Si une fourmi ouvrière rencontre un tel type de phéromone alors qu'elle est en recherche de nourriture, elle se met à suivre sa trace jusqu'à atteindre la nourriture.

Au niveau de l'implémentation, cela se traduit par 6 états dont le fonctionnement est résumé dans le diagramme suivant :

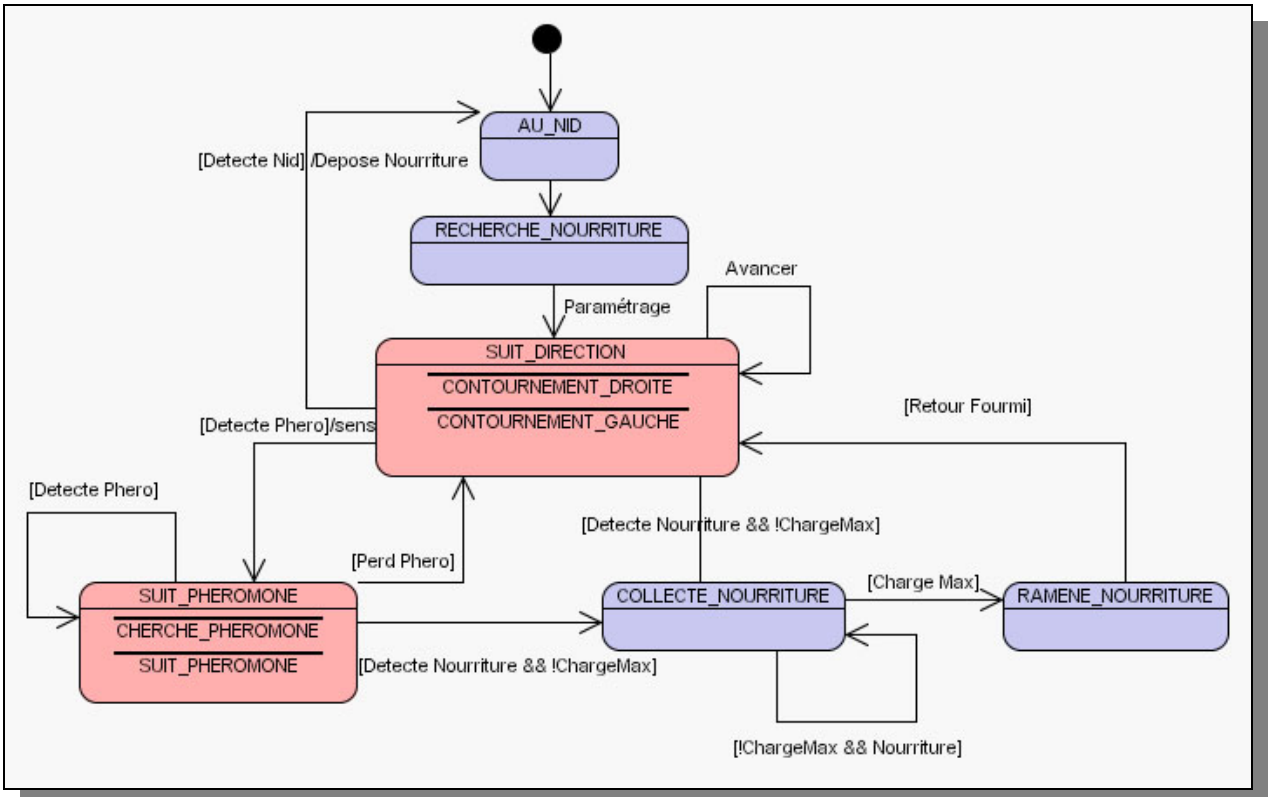


*Diagramme d'états-transitions du comportement de la **fourmi ouvrière** : Point de vue de l'implémentation*

Les fourmis guerrières ont un comportement assez semblable puisqu'elles naissent elles aussi dans leur fourmilière et partent aussitôt à la recherche d'une fourmilière ennemie en se déplaçant aléatoirement sur le terrain. Si elles rencontrent un obstacle, elles l'évitent et si elles rencontrent une fourmi guerrière appartenant à une autre fourmilière, elles l'attaquent (et une fourmi se faisant attaquer réplique) et combattent jusqu'à la mort de l'un des adversaires. Lorsqu'elles atteignent une fourmilière ennemie, elles rentrent à leur fourmilière en laissant des phéromones pour indiquer la cible à attaquer. Une fourmi guerrière qui détecte une trace de phéromone de met donc à la suivre jusqu'à atteindre la

fourmilière ennemie dans le but de tuer la reine présente (l'attaque de la reine n'a pour le moment pas été implémentée par manque de temps).

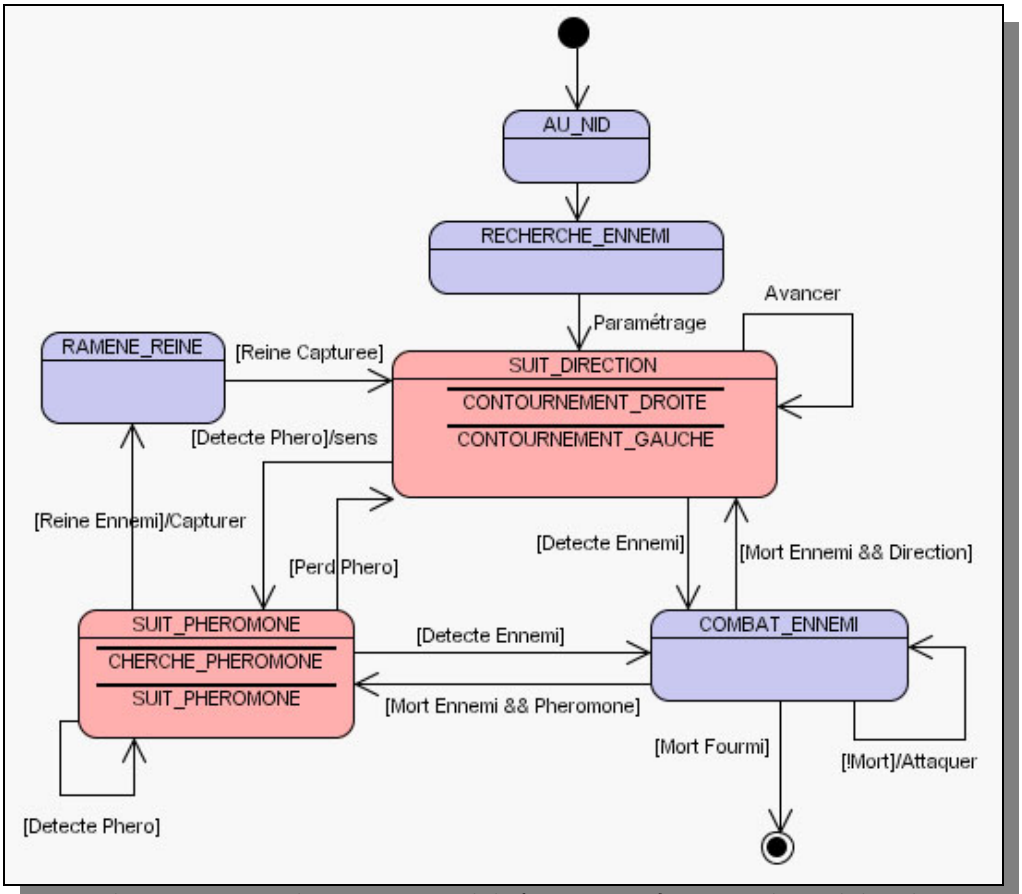


*Diagramme d'états-transitions du comportement de la **fourmi guerrière** : Point de vue de l'implémentation*

Les reines ont un comportement beaucoup plus basique et se contentent d'attendre autour de la fourmilière que le niveau d'énergie atteigne un certain seuil pour pondre de nouvelles fourmis :

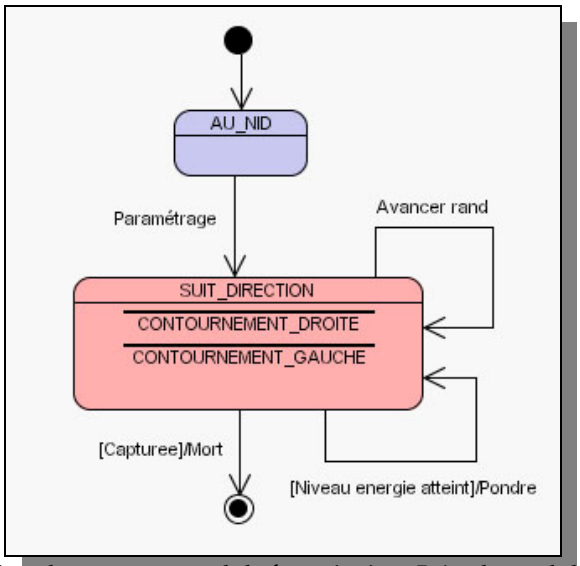


*Diagramme d'états-transitions du comportement de la **fourmi reine** : Point de vue de l'implémentation*

2.4.2 Abstraction des comportements et caractéristiques

Les trois types de fourmis décris précédemment partagent deux capacités principales : Suivre une direction (qui peut être aléatoire) et suivre des phéromones. Ces capacités correspondent aux comportements indiqués en rose dans les diagrammes précédents. Pendant le suivi d'une direction, des phéromones peuvent être déposés et pendant le suivi de phéromones, ces dernières peuvent êtres mises à jour au niveau de la densité et de l'information. Les trois types de fourmis mis en œuvre spécialisent donc un objet fourmi générique (*ObjetFourmi*) qui implémente ces comportements et les méthodes pour les paramétrer. La classe *ObjetFourmi* hérite elle-même de la classe Objet qui autorise l'affichage et permet les interactions dans la simulation.

Nous avons quelque peu modifié la classe Objet en y ajoutant un paramètre d'énergie qui permet une gestion unifiée des interactions entre objets de la simulation. Chaque objet interagit ainsi en terme de transfert d'énergie, une fourmi mangeant une pomme retire de l'énergie à la pomme et la stocke avant de la transférer à la fourmilière par exemple. Chaque objet possède donc une énergie de départ attribuée à la création et qui évolue tout au long de sa vie. Nous avons également supprimé les constantes représentants les différents types d'objets à l'intérieur de la classe *Objet* pour les placer dans les classes du type lui-même. Cette solution a été adoptée pour faciliter la compatibilité avec les parties d'origine du programme et permettre tout de même une évolutivité aisée mais le mieux aurait encore été d'utiliser simplement le paramètre statique *class* d'*Object* pour identifier encore plus simplement les types.

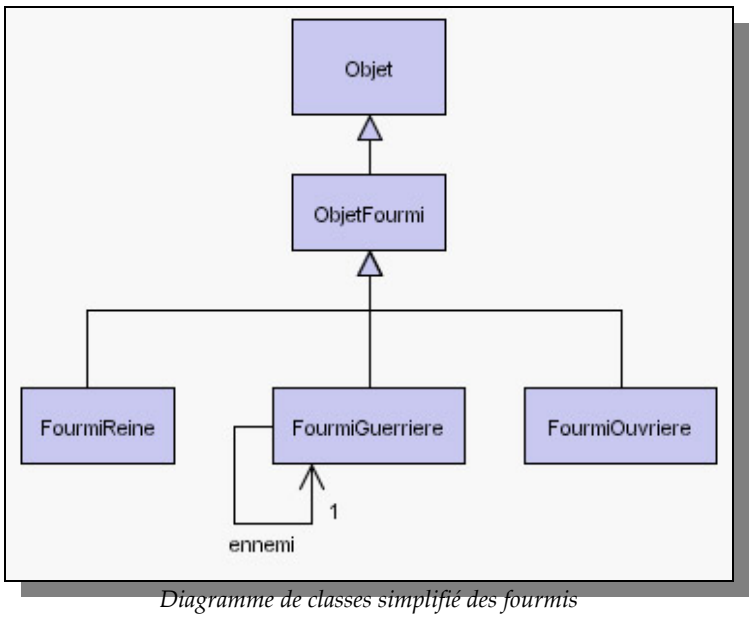


Diagramme de classes simplifié des fourmis

2.5 Optimisation de la localisation

A l'origine, tous les objets de la simulation étaient stockés dans une structure de donnée unique (vecteur) et lorsqu'une fourmi voulait connaître les éléments qui l'entourent sur le terrain (les obstacles par exemple), il fallait parcourir l'ensemble des objets pour réaliser un test de collision. Cette méthode n'est pas du tout efficace tout d'abord parce que seuls certains types d'objets intéressent les fourmis et en plus parce que la plus part des objets testés sont beaucoup trop éloignés de la fourmis pour être pris en compte et sont donc testés pour rien. De plus, la technique de collision mise en place ne suffisait plus pour notre simulation car les fourmis en plus de savoir si elles se trouvent sur un obstacle doivent pouvoir détecter des éléments à distance pour s'orienter comme c'est le cas pour suivre les phéromones ou attaquer un ennemi.

Pour permettre cela, nous avons conçu une classe nommée *TerrainManager* qui fournit des méthodes d'ajout et de détection optimisée d'éléments sur le terrain. Cette classe discrétise en quelque sorte le terrain en stockant les éléments dans une grille 2D qui permet un regroupement géographique des objets et donc une détection très rapide des éléments situés autour d'une position donnée. Cette classe n'est pas manipulée directement par les agents (fourmis) mais par Simulation qui en possède différentes instances pour chaque type d'objets ce qui permet une recherche encore plus efficace d'un type d'objet donné. L'objet Simulation fournît donc une méthode de recherche prenant en paramètre le type d'objet recherché et relaye la requête au *TerrainManager* concerné. Nous avons unifié toutes les recherches sur le terrain, que ce soit les tests de collision ou les détections à distance, pour leur faire profiter de cette classe.

Au niveau de l'implémentation, la classe TerrainManager manipule uniquement des *Localized* ce qui permet d'y stocker n'importe quel type d'objet localisé. La grille est représentée sous forme d'un tableau 2D de *Vector* ce qui autorise donc l'accès direct à une case par offset dont il reste a parcourir le vecteur pour en connaître les occupants. La classe fournit une méthode radarScan qui prend en paramètre une position de scan, une direction (angle en radian, en général l'orientation de la fourmi), un rayon de recherche, un angle de recherche qui permet de limiter le champs de vision et pour finir un type d'objet à recherché pour le cas ou l'on stocke plusieurs types différents dans le même *TerrainManager*. Pour la mise à jour de la grille lors du déplacement d'éléments, une méthode *update* qui re-parcourt l'ensemble des objets ajoutés et les places dans les cases appropriées est fournie. L'ajout de nouveaux occupants se fait par l'intermédiaire de la méthode *add* et l'instanciation d'un *TerrainManager* se fait à partir de la taille du terrain et de la résolution de la grille souhaitée. Des méthodes de calcul rapide de la case correspondante à une coordonnée réelle ont également été implémentées ainsi qu'une méthode pour remplir la grille directement à partir d'un vecteur de *Localized*.

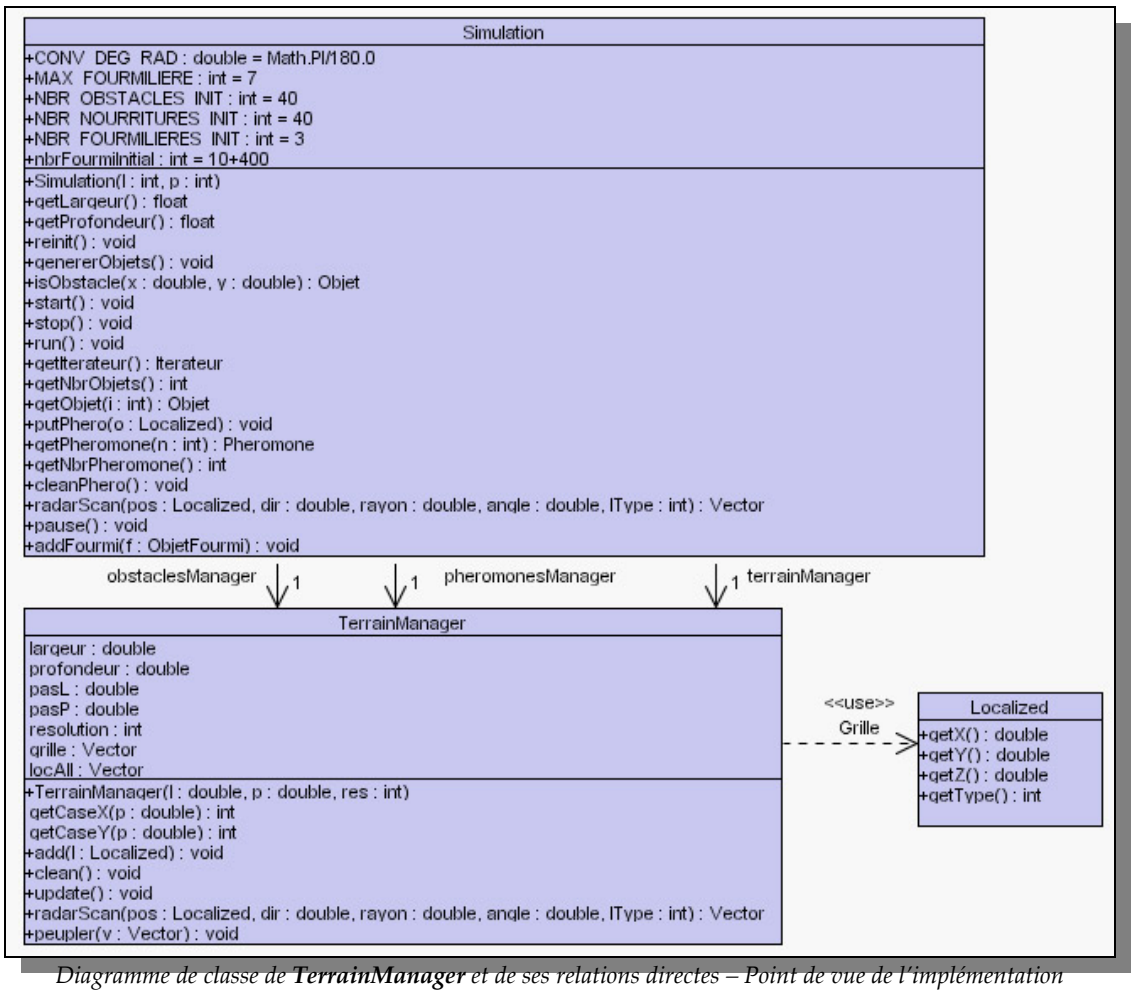
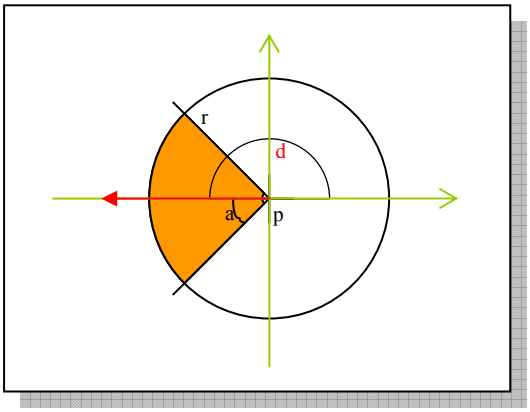


*Diagramme de classe de **TerrainManager** et de ses relations directes – Point de vue de l'implémentation*

Un grand nombre d'opérations de Simulation passent donc maintenant par *TerrainManager* comme la détection d'obstacles (*isObstacle*), La pose de phéromone (*putPhero*) ou le scan radar (*radarScan*).

La méthode de recherche radar est largement paramétrable pour pouvoir simuler de façon réaliste la vision d'un individu et également assurer sa généricité. Voici un schéma de principe du fonctionnement du radar :



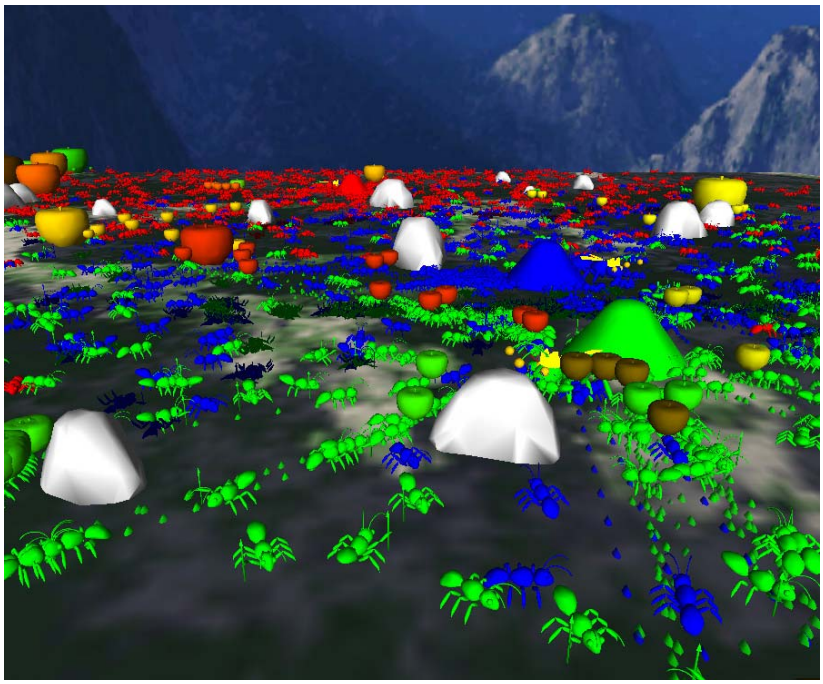
*Cercle de détection du radar avec **p** la position, **r** le rayon, **d** la direction et **a** l'angle de scan.*

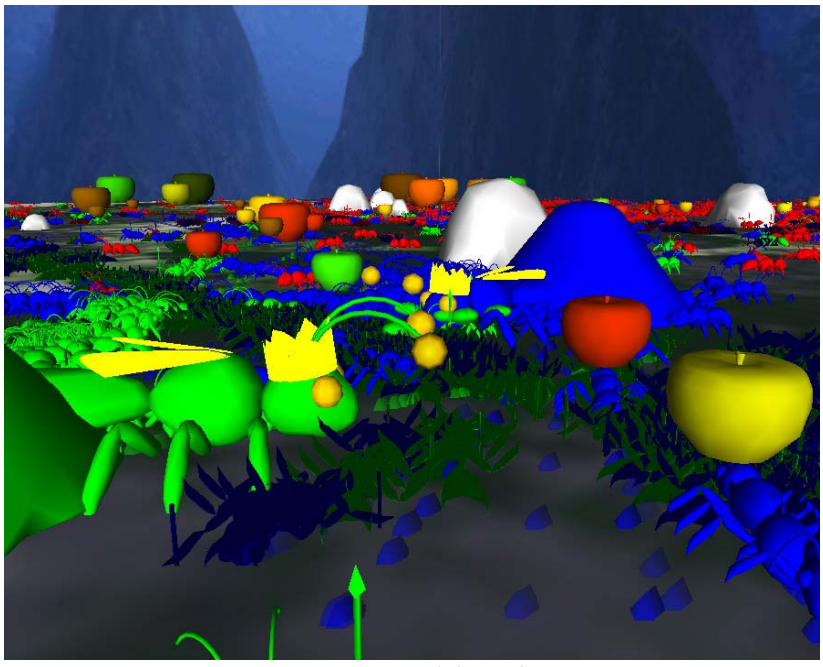
Voici l'implémentation Java de la méthode de recherche d'éléments de *TerrainManager* :

```
/**!BC Scan (2D) à la recherche d'un objet localisé.
 *
 * @param pos Position de l'objet scannant
 * @param dir Direction du scan
 * @param rayon Rayon de scan
 * @param angle Angle de scan
 * @param lType Type d'objet a rechercher, 0 tout
 * @return Objets trouvés
 */
public Vector radarScan(Localized pos, double dir, double rayon, double angle, int lType){
	//Vecteur résultat
	Vector tmpVect=new Vector();
	Localized tmpLoc;

	//Ajustement ds angles
	while(dir>2*Math.PI)
		dir-=2*Math.PI;
	while(dir<0)
		dir+=2*Math.PI;

	//Determinaison des cases concernées
	int cX=getCaseX(pos.getX());
	int cY=getCaseY(pos.getY());

	//Reference temporaire vers une case
	Vector listeLocalized;

	//Parcour de la case et de ses 8 cases adjacentes
	for(int gX=cX-1; gX<=cX+1; gX++)
	for(int gY=cY-1; gY<=cY+1; gY++)
	//Cas des bordures
	if(gX>=0 && gX<resolution && gY>=0 && gY<resolution){
		listeLocalized=grille[gX][gY];
		//Parcours des éléments d'une case
		for(int i=0; i<listeLocalized.size(); i++){
			tmpLoc=(Localized)listeLocalized.get(i);
			//Test du type
			if(lType == 0 || lType == tmpLoc.getType() ){
				double dist;

				// /!\ Arg Trigo ;-)
				dist= Math.sqrt( Math.pow(pos.getX() - tmpLoc.getX(), 2)+
						Math.pow(pos.getY() - tmpLoc.getY(), 2) );

				//Test de distance
				if(dist < rayon){
					double angleObj; //Angle radar de l'objet trouvé
					//Position dans le cercle de scan
					double px=tmpLoc.getX() - pos.getX();
					double py=tmpLoc.getY() - pos.getY();
					//Angle dans le cercle de scan
					angleObj=Math.acos(px/rayon);
					if(py<0)
						angleObj=2*Math.PI - angleObj;

					//Calcul de l'angle entre la direction et l'objet détecté
					if( Math.abs(angleObj-dir) < angle ){
						tmpVect.add(tmpLoc);
					}
				}
			}
		}

	}

	return tmpVect;
}
```

2.6 L'affichage 3D

Nous sommes intervenu à plusieurs niveaux sur l'affichage 3D de la simulation. Nous avons tout d'abord créé des modèles spécifiques à chaque type de fourmis et les avons inclus dans le rendu. Nous avons fait en sorte que les fourmis portent une pomme lorsqu'elles ramènent de la nourriture au nid et que cette dernière soit de la même couleur que celle du terrain ou elles ont pris l'énergie. Nous avons également géré l'affichage des phéromones sous forme de sphères et la représentation de leur évaporation avec l'ajout de *blending* pour les rendre translucides en fonction de leur densité. Nous avons modifié la texture du terrain et ajouté une *sky box* autour de la scène pour simuler un environnement naturel.





Deux images issues de la simulation

Conclusion

Ce projet nous a permis tout d'abord de nous plonger dans un code Java existant pour en faire l'analyse ce qui fut assez formateur et nous a permis de nous rendre compte de l'importance de concevoir une architecture extensible et un code clair et commenté. Cette étape c'est en effet révélée assez rapide à achever principalement grâce à la présence de commentaires clairs et l'utilisation de logiciels de modélisation UML capables de « reverse engineering » sur du code source Java.

La partie d'analyse et de conception nous a permis de nous familiariser avec le formalisme UML ainsi qu'avec deux logiciels de modélisations. Nous avons en effet commencé l'analyse avec *Rational Rose* mais nous avons rapidement été confronté à un grand nombre de lacunes ergonomiques et déficiences fonctionnelles que présente ce logiciel et avons ensuite adopté *Visual Paradigm*, un logiciel de modélisation distribué gratuitement en version *Community*, beaucoup plus fonctionnel.

L'étape de conception nous a poussé à réfléchir à une architecture efficace et extensible qui permette ensuite l'évolutivité et la maintenance de l'application. Nous nous sommes efforcé à fournir un code commenté (Java Doc) comme c'était le cas pour le programme d'origine.

Ce type d'application a l'avantage de pouvoir être étendue et améliorée quasiment à l'infini et nous regrettons de ne pas avoir eu d'avantage de temps pour réaliser d'autres modifications aux quelles nous avions pensé. Par exemple au niveau graphique, il pourrait être intéressant d'utiliser le nouveau système de localisation pour optimiser l'affichage en ne rendant que les éléments visibles (*Frustum Culling*) et non plus tous les objets. Une autre modification aurait également été souhaitable pour faciliter l'évolutivité du programme au niveau de l'identification des objets. L'utilisation de constantes numériques n'est pas une solution idéale et il serait souhaitable d'utiliser directement le mécanisme d'identification des classes proposées par java. Il y aurait également des choses à développer au niveau du paramétrage de la simulation en temps réel. On pourrait penser par exemple à une fenêtre de configuration permettant avec des curseurs d'influer sur un ensemble de paramètres (vitesse des fourmis, durée de vie, force …).

Une fois l'architecture globale de l'application fixée et les outils développés, les modifications sur la simulation deviennent beaucoup plus aisée et avec un peu d'imagination tout devient possible.