

**DANVIN Yohan  
FARGE Julien  
GOUBE Florent  
SEMET Gaëtan**

**Année 2002-2003**

**INSTITUT SUPÉRIEUR D'INFORMATIQUE  
DE MODÉLISATION ET DE LEURS APPLICATIONS**

**Simulation d'une fourmilière**

## **Concept du projet :**

Le projet Fourmizz consiste à l'élaboration d'une simulation d'une colonie de fourmis.

La simulation s'effectue à l'extérieur de la fourmilière sur la recherche de la nourriture et son rapatriement ainsi que sur la défense du lieu de résidence. L'affichage est fait en trois dimensions sur une carte générée aléatoirement, le déplacement est alors représenté pour chaque fourmi.

## **I) La recherche sur la fourmi :**

Afin de réaliser cette simulation la première action qu'il fallut entreprendre fut une recherche sur l'entomologie afin de connaître la réalité de la fourmi : ses capacités de déplacement, sa capacité à trouver de la nourriture, ses moyens de défenses, etc. Ces résultats sont fondamentaux dans la compréhension du projet Fourmizz.

### **1) La hiérarchie d'une colonie de fourmis**

La composition d'une colonie est la suivante :

- Une reine : qui pond et gère la vie de la colonie.
- Les soldats : qui surveillent et gardes la colonie d'attaque extérieur
- Les éclaireuses : qui recherchent la nourriture
- Les récolteuses : qui rapatrient la nourriture à la fourmilières.
- Les nourrices : qui s'occupent des œufs, des nymphes et des larves
- Les males : les reproducteurs de la colonie qui s'en iront lors de la prochaine phase de reproduction.
- Les femelles : les futurs reines des futurs colonies
- Les œufs : le premier stade de l'évolution de la fourmi, ils sont pondus par la reine
- Les larves : deuxième stade de l'évolution de la fourmi.
- Les nymphes : troisième et dernier stade avant d'atteindre le stade de fourmi.

### **2) Le fonctionnement de la recherche de la nourriture**

La recherche de la nourriture se fait par les éclaireuses qui partent à la recherche de divins festins pour toute la colonie que les récolteuses vont se faire le plaisir de rapatrier.

#### **a) Mécanisme de recherche de la nourriture**

La recherche de nouvelles sources de nourriture est assurée par les fourmis éclaireuses qui créés un repère de phéromone pour ses semblables lorsque une ressource de nourriture est découverte.

Leur moyen d'indication est la phéromone de nourriture qui indique le chemin à suivre pour les récolteuses.

Le fonctionnement en est le suivant : les éclaireuses partent en quête de nourriture pour la colonie, en partant elle laisse un sillon, qu'elles suivront lors du retour. Elles recherchent la nourriture quasi aléatoirement en effet elles avancent généralement

rectilignement se fiant à leur odorat et leur vision extrêmement faible. Une fois la nourriture trouvée, elles retournent à la fourmilière en déposant des phéromones indiquant aux récolteuses le chemin à suivre.

### **b) Mécanisme de retour vers la fourmilière**

Les fourmis récolteuses sont du point de vue de leur taille plus petite que les éclaireuses. C'est d'ailleurs pour nous la plus petite fourmi.

Leur fonction est de récupérer toute la nourriture d'une ressource et de la rapporter à la fourmilière tout en optimisant le chemin. Cette optimisation est inconsciente pour la fourmi mais bien réelle.

Leur moyen pour optimiser le retour est aussi la phéromone de nourriture en effet ces fourmis déposent de la phéromone de nourriture lorsque elle rentre avec de la nourriture.

Leur fonctionnement est le suivant : une fois que les récolteuses ont suivi la direction indiquée par l'éclaireuse, et récupéré la nourriture, elles suivent les phéromones dans le sens du retour tout en lissant le parcours pour finir par suivre une direction rectiligne. Elles optimisent d'elles-mêmes le retour et l'allé.

Une fois la nourriture épuisée les fourmis lors d'un retour sans nourriture et indiquent à leur semblable la fin de la ressource. Ensuite les fourmis retourneront à la fourmilière dans l'attente d'une nouvelle ressource.

### **3) Défense de la fourmilière :**

Les fourmis soldats sont les plus grosses fourmis que l'on peut trouver au sein de la colonie, elles ont une morphologie plus massive et sont plus résistante.

Leur fonction est la protection de la colonie.

Leur moyen de défense se résume à la capacité à projeter de l'acide sur l'ennemi, et la phéromone de danger cette dernière est utilisée par toutes les fourmis. En effet lorsque une fourmi d'une colonie rencontre un danger potentiel, elle dépose de la phéromone de danger. Cette phéromone a pour conséquences de repousser les fourmis de la colonie de sorte que ces dernières évitent le danger, et attire les soldats de sorte que ces dernières protègent la colonie.

Leur fonctionnement est le suivant : les soldats recherchent des ennemis dans leur alentours si aucun ennemi ou phéromones de danger se présentent alors les soldats suivent le groupe de fourmis le plus important par le nombre. Si aucun de ses cas ne se présentent elles patrouilleront autour de la fourmilière.

## **II) Le choix du langage**

Certaines contraintes dans l'intelligence artificielle et la création de l'univers 3D ont pour conséquences de faire un choix de langage de programmation. Ces contraintes sont les suivantes :

La modélisation de la vie artificielle se prête bien à la programmation agent donc il fallait un langage qui nous faciliterait ce type de programmation. Puisque la programmation agent, permet de simuler une interaction entre des entités autonomes il nous fallait un langage objet.

De plus la modélisation de l'univers graphique requerrant une certaine rapidité de calcul, le JAVA a été exclu.

Enfin, il nous fallait un langage facile à prendre en main. Donc le choix s'est porté sur le C++, puisque ce langage possède une syntaxe similaire au C que nous avons appréhendé au cours de l'année. De plus, l'utilisation de bibliothèques standards et portables permet d'avoir une application qui peut se compiler aussi bien sous Windows que sous Linux (voir même MacOS).

### **III) Vie artificielle de la fourmi**

Cette partie a été traitée par Yohan Danvin et Julien Farge.

Le concept est le suivant pour les fourmis : nous voulions modéliser les fourmis du monde réel, mais ceci est impossible car cela pose plusieurs contraintes dont nous ne vîmes pas la possibilité d'adaptation dans notre programme.

Ces contraintes sont les suivantes :

- le monde réel c'est-à-dire la planète est infini par rapport à la représentation que s'en fait l'humain. Faire un univers aussi vaste dans notre programme est impossible car stocker une aussi grande quantité de données est impossible.
- Nous ne savons pas réellement comme les fourmis réelles réagissent avec leur environnement, donc il nous a fallu limiter le comportement de nos fourmis aux observations scientifiques qui ont été faites.
- Pour nous les fourmis suivent des plots (token), qui ont été posés à l'aller. Ils représentent le sillon utilisé dans la réalité par les fourmis. Nous avons fait l'hypothèse que les fourmis sont capables d'évaluer la direction approximative de leur nid. Dans notre cas, leur sens de l'orientation est parfait.

De nombreux autres problèmes se sont posés et ont été traités lors de l'élaboration de ce projet.

#### **1) La hiérarchie des classes**

Cf. graphe de hiérarchie des classes en Annexe

Cette hiérarchie est adaptée à notre projet, cette hiérarchie a été mûrement réfléchie par Yohan Danvin.

**Remarque :** ultérieurement on aurait pu créer les classes : œuf, larve, pupe, nymphe directement dérivée de aiAnt c'est pour cela que nous avons implémenté cette classe sans utilité significative à première vue.

#### **2) Le monde**

Le monde est constitué d'un tableau de tableau de secteurs.

La taille du monde est une puissance de 2 nécessairement. Cette taille est fournie par le menu qui permet à l'utilisateur de la choisir.

Le nombre de secteurs est déduit de cette taille et de la taille d'un secteur.

(Remarque : les secteurs sont carrés.)

### **3) Redondance de code**

Exemple : les coordonnées d'un objet et le monde auquel il appartient déterminent le secteur du monde auquel il appartient. Cependant, la classe « objet » (aiObject) possède un champ sector\_. Cette redondance ajoute 4 octets dans chaque objet mais elle améliore sensiblement les performances.

D'une manière générale toute redondance dans le code n'a pas d'impact significatif sur l'occupation de la mémoire, mais est très bénéfique au niveau des performances.

### **4) Le fonctionnement des agents**

Les agents, caractérisants notre simulation, sont :

Les récolteuses : « harvester »

Les éclaireuses : « scout »

Les soldats : « soldier »

Les araignées : « spider »

Les récolteuses et les éclaireuses sont des dérivées de aiMinor donc leurs fonctions communes se situeront dans cette classe.

Les récolteuses ou « harvester » sont les fourmis qui vont rechercher de la nourriture. Leur fonction pour nous se limite à suivre la phéromone « nourriture trouvée » et à rapporter la nourriture à la fourmilière. Elles sont capables de déposer des phéromones de danger de nourriture et des plots de chemin.

Les fonctions qu'elles ont besoin pour réaliser ces actions :

- scanner les environs pour voir ce qu'il y a autour que toutes les fourmis (aiAnt) possèdent, et qui permet de voir les objets qui sont autour ;
- déposer des phéromones de nourriture (commune à l'éclaireuse), et de danger (commune au soldat) ;
- déposer des plots (token) (commune à toutes les fourmis) ;
- trouver la plus grande concentration de phéromones dans les alentours quelque soit la phéromone (commune à toute les fourmis);
- trouver la plus grande concentration de plots (token) (commune de fourmis mineur (aiMinor)) ;
- se déplacer aléatoirement que toutes les fourmis (aiAnt) possèdent;
- porter des objets ;
- reconnaître ennemi ou ami(commun à toutes les fourmis).

Les éclaireuses ou « scout » sont les fourmis qui vont découvrir les sources de nourriture. Leur fonction pour nous se limite à rechercher la nourriture et à rentrer à la fourmilière tout en déposant des phéromones pour que les récolteuses retrouvent cette ressource. Elles sont capables de déposer des phéromones de danger de nourriture et des plots de chemin.

Les soldats ou « soldier » sont les fourmis qui vont protéger la colonie et la fourmilière. Leur fonction pour nous se limite à rechercher le danger et à escorter les convois de récolteuses. Elles sont capables de déposer des phéromones de danger et des plots de chemin.

Les fonctions qu'elles ont besoin pour réaliser ces actions en plus des actions communes sont:

- rechercher un ennemi ou des phéromones de danger ;
- combattre l'ennemi ;
- trouver la plus grande tendance de phéromone de danger.

L'araignée ou « spider » est l'ennemi le plus redoutable de nos fourmis car c'est le seul mais quel ennemi. Ces fonctions sont les suivantes avancées aléatoirement et rechercher des potentiels proies : des fourmis. Pour cela elle utilise les fonctions suivantes :

- avancer aléatoirement ;
- rechercher des ennemis ;
- scanner les environs ;
- combattre l'ennemi.

## **IV) Le graphisme**

Cette partie a été réalisée par Florent Goube et Gaëtan Semet.

### **1) La génération de la carte**

La conception de la carte en 3D est faite aléatoirement. Ainsi à chaque lancement les bosses sont différentes. Donc nous n'avons jamais à l'écran la même carte, ce qui a pour intérêt de voir évoluer les fourmis différemment à chaque utilisation et de voir que les fourmis comme dans la réalité ne font pas attention aux obstacles. En effet les fourmis ne contournent pas les bosses.

#### **A) Le principe de création :**

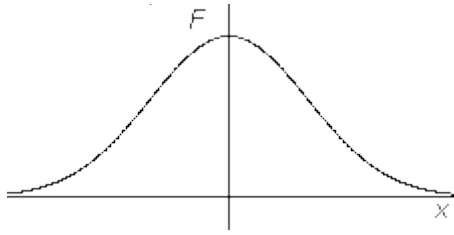
On utilise la génération de la carte par loi normale :

##### **a) Loi normale**

La formule de la loi normale est :

$$F(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

La répartition est la suivante :



La formule que nous utilisons est la suivante :

$$H=r*\exp(-(i^2 + j^2)/r^2)$$

H : hauteur du point

r : rayon du cercle

i et j : coordonnées du point

### **b) Déroulement de la création**

On crée au début une surface plane que l'on va modifier en ajoutant des bosses aléatoirement de telle sorte que le terrain sera unique à chaque utilisation. La méthode de création des bosses est la suivante. On choisit tout d'abord un point au hasard par un random. Ensuite de ce point on dessine un cercle de rayon calculé aléatoirement. Ensuite de chaque point de ce cercle on prend le rayon au quel on applique la loi normale citée si dessus. C'est-à-dire que l'on applique la loi normale sur chaque point du rayon ainsi on calcul la hauteur de chaque point de la bosse. Le rendu est donc pour un point du cercle une forme de vague, qui est semblable à la répartition de la loi normale, et pour le cercle on obtient ainsi une bosse de circonférence ronde. Ensuite on applique ce même procédé sur plusieurs autres points de la carte.

Rendu :



## **B) L'algorithme de principe de création de la carte est :**

Création de la surface plane

Tant que l'on n'est pas arrivé aux nombres de bosses souhaités faire

    Choix d'un point au hasard

    Choix du rayon du cercle au hasard

Tant que tous les points du cercle ne sont pas traités faire

        Pour chaque point du rayon faire

            application de la loi normale autour des coordonnées

        Fait

    Fait

Fait

## **2) Objets et animations**

### **A) C3DObject**

Le but de ce module était d'avoir une gestion des objets et des animations le plus indépendant possible de l'intelligence. Et tout et pour tout, l'intelligence n'a à connaître que le nom des objets, et le nom des animation à envoyer. Si j'ai choisit ça, c'est pour pouvoir facilement changer d'objet graphique sans avoir à toucher un morceau de code. Et aussi pour ne pas avoir à alourdir le programme avec une gestion précise de chaque objet. Par exemple il aurait fallu créer une classe Patte, une classe tête,... Mais si l'on veut pouvoir voir 2000 fournis en même temps de manière fluide, cela nous aurait alourdi le programme inutilement.

#### **Principe de fonctionnement**

Chaque objet que l'on voit à l'écran (fourmis, cailloux, herbe, animaux,...) sauf le terrain est un objet C3DObjElement. Ou pour être plus précis, c'est une arborescence de C3DObjElement. L'organisation en arbre est très pratique pour les mouvements (animations), cela permet d'avoir un squelette par exemple. J'appellerais dans la suite « Objet » l'arborescence et « Élément » un noeud particulier de l'arbre.

Un Élément contient:

- ‡ L'objet graphique : il s'agit d'un objet 3D Studio Max chargé avec une librairie externe. Il s'agit d'un élément qui ne changera pas de forme tout au long du fonctionnement du programme. Par exemple, un bras ou une jambe, une tête, ou un thorax.
- ‡ Une liste de toutes les animations possibles pour cet objet. Par exemple, tourner la tête, marcher, s'arrêter,... Chaque position (x, y, z, anglx, angley, anglez) est RELATIVE par rapport à la position du noeud parent, sauf le root qui doit être (en principe) centré en (0,0,0,0,0,0).
- ‡ Une liste de tous les fils (noeuds) de cet objet. Par exemple, le thorax aura comme fils les pattes, la tête et l'abdomen.
- ‡ Divers éléments, comme par exemple le trigger associé (décrit plus tard), une bounding box pour la gestion éventuelle des collisions.
- ‡ Chaque Élément est identifié par un identificateur unique à l'intérieur de chaque Objet (pas d'unicité des identificateurs des Eléments à l'intérieur du programme).

Remarque importante: quelques soit l'objet, il est nécessaire que la racine de l'arbre ne contienne pas d'objet 3ds ET que sa bounding box englobe TOUT l'objet.

## **B) Fonctionnement des animations**

A chaque instant, le moteur calcul la position où doit se trouver chaque élément. J'entends par « position » l'ensemble des coordonnées spatiales, x, y et z ET aussi les angles de rotations autour des 3 axes x, y et z. L'objet qui englobe ceci s'appelle CPosition.

Il y a en gros 2 philosophies pour les moteurs d'animation: soit l'Objet contient TOUTES les positions inscrites en lui, donc le moteur n'a qu'à, pour une frame donnée, qu'a aller « prendre » la bonne position. C'est très rapide, mais il faut un éditeur puissant, et les mouvements ne seront pas très réalistes. Exemple d'un tel moteur : Quake 2.

Une autre méthode, et c'est celle que j'ai utilisée, est de n'enregistrer que les positions clés (KeyFrames), et les positions intermédiaires seront CALCULEES pendant l'exécution. Un tel moteur est utilisé dans Half Life ou WarCraft III, par exemple. Il existe là aussi plusieurs méthodes d'interpolation de position, j'ai utilisé une simple interpolation linéaire. On aurait pu aussi développer une interpolation quadratique, mais cela rajoute beaucoup de calcul.

De plus, avec ce fonctionnement, on peut accélérer et ralentir les animations très facilement en changeant le nombre de frame à interpoler entre 2 KeyFrames.

Remarque: il existe aussi des techniques d'animation s'appuyant sur un squelette qui permet de bien interpoler les objets eux-même, pour éviter par exemple, qu'un avant-bras ne rentre dans le bras au cour d'un mouvement.

### **Note au sujet des objets graphiques:**

Chaque Objet (fourmi, pierre,...) dispose de sa structure C3DObjElement. Cela fait de la redondance de donnée (principalement au niveau des animations, qui sont stockées plusieurs fois), mais c'est nécessaire car il a été prévu que l'intelligence puisse spécifier une position précise pour une KeyFrame particulière, dans une version future du moteur. Par exemple l'intelligence peut spécifier l'angle de rotation de la tête, donc l'ia disposera de l'adresse de la position maximale de la tête qu'elle pourra modifier.

Cela dis, il n'y a pas de redondance de donnée au niveau de l'objet 3ds, qui est stocké une fois pour toute dans le programme (dans la liste globale t3DObjects g\_objects\_list ) car à aucun moment il n'est prévu de modifier l'objet. Il en est de même pour les textures des objets qui sont stocké et chargés dans opengl une seule fois grâce à la liste t3DTextures g\_textures\_list.

## **C) Gestion des animations**

Le fonctionnement des animations est assez complexe. Le but est de ne pas avoir d'accoup, c'est-à-dire de disgrâce au niveau des changements d'animations inopportuns. Par exemple pour passer de l'animation MARCHE à ARRET, il faut effectuer une étape de transition (arrêt des pattes). Il faut d'abord terminer l'animation MARCHE, passer à l'étape de transition, et ensuite passer sur l'animation ARRET (par exemple, la fourmi peut tourner la tête pendant qu'elle est arrêter pour se demander où elle est).

Tout cela est géré automatiquement par le moteur, en utilisant le mécanisme que j'ai appelé NextAnimation.

L'idée est d'avoir, pour chaque animation, une animation suivante (NextAnimation) par défaut sur laquelle cette animation ira automatiquement quand l'animation sera terminée. Une animation peut ne pas avoir d'animation suivante, dans ce cas, le moteur fera boucler cette

animation. Par exemple, l'animation MARCHE ne doit pas avoir de NextAnimation; ainsi, la fourmi avancera tout le temps, jusqu'à ce qu'on lui dise d'arrêter. Quand on demandera de passer à l'animation ARRET, le moteur attendra de terminer l'animation pour passer sur cette animation de transition.

Typiquement, les animations d'état (MARCHE, ARRET, TETETOURNEE, ...) n'auront pas de NextAnimation, et les animations de transitions auront une NextAnimation sur une animation d'état.

Lorsque l'intelligence doit changer d'animation (passer de l'animation SRC vers l'animation DST), il y a deux choses à faire:

† premièrement, dérouter toutes les animations SRC (qui en toute logique doivent être des animations d'état, c'est-à-dire bouclante) vers des animations de TRANSITION qui correspond au passage de SRC à DST. En clair, cette animation de transition doit avoir DST comme NextAnimation.

† Remplacer tous les NextAnimation qui pointait vers SRC en les faisant pointer vers DST.

Par exemple l'animation TURNER\_TETE\_GAUCHE a pour NextAnimation ARRET. Lorsque la fourmi est à l'arrêt et qu'elle veut tourner la tête à gauche, l'animation ARRET est dérouterée (après qu'elle ai fini sa boucle) vers l'animation TURNER\_TETE\_GAUCHE. Le moteur effectue cette animation. Lorsqu'elle a fini, le moteur se rend compte qu'elle à ARRET comme NextAnimation. Elle branche dessus.

Maintenant, on veut que la fourmi avance. Comme l'animation « tourner la tête à gauche » ne changera raisonnablement pas si elle est à l'arrêt ou en mouvement, il n'est pas utile d'écrire 2 animations `TURNER_TETE_GAUCHE_A_ARRET` et `TURNER_TETE_GAUCHE_EN_MARCHANT`. Donc lors du changement à l'état MARCHER, le moteur remplace la NextAnimation de `TURNER_TETE_GAUCHE` de ARRET à MARCHE.

**Remarque:** il était prévu de faire un système de sélection des animations, basé sur un automate fini, dont toutes les informations seraient stockées dans un autre fichier (ou à l'intérieur même du fichier que celui qui définit des animations). Cet automate devait gérer tout seul que faire lorsqu'on veut changer d'état.

Ceci aurait constitué une couche d'abstraction des animations pour l'intelligence. De cette manière, l'intelligence n'aurait à aucun moment à savoir comment fonctionne la partie animation.

Mais par manque de temps, les changements d'états sont codés en dur, dans le programme.

## **D) Trigger:**

Pour prendre un objet, il faut tendre le bras, associé l'objet à la main, et ramener le bras vers le corps. Pour que tout cela soit le plus transparent possible, il ne faut pas que l'intelligence ait à attendre que le bras soit bien tendu avant de demander l'association de l'objet avec la main. Tout est fait de manière automatique avec le mécanisme de Trigger (gâchette). Quand l'ia veut prendre un objet, elle demande au moteur d'installer un trigger sur telle animation avec tel objet. Plus précisément, elle dit à l'Objet « je veux prendre cet objet qui a pour identification uid ». A l'intérieur de l'Objet, il existe un C3DObjElement (Elément) qui est capable de gérer cet objet. Il y a tout, animation à effectuer au début de la prise d'objet, à quel point précisément associer l'objet. De ce fait l'intelligence n'a vraiment rien à faire de compliquer.

Cela signifie une chose: il ne peut y avoir qu'un seul Elément dans l'Objet qui puisse gérer la prise d'un objet particulier (une graine par exemple). Un Elément peut évidemment gérer plusieurs types d'objet différent.

Un mécanisme similaire au trigger est utilisé pendant le jet d'acide lorsque la fourmi attaque (qui utilise un système de particule).

## **E) SDL**

La librairie SDL (**Simple DirectMedia Layer**) est une couche d'abstraction de l'environnement très performante. Elle permet de développer avec le même code des applications graphiques pour Windows, Linux, BeOS, MacOS, FreeBSD, Solaris, ... Elle ne gère pas les boutons, listes déroulants,... mais fournit la fenêtre (ou éventuellement l'accès en plein écran). Elle permet d'utiliser OpenGL (elle gère le double Buffer) de manière transparente. Avec, l'application sera aussi bien compilable sous Windows que sous Linux (c'était le but non officiel de ce programme). Cette librairie est très rapide et très puissante. Par exemple, le portage sous Linux de Unreal utilise la SDL (en fait, la SDL a été développée à l'origine par un membre de l'équipe de portage de Unreal).

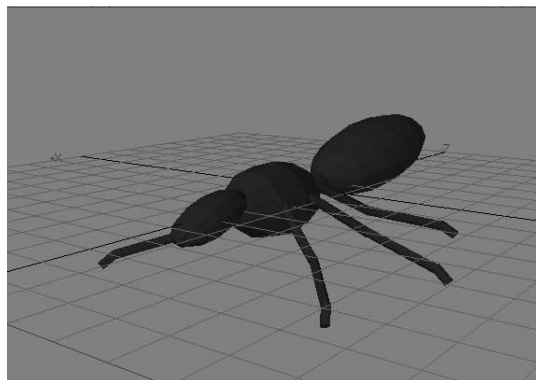
Ainsi, aucune API spécifique Windows n'a été utilisée dans le programme, nous n'avons utilisé que des appels standard (avec la STL) et des appels à la SDL.

De même, nous utilisons SDL\_Mixer pour gérer le son (musique et événement sonores).

Ainsi avec la SDL, nous n'avons pas eu à nous occuper des tâches rébarbatives telles que la création d'une fenêtre, la saisie des événements claviers et souris (qui en plus aurait été différente entre Windows et Linux), pour nous concentrer sur l'essentiel.

**Remarque:** La SDL utilise en interne des appels à DirectX sous Windows (DirectDraw pour les dessins 3D, DirectInput pour la gestion des entrées claviers et souris (voir même joysticks), et DirectSound pour la musique). Sous Linux, elle utilise des mécanismes internes pour émuler les parties qui n'existent pas sous ce système (pas d'équivalent de DirectDraw par exemple), elle utilise des appels directs au serveur X pour la souris ou le clavier,... C'est un « wrapper » de fonctions.

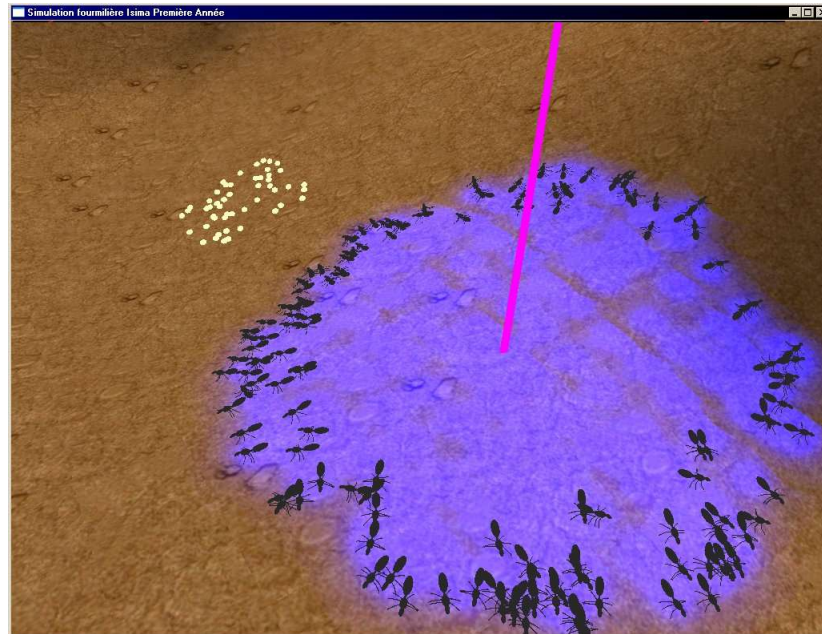
Description en fonctionnement



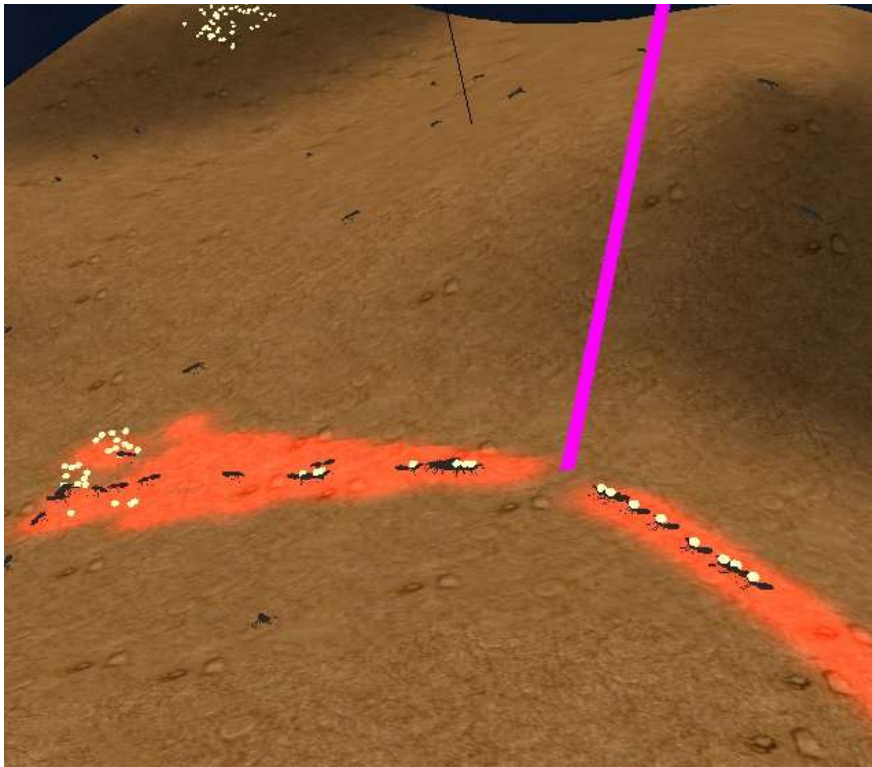
Fourmi en cours de « création »...

```
6:\documents personnels\Fichiers Visual C++\Fourmis\Fourmis_simu\Debug\objects\ouvrier...
Fichier Edition Chercher Afficher Outils Options Langage Aide
1 ////////////////////////////////////////////////////////////////////
2 // Définition de l'objet Fourmi
3 ////////////////////////////////////////////////////////////////////
4 // Arbre:
5 // root (0)
6 // |
7 // |-- Torax (1)
8 // |
9 // |--- Abdomen (2)
10 // |
11 // |--- tete (3)
12 // |
13 // |--- patte avant droite (4)
14 // |
15 // |--- patte milieu droite (5)
16 // |
17 // |--- patte arriere droite (5)
18 // |
19 // |--- patte avant gauche(4)
20 // |
21 // |--- patte milieu gauche(5)
22 // |
23 // |--- patte arriere gauche(5)
24 //
25 ////////////////////////////////////////////////////////////////////
26
27 // Objet : root (invisible)
28 ElementID = 0;
29 BoundingBox = -2.2 0 -3.0
30              2.2 0 -3.0
31              2.2 0 2.5
32              -2.3 0 2.5
33              -2.3 1.8 -3.0
34              2.2 1.8 -3.0
35              2.2 1.8 2.5
36              -2.3 1.8 2.5 ;
37
38 Object3D = NULL;
39 AnimNumber = 1;
40 AnimDefault = default;
41 AnimName= default;
42 AnimNext = NULL;
43 AnimFrameNumber = 1;
44 AnimFramePosition = 0.0 0.0 0.0 // pos
45                    0.0 0.0 0.0; // dir
46 ChildrenNumber = 1;
47 Children = 1;
48 ////////////////////////////////////////////////////////////////////
49
50 // Objet : Torax (fils de root)
51 ElementID = 1;
```

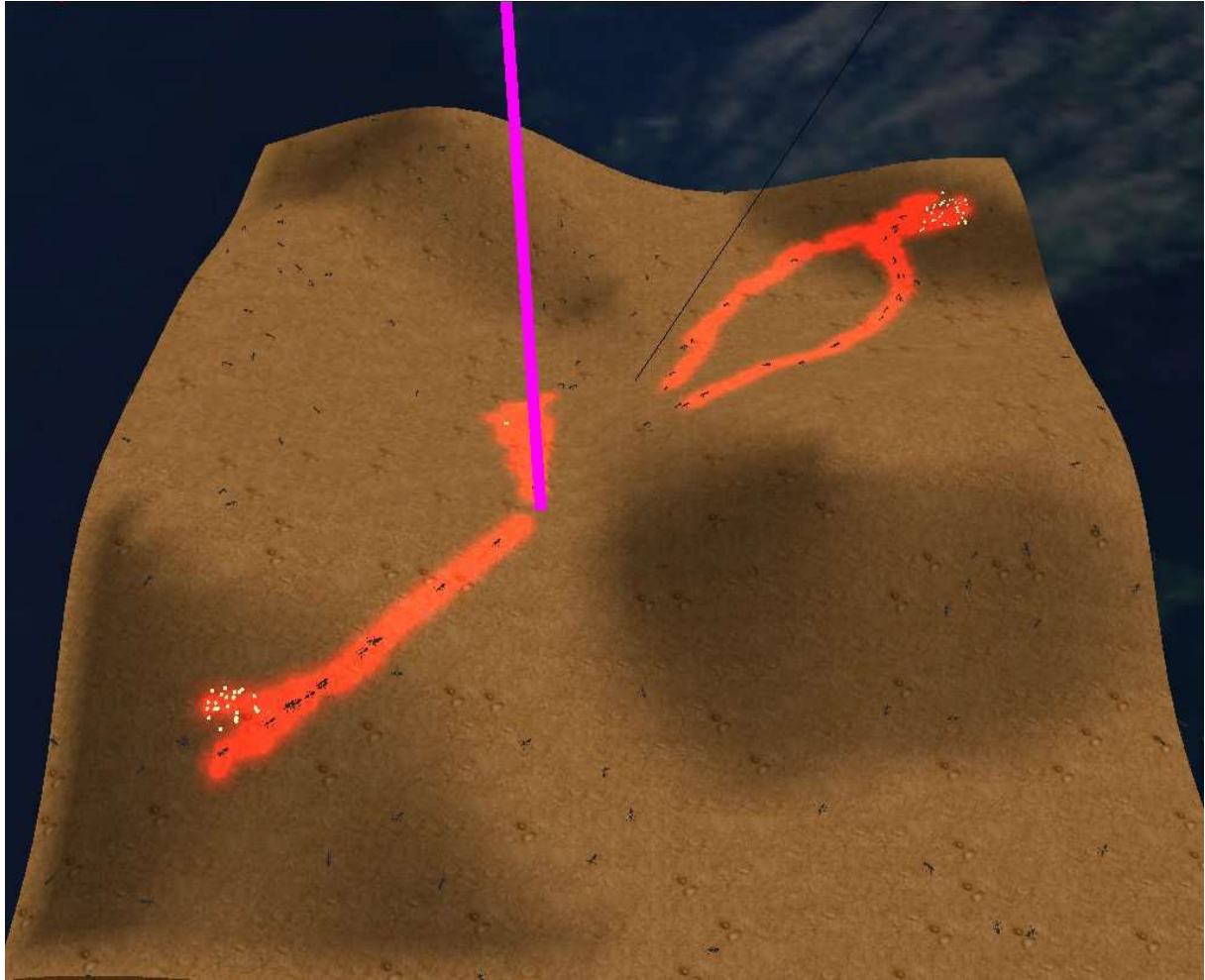
Edition du fichier de description des animations pour la fourmi « scout ».



Début du programme. (300 fourmis).



Les fourmis rapportent à manger. Les traces rouges correspondent à la phéromone «j'ai trouvé à manger, venez tous en chercher, c'est par ici !! »



Vue d'ensemble du déplacement des fourmis. On remarque qu'à ce stade du développement, les fourmis connaissent encore quelques problèmes pour savoir où aller.

### **3) Lumière et ombre**

Cette partie permet de faire la lumière sur le problème des ombres du projet ; en effet pour générer les ombres il faut tout d'abord créer une source de lumière. Nous avons simulé le soleil.

Le choix s'est porté sur une source de lumière directionnelle car les rayons du soleil sont tous de la même direction, en effet durant la journée la lumière ne provient pas du même endroit la terre ayant un sens de rotation. Dans le projet c'est la lumière qui bouge en fonction du monde.

Ensuite il faut créer les ombres, la méthode utilisée est la suivante :

On calcule les normales de tous les vertex (points) puis on fait le produit scalaire avec la source de lumière ce qui nous renvoie la valeur de l'ombre pour le point et ensuite OpenGL fait l'interpolation, et donc calcul l'ombre entre deux points.

Au début le produit scalaire était fait par le programme lui-même ce qui utilisait des ressources systèmes et le processeur. En utilisant les fonction générant les ombres d'OpenGL nous avons libéré certaines de ces ressources en effet OpenGL fait travailler le processeur et la mémoire de la carte graphique.

Exemples de rendu d'ombre :



Remarque : les ombres sur les objets de la carte ne sont pas calculés seul les bosses en possèdent. Donc pas d'ombres pour les fourmis.

#### 4) Le générateur de menu



Pour le menu nous avons choisit de le générer en effet tous les menus du programme sont créés à partir d'un fichier contenant les valeurs spécifiques. Donc le générateur interprète le fichier d'extension « .ini » qui se nomme menu. Il existe trois de ces fichiers un pour chaque menu : le menu d'entrée, le menu associé à la carte et le menu associé à la sortie de la carte.

Décomposition d'un fichier de menu :

Un fichier comprend un ensemble de pages ces dernières contiennent des ensembles d'objet. Ces objets sont interprétés et permettent de créer la fonction qui leur est associé.

Il existe quatre types d'objet :

- bouton : comprend une action qui lui est associé et change lorsque l'on passe le curseur au dessus donc il a les coordonnées des textures qui lui sont associés ;
- option : ce sont des textes qui permettent de changer les paramètres ils ont deux couleur la couleur change lorsque le curseur est au dessus, et il possède une liste de valeur pour l'option ;
- titres : sont de simple texture donc il n'y a que les coordonnées de la texture ;
- effet de transparence : permet de faire des effets de transparence sur le menu.

Remarque : chaque objet possède les coordonnées de l'endroit où il se situe sur le menu.

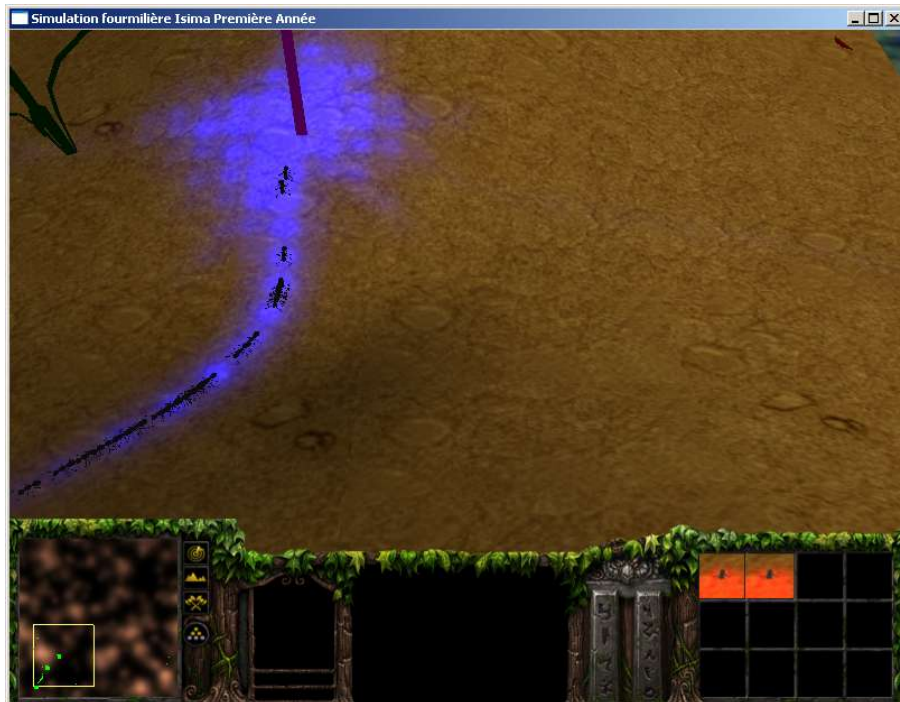


## Conclusion :

Nous aurions pu aller plus en profondeur en simulant plus de types de fourmis ou en ajoutant plus d'interaction en gérant les collisions et le fait que les fourmis sont capables de grimper sur les objets. Nous aurions pu aussi donner la possibilité de gérer le fait d'avoir plusieurs colonies ce que nous avons envisagé.

Mais il nous manquait du temps donc nous nous sommes arrêtés sur la simulation à l'extérieur de la fourmière d'une seule colonie.

Voici quelques captures d'écran de l'application en version (presque) finale:



Note: remarquez le memu « W--Craft III » qu'il nous est impossible de distribuer...

