

Binômes :

Azenkoud Younès – Damon Régis

**Projet de synthèse d'images et d'algorithmique
avancée**

2007

- IMAC 2009 -

PARTIE 1 :	4
ORGANISATION DU PROJET	4
I. REPARTITION DU TRAVAIL	4
1. CONCEPTION DES ALGORITHMES	4
2. IMPLEMENTATION :	4
3. DEBOGAGE ET ANALYSES DES RESULTATS :	4
II. ORGANISATION DU CODE	5
1. STRUCTURE DES DOSSIERS ET FICHIERS DU PROJET (DEPUIS LA RACINE):	5
2. LE MAKEFILE :	5
3. LES LIBRAIRIES :	6
III. ORGANISATION DES DONNEES DU PROJET	7
1. LES CONSTANTES :	7
2. LES STRUCTURES ELEMENTAIRES	8
3. LES STRUCTURES DE DONNEES	9
4. LE GRAPHE	11
PARTIE 2	13
CREATION INFORMATIQUE DE LA FOURMILIERE	13
IV. PARAMETRE DE L'APPLICATION	13
1. INTERPRETATION DES ARGUMENTS DU PROGRAMME	13
2. LECTURE DES INFORMATIONS	14
V. LECTURE ET INTERPRETATION DU FICHIER IMAGE	15
1. LA CREATION DES NŒUDS	15
2. LA CREATION DES ARC	17
3. LIAISON DES NŒUDS ET DES ARCS	18
4. EVALUATION DES PLUS COURT CHEMINS	19
PARTIE 3 :	21
INTERFACE UTILISATEUR	21
VI. AFFICHAGE	21
1. AFFICHAGE DES PIXELS	21
2. LE SYSTEME DE CALQUES	23
VII. L'INTERFACE	24
1. COMMANDES UTILISATEUR	24
2. EXEMPLE DE L'INTERFACE :	25

INTRODUCTION

Fourmimac est un projet mêlant d'algorithmique et de synthèse d'image, et dont les principaux points sont : la récupération des données à partir de fichiers (texte et image), la gestion informatique d'une fourmilière et son affichage. Ainsi, un tel projet permet d'appréhender toute les facettes du développement d'une application.

La complexité du sujet, que l'on peut assimiler à un cahier des charges, nous impose de bien définir et parcelliser les différentes étapes de réalisation du projet.

Avant de commencer à coder, la première partie du travail consiste à analyser le sujet pour définir les grandes phases de développement, les structures de données et les algorithmes principaux. Une fois les besoins clairement établi, les bibliothèques de fonctions propres à l'application, peuvent être implémentées.

Le présent dossier reprend cette démarche que nous avons adoptée pour concevoir le programme.

Ainsi, nous approfondirons, dans une première partie, l'organisation du projet, tant au niveau du code qu'au niveau de la méthodologie de développement et des structures de données.

Puis nous expliquerons les algorithmes concernant la récupération et la mise en forme des informations.

La troisième partie expliquera dans un premier temps l'interface utilisateur, puis décortiquera son implémentation.

Partie 1 :

Organisation du projet

I. Répartition du travail

Le temps et la complexité du développement du projet nécessitent une organisation rigoureuse. Après la distinction des grandes phases, on peut appliquer la même méthodologie pour chacune.

1. Conception des algorithmes

Nous avons entamé le développement de chaque phase (passage de l'image, du texte, création et gestion du graphe...) par une concertation sur l'algorithme à adopter. Cette algorithme prends la forme d'un pseudo code sur papier, ou de schéma pour les structure de données (graphe et settings).

Bien entendu, chaque grandes phases a dut être subdivisée en plusieurs petite phases. Par exemple, en ce qui concerne le « passage de l'image » (librairie *image.c/h*) on a : « lecture de l'image », « récupération des information des salles », « création du graphe »...

2. Implémentation :

L'implémentation de chacune de ses grandes phases à été répartie au sein du binôme, et s'est traduit par le développement d'une librairie dédiée (par exemple *image.c/h* est la librairie chargée de retirer l'information de l'image PPM).

3. Débogage et analyses des résultats :

L'inévitable phase de débogage s'est fait en parallèle d'une discussion sur les rectifications à apporter à l'algorithme et au code.

Par exemple on peut noter que le premier algorithme censé récupérer les informations d'une salle, à été abandonner au profit d'un algorithme plus simple et tout aussi performant.

II. Organisation du code

Le développement d'une telle application, nécessite une organisation solide du code et du projet. Cela s'est traduit par une hiérarchisation particulière des dossiers et des fichiers, un makefile élaboré, et un découpage pertinent des fonctions en bibliothèques.

1. Structure des dossiers et fichiers du projet (depuis la racine):

- `./bck` :
A intervalles réguliers, une archive est stockée dans ce dossier. Le nom de l'archive est du type `Antz_bck_jj.mm.aa(VersionType).tar` avec la date et le type de la version indiquant si elle reste à déboguer, si elle est en cours d'implémentation, si elle a été abandonnée...
- `./bin` :
Contient les bibliothèques compilées en objet `.o`.
- `./include` :
Contient les `.h` des bibliothèques développées.
- `./ini` :
Contient les fichiers d'initialisation de l'application. Cela va des fichiers texte de *settings* aux images, celles de la fourmilière comme des images d'arrière plan.
- `./src` :
Contient les `.c` des bibliothèques développées.
- `./main.c` :
Le fichier principal.
- `./main.o` :
Son objet.
- `./Antz`
L'exécutable.
- `./makefile` :
Le makefile du projet.

2. Le Makefile :

Nous avons conçu le makefile de notre projet pour qu'il soit le plus souple possible tout en respectant l'architecture de nos fichiers. La souplesse vient de l'utilisation des variables suivantes :

- `DEBUG` : { yes | no }
Permet d'ajouter ou non le `-d` pour la compilation.
- `CC` : défini le compilateur. { gcc | g++ }
Selon l'avancée du projet et l'utilisation de Glui ou non.
- `CFLAGS` :
Les arguments du compilateur.

- LDFLAGS :
Les liens vers les bibliothèques externes telle que *-IGL, -lglut, -lm...*
- TARGET :
Le nom de l'exécutable généré.
- MODULE_ROOT :
L'emplacement des librairies. { . }, dans le répertoire courant du projet comme nous venons de le voir.
- INC_PATH :
Emplacement du dossier *./include* depuis MODULE_ROOT.
- SRC_PATH :
Emplacement du dossier *./src* depuis MODULE_ROOT.
- OBJ_PATH :
Emplacement du dossier *./bin* depuis MODULE_ROOT.
- SRC :
Ensemble des noms des fichiers *.c* contenu dans *./src*.
- OBJ :
Ensemble des noms des fichier *.o* généré grace à SRC.
- INCLUDES :
ensemble des noms des librairies *.h* contenue dans *./includes*

De cette manière, nous pouvons, pour tout *.c* ajouté dans le répertoire « *./src* », généré son *.o* correspondant que l'on va stocker dans *./bin*. L'exécutable, qui prendra la valeur de TARGET, sera généré avec toutes les dépendances aux librairies contenu dans *./includes*.

L'ajout de nouvelles librairies est donc facilité, tout comme le changement de compilateur ou d'argument de ce dernier.

3. Les librairies :

Le découpage du sujet en grande phases de développement se traduit par le développement des librairies suivantes :

- globales.h :
Contient la définition des variables globales. Dans notre cas, une structure globale contient toutes les informations de l'application (voir partie III - *structures de données*).
- settings.h :
Ensemble des fonctions chargées de gérer les paramètres de l'application.
- strlib.h :
Opérations sur les chaînes de caractères. (Split, nombre d'occurrences d'une sous-chaîne dans une chaîne...).

- typedef.h :
Ensemble des structures de données utilisées avec leur fonctions associées (nouvel élément, affichage...).
- files.h :
Ensemble des fonctions chargées de la sauvegarde d'informations dans des fichiers (sauvegarde des informations du graph, de l'état d'une partie en cours...).
- image.h :
Ensemble des fonctions récupérant l'information de l'image PPM et construisant le graph.
- graphics.h :
Ensemble des fonctions gérant l'affichage OpenGL
- glutinit.h :
Callbacks de glut.
- antz.h :
Ensemble des fonctions de gestion de la fourmilière.

L'organisation du code, dès le début du projet, a compté facilité son développement.

III. Organisation des données du projet

L'analyse du sujet dans son ensemble, a vite révélé l'utilité de concevoir des structures de données organisées, aussi bien pour la partie 1 que la partie 2. Ainsi les structures nécessaires dans la partie 1 ont été conçues en regard des structures qu'impliquait la partie 2, pour faciliter le passage de l'une à l'autre des parties.

Chaque structure nécessaire possède des fonctions qui permettent de les gérer. Cela va de la fonction d'allocation à la fonction d'affichage sur la sortie standard, qui affiche les informations complètes et formatées d'une structure, facilitant grandement le débogage.

CETTE PARTIE CONCERNE TYPEDEF.H

1. Les constantes :

Pour améliorer la compréhension du code, ainsi que l'implémentation de certaines structures de données, nous avons eu recours à des constantes et

à des variables de type *enum*. Certaines définitions de variables de type *enum* se sont faites conjointement à la définition d'une constante, qui fixe le nombre maximal d'élément pour un certain *enum*.

- Season : {printemps, ete, automne, hiver}
Nom des saisons.
- Attributes : {vitesse, conso, coefConsoReine, consoReine, dureeVie, resistance}
Nom des caractéristiques générales des fourmis, reine comprise.
Cette structure est associée à la constante **ATTRIBUTES_NUMBER**.
- Task : {pucceron, exterieur, occupReine, attente, danger}
Nom des différentes tâches que peut entreprendre une fourmi. Nous avons ajouté **danger** afin de déterminer un comportement spécifique lors d'un coup de pied dans la fourmilière.
Cette structure est associée à la constante **TASKS_NUMBER**.
- Room : {ciel, sol, reine, reserve, ferme, transit, sortie, galerie}
Nom des différents types de salle de la fourmilière. Nous avons ajouté **galerie** car ce champ était utile pour nos algorithmes.
Cette structure est associée à la constante **ROOM_TYPES_NUMBER**.
- FERME = 0 et EXTERNE = 1 :
Dont nous nous servons pour l'organisation des données concernant les différentes productions.

2. Les structures élémentaires

Les structures simples sont les structures dédiés aux algorithmes et non aux des donnée de l'application, dans le sens des paramètres du jeu et de la fourmilière. Elles possèdent toutes les fonctions de base de ce type de structure, plus une fonction affichage pour le débogage.

- Color :
Stock les quatre composantes couleurs d'un pixel RGBA. Nous avons ajouter la composante *alpha* afin d'avoir un affichage plus souple (notamment dans la gestion de plusieurs calques d'affichages superposés avec transparences).
- Image :
Stock les informations relative à une image PPM : Son chemin (relatif ou absolu), sa largeur, sa hauteur, sa profondeur de codage, un table à une dimension de structure **Color** de taille *hauteur*largeur*.
- Pile :
La structure Pile contient un élément *long int item* correspondant à un indice d'un pixel. Cette pile nous sert dans le parcours du chemin d'un centre d'une salle à un autre centre.
- File :
Structure de type File, contenant un pointeur sur un *nœud* du graphe et sur un *arc* associé. Elle nous sert pour l'algorithme de Dijkstra.

3. Les structures de données

- Settings :

Settings est le type de la seule et unique variable globale, c'est une structure qui permettra d'atteindre n'importe quelle donnée de l'application. On distingue quatre grands types d'information que contient *Settings*: les informations de la fourmilière, le graphe, l'image de la fourmilière et les informations d'affichage (calques affichés,...)

```
typedef struct
{
    /* Version de l'application */
    int version;

    /*** image de la fourmiliere ***/
    Image *image;

    /*** Couleurs par type de piece ***/
    Color **room;          /* l'indice est de type Room
                           compris entre reine et
                           sortie*/

    /*** Graph ***/
    byte arcNb;           /* nombre d'arc */
    struct arc **arcArray; /* tableau d'arc */
    byte roomNb;         /*nombre de noeud */
    struct node **roomArray; /* tableau de noeud */

    /*** Fourmiliere ***/
    Fourmiliere *f;

    /*** Affichage ***/
    byte *layer;         /* nombre de calques total */
    Color **layerColor; /* couleur associé à un calque */
    Image **bck;        /* tableau image de fonds, les
                           indice sont les saisons */
    Image *display;     /* image final de tout les
                           calques a afficher en
                           surimpression */

}Settings;
```

- Production :

Cette structure comporte les deux informations liées à une production spécifiques : la durée et la quantité.

```
typedef struct
{
    float quantite;      /* quantité d'un type de production */
    float *duree;       /* tableau d'indice EXTERNE ou Season */
}Production;
```

- Fourmi :

Structure qui représente une fourmi : position, action en cours...

```
typedef struct
{
    float age; /* age de la fourmi */

    Task tacheCourante; /* tache courante */

    /** position **/
    Node *arrivee; /* Salle d'arrivée */
    Node *intermediaire; /* Salle intermédiaire à laquelle
                           la fourmi doit se rendre selon
                           le plus court chemin pour
                           atteindre arrivée */

    struct arc *arcActuel; /* arc sur lequel se trouve la
                           fourmi */

    int position; /* position sur l'arc */

    /** production **/
    float attenteProd; /* temps écoulé depuis le début
                        de la récolte */

}Fourmi;
```

- Fourmilière :

Contient toutes les informations relatives à la partie 2 du sujet, les nœuds du graphe classé selon le type de salle qu'ils représentent, et les variables définissant l'état temporaire de la fourmilière (nombres de fourmis, saison courante, temps passé...)

```
typedef struct sfourmiliere
{
    float cn; /* Coeff Nourriture */
    float cr; /* Coeff Reine */
    float cc; /* Coeff Conseil */
    float alpha; /* coeff Alpha
                 */
    float reserves; /* Reserve à un instant t */
    float victory; /* Nourriture victoire */

    Season currentSeason; /* Saison courante */
    float dureeSaison; /*durée saison */

    Salles **roomType; /* tableau de salles dont les
                       l'indice est de type Room */
    Production **prod; /* tabelau dont l'indice est
                       EXTERNE ou FERME */

    float *antsAttr; /* tableau d'attributs des
                     fourmi dont l'indice est
                     de type Attributes */

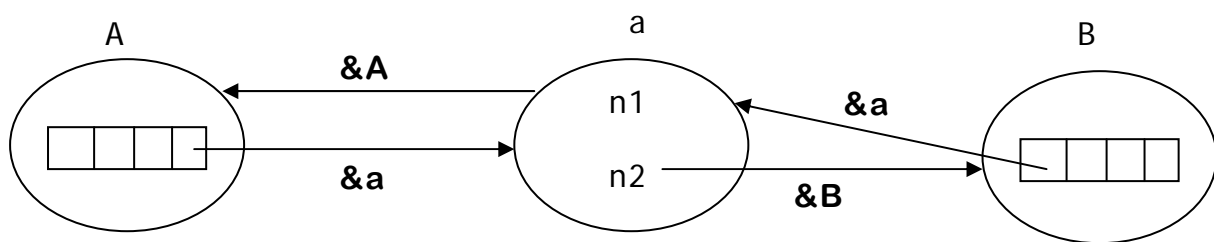
    int nbFourmi; /* nombre de fourmi a t */
    Fourmi **fourmis; /* toutes nos fourmis */

}Fourmiliere;
```

4. Le graphe

Le graphe que nous avons implémenté, utilise des nœuds génériques qui s'adaptent à tous les types de salles. Chaque nœud possède les caractéristique de la salle qu'il représente : un id qui sera l'indice du nœud dans une représentation du graphe par tableau ; la taille de la salle ; le type de la salle ; son centre ; le nombre de fourmi présentes ; le nombre de nourriture ; la quantité maximum de nourriture.

Voici un schéma de représentation d'une partie d'un graphe avec n1 relié avec n2 par un arc a :



- struct arc :

Comprend un pointeur pour chaque salle qu'il relie, ainsi, si on veut orienter le graphe, il suffit de mettre l'un de ces deux pointeur à NULL selon l'orientation que l'on veut donner à l'arc.

La structure arc comprend les informations suivantes : les deux distances (*distance1*, *distances2*) distance en pixels du centre de la salle pointée par n1 au centre de la salle pointée par n2 ; la pile des coordonnées des pixels composants ses deux chemins, respectivement stockées dans *path1*, *path2*.

Le plus court chemin sera stocké dans (*distance1*, *path1*).

L'arc contient aussi les portes et la pondération de la galerie.

```

struct arc
{
    /** the pointer to each room the arc links ***/
    struct node *n1;
    struct node *n2;

    /** Distances from the two path possible for the arc ***/
    int distance1;
    int distance2;

    /** the two paths as a Pile structure ***/
    Pile *path1;
    Pile *path2;

    /** the 4 gates of the path ***/
    long int gate[4];

    /** coefficient of the arc ***/
    double coef;
}

```

- Path :

Chaque nœud contiendra un tableau de n cases comprenant n structures Path. Cette structure contient le chemin le plus court depuis le nœud courant jusqu'au nœud d'indice n.

La structure contient aussi le poids total du chemin. Cela servira, par exemple, à ce qu'une fourmi estime de manière rapide la réserve non pleine la plus proche.

Chaque étape du chemin le plus court est une File comprenant l'adresse du prochain nœud à atteindre ainsi que l'arc par lequel l'emprunter. De cette façon, nous prenons en compte le fait que deux salles A et B puissent être reliées directement par plus d'un arc.

```
struct Path
{
    struct File
    {
        struct node *item;
        struct arc *a;
        struct File *next;
    } *path;
    double distance;
}
```

Ce projet nous a montré à quel point l'organisation des données est un point essentiel du développement d'une application. Il est alors tout à fait judicieux d'essayer d'établir les structures de données solides dès le départ, mais cela ne se fait pas sans une idée assez précise des algorithmes à employer par la suite.

Partie 2

Création informatique de la fourmilière

Une fois les structures de données clairement établies, il reste à les remplir à l'aide d'un fichier image et d'un fichier texte.

Nous décrirons dans un premier temps l'algorithme retenu pour la lecture des informations dans le fichier texte. Les fonctions qui en découlent forment la librairie *settings.h*.

Nous verrons ensuite comment nous avons conçu le graphe à partir de l'image, en décrivant les fonctions de la librairie *image.h*.

IV. Paramètre de l'application

Un fichier texte au formatage prédéfini va déterminer certaine valeur du programme. Ce concept consiste à externaliser le maximum d'information afin d'avoir une grande souplesse d'utilisation de l'application.

CETTE PARTIE CONCERNE SETTINGS.H

1. Interprétation des arguments du programme

Afin de conserver cette souplesse d'utilisation nous avons créé la fonction :

```
Settings *parseSettings(int argc, char** argv);
```

Cette fonction est chargée d'interpréter les valeurs passées en argument de notre programme et d'appeler les fonctions correspondantes.

Pour le moment nous pouvons spécifier un fichier *.txt*, pour l'initialisation des paramètres de l'application, via la commande *-s* :

```
./Antz -s ./repertoire/initialisation.txt
```

Le fichier d'initialisation `./ini/defaultSettings-x.txt` est prévu par default si l'argument n'est pas utiliser.

2. Lecture des informations

La fonction :

```
Settings *readSettingsFile(char *fileName);
```

Prends en paramètre le chemin du fichier d'initialisation et renvoi la structure globale *Settings* de l'application, remplit avec les valeurs du fichier.

- La fonction va vérifier la validité du formatage de chaque ligne.
- Lorsque le champ image est trouvé, elle vérifie qu'il s'agit bien d'une image PPM auquel cas elle appellera la fonction *readPPM* (image.h) qui crée une structure image et la remplit. Le pointeur de cette structure sera stocké dans la structure globale *s*.
- si jamais un problème survient pendant la lecture de l'image, l'image par défaut *default.ppm* est choisie à la place.
- Lorsqu'un champ renseignant la couleur d'une salle est trouvée, une structure couleur sera stockée dans le tableau *s->room[r]*, à l'indice *r*, de type *room*.
- si aucun champ ne correspond à la version 1 d'un fichier d'initialisation, on regarde la version du fichier préalablement stocké dans *s->version*.
- si la version est 2, on appelle la fonction :

```
int readSettingsFile2( char *attr, char *value )
```

- cette fonction return 1 si le champ a bien été analysé, 0 si le champ est inconnu. Dans ce cas, un message est écrit sur la sortie standard indiquant le champ inconnu.
- En cas de succès, le nom du fichier d'initialisation est écrit sur la sortie standard en indiquant que la lecture s'est bien passée.

Cet algorithme à été pensé de manière prendre en compte n'importe quel type de version. Pour une version 3 de FourmImac, il sera rapide et aisé d'ajouter une nouvelle fonction *readSettingsFile3(char *attr, char *value)*.

Des systèmes de fichiers par défaut ont été implantés pour que l'application se lance d'elle même si les valeurs entrées en argument du programme sont erronées.

V. Lecture et interprétation du fichier image

Une fois que toutes les informations du fichier texte sont dans *s*, nous pouvons extraire les informations de l'image grâce à la fonction :

```
Node *graph = createGraph( s );
```

Cette étape va comprendre quatre sous-étapes différentes :

- la création des nœuds
- la création des arcs
- la liaison entre nœud et arc
- le calcul du chemin le plus court de chaque nœud A vers chaque nœud B

CETTE PARTIE CONCERNE IMAGE.H

1. La création des nœuds

Nous avons initialement développé un algorithme qui parcourait chaque salle de l'extérieur vers l'intérieur en formant une spirale jusqu'au centre. Il nous permettait de récupérer la surface de la salle en même temps que les portes des galeries et de trouver le centre d'une salle convexe.

Cependant, cet algorithme implique que l'on parcourt le graphe en même temps que l'on en récupère l'information. La complexité était telle que ni un algorithme récursif, ni un algorithme itératif n'était avantageux.

Nous avons opté pour un système de pondération de chaque pixel. Nous voulons créer un tableau *byte *img*, de longueur *h*w*, dont toutes les valeurs sont initialement à 0, et qui, pour chaque zone de la fourmilière (salle ou couloir), associe un certain indice.

- **Fonction:**

```
byte *getRoomAreas( Settings *s );
```

- On définit la variable N qui donnera l'indice de chaque zone.
- On parcourt l'image en entier.
- Si on rencontre un pixel de salle ou de couloir : on vérifie s'il n'est pas voisin d'un pixel appartenant à une zone, c'est-à-dire un pixel déjà évalué et pour lequel le poids est supérieur à 0 :
- Pour un pixel d'indice i quelconque dont le couleur correspond à une salle ou un couloir, on teste les pixels voisins suivant (en rouge) :

$i-(w-1)$	$i-(w)$	$i-(w+1)$
$i-1$	i	

- on ne teste qu'eux uniquement puisque l'on parcourt l'image de haut en bas et de droite à gauche. L'ordre des testes est donc $i-(w-1) \rightarrow i-(w) \rightarrow i-(w+1) \rightarrow i-1$
- Si l'un de ses pixels est de même couleur et a un poids, on fixe celui de i à ce poids là.
- Si ces 4 voisins ont un poids nul et ne sont pas de la couleur de i , on considère que l'on a découvert une nouvelle zone de l'image, donc on incrémente N et on fixe le poids de i à N .
- Voici un exemple de tableau généré, sur deux salles (Rouge et Bleu, reliées par une galerie verte) :

0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0	2
0	3	3	3	4	0	0	1	0	2
3	3	3	3	4	4	4	1	1	2
3	3	3	3	4	4	4	1	1	1
0	3	3	0	4	4	4	1	1	1
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

- On remarque que des pixels de la pièce bleue n'ont pas le même poids
- Pour réajuster ces poids, on doit appliquer le même algorithme mais à l'envers, c'est-à-dire de $h*w$ jusqu'à 0 en testant les pixels suivants :

--	--	--

	i	i+1
i+(w-1)	i+w	i+(w+1)

Avec cet algorithme avoir une connaissance de toutes les zones de l'image, et donc savoir la taille de salle aussi bien concave, que convexes.

On va alors pouvoir générer tout les nœuds du graphe dans un tableau que l'on stockera dans *s->roomArray*.

2. La création des Arc

Avec le tableau généré précédemment, on va pouvoir créé les structures arc avec la fonction :

```
byte *getImageInfo( byte *img, Node ***room, int *nbRoom, Settings *s );
```

Pour cela on doit établir certain information relative aux arcs : les portes, la pièce qu'il relie, et les chemins haut et bas en huit connexités. Au passage, la fonction *getImageInfo* crée les nœuds avec le tableau généré en 1. et crée en même temps le tableau d'information des arcs.

- On parcourt le tableau précédent.
- Si un pixel pondéré est un pixel de couloir, on teste en 4 connexités ses voisins (sur cette algorithme, ce teste permet d'avoir des chemins en 8 connexité).
- Si, parmi ces voisins, il n'a que des pixels de *sol*, on lui affecte le coefficient 255.
- Si, parmi ces voisins, il n'a que des pixels de *galerie*, on lui affecte le coefficient 0.
- Si, parmi ces voisins, il a un pixel d'une pièce sans pixel de *sol*, on lui affecte le coefficient 254.
- Si, parmi ces voisins, il a un pixel d'une pièce et un pixel de *sol*, on lui affecte le coefficient de la pièce voisine (en regardant dans le tableau précédent).
- On obtient alors le tableau suivant :

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	3	0	0	0	0	0
0	0	0	0	254	255	1	0	0	0
0	0	0	0	254	0	254	0	0	0

Semestre 2				- FourmIMAC -			IMAC 2009		
0	0	0	0	255	255	1	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

- On note qu'une porte est manquante. Ce problème sera résolu lors de l'évaluation des chemins d'une galerie. (L'algorithme considérera que, dans la continuité du parcours de la galerie (voir plus bas), le premier pixel de poids 255 après un pixel de poids 254 est en fait une porte, car les chemins formés par les pixels de poids 255, sont obligatoirement séparés des chemins formés de pixels de poids 254, par une porte)

Il reste alors à créer les deux chemins possibles d'une galerie en empilant les indices de chacun des pixels du chemin.

- **Fonction :**

```
void getArcInfo( byte *img, struct arc ***arcArray, int *nbArc,
Settings *s );
```

- Cette fonction va parcourir ce deuxième tableau.
- Lorsqu'elle rencontre une porte (0<poids <254), elle vérifie qu'elle n'appartient pas à un arc créé.
- Si c'est le cas, on se situe à la porte 1 de la galerie.
- La fonction parcourt la première branche de la galerie, c'est-à-dire qu'elle suit les pixels de poids 255 jusqu'à rencontrer une autre porte, la porte 2.
- A ce moment là, elle a empilé tous les pixels du premier chemin.
- Elle va maintenant se placer sur la porte d'en face, la troisième, à l'aide des pixels de poids 254.
- Une fois arrivé à la porte trois, elle parcourt le deuxième chemin de la galerie jusqu'à la quatrième et dernière porte.

A ce moment là, on a créé tous les arcs à partir des galeries.

3. Liaison des nœuds et des arcs

- **Fonction :**

```
Node *createGraph( Settings *s );
```

- Pour tous les arcs créés, la fonction va les lier aux nœuds correspondant. Cela se fait en regardant le poids de chaque portes d'un arc, qui correspond à l'identifiant d'un nœud.
- Lorsque qu'on lie un arc à une salle, on empile les indices des pixels situés sur le segment [porte [i], centre de la salle].

En même temps que l'on lie les nœuds aux arcs correspondants, on complète chaque chemin de chaque arc, par la ligne droite qui relie la porte

du chemin au centre de la salle.

- **Fonction :**

```
File *lineFromItoJ( long int i, long int j, Settings *s );
```

- cette fonction empile les indices des pixels situés entre l'indice i et j.
- La fonction utilise l'équation de droite $y = a.x + b$.

Ainsi, nous pouvons dès à présent calculer les coefficients de chaque arc :

- **Fonction :**

```
void arcPonderation( struct arc **a, byte arcNb );
```

- Pour tout arc a, cette fonction va placer dans a->distance1 la distance du plus court chemin et dans a->n1 la Pile des indices des pixels du plus court chemin.
- Elle va alors effectuer la somme de tout les a->distance1.
- Puis, pour chaque arc a, elle calculera le coefficient :
 $\text{coeff}(a) = a\text{->distance1} / \text{distanceTotale}$
 Chaque arc possède désormais son poids normalisé.

Nous avons à présent généré le graphe depuis l'image, selon la structure de donnée établie au départ.

CETTE PARTIE CONCERNE SETTINGS.H

4. Evaluation des plus court chemins

On peut désormais calculer l'ensemble des plus courts chemins, pour tous les nœuds du graphe. On utilisera donc l'algorithme de Dijkstra. Pour ce faire nous avons eu recours à deux structures élémentaires : une structure de type Files et la structure Path, qui contient le chemin le plus court sous forme de File et le poids de ce chemin.

- **Fonction :**

```
struct Path *createPath( Node *begin, Node *end );
```

- La fonction createPath est l'implémentation de l'algorithme de Dijkstra. Elle prend donc en paramètre le nœud de départ et le nœud d'arrivée.

- La fonction va retourner une structure Path.
- La File de la structure Path contient les pointeurs successifs des nœuds du plus court chemin, dans le sens NoeudDeFin $\rightarrow N_{i-1} \rightarrow \dots \rightarrow N_1 \rightarrow$ NoeudDeDépart.

A partir de cette fonction, on veut calculer toute le plus court chemin possible :

- **Fonction :**

```
void createAllPaths( );
```

- On stock dans chaque nœuds, un tableau de structure Path de n élément, n étant le nombre de nœuds du graphe.
- Pour le Nœud d'indice i, i appartenant a {0, ..., n-1}, Nœud[i]->tRoom[i] == NULL .
- Afin d'optimiser cette algorithmme qui avait une complexité de (n^2) , nous avons ajouté une condition au calcul de tout les plus court à partir d'un nœud.
- A partie d'un certain Nœud A, lorsque l'on veut calculer le plus court chemin de vers un nœud B, vérifie si le plus court chemin de B vers A n'a pas déjà été calculé.
- Si c'est le cas, on stock le pointeur vers la structure Path, correspondant à B->A, dans A->tRoom[B].
- Ainsi, on économise des instructions et de l'espace mémoire.

Cette étape et l'étape clés du projet.

En effet, le système de pondération de chaque pixel est très souple et permet de garder le maximum d'informations en un minimum de traitement.

Nous avons attaché de l'importance à concevoir un algorithmme qui puisse trouver un graphe pour n'importe quel type de salle, concave ou convexe.

Partie 3 :

Interface Utilisateur

A l'issu de la partie I du sujet, nous avons souhaité nous pencher davantage sur l'affichage graphique et l'interface utilisateur. Là encore, il s'agissait de penser de la structure de données et de l'algorithme de façon modulable, le but étant de pouvoir ajouté des couches de données, en surimpression, de manière rapide et aisée.

Nous avons donc opté pour un système de « calques ».

VI. Affichage

L'affichage de l'image de la fourmilière a, très tôt, été un impératif, notamment pour le débogage lors de la création du graphe.

Mais au fur et mesure de l'accumulation des données à afficher, nous avons créé un système de calques qui permet d'ajouter des couches de données en surimpression de manière structuré et rapide.

1. Affichage des pixels

CETTE PARTIE CONCERNE GRAPHICS.H

L'affichage des pixels pose plusieurs problèmes : pour que l'interface soit un minimum interactive, il faut que l'image puisse être déformée.

Nous avons utilisé deux moyens de représenter les pixels dans cette optique :

- Affichage par quadrilatères :

Cette solution consiste à considérer un pixel comme un quadrilatère et à l'afficher comme tel :

```
void drawPixel( long int a, Color *c )
{
    int x0 = a->s->image->width;
    int y0 = (a/s->image->width);

    int x1 = x0+1;
    int y1 = y0+1;

    glColor4ub( c->r, c->g, c->b, c->alpha );

    glBegin(GL_QUADS);
        glVertex2i( x0,    y0 );
        glVertex2i( x0,    y1 );
        glVertex2i( x1, y1 );
        glVertex2i( x1, y0 );
    glEnd();
}
```

Nous avons utilisé cette solution pour toutes les informations en surimpression (chemin le plus court, ensemble des chemins, graphe...).

- Affichage brut :

Cet affichage consiste à utiliser les méthodes OpenGL suivantes :

```
glPixelStorei( GL_UNPACK_ALIGNMENT, 4 );
glDrawPixels( img->width, img->height, GL_RGBA, GL_UNSIGNED_BYTE,
img->data[0] );
```

Où `img->data[0]` est un tableau à une dimension de *byte*.

Afin de générer ce tableau, nous avons implémenté la méthode :

```
byte *convertToByteArray( Image *img )
```

qui convertit un tableau unidimensionnelle de structure `Color` en un flux d'octet `RGBA`.

Il reste à gérer la déformation de la fenêtre `glut`. Dans la Callback *reshape* de `glut`, nous avons insérer ce code :

```
float dx      = ((float)w/((float)s->bck[0]->width);
float dy      = ((float)h/((float)s->bck[0]->height);
glPixelZoom( dx, dy );
```

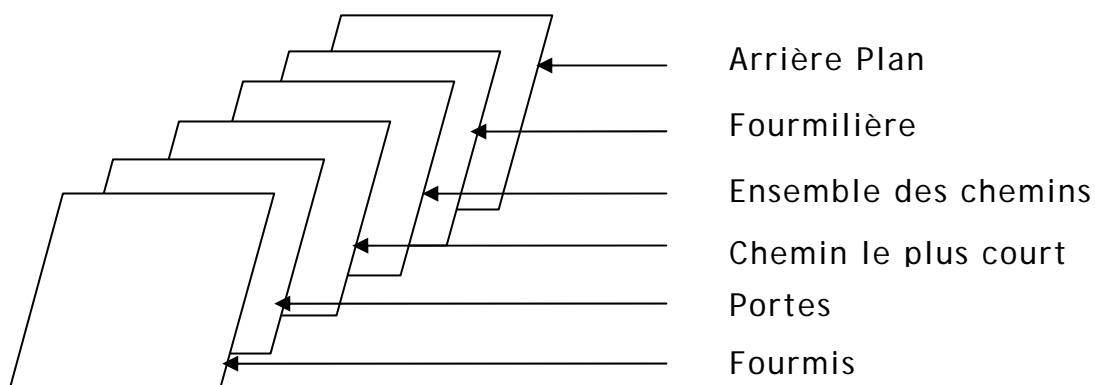
Qui utilise la méthode de zoom d'OpenGL.

Comme nous pouvons le voir dans le code, nous avons réservé cet affichage pour les images d'arrière plan, celle qui nécessite de réafficher le plus de pixels.

2. Le système de calques

Le problème de l'affichage en surimpression est que nous devons superposer plusieurs couches de données qui ne doivent pas interférer les unes avec les autres.

Nous avons donc choisi la hiérarchie de calques suivant :



Si le calque des fourmis doit être affiché, les fourmis prendront l'avantage sur tous les autres calques situés en dessous dans la hiérarchie.

L'ordre des de la variable de type *enum layer* est dans cet ordre.

La structure globale *s* comporte un tableau de *byte layer* qui comporte autant de cases qu'il y a de calques. Si `layer[gates] = 1`, cela veut dire que le calque *gates* doit être affiché.

Chaque calque est associé à une couleur. Le tableau de couleur *layerColor* définit la couleur dans laquelle l'information doit être affichée. Ainsi on peut très rapidement et facilement changer la couleur par laquelle les fourmis sont représentées.

Chaque calque possède une fonction qui réactualise l'affichage de ce calque particulier. Par exemple, pour les calques de l'ensemble des chemins, la fonction qui réactualise ce calque est :

```
void updatePaths();
```

On chaque fois que l'on met à jour un calque, on doit réafficher tout les

calques qui sont à un niveau hiérarchique plus élevé, dans la mesure où $layer[l] = 1$, c'est-à-dire qu'on veut l'affichage de ce calque.

Cette fonction se trouve dans `typedef.c` :

```
void updateFromLayer( int i );
```

Lorsqu'un calque est réactualisé, tous ces pixels de couleur non nulle sont en fait copier dans `s->display` qui est la résultante de tous les calques de surimpressions.

L'exception du calque d'arrière plan, tous les calques qui sont réactualisés, sont en fait copier dans `s->display` dans l'ordre d'importance des calques.

Au final on affiche `s->display` grâce à la fonction de `graphics.h` :

```
void drawImage( Image *img )
```

Nous avons vite trouvé l'intérêt d'organiser de manière solide l'affichage. La structure que nous avons adoptée permet d'ajouter de nouveaux calques assez rapidement.

VII. L'interface

1. Commandes utilisateur

CETTE PARTIE CONCERNE GRAPHICS.H

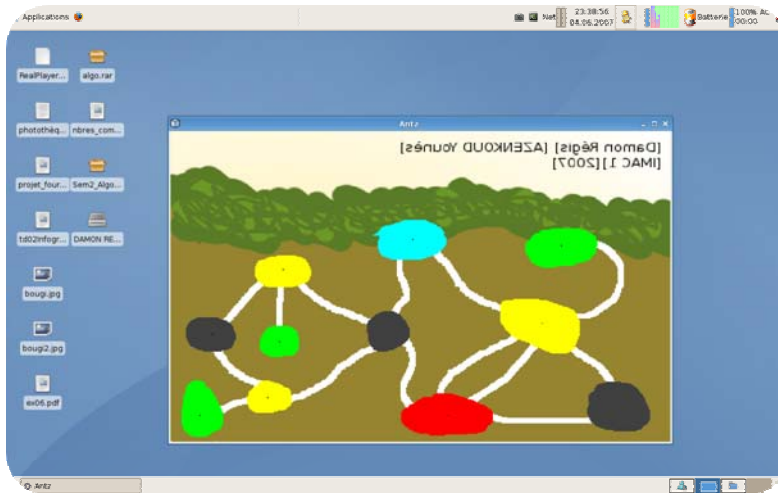
Pour ce qui est de l'interface, nous avons privilégié une interface lisible. Ce se traduit par le fait que la plupart des commandes se font au clavier.

Voici les commandes dont l'utilisateur dispose :

- **q** : Permet de fermer la fenêtre glut.
- **g | G** : Active (respectivement désactive) l'affichage de la fourmilière.
- **p | P** : Active (respectivement désactive) l'affichage des chemins.
- **s | S** : Active (respectivement désactive) l'affichage des chemins les plus courts.

- d | D : Active (respectivement désactive) l'affichage des portes.
- a | A : Active (respectivement désactive) l'affichage des fourmis.

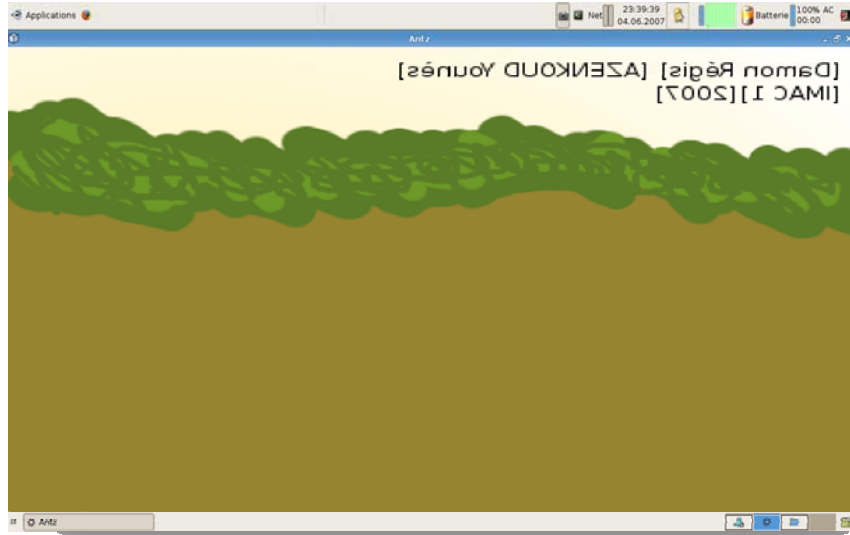
2. Exemple de l'interface :



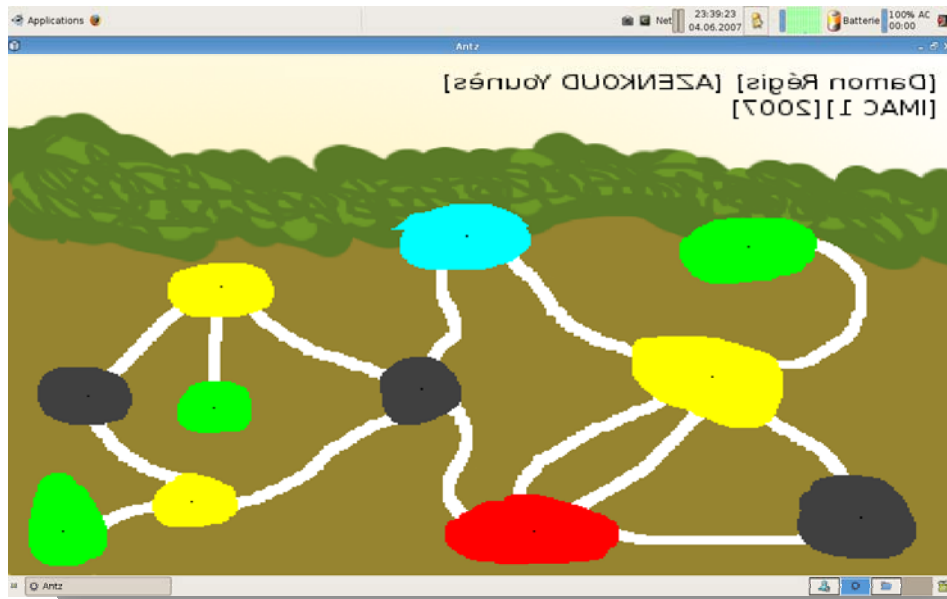
Etirement rétrécissement de la fenêtre.



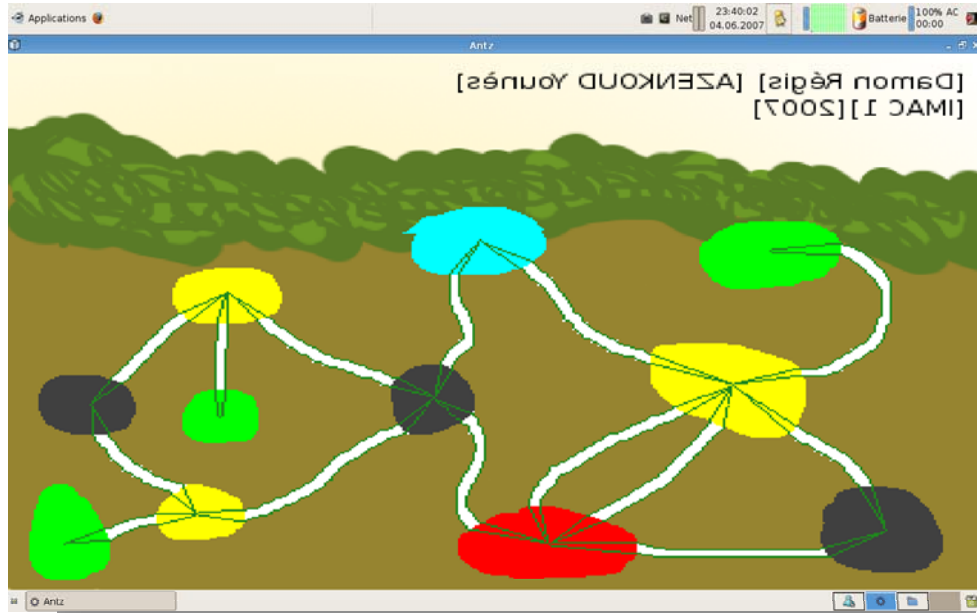
- Background :



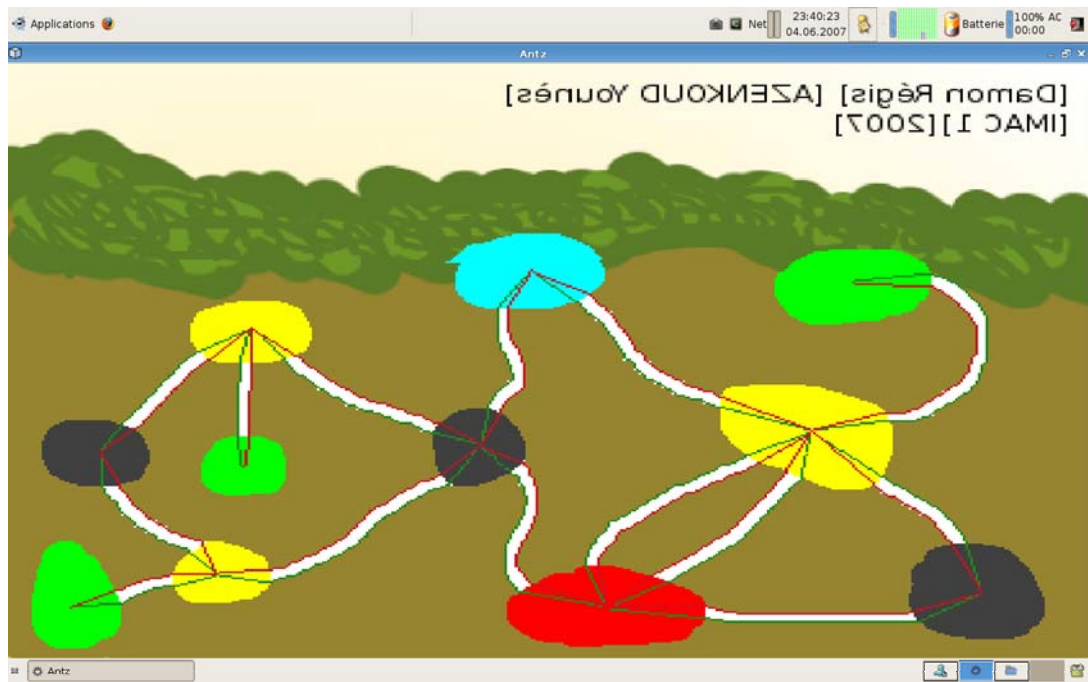
- Graphe (fourmilière);



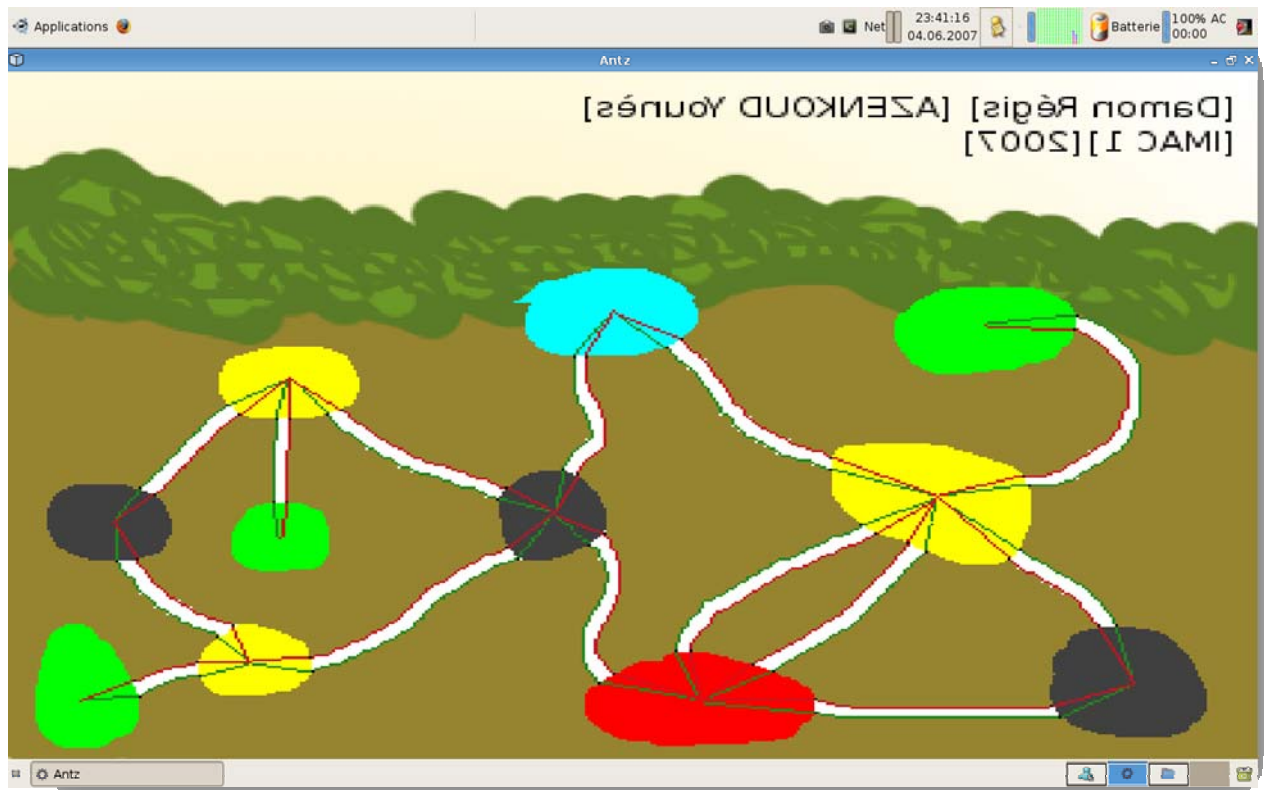
- Ensemble des chemins :



- Chemins les plus court :



- Portes :



CONCLUSION

Ce projet fut très enrichissant. Il nous a permis de comprendre le déroulement du développement d'une application et l'importance de son organisation.

Il nous a aussi permis de maîtriser des notions d'algorithmique fondamentale comme la récupération de données depuis des fichiers texte ou image, la conception et la gestion d'un graphe.

Nous regrettons d'avoir perdu un peu trop de temps sur un algorithme complexe. Cependant, ce projet nous a apporté une vision plus large sur la méthodologie à adopter lors d'un développement d'un logiciel, ce qui devrait nous permettre de mieux éviter ce genre de contretemps.

En l'état, nous avons apporté un soin particulier à la réalisation de la première partie, pour avoir une base solide lors de l'implémentation de la seconde partie, dans l'algorithme est déjà conçu sur le papier.

Nous espérons ainsi que notre fourmilière sera peuplée d'ici la soutenance.

Bibliographie - Annexes

Spécifications de Glut 3.0

Spécifications d'OpenGL

(à la racine du dossier)