

# Projet C++ / Les L-systems

Julien VAN DEN BOSSCHE/Benoît MOULIN

20 janvier 2003

## Table des matières

1	<u>Introduction</u>	2
2	<u>Visualisation du problème</u>	2
3	<u>La classe symbole</u>	3
4	<u>La classe regle</u>	3
5	<u>La classe axiome</u>	3
6	<u>La classe l_systeme</u>	3
7	<u>La classe lecteur</u>	4
8	<u>La classe zoneDessin</u>	4
9	<u>La classe interpretGraphique</u>	4
10	<u>La classe interface</u>	5
11	<u>Problèmes rencontrés</u>	5
12	<u>Copie d'écran</u>	6
13	<u>Diagramme UML</u>	9

# 1 Introduction

Un L-system permet de modeliser la croissance d'une plante à partir de sa forme de base (la graine). Sa forme de base contient déjà la forme finale de la plante.

Un L-system est composé d'un axiome de base, de règles de développement et d'un angle. L'axiome, tout comme les règles sont modéliser par des symboles ; chacun d'eux pouvant avoir des règles. Nous allons donc traduire ce problème en langage orienté objet, et nous expliquerons notre choix sur les différentes classes permettant la modélisation du problème.

# 2 Visualisation du problème

Un l-système est composé d'un *axiome*, de *règles* de croissances et d'un angle.

On pourra donc créer une classe *l-systeme* qui utilisera un objet de type *regle* et un objet de type *axiome*.

Dans un *axiome*, ou dans une *règle* nous avons des *symboles*, qui sont interprétables graphiquement ou non.

Chaque *symbole* à sa *règle* de croissance. Comme dans une *règle* il y a des *symboles* et inversement on ne peut pas définir une classe avant l'autre.

On a donc créer une classe *symbole* qui utilise la classe *regle* incomplète. On utilisera donc un pointeur sur une *règle* et non la *règle* elle meme.

Quand on lira un *l-système* on créera les *symboles* avec un constructeur qui ne prend pas de *règle* en paramètre. Les *règles* seront affectées aux *symboles* quand on les lira.

Nous créons aussi des objets de type *interpreteGraphique* pour interpréter chaque *symbole*. Le constructeur de cette classe prend juste en paramètre le caractère du symbole. Un *symbole* possède un objet de type *interpreteGraphique*.

Pour lire un fichier texte et créer un *l-système* nous avons fait une classe *lecteur*.

Pour la partie graphique nous avons créer une classe *zoneDessin* qui permet de dessiner des lignes dans une Gtk : :DrawingArea. Nous avons créer une classe *interface* qui permet de créer une fenetre dans laquelle se retrouveront des composants graphiques (bouton, label, zone de saisie) qui permettront de gérer un *l-systeme*

(création, représentation...) par l'intermédiaire des méthodes d'un objet *l-systeme*.

### 3 La classe symbole

Cette classe permet de créer des *symboles*. Un *symbole* est défini par son caractère, une *règle* et un interprète graphique. Comme une *règle* est composé de *symboles* on déclare la classe *règle* incomplète. De ce fait on utilise un pointeur sur une *regle*. On a créé des méthodes pour le renvoi des attributs (getRegle...). La méthode appliqueAffecteRegle(regle \*r) permet d'affecter une *règle* à un *symbole*.

### 4 La classe regle

On définit une *règle* par le *symbole* qui est affecté et par la *règle* elle même, qui est une collection de *symboles*. Cette collection est représentée par un vecteur de *symboles*. Cette classe possède des méthodes pour renvoyer les attributs (symbole getSymbole(), vector<symbole> getVector()).

### 5 La classe axiome

Cette classe utilise des *symboles*(et donc des *règles*). On définit un *axiome* par une collection des *symboles* (un vecteur de *symboles*). Elle possède des méthodes de renvoi des attributs.

### 6 La classe l\_systeme

Un *l\_système* est composé d'un angle, d'un *axiome* et d'un ensemble de *règles* de croissance. L'ensemble des *règles* est représentées sous forme d'un vecteur de *règles*. Un *l\_système* peut se développer à un certain niveau. On a donc créer une méthode void developpeNiveau(int n) qui permet de faire évoluer le *l\_système* selon ses *règles* de croissances. Pour se faire, pour chaque *règle* on parcourt l'*axiome* et si un

des caractères correspond à celui de la *règle* alors place dans un nouveau vecteur de *symbole* la *règle* (on parcourt sa collection de *symboles* et on insère chaque symbole) sinon on place le *symbole*. On répète l'opération n fois.

## 7 La classe lecteur

Elle permet de lire un fichier contenant les données d'un *l-système*. On utilise la librairie `fstream`. Le fichier texte est formaté de la manière suivante :

angle ;axiome ;symboleregled devient Uneregle.

L'*axiome*, l'angle et les *règles* sont séparé par des ;.

Pour l'*axiome* on lit la chaîne de caractères, on la stocke puis on la parcourt et pour chaque caractère on crée un *symbole* mais sans lui appliquer de règle. On stocke ces *symboles* dans un vecteur de *règles*.

Pour les *règles* on lit son *symbole* puis on crée une liste de *symboles* avec la chaîne de caractères de la *règle*. Une fois cette *règle* lue on appelle le constructeur de la classe *règle* pour la créer et on stocke cette *règle* dans un vecteur de *règles*. On procède ainsi jusqu'à la fin du fichier.

Ensuite on parcourt l'*axiome* et on regarde si on peut appliquer une *règle* (de celles créées) à chacun de ses *symboles*. On crée alors le *l-système* avec la collection de *symboles* (avec ses règles) de l'*axiome*, l'angle et l'ensemble des *règles*.

## 8 La classe zoneDessin

Nous utilisons la librairie `Gtkmm` pour dessiner. On crée une `DrawingArea` dans laquelle on pourra dessiner des lignes (`draw_line`).

## 9 La classe interpretGraphique

Elle permet de créer une interprétation graphique d'un *symbole*. Son constructeur prend en paramètre le caractère du *symbole*. Ensuite dans le constructeur on regarde si le caractère est un caractère LOGO. Si c'est le cas on met l'attribut `représentable` à `true`. On a créé des méthodes permettant de calculer les coordonnées futures de la tortue LOGO en fonction du *symbole*, des positions

actuelles, du cap actuel et de l'angle. Si le symbole est F on dessine une ligne avec les nouvelles coordonnées.

## 10 La classe interface

Elle permet de créer l'interface graphique du programme. On utilise la librairie Gtkmm pour définir une fenêtre, des boutons, des zones de saisies. Chaque composant est mis dans une grille (Gtk Table) et cette dernière est placée dans la fenêtre(Gtk Window). Chaque bouton fait appel à un callback qui appelle une méthode de la classe interface ou d'une autre classe. Dans cette classe on utilise tous les objets et leurs méthodes permettant de gérer un *l-système*. Cette fenêtre permet d'écrire un *l-systeme* dans un fichier texte en choisissant le nom et l'emplacement par l'intermédiaire d'une boîte de saisie. On peut charger un *l-système* à partir d'un fichier texte existant, le développer et le dessiner à l'écran.

## 11 Problèmes rencontrés

Nous n'avons pas su modéliser la tortue pour qu'elle puisse suivre le développement du *l-système* car la tortue précédente ne s'efface pas. Nous n'avons pas réussi à l'effacer au fur et à mesure du développement.

Bien que la plupart des outils permettant d'exporter des images nous soient fournis dans la classe Ecran, nous n'avons pas traité ce point par manque de temps.

L'image ne se dessine pas totalement à l'écran, il faut iconifier la fenêtre puis la réouvrir pour avoir la totalité du dessin (problème de redraw?).

## 12 Copie d'écran

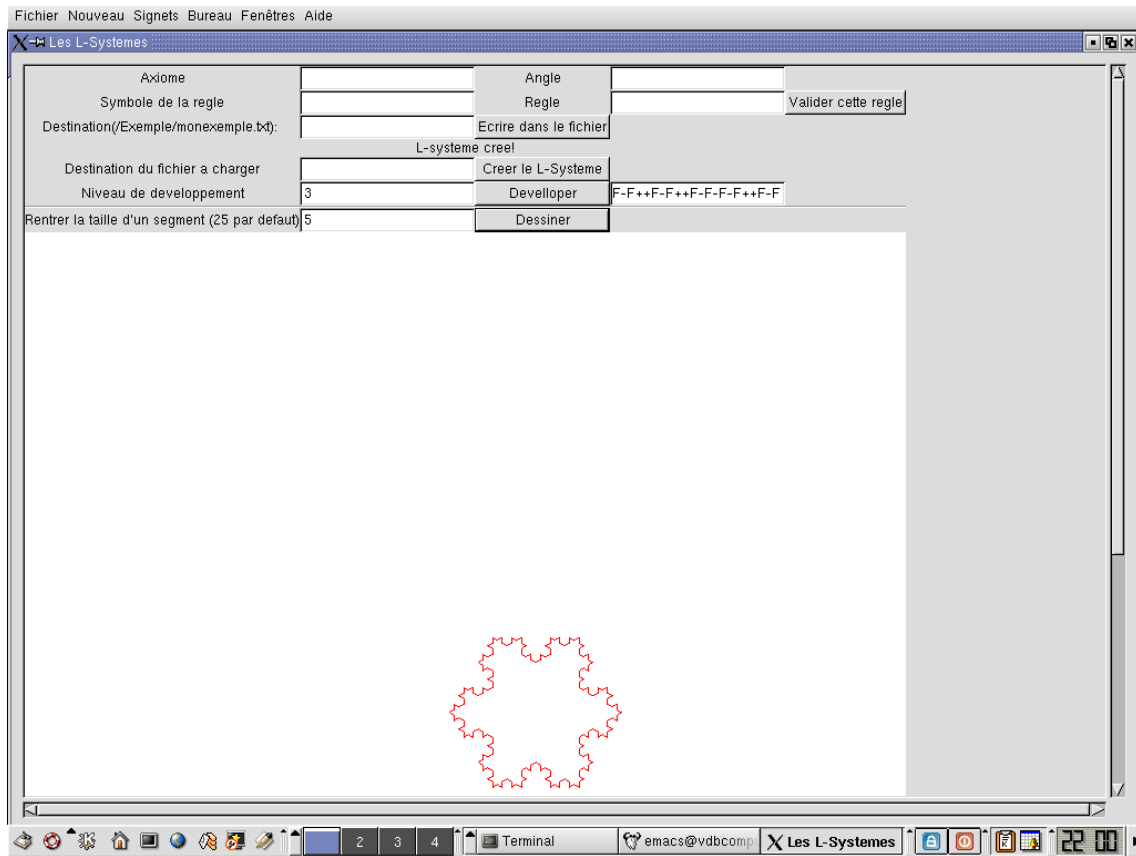


FIG. 1 – Flocon de Koch

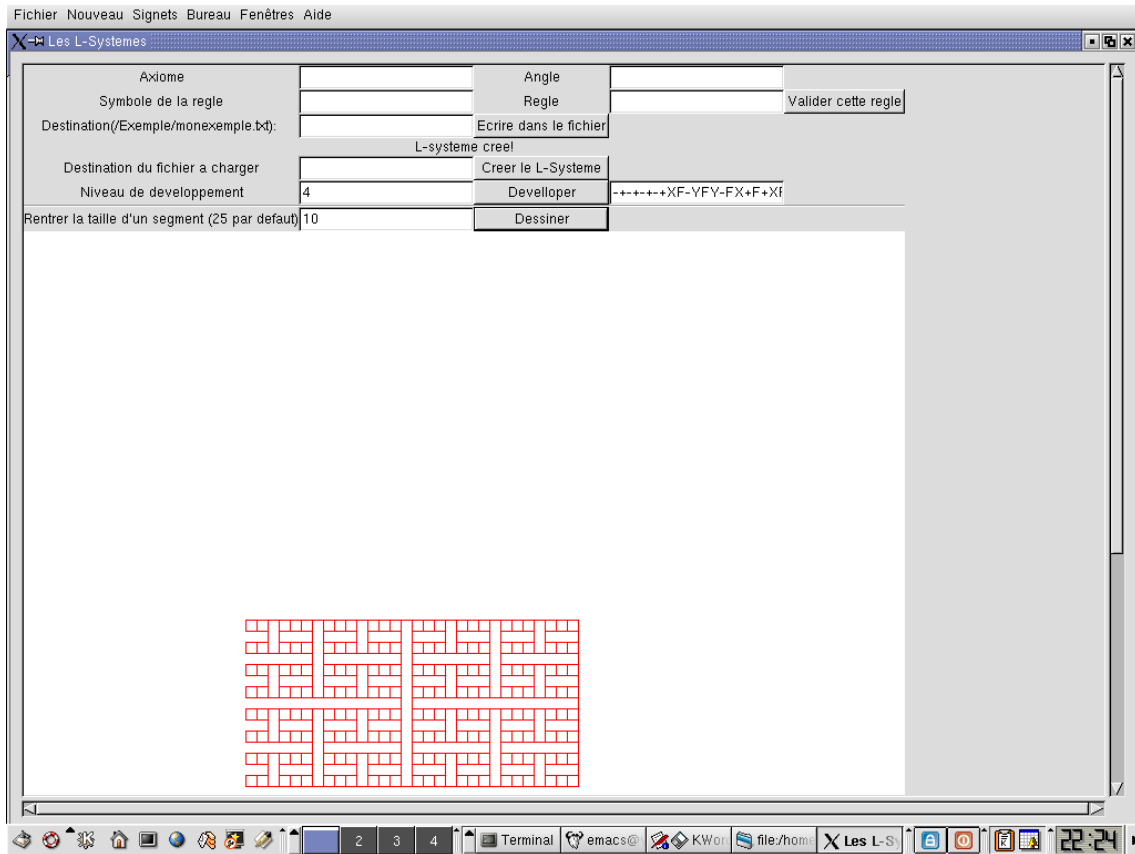


FIG. 2 – Le l-system à deux règles

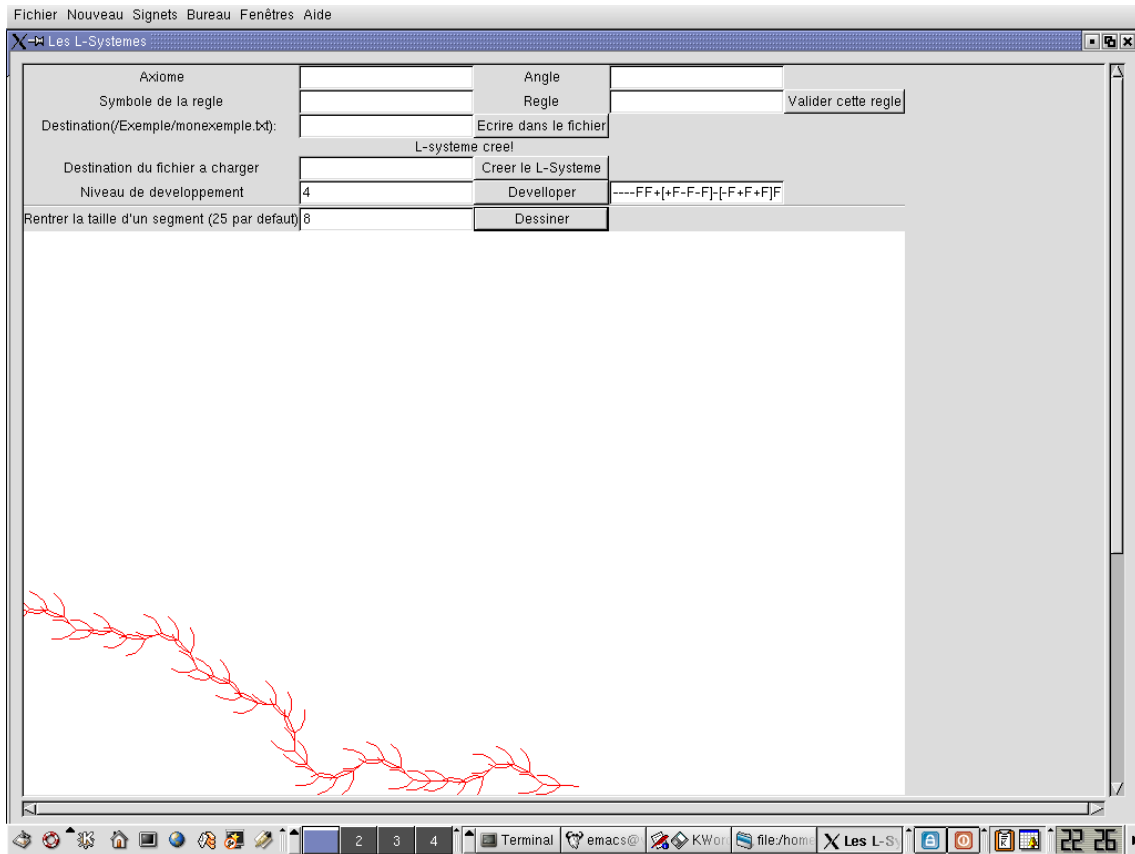


FIG. 3 – La plante

# 13 Diagramme UML

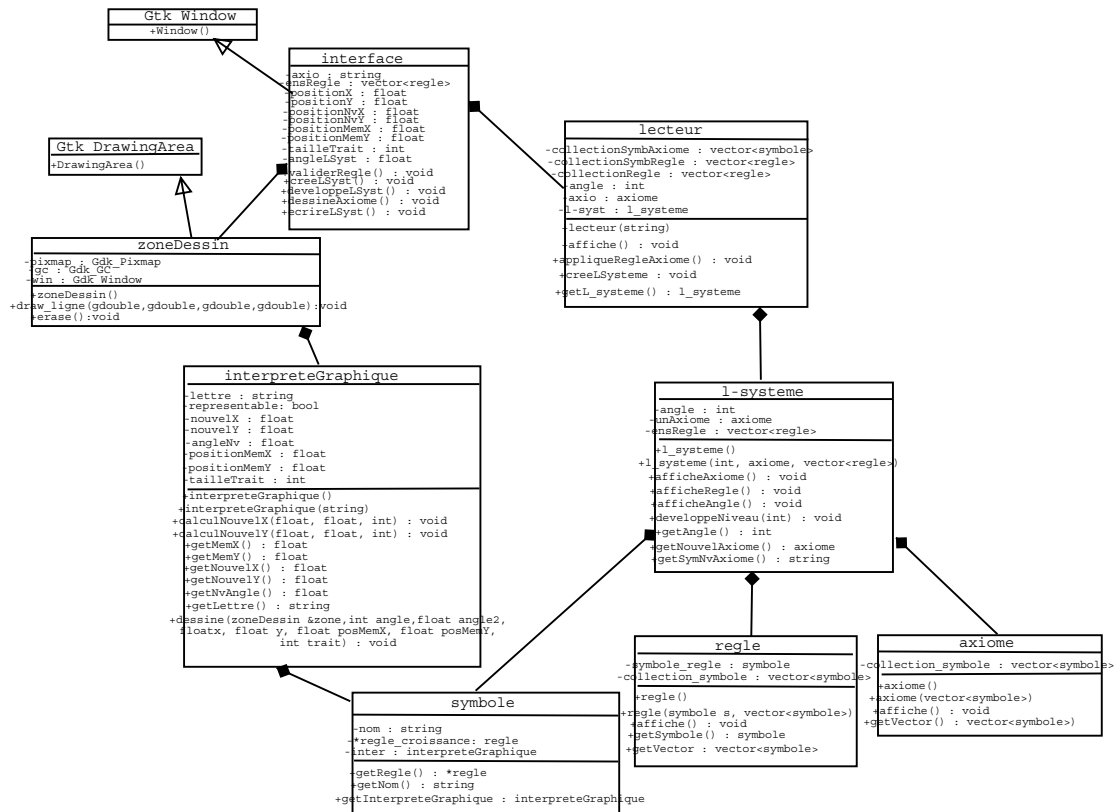


FIG. 4 – Diagramme des classes