UNIVERSITÉ DE NAMUR

Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

THESIS / THÈSE

MASTER EN SCIENCES INFORMATIQUES

Éditeur Graphique Interactif de Graphes

Vause, Michel

Award date:
1991

Awarding institution:
Universite de Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 02. May. 2026

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

Editeur Graphique Interactif de Graphes

par Michel VAUSE

Promoteurs :
Jean-Paul LECLERCQ et Philippe TOINT

Mémoire présenté en vue
de l'obtention du titre
de Licencié et Maître
en Informatique

Année académique 1990-1991

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
Rue Grandgagnage, 21B
B-5000 NAMUR

Editeur Graphique Interactif de Graphes

Michel VAUSE

Résumé :

Un modèle de circulation routière se base sur une mise en relation de diverses données concernant les axes de transport (trafic, caractéristiques physiques, environnement, etc). Pour permettre une vue synthétique, analytique et comparative des problèmes de transport sur ces axes, il est essentiel de disposer d'un éditeur graphique. Le présent travail discute les spécifications de cette interface en fonction des besoins des utilisateurs, le choix d'un système de fenêtrage (X Window), le choix d'outils graphiques (Xlib et Xt), le choix d'un langage de programmation (C++), les algorithmes graphiques utilisés (Cohen-Sutherland, Bresenham), ainsi que les problèmes concrets rencontrés lors de l'implémentation.

Abstract :

Road traffic systems are modelled using conceptual objects of the road network environment (e.g. traffic, physical attributes) and the dependencies between these. To have a synthetic, analytic or comparative view of these objects, a graphical editor or browser tool is essential. The present work discusses the requirements of the proposed tool from the perspective of the users, the choice of a windowing system (X Window), the choice of graphic packages (Xlib and Xt), the preference for an object-oriented language (C++), the graphic algorithms used (Cohen-Sutherland, Bresenham), and the concrete problems encountered during the implementation.

Mémoire présenté en vue de l'obtention du titre
de Licencié et Maître en Informatique
Septembre 1991

Promoteurs : Jean-Paul Leclercq et Philippe Toint.

Merci ...

à mon épouse et mes enfants, qui supportent depuis des années un mari et un père éternel étudiant,

à Messieurs Jean-Paul Leclercq et Philippe Toint, qui m'ont suivi tout au long de ce travail,

à Messieurs Michel Bierlaire et Didier Burton, qui ont partagé leur expérience et m'ont prodigué de judicieux conseils,

aux membres de la firme Stratec et de la Région Wallonne, qui m'ont fait part de leurs suggestions.

Sans eux, ce travail n'aurait pu être mené à bien !

"It is almost a maxim that there is no such thing as portable software, only software that has been ported"

Adrian Nye and Tim O'Reilly

Table des matières

1 Présentation générale 5
1.1 Cadre du travail : la convention AGIR 5
1.2 Objectifs du travail 5
1.3 Contraintes 6
1.4 Les étapes du travail 7

2 Les spécifications du projet 8
2.1 Etude de l'existant 8
2.1.1 Programme sur Apollo 8
2.1.2 Programme XDrive Imauro 8
2.1.3 Programme Trips 9
2.2 Les fonctionnalités souhaitées 9

3 Choix des outils 12
3.1 Choix d'une interface utilisateur 12
3.2 Choix des outils graphiques 12
3.2.1 PHIGS 13
3.2.2 GKS 14
3.2.3 Xlib 15
3.2.4 Conclusion 15
3.3 Choix d'un langage de programmation 17
3.4 Schéma récapitulatif 18

4 L'architecture X de l'éditeur 19
4.1 X Window System 19
4.1.1 Client, Serveur et Protocole 19
4.1.2 Concepts de base du fenêtrage 20
4.1.3 La philosophie de l'événement 22

4.1.4 Un système distribué . . . 23
4.1.5 Le gestionnaire de fenêtres . . . 24
4.1.6 Les couches de développement X . . . 26
4.1.7 La hiérarchie des widgets . . . 28
4.1.8 Gestionnaires d'événements et *callbacks* . . . 30
4.1.9 Modèle de programmation sous X Window . . . 32
4.2 Description générale de l'interface . . . 32
4.3 Description des fonctionnalités . . . 35
4.3.1 Sous-menu Graph . . . 35
4.3.2 Sous-menu Nodes . . . 38
4.3.3 Sous-menu Links . . . 40
4.3.4 Sous-menu Set of nodes . . . 42
4.3.5 Sous-menu Set of links . . . 44
4.3.6 Sous-menu Background . . . 46
4.3.7 Sous-menu Model . . . 48
4.4 Le système de menus . . . 48
4.5 Les widgets de l'interface . . . 50
4.5.1 *topLevelWidget* . . . 50
4.5.2 *frameworkWidget* . . . 50
4.5.3 *menuBarWidget* . . . 51
4.5.4 *coordWidget* . . . 51
4.5.5 *messageWidget* . . . 52
4.5.6 *okWidget* . . . 52
4.5.7 *vertScrollWidget* et *horScrollWidget* . . . 52
4.5.8 *canvasWidget* . . . 53
4.5.9 *fillerWidget* . . . 53
4.5.10 Les popup widgets . . . 53

5 Algorithmes graphiques **55**
5.1 Transformation de fenêtrage . . . 55
5.2 Découpage : algorithme de Cohen-Sutherland . . . 59
5.3 Détection des objets cliqués . . . 62
5.3.1 Position du problème . . . 62
5.3.2 Solution adoptée . . . 63
5.3.3 Problème de la validation . . . 63
5.3.4 Enregistrement des arcs : algorithme de Bresenham . . . 64

5.3.5 Enregistrement des noeuds 68

6 C++ 70

6.1 Quelques rappels concernant C++ 70
6.2 Compilateur g++ 71
6.3 Compatibilité entre C++ et X Window 72
6.4 Compatibilité entre C++ et C 73
6.5 Les classes de base 74
6.5.1 Les tableaux dynamiques 74
6.5.2 Les listes 76
6.5.3 Les ensembles 76
6.6 Les classes *graphiques* 76
6.6.1 La classe *World* 76
6.6.2 La classe *Disp* 77
6.6.3 La classe *View* 77
6.6.4 La classe *WorldPoint* 79
6.6.5 La classe *DispPoint* 80
6.6.6 La classe *WorldLine* 80
6.6.7 La classe *DispLine* 81
6.6.8 La classe *WorldPolyLine* 82
6.6.9 La classe *DispPolyLine* 82
6.6.10 Les classes *Disk, Cross, X, Square, Dot* et *Asterisk* 83
6.7 Les classes relatives à la structure de graphe 83
6.7.1 La classe *GNode* 83
6.7.2 La classe *GLink* 85
6.7.3 La classe *GNodesArray* 86
6.7.4 La classe *GLinksArray* 87
6.7.5 La classe *GGraph* 87
6.8 Les classes relatives à la détection des objets 88
6.8.1 La classe *objectItem* 88
6.8.2 La classe *objectList* 89
6.8.3 La classe *image* 89

7 Ce qui reste à faire 91

7.1 Améliorer l'esthétique interne et l'efficacité du programme 91
7.2 Problèmes spécifiques au g++ 92

7.3 Tout porter en C++ 92
7.4 Fonctionnalités non encore implémentées 93
7.5 Nouvelles fonctionnalités souhaitées 94
7.6 Améliorer la convivialité et l'ergonomie 94
7.7 Interfaçage avec un modèle 94
7.8 Porter l'interface sur PC 95

8 Conclusions **96**

Liste des figures **98**

Bibliographie **99**

Chapitre 1

Présentation générale

1.1 Cadre du travail : la convention AGIR

Le travail présenté dans les pages qui suivent a été réalisé dans le cadre de la convention AGIR (Aide à la Gestion des Infrastructures Routières). Cette convention en matière de gestion du trafic routier et des infrastructures a été établie entre la Région Wallonne, le Groupe de Recherche sur les Transports des Facultés Universitaires Notre-Dame de la Paix (GRT), le Département d'Economie Appliquée de l'Université Libre de Bruxelles (DULBEA) et la société STRATEC.

En ce qui concerne l'aspect *Recherche-Développement*, une tâche essentielle du GRT est la mise au point d'outils d'aide à la décision en matière de transports. En particulier, un projet de modèle régional de circulation doit être développé, qui se basera sur une mise en relation automatique des diverses données concernant les axes de transport dans la Région. Il intégrera des données sur le trafic, les caractéristiques physiques et l'environnement des axes en question. Il permettra à la fois une vue synthétique, analytique et comparative des problèmes de transport sur ces axes. Il pourra servir de tableau de bord aux décideurs régionaux en matière de transport.

1.2 Objectifs du travail

La notion même de réseau routier fait apparaître de manière naturelle une structure de graphe : les nœuds représentent, par exemple, les carrefours, les quartiers, les villes, ... et les arcs correspondent aux voies de transport. Cette structure présente une particularité essentielle : sa géométrie. Effectivement, les nœuds possèdent une

localisation bien précise, les arcs une forme bien définie. Dans ce contexte, il est crucial pour l'utilisateur de pouvoir visualiser cette géométrie du réseau.

Mais ce graphe n'a vraiment d'intérêt que dans la mesure où il est couplé à une base de données : sociologie d'un quartier, attractivité d'un centre commercial, nombre de bandes de circulation d'une route, schéma des feux d'un carrefour, priorité de droite ou non, nombre d'accidents, etc. Il est particulièrement intéressant pour l'utilisateur d'avoir accès à cette base de données (que ce soit pour la modifier, la compléter ou simplement pour la consulter) au travers d'une interface graphique et conviviale.

Dans sa version la plus achevée, un modèle de trafic routier peut permettre d'effectuer certains calculs (génération de flots entre quartiers, calculs des plus courts chemins, etc), y compris des calculs statistiques (corrélation entre nombre d'accidents et limitation de vitesse, etc) et certaines simulations (que se passe-t-il si l'on met telle rue à sens unique ? et si l'on construit un pont à tel endroit ? etc). L'interface graphique développée doit bien sûr pouvoir donner à l'utilisateur la possibilité d'utiliser toutes les fonctionnalités du modèle sous-jacent.

Il apparaît donc que, dans le cadre de n'importe quelle modélisation du trafic, il est essentiel de disposer d'un éditeur graphique de ce modèle.

La volonté de développer un éditeur graphique répond à bien plus qu'un simple souci de convivialité : il doit réellement apporter quelque chose à l'utilisateur. Il s'agit d'un outil essentiel pour permettre une vue à la fois synthétique, analytique et comparative des problèmes de transport sur le réseau. La possibilité de pouvoir visualiser, par exemple, la congestion d'un réseau routier est bien plus parlante que la simple consultation d'un tableau de nombres.

C'est cet éditeur graphique que nous avons été chargés de développer.

1.3 Contraintes

Deux contraintes majeures devaient guider notre travail.

D'une part, l'interface développée doit être indépendante de tout modèle. Elle ne devait pas être réalisée pour pouvoir être utilisée avec tel modèle déjà développé ou à développer : elle devait être construite de la manière la plus souple possible, pour pouvoir s'adapter à l'utilisation de n'importe quel modèle. En particulier, elle peut très bien ne servir que d'interface à l'utilisation d'une base de données. Dans ce contexte, la modularité du produit développé devait absolument être privilégiée.

D'autre part, cette interface doit pouvoir être utilisée aussi bien sur n'importe quelle station de travail que sur PC.

1.4 Les étapes du travail

Notre travail s'est essentiellement déroulé en trois étapes :

1. Etablir les spécifications du programme demandé.
   Ces spécifications ont été obtenues d'une part à partir de l'étude de logiciels existants (programmes développés par les Facultés ou programmes commerciaux) et d'autre part à partir de discussions avec les futurs utilisateurs de l'interface : membres de la firme Stratec et de la Région Wallonne. Une première version des spécifications ainsi obtenues ont été présentées lors d'un séminaire donné le 27 mai 1991, dans le cadre de la convention Agir. Les discussions avec les participants de ce séminaire ont permis de raffiner ces spécifications.

2. Etudier les outils disponibles.
   Dans un second temps, il nous a fallu réaliser un tour d'horizon des différents outils disponibles : interface de fenêtrage (X Window, Windows 3), outils graphiques (PHIGS, GKS, Xlib) et langages de programmation (C, C++). Après en avoir étudié les avantages et les inconvénients, il nous a fallu choisir les outils que nous utiliserions... et apprendre à les utiliser !

3. Implémenter le projet.
   Enfin, dans un dernier temps, nous avons implémenté l'éditeur graphique souhaité.

Chapitre 2

Les spécifications du projet

2.1 Etude de l'existant

Passons d'abord en revue quelques programmes similaires ayant déjà été développés.

2.1.1 Programme sur Apollo

Une interface avait déjà été développée aux Facultés par Messieurs Michel Bierlaire et Didier Burton, dans le cadre d'une précédente convention avec la Région Wallonne. Malheureusement, cette interface a été développée sur matériel Apollo, et en utilisant nombre de spécificités de cette machine. Particulièrement non portable, cette interface devait être complètement réécrite pour pouvoir être utilisée sur d'autres stations, ou a fortiori sur PC.

2.1.2 Programme XDrive Imauro

Une autre interface a été développée par Monsieur Hugo Marien dans le cadre du projet Drive Imauro. Cette interface devrait pouvoir être portée sans trop de problèmes sur PC. Hélas, elle est fortement dépendante du modèle sous-jacent et son absence de modularité la rend difficilement adaptable à d'autres modèles.

2.1.3 Programme Trips

Le programme Trips[1] est en fait un programme commercial de modélisation de trafic qui dispose d'une interface graphique particulièrement performante. Ce programme tourne sur PC, dans un environnement GEM. Mais, comme il s'agit d'un programme commercial dont nous ne disposons pas des sources, il n'est pas possible d'adapter l'interface à d'autres modèles.

2.2 Les fonctionnalités souhaitées

Parmi les fonctionnalités de base dont on souhaite disposer, citons entre autres[2] :

- Visualisation
  - Affichage du nom du projet à l'étude.
  - Faire dessiner à l'écran une configuration urbaine ou régionale quelconque. Notons à ce propos que pour permettre une meilleure visualisation de la région modélisée, il est important d'y faire figurer des points de repère : bâtiments importants, lignes de chemin de fer, cours d'eau,...
  - Traiter ce dessin par des translations, zooms et recadrages.
  - Différenciation des types de nœuds (quartiers, carrefours, points de pénétration,...).
  - Afficher ou non les numéros associés aux carrefours, quartiers ou points de pénétration.
  - Afficher au choix les routes à sens uniques, celles qui ne le sont pas, aucune, ou encore toutes ensemble.
  - Différenciation des arcs en fonction des flots (par exemple par la couleur ou l'épaisseur).
  - Possibilité de transformer directement à l'écran la manière dont un arc est représenté :
    * augmenter le nombre de points intermédiaires,
    * changer une courbe en ligne brisée et réciproquement.

[1]distribué par MVA Systematica, Grande-Bretagne

[2]Pour la plupart, ces fonctionnalités constituent une base minimale pour le logiciel en cours de développement; certaines peuvent cependant être considérées comme accessoires.

- Interaction
  - Accès à une table à digitaliser pour l'introduction des données.
  - Accès direct aux données et aux résultats via la souris.
  - Accès à une imprimante ou à une table traçante pour l'impression des dessins.
  - Accès aux fichiers pour le chargement et le sauvetage des données.
- Edition
  - Initialiser la structure de la banque de données pour un graphe.
  - Ajouter un nœud (à l'écran ou à l'aide de la table à digitaliser) avec toutes ses caractéristiques (attractivité, émissivité, sociologie,...).
  - Supprimer un nœud dans un réseau, avec bien sûr tous les arcs s'y raccrochant.
  - Ajouter une liaison entre deux nœuds (à l'écran ou à l'aide de la table à digitaliser), avec tous ses attributs (longueur, vitesse de déplacement, perte de priorité,...).
  - Supprimer une liaison dans un graphe.
  - Visualisation et modification des caractéristiques d'un nœud ou d'un arc (informations du modèle et/ou représentation graphique).
- Utilisation de modèles
  - Génération des flots entre les quartiers.
  - Répartition de ces flots sur les arcs du graphe.
  - Tracer les chemins supposés empruntés par les utilisateurs (par exemple, plus courts chemins) entre deux nœuds ou entre un nœud et tous les autres, et ce, dans le sens aller et retour.
  - Obtenir les caractéristiques d'un chemin ainsi calculé.
  - Afficher, pour un arc, le flot y circulant dans le sens direct, dans le sens inverse, le flot matinal, ainsi que le flot en soirée, avec finalement la charge totale de l'arc désigné, ainsi que tout autre résultat éventuellement fourni par le modèle.

Le développement d'une telle interface est une entreprise de longue haleine, dont l'aboutissement est impossible en quelques mois. La version actuelle ne constitue donc que l'implémentation d'un sous-ensemble des fonctionnalités souhaitées.

Chapitre 3

Choix des outils

3.1 Choix d'une interface utilisateur

L'éditeur graphique que l'on veut développer doit s'intégrer dans un environnement convivial, présentant à l'utilisateur une interface graphique à fenêtrage.

Sur les stations de travail Unix, un standard de fait s'impose : X Window System.

Sur PC, on assiste actuellement à un véritable raz-de-marée Windows 3. Malheureusement, cet environnement n'existe pas sur les stations de travail. Mais d'autre part, il semble que X Window System devrait également s'imposer sur PC. C'est ainsi qu'un produit comme Desqview/X (qui était annoncé pour le 1er trimestre 1991, mais qui semble avoir pris un peu de retard) permet, après simple recompilation, de porter sur PC n'importe quelle application X. Un PC raisonnablement puissant (386 avec 4 Mo de RAM, par exemple) devrait suffire pour procurer un confort d'utilisation raisonnable. Nous avons donc décidé (en accord avec la firme STRATEC) de réaliser le développement de l'éditeur graphique dans un environnement X Window.

3.2 Choix des outils graphiques

A priori, il semblait intéressant, en ce qui concerne l'aspect graphique proprement dit, d'utiliser des produits existant déjà, tels des "langages graphiques" du type GKS ou PHIGS.

Quelles sont les spécifications que l'on est en droit d'attendre d'un tel langage graphique ?

- Possibilité de travailler en coordonnées indépendantes du périphérique (avec ce que cela implique de facilités de fenêtrage, etc).
- Ensemble de primitives graphiques de base : segments, splines, polygones, cercles, textes,...
- Notion d'objet graphique construit à partir des primitives de base et d'attributs (taille, couleur, visibilité,...).
- Décomposition d'une image en un ensemble d'objets graphiques.
- Edition interactive d'une image et de ses composantes.
- Possibilité de stockage de la représentation graphique.
- Disponibilité de pilotes de périphériques de sortie (imprimantes, tables traçantes).

Mais l'aspect le plus important est peut-être que ce langage graphique doit être disponible sur station de travail et sur micro-ordinateurs (PC).

Présentons d'abord quelques réflexions sur les outils utilisables.

3.2.1 PHIGS

PHIGS (Programmer's Hierarchical Interactive Graphics System) est un langage de description graphique, qui s'impose comme un standard pour les applications graphiques interactives manipulant des dessins en 2 et 3 dimensions. Les spécifications standards de ce langage ont été développées par le Comité Technique X3H3, afin d'être acceptées par le *American National Standards Institute* (ANSI) et l'*International Standards Organisation* (ISO). Pratiquement, chaque constructeur de station de travail propose sa propre version de PHIGS, qui respecte le standard établi et y apporte quelques possibilités supplémentaires.

Les avantages de PHIGS sont d'être particulièrement souple et puissant, et d'être indépendant du périphérique utilisé. Ceci est particulièrement intéressant en ce qui concerne l'utilisation de tables traçantes et imprimantes. Cependant, le code généré est énorme : au minimum 6 Mo pour le moindre petit programme. Et surtout, ce langage graphique n'existe pas sur PC.

Etant donné que l'application développée devra tôt ou tard être portée sur PC, était-il rentable d'investir dans l'utilisation de PHIGS, alors qu'il aurait fallu de

toutes façons envisager une autre solution au moment de passer sur PC ? Nous ne le pensons pas.

3.2.2 GKS

GKS (Graphical Kernel System) est également un langage de description graphique, défini par une norme ANSI, plus ancien que PHIGS, et, dans sa version initiale, limité aux graphiques à 2 dimensions.

En ce qui concerne GKS, si l'on dispose également de l'indépendance vis-à-vis des périphériques utilisés, il semble cependant être moins puissant que PHIGS, notamment en ce qui concerne l'édition interactive des objets graphiques. Il ne possède pas non plus de possibilité de hiérarchiser les objets graphiques manipulés, mais cela ne représente pas un handicap majeur en ce qui concerne notre application.

De nouveau, le problème crucial est l'existence d'une version sur PC. Il existe effectivement une version de GKS sur PC, commercialisée par IBM et qui date de 1984. Depuis, il semble bien que le produit ait été abandonné.

Il existe également une version de GKS tournant sous X, et faisant partie du domaine public. Disposant dès lors des sources, il serait évidemment possible de recompiler celles-ci de manière à pouvoir les utiliser dans un environnement X sur PC. Malheureusement, cette solution présente différents problèmes :

- N'étant pas un produit officiel mais un logiciel du domaine public, il n'y a aucune garantie de fiabilité du produit, ni d'assurance de maintenance ou de développement ultérieur.

- Cette solution ne résoud pas le problème des pilotes de périphériques. En effet, il n'y a pas de pilotes intégrés au programme : la seule possibilité est de "sortir" l'image graphique dans un métafichier : libre à l'utilisateur de créer son propre programme pour transférer ce métafichier sur imprimante ou table traçante. Comme le format des métafichiers GKS est standardisé, l'interprétation de tels métafichiers ne devrait pas poser de gros problèmes.

- Mais le plus grave est la philosophie adoptée par les concepteurs de XGKS en ce qui concerne ses relations avec X Window. L'idéal serait de pouvoir ouvrir une fenêtre X indépendamment de XGKS, puis de communiquer à XGKS l'identificateur de cette fenêtre pour qu'il en fasse une *station de travail* (objet de base de GKS). Malheureusement, cela n'est pas possible. Dès que l'on veut ouvrir une station de travail, XGKS crée une fenêtre X et l'affiche à l'écran.

Il n'y a pas moyen de contrôler les paramètres utilisés pour la création de cette fenêtre (comme par exemple ses dimensions). Il y a bien sûr moyen de redimensionner cette fenêtre par la suite, mais elle est d'abord affichée avec la géométrie par défaut. D'autre part, il y a le problème de faire en sorte que cette fenêtre soit une "sous-fenêtre" d'une autre fenêtre (pour pouvoir utiliser des widgets[1] *menu*, *scrollbar*, ...). Il faudrait procéder à un changement de parent pour la fenêtre créée par XGKS, ce qui ne manquerait pas de poser des problèmes, étant donné que XGKS n'utilise pas de widget, mais travaille uniquement avec Xlib[2].

En l'absence de version de GKS ne posant pas de problème d'interfaçage avec X Window, nous avons également décidé d'abandonner cette solution.

3.2.3 Xlib

Reste bien entendu la possibilité "ultime" : programmer soi-même les routines graphiques nécessaires, à l'aide de Xlib et de Xt. Dans ce cas, les avantages sont d'avoir un code plus compact et une portabilité plus grande. Malheureusement, c'est au prix d'une difficulté de programmation accrue : on manipule des images "bitmap", la difficulté la plus importante se situant au niveau de la sélection de l'objet graphique (nœud ou arc) affiché à l'écran. Ensuite (et ce n'est pas le moins grave), il faut écrire ses propres pilotes de périphériques, puisque cette solution n'est pas indépendante des périphériques.

D'autre part, il faut se montrer moins exigeant en ce qui concerne la représentation graphique des arcs : se contenter de segments de droite, c'est-à-dire renoncer à la possibilité de représenter les arcs par des courbes. Mais à notre avis, il n'est pas indispensable d'utiliser des courbes : on peut concevoir de travailler par "segments", avec éventuellement de "nombreux" points intermédiaires.

3.2.4 Conclusion

Il ressort des discussions que nous avons eues avec la firme STRATEC que la disponibilité du produit sur PC est un critère primordial. Des produits semblables,

[1]Le concept de *widget* sera explicité en détail au chapitre suivant.

[2]Xlib et Xt sont des composantes logicielles de base du système X Window. Nous y reviendrons au chapitre 4.

tournant sur PC, existent déjà sur le marché : SATURN[3] (dont l'interface graphique est actuellement très rudimentaire, mais dont une version améliorée est annoncée) et surtout TRIPS, dont l'interface graphique est particulièrement performante. Dans l'optique d'une commercialisation du produit final, il faut faire au moins aussi bien, mais sans imposer aux clients potentiels un équipement démesuré. Nous avons donc décidé finalement de travailler avec Xlib et Xt.

Revoyons rapidement les problèmes posés par l'utilisation de Xlib plutôt qu'un langage de haut niveau.

Dessin

Le dessin de "marqueurs" de différents types (cercles, disques, carrés, croix, astérisques,...) pour les nœuds ne pose pas de problème.

Le dessin des arcs ne pose pas plus de problème, sinon que l'on abandonne l'approche "spline" au profit d'une approximation par segments brisés.

Fenêtrage et Découpage

Il faudra appliquer un algorithme de découpage tel que le classique algorithme de Cohen-Sutherland.

Détection des objets "cliqués" à l'écran

C'est un des deux problèmes majeurs.

Une solution est de mémoriser dans une matrice l'association entre point écran et identificateur de l'objet affiché. Pour ce faire, il faut simuler le tracé du segment ou du cercle (à l'aide, par exemple, de l'algorithme de Bresenham ou de la méthode de l'"analyseur digital différentiel") afin de connaître les points écran affectés par le tracé du segment ou du cercle.

Pour limiter la place mémoire nécessaire et surtout accélérer le processus, on peut songer à représenter dans un élément de cette matrice un ensemble de points écran (par exemple un carré de 5 x 5 pixels). Se pose alors le problème du risque d'imprécision lors de la sélection interactive d'un objet à l'écran. Une solution (adoptée dans le programme XDrive, mais à nos yeux trop lourde) est de demander systématiquement confirmation à l'utilisateur. Une autre possibilité est d'indiquer interactivement, lors du mouvement de la souris, quel est l'objet actuellement

[3]développé par Dirck Van Vliet (Institute for transport Studies, University of Leeds) et Mike Hall (Atkins Planning, Epsom)

sélectionné. Le fait de relâcher le bouton de la souris constitue alors la confirmation de l'utilisateur.

Pilotes de périphériques

C'est l'autre problème majeur.

Il est envisagé de produire des fichiers aux formats Postscript (imprimante) et HPGL (tables traçantes).

Etant donné que les seuls éléments à imprimer sont du texte, des segments de droite et des arcs de cercle, cela ne devrait pas constituer un problème insurmontable.

3.3 Choix d'un langage de programmation

Notre choix s'est porté sur C++, pour différentes raisons.

D'abord, pour des raisons personnelles. Il nous semblait intéressant, dans le cadre d'un travail de fin d'études, de profiter de ce mémoire pour nous initier à la programmation orientée objet en général, et au C++ en particulier.

D'autre part, les articles que nous avions lu concernant ce langage[4] nous laissaient supposer que son utilisation était particulièrement bien adaptée aux problèmes graphiques auxquels nous étions confrontés.

En ce qui concerne X Window lui-même, la programmation orientée objet apparaît également comme naturelle. Même si l'interface de programmation proposée n'existe pas au départ en C++, la conception même de X Window utilise largement les concepts de l'orientation objet. Les différents types de *widgets* utilisables sous Motif peuvent être facilement considérés comme des classes d'objets, et la structure hiérarchique de ces *widgets* est amplement basée sur la notion d'héritage, un des trois concepts de base (avec l'encapsulation et le polymorphisme[5]) de l'orientation objet. C'est d'ailleurs tellement vrai que certains informaticiens, comme Jean-Daniel Fekete, ont développé des librairies (WWL[6] par exemple) qui permettent de considérer les *widgets* comme des objets, et de programmer en C++. Nous avions l'intention d'utiliser WWL.

[4]La littérature sur ce sujet est plus qu'abondante : la programmation orientée objet a le vent en poupe et apparaît comme un des apports majeurs à l'informatique de ces dernières années.

[5]Ces trois notions seront exliquées au chapitre 6

[6]WWL, a Widget Wrapper Library for C++, librairie développée par Jean-Daniel Fekete, Laboratoire de Recherche en Informatique, Faculté d'Orsay, France.

Malheureusement, dans la pratique, les choses se sont révélées bien moins idylliques que prévu, et finalement, faute de temps pour résoudre les différents problèmes qui se sont posés à nous, il nous a fallu adopter une position de compromis, et ne programmer qu'une partie de l'éditeur en C++, l'autre étant écrite en C traditionnel. Nous reviendrons sur les différents problèmes rencontrés au chapitre 6.

3.4 Schéma récapitulatif

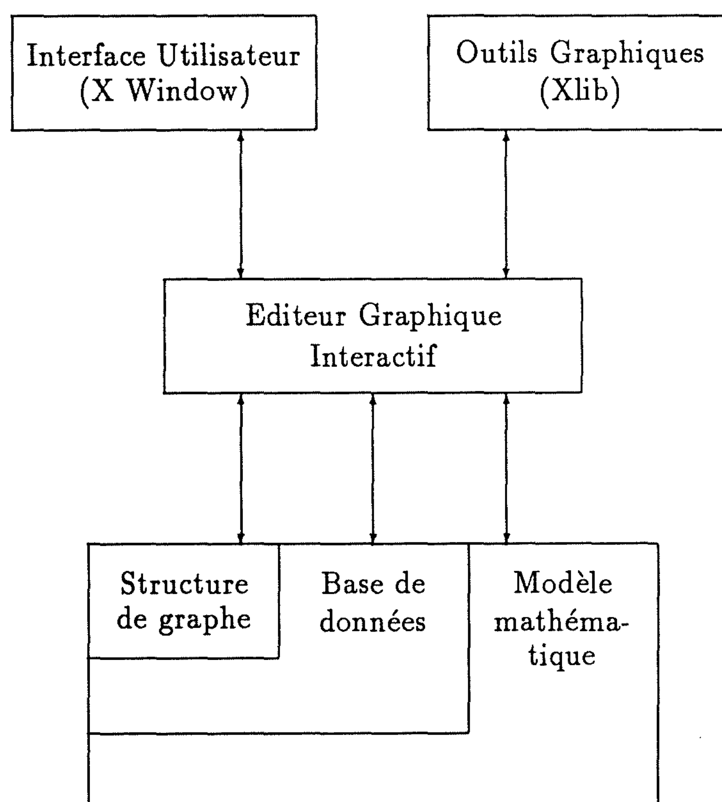

Figure 3.1: Relations entre l'éditeur, les outils utilisés et le modèle sous-jacent

Chapitre 4

L'architecture X de l'éditeur

4.1 X Window System

4.1.1 Client, Serveur et Protocole

Dans les systèmes traditionnels, si une application veut produire un affichage graphique sur l'écran d'un ordinateur, elle appelle une sous-routine d'une librairie graphique. Cette sous-routine effectue la tâche demandée (par exemple, tracer un segment de droite). La tâche une fois exécutée, le contrôle des opérations retourne à l'application.

Avec X Window, la librairie graphique est remplacée par une application, appelée le *serveur X*. C'est cette application qui a le contrôle complet de l'écran d'affichage. Une application qui désire produire un affichage graphique demande au serveur X d'effectuer une tâche spécifique en lui envoyant un "message" qui décrit la tâche désirée. Envoyer un message au serveur X rend le contrôle immédiatement à l'application, et peut ou non provoquer une réponse de la part du serveur. Les différents types de messages sont collectivement appelés le *protocole X*. Un message peut demander de tracer une ligne, un autre un cercle, un troisième d'afficher du texte, etc.

Une application qui réalise un affichage graphique en envoyant des messages suivant le protocole X est appelé un *client X*.

En retour, le serveur X peut renvoyer des messages spéciaux à un client X, tels que des messages d'événements (voir plus loin) ou des messages d'erreurs. Ces messages spéciaux font également partie du protocole X.

Typiquement, des clients X créent des fenêtres pour leur affichage. Générale-

ment, un client X peut créer et utiliser plusieurs fenêtres simultanément.

Notons qu'un serveur X peut gérer les sorties graphiques pour plusieurs clients X simultanément.

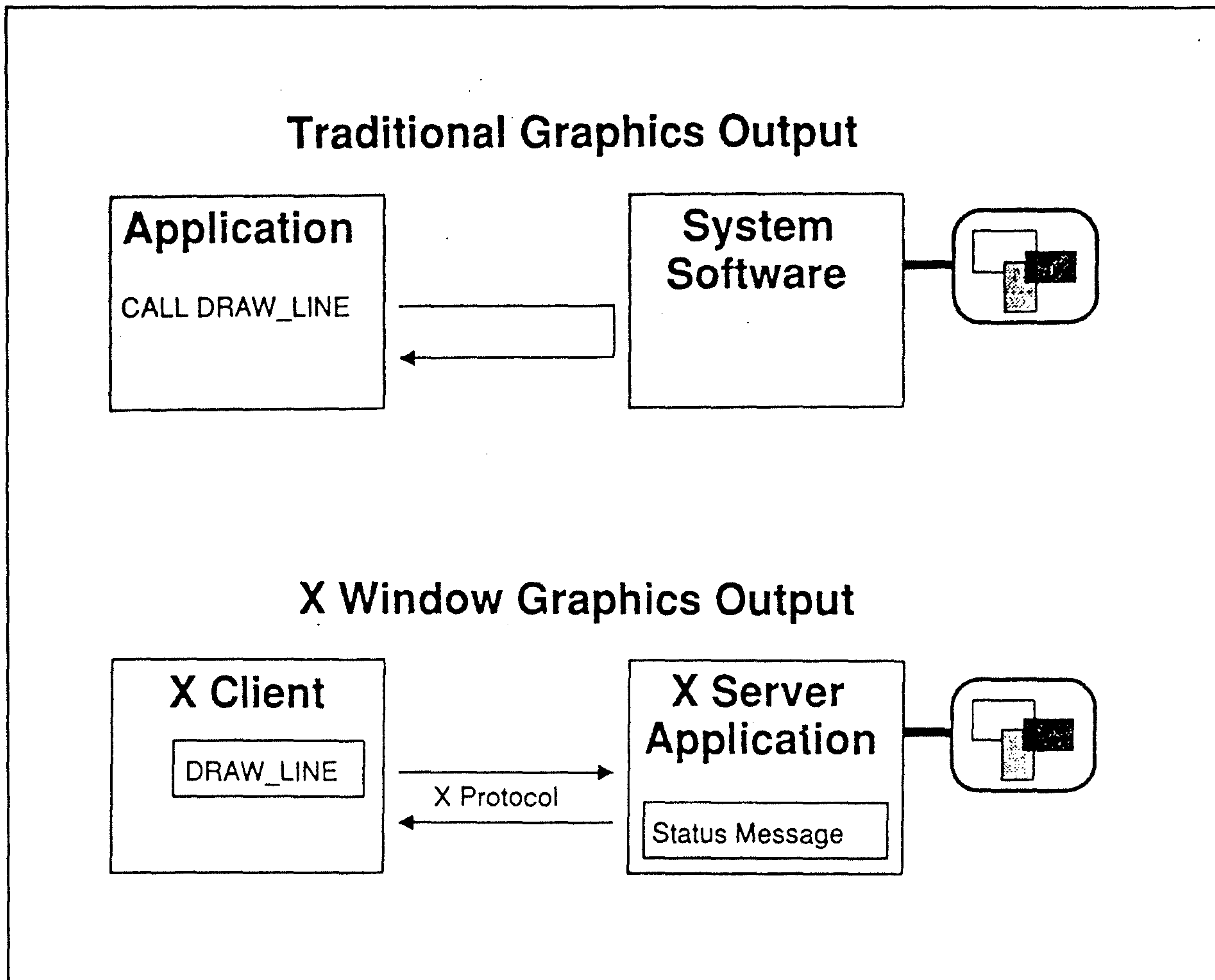

Figure 4.1: Différences entre une application traditionnelle et une application X

4.1.2 Concepts de base du fenêtrage

La notion de fenêtre

Dans X, l'objet fondamental est la *fenêtre*. En fait, une fenêtre représente simplement une section rectangulaire de l'écran. Au contraire des fenêtres d'autres systèmes de fenêtrage, une fenêtre X ne possède ni barre titre, ni barre de déroulement,

ni aucune autre décoration. Une fenêtre X apparaît simplement comme un rectangle d'une certaine couleur de fond, ou décoré d'un certain motif. Chaque fenêtre possède également un bord. Les applications peuvent combiner deux fenêtres ou plus pour créer des barres titre, des barres de déroulement ou d'autres composants de haut niveau d'une interface utilisateur.

Le serveur X crée une fenêtre en réponse à une requête d'un client. Le serveur mémorise et gère la structure de données représentant la fenêtre : le client fait référence à la fenêtre en utilisant l'identificateur de cette fenêtre. Les clients peuvent envoyer au serveur des requêtes pour modifier la taille, la position, la couleur ou toute autre caractéristique d'une fenêtre. Le client peut également demander au serveur de placer un texte ou de réaliser des opérations graphiques dans une fenêtre.

La hiérarchie des fenêtres

X organise les fenêtres suivant une hiérarchie, appelée *arbre des fenêtres*.

La fenêtre se trouvant au sommet de cet arbre est appelée la *fenêtre racine*. Le serveur X crée automatiquement une fenêtre racine pour chaque écran qu'il contrôle. La fenêtre racine occupe la totalité de l'écran physique, et ne peut être ni déplacée, ni redimensionnée.

Chaque fenêtre (à l'exception de la fenêtre racine) possède une fenêtre *parent*, et peut également avoir des fenêtres *enfants* ("sous-fenêtres").

X n'impose que peu de restrictions sur la taille ou la position d'une fenêtre. Cependant, seule la partie de la fenêtre se trouvant à l'intérieur des limites de son parent sera visible. Le serveur "coupe" ce qui dépasse les frontières de la fenêtre parent.

X permet aux fenêtres de se superposer. L'*ordre d'empilement* détermine quelles fenêtres ou parties de fenêtres seront visibles à l'écran. L'ordre d'empilement des fenêtres ne peut être modifié qu'entre fenêtres sœurs.

Le système de coordonnées

Chaque fenêtre X, y compris la fenêtre racine, possède son propre système de coordonnées. Les coordonnées du coin supérieur gauche de chaque fenêtre sont $(0, 0)$. La coordonnée x croît de la gauche vers la droite, tandis que la coordonnée y croît du haut vers le bas. La position d'une fenêtre (en fait, les coordonnées du coin supérieur gauche de la fenêtre) est toujours spécifiée par rapport au système de coordonnées de son parent.

Mapping et visibilité des fenêtres

Bien que chaque fenêtre X soit associée à une région rectangulaire de l'écran, toutes les fenêtres ne sont pas nécessairement visibles à l'utilisateur. Lorsque le serveur crée une fenêtre, il alloue et initialise une structure de données qui représente la fenêtre à l'intérieur du serveur, mais il n'appelle pas les routines (dépendant du hardware) qui affichent la fenêtre à l'écran. Les clients doivent demander explicitement au serveur d'afficher la fenêtre en lui envoyant une requête de *mapping*.

Maintenance du contenu des fenêtres

Dans un système de fenêtrage où les fenêtres peuvent se recouvrir, le contenu de chaque fenêtre doit être préservé lorsqu'une fenêtre est recouverte par une autre, de telle sorte que ce contenu puisse être restauré par la suite. Beaucoup de systèmes maintiennent et restaurent le contenu d'une fenêtre, de telle sorte que les applications ne sont pas concernées par le processus.

Par contre, en X, la responsabilité de la maintenance du contenu d'une fenêtre est du ressort du client qui utilise la fenêtre.

4.1.3 La philosophie de l'événement

Typiquement, les clients X sont suspendus jusqu'à ce qu'une action se produise, au niveau du serveur X, qui concerne le client. Le client X est alors redémarré par l'envoi de la part du serveur d'un message spécial du protocole X.

Ces messages d'événements incluent ceux qui :

- demandent à un client X de redessiner sa fenêtre (par exemple, si une partie de sa fenêtre est découverte par le mouvement d'une autre fenêtre)[1],
- informent que la taille d'une fenêtre a changé,
- informent qu'une touche du clavier a été pressée,
- etc.

Le client X traite ces messages, puis revient à un état suspendu, jusqu'à ce qu'un autre message soit reçu.

[1]C'est une caractéristique du système de fenêtrage X Window de ne pas s'occuper de redessiner les fenêtres lorsque c'est nécessaire : c'est une tâche qui est du ressort de chaque client.

Cette approche est en opposition directe avec la façon dont les applications traditionnelles sont écrites. Ces applications sont "orientées procédures" et sont écrites pour assumer un rôle actif dans l'interrelation entre l'utilisateur et le programme. Typiquement, le programme dirige l'utilisateur à travers l'exécution de la tâche, le forçant à travers un ensemble de procédures prédéfinies. Le programme n'acceptera l'intervention de l'utilisateur qu'à certains moments précis.

4.1.4 Un système distribué

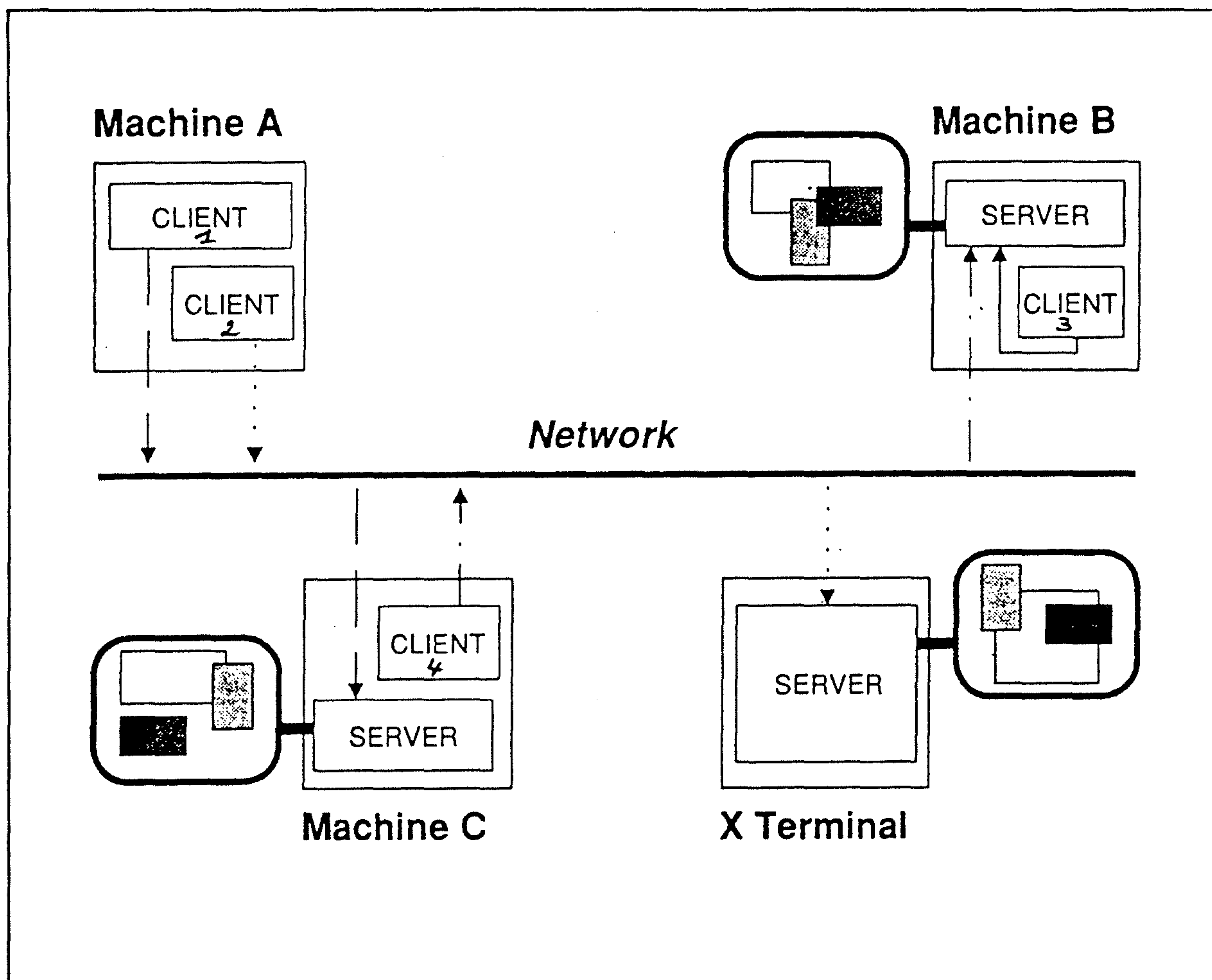

Figure 4.2: Fonctionnement de X Window en réseau

Puisque le client X communique avec le serveur X par l'envoi de messages d'information, il est possible aux requêtes du protocole X d'être envoyées, à travers

un réseau, à un serveur X tournant sur une machine différente. En fait, X Window, bâti autour d'un système de messages, est spécifiquement conçu pour être un système graphique fonctionnant sur réseau.

Dans la figure 4.2, on voit que le client X numéro 1, s'exécutant sur la machine A produit son affichage sur la machine C (en utilisant le serveur X de C), alors que le client X numéro 4, s'exécutant sur C produit son affichage sur l'écran de B.

Les différentes machines du réseau ne doivent pas nécessairement être de la même marque, ni même tourner sous le même système d'exploitation, puisque toutes les communications entre les clients X et les serveurs X sont effectuées à travers le réseau suivant un protocole de messages bien définis (le protocole X).

Un terminal X est une machine sur laquelle tourne uniquement un serveur X. Son seul objectif est d'afficher des graphiques pour des clients X tournant sur d'autres machines du réseau. Dans la figure 4.2, un des clients s'exécutant sur la machine A (client numéro 2) s'affiche sur un terminal X[2].

4.1.5 Le gestionnaire de fenêtres

Le serveur X produit seulement un affichage graphique en accord avec les requêtes du protocole X, mais ne fournit pas à l'utilisateur de fonctions pour contrôler la taille, la position et l'ordre d'empilement des fenêtres affichées.

Un client X spécial, appelé le gestionnaire de fenêtres (*Window Manager*) réalise cela. Ce programme dispose de privilèges spéciaux et a l'autorisation de superviser toutes les fenêtres affichées par le serveur X.

Typiquement, le gestionnaire de fenêtres place également quelques "décorations" autour de la fenêtre de chaque client X, décorations qui incluent des boutons de redimensionnement ou de déplacement, ainsi qu'une barre de titre.

C'est donc une fonction du gestionnaire de fenêtres de redimensionner, déplacer ou réarranger une fenêtre en accord avec les souhaits de l'utilisateur, qui clique sur un élément de décoration de la fenêtre, ou qui effectue une sélection à partir d'un menu du gestionnaire de fenêtres.

Il existe différents gestionnaires de fenêtres. Les plus connus sont :

[2]Typiquement, la majorité des implémentations de X Window sur PC se limitent à considérer le PC comme un terminal X. Ceci en raison des limitations de la mémoire des PC, qui font qu'un serveur X occupe normalement toute la mémoire du PC. Cependant, avec DESQview/X, les PC devraient pouvoir faire tourner simultanément un serveur X, des applications DOS et des clients X.

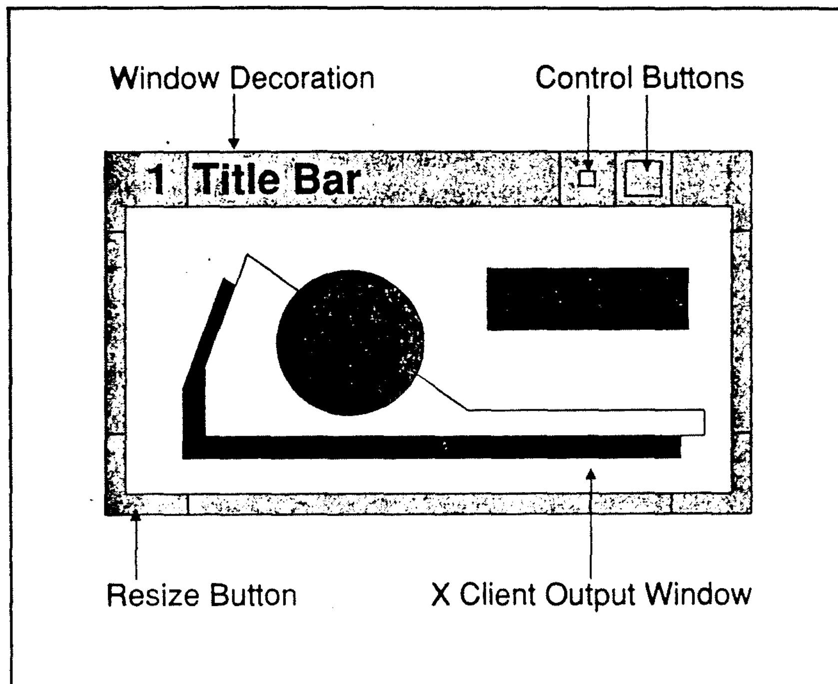

Figure 4.3: Les "décorations" d'une fenêtre X Window

- Tab
- OPEN LOOK
- OSF/Motif

Du fait de la conception de X Window, un gestionnaire de fenêtres peut être "fermé" et un autre "ouvert" pendant l'exécution, sans affecter aucun des clients X actuellement affichés à l'écran. Les anciennes décorations des fenêtres disparaissent de l'écran et sont remplacées par de nouvelles, créées par le nouveau gestionnaire de fenêtres.

Suivant la philosophie de X Window, une application peut donc être développée sans savoir sous quel gestionnaire de fenêtres elle va tourner. L'application utilise simplement les propriétés nécessaires pour communiquer avec le gestionnaire en question, qui fixera alors ou non une politique précise. Par exemple, *RTL* de Siemens fixe une politique de fenêtres contiguës; *uwm* du M.I.T. suit les requêtes des applications dans ce domaine.

Nous n'avons donc pas à nous préoccuper du gestionnaire de fenêtres qui sera utilisé.

4.1.6 Les couches de développement X

Xlib

Pour qu'un client X soit capable de communiquer avec un serveur X, il doit pouvoir générer des requêtes suivant le protocole X et les transmettre au serveur. Mais construire ces requêtes peut être fastidieux, et une librairie a été développée : *Xlib*, qui assure, entre autres, la gestion de la hiérarchie des fenêtres et la définition des attributs graphiques.

Xlib est (en général) l'interface de plus bas niveau qu'un client X utilise pour communiquer avec le serveur X. Il s'agit d'un ensemble de sous-routines écrites en C. Un client X peut être écrit en n'utilisant que Xlib, mais c'est assez rare.

Les toolkits

Etant donné que Xlib est assez rudimentaire (et donc assez lourd à utiliser), on a développé une autre couche au sommet de Xlib : les *toolkits*, basés sur Xlib, mais plus souples à utiliser[3].

Parmi les différents toolkits disponibles, on peut citer entre autres :

- InterViews (Stanford)
- XRay (HP)
- Andrew (CMU)
- Xtoolkit (M.I.T.)
  Un toolkit relativement rudimentaire, fourni par le MIT. C'est en fait le prolongement naturel de Xlib. Il s'agit d'ailleurs d'un standard reconnu par le Consortium X comme faisant partie du X Window System.
- Xol
  Toolkit se conformant au standard OPEN LOOK[4].
  Fourni par AT&T.

[3]Mais il est évident qu'un client X peut toujours (et il le fait souvent) appeler directement des fonctions Xlib, même s'il utilise un toolkit.

[4]OPEN LOOK n'est en fait lui-même ni un toolkit, ni un gestionnaire de fenêtres. Il s'agit plutôt d'une spécification de design pour l'apparence d'une interface utilisateur.
Xol et Xview sont deux toolkits qui adhèrent à cette spécification et offrent donc le même "look and feel".

- Xview
  Toolkit se conformant également au standard OPEN LOOK, mais avec une interface de programmation (SunView) différente de Xol.
  Fourni par SUN Microsystems.

- OSF/Motif
  Ce toolkit est fourni par l'Open Software Foundation et propose un "look 3D". Ce toolkit (et le gestionnaire de fenêtres complémentaires) sont soutenus par un consortium de compagnies (l'OSF), qui regroupe entre autres DEC, Hewlett-Packard et Microsoft.
  C'est le toolkit le plus répandu : il tend à s'imposer comme un standard de fait, notamment au niveau des PC. C'est donc lui que nous avons décidé d'adopter.

Certains toolkits peuvent être regardés comme une seule entité, mais d'autres, comme les toolkits Xtoolkit, Xol et OSF/Motif sont en fait constitués de deux librairies : l'Intrinsic library et la Widget library.

Intrinsic library Il s'agit d'une abstraction de Xlib, qui présente un modèle particulier de design pour l'interface application/utilisateur.

OSF/Motif et Xol utilisent en fait une forme modifiée de Xt, la Intrinsic library du toolkit Xtoolkit.

Widget library Les widgets sont des objets génériques d'interface utilisateur, tels des boutons, des menus, des barres de défilement, etc. Nous développerons cette importante notion dans la prochaine section.

Xaw (Athena Widget Set) est la Widget library de Xtoolkit.

Schéma récapitulatif

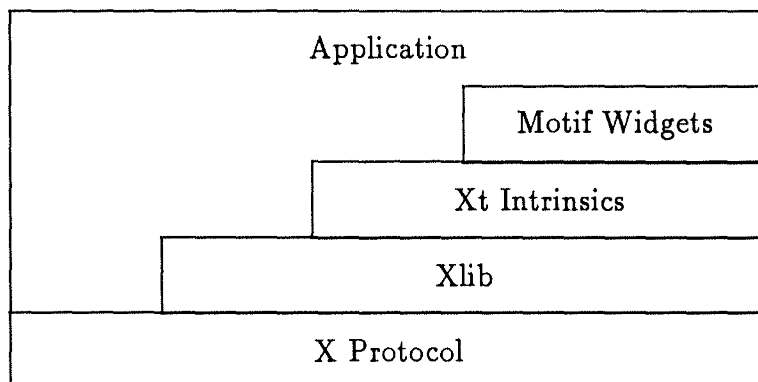

Figure 4.4: X Window : les couches de développement

4.1.7 La hiérarchie des widgets

Une *widget* est une fenêtre X à laquelle sont attachées diverses fonctions, qui permettent de manipuler cette fenêtre et de réagir à certains types d'événements. Chaque widget contient également une structure de données correspondant à l'information utilisée par les fonctions de la widget.

En fait, X Window définit une architecture orientée objet qui organise les widgets en classes. En général, une classe est un ensemble d'objets qui possèdent des caractéristiques semblables. Les objets appartenant à une classe sont appelés des instances de cette classe.

Dans la librairie *XtIntrinsics*, la fonction *XtCreateWidget ()* crée une instance de la classe de widgets donnée en argument.

La notion d'héritage est un autre concept de l'orientation objet utilisée par *XtIntrinsics*. D'une manière générale, une classe hérite des caractéristiques d'une autre classe (parfois appelée superclasse), un peu comme les enfants héritent des caractéristiques génétiques de leurs parents. Par exemple, Motif définit une classe de widgets *Label*, et une sous-classe de celle-ci : la classe *PushButton*. Un objet de la classe *PushButton* hérite de la classe *Label* le fait d'avoir un label inscrit à l'intérieur d'un rectangle. Mais il possède en plus la possibilité de réagir lorsqu'un utilisateur clique le bouton qui le représente.

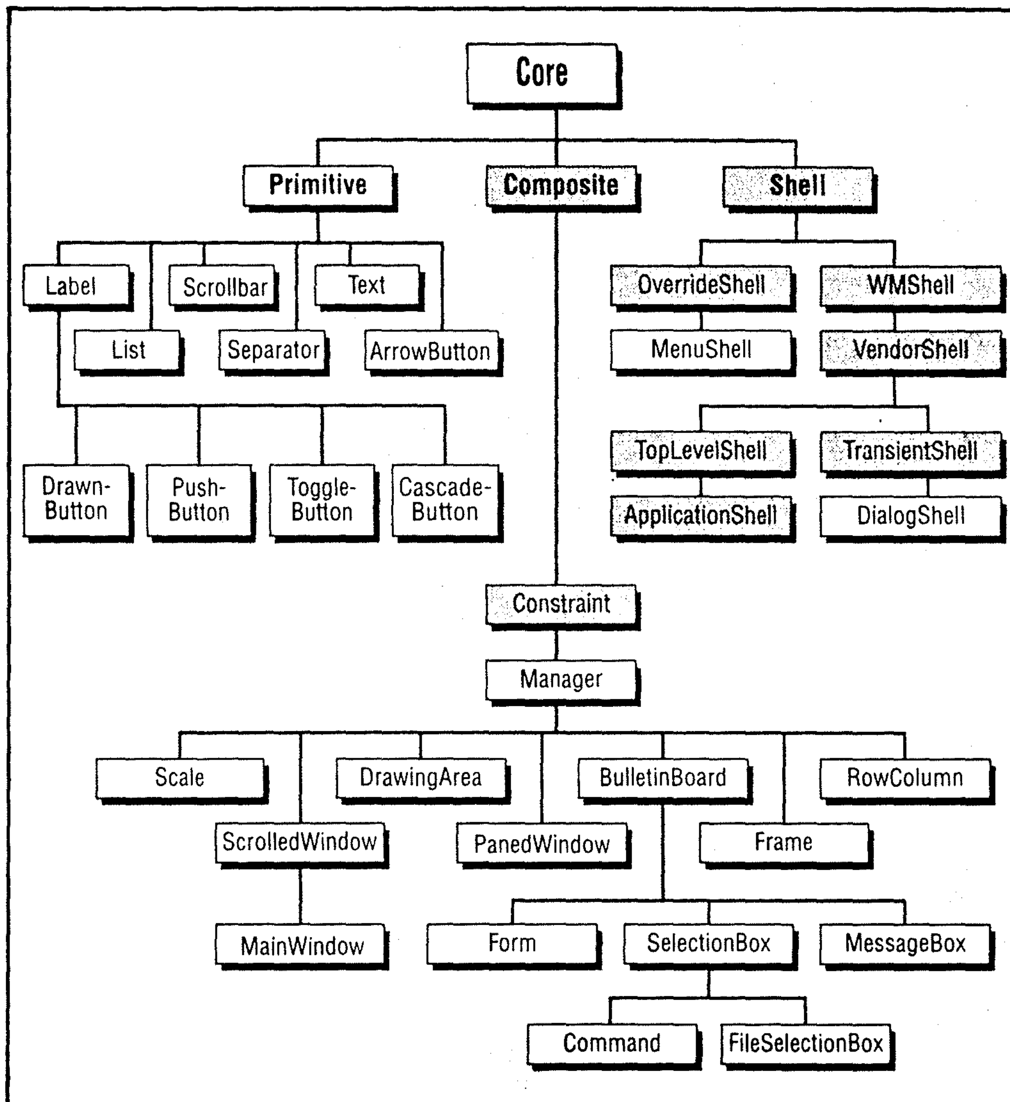

Figure 4.5: La hiérarchie des widgets *Motif*

4.1.8 Gestionnaires d'événements et *callbacks*

Les gestionnaires d'événements

Un gestionnaire d'événements (*event handler*) est une procédure appelée par *Intrinsics* lorsqu'un type spécifique d'événement se produit à l'intérieur d'une widget.

Le programmeur de l'application peut ajouter ses propres gestionnaires d'événements au moyen de la fonction *XtEventHandler ()*. Cette fonction admet comme paramètres :

- L'identification de la widget concernée.
- L'identification du type d'événement que l'on veut traiter. Citons quelques événements parmi les plus courants :
  - l'utilisateur a pressé une touche du clavier,
  - l'utilisateur a relâché une touche du clavier,
  - l'utilisateur a pressé un bouton de la souris,
  - l'utilisateur a relâché un bouton de la souris,
  - l'utilisateur a déplacé la souris,
  - l'utilisateur a déplacé la souris, un des boutons étant enfoncé,
  - le curseur est entré à l'intérieur d'une fenêtre,
  - le curseur a quitté une fenêtre,
  - une partie d'une fenêtre, initialement cachée, est de nouveau visible,
  - une fenêtre a été redimensionnée,
  - etc.
- La fonction (définie par l'application) qui doit être appelée lorsque l'événement se produit.
- L'adresse d'un paramètre quelconque que l'on voudrait communiquer à la fonction appelée.

Les *callbacks*

Certaines widgets offrent la possibilité de définir des procédures qui seront appelées lorsque certaines conditions, spécifiques à la classe de widgets concernée, se produiront. Ces fonctions sont enregistrées dans ce que l'on appelle des listes de callbacks. Chaque widget gère une liste de telles fonctions pour chaque type de callbacks supporté par la classe de widget. Les types de callbacks supportés par une classe font partie de la définition de la classe, et sont donc à ce titre hérités par les classes dérivées. C'est ainsi que chaque widget possède une liste du type *XtNdestroyCallback*. Chaque fonction appartenant à cette liste d'une widget sera appelée lorsque cette widget sera détruite.

Les applications peuvent ajouter une fonction à une liste de callbacks au moyen de la fonction *XtAddCallback ()*. Cette fonction admet comme paramètres :

- l'identification de la widget concernée,
- le type de callbacks concerné,
- la fonction à appeler lorsque la condition se produit,
- l'adresse d'un paramètre quelconque que l'on voudrait communiquer à la fonction appelée.

Les callbacks diffèrent des gestionnaires d'événements par le fait qu'ils sont appelés par la widget plutôt que par *Intrinsics*, et qu'ils ne sont pas nécessairement liés à un événement particulier. Effectivement, l'utilisateur a la possibilité de modifier la condition qui provoque l'appel des fonctions d'une liste de callbacks. Voyons cela sur un exemple. Considérons la liste de callbacks *XtNselect*. La sélection est un concept abstrait qui n'est pas nécessairement lié à un événement particulier. Par défaut, l'utilisateur sélectionne une widget lorsqu'il presse le bouton de gauche de la souris lorsque le curseur se trouve à l'intérieur de la fenêtre. Si l'utilisateur a enregistré une fonction *quit ()* dans la liste *XtNselect*, il lui suffit donc de presser le bouton gauche de la souris pour quitter l'application. Mais il peut redéfinir la condition de sélection en indiquant[5], par exemple, qu'elle correspond au fait de presser la touche *Q*. Dès lors, c'est en pressant cette touche que l'utilisateur quittera l'application.

[5] dans un fichier, qui ne sera utilisé qu'au moment de l'exécution : il ne faut même pas recompiler le programme !

4.1.9 Modèle de programmation sous X Window

La plupart des applications qui utilisent *Xt Intrinsics* possèdent une structure semblable, et chaque application doit réaliser différentes étapes de base qui sont :

1. Initialiser *Intrinsics.*
   Cette étape établit une connexion avec le serveur X, alloue les ressources nécessaires et effectue différentes initialisations de la couche *Intrinsics.*

2. Créer les widgets.
   Chaque programme crée une ou plusieurs widgets, afin de construire l'interface entre le programme et l'utilisateur.

3. Enregistrer les *callbacks* et les gestionnaires d'événements.
   Les *callbacks* et les gestionnaires d'événements sont des fonctions définies par l'application, qui réagissent aux actions de l'utilisateur à l'intérieur de chaque widget.

4. Réaliser tous les widgets.
   C'est la réalisation d'une widget qui crée la fenêtre X utilisée par la widget.

5. Entrer dans la boucle d'événements.
   La plupart des applications X sont complètement *event-driven*, et sont donc construites pour boucler "indéfiniment" : attente d'un événement - traitement de cet événement - attente d'un événement - ...

4.2 Description générale de l'interface

De quoi l'utilisateur a-t-il besoin ?

- d'un menu lui permettant d'avoir accès aux différentes fonctionnalités de l'éditeur et du menu sous-jacent,
- de la représentation graphique du réseau routier,
- de barres de défilement lui permettant de déplacer la zone visualisée à l'écran.

Il peut être particulièrement utile d'ajouter à ces trois éléments essentiels deux zones de dialogue avec le programme :

- une zone d'affichage des coordonnées du curseur,
- une zone où, en fonction des commandes choisies par l'utilisateur, le programme le guide en lui indiquant la marche à suivre.

Voici donc à quoi ressemble l'interface développée :

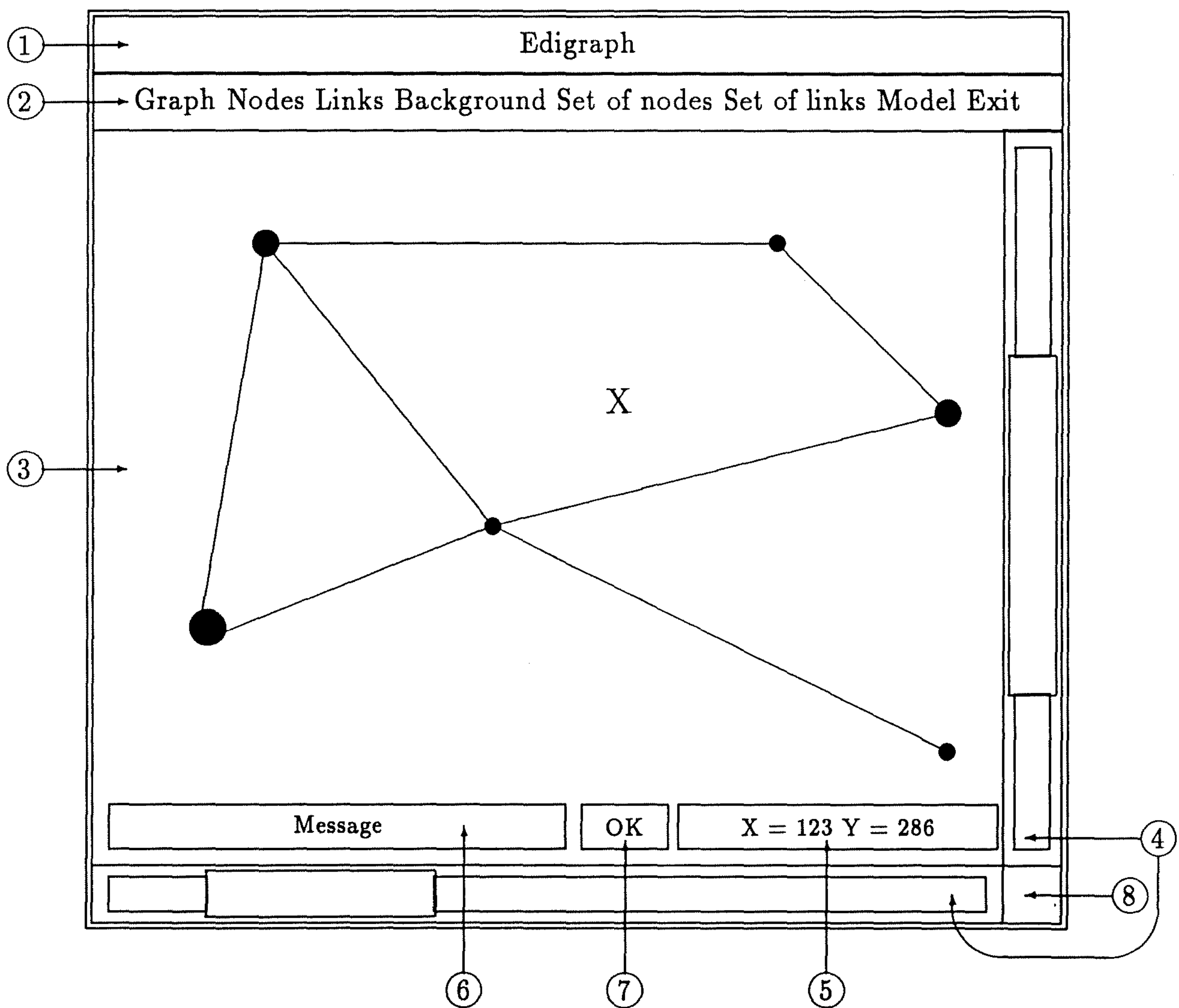

Figure 4.6: L'interface

L'éditeur se présente sous la forme d'un ensemble de fenêtres que nous allons rapidement passer en revue :

1. La barre de titre, qui contient le nom du programme.
   En réalité, il ne s'agit pas d'une fenêtre, mais d'une ornementation ajoutée par le gestionnaire de fenêtres (*window manager*).

2. La barre de menus.
   Il s'agit en fait de menus déroulants, sur lesquels nous aurons l'occasion de revenir.
   Les différentes fonctionnalités ont été regroupées par "grands thèmes" :

   - ce qui concerne le graphe dans son ensemble,
   - ce qui concerne les nœuds individuellement,
   - ce qui concerne les arcs individuellement,
   - ce qui concerne les éléments du background,
   - ce qui concerne un ensemble de nœuds (ensemble pouvant être défini de différentes manières),
   - ce qui concerne un ensemble d'arcs (ensemble pouvant être défini de différentes manières),
   - les fonctionnalités spécifiques au modèle sous-jacent,
   - la sortie du programme (avec possibilité d'annuler).

3. La fenêtre contenant le dessin du graphe proprement dit.

4. Les barres de défilement (*scrollbars*) verticale et horizontale, qui permettent de déplacer la zone d'affichage du graphe.

5. La fenêtre contenant les coordonnées du curseur.

6. La fenêtre contenant les messages d'attente du programme envers l'utilisateur.
   Par exemple "Waiting for a menubar selection ...".

7. Une fenêtre à existence temporaire, contenant le label *OK*.
   Cette fenêtre apparaît lorsque l'on a choisi la commande *Sélection des nœuds par la souris* ou *Sélection des arcs par la souris.* Lorsque l'utilisateur a terminé sa sélection, il valide celle-ci en cliquant sur cette fenêtre, qui disparaît alors.

8. Une fenêtre vide, qui sert seulement à assurer la cohérence des positions des fenêtres entre elles (voir la section sur la *frameworkWidget*).

Outre ces fenêtres fixes, il existe également de nombreuses fenêtres apparaissant au milieu de l'écran, en fonction des commandes sélectionnées : fenêtre de demande d'information à l'utilisateur (identificateur de nœud, d'arc, ...), fenêtre présentant la palette de couleurs disponibles, fenêtre présentant les différents types de représentation pour les nœuds, etc.

En fait, il s'agit bien plus que de simples fenêtres, c'est-à-dire des zones d'affichage à l'écran, définies par leur position, leur géométrie et leur contenu. C'est bien sûr cela, mais ce sont également des ensembles de données et de routines (*callbacks*) qui permettent au programme de réagir à différents événements susceptibles de se produire : que se passe-t-il si l'utilisateur clique sur tel bouton? Que se passe-t-il si la fenêtre voit sa taille modifiée ? Que se passe-t-il si l'on veut modifier la position des fenêtres filles ? Etc. Bref, ce sont des *widgets*.

4.3 Description des fonctionnalités

4.3.1 Sous-menu Graph

Create

La fonctionnalité *Create* a pour effet de créer le graphe à partir d'un fichier ne contenant que des informations élémentaires : identificateur et coordonnées pour les nœuds; identificateur, nœuds extrémités et coordonnées des points intermédiaires pour les arcs. Les autres caractéristiques (couleur, taille, type de représentation, ...) sont attribués arbitrairement.

Save

La fonctionnalité *Save* a pour objet de sauvegarder l'ensemble des informations graphiques relatives à la session de travail en cours : non seulement toutes les caractéristiques concernant le graphe lui-même, mais également celles concernant la zone actuellement affichée à l'écran, le facteur d'agrandissement, ...

Load

La fonctionnalité *Load* a pour objet de récupérer toutes ces informations.

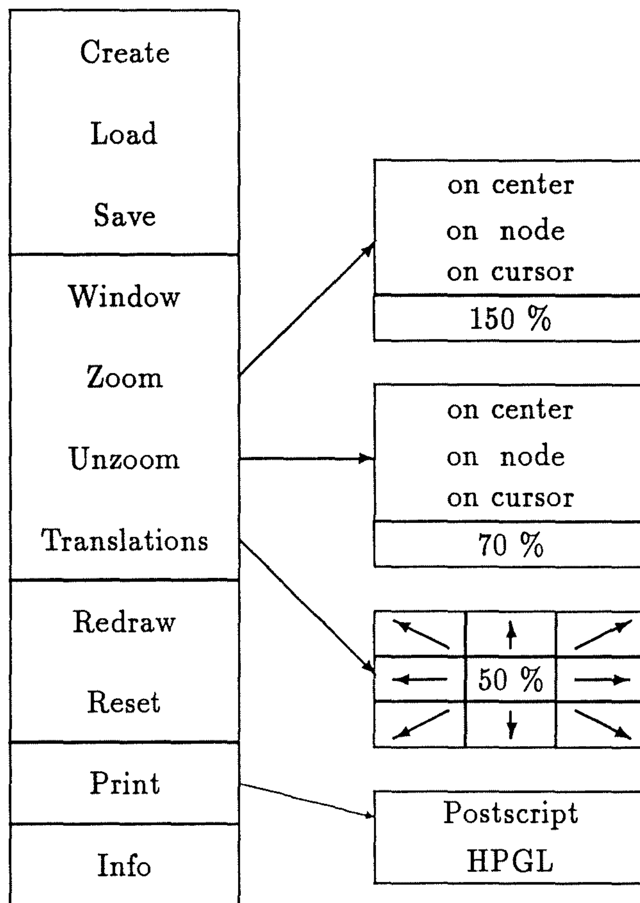

Figure 4.7: Description du sous-menu *Graph*

Window

La fonctionnalité *Window* permet d'agrandir le graphe. L'utilisateur est invité à délimiter, à l'aide de la souris, la zone qu'il veut voir à l'écran.

Ici se pose un problème. Comment réagir si la fenêtre définie par l'utilisateur n'a pas les mêmes proportions (même rapport longueur/largeur) que la région de l'écran disponible pour l'affichage ? Différentes solutions sont possibles :

- Augmenter la dimension déficiente, de telle sorte que l'on obtienne les mêmes proportions. C'est la solution retenue dans le programme XDrive. Dans ce cas, l'affichage ne correspond pas exactement à ce que l'utilisateur a demandé.

- Faire en sorte que le problème ne se pose pas : lorsque l'utilisateur détermine la fenêtre, le rectangle tracé apparaît directement avec les proportions correctes.

Cette solution se ramène à la première, sinon que l'utilisateur est au courant de ce qu'il verra.

- N'afficher que (et exactement) la fenêtre déterminée par l'utilisateur, même si ces proportions s'éloignent très fort des celles de l'écran.

Nous avons décidé d'adopter la dernière solution. Si à l'usage elle crée des problèmes à l'utilisateur (par exemple lors d'un *Unzoom* ultérieur), cette décision peut bien sûr être remise en question.

Zoom

La fonctionnalité *Zoom* permet également d'agrandir le graphe, mais de manière beaucoup plus facilement reproductible[6]. Il est loisible à l'utilisateur de modifier le facteur d'agrandissement. Il doit ensuite sélectionner la méthode de recentrage éventuel du graphe. La nouvelle zone affichée à l'écran :

- ne voit pas son centre modifié (*on center*);
- a pour nouveau centre un nœud particulier (*on node*) : le programme demande son identificateur à l'utilisateur;
- a pour nouveau centre la position du curseur au moment où l'utilisateur clique sur la souris. Lorsque l'utilisateur choisit cette méthode de recentrage, les coordonnées du curseur sont affichées dans la *widget* Coordonnées.

Unzoom

La fonctionnalité *Unzoom* présente exactement les mêmes possibilités, mais le facteur d'agrandissement est inversé.

Translations

La fonctionnalité *Translations* permet à l'utilisateur de déplacer la zone du graphe affichée à l'écran. Bien sûr, l'utilisateur peut utiliser les barres de déroulement (*scrollbars*), mais il est important pour l'utilisateur, toujours dans un souci de reproductibilité du résultat, qu'il puisse fixer de manière précise l'ampleur du

[6]Dans le cas où l'utilisateur veut conserver une trace écrite de plusieurs simulations différentes, il est essentiel pour lui que ces documents imprimés correspondent exactement à la même région du graphe.

déplacement. Celle-ci peut être modifiée, et est exprimée en pourcentage de la taille de la zone affichée à l'écran. L'utilisateur doit alors sélectionner une des huit directions de déplacement possibles.

Redraw

La fonctionnalité *Redraw* permet à l'utilisateur, au cas où cela serait nécessaire, de rafraîchir l'affichage, en procédant à l'effacement de la fenêtre et à un nouveau dessin du graphe.

En fait, lorsqu'une modification intervient dans le graphe, le programme le redessine complètement, plutôt que de procéder à une modification partielle, éventuellement imparfaite, de l'affichage. Dans la pratique, nous n'avons jamais eu à utiliser la fonctionnalité *Redraw*, qui semble donc superflue.

Reset

La fonctionnalité *Reset* a pour effet de réinitialiser la position ainsi que le facteur d'agrandissement de la zone affichée du graphe.

Print

La fonctionnalité *Print* offre la possibilité à l'utilisateur de créer un fichier d'impression du graphe (ou plus exactement, d'une partie de celui-ci, telle qu'elle apparaît à l'écran), soit au format Postscript, soit au format HPGL.

Info

La fonctionnalité *Info* permet à l'utilisateur de consulter et d'éditer l'information globale concernant le graphe. Cette information est fournie par le modèle sous-jacent. On pourrait par exemple y trouver le nom de la ville dont le trafic est modélisé, la date à laquelle les données ont été collectées, etc.

4.3.2 Sous-menu Nodes

Add

La fonctionnalité *Add* permet à l'utilisateur d'ajouter un nœud au graphe. Il en indique la position dans le graphe à l'aide de la souris (les coordonnées du

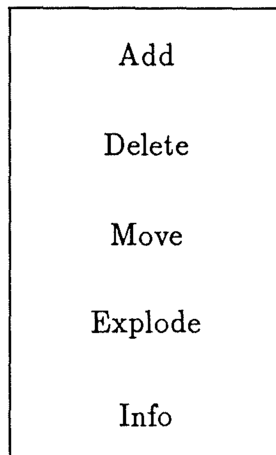

Figure 4.8: Description du sous-menu *Nodes*

curseur apparaissent dans la *widget* Coordonnées). La fonction correspondante du modèle sous-jacent est appelée, afin d'éventuellement introduire toute information nécessaire concernant ce nœud.

Delete

La fonctionnalité *Delete* permet à l'utilisateur de supprimer un nœud du graphe. Il indique le nœud à supprimer en cliquant dessus avec la souris. Tous les arcs partant de ce nœud ou y arrivant sont également supprimés. La fonction correspondante du modèle sous-jacent est appelée afin de maintenir la cohérence de la base de données.

Il est à noter que lorsqu'un noeud est supprimé, il ne disparaît pas du tableau qui contient les nœuds du graphe : son identificateur interne est simplement mis à zéro. Cette façon de procéder présente deux avantages. D'une part, elle ne nécessite pas de renumérοter tous les nœuds qui suivent, dans le tableau, le nœud que l'on veut supprimer. D'autre part, le nœud existant toujours dans la structure du graphe, il sera facile d'implémenter une fonction *Undelete*. L'utilisation de la fonctionnalité *Save* aura pour effet de sauvegarder le graphe dans son état modifié, et donc de rendre les suppressions effectives et irréversibles. On pourrait songer à implémenter une fonctionnalité *Purge*, qui aurait le même effet, mais sans interrompre la session de travail.

Move

La fonctionnalité *Move* permet à l'utilisateur de déplacer un nœud. Il en indique la nouvelle position à l'aide de la souris (les coordonnées du curseur apparaissent dans la *widget* Coordonnées).

Explode

Actuellement, pour calculer le plus court chemin d'un nœud vers un autre, seuls les coûts sur les arcs interviennent. En d'autres termes, l'algorithme actuel ne tient pas compte du coût qui pourrait résulter du simple passage à travers un carrefour. On ne peut pas affirmer que le coût associé à un carrefour est attribué à l'arc qui le précède. En effet, en vertu de la règle de priorité de droite, par exemple, il est évident que le coût associé à la traversée d'un carrefour dépend du noeud d'où l'on vient et du noeud où l'on va. Pour attribuer un coût au passage par un noeud, on a le choix entre deux grandes possibilités. La première consiste en un éclatement du graphe initial. La seconde consiste à stocker les coûts correspondant au passage par un carrefour à un endroit de la structure de données actuellement utilisée dans le modèle, et à s'en servir au moment opportun. Cette deuxième solution est appelée *turn*. Quelle que soit la solution adoptée, il y a là des informations supplémentaires attachées à chaque noeud, et qu'il est essentiel de pouvoir éditer de manière graphique et interactive.

Info

La fonctionnalité *Info* permet à l'utilisateur de consulter et d'éditer l'information relative à un nœud spécifié par l'utilisateur. Cette information dépend du modèle sous-jacent.

4.3.3 Sous-menu Links

Add

La fonctionnalité *Add* permet à l'utilisateur d'ajouter un arc au graphe. Il en indique les deux nœuds extrémités à l'aide de la souris. La fonction correspondante du modèle sous-jacent est appelée, afin d'éventuellement introduire toute information nécessaire concernant cet arc.

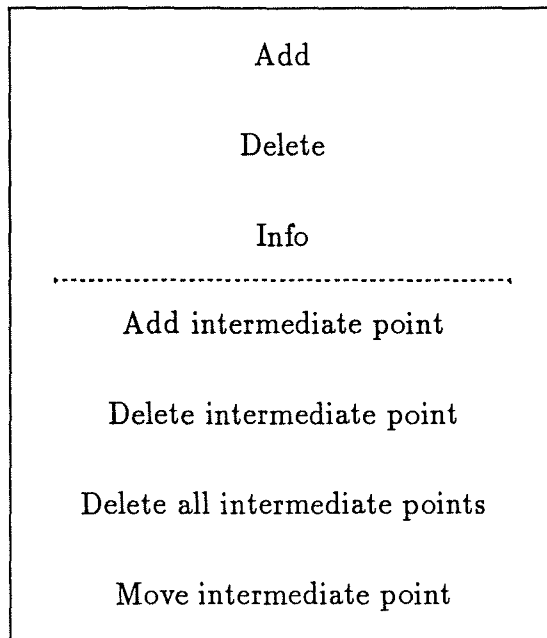

Figure 4.9: Description du sous-menu *Links*

Delete

La fonctionnalité *Delete* permet à l'utilisateur de supprimer un arc du graphe. Il indique l'arc à supprimer en cliquant dessus avec la souris. La fonction correspondante du modèle sous-jacent est appelée afin de maintenir la cohérence de la base de données.

De même que lorsque l'on supprime un nœud, l'arc supprimé ne disparaît pas de la structure de graphe, et pourra donc facilement être récupéré par une fonction *Undelete*.

Info

La fonctionnalité *Info* permet à l'utilisateur de consulter et d'éditer l'information relative à un arc spécifié par l'utilisateur. Cette information dépend du modèle sous-jacent.

Add intermediate point

Cette fonctionnalité permet d'ajouter un point intermédiaire pour le tracé d'un arc. L'utilisateur indique d'abord le segment de l'arc sur lequel il veut ajouter ce point intermédiaire, puis il indique la position de ce point intermédiaire à l'aide de la souris (les coordonnées du curseur apparaissent dans la *widget* Coordonnées).

Delete intermediate point

Cette fonctionnalité permet de supprimer un point intermédiaire. L'utilisateur indique d'abord quel est l'arc concerné. Le programme affiche alors de manière explicite quels sont les points intermédiaires de l'arc (ou un message d'erreur si l'arc ne contient pas de point intermédiaire). Il reste à l'utilisateur à préciser le point qu'il veut supprimer.

Delete all intermediate points

Il suffit à l'utilisateur d'indiquer quel est l'arc dont il veut supprimer tous les points intermédiaires.

Move intermediate point

Cette fonctionnalité permet de déplacer un point intermédiaire. L'utilisateur indique d'abord quel est l'arc concerné. Le programme affiche alors de manière explicite quels sont les points intermédiaires de l'arc (ou un message d'erreur si l'arc ne contient pas de point intermédiaire). Il reste à l'utilisateur à préciser le point qu'il veut déplacer et à indiquer la nouvelle position de ce point intermédiaire à l'aide de la souris (les coordonnées du curseur apparaissent dans la *widget* Coordonnées).

4.3.4 Sous-menu Set of nodes

Les fonctionnalités ici accessibles se répartissent en deux groupes : d'une part celles qui servent à sélectionner un ensemble de nœuds, et d'autre part celles qui permettent de manipuler l'ensemble de nœuds sélectionnés. L'appel à une fonctionnalité du deuxième groupe alors qu'aucun nœud n'a été sélectionné provoque l'affichage d'un message d'erreur.

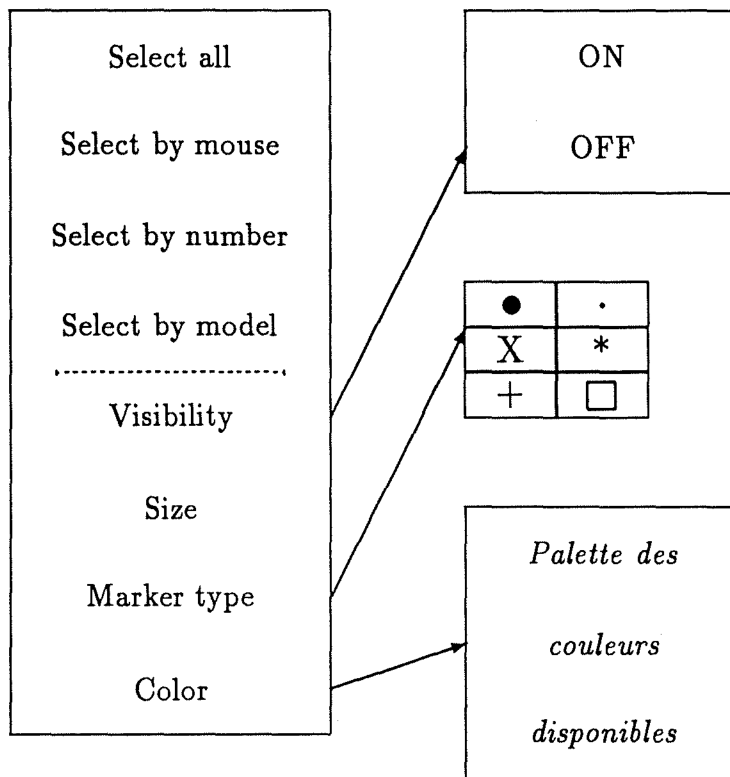

Figure 4.10: Description du sous-menu *Set of nodes*

Select all

Cette fonctionnalité permet de sélectionner tous les nœuds.

Select by mouse

L'utilisateur indique à l'aide de la souris quels sont les nœuds qu'il veut sélectionner. Lorsqu'il a terminé, il valide sa sélection en cliquant sur la fenêtre *Ok*.

Select by number

Le programme demande à l'utilisateur la liste des identificateurs des nœuds qu'il veut sélectionner.

Select by model

Le modèle sous-jacent propose différentes sélections de nœuds possibles. Par exemple, les carrefours avec feux de signalisation, ou bien ceux avec priorité de droite. Etc. Cette fonctionnalité est bien sûr dépendante du modèle sous-jacent.

Visibility

Cette fonctionnalité permet de rendre visibles ou invisibles l'ensemble de nœuds sélectionnés.

Size

Cette fonctionnalité permet de modifier la taille de la représentation des nœuds sélectionnés. Il est à noter que, dans une certaine mesure, la taille des marqueurs utilisés pour la représentation des nœuds dépend également du facteur d'agrandissement du graphe à l'affichage.

Marker type

Cette fonctionnalité propose les différents types de marqueurs disponibles, et modifie la représentation des nœuds sélectionnés en fonction du choix de l'utilisateur.

Color

Cette fonctionnalité propose la palette des différentes couleurs disponibles, et modifie la couleur des nœuds sélectionnés en fonction du choix de l'utilisateur.

4.3.5 Sous-menu Set of links

De la même façon que pour les *Set of nodes*, les fonctionnalités ici accessibles se répartissent en deux groupes : sélection et manipulation.

Select all

Cette fonctionnalité permet de sélectionner tous les arcs.

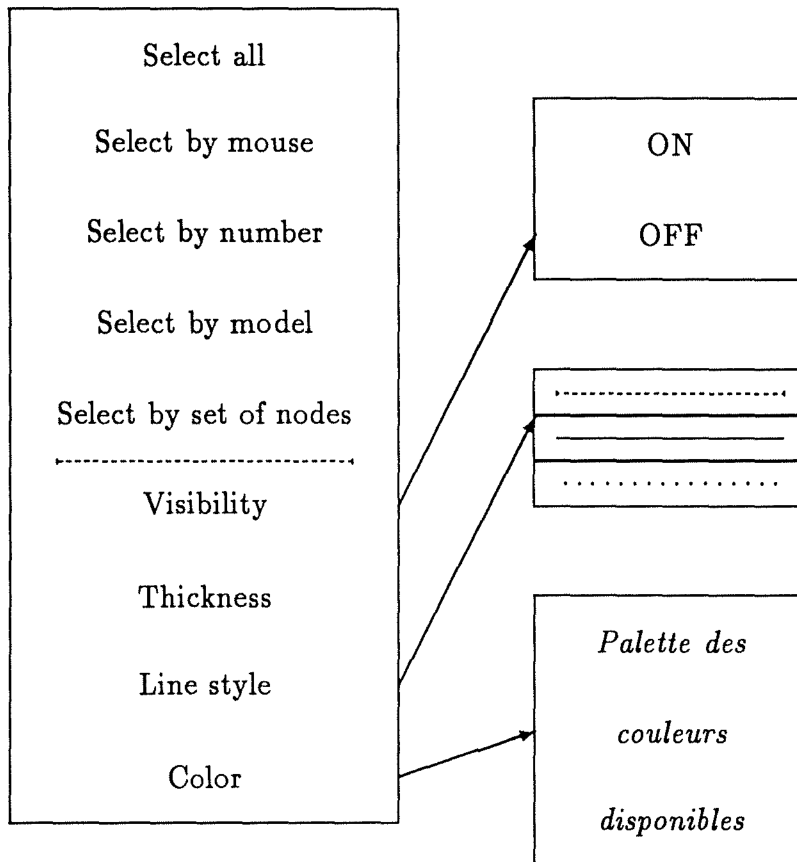

Figure 4.11: Description du sous-menu *Set of links*

Select by mouse

L'utilisateur indique à l'aide de la souris quels sont les arcs qu'il veut sélectionner. Lorsqu'il a terminé, il valide sa sélection en cliquant sur la fenêtre *Ok*.

Select by number

Le programme demande à l'utilisateur la liste des identificateurs des arcs qu'il veut sélectionner.

Select by model

Le modèle sous-jacent propose différentes sélections d'arcs possibles. Par exemple, les routes nationales, provinciales et communales. Ou bien les voies à sens

unique. Etc. Cette fonctionnalité est bien sûr dépendante du modèle sous-jacent.

Select by set of nodes

Cette façon de sélectionner les arcs suppose qu'un ensemble de nœuds aient déjà été sélectionnés. L'ensemble des arcs ainsi sélectionnés est constitué de tous les arcs partant ou aboutissant à l'un des nœuds précédemment sélectionnés.

Visibility

Cette fonctionnalité permet de rendre visibles ou invisibles l'ensemble des arcs sélectionnés.

Thickness

Cette fonctionnalité permet de modifier l'épaisseur du tracé des arcs sélectionnés.

Line style

Cette fonctionnalité propose les différents styles de tracé disponibles (continu, tirets ou pointillés), et modifie le tracé des arcs sélectionnés en fonction du choix de l'utilisateur.

Color

Cette fonctionnalité propose la palette des différentes couleurs disponibles, et modifie la couleur des arcs sélectionnés en fonction du choix de l'utilisateur.

4.3.6 Sous-menu Background

Add

Cette fonctionnalité permet d'ajouter un élément au décor. L'utilisateur doit préciser s'il s'agit d'un élément "ouvert" (*polyline* : lignes de chemins de fer, cours d'eau, ...) ou "fermé" (*polygon* : bâtiments, parcs, ...). L'utilisateur indique alors à l'aide de la souris (ou d'une table à digitaliser) les différents points qui permettent le tracé de cet élément du décor.

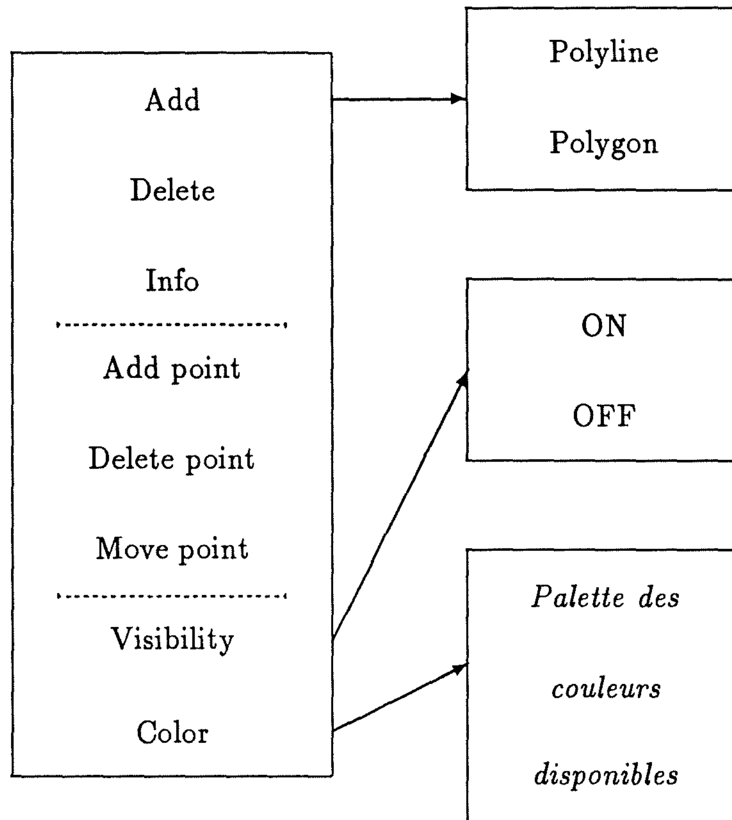

Figure 4.12: Description du sous-menu *Background*

Delete

Cette fonctionnalité permet de supprimer un élément du décor.

Info

Cette fonctionnalité permet à l'utilisateur de consulter et d'éditer l'information relative à un élément du décor. Cette information dépend du modèle sous-jacent.

Add point - Delete point - Move point

Ces fonctionnalités permettent à l'utilisateur d'ajouter, supprimer ou déplacer un point pour le tracé de l'élément de décor spécifié.

Visibility

Cette fonctionnalité permet de rendre visible ou invisible l'élément du décor sélectionné.

Color

Cette fonctionnalité propose la palette des différentes couleurs disponibles et modifie la couleur de l'élément du décor sélectionné en fonction du choix de l'utilisateur.

4.3.7 Sous-menu Model

Il est évident que les fonctionnalités disponibles ici sont complètement dépendantes du modèle sous-jacent.

4.4 Le système de menus

Voyons rapidement comment le système de menus en cascade a été conçu pour être le plus souple possible et pouvoir s'adapter sans problème à une extension des fonctionnalités. Le mécanisme utilisé est inspiré de [9].

Nous définissons un type structuré *menuStruct*, qui se présente comme suit :

```
typedef struct _menuStruct

 {char                  *name;                   /* name of the button */
  void                 (*func)();            /* callback to be invoked */
  caddr_t               data;                 /* data for the callback */
  struct _menuStruct *subMenu; /* data for submenu of this button */
  int                   nbrSubItems;  /* how many items in sub-menu */
  char                 *subMenuTitle;              /* title of submenu */
 } menuStruct;
```

Dès lors, l'ensemble des informations nécessaires pour la construction du menu sont contenues dans des tableaux de type *menuStruct*, définis dans le fichier *menu.h*. Considérons par exemple les fonctionnalités relatives au sous-menu *Graph*. Elles correspondent à la variable *menuGraph []*, définie comme suit :

```
static menuStruct menuGraph[] =
  { { "Create", graphCreate, NULL, NULL, 0, NULL},
    { "Load", graphLoad, NULL, NULL, 0, NULL},
    { "Save", graphSave, NULL, NULL, 0, NULL},
    { NULL, NULL, NULL, NULL, 0, NULL},
    { "Window", graphWindow, NULL, NULL, 0, NULL},
    { "Zoom", graphZoom, NULL, NULL, 0, NULL},
    { "UnZoom", graphUnzoom, NULL, NULL, 0, NULL},
    { "Translations", graphTranslations, NULL, NULL, 0, NULL},
    { NULL, NULL, NULL, NULL, 0, NULL},
    { "Redraw", graphRedraw, NULL, NULL, 0, NULL},
    { "Reset", graphReset, NULL, NULL, 0, NULL},
    { NULL, NULL, NULL, NULL, 0, NULL},
    { "Print", NULL, NULL, menuPrint, XtNumber (menuPrint), NULL},
    { NULL, NULL, NULL, NULL, 0, NULL},
    { "Info", graphInfo, NULL, NULL, 0, NULL}
  };
```

Chaque élément du tableau correspond à une option du sous-menu. Le premier champ de chaque élément contient un pointeur vers le nom de l'option. Si ce pointeur est nul (c'est le cas pour les 4ème, 9ème, 12ème et 14ème éléments), il s'agit simplement d'un séparateur. Le deuxième champ contient un pointeur vers la fonction à appeler si l'option est sélectionnée. Si ce pointeur est nul (c'est le cas du 13ème élément), c'est que l'option choisie correspond à un sous-menu. Le quatrième champ est alors un pointeur vers le sous-menu correspondant (en l'occurrence *menuPrint*), et le cinquième champ indique le nombre d'options de ce sous-menu (la fonction *XtNumber* permet de connaître le nombre d'éléments d'un tableau). Dans le même fichier *menu.h*, on trouve bien sûr la définition de *menuPrint* :

```
static menuStruct menuPrint[] =
  { { "Postscript", postscriptPrint, NULL, NULL, 0, NULL},
    { "HPGL", HPGLPrint, NULL, NULL, 0, NULL}
  };
```

Supposons maintenant que l'on décide d'implémenter la possibilité d'obtenir un fichier Postscript couleurs. Il suffit simplement :

1. de modifier la définition de *menuPrint* :

```
static menuStruct menuPrint[] =
  { { "Postscript", NULL, NULL, menuPostscript,
 XtNumber (menuPostscript), NULL},
    { "HPGL", HPGLPrint, NULL, NULL, 0, NULL}
  };
```

2. de définir *menuPostscript* :

```
static menuStruct menuPostscript[] =
  { { "Black/White", postscriptPrint, NULL, NULL, 0, NULL},
    { "Color", colorPostscriptPrint, NULL, NULL, 0, NULL}
  };
```

3. d'implémenter la fonction *colorPostscriptPrint.*

4.5 Les widgets de l'interface

4.5.1 *topLevelWidget*

Toutes les applications doivent appeler la fonction *XtInitialize ()* avant d'appeler n'importe quelle autre fonction de *Intrinsics.* Cette fonction établit une connexion avec le serveur X et procède à différentes initialisations. Ensuite, *XtInitialize ()* crée une widget de classe *TopLevelShell* (en l'occurrence *topLevelShell*), et retourne son identificateur. Cette widget sert de base pour toutes les autres widgets de l'application.

4.5.2 *frameworkWidget*

Une widget de la classe *TopLevelShell* ne peut avoir qu'une "fenêtre fille". Dès lors, il faut généralement passer par l'intermédiaire d'une autre widget (en l'occurrence *frameworkWidget*), appartenant à une classe dérivée de la classe *Composite*, si l'on veut que l'application puisse gérer plusieurs widgets.

La widget *frameworkWidget* appartient à la classe *Form.* Les widgets de cette classe servent de conteneurs pour les autres widgets de l'application. La caractéristique de cette classe réside dans la manière dont on fixe la position des fenêtres filles. Pour chacune d'entre elles, et pour chacun de leurs quatre côtés, on doit préciser son mode d'"attachement" : soit à la fenêtre mère, soit à une des fenêtres

sœurs. C'est ainsi que, par exemple, la fenêtre contenant le dessin du graphe est attachée[7] :

- en haut : à la barre de menus,
- à droite : à la barre de défilement verticale,
- en bas : à la barre de défilement horizontale.
- à gauche : à la fenêtre mère (*frameworkWidget*).

4.5.3 *menuBarWidget*

La barre de menu appartient à la classe de widgets *RowColumn*. Tout comme les widgets de la classe *Form*, les widgets de cette classe servent de conteneurs pour d'autres widgets. Celles-ci sont arrangées en lignes et/ou colonnes. Différentes variantes sont possibles : ligne unique, colonne unique, widgets filles de tailles identiques pour un aspect symétrique, etc.

La fonction *XmCreateMenuBar ()* crée une widget de la classe *RowColumn* (en l'occurrence *menuBarWidget*), et la configure pour pouvoir être utilisée comme une barre de menu[8]. Notamment, les seules widgets pouvant être définies comme ses filles doivent appartenir à la classe *CascadeButton*. A chacune des widgets de cette classe est associé un sous-menu.

4.5.4 *coordWidget*

coordWidget appartient à la classe de widgets *Text*. Cette classe offre un éditeur de texte rudimentaire, mono-ligne ou multi-lignes. La valeur par défaut, qui est utilisée ici, est l'éditeur mono-ligne.

Puisque *coordWidget* doit simplement afficher les coordonnées du curseur à l'écran, sans laisser à l'utilisateur la possibilité d'éditer ces coordonnées, le booléen *XmNeditable* est mis à *False*. On n'utilise donc en fait pas les possibilités d'édition de cette classe.

[7]voir figure 4.6

[8]Dans *Motif*, il n'y a pas de classe spécifique de widgets *MenuBar*

4.5.5 *messageWidget*

Cette widget, qui sert à afficher de courts messages d'aide à l'utilisateur, appartient également à la classe *Text*, et est utilisée avec les mêmes valeurs que *coordWidget*.

4.5.6 *okWidget*

Cette widget appartient à la classe *PushButton*. Les widgets de cette classe consistent en une chaîne de caractères (c'est le cas ici) ou un dessin, entourés d'un liseré. Lorsque la widget est sélectionnée, ce liseré change, pour donner l'impression que le bouton est pressé puis relâché.

Différentes listes de *callbacks* sont associées aux widgets de cette classe, qui gèrent les réactions du programme au fait de sélectionner, enfoncer ou relâcher le bouton.

4.5.7 *vertScrollWidget* et *horScrollWidget*

Ces deux widgets appartiennent à la classe *Scrollbar*. Une widget de cette classe est constituée de deux flèches, placées de part et d'autre d'un grand rectangle. Ce rectangle représente l'ensemble des données que l'utilisateur manipule. A l'intérieur, un rectangle plus petit, le curseur, représente les données actuellement visibles à l'écran.

Le champ *XmNorientation*, qui peut prendre comme valeurs *XmVERTICAL* et *XmHORIZONTAL*, permet de disposer de barres de défilement horizontales et verticales.

Différentes listes de *callbacks* sont associées aux widgets de cette classe, qui gèrent les réactions du programme au fait de déplacer le curseur. Elles sont bien sûr indispensables, puisqu'il faut actualiser en conséquence la partie de carte routière dessinée à l'écran.

Deux données sont essentielles à la gestion des barres de défilement : la position du curseur à l'intérieur du rectangle de référence, et sa taille, exprimée en pourcentage de ce rectangle de référence. Les fonctions *XmScrollBarGetValues* et *XmScrollBarSetValues* permettent respectivement de connaître ces valeurs et de les modifier. Ces deux fonctions sont utilisées lorsque l'utilisateur fait appel aux fonctionnalités *Window, Zoom, Unzoom* et *Translations* : la configuration des barres de déroulement doit rester cohérente avec l'affichage.

4.5.8 *canvasWidget*

canvasWidget est la widget utilisée pour le dessin proprement dit. Elle appartient à la classe *DrawingArea.* Une widget de cette classe est simplement une fenêtre vide. Elle n'offre en elle-même aucune possibilité particulière de dessin, et ne définit aucun comportement, sinon une liste de *callbacks* associées à la pression d'une touche du clavier ou d'un bouton de la souris.

4.5.9 *fillerWidget*

fillerWidget est également une widget de la classe *DrawingArea.* Elle ne joue aucun rôle, sinon d'assurer la cohérence des positions relatives des différentes widgets de l'application à l'intérieur de la *frameworkWidget.*

4.5.10 Les popup widgets

Nous regroupons sous le nom de "popup widgets" l'ensemble des fenêtres créées en réponse à certaines commandes de l'utilisateur : elles servent à dialoguer avec l'utilisateur, et ne sont pas définies comme des descendantes de *topLevelShell.* Elles sont créées au moyen de "fonctions d'utilité" (*convenience functions*).

Un *Dialogue* est un ensemble de widgets utilisé dans un but spécifique. Un *Dialogue* est constitué d'une widget de la classe *DialogShell*[9] (l'analogue de la *topLevelWidget* de l'application), d'une widget de la classe *BulletinBoard* ou d'une classe dérivée (l'analogue de la *frameworkWidget* de l'application : la classe *Form* est dérivée de la classe *BulletinBoard*) et de différentes autres widgets appartenant aux classes *Label, PushButton* et *Text.* L'ensemble des widgets qui constituent un *Dialogue* peuvent être construites une à une, mais il est plus facile d'utiliser une fonction d'utilité, qui crée l'ensemble des widgets en une fois.

La fonction *XmCreateMessageDialog* est utilisée pour envoyer un message d'erreur à l'utilisateur (identificateur inconnu, nombre erroné d'éléments dans le fichier de création du graphe, etc). La widget ainsi créée contient le message (widget de classe *Label*) ainsi qu'un bouton (widget de classe *PushButton*) dont le label par défaut est "OK".

[9]Une des caractéristiques des widgets de cette classe est de ne pas avoir d'existence autonome, en ce sens qu'elles ne peuvent pas être iconifiées séparément. Toutes les widgets de cette classe associées à une widget de classe *TopLevelShell* sont iconifiées et déiconifiées en même temps que la widget de l'application.

La fonction *XmCreatePromptDialog* est utilisée pour demander une information à l'utilisateur (identificateur d'un nœud, d'un arc, etc). La widget ainsi créée contient un message (widget de classe *Label*), une widget de classe *Text* permettant à l'utilisateur d'introduire l'information demandée, ainsi qu'un bouton (widget de classe *PushButton*) dont le label par défaut est "OK".

La fonction *XmCreateFormDialog* permet de créer une widget de la classe *Form* comme descendante de la widget *DialogShell*. Elle est utilisée pour présenter à l'utilisateur les types de marqueurs pour un nœud, les styles de tracé d'arcs, les mouvements possibles de translation et les méthodes de recentrage du graphe dans le cas d'un zoom. A chaque fois, la widget de classe *Form* contient différentes widgets de classe *PushButton* qui permettent à l'utilisateur d'opérer son choix. Dans le cas d'une translation ou d'un zoom, elle contient également une widget de classe *Text* qui permet d'éditer respectivement les facteurs de translation et d'agrandissement.

La fonction *XmCreateBulletinBoardDialog* permet de créer une widget de la classe *BulletinBoard* comme descendante de la widget *DialogShell*. Elle est utilisée pour présenter à l'utilisateur la palette des différentes couleurs disponibles. La widget de classe *BulletinBoard* ainsi créée contient 66 widgets de classe *PushButton*, chacune correspondant à une couleur différente. La position de chacune de ces widgets à l'intérieur de la widget mère est définie par la donnée de ses coordonnées (x, y).

Chapitre 5

Algorithmes graphiques

5.1 Transformation de fenêtrage

Pour de nombreuses applications, travailler dans le système de coordonnées du dispositif d'affichage n'est pas satisfaisant. En pratique, les objets sont représentés par des coordonnées réelles, correspondant à des dimensions physiques, alors que les coordonnées écran sont entières. Dès lors, on préfère décrire l'image dans un système de coordonnées défini par l'utilisateur (*world coordinate system*). Il faut alors utiliser une transformation appropriée pour effectuer la conversion vers le système de coordonnées de l'écran[1]. Cette transformation, appelée transformation de visualisation (*viewing transformation*), peut être choisie de telle sorte qu'elle applique la totalité de l'image à l'écran, ou qu'elle n'en rende visible qu'une partie, le reste étant éliminé par une routine de découpage.

D'une manière générale, donc, pour toute visualisation (d'une partie) de l'image, deux opérations préliminaires à l'affichage doivent être accomplies : le découpage et la transformation de visualisation :

[1]On peut également introduire un système de coordonnées intermédiaire, permettant de travailler sur le dispositif d'affichage, mais sans devoir en connaître sa hauteur et sa largeur en points : les coordonnées normalisées. Un tel système est requis lorsque l'image doit être mémorisée sous une représentation indépendante du dispositif de sortie, permettant par exemple d'afficher un dessin à l'écran, puis de l'envoyer au traceur sans devoir le recalculer.
La transformation de visualisation se fait alors en deux étapes : des coordonnées utilisateur aux coordonnées normalisées, permettant un stockage du dessin sous une forme indépendante du matériel, puis en coordonnées du dispositif d'affichage. Suite à la prépondérance des applications interactives, le stockage d'une image sous forme normalisée pour affichage ou traçage ultérieur ne présente plus guère d'intérêt : on préfère la reconstruire pour chaque opération d'affichage et éviter le passage par les coordonnées normalisées.

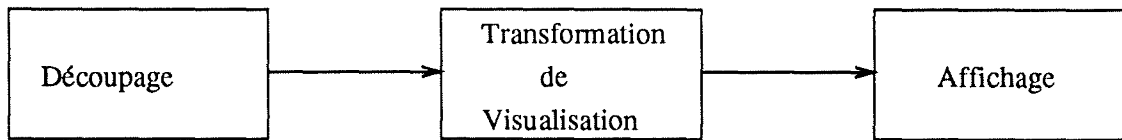

Figure 5.1: Opérations préliminaires à l'affichage

On a tout intérêt à d'abord effectuer l'opération de découpage (qui de toutes façons doit être faite) avant d'appliquer la transformation de fenêtrage : on ne perd alors pas de temps à appliquer cette dernière à des éléments de l'image totalement invisibles !

Une transformation de visualisation générale permet d'appliquer n'importe quel changement d'échelle, rotation ou translation à la définition de l'image dans les coordonnées de l'utilisateur. Cependant, il est rare que l'on doive appliquer des rotations. Une transformation de visualisation n'appliquant aucune rotation est appelée transformation de fenêtrage (*windowing transformation*). Ce type de transformation, moins général, est cependant suffisant dans 95 % des situations, et c'est le cas dans l'application qui nous occupe.

La tranformation de fenêtrage (voir [11], [12], [13] et [14]) est ainsi nommée parce qu'elle fait intervenir une fenêtre (*window*) dans l'espace des coordonnées utilisateur, qui entoure l'information que l'on veut afficher. Effectivement, la surface du dispositif d'affichage est limitée, alors que l'univers utilisateur est un plan infini. Il faut donc définir une région rectangulaire dans l'espace utilisateur, alignée avec les axes, telle que seules les parties des objets à l'intérieur de cette zone apparaîtront sur le dispositif d'affichage.

Outre la fenêtre dans le monde réel, on peut également définir une zone d'affichage (*viewport*), c'est-à-dire un rectangle dans l'écran où l'on veut que le contenu de la fenêtre soit affiché[2].

En résumé, on utilise la fenêtre pour définir *ce que* l'on veut afficher; on utilise la zone d'affichage pour spécifier *où* l'on veut que l'affichage se fasse à l'écran.

[2]La traduction des termes anglais varie d'un ouvrage français à l'autre : Schweizer [14] traduit *window* par *clôture* et *viewport* par *fenêtre*, alors que Lucas [12] choisit la traduction inverse : *fenêtre* pour *window* (ce qui paraît plus logique) et *clôture* pour *viewport* (ce qui ne nous semble pas un terme très heureux, et que nous avons préféré remplacer par *zone d'affichage*).

Normalement, il faut veiller à ce que la fenêtre et la zone d'affichage aient des rapports largeur/hauteur semblables, sinon l'image affichée est déformée.

La transformation que l'on applique à chaque point de la fenêtre en coordonnées utilisateur pour passer au point correspondant de la zone d'affichage en coordonnées écran est très simple, et se compose d'une translation, d'un changement d'échelle et d'une seconde translation :

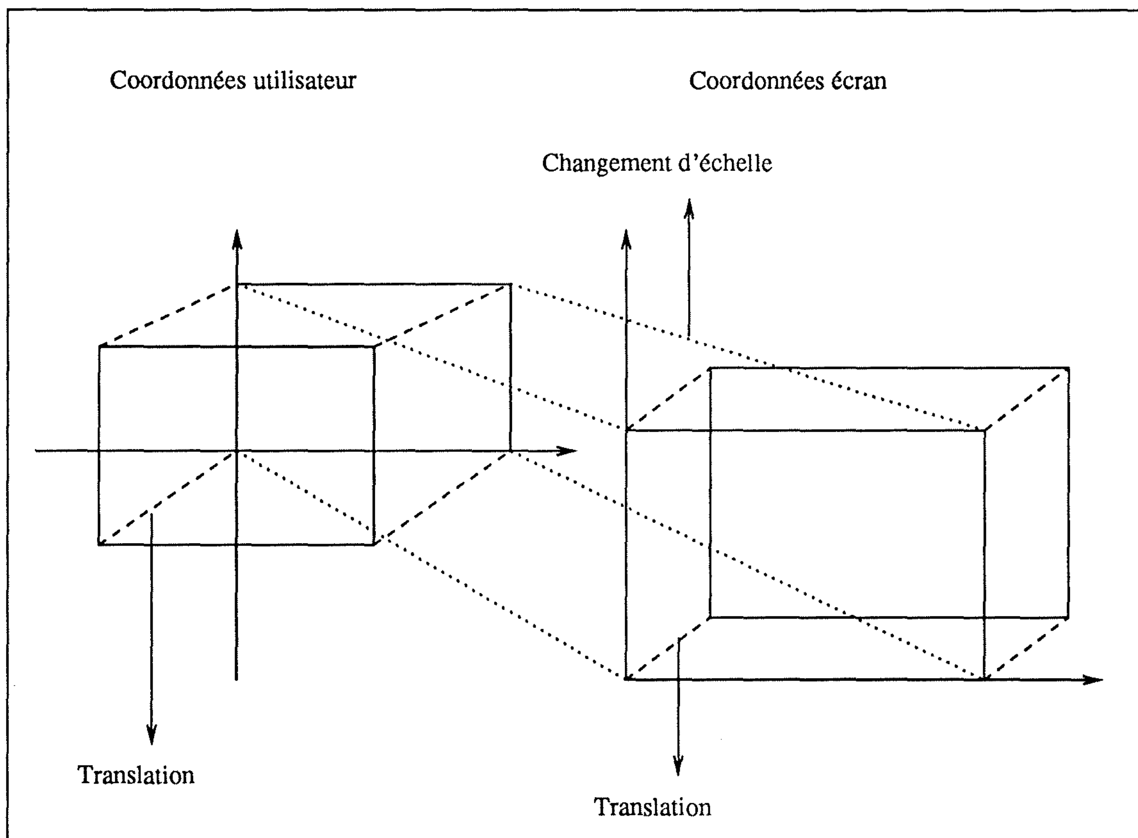

Figure 5.2: Passage des coordonnées utilisateur aux coordonnées écran

Supposons que les bords de la fenêtre soient donnés par $x = W_{xg}, x = W_{xd}, y = W_{yb}$ et $y = W_{yh}$ (relations exprimées dans les coordonnées utilisateur) et que les bords correspondants de la zone d'affichage soient donnés par $x = V_{xg}, x = V_{xd}, y = V_{yb}$ et $y = V_{yh}$ (relations exprimées dans les coordonnées écran). Dès lors, le point (x_u, y_u) en coordonnées utilisateur se transforme en le point (x_e, y_e) en coordonnées écran suivant :

$$x_e = \frac{V_{xd} - V_{xg}}{W_{xd} - W_{xg}}(x_u - W_{xg}) + V_{xg}$$

$$y_e = \frac{V_{yh} - V_{yb}}{W_{yh} - W_{yb}}(y_u - W_{yb}) + V_{yb}$$

La première translation $[-W_{xg}, -W_{yb}]$ correspond à ramener la fenêtre à l'origine du repère utilisateur, avant de faire le changement d'échelle. La seconde translation $[V_{xg}, V_{yb}]$ tient compte du décalage de la zone d'affichage par rapport à l'origine de l'écran.

Le système d'équations plus haut peut être simplifié en posant :

$$\begin{aligned} a &= \frac{V_{xd} - V_{xg}}{W_{xd} - W_{xg}} \\ b &= V_{xg} - aW_{xg} \\ c &= \frac{V_{yh} - V_{yb}}{W_{yh} - W_{yb}} \\ d &= V_{yb} - cW_{yb} \end{aligned}$$

ce qui donne :

$$\begin{aligned} x_e &= ax_u + b \\ y_e &= cy_u + d \end{aligned}$$

On s'arrange dès lors pour calculer les valeurs de a, b, c et d lorsque la fenêtre et la zone d'affichage sont définies, de telle sorte que l'on peut appliquer la transformation à chaque point par un calcul ne faisant intervenir que deux additions et deux multiplications.

La transformation inverse est très utile pour calculer les coordonnées utilisateur (du curseur graphique, par exemple) à partir des coordonnées écran :

$$\begin{aligned} x_u &= \frac{W_{xd} - W_{xg}}{V_{xd} - V_{xg}}(x_e - V_{xg}) + W_{xg} \\ y_u &= \frac{W_{yh} - W_{yb}}{V_{yh} - V_{yb}}(y_e - V_{yb}) + W_{yb} \end{aligned}$$

De même, en posant :

$$\begin{aligned} a' &= \frac{W_{xd} - W_{xg}}{V_{xd} - V_{xg}} \\ b' &= W_{xg} - a'V_{xg} \\ c' &= \frac{W_{yh} - W_{yb}}{V_{yh} - V_{yb}} \\ d' &= W_{yb} - c'V_{yb} \end{aligned}$$

on obtient :

$$\begin{aligned} x_u &= a'x_e + b' \\ y_u &= c'y_e + d' \end{aligned}$$

5.2 Découpage : algorithme de Cohen-Sutherland

Le découpage (*clipping*) consiste à ne considérer que la partie de l'objet qui sera représentée sur le dispositif d'affichage. Cette opération est indispensable si l'on ne désire pas afficher un objet dans son ensemble, mais dans une vue partielle, correspondant à l'agrandissement d'un détail. Mais le processus de sélection qui consiste à déterminer quelles sont les parties visibles de l'image n'est pas évident. Certaines lignes peuvent se trouver en partie dans la région visible de l'image, et en partie en-dehors. Ces lignes ne peuvent évidemment pas être négligées. Il ne suffit donc pas de simplement déterminer si les extrémités de la ligne se trouvent dans la partie visible.

La façon correcte de sélectionner l'information visible pour l'affichage est de diviser chaque élément de l'image en sa partie visible et sa partie invisible.

La base de ces opérations de découpage est une simple paire d'inégalités, qui détermine si un point (x, y) est visible :

$$\begin{aligned} x_{gauche} \leq & \; x \leq x_{droit} \\ y_{bas} \leq & \; y \leq y_{haut} \end{aligned}$$

Mais il serait tout-à-fait inefficace, pour découper une image, de convertir tous les éléments de l'image en points et d'utiliser ces inégalités[3].

Notons d'abord qu'un segment qui n'est que partiellement visible est divisé par les bords de l'écran en une ou deux portions invisibles, mais en un seul segment visible. Cela implique que le segment visible d'une droite peut être simplement déterminé en calculant ses deux extrémités.

L'algorithme de Cohen-Sutherland (voir [10], [11], [12], [13] et [14]), non seulement trouve rapidement les extrémités de la partie visible, mais accepte encore plus vite les segments qui sont totalement visibles. Il détecte (et rejette) tout aussi rapidement les segments qui sont situés entièrement d'un côté de la fenêtre. Cela en fait

[3]Dans le cas d'un point, le découpage est évident : il n'y a pas de partie visible et de partie invisible : un point est simplement visible ou invisible !

un très bon algorithme pour découper les figures qui sont beaucoup plus grandes que l'écran, ou, à l'inverse, qui sont presque toutes contenues dans la fenêtre.

L'algorithme est basé sur le principe que tout segment est soit entièrement dans la fenêtre, soit peut être divisé en deux, de telle sorte qu'une des deux parties puisse être trivialement rejetée. Il se décompose en deux parties :

1. On détermine d'abord si l'on ne se trouve pas dans un des deux cas triviaux :
    - le segment est tout entier à l'intérieur de la fenêtre,
    - le segment est tout entier à gauche, à droite, en-dessous ou au-dessus de la fenêtre.
2. Si le segment ne satisfait aucun de ces deux tests,
   Alors il est divisé en deux parties, et ces deux tests sont appliqués à chaque partie.

On recommence à appliquer le processus jusqu'à l'élimination complète des parties cachées et la découverte de l'éventuel segment tout entier contenu dans la fenêtre.

Le test de rejet/acceptation est implémenté de la manière suivante. On prolonge les bords de la fenêtre de telle manière que l'espace soit découpé en neuf régions (voir figure 5.3). Chacune de ces régions possède un code à 4 bits, dont l'interprétation est la suivante (les bits sont numérotés de la droite vers la gauche) :

- 1er bit : on se trouve à gauche du bord gauche de la fenêtre
- 2ème bit : on se trouve à droite du bord droit de la fenêtre
- 3ème bit : on se trouve en-dessous du bord inférieur de la fenêtre
- 4ème bit : on se trouve au-dessus du bord supérieur de la fenêtre

On assigne aux deux extrémités du segment le code correspondant à leur position.

Il est clair que si les codes des deux extrémités sont nuls, le segment se trouve entièrement dans la fenêtre, et est donc entièrement visible. On peut également facilement se convaincre que si l'intersection logique des deux codes est non nulle, c'est que le segment se trouve complètement en dehors de la fenêtre.

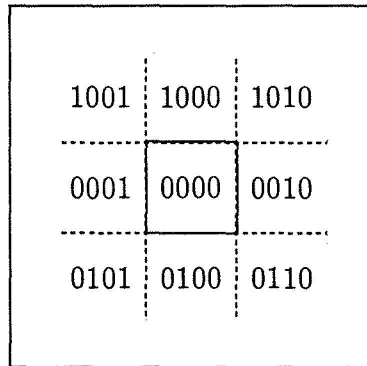

Figure 5.3: Codes de Cohen-Sutherland

Si le segment ne peut être éliminé par l'un de ces deux tests, il doit être subdivisé. Une méthode simple de subdivision est de trouver le point d'intersection du segment avec un bord de la fenêtre et de "laisser tomber" la partie qui se trouve en-dehors de la fenêtre. Il n'est pas évident de déterminer quelle division il faut d'abord faire, mais si le test de rejet est appliqué de manière itérative[4], l'ordre suivant lequel les subdivisions sont effectuées est sans importance sur le résultat final.

Nous avons adopté l'ordre suivant :

- Le bit 1 du code est non nul : intersection avec le bord supérieur de la fenêtre; la partie supérieure du segment est éliminée.
- Le bit 2 du code est non nul : intersection avec le bord inférieur de la fenêtre; la partie inférieure du segment est éliminée.
- Le bit 3 du code est non nul : intersection avec le bord droit de la fenêtre; la partie droite du segment est éliminée.
- Le bit 4 du code est non nul : intersection avec le bord gauche de la fenêtre; la partie gauche du segment est éliminée.

[4]Dans le cas le plus mauvais, il faut procéder à quatre itérations.

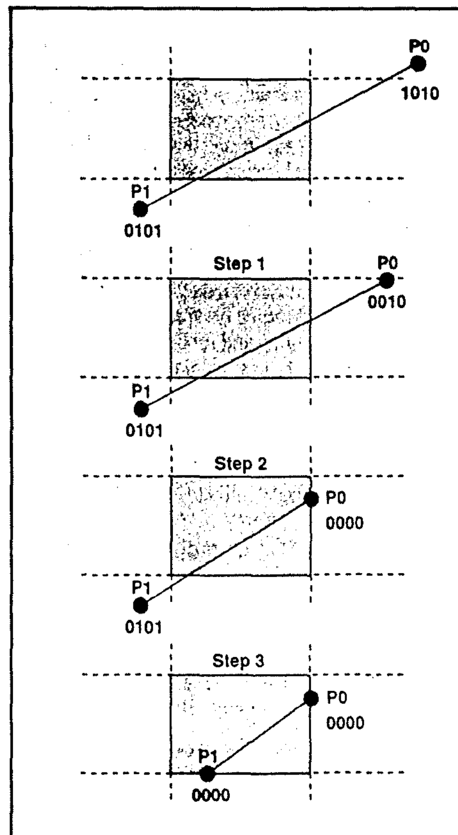

Figure 5.4: Exemple d'application de l'algorithme de Cohen-Sutherland

5.3 Détection des objets cliqués

5.3.1 Position du problème

Lors de l'utilisation du logiciel, il est fréquent que l'on ait à sélectionner un objet graphique (noeud, arc, élément du background) à l'aide de la souris, que ce soit pour consulter/modifier l'information qui lui est associée ou dans un but d'édition graphique (suppression, déplacement). C'est là un des gros problèmes qui s'est posé. En effet, si la plupart des langages graphiques (PHIGS, GKS) offrent à leurs utilisateurs la possibilité de référencer automatiquement un objet graphique rien qu'en cliquant à l'aide de la souris, notre décision de développer le logiciel à l'aide des seuls outils de bas niveau Xlib et Xt nous force à créer nous-mêmes cette fonctionnalité.

5.3.2 Solution adoptée

En fait, l'écran d'affichage peut être assimilé à une énorme grille de cellules élémentaires (*pixels*), chacune de ces cellules étant "allumée" dans une certaine couleur par les routines graphiques utilisées.

Dès lors, une idée qui vient automatiquement à l'esprit est de construire une matrice de mêmes dimensions que l'écran, chaque élément de cette matrice correspondant à un pixel et contenant un pointeur vers l'objet graphique qui "utilise" ce pixel.

Malheureusement, cette solution n'est pas applicable telle quelle : elle est beaucoup trop gourmande en mémoire. En effet, avec un écran de 1000 pixels sur 750, c'est une matrice de 750 000 cellules qu'il faudrait construire !

Il faut donc adapter cette solution en construisant une matrice beaucoup plus petite, et où chaque cellule correspond non plus à un pixel, mais à un ensemble de pixels. En utilisant une matrice de 200 cellules sur 150, chaque cellule correspond à un groupe de 5 pixels sur 5, et on réalise ainsi un gain de mémoire d'un facteur 25.

Plusieurs objets pouvant se superposer à l'intérieur d'une telle zone de 5 pixels sur 5, les cellules de la matrice contiennent non pas un pointeur vers un objet graphique, mais vers une liste d'objets graphiques. Chaque élément de cette liste mémorise le type de l'objet (nœud, arc, point intermédiaire d'un arc, élément de décor) et l'identificateur interne de cet élément.

5.3.3 Problème de la validation

Une fois que l'on a déterminé quel était l'objet cliqué, il faut être sûr qu'il s'agit bien de celui que voulait désigner l'utilisateur. Dans le cas d'un réseau "aéré", il y a peu de risques de confusion, mais si plusieurs objets graphiques sont proches les uns des autres, une confusion est toujours possible. Si dans le cas d'une simple consultation d'information, une telle erreur ne prête guère à conséquence, il n'en est pas de même s'il s'agit de la suppression de l'objet cliqué.

La solution adoptée dans le programme Xdrive n'est guère satisfaisante. Elle consiste à mettre graphiquement en évidence l'objet cliqué et à demander à l'utilisateur une confirmation explicite. Cette confirmation est particulièrement pénalisante, puisqu'elle intervient systématiquement, même dans le cas d'une simple consultation d'information. D'autre part, elle n'a pas beaucoup de sens dans les cas où la fenêtre de demande de confirmation cache justement la partie de l'écran où se trouve l'objet cliqué. Un autre problème se pose lorsque par malheur on a cliqué

"à côté" de l'objet : il faut accuser réception du message d'erreur et reprendre les opérations depuis le début.

La solution que nous avons adoptée est de distinguer les différents types d'événements possibles en ce qui concerne l'utilisation de la souris : presser le bouton, déplacer la souris et relâcher le bouton. Les deux premiers événements provoquent une détection de l'objet cliqué et sa mise en évidence graphique en temps réel. L'action de l'utilisateur de relâcher le bouton de la souris est alors interprété comme une validation de sa part. Dans le cas d'une suppression d'objet, une demande de confirmation explicite est maintenue.

5.3.4 Enregistrement des arcs : algorithme de Bresenham

Reste le problème de la maintenance de la matrice utilisée. Chaque fois qu'un objet graphique est ajouté ou supprimé, il faut mettre à jour les cellules concernées de la matrice. Le problème étant bien sûr de savoir quelles sont les cellules concernées...

Dans le cas du tracé d'un segment de droite, on utilise l'algorithme de Bresenham (voir [11], [12], [13] et [14]).

En fait, le problème se ramène à créer des segments de droite point par point. Un segment de droite sera défini par la donnée des coordonnées de ses extrémités (indices des cellules dans la matrice). Le problème consiste à engendrer la suite de points (cellules) situés sur la grille représentant l'écran, approchant le mieux le segment demandé (voir figure 5.5).

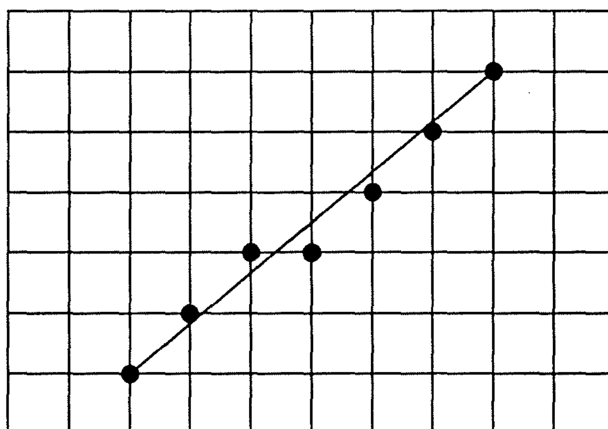

Figure 5.5: Tracé d'un segment : position du problème

Le problème est donc de choisir les points les plus proches du segment à tracer. Localement, le problème est donc : arrivé en un point donné, comment choisir le point suivant parmi les huit candidats possibles (voir figure 5.6).

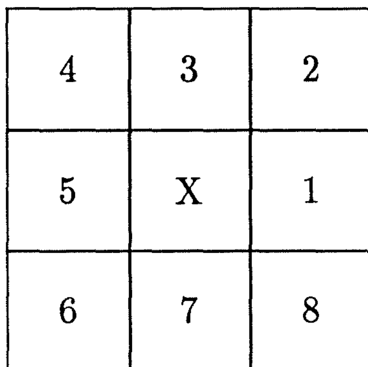

Figure 5.6: Voisinage d'un point

Bresenham propose pour les octants du plan (chiffres à l'intérieur) et les mouvements (chiffres extérieurs) la numérotation indiquée à la figure 5.7.

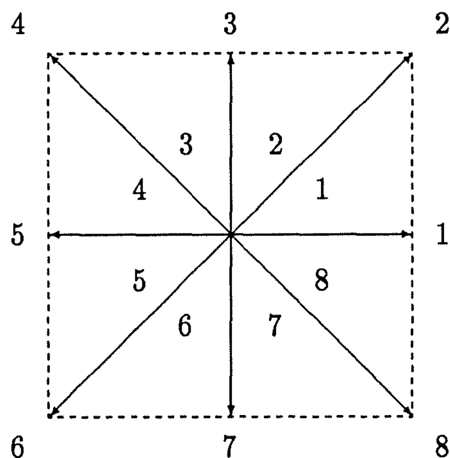

Figure 5.7: Octants du plan et mouvements possibles

L'octant 1 est défini de manière plus précise par l'ensemble des points (x, y) tels que :

$$x > y$$

$$y \geq 0$$

donc, le demi-axe est inclus, mais pas la diagonale.

De même pour l'octant 2 :

$$\begin{aligned} y &\geq x \\ x &> 0 \end{aligned}$$

donc la diagonale est incluse, mais pas le demi-axe.

Et ainsi de suite par rotation, ce qui conduit à un découpage symétrique du plan.

Relativement à son début pris comme origine, un segment est totalement inclus dans un octant, et donc le tracé du segment ne met en oeuvre que les deux mouvements correspondant à cet octant.

L'algorithme se divise alors en deux parties :

- détermination de l'octant,
- calcul de la séquence des mouvements.

Déterminons d'abord l'octant où est situé le segment. Soit :

$$\begin{aligned} \Delta x &= x_f - x_d \\ \Delta y &= y_f - y_d \end{aligned}$$

où (x_d, y_d) sont les coordonnées du début du segment, et (x_f, y_f) celles de la fin.

Dès lors, le segment appartient au 1er octant si et seulement si :

$$\begin{aligned} \Delta x &> \Delta y \\ \Delta y &> 0 \end{aligned}$$

On peut de même caractériser l'appartenance d'un segment à l'un des 7 autres octants.

Calculons maintenant la séquence des mouvements. Pour cela, supposons que le segment donné appartienne au 1er octant, et que le dernier mouvement ait conduit au point de coordonnées (r, q). Les deux candidats possibles pour l'étape suivante sont :

- soit le point $(r+1, q+1)$ correspondant au mouvement diagonal,
- soit le point $(r+1, q)$ correspondant au mouvement axial.

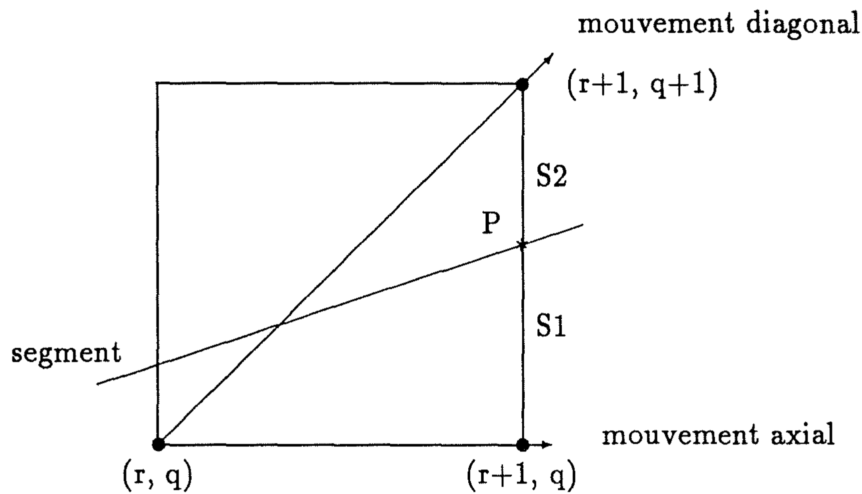

Figure 5.8: Choix du point suivant

Le choix entre ces possibilités est basé sur la comparaison des longueurs des segments S_1 et S_2 :

SI $S_1 < S_2$
ALORS mouvement axial,
SINON mouvement diagonal.

Or le point P de coordonnées $(r+1, y)$ appartient au segment à tracer. En posant :

$$\frac{y_f - y_d}{x_f - x_d} = \frac{b}{a}$$

on obtient :

$$y = \frac{b}{a}(r+1)$$

et donc :

$$\begin{aligned} S_1 &= y - q \\ &= \frac{b}{a}(r+1) - q \end{aligned}$$

$$\begin{aligned} S_2 &= (q+1) - y \\ &= (q+1) - \frac{b}{a}(r+1) \end{aligned}$$

Dès lors :

$$\begin{array}{rcl} S_1 & < & S_2 \\ & \Updownarrow & \text{puisque ici } a > 0 \\ 2(br - aq) + 2b - a & < & 0 \end{array}$$

Posons $S(r,q) = 2(br - aq) + 2b - a$
On a : $S(0,0) = 2b - a$
Dans le cas d'un mouvement axial, le point suivant est $(r+1,q)$, et :

$$\begin{array}{rcl} S(r+1,q) & = & 2[(b(r+1) - aq] + 2b - a \\ & = & S(r,q) + 2b \end{array}$$

De même dans le cas d'un mouvement diagonal :

$$\begin{array}{rcl} S(r+1,q+1) & = & 2[(b(r+1) - a(q+1)] + 2b - a \\ & = & S(r,q) + 2b - 2a \end{array}$$

D'où l'algorithme :

$S = 2b - a$
JUSQUA ce qu'on arrive en (x_f, y_f) *faire*
    SI $S < 0$
        ALORS mouvement axial
            $S = S + 2b$
        SINON mouvement diagonal
            $S = S + (2b - 2a)$
    FINSI
FINJUSQUA.

Cet algorithme est bien sûr applicable aux autres octants.

L'avantage de cet algorithme est qu'il n'utilise qu'une arithmétique entière. C'est un atout particulièrement important du point de vue de la rapidité d'exécution du programme.

5.3.5 Enregistrement des noeuds

En ce qui concerne les noeuds, on pourrait envisager des routines d'enregistrement spécifiques à chaque type de marqueurs[5]. Cependant, nous avons décidé, dans

[5] En ce qui concerne les cercles, on peut utiliser l'algorithme de Bresenham-Michener [11].

un premier temps, d'enregistrer les nœuds comme s'ils occupaient une surface carrée (ce qui est particulièrement simple à implémenter). Si à l'utilisation il s'avérait que ce choix pose problème, il serait toujours possible d'implémenter des routines d'enregistrement spécifiques.

Chapitre 6

C++

6.1 Quelques rappels concernant C++

La programmation orientée objet présente trois particularités majeures :

1. L'*encapsulation*, c'est-à-dire la capacité de réunir dans une même variable à la fois des données et du code.

2. L'*héritage*, qui permet de réutiliser le code des objets déjà créés dans de nouveaux objets.

3. Le *polymorphisme* ou capacité des objets de passer les uns pour les autres.

En C++, le modèle d'objet est appelé une classe, et les objets construits sur ce modèle des instances de cette classe.

A chaque classe d'objets sont donc associées un certain nombre de fonctions, qui participent à sa définition et lui sont propres. Ces fonctions sont généralement appelées méthodes. Elles acceptent ou non des paramètres et renvoient éventuellement des résultats. Les données internes et les fonctions associées sont appelées les membres de la classe. La modularité qui découle de cette encapsulation ne peut que faciliter la maintenance et le déboggage de ce type de programmation.

Parmi les fonctions membres d'une classe, il faut signaler les *constructeurs* et les *destructeurs*. Un constructeur est une fonction à laquelle il faut faire appel pour fabriquer une instance. Une classe peut avoir plusieurs constructeurs, en fonction du type des objets passés en arguments. Le destructeur sert à détruire un objet. Son appel peut être fait soit explicitement par le programmeur, soit automatiquement à la sortie du bloc.

Une classe peut être dérivée d'une autre classe. Par héritage, toutes les instances de la classe dérivée possèdent alors tous les membres de la classe sur laquelle est basée la classe dérivée : aussi bien les données que les définitions des méthodes. Les instances d'une classe possèdent également d'autres données et/ou d'autres méthodes que celles héritées, et c'est ce qui fait leur spécificité par rapport à la classe hiérarchiquement supérieure.

Les membres d'une classe (aussi bien les données que les fonctions) sont de l'un des trois types suivants :

- Membres privés.
  Il s'agit du statut par défaut. Les membres privés ne sont accessibles que par les fonctions membres de la même classe et les fonctions déclarées amies (*friend*).

- Membres publics.
  Ceux-ci sont accessibles par tous. Il s'agit en quelque sorte de l' "interface" de la classe.

- Membres protégés.
  Ces membres sont accessibles par les fonctions membres de la classe, ainsi que par celles qui sont membres de classes dérivées.

Dans la programmation par objets, on ne connaît pas toujours le type des données au moment de la compilation. C'est seulement lors de l'exécution que le type est déterminé par le contexte de l'application. Le polymorphisme permet de suivre ce comportement en sélectionnant, à l'exécution, de façon dynamique, la procédure à appeler.

6.2 Compilateur g++

En ce qui concerne le compilateur C++, notre choix s'est porté sur le compilateur g++ de GNU[1].

Il a l'avantage d'être largement répandu et facilement disponible : il s'agit d'un produit du domaine public. La librairie WWL (qui permet de considérer les différentes fenêtres de X Window comme des classes d'objets), que nous avions l'intention d'utiliser, a notamment été testée avec ce compilateur.

[1] Free Software Foundation, Cambridge, USA

Un autre avantage est d'être disponible sur PC.

Malheureusement, ce compilateur, lorsqu'il est utilisé sur des machines MIPS, a le désavantage de ne pas pouvoir générer l'information nécessaire à l'utilisation d'un *debugger* (*dbx*, par exemple).

Nous avons utilisé la version 1.37.1 de g++.

Lors de l'utilisation de ce compilateur, nous avons rencontré un comportement bizarre, pour lequel nous n'avons aucune explication. Nous ne parvenions pas à faire accepter au compilateur un pointeur vers une fonction comme paramètre. Toutefois, à force de tâtonnements, nous avons remarqué que si l'on ajoutait un second paramètre (un *int*, par exemple), cela fonctionnait. Tous les passages de fonctions comme paramètres s'accompagnent donc du passage d'un entier, qui n'a aucune signification : par exemple `list.scan (1,print);`. L'erreur a été signalée aux concepteurs du compilateur, mais sans réponse jusqu'à présent.

6.3 Compatibilité entre C++ et X Window

C'est lorsque nous avons essayé de compiler un programme C++ utilisant X Window que les problèmes ont commencé.

Nous nous sommes tout d'abord rendus compte qu'il était essentiel d'utiliser le Release 4 de la version 11 de X Window. Effectivement, le Release 3 ne contient pas les prototypes des fonctions, ce qui est primordial lorsque l'on travaille en C++. Or, la version officielle (DEC) dont nous disposons pour le moment est le Release 3.

Nous disposons également du Release 4 en version "domaine public", mais les problèmes n'ont pas tous été résolus pour autant. Effectivement, à la compilation, une erreur est détectée dans *Intrinsic.h* (fichier de X Window), lors de la définition du type *String*. En désespoir de cause, nous nous sommes adressés à Jean-Daniel Fekete (auteur de WWL), qui nous a indiqué que pour corriger cette erreur dans *Intrinsic.h*, il fallait appliquer le *patch* (fichier de corrections) 10 de X11R4. En fait, le Release 4 dont nous disposions était la version originale, et aucun fichier de corrections ne lui avait été appliqué. Nous avons essayé d'appliquer ces fichiers de corrections, mais, pour une raison inconnue, sans succès. Il aurait sans doute fallu reprendre l'installation de X Window à zéro.

Pressés par le temps, nous avons décidé de laisser ce problème en suspens et d'écrire la partie du programme utilisant X Window en C traditionnel.

6.4 Compatibilité entre C++ et C

Cette décision a alors posé le problème de l'interfaçage entre C et C++.

En C++, la procédure utilisée par le compilateur pour encoder de manière interne le nom d'une fonction diffère de celle utilisée par les compilateurs C traditionnels. Ces derniers encodent simplement le nom de la fonction, alors que les compilateurs C++ ajoutent au nom de la fonction des informations concernant leurs arguments, ainsi que le type retourné par la fonction. Il s'agit là d'une amélioration sensible en ce qui concerne la détection des erreurs. Examinons le programme suivant (repris de [16]) :

```
/* file mymath.c */
double mysqrt (double d)
{
/* do sqrt */
};

/* file myprog.c */
extern double mysqrt (int d);
double f;
int d = 55;
f = mysqrt (d);
```

Le compilateur C ne détectera aucune erreur. Effectivement, il n'a aucun moyen de savoir que la déclaration externe de *mysqrt()* a un type d'argument erroné (*int* au lieu de *double*). Dès lors, le *linker* établira un lien entre l'appel de fonction dans le fichier *myprog.c* avec la définition de la fonction dans le fichier *mymath.c*, puisque dans les deux cas, il voit le nom *mysqrt*. En conséquence, l'appel à la fonction *mysqrt()* retournera une réponse erronée, puisque la variable entière *d* n'aura pas été transformée en double.

Par contre, avec le compilateur C++, l'encodage des fonctions ressemblera respectivement à *dbl_mysqrt_dbl* et *dbl_mysqrt_int*. Dès lors, au moment du *linkage*, l'erreur sera détectée.

Mais cette procédure d'encodage du nom des fonctions pose un problème lorsque l'on veut appeler dans un programme C++, une fonction définie dans un programme C (ou l'inverse). Considérons l'exemple suivant :

```
/* file mymath.c */
```

```
double mysqrt (double d)
{
/* do sqrt */
};

// file myprog.cc
extern double mysqrt (double d);
double f;
int d = 55;
f = mysqrt (d);
```

Au moment de l'édition des liens, une erreur sera détectée, puisque le *linker* cherchera une fonction *dbl_mysqrt_dbl*, et ne trouvera qu'une fonction *mysqrt*.

Pour résoudre ce problème, il faut pouvoir demander au compilateur C++ d'encoder le nom de la fonction de manière traditionnelle. Cela se fait au moyen de la directive "C". Il faut donc écrire :

```
// file myprog.cc
extern "C" double mysqrt (double d);
double f;
int d = 55;
f = mysqrt (d);
```

Toutefois, nous avons encore rencontré un problème, qui semble spécifique à l'utilisation conjointe des compilateurs g++ et cc. Lorsqu'une fonction définie dans un programme compilé avec cc, acceptant comme paramètre un *int* et un *float*, est appelée à partir d'un programme compilé avec g++, le passage des paramètres est incorrectement réalisé. Si les paramètres sont soit un *int*, soit un *float*, soit deux *int*, soit deux *float*, il n'y a aucune difficulté : les problèmes surgissent lorsque l'on mélange ces deux types de paramètres. Mais si l'on utilise un *double* au lieu d'un *float*, il n'y a plus de problème. Nous avons donc remplacé tous les *float* par des *double*.

6.5 Les classes de base

6.5.1 Les tableaux dynamiques

La structure de données classique "array" présente quelques inconvénients :

1. La taille d'un tableau doit être une expression constante. Le programmeur, cependant, ne connaît pas toujours, au moment de la compilation, la taille nécessaire du tableau. Un type de tableau plus flexible devrait permettre au programmeur de spécifier (ou de modifier) la dimension du tableau au moment de l'exécution du programme.

2. Il est nécessaire de passer la taille du tableau à une fonction en même temps que le tableau lui-même. Des arguments de type tableau seraient plus faciles à utiliser si un tableau connaissait sa taille.

3. Il serait agréable de pouvoir réaliser des affectations de tableau à tableau.

4. Il n'y a pas de validation de l'index utilisé pour l'adressage des éléments du tableau.

Puisque nous travaillons en C++, il était intéressant de profiter de l'occasion pour définir une classe d'objets "tableaux dynamiques" (sans toutefois introduire de validation de l'index utilisé pour l'adressage des éléments du tableau).

D'une manière générale, l'implémentation de cette classe d'objets ne présente pas de difficulté majeure : seule la "fonction-opérateur" d'indiçage est délicate à implémenter parce qu'elle doit supporter des possibilités à la fois de lecture et d'écriture. La partie lecture est simple : accepter la valeur de l'index et retourner l'élément approprié. La partie écriture n'est pas si simple. Pour que l'expression *monTableau[i]* puisse apparaître à la gauche de l'opérateur d'assignation, il faut que cette expression puisse être évaluée comme une *lvalue*[2]. Cela est obtenu en spécifiant la valeur retournée comme une référence (une référence sert d'alias pour une autre variable).

Malheureusement, pour que cet opérateur d'indiçage fonctionne correctement, le type des éléments contenus dans le tableau doit être connu. Il ne nous a donc pas été possible de construire une classe générique de tableaux dynamiques, dont auraient pu hériter les tableaux dont nous avions besoin (tableaux de nœuds, d'arcs, de points intermédiaires). Nous avons donc dû, à chaque fois, redéfinir la classe à partir de rien.

[2]Une "valeur à gauche" (*lvalue* ou *left value*) est une expression à laquelle on peut affecter une valeur. La *lvalue* la plus simple prend la forme d'une variable (identificateur). Une *lvalue* est requise du côté gauche d'un opérateur d'affectation. En d'autres mots, une *lvalue* contient une adresse.

6.5.2 Les listes

La notion de liste est bien connue, et ne présente pas de difficulté d'implémentation sous forme de classe d'objets.

Notons une fonction particulièrement intéressante, *scan*, qui, acceptant comme paramètre un pointeur vers une fonction, applique cette fonction à tous les éléments de la liste.

6.5.3 Les ensembles

Si l'on veut être efficace, la classe *list* est insuffisante pour les fonctionnalités concernant les ensembles de nœuds et d'arcs. Effectivement, si l'utilisateur sélectionne tous les nœuds (par exemple), il faut d'abord construire une liste reprenant tous les nœuds. Puis, lorsque l'utilisateur demandera de changer la taille des marqueurs représentant ces nœuds (par exemple), il faudra balayer toute la liste pour appliquer le changement demandé. Dans un tel cas, il serait plus efficace de mémoriser le fait que tous les nœuds sont sélectionnés, et, lorsque le changement est demandé, de travailler directement sur le tableau de nœuds.

Nous avons donc introduit la classe *set*, constituée d'une liste (c'est-à-dire un pointeur) et d'un entier (type énuméré) qui indique le type de sélection opéré :

- NONE : aucune sélection n'a encore été faite.
- SOME : une partie des éléments seulement ont été sélectionnés : on travaille avec la liste.
- ALL : tous les éléments ont été sélectionnés : on travaille avec le tableau.

6.6 Les classes *graphiques*

6.6.1 La classe *World*

Il n'existe qu'un objet de cette classe, *theWorld*, qui modélise le système de coordonnées de l'utilisateur.

Membres privés

Chaque objet contient quatre données privées, *xMin, xMax, yMin* et *yMax*, qui correspondent aux limites à l'intérieur desquelles se trouve la carte routière.

Constructeur

Un seul constructeur, à partir de 4 réels.

Membres publics

Quatre fonctions élémentaires, donnant accès aux données internes.

La fonction *resize* (arguments : 4 réels) permet de redimensionner l'objet.

6.6.2 La classe *Disp*

Il n'existe qu'un objet de cette classe, *theDisp*, qui modélise le dispositif d'affichage.

Membres privés

Les coordonnées en x et y sont supposées commencer à zéro. Chaque objet contient donc deux données privées entières, *xMax* et *yMax*, qui correspondent aux coordonnées maximales du dispositif d'affichage.

Constructeur

Un seul constructeur, sans paramètre, qui n'initialise pas *xMax* et *yMax*.

Membres publics

Deux fonctions élémentaires, donnant accès aux données internes.

Une fonction *resize* (arguments : deux entiers) qui permet de redimensionner l'objet. Etant donné que le constructeur n'initialise pas *xMax* et *yMax*, cette fonction doit absolument être appelée avant de pouvoir utiliser l'objet.

6.6.3 La classe *View*

Dans l'état actuel du projet, cette classe ne compte qu'un objet, *theView*, mais il est envisagé de pouvoir en définir plusieurs, afin de pouvoir visualiser simultanément plusieurs parties du réseau routier, ou la même partie sous des hypothèses différentes.

Chaque objet correspond à la visualisation sur écran d'une partie de l'image. Plus précisément, il correspond donc à une fenêtre (système de coordonnées de l'utilisateur) et à une zone d'affichage (système de coordonnées de l'écran).

Membres privés

Chaque objet contient quatre données réelles (*wLeft, wRight, wBottom* et *wTop*), qui correspondent aux limites de la fenêtre, et quatre données entières (*vLeft, vRight, vBottom* et *vTop*), qui correspondent aux limites de la zone d'affichage.

Quatre données entières supplémentaires (*vL, vR, vB* et *vT*) correspondent aux limites à l'intérieur desquelles l'affichage aura réellement lieu : ce sont elles qui seront utilisées dans les calculs de la transformation de fenêtrage. On ne peut prendre *wLeft, wRight, wBottom* et *wTop* sous peine de déformer l'image affichée.

C'est la fonction privée *adapt* qui calcule *vL, vR, vB* et *vT* de telle sorte que :

$$\frac{vR - vL}{vT - vB} = \frac{wRight - wLeft}{wTop - wBottom}$$

Le membre *scale* mémorise le rapport d'agrandissement, c'est-à-dire le rapport entre la zone affichée et la zone affichable (totalité de l'image).

Constructeur

Un seul constructeur, sans paramètre, qui n'initialise pas les membres de l'objet.

Membres publics

Neuf fonctions élémentaires permettent d'avoir accès aux données internes *wLeft, wRight, wBottom, wTop, vLeft, vRight, vBottom, vTop* et *scale*.

Les fonctions *resizeWindow* et *resizeViewport* permettent de redimensionner respectivement la fenêtre et la zone d'affichage. Chacune de ces deux fonctions fait appel à *adapt*, afin de mettre à jour *vL, vR, vB* et *vT*. Etant donné que le constructeur ne procède à aucune initialisation, ces deux fonctions doivent absolument être appelées avant de pouvoir utiliser l'objet.

La fonction *resetWindow* réinitialise la zone d'affichage à la totalité de l'image.

Les fonctions *horRecenter* et *vertRecenter* permettent de recentrer horizontalement et verticalement la zone d'affichage. L'argument est exprimé en pourcentage de la zone affichée. Par contre, le recentrage obtenu par la fonction *recenter* est absolu : ses arguments sont les coordonnées du nouveau centre de la zone d'affichage.

Les fonctions *horTranslate* et *vertTranslate* permettent de déplacer horizontalement et verticalement la zone d'affichage. L'argument est exprimé en pourcentage de la zone affichée.

La fonction *zoom* permet d'agrandir ou de réduire la zone d'affichage.

6.6.4 La classe *WorldPoint*

Cette classe modélise les points dans le système de coordonnées de l'utilisateur.

Membres privés

Chaque objet contient bien sûr les deux coordonnées réelles du point.

La classe contient également quatre données réelles déclarées ***static***[3], correspondant aux valeurs a', b', c' et d' nécessaires pour l'application de la transformation de fenêtrage inverse.

Constructeurs

Un constructeur sans paramètre (qui ne fait rien) est nécessaire pour certaines déclarations de variables du type *WorldPoint*.

Un constructeur avec deux réels en entrée initialise simplement les coordonnées du point avec ces paramètres.

Un constructeur acceptant un *DispPoint*[4] en entrée procède à l'initialisation des coordonnées par application de la transformation de fenêtrage inverse.

Membres publics

Les fonctions *redefX* et *redefY* permettent de modifier les coordonnées du point.

La fonction *isIn* retourne un booléen qui indique si le point se trouve ou non à l'intérieur de la fenêtre définie dans *theView*.

La fonction *code* retourne un entier qui représente la position du point par rapport à la fenêtre (cfr. algorithme de Cohen-Sutherland).

Deux fonctions élémentaires, *getX* et *getY* permettent d'avoir accès aux données internes x et y.

La fonction statique *setABCD* calcule les valeurs des paramètres a', b', c' et d' de la transformation de fenêtrage inverse.

---

[3]Un membre déclaré *static* n'est pas stocké à l'intérieur de chaque objet, mais à part de ceux-ci. Il a donc la même valeur pour tous les objets de la classe. Une fonction déclarée *static* peut accéder à ce membre (aussi bien en lecture qu'en écriture) sans qu'un seul objet de la classe n'existe.

[4]Voir section suivante : il s'agit simplement d'un point défini dans le système de coordonnées de l'écran.

6.6.5 La classe *DispPoint*

Cette classe modélise les points dans le système de coordonnées de l'écran.

Membres privés

Chaque objet contient bien sûr les deux coordonnées entières du point.

La classe contient également quatre données réelles déclarées *static*, correspondant aux valeurs a, b, c et d nécessaires pour l'application de la transformation de fenêtrage.

Constructeurs

Un constructeur sans paramètre (qui ne fait rien) est nécessaire pour certaines déclarations de variables du type *DispPoint*.

Un constructeur avec deux entiers en entrée initialise simplement les coordonnées du point avec ces paramètres.

Un constructeur acceptant un *WorldPoint* en entrée procède à l'initialisation des coordonnées par application de la transformation de fenêtrage.

Membres publics

Deux fonctions élémentaires permettent d'avoir accès aux coordonnées du point.

La fonction statique *setABCD* calcule les valeurs des paramètres a, b, c et d de la transformation de fenêtrage.

6.6.6 La classe *WorldLine*

Cette classe modélise les segments dans le système de coordonnées du l'utilisateur.

Membres privés

Chaque objet contient deux *WorldPoint*, correspondant aux deux extrémités du segment.

La fonction *swap*, notamment utilisée par la fonction publique *clip*, a pour effet d'échanger les deux extrémités du segment.

Constructeur

Un constructeur sans paramètre (qui ne fait rien) est nécessaire pour certaines déclarations de variables du type *DispPoint*.

Un constructeur évident acceptant deux *WorldPoint* en entrée.

Membres publics

Deux fonctions élémentaires permettent d'avoir accès aux deux extrémités du segment.

La fonction *clip* retourne un booléen qui indique si oui ou non au moins une partie du segment est visible. Dans l'affirmative, les deux extrémités du segment ont éventuellement été modifiées de telle sorte que le segment ainsi obtenu correspond à la partie visible du segment initial.

6.6.7 La classe *DispLine*

Cette classe modélise les segments dans le système de coordonnées de l'écran.

Membres privés

Chaque objet contient deux *DispPoint*, correspondant aux deux extrémités du segment, ainsi que trois entiers représentant respectivement le style de ligne (type énuméré : continu, tirets ou pointillés), l'épaisseur du trait et la couleur (index dans la palette des couleurs disponibles).

Constructeurs

Un constructeur sans paramètre, qui procède à des initialisations par défaut.

Un constructeur évident acceptant deux *DispPoint* en entrée.

Il existe également un constructeur acceptant une *WorldLine* en entrée. On suppose que cette *WorldLine* a préalablement été découpée de telle sorte qu'elle est entièrement à l'intérieur de la fenêtre, c'est-à-dire entièrement visible.

Membres publics

La fonction *draw* permet de tracer le segment à l'écran.

6.6.8 La classe *WorldPolyLine*

Cette classe modélise une ligne brisée (suite de segments contigus) dans le système de coordonnées de l'utilisateur.

Membres privés

Cette classe se présente en fait comme un tableau dynamique de *WorldPoint*. Chaque objet contient donc un entier, *size*, qui représente le nombre d'éléments du tableau, et un pointeur vers le type *WorldPoint*.

Constructeurs

Un constructeur sans paramètre, qui initialise le tableau à zéro élément.

Membres publics

Une fonction qui permet d'obtenir la taille du tableau.

Une fonction qui permet de redimensionner le tableau.

Un opérateur d'indiçage, qui permet l'accès à un élément du tableau, aussi bien en lecture qu'en écriture.

6.6.9 La classe *DispPolyLine*

Cette classe modélise une ligne brisée (suite de segments non nécessairement contigus) dans le système de coordonnées de l'écran.

Membres privés

Cette classe se présente en fait comme un tableau dynamique de *DispLine*. Chaque objet contient donc un entier, *size*, qui représente le nombre d'éléments du tableau, et un pointeur vers le type *DispLine*.

Constructeurs

Un constructeur sans paramètre, qui initialise le tableau à zéro élément.

Un constructeur qui admet comme paramètre une *WorldPolyLine*. Pour construire chaque élément du tableau *DispPolyLine*, on procède au découpage[5] du segment correspondant dans le paramètre *WorldPolyLine*.

[5] Par application de l'algorithme de Cohen-Sutherland.

Membres publics

Une fonction qui permet d'obtenir la taille du tableau.

Une fonction qui permet de redimensionner le tableau.

Un opérateur d'indiçage, qui permet l'accès à un élément du tableau, aussi bien en lecture qu'en écriture.

La fonction *draw* permet de tracer la ligne brisée à l'écran.

6.6.10 Les classes *Disk, Cross, X, Square, Dot* et *Asterisk*

Ces classes constituent les différents types de marqueurs utilisés pour représenter les nœuds.

Membres privés

Chaque objet de chacune de ces classes contient un *DispPoint*, qui représente sa localisation, et un entier, qui représente sa couleur.

Chaque objet de chacune de ces classes, à l'exception de *Dot*, contient également un entier qui représente la taille de l'objet.

Chaque objet des classes *Cross, X* et *Asterisk* possède en outre un entier qui représente l'épaisseur du trait utilisé pour tracer l'objet.

Constructeurs

Chaque classe admet un constructeur évident, qui permet d'initialiser tous les membres privés.

Membres publics

La fonction *draw* permet d'afficher chaque objet à l'écran.

6.7 Les classes relatives à la structure de graphe

6.7.1 La classe *GNode*

Cette classe modélise la représentation graphique des nœuds d'un graphe. Elle est basée sur la classe *WorldPoint*.

Membres privés

Chaque objet contient deux identificateurs : *idNode*, qui est l'identificateur du modèle sous-jacent, et *number*, qui est un identificateur interne à l'éditeur développé. Il s'agit en fait simplement de l'indice du nœud dans le tableau qui représente l'ensemble des nœuds du graphe. Cet identificateur interne est utilisé pour accélérer les opérations d'accès.

Chaque objet contient également deux listes d'entiers, *origin* et *dest*, qui sont les listes des arcs (identificateurs internes : voir la classe *GLink*) dont le nœud est origine ou extrémité.

Enfin, chaque objet contient des attributs graphiques : couleur, taille, type de marqueur utilisé pour la représentation, booléen indiquant la visibilité ou non du nœud.

Constructeurs

Un constructeur sans paramètre (qui ne fait rien) est nécessaire pour certaines déclarations de variables de type *GNode*.

Membres publics

Six fonctions élémentaires permettent d'avoir accès aux données internes de l'objet : ce sont *getId, getNumber, getColor, getSize, getMarker* et *getVisibility*.

Six autres fonctions élémentaires permettent de fixer la valeur de ces mêmes données internes. Il s'agit de *set, setNumber, setColor, setSize, setMarker* et *setVisibility*.

La fonction *draw* permet d'afficher le nœud à l'écran, alors que la fonction *highlight* permet de mettre le nœud en évidence. Il est à noter que cette dernière fonction travaille en mode *XOR* et que de ce fait, un deuxième appel de cette fonction a pour effet de restituer au nœud son affichage normal.

La fonction *deleteConnexion* a pour effet de supprimer tous les arcs qui ont ce nœud pour extrémité (origine ou destination).

Les fonctions *addOrigin* et *addDestination* ont pour effet d'ajouter un arc à la liste des arcs dont le nœud est une extrémité (origine ou destination). Les fonctions *eraseOrigin* et *eraseDestination* ont pour effet de supprimer de ces mêmes listes l'arc spécifié en argument.

La fonction *move* a pour effet de déplacer un nœud. Elle ne se contente pas de mettre à jour la position du nœud, mais modifie également tous les arcs concernés

par ce changement de position (cfr. classe *GLink*).

6.7.2 La classe *GLink*

Cette classe modélise la représentation graphique des arcs d'un graphe.

Membres privés

Chaque objet contient deux identificateurs : *idLink*, qui est l'identificateur du modèle sous-jacent, et *number*, qui est un identificateur interne à l'éditeur développé. Il s'agit en fait simplement de l'indice de l'arc dans le tableau qui représente l'ensemble des arcs du graphe. Cet identificateur interne est utilisé pour accélérer les opérations d'accès.

Chaque objet contient également deux entiers, *Or* et *Ext*, qui sont les identificateurs internes des deux nœuds extrémités de l'arc. Un booléen, *uniqueSens*, indique en outre si l'arc est à sens unique ou pas.

Chaque objet contient aussi des attributs graphiques : couleur, épaisseur du trait, type de trait (continu, tirets ou pointillés), booléen indiquant la visibilité ou non de l'arc.

Enfin, chaque objet contient *interPoints*, une *WorldPolyLine*, qui est la description du tracé de l'arc dans les coordonnées de l'utilisateur.

Constructeurs

Un constructeur sans paramètre (qui ne fait rien) est nécessaire pour certaines déclarations de variables de type *GLink*.

Membres publics

Neuf fonctions élémentaires permettent d'avoir accès aux données internes de l'objet : ce sont ***getId, getNumber, getOr, getEx, getUniqueSens, getColor, getWidth, getLineStyle*** et ***getVisibility***.

Sept autres fonctions élémentaires permettent de fixer la valeur de ces mêmes données internes. Il s'agit de ***set, setNumber, setColor, setWidth, setStyle, setUniqueSens*** et ***setVisibility***.

La fonction *draw* permet de tracer l'arc à l'écran, alors que la fonction ***highlight*** permet de mettre cet arc en évidence. Comme pour les nœuds, cette dernière fonction travaille en mode *XOR*.

Enfin, une série de fonctions permettent de gérer les points intermédiaires du tracé de l'arc. *indicatePoints* permet d'afficher explicitement tous les points intermédiaires, alors que *highlightPoint* permet de mettre le point spécifié en évidence. *removePoint* et *removePoints* permettent de supprimer soit le point intermédiaire spécifié, soit tous les points intermédiaires. *movePoint* permet de déplacer le point intermédiaire spécifié. Enfin, *addPoint* permet d'introduire un point intermédiaire supplémentaire dans la description de l'arc.

6.7.3 La classe *GNodesArray*

Cette classe modélise l'ensemble des nœuds du graphe.

Membres privés

Cette classe se présente en fait comme un tableau dynamique de *GNode*. Chaque objet contient donc un entier, *size*, qui représente le nombre d'éléments du tableau, et un pointeur vers le type *GNode*.

Constructeur

Un constructeur sans paramètre, qui initialise le tableau à zéro élément.

Membres publics

Une fonction qui permet d'obtenir la taille du tableau.

Un opérateur d'indiçage, qui permet l'accès à un élément du tableau, aussi bien en lecture qu'en écriture.

Trois fonctions permettent de modifier la taille du tableau : avec *resize*, on perd le contenu du tableau, tandis que *shrink* et *extend* permettent respectivement de réduire ou d'augmenter la taille du tableau, mais sans perte du contenu.

La fonction *existNode* accepte en entrée un identificateur de nœud du modèle sous-jacent. Elle retourne 0 si le nœud n'existe pas dans le tableau; s'il existe, elle retourne l'identificateur interne du nœud.

La fonction *accessNode* permet d'accéder (par un pointeur) à un nœud spécifié par son identificateur interne.

La fonction *remove* permet de supprimer un nœud. En fait, le nœud ne disparaît pas du tableau. Plus simplement, son identificateur interne est mis à zéro. Cette

façon de procéder offre la possibilité d'une éventuelle récupération par une fonction *undelete* qui reste à implémenter.

La fonction *create*, qui accepte en entrée un nom de fichier, permet de créer l'ensemble des nœuds à partir d'un fichier. En retour, cette fonction permet de connaître l'extension spatiale du graphe ainsi créé.

La fonction *draw* provoque l'affichage de tous les nœuds du graphe en "répercutant" cet ordre à chaque élément du tableau.

6.7.4 La classe *GLinksArray*

Cette classe modélise l'ensemble des arcs du graphe.

Membres privés

Cette classe se présente comme un tableau dynamique de *GLink*. Chaque objet contient donc un entier, *size*, qui représente le nombre d'éléments du tableau, et un pointeur vers le type *GLink*.

Constructeur

Un constructeur sans paramètre, qui initialise le tableau à zéro élément.

Membres publics

Une fonction qui permet d'obtenir la taille du tableau.

Un opérateur d'indiçage tout aussi traditionnel.

De même que pour le tableau de nœuds, trois fonctions permettent de redimensionner le tableau : *resize, shrink* et *extend*.

Les fonctions *remove, create* et *draw* fonctionnent de la même manière que pour les nœuds.

6.7.5 La classe *GGraph*

Cette classe modélise le graphe, c'est-à-dire les nœuds et les arcs.

Membres privés

Chaque objet (en fait, il n'en existe qu'un : *theGraph*) contient un *GNodesArray* : N et un *GLinksArray* : L.

Constructeur

Un constructeur sans paramètre, qui appelle les constructeurs de N et de L, et initialise donc les deux tableaux correspondant à zéro.

Membres publics

N et L n'étant pas directement accessibles, beaucoup des fonctions de la classe *GGraph* servent simplement d'intermédiaires pour accéder aux fonctions correspondantes de N et de L. C'est le cas pour *shrinkNodes, removeNode, accessNode, getNodesNumber, existNode* et *moveNode* en ce qui concerne les nœuds; pour *shrinkLink, removeLink, accessLink, getLinksNumber* et *existLink* en ce qui concerne les arcs; pour *removePointsLink, removePoint, indicatePointsLink, movePoint, highlightPoint* et *getPoint* en ce qui concerne les points intermédiaires utilisés dans la description des arcs.

La fonction *create* appelle successivement les fonctions correspondantes de N et de L, et s'occupe également d'initialiser correctement *theWorld* et *theView.*

La fonction *draw*, avant d'appeler les fonctions correspondantes de N et de L, procède à l'effacement de l'affichage et à la remise à zéro de *theImage* (voir section suivante : les classes relatives à la détection des objets).

Les fonctions *addLink* et *addNode* permettent d'ajouter au graphe respectivement un arc et un nœud.

6.8 Les classes relatives à la détection des objets

6.8.1 La classe *objectItem*

Cette classe, exclusivement privée, décrit les objets qui constituent les listes *objectList.*

Membres privés

Outre un pointeur vers un objet de même type (concept de liste oblige), chaque objet contient deux entiers : *type* indique de quel type d'objet il s'agit (nœud, arc, élément de décor, point intermédiaire) et *id* est un identificateur de l'objet.

Constructeurs

Un constructeur sans paramètre initialise à "vide", alors qu'un constructeur avec deux entiers en entrée initialise *type* et *id*. Dans les deux cas, le pointeur *next* est mis à NULL.

6.8.2 La classe *objectList*

Cette classe représente des listes d'*objectItem*.

Membres privés

Chaque objet se résume à un pointeur qui vaut NULL si la liste est vide, ou bien pointe vers un *objectItem*, premier élément de la liste.

Constructeurs

Un constructeur sans paramètre permet d'initialiser une liste vide.

Un constructeur avec deux entiers en entrée initialise une liste à un élément, construit à partir des deux paramètres.

Membres publics

La fonction booléenne *isEmpty* indique si la liste est vide ou pas.

La fonction *insert* permet d'insérer en début de liste un nouvel élément.

La fonction *removeAll* remet toute la liste "à zéro", alors que *remove* supprime de la liste l'élément de type et identificateur spécifiés en arguments.

La fonction *retrieve* retourne un pointeur vers l'identificateur du premier élément de la liste dont le type correspond à celui spécifié en entrée.

6.8.3 La classe *image*

Cette classe est à la base de la détection des objets cliqués. Elle mémorise tous les objets dessinés à l'écran. Un seul objet de cette classe est utilisé : *theImage*.

Membres privés

Cette classe se présente en fait comme une matrice 200x150 d'éléments de type *objectList* (donc comme une matrice de pointeurs).

Chaque cellule de la matrice correspondant sur l'écran à une région de 5 pixels sur 5 pixels, la fonction privée *conversion* calcule les indices x et y de la cellule de la matrice qui correspond au *DispPoint* donné en argument.

Constructeur

Un constructeur sans paramètre (qui ne fait rien) est nécessaire pour la déclaration de *theImage*.

Membres publics

La fonction *clear* remet à NULL tous les pointeurs contenus dans la matrice, en libérant éventuellement la place mémoire utilisée par le mécanisme d'enregistrement des objets.

La fonction *recordLine* accepte en entrée une *DispLine* et deux entiers (type et identificateur de l'objet). En utilisant l'algorithme de Bresenham, elle met à jour toutes les listes correspondant aux cellules "traversées" par le segment.

La fonction *recordBox* accepte en entrée deux *DispPoint* et deux entiers (type et identificateur de l'objet). Les deux points passés en arguments correspondent respectivement aux coins supérieur gauche et inférieur droit d'un rectangle contenant l'objet à enregistrer. La fonction met à jour toutes les listes correspondant aux cellules "recouvertes" par ce rectangle.

Les fonctions *eraseLine* et *eraseBox*, acceptant respectivement les mêmes arguments que *recordLine* et *recordBox*, effectuent les mises à jour en sens contraire : les objets sont supprimés des listes correspondant aux cellules concernées. Ces deux fonctions sont implémentées, mais dans la pratique, on n'a pas eu à s'en servir. Effectivement, elles devraient être utilisées lorsque l'on supprime un nœud ou un arc. Or, à ce moment, on redessine le graphe (appel à *theGraph.redraw*), ce qui implique non seulement que l'on efface l'écran d'affichage, mais également que l'on réinitialise *theImage* à zéro.

Chapitre 7

Ce qui reste à faire

7.1 Améliorer l'esthétique interne et l'efficacité du programme

Beaucoup reste à faire en ce qui concerne l'efficacité du programme. C'est un point particulièrement important dans l'optique de l'utilisation de l'éditeur sur PC : une machine moins rapide peut rendre critique l'utilisation de certaines fonctionnalités.

En ce qui concerne les tableaux dynamiques de noeuds et d'arcs, on peut songer à les garder constamment triés sur l'identificateur du modèle, et utiliser une procédure de recherche dichotomique pour l'accès à un noeud ou un arc particulier.

Dans de nombreux cas, il y a des appels redondants à la routine de dessin du graphe. Par exemple, lorsque l'on change le type de marqueurs pour un ou plusieurs nœuds, la méthode *theGraph.redraw ()* est appelée à deux reprises : une fois parce qu'il y a eu modification du graphe; une fois parce que la disparition de la fenêtre proposant les différents types de marqueurs disponibles provoque un événement *Exposure*, qui a pour effet de faire redessiner le graphe.

Et l'on pourrait encore citer bien d'autres situations où l'efficacité du programme pourrait être améliorée.

D'autre part, les avantages du C++ n'ont pas tous été exploités. Si l'encapsulation a bien été exploitée, il n'en est pas de même de l'héritage. Les différents tableaux dynamiques (tableaux de nœuds, d'arcs et de points intermédiaires), par exemple, n'ont pas de classe de base commune, alors que nombre de leurs méthodes sont communes. Le problème du pointeur vers des données de types différents devrait pouvoir trouver une solution. Toutes les listes d'éléments réduits à un

entier ont pour classe de base commune la classe *IntList*, mais les listes d'éléments constitués de deux entiers (les listes utilisées dans la classe *image*) n'utilisent pas cette classe de base. Cela devrait pouvoir être amélioré.

7.2 Problèmes spécifiques au g++

Il nous faut résoudre les problèmes spécifiques rencontrés dans l'utilisation du compilateur g++.

Le problème du passage simultané de paramètres *float* et *int* n'en est pas vraiment un. D'une part, il a été résolu par l'utilisation du type *double* au lieu du type *float*. D'autre part, ce problème n'apparaît en fait que dans le cas où l'on utilise simultanément les compilateurs g++ et cc. Dans la mesure où toute l'application est portée en C++ (voir section 5.3), le problème n'existe plus[1].

En ce qui concerne la difficulté de passage de fonctions en paramètres, il faut approfondir le problème. Il s'agit là d'un comportement pour le moins bizarre, qui doit trouver une solution, peut-être dans une version ultérieure de g++[2].

Une autre solution, beaucoup plus radicale, pourrait être d'envisager l'utilisation d'un autre compilateur C++.

7.3 Tout porter en C++

Convaincus par les avantages de C++, il nous reste bien sûr à porter toute l'application dans ce langage. Pour ce faire, il faut attendre de disposer d'une version 11 Release 4 de X Window System, exempte d'erreurs, ce qui ne saurait tarder[3].

Dès lors, nous pourrons profiter de toute la puissance du C++, par exemple en utilisant la librairie WWL (Widget Wrapper Library) développée par Jean-Daniel Fekete. Cette librairie encapsule tout X Window, de telle sorte que chaque type de Widget devient une classe d'objets. Les méthodes attachées à ces classes de widgets permettent une écriture beaucoup plus compacte. Ainsi, à titre d'exemple, voici ce

[1]Le problème peut néanmoins subsister en ce qui concerne l'interfaçage de l'éditeur avec le modèle. Rien n'impose en effet que le modèle soit écrit en C++. Mais comme le modèle doit de toutes façons être recompilé avec l'éditeur, on peut tenter de le recompiler avec g++.

[2]peut-être la version 1.39, qui n'a pas encore été installée.

[3]En fait, nous venons de la recevoir, mais elle n'est pas encore installée.

que donne l'utilisation de la classe *WXmString* (WWL) au lieu du type *XmString* (Motif) :

```
{
  XmString string = XmStringCreateLtoR ("Hello world !");
  Arg      args;

  XtSetArg (args, XmNlabelString, (XtArgVal) string);
  XtSetValue (widget, &args, 1);
}
```

devient simplement :

```
  widget.LabelString ("Hello world !");
```

7.4 Fonctionnalités non encore implémentées

Comme signalé dans la section étudiant les fonctionnalités désirées, le développement de l'interface graphique telle qu'elle est souhaitée ne pouvait être réalisé en huit mois. La version actuelle ne constitue donc que l'implémentation d'un sous-ensemble des fonctionnalités demandées.

Parmi les fonctionnalités qu'il reste à développer, citons :

- Impression.
  Il s'agit là, ainsi que nous l'avons déjà signalé, d'un *gros morceau.* Il reste à créer des fichiers aux formats Postscript et HPGL à partir des données contenues dans l'objet *theGraph.* En pratique, il faudra ajouter à chaque classe d'objets graphiques deux méthodes : *printPost* et *printHPGL.*

- Background.
  Faute de temps, rien de ce qui concerne le Background n'a été implémenté. Il ne s'agit pas là d'un gros problème, puisque les objets et méthodes graphiques de base existent déjà, notamment la classe *WorldPolyLine.* Il faudra cependant développer une classe *Polygon*, qui disposera d'une méthode de remplissage.

- Explosion des noeuds.
  La fonctionnalité *Explode* reste à implémenter.

- Chargement et Sauvegarde.
  Les fonctionnalités *Save* et *Load* doivent encore être implémentées.

7.5 Nouvelles fonctionnalités souhaitées

Toutes les fonctionnalités demandées au départ ne sont pas encore implémentées que déjà de nouvelles apparaissent à l'occasion de discussions avec de futurs utilisateurs potentiels.

Citons, par exemple, une fonction *Undelete*, qui permettrait de récupérer un noeud ou un arc effacé par inadvertance.

7.6 Améliorer la convivialité et l'ergonomie

Même si cette interface graphique a été développée dans un souci d'accroître la convivialité et l'ergonomie du modèle de trafic, il est illusoire de croire qu'elle apportera dès le départ un confort d'utilisation maximal. Ce n'est que par son utilisation prolongée que l'on pourra découvrir les améliorations à apporter.

Une première prise en mains lors des tests a déjà permis de soulever l'une ou l'autre question. Par exemple, lorsque l'on fait appel à la fonctionnalité *Translations* : la fenêtre indiquant les différentes directions de déplacement possibles doit-elle disparaître une fois la translation exécutée, ou bien doit-elle rester à l'écran, en vue d'une utilisation ultérieure ?

Une autre amélioration qu'il faudra certainement apporter au programme est l'augmentation de la sécurité d'utilisation par une vérification accrue des valeurs introduites par l'utilisateur (non redondance d'identificateurs lors de la création de nœuds ou d'arcs, etc).

Nul doute que bien d'autres suggestions seront faites par les futurs utilisateurs !

7.7 Interfaçage avec un modèle

Il faut bien se rendre à l'évidence : le programme développé jusqu'ici n'est qu'une coquille vide. Il reste maintenant à *interfacer l'interface* avec des bases de données et des modèles concrets.

7.8 Porter l'interface sur PC

Rappelons une contrainte essentielle qui avait été fixée dès le début : l'éditeur graphique développé doit pouvoir être utilisé sur PC. Les outils utilisés ne devraient pas poser de problèmes, mais notre expérience nous a appris que les concepts de compatibilité et de portabilité sont relativement flous.

Il reste donc à compiler les sources du programme dans un environnement PC.

Chapitre 8

Conclusions

Au début de ce travail, nous nous étions fixé trois objectifs. Revoyons ce qui a été fait et qui reste à faire.

1. Etablir, en collaboration avec ses futurs utilisateurs, les spécifications d'un éditeur graphique de réseau routier, qui permette :

   - de visualiser la géométrie de ce réseau,
   - de consulter et modifier la base de données qui lui est associée,
   - de visualiser de manière graphique certains éléments de cette base de données,
   - d'utiliser les fonctionnalités de modèles de gestion de trafic que l'on grefferait à cet éditeur.

   Les spécifications que nous avons obtenues constituent un minimum. La modularité du produit développé rend possible l'ajout de nouvelles fonctionnalités qui seraient suggérées par les utilisateurs.

2. Choisir les outils à utiliser.
   En ce qui concerne le système de fenêtrage, la contrainte de disponibilité sur station de travail et sur PC imposait pratiquement l'utilisation de X Window.
   En ce qui concerne l'ensemble de *widgets*, notre choix s'est porté sur le plus largement répandu : Motif.
   En ce qui concerne les outils graphiques, nous avons malheureusement dû, pour des raisons de compatibilité, renoncer aux outils les plus performants (PHIGS, GKS) et nous contenter de Xlib, librairie de bas niveau offerte par

X Window.
Enfin, comme langage de programmation, nous avons choisi C++. Hélas, pour des raisons temporaires d'incompatibilité avec la version de X Window dont nous disposions, nous avons dû écrire une partie du programme en C++, et l'autre en C. Disposant d'une version adéquate de X Window, il nous reste à réécrire en C++ la partie concernant le système de fenêtrage. Nous en profiterons pour utiliser la librairie WWL.

3. Implémenter l'éditeur.
Il était clair, dès le départ, qu'il était utopique de tout vouloir faire en huit mois. Dès lors, il a fallu nous contenter d'implémenter un sous-système cohérent des fonctionnalités désirées. Ce sous-système fonctionne correctement, même si du point de vue de la programmation et/ou de l'ergonomie, des améliorations peuvent être apportées. Il nous reste bien sûr à poursuivre l'implémentation.

Heureux. Nous sommes heureux d'avoir beaucoup appris (langage C++, programmation sous X Window). Nous sommes heureux d'avoir pu mener à bien une part importante du développement de cet éditeur graphique interactif. Et nous sommes encore plus heureux d'avoir l'occasion, dans les mois qui viennent, de poursuivre le travail entamé.

Liste des figures

3.1 Relations entre l'éditeur, les outils utilisés et le modèle sous-jacent . 18

4.1 Différences entre une application traditionnelle et une application X 20
4.2 Fonctionnement de X Window en réseau 23
4.3 Les "décorations" d'une fenêtre X Window 25
4.4 X Window : les couches de développement 28
4.5 La hiérarchie des widgets *Motif* 29
4.6 L'interface 33
4.7 Description du sous-menu *Graph* 36
4.8 Description du sous-menu *Nodes* 39
4.9 Description du sous-menu *Links* 41
4.10 Description du sous-menu *Set of nodes* 43
4.11 Description du sous-menu *Set of links* 45
4.12 Description du sous-menu *Background* 47

5.1 Opérations préliminaires à l'affichage 56
5.2 Passage des coordonnées utilisateur aux coordonnées écran 57
5.3 Codes de Cohen-Sutherland 61
5.4 Exemple d'application de l'algorithme de Cohen-Sutherland 62
5.5 Tracé d'un segment : position du problème 64
5.6 Voisinage d'un point 65
5.7 Octants du plan et mouvements possibles 65
5.8 Choix du point suivant 67

Bibliographie

X Window System

[1] Adrian Nye. *Xlib Programming Manual.* O'Reilly & Associates, Sebastopol, USA, 1989.

[2] Adrian Nye. *Xlib Reference Manual.* O'Reilly & Associates, Sebastopol, USA, 1989.

[3] Adrian Nye and Tim O'Reilly. *X Toolkit Intrinsics Programming Manual.* O'Reilly & Associates, Sebastopol, USA, 1990.

[4] Tin O'Reilly. *X Toolkit Intrinsics Reference Manual.* O'Reilly & Associates, Sebastopol, USA, 1990.

[5] Open Software Foundation. *OSF/Motif : Programmer's Guide.* Prentice-Hall, Englewood Cliffs, USA, 1991.

[6] Open Software Foundation. *OSF/Motif : Programmer's Reference.* Prentice-Hall, Englewood Cliffs, USA, 1991.

[7] Valerie Quercia and Tim O'Reilly. *X Window System User's Guide.* O'Reilly & Associates, Sebastopol, USA, 1989.

[8] Robert W. Scheifler, James Gettys, and Ron Newman. *X Window System.* Digital Press, Bedford, USA, 1988.

[9] Douglas A. Young. *X Window systems programming and applications with Xt.* Prentice-Hall, Englewood Cliffs, USA, 1989.

Algorithmes graphiques

[10] Victor J. Duvanenko, W. E. Robbins, and Ronald S. Gyurcsik. Improving line segment clipping. *Dr. Dobb's Journal,* 15(7):36–45, July 1990.

[11] J.D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley Publishing Company, Reading, USA, 1982.

[12] Michel Lucas (sous la direction de). *La réalisation des logiciels graphiques interactifs.* Editions Eyrolles, Paris, 1982.

[13] William M. Newman and Robert F. Sproull. *Principles of Interactive Computer Graphics.* New York, 1979.

[14] Philippe Schweizer. *Infographie II.* Presses Polytechniques Romandes, Lausanne, 1987.

C++

[15] Stanley B. Lippman. *C++ primer.* Addison-Wesley Publishing Company, Reading, USA, 1989.

[16] Keith Weiskamp and Bryan Flamig. *The complete C++ Primer.* Academic Press, London, 1990.