Titre: Title:	Modélisation interactive de paramètres rhéologiques et géométriques pour des procédés d'injection sur renfort
Auteur: Author:	Yanik Benoit
Date:	1996
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Benoit, Y. (1996). Modélisation interactive de paramètres rhéologiques et géométriques pour des procédés d'injection sur renfort [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8973/

UNIVERSITÉ DE MONTRÉAL

MODÉLISATION INTERACTIVE DE PARAMÈTRES RHÉOLOGIQUES ET GÉOMÉTRIQUES POUR DES PROCÉDÉS D'INJECTION SUR RENFORT

YANIK BENOIT
DÉPARTEMENT DE GÉNIE MÉCANIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE MÉCANIQUE)
NOVEMBRE 1996

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MODÉLISATION INTERACTIVE DE PARAMÈTRES RHÉOLOGIQUES ET GÉOMÉTRIQUES POUR DES PROCÉDÉS D'INJECTION SUR RENFORT

présenté par : BENOIT Yanik

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de :

M. GAUVIN Raymond, D.Sc.A., président

M. TROCHU François, Ph.D., membre et directeur de recherche

M. FORTIN Clément, Ph.D., membre

Remerciements

Je tiens à remercier mes collègues Pierre Ferland et Jérôme Chouinard, avec lesquels j'ai eu la chance de travailler pendant une courte année. Leur contribution à ce projet est capitale. Je remercie également mon directeur de recherche, M. François Trochu, pour la confiance dont il m'a témoignée tout au long de ce projet, ainsi que les membres du jury, qui ont accepté de lire ce document dans des délais très brefs.

Finalement, je remercie ma conjointe Élisabeth et ma famille pour leur patience et leur amour.

Résumé

Le procédé RTM consiste à injecter une résine thermodurcissable dans un moule rempli d'un renfort fibreux. Plusieurs paramètres sont impliqués dans la simulation numérique de ce procédé. Certains paramètres sont relativement complexes : la définition des caractéristiques rhéologiques et chimiques de la résine, par exemple, ou la définition des caractéristiques physiques de la préforme. Il existe dans la littérature plusieurs modèles paramétriques pour décrire la variation d'un paramètre physique tel que la viscosité d'une résine en fonction de la température, du temps et/ou du degré d'avancement de la réaction chimique. Cependant, l'identification expérimentale des coefficients de tels modèles est une opération coûteuse. L'utilisation de modèles interpolés constitue alors une alternative aux modèles paramétriques.

Ce mémoire se consacre à la conception et à l'implémentation d'un système d'édition interactive des paramètres de simulation du procédé RTM. Nous insistons principalement sur les paramètres de simulation auxquels il est possible d'associer une fonction d'une ou deux variables indépendantes. Un aspect important de ce travail consiste donc à l'élaboration d'une bibliothèque de courbes et de surfaces, constituée de modèles paramétriques et de fonctions d'interpolation. La conception orientée objet de cette bibliothèque est importante, puisqu'elle nous permet d'intégrer rapidement de nouveaux modèles paramétriques au code de calcul, tout en laissant la possibilité aux utilisateurs du logiciel d'avoir recours aux modèles interpolés.

Un interpréteur a été développé pour ce projet. La représentation d'un fichier texte en arbre qu'il génère est une caractéristique importante du système que nous proposons. Elle nous permet d'enregistrer sur disque, mais surtout de construire en mémoire vive des objets complexes.

La visualisation et l'édition interactive de courbes et de surfaces constitue un aspect important de ce travail. Nous présentons une bibliothèque graphique orientée objet permettant la visualisation dynamique d'une scène 3D. Cette bibliothèque est implémentée dans le langage graphique OpenGL. Nous utilisons cette bibliothèque pour la construction d'un éditeur de courbes et de surfaces interactif. Un modèle d'encapsulation des interfaces utilisateur qui s'est avéré très intéressant d'un point de vue génie logiciel est aussi présenté.

Finalement, nous démontrons les nouvelles fonctionnalités du code de calcul par un exemple complet de simulation du remplissage et de la cuisson d'une pièce dont la cinétique chimique est modélisée par krigeage et dont les conditions frontières thermiques varient dans le temps.

Abstract

Resin Tranfer Molding (RTM) consists of injecting thermoset resin in a closed mold filled with fibers. Many parameters take part in the numerical simulation of the process. Rheological and chemical properties of the resin system as well as physical properties of the preform are relatively complex simulation parameters. Many parametric models are available in the litterature to describe the effect of temperature, time or conversion on a physical parameter such as resin viscosity. However, there is an important cost implied for the identification of the constants in these models. In that context, interpolative models may be a better solution.

This work presents a system for interactive editing of the many parameters implied in the numerical simulation of an RTM process. An object-oriented library of mathematical functions that include parametric models as well as interpolative functions is described. Curves and surfaces from that library can be linked to simulation parameters. The object-oriented structure of that library is important since it enables us to easily add new parametric models to the system. Interpolative models generalize the system.

A parser has been developed to deal with the problem of reading and writing complex objects on disk. An original protocol based on a tree representation of a text file make objects responsible for their own reading and writing.

Viewing as well as interactive editing of curves and surfaces is an important part of this work. A graphics library based on OpenGL that is used throughout the project for 3D visualisation is presented. A curve editor and a surface editor based on that library has been developed.

Finally, a complete example demonstrating the new capabilities of the simulation core is given. This example deals with filling and curing of a part and demonstrates the use of an interpolative function to model a complex kinetic reaction. It also demonstrates the use of time varying thermal boundary conditions.

Table des matières

Liste des annexes

Liste des tableaux

Liste des figures

Introduction

La simulation numérique du procédé RTM ("resin transfer molding") permet d'analyser, en peu de temps et à faible coût, différentes stratégies d'injection afin d'optimiser le moule et les paramètres d'injection. Le nombre de variables intervenant dans la simulation du procédé est important. Mentionnons la position des points d'injection et des évents, la régulation thermique du moule, la pression d'injection, la viscosité de la résine, la réaction chimique de polymérisation de la résine, la déformation de la préforme sous une certaine pression, les caractéristiques physiques des tissus, du moule, etc.

Le module Dataflot, sujet de ce mémoire, s'inscrit dans le cadre du projet RTMFLOT, logiciel de simulation numérique du procédé RTM développé par le Centre de Recherche Appliquée Sur les Polymères (CRASP). Le code de calcul FLOT est responsable de la résolution des équations. Les utilisateurs du logiciel formulent régulièrement des demandes pour l'intégration de nouveaux modèles au code de calcul. La conception du module Dataflot est donc motivée à l'origine par un besoin de restructuration du module d'édition des paramètres de simulation de RTMFLOT, ainsi que par la restructuration du module d'entrée du code de calcul. Cette refonte était rendue nécessaire par la généralisation des fonctionnalités du logiciel pour permettre la simulation des moules RTM chauffés et, ultérieurement, pour inclure les procédés d'injection sur renfort avec paroi mobile (injection-compression). Une autre motivation importante est le concept de

base de données orientée objet afin d'enregistrer les paramètres physiques des résines et des renforts, de même que l'ensemble des informations requises par la simulation.

Le projet Dataflot consiste en plusieurs sous-projets : le logiciel Dataflot lui-même, un éditeur de courbes, un éditeur de surfaces ainsi qu'un interpréteur utilisé pour la lecture d'une base de données de paramètres de simulation.

Dataflot est le logiciel d'édition des paramètres de simulation. Il permet à l'utilisateur de créer interactivement un fichier de simulation qui sera par la suite utilisé par le code de calcul FLOT. Dataflot est doté d'une interface graphique dynamique construite dans l'environnement X/Motif, qui intègre des fenêtres de visualisation implémentées dans le langage graphique OpenGL. Il permet, en outre, la gestion d'une base de données de résines et de propriétés physiques de matériaux. L'utilisateur a la possibilité d'associer à un paramètre de simulation un modèle paramétrique ou un modèle interpolé. L'éditeur de courbes et de surfaces permet de construire les instances de courbes et de surfaces qui seront éventuellement chargées dans Dataflot et donc transmises au code de calcul. Voilà pour la portion "visible" du logiciel, celle qui intéresse l'utilisateur. Cependant, derrière ces nombreuses interfaces graphiques se cache une architecture logicielle originale, partagée par le code de calcul, Dataflot, l'éditeur de courbes et l'éditeur de surfaces. Le but de ce mémoire est d'expliquer le travail de conception réalisé dans le cadre du projet Dataflot.

Avant de débuter un travail de conception logicielle, on doit traverser la phase d'analyse, dont l'objectif est de comprendre le domaine de l'application. C'est pourquoi nous présentons au chapitre 1 une revue sommaire des paramètres physiques impliqués dans la simulation. Cette revue est importante puisqu'elle nous permettra de bien cerner les besoins immédiats ainsi que d'anticiper certaines demandes futures des utilisateurs. Nous constaterons que les modèles paramétriques sont parfois coûteux à obtenir, ce qui nous mènera au développement d'un système intégrant modèles paramétriques et modèles interpolés.

Le chapitre 2 décrit une bibliothèque de fonctions mathématiques. Cette bibliothèque implémente les différents modèles paramétriques disponibles pour la simulation ainsi que des fonctions d'interpolation générales telles que le krigeage dual. Nous insistons dans ce chapitre sur la conception orientée objet de la bibliothèque. L'évaluation d'une fonction est une opération polymorphe, c'est-à-dire que le module qui l'appelle n'a pas connaissance de la forme précise de la fonction, mais seulement de son interface (le nombre de variables indépendantes). Bien que nous nous intéressions principalement aux courbes et aux surfaces dans ce travail, nous généralisons cette architecture aux fonctions de N variables. C'est aussi dans ce chapitre que nous introduisons un aspect fondamental de ce travail, c'est-à-dire le protocole de lecture et d'écriture d'objets complexes. Ayant opté pour un format de fichier structuré en objets, format très souple facilitant l'introduction de nouveaux paramètres, un interpréteur a été développé spécifiquement pour le projet Dataflot. Son utilisation est décrite dans ce chapitre.

La visualisation et l'édition de courbes et de surfaces constituent un aspect important de ce projet. C'est pourquoi nous présentons au chapitre 3 une bibliothèque graphique fondée sur le langage graphique OpenGL définissant des visualisateurs de scènes 2D et 3D. Il est très important aussi de pouvoir interagir avec les objets que l'on visualise. Ces visualisateurs permettent la sélection d'objets graphiques à partir du mécanisme de sélection d'OpenGL. Cependant, la sélection d'objets graphiques en 3D avec OpenGL pose certains problèmes. Nous présenterons quelques solutions originales à ces problèmes.

Le chapitre 4 traite d'interaction avec les courbes et les surfaces. En particulier, un visualisateur de courbes fondé sur la bibliothèque graphique décrite au chapitre 3 sert de base à la construction d'un éditeur de courbes interactif. Les points importants de la conception d'un éditeur de surfaces sont aussi présentés dans ce chapitre.

Le chapitre 5 présente un aperçu des fonctionnalités offertes par le logiciel Dataflot. Ce chapitre s'adresse principalement au lecteur qui ne connaît pas le logiciel.

Le chapitre 6 explique comment la bibliothèque de courbes et de surfaces est utilisée dans FLOT et Dataflot. Il y est donc question de l'architecture du module d'entrée du code de calcul ainsi que de l'architecture des panneaux de Dataflot. Nous décrivons l'utilisation des courbes et surfaces dans la définition d'une résine. Le gestionnaire de base de données de Dataflot y est aussi décrit. Nous constaterons certaines lacunes dans la gestion de base de données et proposerons une généralisation de ce module de

Dataflot. Nous résumons aussi dans ce chapitre l'utilisation présente et à venir des courbes et des surfaces dans la simulation numérique du procédé RTM.

Finalement, un exemple d'application est traité pour illustrer un cas de moulage par RTM faisant appel aux concepts développés dans ce mémoire. Dans cet exemple un modèle de cinétique de polymérisation de la résine interpolé par krigeage est développé. Nous utilisons aussi dans cet exemple un modèle paramétrique de viscosité faisant intervenir la température et le degré de polymérisation de la résine. Le remplissage du moule est simulé avec un débit variable dans le temps. Enfin, nous simulons la cuisson de la pièce pendant un cycle de chauffage à la fin de l'injection.

Chapitre 1

Revue des paramètres rhéologiques et géométriques du procédé RTM

Dans ce chapitre, nous passons en revue quelques modèles utilisés pour décrire les principaux paramètres physiques impliqués dans la simulation du procédé RTM. Ce chapitre ne se veut pas une revue exhaustive. Nous cherchons uniquement à mettre en évidence le besoin devenu incontournable de remplacer dans la simulation numérique du procédé RTM certaines valeurs constantes comme la perméabilité, la porosité ou la viscosité, par des fonctions plus complexes.

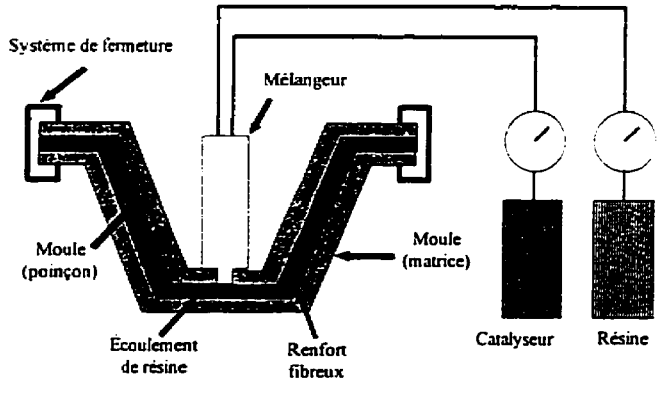


Figure 1.1 Procédé RTM

1.1 Procédé RTM

Le procédé RTM (moulage par transfert de résine) est utilisé pour la fabrication de pièces composites. Il consiste à injecter une résine thermodurcissable dans un moule fermé contenant un renfort fibreux sec. Lorsque la résine a entièrement imprégné le renfort, on provoque la polymérisation de la résine. On démoule la pièce lorsque le degré de polymérisation est satisfaisant.

1.2 Loi de Darcy

Un modèle communément accepté pour simuler l'écoulement d'un fluide en milieu poreux est la loi de Darcy, que l'on note souvent sous forme vectorielle :

$$\vec{V} = -\frac{K}{\mu}\nabla P, \tag{1.1}$$

où $\vec{V}$ est la vitesse de Darcy ($m.s^{-1}$), K est un tenseur de perméabilité (m^2), μ est la viscosité du fluide ($Pa.s$) et P est la pression (Pa).

Lorsque le fluide est incompressible, pour qu'il y ait conservation de la masse, la condition suivante doit de plus être respectée (équation de la divergence):

$$\nabla \cdot \vec{V} = 0 \tag{1.2}$$

En combinant les équations 1.1 et 1.2, on obtient

$$\nabla \cdot \left(\frac{K}{\mu} \nabla P \right) = 0 \qquad (1.3)$$

L'équation 1.3 est résolue numériquement avec la méthode des éléments finis. Les conditions frontières du problème sont :

- $P = P_0$ sur le front,
- $P = f(t)$ sur le contour d'injection (pression imposée) ou $\vec{V} \cdot \vec{n} = Q$ (débit imposé),
- $\vec{V} \cdot \vec{n} = 0$ sur la paroi imperméable du moule.

1.3 Présentation des familles de paramètres physiques

Nous nous intéressons principalement, dans ce travail, à deux familles de paramètres physiques : les paramètres reliés aux renforts et les paramètres reliés aux résines.

Les paramètres physiques reliés au renfort sont :

- la compaction,
- le taux de fibres,
- l'orientation des fibres,
- la perméabilité,

- la chaleur massique, la densité, ...

Les paramètres physiques reliés aux résines sont :

- la cinétique de polymérisation,
- la viscosité,
- la chaleur massique, la densité, ...

1.4 Drapage

Le drapage est l'opération qui consiste à empiler des tissus dans un moule, afin d'obtenir un composite stratifié. Cette opération est réalisée le plus souvent manuellement. Les tissus doivent être positionnés dans le moule selon un plan de drapage très précis. Un plan de drapage spécifie la découpe à plat de chaque couche, le référentiel dans lequel effectuer cette découpe, ainsi que le positionnement d'une couche par rapport à la couche précédente. Par exemple, le plan de drapage d'une pièce plane, rectangulaire, pourrait spécifier que la chaîne de chaque couche doit faire un angle de 45^0 par rapport à celle de la couche précédente. On imagine facilement la complexité d'un plan de drapage d'une pièce épaisse, vrillée, à section variable, constituée dans les parties épaisses de quelques 200 couches. Une pale de turbine est un exemple d'une telle pièce.

Initialement, l'angle entre la chaîne et la trame est de 90^0 en tout point du tissu. Pour que le renfort épouse une surface complexe, le tissu doit subir un cisaillement en chaque

point, impliquant une variation de l'angle entre la chaîne et la trame. Or le taux de fibres local dépend de cet angle. La connaissance d'un angle de cisaillement moyen entre la chaîne et la trame sur chaque élément du modèle éléments finis est donc importante, puisque cet angle influence localement la porosité, et donc la perméabilité d'un renfort fibreux.

Plusieurs algorithmes ont été développés afin de déterminer l'angle entre les fibres en chaque point d'un tissu drapé sur une surface complexe (Van der Ween, 1991). Nous en avons d'ailleurs étudié un au cours d'un stage à la société française SNECMA (Benoit, 1994). L'algorithme étudié au cours de ce stage a fait l'objet d'une publication (Trochu et al., 1996).

1.5 Taux de fibres

On calcule un taux de fibres (fraction volumique de fibres) moyen dans une pièce à partir de la formule suivante :

$$v_f = \frac{N * \rho_s}{h * \rho_v}, \tag{1.4}$$

où N est le nombre de plis de la préfome, ρ_S est la densité superficielle du tissu ($g.m^{-2}$), ρ_V est la densité volumique du matériau ($g.m^{-3}$) constituant le tissu (verre, carbone, etc.) et h est l'épaisseur de la cavité (m).

Le module de calcul actuel utilise la porosité uniquement dans l'algorithme de remplissage, afin de connaître le volume de résine que peut contenir chaque élément fini. Cette porosité est une valeur assignée par l'utilisateur pour tous les éléments d'une zone du maillage.

1.6 Compaction d'une préforme

Lorsqu'une préforme subit une certaine pression, les tissus se compactent, le volume de la préforme diminue et par le fait même le taux de fibres (v_f) de la pièce augmente. Le taux de fibres d'une pièce est très important, puisque ses propriétés mécaniques en dépendent. Dans cette section, on s'intéresse aux modèles reliant le taux de fibres v_f à la pression P appliquée sur la préforme.

Un modèle de compaction a au moins deux utilités. Dans un premier cas, il permet de déterminer la pression de fermeture d'un moule requise pour obtenir un certain taux de fibres. Dans un autre cas, la pression d'injection du fluide entraîne une variation de l'épaisseur de la préforme autour du point d'injection. Un modèle de compaction permet de prédire cette variation.

Gauvin et al. (1986, 1988) proposent un modèle semi-empirique de compaction faisant intervenir quatre paramètres A_0, A_1, A_2, A_3. Ces paramètres sont calculés à partir de résultats expérimentaux de compaction sur une seule couche de tissu.

$$v_f(P) = \frac{10v_0}{10A_0P + A_1\ln(10P) + \frac{A_2}{10P} + A_3}, \qquad (1.5)$$

Pour caractériser une préforme constituée de plusieurs couches de tissus différents, on simplifie souvent le calcul de v_f par une somme pondérée du taux de fibres de chaque couche :

$$v_{f_{moy}} = \frac{1}{H}\sum_{i=1}^{n} h_i v_{f_i} \qquad (1.6)$$

où H est l'épaisseur totale de la préforme et h_i est l'épaisseur de la couche i. Lemenn (1994) passe en revue d'autres modèles de compaction. Batch et al. (1990) proposent le modèle suivant :

$$P = K(v_f - v_0), \qquad (1.7)$$

où K est une constante d'élasticité. La valeur de K dépend du régime, linéaire ou non-linéaire :

$$K = K_0 \qquad \text{pour } v_f < v_{fcont} \qquad (1.8)$$

$$K = K_0 \frac{(1-\eta')(\frac{1}{v_0} - \frac{1}{v_f})}{\eta'(\frac{1}{v_f} - \frac{1}{v_m})} \quad \text{pour } v_f \geq v_{fcont}, \qquad (1.9)$$

où le seuil de transition linéaire/non linéaire v_{fcont} se calcule par

$$v_{fcont} = \left[\frac{1}{v_0} - \eta' \left(\frac{1}{v_0} - \frac{1}{v_m} \right) \right]^{-1}, \tag{1.10}$$

où v_m est la fraction volumique maximum avant cassure des fibres, η' un facteur d'inefficacité des fibres au compactage et K_0 une constante d'élasticité des fibres.

Taylor (1948) propose toutefois un modèle plus simple :

$$v_f = v_1 + c_c \log_{10}\left(\frac{P}{P_1}\right) \tag{1.11}$$

où P_1 est généralement la pression atmosphérique, v_1 est le taux de fibres mesuré à P_1, et c_C est une constante caractérisant le tissu.

Les résultats expérimentaux de Lemenn (1994) démontrent que, pour les tissus étudiés, le modèle de Gauvin fournit les meilleures prédictions. Il observe aussi une différence importante pour les courbes de compaction de 3 et 6 couches. Ces différences sont dues aux interfaces entre les couches, mais elles tendent à disparaître quand le nombre de couches augmente.

D'autres facteurs importants sont à considérer dans l'étude de la compaction des renforts. Mentionnons les recompactions successives d'une préforme, qui ont pour effet de réduire la pression requise pour obtenir un taux de fibres donné, d'un essai de

compaction à l'autre. Cet effet est significatif pour les quelques premières compactions de la préforme, mais disparaît progressivement (Lemenn, 1994).

La relaxation de la préforme n'est pas négligeable et doit aussi être prise en compte. Après compaction, on observe que la pression nécessaire pour maintenir constante l'épaisseur de la préforme décroît dans le temps (Lemenn, 1994).

Ces phénomènes de recompaction et de relaxation sont encore à l'étude et nous n'en tiendrons pas compte pour l'instant dans la simulation numérique du procédé, bien qu'ils constituent un axe de développement. C'est pour les introduire facilement dans le logiciel que nous avons adopté une conception modulaire selon l'approche orientée objet.

1.7 Perméabilité

La perméabilité (K) est un paramètre déterminant dans la résolution de l'équation de Darcy. La principale dépendance de K est vis-à-vis du taux de fibres. Si pour des matériaux particuliers, tels les mats de verre, des relations empiriques ont été proposées, la valeur de K est généralement mesurée. En principe, on s'intéresse aux trois valeurs de K dans les directions principales. Le tenseur [K] se résume donc à trois fonctions du taux de fibres v_f:

$$K_1 = f_1(v_f)$$

$$K_2 = f_2(v_f) \tag{1.12}$$

$$K_3 = f_3(v_f)$$

Dans la version actuelle du code de calcul, trois valeurs de perméabilité constantes (K_1, K_2, K_3) sont assignées par l'utilisateur à chaque zone du maillage éléments finis. FLOT ne calcule pas v_f pour en déduire K. Dans un proche avenir, ce calcul sera nécessaire et il faudra alors substituer des courbes de perméabilité aux constantes actuelles.

1.8 Modèles de viscosité

On note deux approches pour modéliser la viscosité d'une résine. Dans un premier cas, on considère la viscosité comme dépendante du temps, et les coefficients du modèle dépendent de la température. La forme générale de ce modèle est la suivante (Fountain et al.,1975; Yousefi, 1996):

$$\mu(T) = \mu_0 e^{kT}, \tag{1.13}$$

où μ_0 et k dépendent de la température selon une loi d'Arrhénius. L'autre approche considère la viscosité comme dépendante de la température, et les coefficients du modèle dépendent alors de la cinétique de polymérisation. La forme générale de ce modèle est la suivante :

$$\log \mu(T) = \log \mu(T_s) + \frac{a(T - T_s)}{b + (T - T_s)}, \tag{1.14}$$

où a et b sont des constantes, μ est la viscosité, T la température, et T_S, $\mu(T_S)$ des fonctions du degré de conversion α.

Castro (1980) et Wang et al. (1990) ont introduit le modèle suivant :

$$\mu = \mu_0 \left[\frac{\alpha_g}{\alpha_g - \alpha} \right]^{f(\alpha,T)}, \tag{1.15}$$

où $\mu_0 = \mu(\alpha = 0, T) = A_\mu \exp(\Delta E_\mu / RT)$ et α_g est le degré de conversion au gel.

Ferland (1994) constate que le modèle suivant, qui est relativement simple, est souvent utilisé:

$$\mu(T,\alpha) = Ae^{\frac{B}{T}+C\alpha} \tag{1.16}$$

où A, B et C sont des constantes à mesurer qui dépendent de la résine, T est la température et α le degré de polymérisation.

1.9 Cinétique de polymérisation d'une résine

Un modèle simple de cinétique de polymérisation à une seule réaction est défini par:

$$\dot{\alpha} = k(1-\alpha)^n \tag{1.17}$$

où α est le degré de polymérisation, n est l'exposant de la réaction et k est une fonction d'Arrhénius, c'est-à-dire

$$k = k_0 \exp(-E / RT), \tag{1.18}$$

où k_0 est une constante, E l'énergie d'activation, R la constante des gaz et T la température.

En réalité, plusieurs réactions chimiques se produisent simultanément pendant la polymérisation. Kamal et Sourour (1973, 1974) proposent le modèle suivant pour tenir compte de n réactions simultanées :

$$\begin{aligned} \alpha &= \sum_{i=1}^{n} C_i \alpha_i \\ \dot{\alpha}_i &= K_i \alpha_i^{m_i} (1-\alpha_i)^{p_i} \\ \sum_{i=1}^{n} C_i &= 1 \end{aligned} \tag{1.19}$$

où $\dot{\alpha}_i$ est le taux de conversion de la sous-réaction i, K_i est une fonction de la température (Arrhénius), m_i et p_i sont les exposants de la réaction.

Le système 1.19 ne tient pas compte du phénomène de vitrification qui arrête la réaction avant que la polymérisation soit complète. Le modèle de Gonzalez-Romero et al. (1989) tient compte de la vitrification en introduisant un nouveau paramètre α_{max}, degré de conversion au-delà duquel la réaction s'arrête :

$$\dot{\alpha} = A \exp(-E / RT)(\alpha_{max} - \alpha)^n g(\alpha), \tag{1.20}$$

où $g(\alpha)$ a généralement la forme suivante :

$$g(\alpha) = \exp(m\alpha), \tag{1.21}$$

où m est une constante.

La chaleur dégagée par la réaction chimique dépend du taux de conversion :

$$Q = H_R\dot{\alpha}, \tag{1.22}$$

où H_R est l'enthalpie de la résine.

En pratique, peu d'industriels sont en mesure de fournir un modèle de cinétique pour leurs résines. D'une part, les résines sont très nombreuses. Dans le cas des résines polyesters ou vinylesters, la cinétique dépend fortement des mélanges et additifs qui changent constamment. De plus, la détermination des constantes d'un modèle est une opération coûteuse que peu d'entreprises sont prêtes à financer. Par conséquent, les utilisateurs du logiciel RTMFLOT s'orientent de plus en plus vers l'utilisation de courbes et de surfaces produites expérimentalement. On ne cherche plus à modéliser un phénomène, on se contente de le reproduire par interpolation.

En ce sens, il est intéressant de remplacer un modèle de viscosité $\mu(T, \alpha)$ par $\mu(T, t)$, où t est le temps et T la température. Ainsi, on évite la détermination des paramètres d'un modèle cinétique. On effectue plutôt une série de mesures de viscosité dans le temps, pour quelques valeurs de température fixées. On construit une fonction d'interpolation pour $\mu(T, t)$ à partir de ces mesures.

1.10 Autres paramètres

On peut ajouter d'autres paramètres définis par une dépendance à une ou deux variables à la liste précédente. Mentionnons certaines valeurs reliées à l'analyse thermique, telles que la chaleur massique ou la conductivité thermique, qui dépendent de la température (Audet, 1996; Yousefi, 1996).

1.11 Conclusion

Dans ses premières versions, le code de calcul FLOT utilisait des propriétés constantes pour caractériser les matériaux intervenant dans la simulation. Ceci se justifiait par le degré d'avancement des technologies d'injection sur renfort, la puissance de calcul de l'époque et la possibilité d'évaluer les dépendances physiques. Aujourd'hui, les utilisateurs du logiciel cherchent à réaliser des simulations plus réalistes qui prennent en considération les variations des propriétés physiques de certains paramètres. Les procédés d'injection sur renfort évoluent aussi. Par exemple, les moules sont chauffés pour accélérer l'injection en diminuant la viscosité de la résine. Il faut donc tenir compte de la dépendance de la viscosité par rapport à la température et au degré d'avancement de la réaction de polymérisation. Il est aussi important de pouvoir simuler les procédés de type injection-compression qui impliquent un déplacement de la paroi du moule pendant l'injection.

En résumé, il est impératif de substituer aux valeurs constantes des paramètres physiques des courbes ou des surfaces. Ce projet est dédié à l'intégration de ces fonctionnalités dans le module de calcul FLOT.

Chapitre 2

Bibliothèque de courbes et de surfaces

Une application de calcul scientifique fait appel continuellement à l'évaluation de fonctions d'une, de deux, ou de plusieurs variables. L'évaluation d'une fonction est une opération polymorphe : son interface est toujours la même, mais son implémentation varie selon la classe de la fonction.

Nous présentons deux bibliothèques orientées objet dans ce chapitre, `libcurve` et `libsurf`. La première implémente des fonctions d'une variable (courbes) alors que la seconde implémente des fonctions de deux variables (surfaces). On retrouve dans ces bibliothèques des fonctions explicites et paramétriques. La généralisation aux fonctions de N variables est proposée. Nous définissons aussi une hiérarchie d'ensembles de points de contrôles, de structure 1D, 2D et 3D.

Nous débutons par l'explication de concepts de programmation orientée objet qui reviendront à quelques reprises dans ce mémoire, et que nous estimons nécessaires à la compréhension de ce travail.

2.1 Conception et programmation orientées objet

2.1.1 Notation utilisée (méthode OMT)

À quelques reprises dans ce mémoire, des diagrammes seront nécessaires pour exprimer le plus clairement possible les aspects importants de la conception orientée objet d'un logiciel. Nous avons choisi pour ce faire la notation OMT ("Object Modeling Technique"), développée par Rumbaugh et al. (1991). Nous présentons dans cette section deux types de diagrammes utilisés dans ce mémoire, ainsi que les symboles qui s'y rattachent.

Un diagramme d'objets montre les associations qui existent entre les différentes classes d'un système. Le symbole OMT pour représenter une *classe* est une boîte subdivisée en trois parties. La partie du haut contient le nom de la classe, celle du milieu contient les attributs de la classe et la troisième contient les opérations que la classe supporte. On représente un *objet* par une boîte aux coins arrondis. Les attributs et les opérations sont facultatifs. Par exemple, sur la figure 2.1, nous nous contentons d'indiquer le nom et les attributs de chaque classe. Une association a toujours un nom. Par exemple, nous pourrions nommer *travaille pour* l'association entre un employé et sa compagnie. En général, le nom d'une association est trivial et nous préférons l'omettre. Par contre, il est fréquent d'indiquer la multiplicité d'une association. Un point noir signifie *plusieurs*. L'absence d'indication signifie *un seul*. Ainsi, sur la figure 2.1, une personne

travaille pour une seule compagnie, alors qu'une compagnie emploie plusieurs personnes.

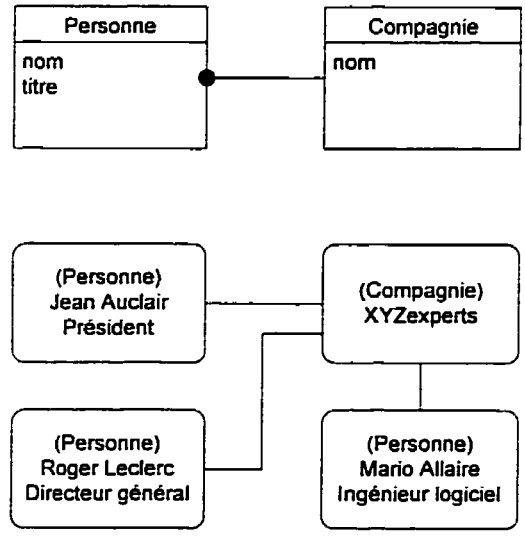


Figure 2.1 Diagramme de classes vs diagramme d'instances

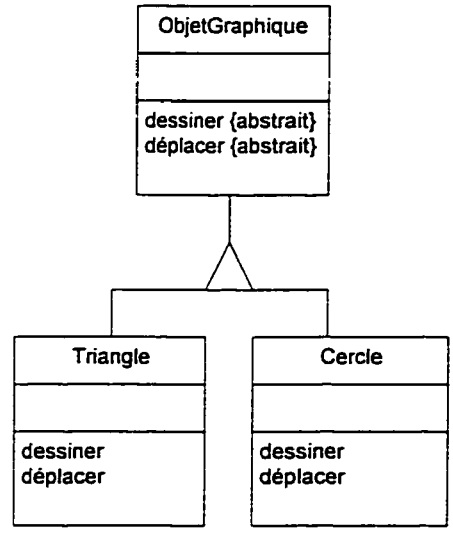


Figure 2.2 Notation OMT utilisée pour une relation d'héritage

La figure 2.2 montre un exemple de relation d'héritage. La super-classe `ObjetGraphique` est une classe abstraite puisque les deux opérations qu'elle supporte sont abstraites. Une opération abstraite dans une super-classe oblige les sous-classes à l'implémenter.

Nous aurons recours à quelques reprises au diagramme d'événements, qui illustre un scénario d'échange de messages entre objets. Sur la figure 2.3, un objet de classe `classe_A` transmet d'abord le message `message_X` à une instance de `classe_B`, puis `message_Y` à un objet de classe `classe_C`. Finalement, l'objet `classe_C` répond à l'objet `classe_A` par le message `message_Z`.

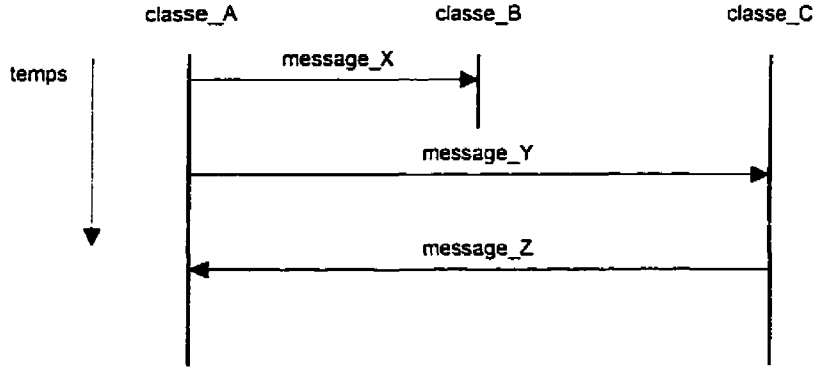


Figure 2.3 Diagramme d'événements

2.1.2 Constructeur de copie virtuel

Nous introduisons dans cette section le concept de constructeur de copie virtuel, notion qui reviendra tout au long de ce travail.

Voici d'abord une situation typique qui illustre la nécessité des constructeurs de copie virtuels : on souhaite créer *dynamiquement* un nouvelle instance, copie conforme d'un autre objet dont on ne connaît pas le type précis, mais seulement la super-classe. Par exemple, créer une copie conforme d'une courbe, en sachant seulement que l'objet à copier est une courbe, mais sans connaître la nature de cette courbe (exponentielle, krigée, etc.). Notre but est de copier la courbe "en entier", et non simplement les propriétés de la super-classe. La solution retenue consiste à demander à la courbe de se copier elle-même, puisque seule la courbe connaît complètement ses spécificités.

Pour y arriver, on dote la super-classe `CrvCurve` de l'opération virtuelle pure `copyCurve()`. Cette opération doit absolument être réalisée par les classes les plus dérivées dans une hiérarchie. Seule la classe la plus dérivée dans une hiérarchie a connaissance de tous les membres qui la constituent. L'implémentation retenue pour le constructeur de copie virtuel est la suivante :

```
CrvCurve* CrvConstant::copyCurve() {
   return new CrvConstant(*this);
}
```

Autrement dit, une instance de la classe `CrvConstant`, sachant qu'elle est une courbe constante, crée une copie d'elle-même à partir du pointeur `this`. Rappelons que `this` est une variable membre de tout objet et qu'il s'agit d'un pointeur à l'objet lui-même.

Pour que la copie s'effectue correctement avec cette implémentation, le constructeur de copie doit avoir été défini à tous les niveaux de la hiérarchie.

2.1.3 Conversion du type d'un pointeur vers le bas

Le langage C++ est très permissif en ce qui concerne la conversion de pointeurs. Ceci se manifeste principalement lors d'une conversion de pointeur d'un type donné vers le type `void *`. On montre dans l'exemple suivant que le langage C++ est très permissif en ce qui concerne la conversion de pointeurs :

```
Sphere *sphere = new Sphere;
void *temp = (void *) sphere;
Triangle *triangle = (Triangle *) temp;
```

Nous avons ainsi converti un pointeur à une instance de la classe `Sphere` en pointeur à une instance de la classe `Triangle`. Cependant, le pointeur `triangle` résultant n'est pas valide et son utilisation génère une erreur fatale.

Malgré les dangers reliés à la conversion du type d'un pointeur, celle-ci nous est parfois très utile. Nous avons rencontré le problème suivant à quelques reprises dans ce projet : identifier la sous-classe d'un objet à partir d'un pointeur à la super-classe, et convertir le pointeur à la super-classe en pointeur à la sous-classe afin d'accéder aux membres de la sous-classe.

Par exemple, plaçons nous dans le contexte de la sélection d'un objet à l'écran avec la souris. Supposons que le sous-système qui gère la sélection d'un objet retourne un

pointeur à la super-classe `GraphicObject` de l'objet sélectionné. Pour une raison quelconque, nous voulons savoir si l'objet sélectionné est une sphère. De plus, si c'est une sphère, nous voulons l'interroger pour connaître son rayon. L'exemple suivant montre la solution retenue pour résoudre de problème :

```
GraphicObject *object = getSelectedObject(); // abstraction du
                                             // sous-système de sélection
if (object->isSphere()) {
   Sphere *sphere = (Sphere *) object->getObjectPtr();
   float radius = sphere->getRadius();
   ...
}
```

La classe `GraphicObject` définit un ensemble de méthodes virtuelles permettant d'identifier la classe de l'objet : par exemple, `isSphere()`, `isTriangle()`, etc ... Ces méthodes retournent toutes FAUX par défaut. La classe concernée redéfinit sa méthode d'identification pour qu'elle retourne VRAI. Par exemple, la classe `Sphere` redéfinit `isSphere()` pour qu'elle retourne VRAI.

L'objet client doit d'abord identifier la sous-classe de `GraphicObject` par une série de tests (aussi longue qu'il y a de sous-classes). Puis il procède à la conversion vers le bas du pointeur. Cette opération est délicate. La compilateur n'accepte pas une conversion directe vers le bas comme ceci :

```
Sphere* sphere = (Sphere *) object;
```

Une autre très mauvaise idée serait de déjouer le compilateur en utilisant temporairement un `void*` :

```
void *temp = (void *) object;
Sphere *sphere = (Sphere *) temp;
```

Même si le compilateur accepte ces deux lignes, l'exécution du programme sera catastrophique.

C'est pourquoi nous avons défini le protocole suivant : chaque sous-classe redéfinit la méthode `getObjectPtr()` de la super-classe de façon à ce qu'elle retourne un pointeur à elle-même. Par exemple, voici le code de la méthode `getObjectPtr()` pour une sphère :

```
void* Sphere::getObjectPtr() {
   return (void *) this;
}
```

Le pointeur retourné est bien un `Sphere*`, converti en `void*` simplement pour uniformiser la signature de l'opération de la sous-classe avec celle de la super-classe. En C++, deux signatures ne peuvent différer que par le type de la valeur de retour.

2.1.4 Post-constructeur

Un post-constructeur a pour rôle de reconstruire un objet déjà construit avec de nouveaux attributs. En général, si N constructeurs sont définis sur une classe, on devrait retrouver N post-constructeurs de signature identique à celles des constructeurs. Chaque constructeur ne fait qu'appeler le post-constructeur qui lui est associé.

Prenons l'exemple d'un objet graphique de classe `Sphere`. Considérons deux attributs: son rayon et sa couleur. Le constructeur de `Sphere` a la signature suivante :

```
Sphere::Sphere(float radius);
```

Le post-constructeur qui lui est associé est :

```
Sphere::setRadius(float radius);
```

Le constructeur ne fait qu'appeler le post-constructeur de même signature :

```
Sphere::Sphere(float radius) {
   setRadius(radius);
}
```

Modifier le rayon d'une sphère déjà construite est une opération de post-construction, car la propriété fondamentale de la sphère est modifiée. Par contre, `setColor()` ne peut pas être vu comme un post-constructeur, car la couleur d'une sphère est un attribut de seconde importance. Il serait illogique d'avoir un constructeur qui prend en paramètre la couleur de la sphère et non son rayon.

Certains concepteurs adoptent la convention suivante : toutes les méthodes `set()` d'une classe ne font que copier des valeurs dans l'objet. Aucun traitement complexe n'est effectué. Nous préférons adopter une règle de cohérence d'un objet : un objet doit toujours être fonctionnel et cohérent. Par exemple, la méthode `setRadius()` doit

faire plus que simplement copier la valeur du rayon dans l'objet. Elle doit regénérer la représentation graphique de l'objet afin qu'elle soit cohérente avec le nouveau rayon..

2.2 Fonctions explicites et fonctions paramétriques

Nous nous intéressons, dans ce travail, à deux familles de courbes et de surfaces : explicites et paramétriques. En général, *explicite* s'utilise en opposition à *implicite*. Par exemple, la courbe implicite

$$f(x,y) = x^2 + y^2 - r^2 = 0$$

définit un cercle de rayon r centré à l'origine. On dit que cette courbe est implicite puisqu'elle définit indirectement les points constituant la courbe. À partir d'un point (x, y), on vérifie si l'équation (2.1) est vérifiée et si c'est le cas, le point (x, y) fait partie du cercle.

Tout au long de ce mémoire, nous utiliserons *explicite* en opposition à *paramétrique*. Les paragraphes suivants définissent la forme générale de ces courbes et surfaces :

- Courbe explicite :

$$y = f(x)$$

où y est la variable dépendante et x est la variable indépendante. Il s'agit d'une fonction scalaire d'une variable scalaire.

exemple : $y = x^3 + 3x^2 + x + 1$, courbe polynômiale de degré 3.

- Courbe paramétrique :

$$\vec{P}(t) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} f_1(t) \\ f_2(t) \\ f_3(t) \end{bmatrix},$$

où t est la variable indépendante (le paramètre) et **P** est la position d'un point dans l'espace (variable dépendante). Une courbe paramétrique est une fonction vectorielle d'une variable scalaire.

exemple : $\vec{P}(t) = \begin{bmatrix} \cos(2\pi t) \\ \sin(2\pi t) \\ 0 \end{bmatrix}$, cercle de rayon 1 centré en (0, 0).

- Surface explicite :

$$z = f(x,y),$$

où x et y sont les variables indépendantes et z est la variable dépendante. Une surface explicite est une fonction scalaire de deux variables scalaires.

exemple : $z = a_0 + a_1 x + a_2 y + a_3 x^2 + a_4 xy$, surface polynômiale de degré 2 en x et de degré 1 en y.

- Surface paramétrique :

$$\vec{P}(u,v) = \begin{bmatrix} f_1(u,v) \\ f_2(u,v) \\ f_3(u,v) \end{bmatrix},$$

où u et v sont les variables indépendantes (paramètres) et **P** est la position d'un point dans l'espace. Il s'agit d'une fonction vectorielle de deux variables scalaires.

exemple : $\vec{P}(u,v) = \begin{bmatrix} \cos(2\pi u) \\ \sin(2\pi u) \\ -\cos(\pi v) \end{bmatrix}$, sphère creuse de rayon 1 centrée à l'origine.

- Solide paramétrique :

$$\vec{P}(u,v,w) = \begin{bmatrix} f_1(u,v,w) \\ f_2(u,v,w) \\ f_3(u,v,w) \end{bmatrix},$$

où u, v et w sont les variables indépendantes (paramètres) et **P** est la position d'un point dans l'espace.

exemple : $\vec{P}(u,v,w) = \begin{bmatrix} w\cos(2\pi u) \\ w\sin(2\pi u) \\ -w\cos(\pi v) \end{bmatrix}$, sphère pleine de rayon 1 centrée à l'origine.

2.3 Modèles paramétriques et fonctions paramétriques

La distinction entre fonction paramétrique et modèle paramétrique est importante pour la compréhension de la suite de ce texte. Les fonctions paramétriques sont utilisées, en général, dans une application géométrique : modélisation géométrique, génération de maillage, etc. On les utilise pour décrire mathématiquement des courbes, des surfaces et des solides dans l'espace.

Un modèle paramétrique, dans ce mémoire, fait référence à une fonction explicite utilisée dans le contexte de la modélisation d'un phénomène physique. Un ensemble de coefficients constants à déterminer à partir de mesures expérimentales paramétrisent la fonction explicite. Les équations 1.5 et 1.16 constituent des exemples de modèles paramétriques.

2.4 Lecture/écriture d'objets complexes

Cette section est fondamentale pour l'ensemble du projet. Elle décrit la méthode utilisée pour construire un objet complexe à partir d'une chaîne de caractères contenue dans un fichier. Nous utilisons une représentation en arbre intermédiaire entre le fichier et la mémoire vive. Un objet complexe se construit lui-même à partir de cet arbre.

2.4.1 Construction d'un arbre à partir d'un fichier texte

Le premier traitement à appliquer au fichier texte dans le but de construire en mémoire vive les objets qu'il définit consiste à le transformer sous la forme intermédiaire d'arbre. Un tel arbre est constitué de couples (jeton, valeur). Un jeton est ici le nom d'un paramètre. Par exemple, TEMPERATURE. La valeur associée au jeton est parfois simple (un nombre réel, par exemple), mais elle peut être complexe. L'exemple suivant montre comment ces deux types de valeurs coexistent dans un fichier Dataflot :

```
// valeur simple
TEMPERATURE : 273

// valeur complexe
VISCOSITY {
   FAMILY : Curve
   TYPE : Exponential
   A : 1.0
   B : 1.0
}
```

Le jeton TEMPERATURE a pour valeur un nombre réel tandis que le jeton VISCOSITY a une valeur complexe, soit une courbe exponentielle.

Définissons plus précisément un élément de l'arbre. Il s'agit d'une structure nommée `IONode` et définie de la façon suivante :

```
struct IONode {

   char *token;
   char *value;
   List<IONode> *content;

};
```

On remarque dans cette structure la présence d'un pointeur à une liste générique (le "template" `List`) de `IONode`.

Un arbre est donc représenté en mémoire par une liste de `IONode`, ou plus simplement `List<IONode>`.

Lorsqu'un jeton a une valeur simple, seuls les champs `IONode::token` et `IONode::value` sont utilisés. La description d'une valeur complexe se trouve dans la liste pointée par `IONode::content`.

La fonction `CreateParsingTree()` est responsable de la construction d'un arbre de jetons. Elle prend en paramètre un nom de fichier et elle retourne une liste de `IONode`.

L'appel à `CreateParsingTree()` avec un fichier contenant les deux jetons de l'exemple précédent génère un arbre tel que montré à la figure 2.4. On voit sur cette figure deux listes de `IONode`. La première contient les jetons Temperature et VISCOSITY. Temperature ayant une valeur simple, son champ `content` est initialisé à NULL. Par contre, VISCOSITY ayant une valeur complexe, son champ `content` pointe à une seconde liste, laquelle définit une courbe exponentielle.

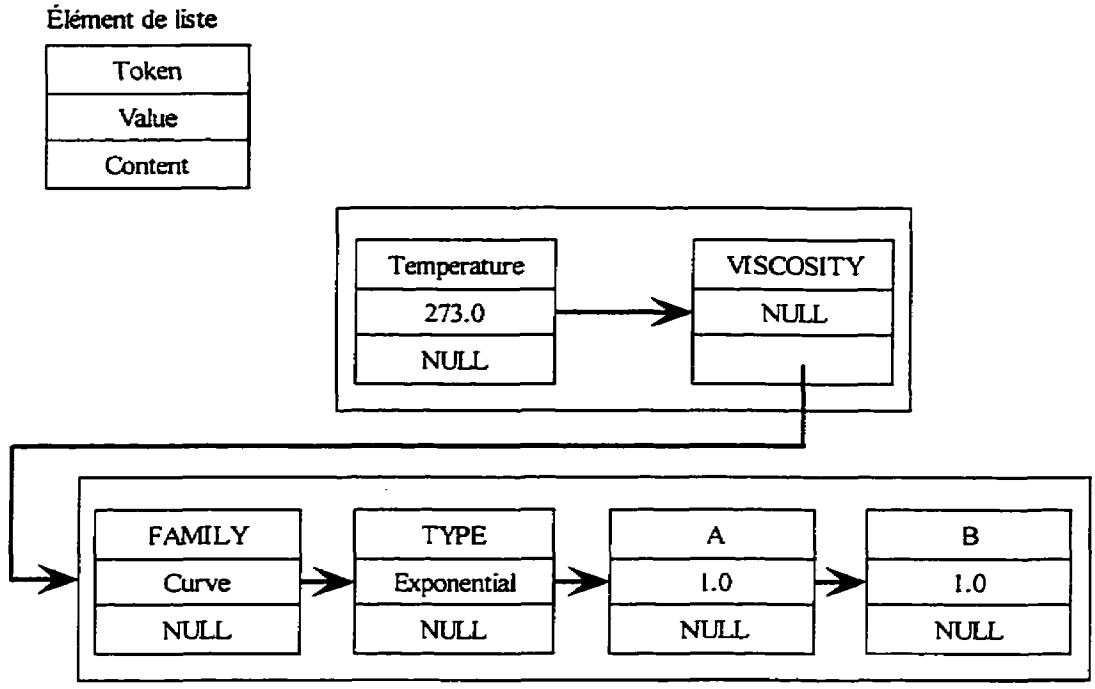


Figure 2.4 Arbre de jetons en mémoire

2.4.2 Construction d'objets complexes à partir d'un arbre

Pour construire un objet à partir d'une arbre de jetons, nous disposons d'un ensemble de fonctions de lecture pour chaque type élémentaire du langage C. Ces fonctions ont toutes la même forme :

- `int ReadInt(IONode* node);`
- `float ReadFloat(IONode* node);`
- `char *ReadString(IONode* node);`

Elles prennent en paramètre un `IONode` et interprètent la chaîne de caractères `IONode::value` comme étant un entier, un nombre réel ou une chaîne de caractères. Elles utilisent donc à l'interne des appels aux fonctions de la librairie C standard, telles que `atoi()` et `atof()`, qui construisent un entier ou un nombre réel à partir d'une chaîne de caractères. Par exemple, on implémente `ReadFloat()` ainsi :

```
float ReadFloat(IONode* node) {
   return atof(node->value);
}
```

Une autre fonction très utile a la signature suivante :

```
IONode* FindToken(List<IONode> *liste, char *token);
```

Cette fonction permet de rechercher un jeton particulier dans un arbre. Supposons que le code de calcul recherche le jeton TEMPERATURE. Un appel à `FindToken()` permet de localiser ce jeton. Ensuite, puisque FLOT sait a priori que ce jeton a une valeur simple et que cette valeur est de type réel, il appelle `ReadFloat()`. C'est ce que nous montrons dans l'exemple suivant :

```
List<IONode>* list = CreateParsingTree("test.dtf");
IONode* node = FindToken(list, "TEMPERATURE");
float temperature = ReadFloat(node);
```

La lecture d'un objet complexe se fait à l'aide des fonctions simples que nous venons de décrire. Nous rajoutons cependant une autre fonction très utile :

```
List<IONode>* FindFamilyMembers(List<IONode> *list, char*
              family);
```

Le rôle de cette fonction est de parcourir récursivement la liste qui lui est passé en paramètre et d'identifier tous les noeuds dont le jeton est FAMILY et dont la valeur correspond au paramètre `family`. Elle retourne la liste des noeuds trouvés. Son utilité sera décrite un peu plus loin.

Chaque super-classe d'objet a une fonction de lecture générique. Par exemple, pour lire une courbe ou une surface, nous avons défini les fonctions suivantes :

- `CrvCurve* ReadCurve(IONode* node);`
- `SrfSurface* ReadSurface(IONode* node);`

Pour construire une courbe à partir d'un arbre, la fonction `ReadCurve()` doit d'abord identifier la classe de la courbe à construire. Pour ce faire, elle recherche dans l'arbre qui lui est passé en paramètre le jeton TYPE. Puis `ReadCurve()` construit une instance par défaut de cette classe et lui propage aussitôt l'opération de lecture, puisque c'est la courbe elle-même qui connaît les jetons dont elle a besoin pour se reconstruire :

```
CrvCurve* ReadCurve(IONode *node) {

   IONode* type = FindToken(node->content, "TYPE");
   Curve* curve = NULL;
   if (strcmp(type->value, "exponential") == 0) {
      curve = new CrvExponential;  // constructeur par défaut
   }
   else if ... // on a une condition pour chaque type de courbe
               // défini
   else if ...
   else {}
```

```
    // on demande à la courbe de se post-construire à partir
    // du noeud de l'arbre de jetons la décrivant
    curve->read(node);
    return curve;

}
```

La classe `CrvExponential` (courbe exponentielle implémentant l'équation $f(x) = Ae^{Bx}$) définit le post-constructeur `read()` de la façon suivante :

```
CrvExponential::read(IONode *node) {

    IONode *nodeA, *nodeB;
    nodeA = FindToken(node->content, "A");
    nodeB = FindToken(node->content, "B");
    myCoefA = ReadFloat(nodeA);
    myCoefB = ReadFloat(nodeB);

}
```

On déduit de cet exemple que tous les objets complexes qui savent se lire eux-mêmes doivent définir un post-constructeur nommé `read()`, qui reçoit en paramètre un arbre de jetons.

Nous avons relevé deux scénarios typiques d'utilisation de la fonction de lecture d'un objet complexe.

Dans le premier scénario, un programme tel que FLOT recherche un jeton particulier dans un fichier, par exemple VISCOSITY. Il sait a priori que ce jeton est une courbe. Il appelle donc `ReadCurve()` après avoir localisé le jeton avec `FindToken()` :

```
List<IONode> list = CreateParsingTree("test.dtf");
IONode node = FindToken(&list, "VISCOSITY");
Curve *viscosity = ReadCurve(&node);
```

Un deuxième scénario consiste à récupérer dans un fichier tous les objets d'une certaine famille, sans se soucier de leur jeton. C'est dans cette situation qu'intervient la fonction `FindFamilyMembers()` décrite plus haut. Par exemple, un éditeur de courbes permet le chargement de toutes les courbes contenues dans un fichier, sans considérer leur jeton. On implémente cette fonctionnalité avec un code semblable à celui-ci :

```
List<IONode> list = CreateParsingTree("test.dtf");
List<IONode>* famList = FindFamilyMembers(&list, "Curve");

for (int i=0; i<famList->getSize(); i++) {
   IONode node = (*famList)[i];
   Curve *curve = ReadCurve(&node);
   ...
}
```

2.4.3 Enregistrement d'objets complexes dans un fichier texte

Nous disposons d'un ensemble de fonctions d'écriture pour chacun des types élémentaires du langage C :

- `IONode WriteInt(char *token, int value);`
- `IONode WriteFloat(char *token, float value);`
- `IONode WriteString(char *token, char* value);`

Ces fonctions construisent une instance de la structure `IONode` et initialisent l'attribut `IONode::value` à l'aide de la fonction `sprintf()`.

De la même façon, on définit une fonction d'écriture pour les super-classes d'objets complexes :

- `IONode* WriteCurve(char *token, Curve *value);`

- `IONode* WriteSurface(char *token, Surface *value);`

Un objet participant au protocole de lecture/écriture de Dataflot doit définir l'opération `write()`. Cette opération retourne une liste des jetons constituant la description de l'objet complexe. Voici un exemple d'implémentation de `write()` pour la classe `CrvExponential` :

```
List<IONode>* CrvExponential::write() {

   List<IONode> *list = new List<IONode>;
   IONode nodeFamily, nodeType, nodeA, nodeB;

   // construction des jetons simples
   nodeFamily = WriteString("FAMILY", "Curve");
   nodeType = WriteString("TYPE", "Exponential");
   nodeA = WriteFloat("A", myCoefA);
   nodeB = WriteFloat("B", myCoefB);
   // insertion des jetons simples en fin de liste
   list->insertEnd(nodeFamily);
   list->insertEnd(nodeType);
   list->insertEnd(nodeA);
   list->insertEnd(nodeB);

   return list;

}
```

L'implémentation de `WriteCurve()` consiste simplement à construire une instance de `IONode`, puis à assigner le résultat de l'opération `write()` de l'objet reçu en paramètre au champ `IONode::content` :

```
IONode* WriteCurve(char *token, CrvCurve* curve) {

   IONode* headNode = new IONode;
   headNode->token = strdup(token);
```

```
    List<IONode>* list = curve->write();
    headNode->content = list;

    return headNode;

}
```

Ainsi, à partir d'une instance d'un objet complexe, une surface par exemple, on obtient sa représentation sous forme d'arbre par un simple appel à `WriteSurface()`. Cet arbre est ensuite facilement converti en chaîne de caractères et enregistré dans un fichier.

2.5 Points de contrôle

La classe `CrvPoint` définit un point de contrôle. Les points de contrôle qu'utilisent les courbes et les surfaces sont des points à quatre dimensions, la quatrième composante du point servant de coordonnée homogène (elle est toutefois inutilisée jusqu'à maintenant). On définit des fonctions de lecture et de modification pour chacune des composantes.

---

classe :

`CrvPoint`

rôle :

contenir les quatre composantes de la position d'un point en coordonnées homogènes.

principales méthodes :

- `getX, getY, getZ, getW :`
  fonctions de lecture.
- `setX, setY, setZ, setW :`
  fonctions de modification.
- `translate :`
  déplacement relatif du point de contrôle.

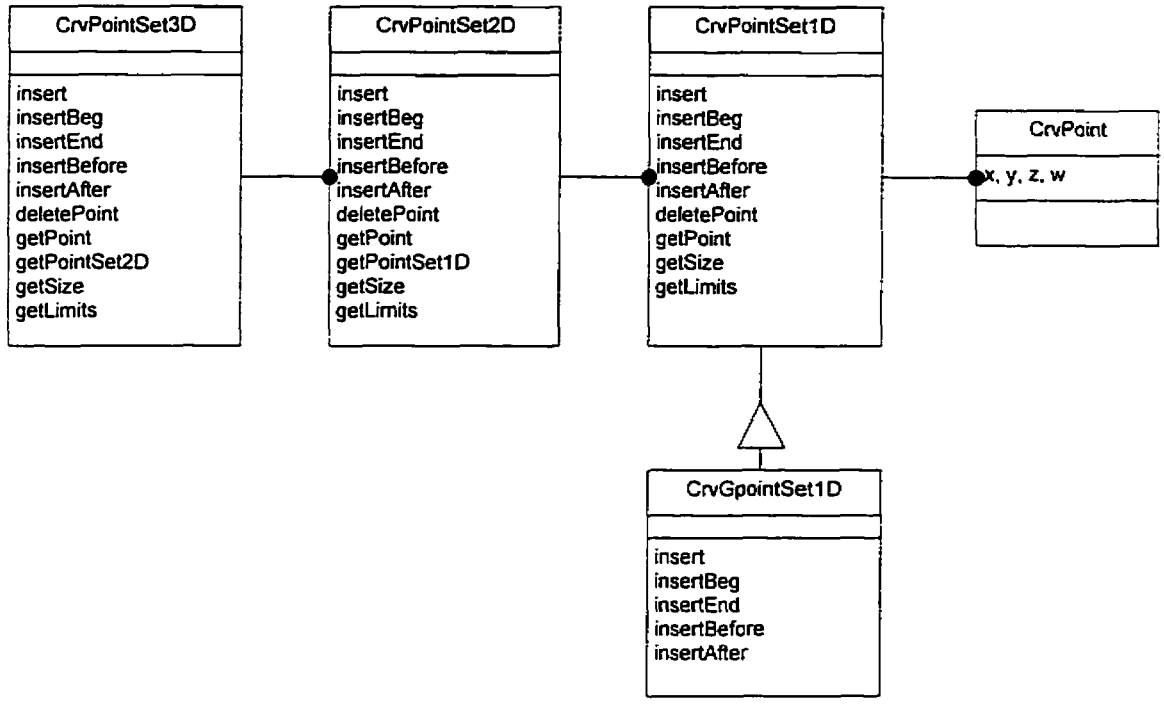


Figure 2.5 Diagramme d'objets décrivant les ensembles de points de contrôle

2.6 Ensembles de points de contrôle

Certaines courbes sont construites à partir d'un ensemble de points de contrôle. Les courbes krigées en constituent un exemple. Ces courbes sont associées à une classe dont le seul rôle est de contenir des points de contrôle.

La figure 2.5 montre le diagramme d'objets des ensembles de points de contrôle. La classe `CrvPointSet1D` est la classe mère des ensembles de points qu'utilisent les courbes. On peut voir cette classe comme étant un ensemble adressable en une

dimension, c'est-à-dire qu'on accède à un point en spécifiant un seul indice. Les fonctionnalités habituelles d'un ensemble (ajout d'un élément, retrait, etc.) peuvent être implémentées en utilisant une liste chaînée ou un tableau. En pratique, nous utilisons la liste chaînée, qui est moins performante en terme de temps d'accès, mais qui simplifie l'implémentation.

Voici la liste des principales opérations supportées par `CrvPointSet1D` :

---

classe :

`CrvPointSet1D`

rôle :

contenir des points de contrôle.

principales méthodes :

- `insert` :
  ajout d'un point de contrôle.
- `insertBeg, insertEnd` :
  insertion en tête et en queue de liste.
- `insertBefore, insertAfter` :
  insertion d'un point avant/après un point spécifique.
- `deletePoint` :
  retrait d'un point.
- `getPoint` :
  retourne une instance de `CrvPoint` spécifique.
- `getSize` :
  retourne le nombre de points dans la liste.
- `getLimits` :
  retourne les limites de la boîte contenant tous les points.

Certaines courbes comme les courbes linéaires par morceaux requièrent que leurs points de contrôle soient ordonnés par rapport à la variable indépendante, afin d'optimiser

l'évaluation de la fonction en un point. C'est pourquoi nous avons créé la classe `CrvGpointSet1D`. Cette classe ressemble beaucoup à `CrvPointSet1D`, sauf pour ses méthodes d'insertion d'un point, qu'elle redéfinit de façon à ce que la condition suivante soit toujours vérifiée : $x_{i+1} > x_i$.

Toutes les courbes paramétriques utilisent la classe `CrvPointSet1D`, tandis que les courbes explicites utilisent `CrvGpointSet1D`.

Certaines méthodes de `CrvPointSet1D` requièrent l'identificateur du point concerné. C'est la cas, par exemple, de l'insertion d'un point *devant* un autre point, ou du retrait d'un point. Ces méthodes supportent toutes deux types d'identification du point concerné : directe et indirecte. La méthode directe nécessite la connaissance de la position précise du point concerné dans la liste. En pratique, le client connaît rarement cette position. Par exemple, `CrvGpointSet1D` modifie l'ordre des points à chaque insertion. La méthode indirecte consiste à passer un pointeur à l'objet concerné. On détermine indirectement la position du point dans la liste par comparaison avec chaque élément. Par exemple, `deletePoint(int pos)` retire le point à la position `pos`, tandis que `deletePoint(CrvPoint *point)` doit d'abord trouver `point` dans la liste.

La généralisation à deux dimensions de `CrvPointSet1D` est `CrvPointSet2D`. Il s'agit essentiellement d'une liste de listes de points de contrôle, autrement dit une liste de `CrvPointSet1D`.

- **classe** :
- `CrvPointSet2D`
- **rôle** :
- contenir des instances de `CrvPointSet1D`.
- **principales méthodes :**
- `insert` :
  ajout d'un ensemble 1D de points de contrôle.
- `insertBeg, insertEnd` :
  insertion en tête et en queue de liste d'un ensemble 1D de points de contrôle.
- `insertBefore, insertAfter` :
  insertion d'un ensemble de points de contrôle 1D avant/après celui contenant un point particulier.
- `deletePointSet1D` :
  retrait d'une instance de `CrvPointSet1D`.
- `getPointSet1D` :
  retourne une instance de `CrvPointSet1D`.
- `getPoint` :
  retourne une instance spécifique de `CrvPoint`.
- `getSize` :
  retourne le nombre d'instances de `CrvPointSet1D` dans la liste.
- `getLimits` :
  retourne les limites d'une boîte contenant tous les points.

Toutes les surfaces paramétriques utilisent la classe `CrvPointSet2D`. Les surfaces explicites krigées sont définies à partir de points non structurés dans le plan XY. C'est pourquoi elles utilisent la classe `CrvPointSet1D`. Les surfaces explicites nécessitant des points de contrôle à structure 2D devraient utiliser `CrvPointSet2D`. Cependant, nous n'avons pas eu à développer de telles surfaces.

La classe `CrvPointSet3D` implémente un ensemble de points de contrôle de structure tridimensionnelle. Il s'agit en fait d'une liste de `CrvPointSet2D`. On adresse donc un point de contrôle avec trois indices. Bien que nous ayions implémenté cette classe, elle n'est pas utilisée pour l'instant dans le projet. Elle pourra servir, entre autres, pour le krigeage de solides paramétriques.

---

classe :

`CrvPointSet3D`

rôle :

contenir des instances de `CrvPointSet2D`.

principales méthodes :

- `insert` :
  ajout d'une matrice de points de contrôle.
- `insertBeg, insertEnd` :
  insertion en tête et en queue de liste d'une matrice de points de contrôle.
- `insertBefore, insertAfter` :
  insertion d'une matrice de points de contrôle avant/après celle contenant le point spécifié.
- `deletePointSet2D` :
  retrait d'une instance de `CrvPointSet2D`.
- `getPointSet2D` :
  retourne une instance de `CrvPointSet2D`.
- `getPoint` :
  retourne une instance de `CrvPoint`.
- `getSize` :
  retourne le nombre d'instances de `CrvPointSet2D` dans la liste.

- `getLimits` :
  retourne les limites de la boîte contenant tous les points.

2.7 Courbes

2.7.1 Conception

La figure 2.6 donne une vue d'ensemble des classes impliquées dans la définition d'une courbe. La classe mère de toutes les courbes se nomme `CrvCurve`. C'est une classe abstraite qui spécifie les opérations communes à toutes les courbes :

classe :

`CrvCurve`

rôle :

spécifier le prototype des opérations que toutes les courbes doivent définir.

principales méthodes :

- `evaluate` :
  retourne la valeur de la fonction en un point.
- `updatePlotCoords` :
  mise à jour des coordonnées d'affichage pour le domaine spécifié.
- `getPlotCoords` :
  retourne un tableau de coordonnées (x, y, z) utilisées pour l'affichage de la courbe.
- `fit` :
  post-construction de la courbe à partir d'un tableau de coordonnées (x,y,z). Cette opération n'est pas disponible sur toutes les classes dérivées.
- `fitWithCtrlPts` :
  post-construction de la courbe à partir de ses points de contrôle. Cette opération n'est pas disponible sur toutes les classes dérivées.
- `getSimpleEditionCoefs` :

retourne un tableau de coefficients décrivant la courbe. Cette opération n'est pas disponible sur toutes les classes dérivées.

- `resetWithSimpleEditionCoefs` :
  post-construction de la courbe à partir d'un tableau de coefficients.
- `getLimits` :
  retourne les coordonnées d'une boîte contenant la courbe pour un domaine spécifié. Méthode utilisée par les classes de visualisation.

2.7.1.1 Évaluation d'une fonction d'une variable indépendante

Une application de calcul n'a qu'une seule opération à connaître pour utiliser une courbe. Il s'agit d'`evaluate()`, dont la signature exacte est la suivante :

```
int CrvCurve::evaluate(const float* indVar, int nbIndVar,
float* depVar, int order);
```

Il est possible qu'une courbe sache comment optimiser `evaluate()` lorsque le calcul est requis simultanément sur plusieurs points. Voilà pourquoi `evaluate()` reçoit un *tableau* de variables indépendantes. Évidemment, ce tableau peut contenir une seule valeur, pour les cas les plus courants où le client requiert une évaluation ponctuelle. Le paramètre `indVar` pointe au premier élément d'un tableau de variables indépendantes. Le paramètre `nbIndVar` est le nombre de valeurs que contient ce tableau.. Le résultat de l'évaluation est retourné dans `depVar`. Le paramètre `order` prend les valeurs 0, 1 ou 2, selon que le client demande l'évaluation de la fonction, de sa dérivée ou de sa dérivée seconde.

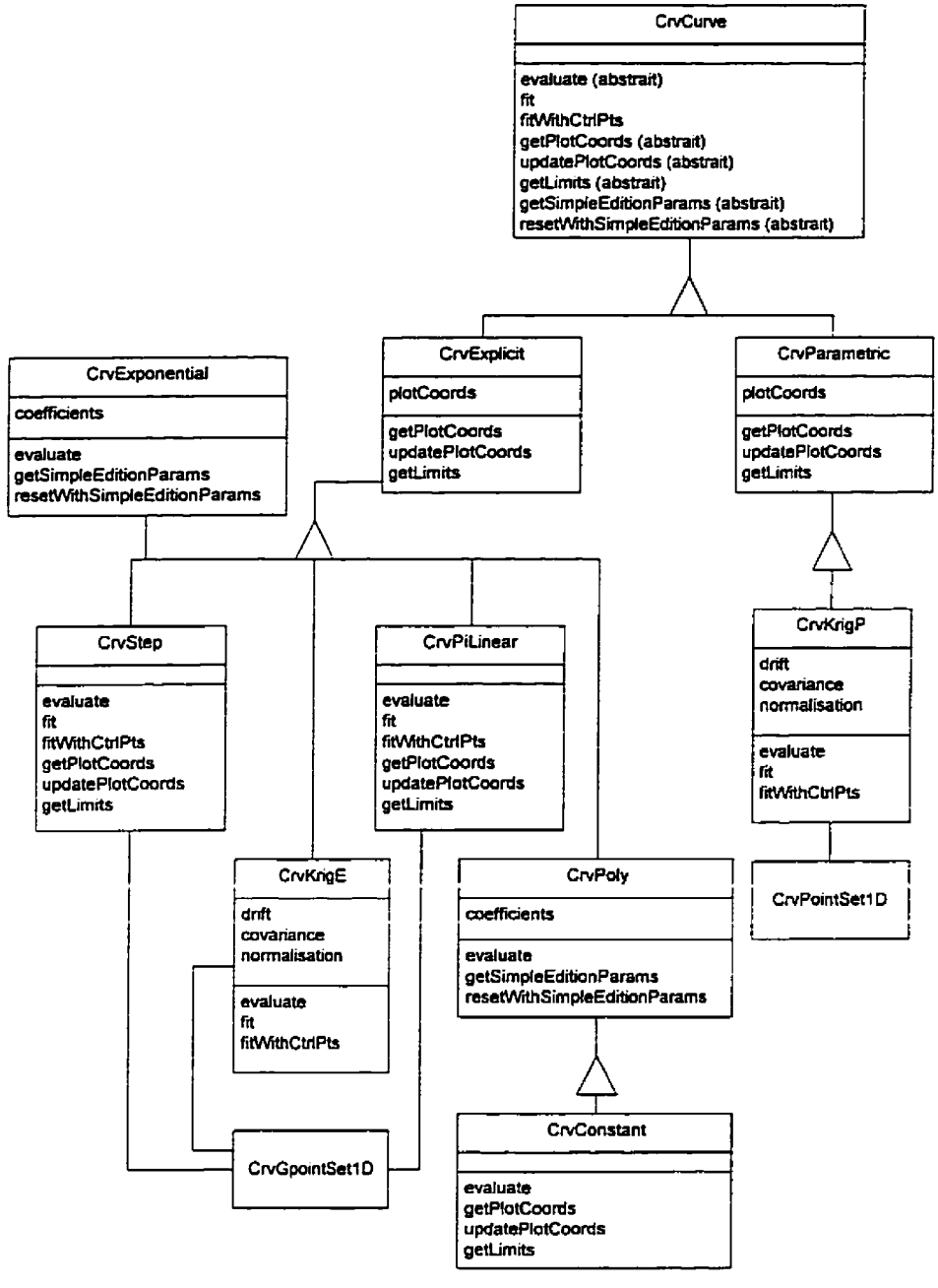


Figure 2.6 Modèle objet de la bibliothèque de courbes

Quelques précisions sont nécessaires sur le tableau de retour `depVar`. Le client passe à `evaluate()` l'adresse du premier élément d'un tableau de `float`. Le client est responsable de l'allocation de l'espace mémoire adéquat, sur le "stack"ou dans le "heap". La taille du tableau dépend de la classe de la courbe : explicite ou paramétrique. Une courbe explicite écrit `nbIndVar float` dans ce tableau. Une courbe paramétrique écrit *toujours* `3*nbIndVar float`, peu importe que la courbe paramétrique soit plane ou non.

Voilà, à peu de choses près, tout ce qu'un développeur d'une application de calcul doit savoir pour utiliser une courbe dérivée de la classe `CrvCurve`. Toutes les autres opérations définies sur une courbe ne sont utiles que dans le contexte d'applications graphiques d'édition interactive.

2.7.1.2 Édition "simple" d'une courbe

Certaines courbes ont la caractéristique d'être définies *seulement* par des coefficients constants. Par exemple, une courbe polynômiale de degré 2 : $y(x) = c_0 + c_1 x + c_2 x^2$. D'autres courbes calculent leurs coefficients à partir de points de contrôle (une fonction spline, par exemple). Supposons qu'un utilisateur d'un logiciel interactif souhaite modifier les coefficients du premier type de courbe. C'est ce que nous appelons le mode d'édition *simple* d'une courbe. Le programme d'édition demande à la courbe un tableau de valeurs numériques (ses coefficients) ainsi que l'information textuelle décrivant chaque coefficient. Il affiche cette information dans une boîte de dialogue. Lorsque

l'usager a terminé l'édition, le programme transmet à la courbe ses nouveaux coefficients.

L'opération `handlesSimpleEdition()` permet à l'application de savoir si une courbe supporte le mode d'édition simple. Dans l'affirmative, l'opération `getSimpleEditionCoefs()` est valide :

```
void CrvCurve::getSimpleEditionCoefs(char ***labels, float
**values, int *nbCoefs);
```

Le paramètre `values` retourne les valeurs numériques des coefficients, `labels` une description textuelle, et `nbCoefs` le nombre de coefficients.

L'opération `resetWithSimpleEditionCoefs()` permet de transférer à une courbe ses nouveaux coefficients :

```
void CrvCurve::resetWithSimpleEditionCoefs(float *values);
```

Il n'est pas nécessaire de transmettre le nombre de coefficients contenus dans le tableau `values`. La courbe "sait" ce qu'elle doit y trouver. Le nombre de coefficients et l'ordre dans lequel ils se présentent doit être consistant avec `getSimpleEditionParams()`.

2.7.1.3 Représentation graphique

Un aspect important de l'architecture retenue concerne la représentation graphique d'une courbe. Par représentation graphique, nous entendons un ensemble de points caractéristiques d'une courbe, que l'on relie par des segments de droite pour la dessiner. Certaines courbes ont une représentation graphique exacte : une courbe linéaire par morceaux, par exemple. En général, une courbe n'a pas de représentation graphique exacte. On la discrétise alors en un grand nombre de segments de droite.

La représentation graphique d'une courbe dépend du domaine que l'on souhaite visualiser. On modifie cette représentation par un appel à :

```
void CrvCurve::updatePlotCoords(float domainMin, float
domainMax, int resolution);
```

Les paramètres `domainMin` et `domainMax` spécifient le domaine graphique d'intérêt, tandis que `resolution` spécifie le nombre de segments de droite approximant la courbe (notons qu'il s'agit du nombre de *segments* et non du nombre de points).

Lorsqu'une courbe a une représentation exacte, elle redéfinit `updatePlotCoords()`, `getPlotCoords()` et `getLimits()`. Les seules classes à redéfinir ces méthodes jusqu'à maintenant sont : `CrvConstant`, `CrvPiLinear`, `CrvStep` et `CrvPoly` (pour les cas particuliers où le degré du polynôme est 0 ou 1).

Une classe client accède à la représentation graphique d'une courbe avec `getPlotCoords()`. Nous reviendrons au chapitre 4 sur le protocole d'affichage d'une courbe.

2.7.1.4 Courbes définies par des points de contrôle

Une courbe définie par des points de contrôle reçoit un tableau de coordonnées dans son constructeur. Elle crée ensuite l'ensemble de points de contrôle qui lui convient. S'il s'agit d'une courbe explicite, elle crée une instance de `CrvGpointSet1D`. S'il s'agit d'une courbe paramétrique, elle crée une instance de `CrvPointSet1D`.

Par exemple, la définition du constructeur d'une courbe linéaire par morceaux est la suivante :

```
CrvPiLinear::CrvPiLinear(float *coords, int nbCoords, int dim) {
   mySet = new CrvGpointSet1D(coords, nbCoords, dim);
   fitWithCtrlPts();
}
```

Certaines opérations de `CrvCurve` ne sont valides que sur une courbe définie à partir de points de contrôle. C'est le cas des opérations `fit()` et `fitWithCtrlPts()`. La signature de `fitWithCtrlPts()` est :

```
void CrvCurve::fitWithCtrlPts();
```

Cette opération a l'effet d'un post-constructeur. Elle reconstruit la courbe à partir des points de contrôle spécifiés. Par exemple, une courbe krigée résoud le système linéaire

du krigeage (Trochu, 1993) à partir des points de contrôle courants lorsqu'elle reçoit le message `fitWithCtrlPts()`. Cette opération s'utilise surtout dans le contexte d'une application interactive : par exemple, l'éditeur de courbes déplace un point, puis il informe la courbe qui dépend de ce point que celle-ci doit se reconstruire.

L'opération `fit()` possède la même sémantique que `fitWithCtrlPts()`. Cependant, elle reconstruit la courbe à partir des points passés en paramètre plutôt qu'à partir des points courants. L'opération `fit()` ne fait que regénérer l'ensemble de points de contrôle, puis elle appelle `fitWithCtrlPts()`, comme le montre l'extrait de code suivant :

```
void CrvPiLinear::fit(float *coords, int nbCoords, int
dimension) {

   myPointSet->resetWithCoordBuf(coords, nbCoords, dimension);
   fitWithCtrlPts();

}
```

Les deux sous-classes directes de `CrvCurve` sont `CrvExplicit` et `CrvParametric`. Le rôle principal de ces deux classes est de fournir certaines méthodes par défaut. Par exemple, la méthode `updatePlotCoords()` de `CrvExplicit` construit les points caractéristiques de la courbe par une simple évaluation de la fonction sur plusieurs points. La position de ces points discrétise le domaine en intervalles d'égale longueur. Si la courbe possède des propriétés graphiques particulières que seule la classe dérivée connaît, celles-ci ne seront pas visibles. La

figure 2.7 illustre cette situation. La représentation graphique générée par `CrvExplicit::updatePlotCoords()` approxime la représentation graphique exacte. Dans cet exemple, on voit que la résolution spécifiée n'est pas compatible avec la courbe réelle (en pointillés), ce qui mène à une fausse représentation graphique. En augmentant la résolution, on s'approche de la courbe réelle, mais au prix de nombreuses évaluations de la fonction.

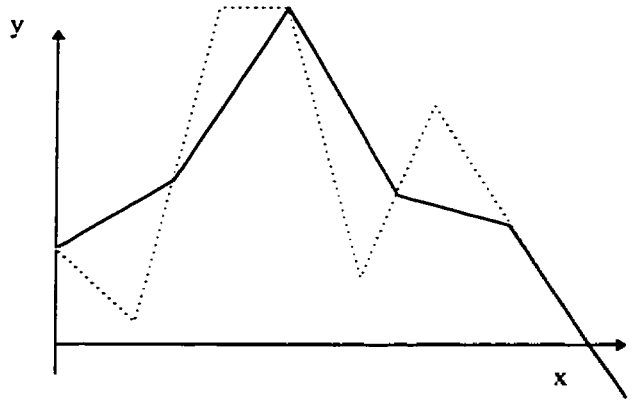


Figure 2.7 Problème de résolution incompatible avec la représentation graphique d'une courbe

2.7.2 Implémentation

2.7.2.1 Courbe polynômiale

La classe `CrvPoly` définit un polynôme de degré n

$$y(x) = c_0 + c_1 x + c_2 x^2 + ... + c_n x^n .$$

On utilise l'algorithme de Horner (Sedgewick, 1990) pour minimiser le nombre de multiplications requises par l'évaluation. Par exemple, le polynôme

$p(x) = x^4 + 2x^3 - 3x^2 + 4x + 1$ peut être écrit sous la forme plus compacte

$p(x) = x(x(x(x+2)-3)+4)+1$.

Le code C suivant, utilisé pour évaluer un polynôme de degré N, implémente l'algorithme de Horner :

Soient p un tableau de N+1 coefficients où p[0] = c_0 et p[N] = c_n, x la variable indépendante et y la variable dépendante.

```
y = p[N];
for (i = N; i>=0; i--) {
   y = x*y + p[i];
}
```

Cet algorithme nécessite N-1 multiplications et N additions.

Dans le cas où tous les coefficients du polynôme sauf un sont nuls, l'évaluation revient simplement à élever un nombre à une certaine puissance. L'utilisation de la méthode de Horner, dans ce cas, nécessite N-1 multiplications. Il est possible de faire beaucoup mieux. Par exemple, le polynôme $p(x) = x^{55}$ se réduit à $p(x) = x^{32}x^{16}x^4x^2x$, ce qui nécessite seulement 8 multiplications. 54 multiplications auraient été nécessaires avec la méthode de Horner. La classe `CrvPoly` implémente cette optimisation avec l'algorithme suivant, inspiré de Sedgewick (1990) :

Soit `degree` le degré de l'exponentiation requise et `x` la variable indépendante. On calcule la variable dépendante y avec :

```
float accum = 1.;
for (int i=7; i>=0; i--) {
   accum *= accum;
   if ((1 << i) & degree) accum *= x;
}
y = accum;
```

Le degré de l'exponentiation doit être inférieur à 256.

2.7.2.2 Courbe constante

La classe `CrvConstant` dérive de `CrvPoly`. Elle redéfinit `evaluate()`, afin de l'optimiser. La méthode `evaluate()` retourne, sans effectuer un seul test sur la validité de la courbe, la valeur du coefficient constant du polynôme.

2.7.2.3 Courbe linéaire par morceaux

La classe `CrvPiLinear` définit une courbe linéaire par morceaux (figure 2.8). L'algorithme d'évaluation de cette courbe est une optimisation par rapport à la courbe krigée, qui supporte le même type d'interpolation. Le nombre de multiplications à effectuer est ici beaucoup moins important que pour une courbe krigée.

Étant donnés n+1 points de contrôle (x_0, y_o) ... (x_n, y_n) ordonnés de façon telle que $x_{i+1} \geq x_i$, l'algorithme utilisé pour évaluer la fonction est le suivant :

1. Si $x_0 < x < x_n$, trouver les points de contrôle (x_i, y_i) (x_{i+1}, y_{i+1}) tels que $x_i \leq x \leq x_{i+1}$

2. La valeur interpolée y s'obtient par :

$$y = \frac{(y_1 - y_0)}{(x_1 - x_0)}(x - x_0) + y_0 \qquad \text{si } x \leq x_0$$

$$y = \frac{(y_{i+1} - y_i)}{(x_{i+1} - x_i)}(x - x_i) + y_i \qquad \text{si } x_0 < x < x_n$$

$$y = \frac{(y_n - y_{n-1})}{(x_n - x_{n-1})}(x - x_n) + y_n \qquad \text{si } x \geq x_n$$

2.7.2.4 Courbe constante par morceaux

La classe `CrvStep` définit une courbe constante par morceaux (figure 2.9). L'algorithme utilisé pour évaluer cette fonction est le suivant :

Étant donnés n+1 points de contrôle (x_0, y_0) ... (x_n, y_n) ordonnés de façon telle que $x_{i+1} > x_i$, et une coordonnée x où l'on souhaite évaluer la fonction, la valeur interpolée y s'obtient par :

$$y = y_0 \qquad \text{si } x \leq x_0$$

$$y = y_i \qquad \text{si } x_i < x \leq x_{i+1}$$

$$y = y_n \qquad \text{si } x > x_n$$

2.7.2.5 Krigeage dual 1D explicite

La classe `CrvKrigE` est une interface à la bibliothèque de krigeage *LIBKRIG*.

La méthode `fitWithCtrlPts()` crée un objet `SampSet` de *LIBKRIG*. Puis elle effectue le krigeage du champ scalaire 1D (krigeage *géométrique* dans la terminologie du krigeage), avec la fonction `KG_KrigField()`. L'interpolation se fait par un appel à la fonction `KG_InterpolFieldOnPointR()`.

2.7.2.6 Krigeage dual 1D paramétrique

La classe `CrvKrigP` implémente le krigeage de courbes paramétriques en utilisant, elle aussi, *LIBKRIG*. Le krigeage de la courbe se fait avec `KG_KrigCurve()`. Pour interpoler un point sur la courbe, on utilise `KG_InterpolCurveR()`.

2.7.2.7 Courbe exponentielle

La classe `CrvExponential` implémente la fonction

$$f(x) = Ae^{Bx},$$

où A et B sont des coefficients constants.

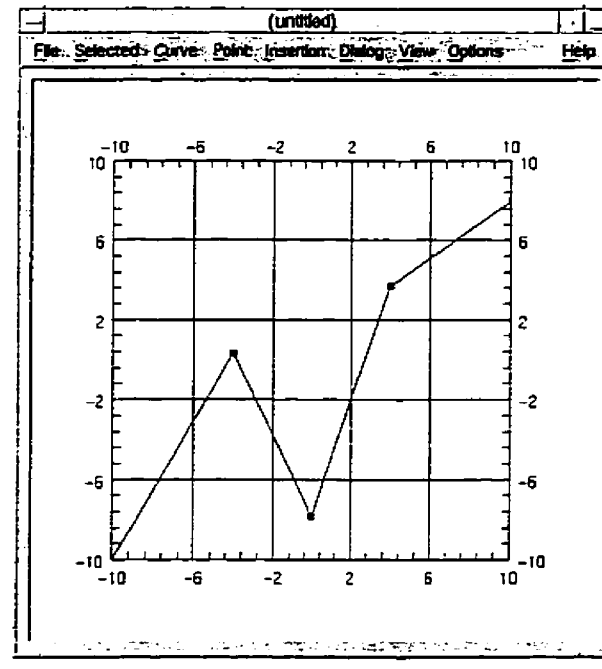


Figure 2.8 Courbe linéaire par morceaux

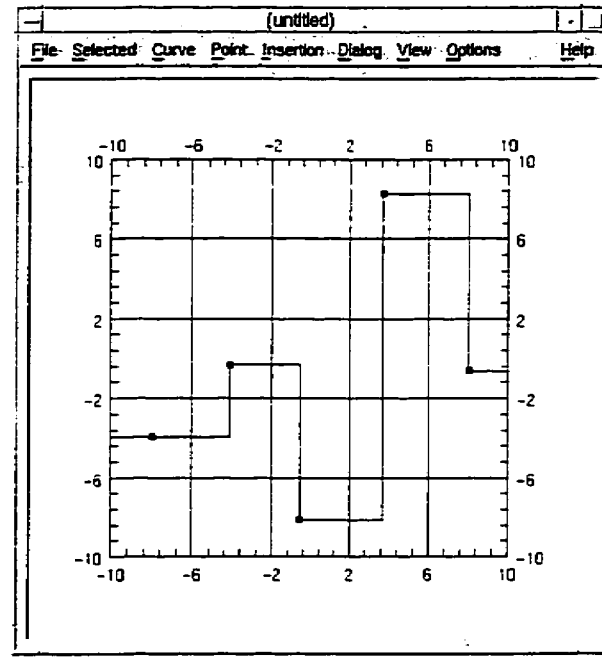


Figure 2.9 Courbe constante par morceaux

2.8 Surfaces

2.8.1 Conception

La classe mère de toutes les surfaces se nomme `SrfSurface`. C'est une classe abstraite qui spécifie les opérations supportées par toutes les surfaces. Essentiellement, `SrfSurface` généralise les opérations de `CrvCurve`. La figure 2.10 donne une vue d'ensemble des classes impliquées dans la bibliothèque de surfaces. Les opérations définies sur une surface sont :

---

classe :

`SrfSurface`

rôle :

spécifier le prototype des méthodes que toutes les surfaces doivent définir.

principales méthodes :

- `evaluate` :
  retourne la valeur de la fonction en un point.
- `updatePlotCoordsIso` :
  construction d'un tableau de points pour la représentation graphique isoparamétrique.
- `updatePlotCoordsSolid` :
  construction d'un tableau de points pour la représentation graphique maillée.
- `updatePlotCoordsSolidSmooth` :
  même chose que `updatePlotCoordsSolid()` avec en plus l'évaluation des normales aux sommets.
- `updatePlotCoordsSolidFlat` :
  même chose que `updatePlotCoordsSolid()` avec en plus l'évaluation des normales au milieu des faces.
- `getPlotCoordsIso, getPlotCoordsSolid, getPlotCoordsSolidFlat, getPlotCoordsSolidSmooth` :
  retourne la représentation graphique construite par l'update correspondant.

- `fit` :
  post-construction de la surface à partir d'une matrice de points de contrôle.
- `fitWithCtrlPts` :
  post-construction de la surface à partir de ses points de contrôle.
- `handlesSimpleEdition` :
  retourne VRAI si la suface supporte le mode d'édition simple.
- `getSimpleEditionCoefs` :
  retourne un tableau de coefficients décrivant la surface.
- `resetWithSimpleEditionCoefs` :
  post-construction de la surface à partir d'un tableau de coefficients.
- `getLimits` :
  retourne les coordonnées d'une boîte contenant la surface, sur un domaine spécifié. Opération requise par le visualisateur de surfaces.

2.8.1.1 Évaluation d'une fonction de deux variables

La signature d'`evaluate()` sur une fonction de deux variables indépendantes est la suivante :

```
int evaluate(const float *indVar, int nbIndVar, float
*depVar, int orderX, int orderY);
```

La variable `indVar` contient les points sur lesquels la fonction est évaluée. Encore une fois, pour qu'une méthode spécifique puisse optimiser `evaluate()`, on lui passe un *tableau* de points. Le nombre de `float` que contient `indVar` est 2*`nbIndVar`. Le tableau 2.1 résume les combinaisons valides des paramètres `orderX` et `orderY`.

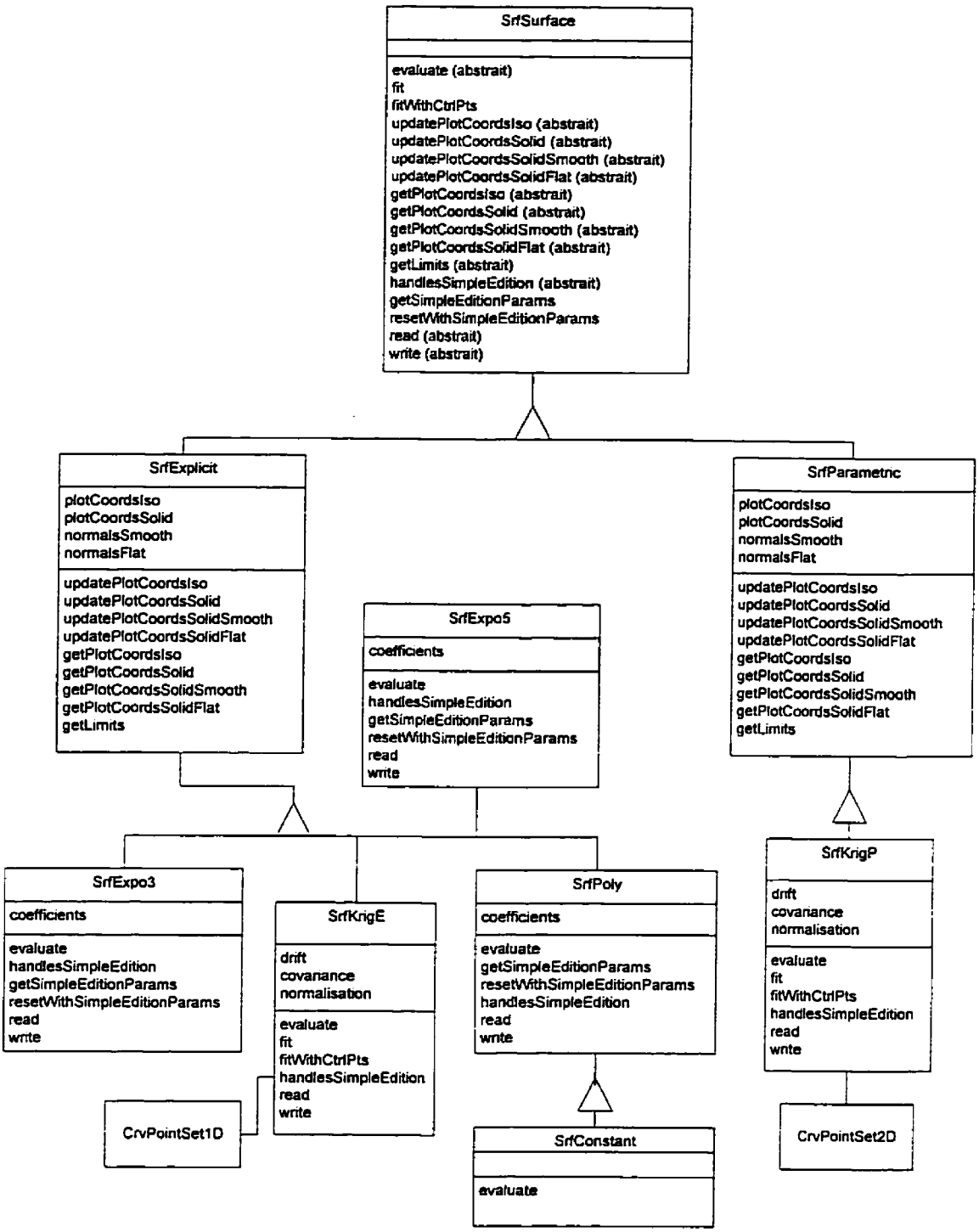


Figure 2.10 Modèle objet de la bibliothèque de surfaces

orderX	orderY	interprétation
0	0	f
1	0	$\frac{\partial f}{\partial x}$
0	1	$\frac{\partial f}{\partial y}$
2	0	$\frac{\partial^2 f}{\partial x^2}$
0	2	$\frac{\partial^2 f}{\partial y^2}$
1	1	$\frac{\partial^2 f}{\partial x \partial y}$

Tableau 2.1 Interprétation des paramètres orderX et orderY dans l'évaluation d'une surface.

2.8.1.2 Édition "simple" d'une surface

Lorsqu'une surface est décrite par une série de coefficients constants, elle possède la propriété d'édition "simple". C'est ce que signale la valeur de retour de l'opération `handlesSimpleEdition()`. Lorsque celle-ci retourne VRAI, `getSimpleEditionCoefs()` et `resetWithSimpleEditionCoefs()` sont valides (voir le paragraphe 2.7.1.2).

2.8.1.3 Représentation graphique

Une classe dérivée de `SrfSurface` contient et sait comment générer sa propre représentation graphique. Il s'agit d'une matrice de points caractéristiques de la surface, qu'on utilise pour la dessiner. Dans ce paragraphe, nous nous contentons d'expliciter la

signature des opérations touchant à la représentation graphique d'une surface. On verra au chapitre 4 comment utiliser cette information pour dessiner une surface.

Toutes les surfaces doivent supporter deux types de représentation graphique, isoparamétrique et maillée :

1. La *représentation isoparamétrique* consiste à tracer sur la surface un réseau de courbes, en maintenant constante, tour-à-tour, une des deux variables indépendantes. Ce mode de représentation est donc autant valable pour les surfaces paramétriques qu'explicites. Dans un cas on quadrille le plan X-Y, dans l'autre le plan U-V. L'opération responsable de la mise-à-jour de la représentation isoparamétrique d'une surface est `updatePlotCoordsIso()` :

   ```
   void updatePlotCoordsIso(float minX, float maxX, float
   minY, float maxY, int nbX, int nbY, int resoX, int
   resoY);
   ```

   Les paramètres `minX`, `maxX`, `minY`, `maxY` définissent le domaine graphique à visualiser, tandis que `nbX` et `nbY` spécifient le nombre de courbes iso-X et iso-Y à générer. Les paramètres `resoX` et `resoY` spécifient le nombre de segments de droite approximant les courbes iso-X et iso-Y.

2. La *représentation maillée* est fondée sur la discrétisation de la surface en petits triangles. On construit ces triangles à partir de l'évaluation de la fonction sur une

grille rectangulaire de points. La normale à la surface est importante si on utilise un modèle d'illumination tel que Gouraud (Watt, 1989, 1992). Très souvent, afin de réduire le temps d'affichage, on se contente d'un modèle d'illumination iso-intensité ("flat shading"). Dans ce cas, nous avons besoin d'une seule normale par triangle. C'est pourquoi les trois opérations suivantes sont disponibles :

- `void updatePlotCoordsSolid(float minX, float maxX, float minY, float maxY, int resoX, int resoY);`
- `void updatePlotCoordsSolidSmooth(float minX, float maxX, float minY, float maxY, int resoX, int resoY);`
- `void updatePlotCoordsSolidFlat(float minX, float maxX, float minY, float maxY, int resoX, int resoY);`

Ces trois opérations reconstruisent l'information graphique de la surface en évaluant la fonction sur une grille définie par `minX`, `maxX`, `minY`, `maxY`. On échantillonne au total `resoX*resoY` points sur la surface. La méthode `updatePlotCoordsSolidSmooth()` évalue la fonction ainsi que la normale à la surface, en chaque point, tandis que `updatePlotCoordsSolidFlat()` évalue la fonction en chaque point et approxime la normale au barycentre du triangle par un simple produit vectoriel.

Un objet client a quatre méthodes d'accès à la représentation graphique d'une surface : `getPlotCoordsIso()`, `getPlotCoordsSolid()` ,

`getPlotCoordsSolidSmooth()` et `getPlotCoordsSolidFlat()`. On doit s'assurer que l' `update()` approprié ait été fait avant d'appeler `getPlotCoords()`. Nous reviendrons au chapitre 4 sur ces opérations, en expliquant la structure de l'information retournée et son utilisation dans le contexte d'OpenGL.

2.8.1.4 Surfaces définies par des points de contrôle

Toutes les surfaces définies par des points de contrôle reçoivent en paramètre, dans leur constructeur, une matrice de points. Par exemple, le constructeur de la classe `SrfKrigE` est :

```
SrfKrigE::SrfKrigE(float **rows, int *nbColumns, int
nbRows, int dimCtrlPts);
```

La paramètre `rows` est une matrice de points de contrôle. Les lignes de cette matrice n'ont pas nécessairement toutes la même longueur. La variable `nbColumns` spécifie, pour chaque ligne, le nombre de points qu'elle contient, tandis que `dimCtrlPts` indique le nombre de composantes spécifiées pour chaque point de contrôle. Les valeurs permises pour `dimCtrlPts` sont 2, 3 ou 4. Jusqu'à maintenant, seules les surfaces paramétriques krigées tirent profit de cette matrice de points à largeur variable.

Notons que les surfaces krigées explicites utilisent un ensemble de points de contrôle à structure 1D. La matrice passée au constructeur, dans ce cas, peut avoir indifféremment une ou plusieurs lignes.

L'opération `fit()` reçoit une matrice de points de contrôle dans le format que nous venons de décrire :

```
void SrfSurface::fit(float **rows, int *nbColumns, int
nbRows, int dimCtrlPts);
```

Elle post-construit l'objet de classe `CrvPointSet2D` lié à la surface en appelant la méthode suivante :

```
void CrvPointSet2D::resetWithCoordBuf(float **rows, int
*nbColumns, int nbRows, int dimCtrlPts);
```

Puis la méthode `fitWithCtrlPts()` utilise les points de contrôle courants pour post-construire la courbe.

2.8.2 Implémentation

2.8.2.1 Surface polynômiale

La classe `SrfPoly` définit une fonction polynômiale de deux variables indépendantes x et y. Son constructeur reçoit en paramètre les degrés en x et en y de la fonction, ainsi qu'un tableau de coefficients. La taille de ce tableau croît très vite avec le degré en x et y du polynôme. Par exemple, un polynôme de degré 3 en x et y possède 10 coefficients :

$$f(x,y) = a_0 + a_1x + a_2y + a_3x^2 + a_4xy + a_5y^2 + a_6x^3 + a_7x^2y + a_8xy^2 + a_9y^3$$

Les coefficients passés au constructeur doivent apparaître dans l'ordre implicite défini par cet exemple, le premier élément étant le terme constant du polynôme. Si un coefficient est nul, il doit apparaître malgré tout dans le tableau passé au constructeur. Par exemple, pour construire $f(x,y) = 2y + 5xy$, on transmet au constructeur de `SrfPoly` le tableau {0., 0., 2., 0., 5.}.

Voici un extrait de code implémentant l'évaluation de ce polynôme en un point (x, y) :

Soient M le degré du polynôme selon x et N le degré selon y. Le tableau `powX` contient M+1 éléments avec les puissances de x de 0 à M. `powY` est l'équivalent de `powX` pour la variable y. On peut écrire `powX[0]=powY[0]=1`. Le tableau `coefs` contient les coefficients du polynôme.

```
float sum = coefs[0];
int ind = 1;
for (int k=1; k <= max(m, n); k++) {
   for (int i=k; i >= 0; i--) {
      if (i <= m && (k-i) <= n) {
         sum += coefs[ind]*powX[i]*powY[k-i];
         ind++;
      }
   }
}
return sum;
```

Ce code n'est pas optimal quant au nombre de multiplications.

2.8.2.2 Surface constante

La classe `SrfConstant` dérive de `SrfPoly`. Elle redéfinit `evaluate()` pour éviter les quelques comparaisons qui seraient nécessaires dans

`SrfPoly::evaluate()`. Elle retourne directement le coefficient constant de la classe `SrfPoly`.

2.8.2.3 Krigeage dual 2D explicite

La classe `SrfKrigE` sert d'interface avec la librairie *LIBKRIG*. On effectue l'interpolation par l'appel de `KG_InterpolFieldOnPointR()`.

2.8.2.4 Krigeage dual 2D paramétrique

L'évaluation d'un point sur la surface se fait par un appel à la fonction `KG_InterpolSurfOnUVR()`.

2.8.2.5 Surfaces exponentielles

La classe `SrfExpo3` implémente la fonction exponentielle

$$f(x,y) = ae^{\left(\frac{b}{x}+\kappa y\right)},$$

où a, b et κ sont des coefficients constants.

La classe `SrfExpo5` implémente

$$f(x,y) = k_1 \cdot \left(e^{\frac{k_2}{x}}\right) \cdot \left(\frac{k_3}{k_3 - y}\right)^{k_4 + k_5 y},$$

où k_1, k_2, k_3, k_4 et k_5 sont des constantes.

2.9 Exemples de fichiers de courbes et de surfaces en format Dataflot

- Courbe linéaire par morceaux :

```
piecewise_linear_curve {
   FAMILY : "Curve";
   TYPE : "Piecewise_linear";
   CTRL_POINTS {
      FAMILY : "PointSet";
      TYPE : "PointSet1D_G";
      POINTS {
         NUMBER : 3;
         DIMENSION : 2;
         DATA : \
            0., 0., 1., 1., 2., 2.;
      }
   }
}
```

- Courbe krigée explicite :

```
kriged_explicit_curve {
   FAMILY : "Curve";
   TYPE : "Kriged_explicit";
   CTRL_POINTS {
      FAMILY : "PointSet";
      TYPE : "PointSet1D";
      POINTS {
         NUMBER : 3;
```

```
            DIMENSION : 2;
            DATA : \
               0., 0., 1., 1., 2., 2.;
         }
         VECTORS {
            NUMBER : 1;
            DIMENSION : 2;
            CONNECTION : \
               1;
            DATA : \
               1., 0.;
         }
      }
      DRIFT : "Linear";
      COVARIANCE : "Cubic";
      NORMALIZATION : "Position";
}
```

- Surface krigée explicite :

```
kriged_explicit_surface {

      FAMILY : "Surface";
      TYPE : "Kriged_explicit";
      CTRL_PTS {
         FAMILY : "PointSet";
         TYPE : "PointSet1D";
         POINTS {
            NUMBER : 3;
            DIMENSION : 3;
            DATA : \
               0., 0., 0., 1., 1., 1., 2., 2., 2.;
         }
         VECTORS {
            NUMBER : 1;
            DIMENSION : 2;
            CONNECTION : \
               1;
            DATA : \
               1., 0.;
         }
      }
      DRIFT : "Linear";
      COVARIANCE : "Cubic";
      NORMALIZATION : "Global";

}
```

- Surface krigée paramétrique :

```
kriged_parametric_surface {

    FAMILY : "Surface";
    TYPE : "Kriged_parametric";
    CTRL_POINTS {
        POINTS {
            DIMENSION : 3;
            ROWS : 2;
            COLUMNS : 2, 3;
            DATA : \
              0., 0., 0., 1., 1., 1., \  /* courbe v=0 */
              2., 2., 2., 3., 3., 3., 4., 4., 4.;  /* courbe v=1 */
        }
    }
    DRIFT_U : "Linear";
    COVARIANCE_U : "Cubic";
    NORMALIZATION_U : "Position";
    DRIFT_V : "Linear";
    COVARIANCE_V : "Cubic";
    NORMALIZATION_V : "Position";

}
```

2.10 Évaluation d'une fonction de N variables indépendantes

Il est possible de généraliser la bibliothèque de courbes et de surfaces de façon à pouvoir évaluer une fonction de N variables indépendantes. Il suffit de faire dériver les classes `SrfSurface` et `CrvCurve` d'une nouvelle classe baptisée `Function`. Cette classe spécifie l'opération `evaluate()` comme ceci :

```
int Function::evaluate(const float *indVar, float *depVar,
int *dimDepVar);
```

Cette signature s'utilise dans le contexte de l'évaluation de la fonction en un seul point. Il s'agit d'un point à N composantes (la méthode `evaluate()` n'a pas besoin de

connaître la valeur de N). La variable dépendante peut être de dimension quelconque. On connaît sa dimension exacte avec `dimDepVar`. Les valeurs calculées sont retournées dans le tableau `depVar`. Le client de cette opération est responsable de l'allocation de l'espace mémoire suffisant pour `depVar` (selon le domaine de l'application).

Une courbe explicite est un cas particulier d'une fonction de N variables, où N-1 variables sont ignorées dans le calcul de la fonction. La classe `CrvExplicit` devrait redéfinir l'opération `evaluate()` générale, de façon à appeler la méthode `evaluate()` de la courbe :

```
int CrvExplicit::evaluate(const float *indVar, float *depVar,
int *dimDepVar) {

   // on suppose que la premiere composante de indVar est celle
   // qui nous interesse
   evaluate(&indVar[0], 1, depVar);
   *dimDepVar = 1;

}
```

C'est ici qu'intervient la limitation fondamentale de cette classe. On ne sait pas quelle composante utiliser pour évaluer la fonction. Est-ce la première, la deuxième, la $N^{ième}$?

On doit donc mettre en correspondance les variables de la fonction. Par exemple, on utilise $f(x) = x^2$ pour en faire $f(x,y,z) = z^2$. On a recours à une *table de correspondance*. Dans cet exemple, la première entrée de la table de correspondance contient 3. Ceci indique à la fonction qu'elle doit remplacer la première variable par la

troisième. Voici la nouvelle `evaluate()` après introduction d'une table de correspondance :

```
CrvExplicit::evaluate(const float *indVar, float *depVar, int
*dimDepVar) {

   // corresp() est une méthode retournant l'indice de
   // correpondance d'une variable independante
   int ind = corresp(0);
   evaluate(&indVar[ind], 1, depVar);
   *dimDepVar = 1;

}
```

Pour une surface explicite, on aurait :

```
SrfExplicit::evaluate(const float *indVar, float *depVar, int
*dimDepVar) {

   int ind0 = corresp(0);
   int ind1 = corresp(1);
   float pos[2];
   pos[0] = indVar[ind0];
   pos[1] = indVar[ind1];
   evaluate(&pos, 1, depVar);
   *dimDepVar = 1;

}
```

Une application interactive obtient et modifie la table de correspondance d'une fonction à l'aide des opérations suivantes :

- `virtual void Function::getCorrespVector(int **vector, int *nbElem) = 0;`

- `virtual void Function::setCorrespVector(int *vector) = 0;`

Par défaut, la table de correspondance d'une fonction de N variables indépendantes contient [0, 1, 2, 3, 4, ... N-1].

2.11 Modèle paramétrique général interprété

Il aurait été possible de remplacer toutes les classes définissant des modèles paramétriques, telles que `CrvPoly`, `SrfExpo3`, et `SrfExpo5` par une seule classe. Cette classe hypothétique implémenterait la construction d'une fonction interprétée à partir d'une chaîne de caractères définissant la fonction. Un exemple d'une telle chaîne de caractères est "f(x, y)=cos(x)*sin(y)". Le problème principal consiste à générer une version interprétée d'une telle fonction. Nous aurions pu utiliser certains outils disponibles sur UNIX pour y arriver. Nous avons préféré nous abstenir pour la simple raison que l'effort de programmation n'est pas justifié par rapport à la faible performance que l'on obtient de ce type de fonction. Dans le contexte du calcul éléments finis, la performance est plus importante que la généralité.

Chapitre 3

Bibliothèque graphique

Dans ce chapitre, nous présentons la bibliothèque graphique développée pour répondre à nos nombreux besoins de visualisation 3D. Nous définissons un visualisateur d'une scène 3D qui offre la fonctionnalité de sélection d'un objet graphique. Cette fonctionnalité est fondamentale pour une application d'édition interactive. Cependant, nous constaterons que la sélection d'objets fondée sur le langage graphique OpenGL pose plusieurs problèmes.

3.1 Objet graphique

Un objet graphique a comme principale responsabilité de savoir comment s'afficher à l'écran. Il connaît aussi son étendue géométrique (la boîte qui le contient) en coordonnées universelles ("world coordinates"). Cette information est importante pour un objet qui gère la visualisation de la scène. Un objet graphique peut aussi répondre à des messages généraux du type `Select()`, `Delete()`, `Translate()`, etc ... s'il est utilisé dans le contexte d'une application interactive d'édition.

classe :

`GraphicObject`

rôle :

super-classe de tous les objets graphiques qui savent s'afficher à l'écran.

principales opérations :

- `display()` :
  génération des primitives graphiques pour l'affichage de l'objet.
- `getLimits()` :
  retourne les coordonnées universelles définissant l'étendue de l'objet.
- `select()` :
  sélectionne l'objet (changement d'état).
- `delete()` :
  message transmis à l'objet lorsque sa destruction est requise (pas la même chose que le destructeur).
- `translate()` :
  déplacement relatif de l'objet graphique dans son référentiel objet.
- `scale()` :
  mise à l'échelle de l'objet graphique dans le référentiel de l'objet.

3.2 Classes de visualisation

Une classe de visualisation gère une liste d'objets graphiques, constituant une scène. Son principal rôle est de fournir les fonctionnalités requises pour la modification interactive du point de vue de l'observateur : mise à l'échelle, translation, rotation de la scène. Une classe de visualisation gère aussi la sélection d'objets graphiques. La description que nous faisons des classes de visualisation se rapporte à l'environnement OpenGL/X, bien que le modèle s'applique à d'autres environnements. La figure 3.1 illustre la relation d'héritage entre les classes de visualisation.

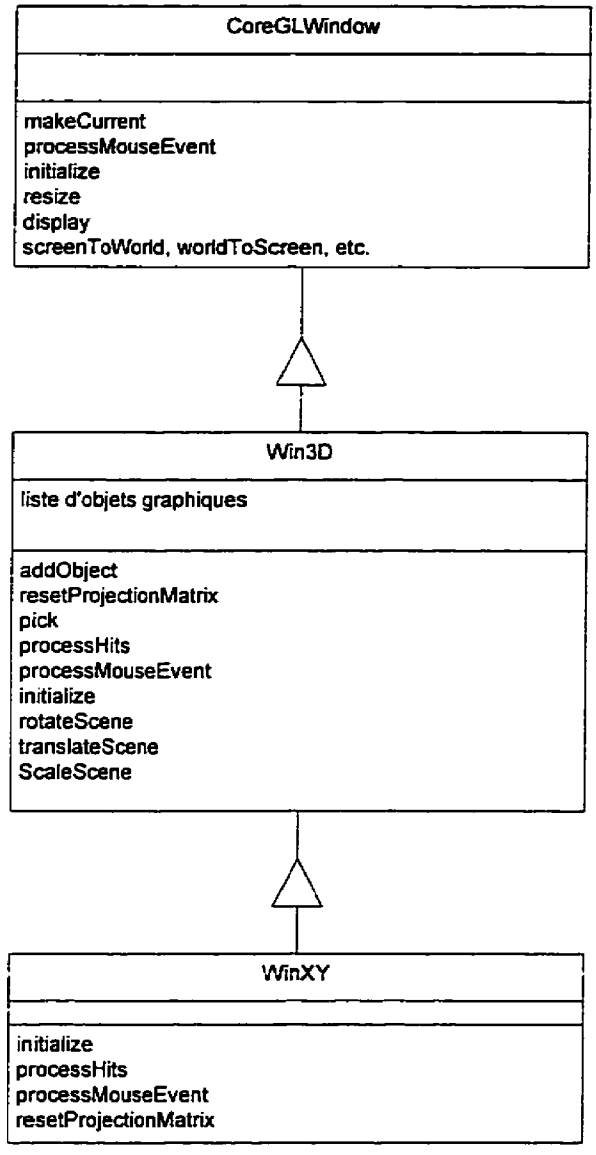


Figure 3.1 Hiérarchie des classes de visualisation

3.2.1 Classe CoreGLWindow

La classe `CoreGLWindow` est la classe mère de toutes les classes de visualisation. Son rôle principal est de servir d'interface entre le gestionnaire d'événements (X Windows),

qui détecte les événements reliés à la souris (pression d'un bouton, déplacement, etc.) et les transmet à l'objet de visualisation. L'objet de visualisation est informé d'un événement par l'intermédiaire de la méthode `processMouseEvent()`. Les classes dérivées de `CoreGLWindow` peuvent redéfinir cette méthode afin d'associer une action à un événement de souris.

---

classe :

`CoreGLWindow`

rôle :

super-classe de tous les objets de visualisation. Reçoit les événements reliés à la souris.

principales opérations :

- `display()` :
  affiche la scène.
- `initialize()` :
  initialisation du contexte OpenGL.
- `processMouseEvent()` :
  appelée par X lorqu'un événement de souris survient.
- `makeCurrent()` :
  rend courant le contexte OpenGL lié à l'instance de cette classe de visualisation.
- `resize()` :
  message envoyé par X lorsque la taille de la fenêtre change. L'objet de visualisation doit mettre à jour sa matrice de projection afin qu'elle préserve, au besoin, le ratio d'écran. Le viewport doit aussi être ajusté à la nouvelle taille de la fenêtre.
- `screenToWorld()` :
  calcul d'une coordonnée universelle à partir d'une coordonnée écran, utilisant la matrice de projection de cet objet de visualisation.

3.2.2 Classe Win3D

La classe `Win3D` définit un visualisateur de scène 3D. Une scène est, dans ce contexte, une liste d'objets graphiques. Cette classe implémente aussi un mécanisme de sélection d'objets graphiques.

Du point de vue client, la principale méthode de cette classe est `addObject()`. Elle permet l'ajout dynamique d'un objet graphique à la scène. L'objet de visualisation doit s'adapter au nouvel objet, c'est-à-dire mettre à jour sa matrice de projection afin de visualiser l'ensemble de la scène. Pour ce faire, il appelle la méthode `getLimits()` de l'objet graphique afin de connaître son étendue, en coordonnées universelles.

Une instance de cette classe modifie la position de l'observateur en fonction du déplacement de la souris et du modificateur actif (SHIFT, ALT, CTRL). L'interprétation par défaut est :

- CTRL-bouton gauche : rotation de la scène autour d'un point. Ce point est, par défaut, le centre de la boîte englobant la scène.
- SHIFT-bouton gauche : translation de l'ensemble de la scène.
- ALT-bouton gauche : application d'un facteur d'échelle à l'ensemble de la scène ("zoom").

classe :

`Win3D`

rôle :

Implémente le contrôle interactif du point de vue de l'observateur d'une scène 3D, à partir de la souris. Implémente la sélection étendue et ponctuelle d'objets graphiques. Gère l'ajout dynamique d'un objet graphique à la scène.

principales méthodes :

- `addObject()` :
  ajout dynamique d'un objet graphique au visualisateur.
- `display()` :
  appelle la méthode display() de tous les objets graphiques contenus dans le visualisateur.
- `pick()` :
  appelle la méthode displaySelect() de tous les objets graphiques, dans le but d'obtenir le tableau d'enregistrements, résultat de la sélection en OpenGL. Appel de processHits() avec ce tableau.
- `processHits()` :
  détermine, à partir du tableau d'enregistrements de sélection, quels sont les objets sélectionnés. Un algorithme différent est utilisé s'il s'agit de sélection ponctuelle ou étendue.
- `initialize()` :
  initialisation du contexte OpenGL. Activation du test sur la profondeur. Définition des lumières. Initialisation de la matrice ModelView.
- `resetProjectionMatrix()` :
  reconstruction du point de vue de l'observateur par défaut, à partir des limites de la scène. Appelée lorsqu'un objet est ajouté à la scène.
- `processMouseEvent()` :
  interprète les événements de souris. Appel de `rotateScene()` sur réception de CTRL/left button, de `translateScene()` sur réception de SHIFT/left button, de `scaleScene()` sur réception de ALT/left button.

3.2.3 Classe WinXY

La classe `WinXY` dérive de `Win3D`. Elle redéfinit certaines méthodes de `Win3D` pour les besoins spécifiques de la visualisation 2D. Les applications 2D (ex : éditeur de courbes) dérivent de `WinXY`.

La méthode `initialize()` est redéfinie afin de désactiver le test sur la profondeur ("z-buffer"), inutile en 2D. La méthode `display()` doit aussi être redéfinie car l'effacement du tampon de profondeur, opération coûteuse, n'est pas nécessaire. La sélection d'un objet en 2D est moins complexe que la sélection en 3D. C'est pourquoi `processHits()` est redéfinie. On redéfinit aussi `processMouseEvent()`, afin de rendre impossible la rotation de la scène. Finalement, `resetProjectionMatrix()` doit aussi être redéfinie, car l'initialisation du point de vue de l'observateur n'est pas la même en 2D qu'en 3D.

3.3 Sélection d'un objet graphique avec OpenGL

Le langage graphique OpenGL met à la disposition des programmeurs un ensemble de fonctions utiles pour la sélection d'un objet graphique à l'écran. Malheureusement, à cause de l'implémentation de ce mécanisme, on doit recourir à de nombreuses astuces pour finalement arriver à sélectionner un objet dans une scène 3D. Cette section vise à expliquer les problèmes que l'on rencontre en utilisant la sélection sous OpenGL, ainsi que quelques solutions originales à ces problèmes.

3.3.1 Types de sélection considérés

Deux types de sélection sont considérés dans cette section : la sélection ponctuelle et la sélection étendue. La sélection ponctuelle consiste à obtenir l'identificateur du seul objet **visible** pointé par la souris au moment où le processus de sélection est déclenché. Dans le second cas, l'utilisateur dessine une région rectangulaire à l'écran avec la souris. On doit alors obtenir tous les identificateurs distincts des objets **visibles** dans cette région.

3.3.2 Vue d'ensemble de la sélection avec OpenGL

Pour dessiner une scène avec OpenGL, on génère un certain nombre de primitives graphiques (lignes, triangles, etc.) définies par une succession de sommets. OpenGL utilise ses variables d'état (couleur, position des lumières, matrice de projection, etc.) pour transformer ces primitives en un ensemble de pixels colorés qui se retrouveront éventuellement à l'écran. Cette séquence d'opérations est caractéristique du mode *render* d'OpenGL, mais il existe d'autres modes d'opération qui modifient les opérations du pipeline graphique.

Le mode Select, par exemple, est utilisé pour la sélection d'objets. L'idée est de transmettre à OpenGL les mêmes primitives que celles utilisées en mode *render* pour dessiner la scène, à la différence qu'ici les attributs graphiques (couleur, etc.) n'ont pas d'importance; seule la géométrie compte. OpenGL utilise les primitives qu'il reçoit pour réaliser des calculs d'intersection. Une primitive qui a une intersection ou est contenue

dans la boîte de découpage courante ("clipping region", telle que définie par la matrice de projection courante) génère ce qu'on appelle une *touche* (traduction libre du mot anglais "hit"). C'est l'événement "touche" qui est à la base de la sélection avec OpenGL.

Afin d'utiliser correctement le mécanisme de sélection d'OpenGL, certaines étapes sont nécessaires. Tout d'abord, on doit informer OpenGL, qui agit en tant que serveur, de l'espace mémoire client à utiliser pour les résultats de la sélection. Ce qui est fait par l'appel à la fonction `glSelectBuffer()`. Les paramètres de cette fonction sont l'adresse d'un vecteur d'entiers et la taille du vecteur.

La fonction `glRenderMode()` est la fonction utilisée pour passer d'un mode d'opération à l'autre. Les modes disponibles sont :

- *render*,
- *select*,
- *feedback*.

Le mode *render* est le mode par défaut. Dans ce mode, tel que mentionné plus haut, les primitives envoyées dans le pipeline graphique sont converties en un ensemble de pixels ("rasterization") et se rendent à l'écran. Le mode *select* est utilisé pour la sélection. Les primitives sont comparées à la matrice de projection courante et génèrent éventuellement l'événement touche. Le troisième mode disponible, le mode *feedback*, ne nous concerne

pas ici. Mentionnons simplement qu'il peut être utile pour générer une représentation vectorielle d'une scène OpenGL, en vue par exemple de l'exporter en format *PostScript*.

Le but ultime de la sélection est d'obtenir le ou les identificateurs des objets graphiques sélectionnés. Pour ce faire, l'application manipule une pile de noms ("name stack"). Un *nom* est en fait l'identificateur d'un objet graphique. Juste avant l'étape de génération des primitives pour un objet graphique donné, on met son identificateur sur la pile de noms, par un appel à `glLoadName()`.

Nous décrivons maintenant la structure appelée "hit record" en anglais, et que nous traduisons par *enregistrement de sélection*. Un enregistrement de sélection peut être généré seulement lors d'un appel à `glRenderMode()`, ou lors d'une manipulation de la pile de noms (par exemple, par un appel à `glLoadName()`). Il faut qu'il y ait eu touche depuis la dernière génération d'un enregistrement de sélection pour qu'un nouvel enregistrement soit créé.

Un enregistrement de sélection est composé de trois champs, dans l'ordre :

- Le nombre d'identificateurs sur la pile de noms au moment de la touche.
- Les valeurs z min. et z max., en coordonnées fenêtre, des primitives qui ont causé une touche, depuis la génération du dernier enregistrement de sélection.
- Le contenu de la pile de noms au moment de la touche, le nom du fond de la pile apparaissant le premier dans cette liste.

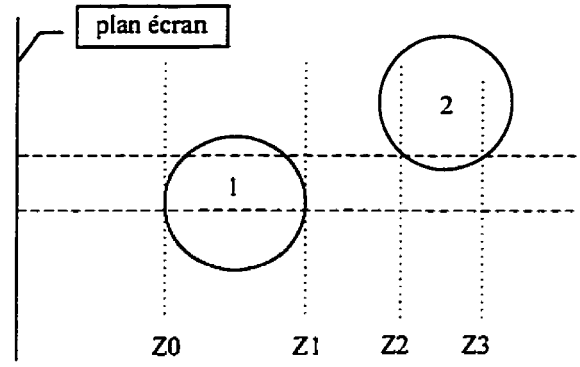


Figure 3.2 Sélection étendue de deux sphères

Toutes les applications que nous avons développées utilisent la pile de noms de la façon la plus simple qui soit : en tout temps, il y a au plus un élément sur la pile de noms. La pile de noms est utile pour construire des modèles hiérarchiques. L'identificateur de l'objet parent est d'abord placé sur la pile, puis celui de l'enfant. Si l'enfant génère une touche, l'enregistrement de sélection contiendra l'identificateur du parent et celui de l'enfant. Par exemple, pour dessiner une portière de voiture, l'identificateur de la portière serait mis sur la pile avant celui de la vitre. Si l'utilisateur sélectionne une vitre, on saura facilement à quelle portière elle appartient à l'aide de l'information retournée par OpenGL dans le tableau de sélection.

Par exemple, la figure 3.2 montre une sélection étendue de deux sphères. Le tableau de sélection retourné par OpenGL contient deux enregistrements. Supposons que la sphère 2 ait été dessinée avant la sphère 1. Le contenu du tableau retourné est le suivant: {1, Z2, Z3, 2, 1, Z0, Z1, 1}. Notons que, même si une sphère est discrétisée en plusieurs petits triangles pour l'affichage, seulement deux enregistrements de sélection sont

retournés. Par contre, si chaque triangle discrétisant la sphère était considéré comme un objet graphique sélectionable, on obtiendrait un très grand nombre d'enregistrements de sélection, chacun ayant une faible étendue en Z.

Lorsque toutes les primitives constituant la scène ont été transmises à OpenGL, on obtient le vecteur d'enregistrements de sélection par un appel à `glRenderMode()`, qui change le mode *select* au mode *render*, par exemple. Autrement dit, c'est le changement de mode qui déclenche le transfert des données de sélection du serveur au client. `glRenderMode()` retourne alors le nombre d'enregistrements présents dans le vecteur de sélection.

C'est ici qu'intervient le problème fondamental de la sélection en OpenGL : le programmeur doit parcourir lui-même le vecteur de sélection afin de déterminer quels sont les objets visibles. Car OpenGL retourne dans le vecteur de sélection tous les identificateurs des objets qui ont causé une touche, que ces objets soient visibles ou non. Revenons à la figure 3.2 pour illustrer ce problème. On voit sur cette figure une vue en coupe d'une scène constituée de deux sphères. La zone de sélection spécifiée (sélection étendue) causera deux touches. La sphère 2, qui n'est pas visible sur la zone de sélection, fera elle aussi partie du tableau d'enregistrements. Dans la plupart des cas, cet effet est indésirable. On ne souhaite pas sélectionner un objet que l'on ne voit pas à l'écran. On doit donc trouver un algorithme pour filtrer le tableau d'enregistrements, qui élimine les objets invisibles.

3.3.3 Implémentation de la sélection dans WinGL

La section 3.1 a décrit le rôle de la méthode `Display()` d'un objet graphique. Rappelons que cette méthode est responsable de la génération des primitives pour une instance d'objet graphique. Cette méthode est déclarée virtuelle pure dans la super-classe `GraphicObject`.

Nous avons doté la classe `GraphicObject` de la méthode `DisplaySelect()`, qui peut se résumer ainsi :

```
GraphicObject::DisplaySelect() {

   // le pointeur this est mis sur la pile de noms
   glLoadName(this);
   Display();// génération des primitives de l'objet

}
```

Soulignons que l'appel `glLoadName(this)` constitue un aspect important de notre méthode de sélection. Plutôt que de mettre un nombre entier dans la pile de noms, ce qui identifie indirectement l'objet graphique, on y met le pointeur à l'objet graphique. Ce qui permet ensuite de transmettre directement un message à l'objet graphique sélectionné, sans passer par une phase d'identification de l'objet.

Lorsque la sélection d'un objet est requise, le visualisateur passe en mode Select. Puis il parcourt sa liste d'objets graphiques et appelle la méthode `DisplaySelect()` de chacun de ces objets. Ce qui a pour effet de générer les enregistrements de sélection.

Finalement, le visualisateur retourne en mode *render*, par un appel à `glRenderMode()` et obtient le tableau de sélection.

La responsabilité du filtrage du tableau de sélection revient à la méthode `processHits()` de l'objet de visualisation. L'implémentation de `processHits()` disponible par défaut dans Win3D et WinXY est triviale : la méthode `Select()` de chaque objet graphique du tableau d'enregistrements est appelée. Aucun test de visibilité n'est effectué. Le développeur, connaissant les spécificités de la scène qu'il traite, peut dériver une classe de `Win3D` ou `WinXY`, et redéfinir `processHits()` selon ses besoins.

3.3.4 Sélection ponctuelle en 2D

Une application telle que l'éditeur de courbes doit implémenter la sélection d'objets graphiques. La sélection est ici simplifiée car il s'agit d'une scène 2D. Le problème de la sélection d'objets cachés ne se pose pas. Cependant, il peut arriver que certains objets se superposent à l'endroit pointé par la souris. On doit lever l'ambiguité sur l'objet sélectionné d'une certaine façon. Nous associons, pour ce faire, une priorité aux différentes classes d'objets graphiques. Par exemple, un point de contrôle a prorité sur une courbe.

Rappelons que, même si une scène a deux dimensions, OpenGL travaille toujours en trois dimensions; les coordonnées Z des sommets générés par des appels à

`glVertex2f()` reçoivent simplement une valeur nulle. Donc, même si l'éditeur de courbes travaille en 2-D, rien ne nous empêche de travailler virtuellement en 3-D. Afin de donner priorité aux points de contrôle, on peut, par exemple, les dessiner sur le plan Z=0. Les courbes, elles, seront tracées en Z=1. Lorsque dans la méthode `processHits()` de l'éditeur de courbes, on parcourt le tableau de sélection à la recherche de l'objet sélectionné, on considère les valeurs Z min et Z max présentes dans chaque enregistrement de sélection. Une enregistrement avec Z min = 0 et Z max = 0 nous indique que nous avons sélectionné un point de contrôle et que les autres objets présents dans le tabeau de sélection peuvent être ignorés.

3.3.5 Sélection ponctuelle en 3D

La présence de l'information Z min et Z max dans chaque enregistrement de sélection est capitale pour la sélection ponctuelle en 3-D. En effet, la sélection ponctuelle peut être vue comme le lancé d'un rayon à travers toute la scène (figure 3.3). Puisque ce rayon est infiniment mince, l'enregistrement dont le Z min est plus petit que tous les autres Z min nous permet très souvent d'identifier correctement l'objet sélectionné.

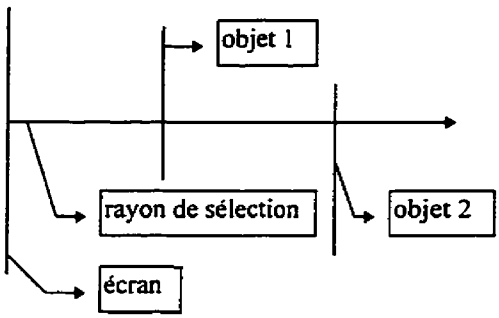


Figure 3.3 Sélection ponctuelle en 3D

Cette solution n'est toutefois pas générale, car elle dépend de la position relative des primitives ainsi que leur taille par rapport à l'étendue de la scène. La figure 3.3 montre une vue en coupe d'une scène constituée de deux polygones, chacun étant un objet graphique sélectionnable. Supposons qu'il s'agisse de deux triangles et que ces deux primitives génèrent une touche. En se basant sur les valeurs de zmin et zmax retournées, on choisira l'objet 2 comme étant le seul objet visible, alors que ce n'est pas le cas. Pour remédier à cette situation, on peut subdiviser l'objet 1 et l'objet 2 en plusieurs petits triangles. Celà a pour effet de ralentir l'affichage de la scène; par contre, il est plus probable qu l'objet sélectionné soit le bon. En général, pour les objets à géométrie courbe, que l'on discrétise toujours en un grand nombre de triangles pour l'affichage, cette méthode donne de bons résultats.

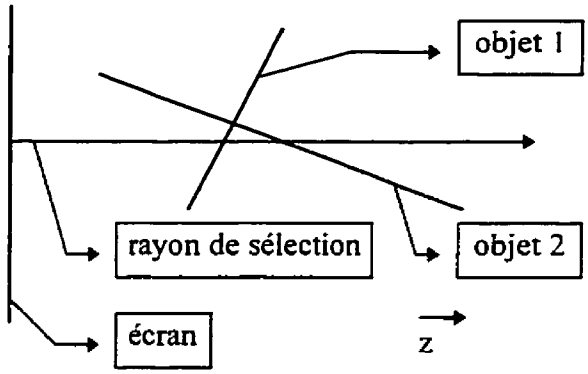


Figure 3.4 Sélection ponctuelle en 3D menant à la sélection d'un objet caché

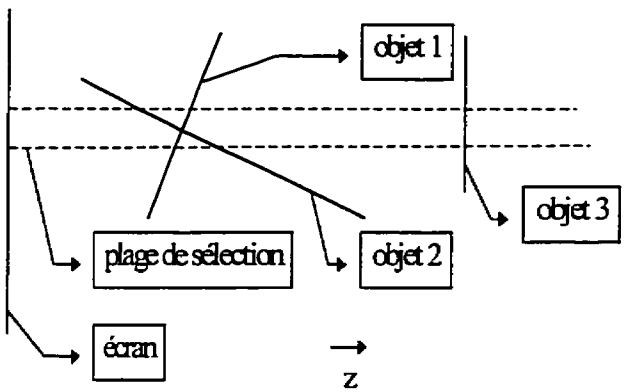


Figure 3.5 Sélection étendue en 3D

3.3.6 Sélection étendue en 3D

La sélection étendue en 3D est très problématique, sous OpenGL. Le problème vient du fait que la zone de sélection est une certaine région étendue de l'écran; il peut donc y avoir plus d'un objet visible dans cette zone, tandis que d'autres objets sont complètement cachés. La figure 3.5 illustre une cas où l'objet 1 et l'objet 2 seulement sont visibles dans la fenêtre de sélection. Cependant l'objet 3 fait lui-aussi partie du tableau de sélection retourné par OpenGL, et il doit être éliminé d'une certaine façon. À

partir de quelle information peut-on éliminer cet objet ? La seule information disponible est, encore une fois, les coordonnées Z min et Z max de chaque enregistrement de sélection dans le repère de la caméra. Conserver uniquement l'objet dont le Z est minimal ne résoud certainement pas le problème. L'objet 3 n'est pas visible sur cet exemple, mais il pourrait bien l'être partiellement sur un autre. L'information sur l'étendue en Z n'est pas suffisante. Nous aurions possiblement besoin d'une information sur les limites en X et Y, en coordonnées normalisées, pour résoudre l'ambiguité, mais elle n'est pas disponible.

3.3.7 Sélection étendue en 3D fondée sur l'utilisation du tampon de stencil d'OpenGL

La recherche d'une solution générale au problème de la sélection étendue en 3D avec OpenGL nous a mené à développer un algorithme original, fondé sur l'utilisation du tampon de stencil ("stencil buffer") d'OpenGL.

Bien que cette méthode soit théoriquement correcte, elle ne peut être utilisée, en pratique, car le nombre d'objets graphiques sélectionnables dans une scène dépend de la profondeur du tampon de stencil. L'implémentation d'OpenGL que nous avons à notre disposition possède un tempon de seulement 4 bits de profondeur. Ce qui implique qu'une scène doit contenir moins de 16 objets graphiques sélectionnables (alors que pour une application de maillage, on compte des milliers d'objets sélectionnables). Nous décrivons la méthode malgré tout.

Rappelons d'abord ce qu'est le test de stencil. Chaque pixel d'un fenêtre OpenGL a une valeur qui lui est associée dans le tampon de profondeur et dans le tampon de stencil. Supposons qu'un fragment vient d'être généré dans le pipeline graphique. Ce fragment subit plusieurs tests avant de se rendre à l'écran. Le test de stencil intervient avant le test sur la valeur de z du fragment. Une des variables d'état d'OpenGL est une valeur de référence pour le test de stencil. L'opérateur du test est une autre variable d'état. Les opérateurs possibles sont : <, <=, >, >=, ==, !=, TOUJOURS, JAMAIS. Par exemple, si la valeur de référence du test est 5, que l'opérateur est > et que le contenu courant du tampon de stencil pour le fragment qui passe le test est 6, le fragment échoue le test. La fonction `glStencilFunc()` est utilisée pour spécifier du même coup l'opérateur et la valeur de référence du test de stencil. Il est possible d'associer différentes actions au test de stencil. On peut associer une action à chacune des situations suivantes:

1. Le test de stencil échoue.

2. Le test de stencil est bon mais le test de profondeur échoue.

3. Le test de stencil est bon et le test de profondeur est bon.

La fonction `glStencilOp()` est utilisée pour associer une action à une de ces situations. Quelques valeurs possibles sont : CONSERVE, REMPLACE, INCRÉMENTE. Par exemple, si on associe l'action REMPLACE à la situation 1, la

valeur contenue dans le tampon de stencil est remplacée par la valeur de référence du test de stencil lorsque celui-ci échoue.

Nous utilisons le test de stencil pour la sélection de la façon suivante. Chaque objet graphique a un identificateur (en fait, le pointeur à l'objet graphique) qui lui est propre. Juste avant la génération des primitives d'un objet graphique, on obtient son identificateur unique. C'est cet identificateur qui est utilisé comme valeur de référence dans le test de stencil. L'opérateur du test de stencil est TOUJOURS, ce qui implique qu'un fragment se rend toujours au test de profondeur. La configuration des actions associées aux différentes situations est la suivante :

1. CONSERVE

2. CONSERVE

3. REMPLACE

Autrement dit, à la fin de la pixelisation de toutes les primitives de la scène, chaque élément du tampon de stencil contient l'identificateur de l'objet visible pour le pixel correspondant.

Tel que mentionné plus haut, cette méthode est correcte, mais en pratique on ne peut l'utiliser. Mis à part le problème de la profondeur du tampon de stencil, il se pose un problème important au niveau du traitement des résultats : on doit filtrer la région du tampon de stencil qui nous intéresse afin d'éliminer les occurences multiples d'un même

identificateur. Un algorithme de hashing permet de faire des économies sur l'espace mémoire à utliser pour réaliser le filtrage. Mais on doit lire tous les pixels concernés par la zone de sélection. Pour une fenêtre de sélection de taille modeste, par exemple 200x200 pixels, on a tout de même 40000 valeurs à traiter. Une très grande plage de sélection, par exemple 1000x1000 pixels, nous oblige à traiter 1000000 identificateurs.

3.3.8 Sélection étendue en 3D: éditeur interactif de groupes de faces

Nous avons eu, au cours de ce travail, à développer une solution au problème de la sélection étendue pour une application d'édition d'un maillage éléments finis. Il s'agit d'un éditeur interactif permettant de sélectionner et de grouper des faces d'éléments finis, afin de spécifier les conditions frontières. Par exemple, il est possible d'ajouter, avec cet outil, des points d'injection à un maillage.

L'éditeur prend à l'entrée un maillage, contenant possiblement plusieurs zones distinctes. L'usager peut sélectionner les faces qui se trouvent sur l'enveloppe externe du maillage (c'est-à-dire les faces qui ne sont pas connectées à un élément voisin), mais il doit pouvoir aussi sélectionner des faces internes. Le problème de la sélection de faces internes n'est pas simple. On doit avoir recours à des plans de coupe, permettant l'exploration de l'intérieur du maillage. Cependant, nous avons choisi de simplifier le problème en limitant les faces internes pouvant être sélectionnées à celles se situant à la frontière de deux zones du maillage. Même si ces faces sont à l'intérieur du maillage, on

arrive à les voir, avec un minimum d'efforts, en désactivant la visualisation des zones périphériques du maillage.

Les faces de chaque élément fini sont des entités sélectionnables, on doit donc en faire des objets graphiques. Toutes les faces externes et à la frontière de deux zones du maillage sont des instances de la classe `GraphicObject`.

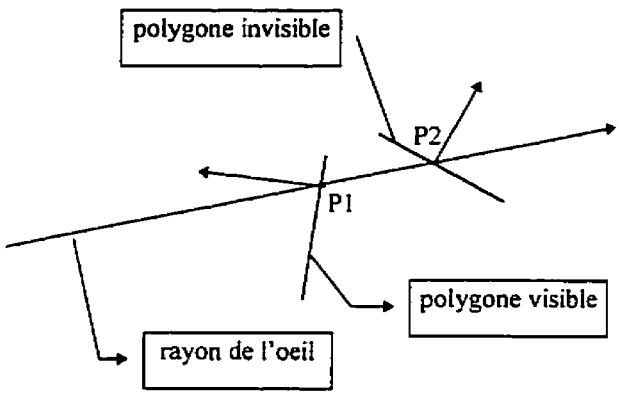


Figure 3.6 Élimination de primitives par l'algorithme de la normale

Nous pouvons utiliser diverses astuces pour solutionner le problème de la sélection étendue pour cette application. Tout d'abord, puisque nous travaillons dans le domaine du RTM, les maillages que nous devons traiter ont tous la propriété d'avoir une surface externe fermée. Ceci nous permet d'utiliser l'algorithme d'élimination de faces cachées à partir de la normale, que nous appelons par la suite test de la normale ("culling" en anglais). Ce test est supporté par OpenGL. On l'utilise afin de réduire à la source le nombre de faces impliquées dans la sélection. Rappelons ce qu'est le test de la normale. Il consiste à déterminer si une primitive est visible, à partir du produit scalaire du vecteur

de l'oeil et de la normale à la primitive. Ce test n'est valable qu'en présence d'une surface fermée. La figure 3.6 montre que, le produit scalaire étant positif au point P1, ce polygone est nécessairement visible. A cause du signe négatif du produit scalaire en P2, ce polygone est nécessairement invisible. Le test de la normale a lieu avant le test de comparaison avec la matrice de projection dans le pipeline graphique, c'est pourquoi on peut l'utiliser pour éliminer du processus de sélection un certain nombre de faces qui ne sont pas visibles.

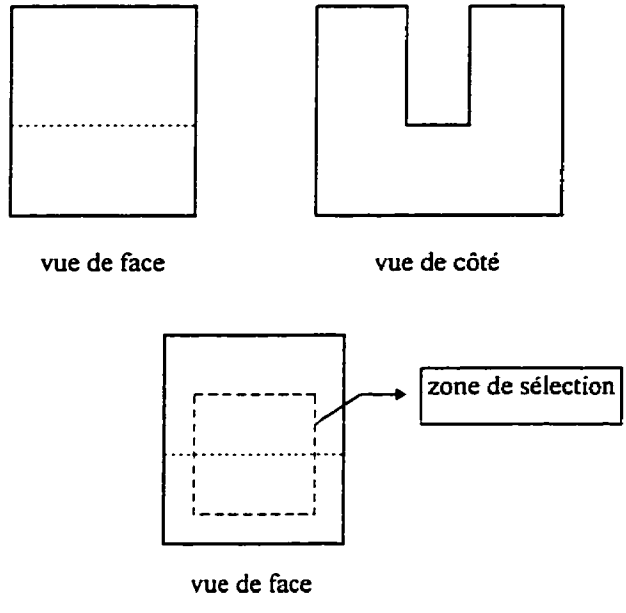


Figure 3.7 Création de groupes de faces disjoints

On doit aussi considérer le fait que certaines faces internes ne sont pas éliminées par ce test et peuvent éventuellement participer à la sélection. Ces faces non éliminées peuvent faire partie de plusieurs zones. Une première hypothèse concernant l'intention de l'utilisateur est que, en général, il ne souhaite pas regrouper des faces faisant partie de

zones physiques différentes. C'est pourquoi nous avons introduit dans l'application la notion de zone courante d'édition, que l'usager peut changer en tout temps. Toutes les faces qui ne font pas partie de la zone courante d'édition ne sont pas sélectionnables. Ce qui permet, encore une fois, de réduire le nombre de faces impliquées dans la sélection.

On doit aussi considérer le cas des groupes de faces disjoints. On peut imaginer certaines situations où, à cause de la géométrie du maillage et de la position de l'observateur, on sélectionne des faces faisant partie de la même zone mais qui génèrent des groupes disjoints de faces, c'est-à-dire un groupe de faces constitué d'au moins deux sous-groupes, qui ne sont reliés par aucun noeud. La figure 3.7 illustre cette situation. Encore une fois, nous émettons une hypothèse: un groupe de faces disjoint n'est pas voulu par l'utilisateur. C'est pourquoi nous avons développé un algorithme qui permet de ne conserver que la portion valide d'un groupe disjoint de faces. Voici cet algorithme:

Soit S un ensemble de faces initialement vide et D l'ensemble de faces possiblement disjoint, résultat brut du processus de sélection.

1. À partir des limites Z min et Z max de chaque enregistrement de sélection, obtenir la face la plus rapprochée de l'observateur. Retirer cette face de D et l'ajouter à S.

2. Trouver dans D une face qui a un noeud commun avec une face de S. Retirer cette face de D et l'ajouter à S.

3. Répéter 2 jusqu'à ce qu'aucune face ne puisse être ajoutée à S.

Le résultat de cet algorithme est un ensemble de faces S qui sont toutes connectées.

3.3.9 Conclusions sur la sélection

On constate que, pour une application donnée, il est toujours possible de trouver des astuces pour rendre fonctionnelle la sélection sur OpenGL. Cependant, l'effort requis pour ce faire est considérable.

On peut maintenant s'interroger sur la pertinence du mécanisme de sélection offert par OpenGL. Dans quel type d'application l'usager souhaite-t-il sélectionner un objet qu'il ne voit pas à l'écran ? Nous n'avons pas trouvé réponse à cette question.

À notre avis, le mécanisme de sélection d'OpenGL intervient trop tôt dans le pipeline graphique pour qu'il soit vraiment utile. C'est qu'OpenGL détermine la sélection d'une primitive dans le repère de l'oeil, alors qu'il est trop tôt, à ce stade, pour savoir si la primitive est visible ou non. Nous pensons qu'OpenGL, dans une version future, devrait implémenter un mécanisme de sélection fondé sur la visibilité d'une primitive, c'est à dire intervenant après le test sur la profondeur.

Chapitre 4

Interaction avec les courbes et les surfaces

Ce chapitre décrit un ensemble d'objets essentiels à la manipulation interactive des courbes et des surfaces. Nous définirons d'abord un ensemble d'objets graphiques compatibles aux visualisateurs décrits au chapitre 3. Un objet graphique courbe utilise la représentation graphique d'une courbe (chapitre 2) pour s'afficher. Nous augmenterons les fonctionnalités du visualisateur de scène 3D pour en faire un visualisateur de courbes et de surfaces. Finalement, en ajoutant quelques fonctionnalités à ce visualisateur, nous obtiendrons un éditeur.

4.1 Objets graphiques

4.1.1 Courbe graphique

Tel que mentionné au chapitre 2, la représentation graphique d'une courbe est contenue dans la classe de calcul (classe de `libcurve`). Cependant, ce n'est pas l'objet de calcul qui a la responsabilité de générer les primitives d'affichage dans le langage graphique utilisé. Il est en effet déconseillé, d'un point de vue génie logiciel, d'établir un lien trop étroit entre un objet de calcul et son interface usager. C'est pourquoi nous définissons la classe `IcCurve`, exportée par `libicurve`, qui dérive de la classe `GraphicObject`

(WinGL) et dont les seules responsabilités sont de définir la méthode `Display()`, qui génère les primitives d'affichage, ainsi que de contenir les attributs graphiques de la courbe (couleur, etc.).

La classe `IcCurve` est donc *associée* à une courbe (elle contient un pointeur à la classe `CrvCurve`), sans toutefois connaître la nature exacte de cette courbe. Elle utilise la méthode `getPlotCoords()` de `CrvCurve` pour obtenir les points caractéristiques de la courbe. Le tableau de points retourné par `getPlotCoords()` doit être tel que la méthode `Display()` de `IcCurve` soit simplement :

```
float *plotCoords=NULL;
int nbCoords=0;
curve->getPlotCoords(&plotCoords, &nbCoords);
glBegin(GL_LINE_STRIP);
   for (int i=0; i<nbCoords; i++) {
      glVertex3f(plotCoords[3*i], plotCoords[3*i+1],
         plotCoords[3*i+2]);
   }
glEnd();
```

Ce code a pour effet de relier graphiquement chaque sommet retourné par `getPlotCoords()` par un segment de droite (figure 4.1).

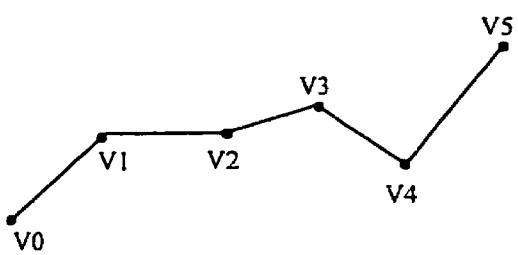


Figure 4.1 Affichage d'une courbe discrétisée

Les attributs graphiques d'une courbe sont :

- Sa couleur,
- sa résolution (le nombre de segments de droite l'approximant),
- son domaine graphique,
- les attributs de ses points de contrôle (commutateur de visibilité, couleur, taille, type de marqueur).

L'attribut `resolution` de `IcCurve` détermine le nombre de segments de droite qui seront générés pour discrétiser la courbe à des fins d'affichage. Cependant, il est possible qu'une courbe ignore cet attribut. Par exemple, une courbe linéaire par morceaux ignore la résolution qui lui est spécifiée lors de l'appel à `updatePlotCoords()`, car cette courbe est toujours représentée exactement par des segments de droite reliant ses points de contrôle. Par contre, pour obtenir une représentation graphique raisonnable d'une courbe possédant des courbures très prononcées, l'usager doit augmenter lui-même l'attribut de résolution, car une courbe ne sait pas, pour l'instant, comment tenir compte de la courbure locale et de la fenêtre de visualisation courante lors de sa discrétisation. Ceci est la raison qui nous motive à définir `updatePlotCoords()` dans les classes de calcul : une courbe connaît la représentation graphique qui lui est appropriée. Les classes `CrvExplicit` et `CrvParametric`, dont dérivent toutes les courbes, fournissent une méthode

`updatePlotCoords()` par défaut. Les courbes constante, linéaire par morceaux, constante par morceaux, ainsi que les polynômes du premier degré sont, pour l'instant, les seules à avoir une représentation graphique exacte, c'est pourquoi elles redéfinissent `updatePlotCoords()`.

Le lecteur sera peut être surpris de constater que le "domaine" de la courbe fait partie de ses attributs graphiques. Cet attribut permet à l'utilisateur de visualiser une courbe paramétrique hors de son domaine graphique usuel défini sur l'intervalle (0, 1). Par exemple, en spécifiant un domaine graphique de (-1, 2), l'utilisateur peut observer la forme de la courbe extrapolée. Le domaine graphique d'une courbe explicite n'a aucune signification dans le contexte de notre visualisateur de courbes. C'est en effet le visualisateur de courbes qui impose aux courbes explicites ses limites courantes en X comme domaine graphique.

4.1.2 Surface graphique

La surface graphique suit le même modèle que la courbe graphique. Les points caractéristiques de la surface sont contenus dans la surface de calcul (`libsurf`). La classe implémentant l'affichage en OpenGL d'une surface est `IsSurface` (`libisurf`).

Deux types de représentation graphique sont disponibles pour une surface : isoparamétrique et maillée (voir le paragraphe 2.8.1.3). Pour afficher une surface avec la

représentation graphique isoparamétrique, la classe `IsSurface` appelle la méthode `getPlotCoordsIso()` de `SrfSurface`. La signature de cette méthode est :

```
void SrfSurface::getPlotCoordsIso(float*** curves, int*
nbCurves, int** nbVerticesPerCurve);
```

Le paramètre `curves` sert à retourner, par adresse, un tableau de tableaux de sommets, lesquels serviront à afficher des courbes par la méthode expliquée au paragraphe 4.1.1. Chaque tableau de sommets définit une courbe graphique isoparamétrique. Le paramètre `nbCurves` retourne le nombre de courbes contenues dans le tableau `curves`, `nbVerticesPerCurve` retourne un tableau d'entiers, permettant de connaître le nombre de sommets de chaque courbe. Ainsi chaque courbe isoparamétrique peut avoir une résolution différente. En pratique, nous n'aurons que deux valeurs distinctes dans ce tableau : la résolution des courbes iso-U et iso-V, U et V étant les paramètres de la surface.

Pour afficher une surface avec la représentation graphique maillée, la classe `IsSurface` doit transmettre à OpenGL un ensemble de triangles construits à partir d'une matrice de sommets. La méthode `getPlotCoordsSolid()` de `SrfSurface` retourne cette matrice de sommets. La signature de `getPlotCoordsSolid()` est :

```
void getPlotCoordsSolid(const float ***vertices, int
*nbRows, int *nbColumns);
```

La méthode `Display()` de `IsSurface` utilise la matrice de sommets retournée, `vertices`, pour générer un ensemble de triangles dont la topologie est illustrée par la figure 4.2. Sur cette figure, le numéro de chaque triangle indique l'ordre dans lequel ils sont transmis à OpenGL.

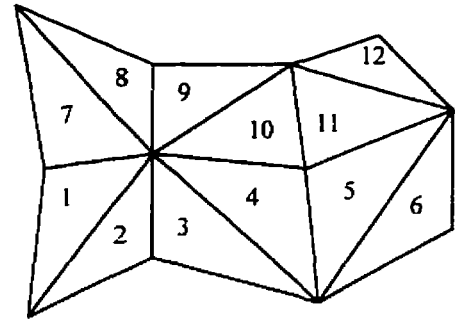


Figure 4.2 Topologie utilisée pour discrétiser une surface en triangles

La méthode `getPlotCoordsSolidSmooth()` est appelée par `IsSurface` lorsque le mode d'affichage nécessite les normales à chaque sommet. Elle retourne donc une matrice de sommets ainsi qu'une matrice de normales à la surface sur chaque sommet. La matrice de normales a la même structure que la matrice de sommets.

En mode iso-intensité, `IsSurface` appelle plutôt `getPlotCoordsSolidFlat()`. La matrice retournée de normales n'a pas la même structure que la matrice de sommets, puisqu'une seule normale est calculée par triangle. L'ordre dans lequel les normales apparaissent dans la matrice suit la numérotation des triangles de la figure 4.2.

Les attributs graphiques d'une surface sont :

- sa couleur,
- son mode d'affichage (isoparamétrique, triangles iso-intensité, triangles dégradé, triangles arêtes seulement),
- les attributs du mode d'affichage (nombre de courbes iso-u, nombre de courbes iso-v, résolution d'échantillonnage selon u, résolution d'échantillonnage selon v),
- son domaine graphique,
- les attributs graphiques de ses points de contrôle (commutateur de visibilité, couleur, taille, type de marqueur).

Le domaine graphique de la surface permet de visualiser, pour une surface paramétrique, un domaine autre que celui par défaut défini sur le carré [0, 1] x [0, 1]

4.1.3 Point de contrôle graphique

La classe `IcPoint` a pour responsabilité d'afficher le point de contrôle (`CrvPoint`) qui lui est associé. `IcPoint` dérive de `GraphicObject`. Une instance de `IcPoint` est donc un objet "sélectionnable" et affichable.

La méthode `Display()` de `IcPoint` est tout simplement :

```
void IcPoint::Display() {
```

```
    glBegin(GL_POINTS);
       glVertex4f(point->getX(), point->getY(), point->getZ(),
          point->getW());
    glEnd();
}
```

Les attributs graphiques d'un point sont sa taille, sa couleur et son type de marqueur. Cependant, puisque tous les points de contrôle d'une courbe ou d'une surface ont les mêmes attributs graphiques, une instance de `IcPoint` doit communiquer avec son "parent" pour obtenir ses attributs graphiques. Voilà pourquoi la classe `IsPoint` a été créée. La classe `IsPoint` est en tout point identique à `IcPoint`, sauf pour ce qui est du "parent" avec lequel elle communique, qui est de la classe `IsSurface`.

4.2 Encapsulation d'une interface utilisateur

Le modèle que nous avons développé pour traiter les interfaces graphiques repose sur la notion de client et de serveur. L'interface agit comme serveur, c'est-à-dire qu'elle offre un ensemble de méthodes de type `get()` et `set()` pour permettre l'interrogation et la modification de son contenu. Le client est un objet qui utilise les services de l'interface pour obtenir des informations de l'utilisateur.

Toutes les interfaces développées au cours de ce projet sont construites sur le principe d'encapsulation. Idéalement, l'environnement de fenêtrage, que ce soit *Motif* ou *Windows*, ne devrait pas être visible à l'extérieur de la classe d'encapsulation. Une telle classe sert à dissimuler et confiner les détails de l'implémentation de l'interface. Les

principales opérations qui concernent le client d'une interface usager sont celles qui lui permettent de mettre à jour son contenu et de récupérer le contenu.

Prenons l'exemple d'une boîte de dialogue demandant à l'usager d'entrer les coordonnées X, Y, Z, W d'un point. Malgré la simplicité de cette interface, il est essentiel, du point de vue génie logiciel, de l'encapsuler. La classe `PointDialog` joue ce rôle :

```
class PointDialog {

public :
   // constructeur
   PointDialog(Widget parent, float x, float y, float z, float
w);

   // destructeur
   ~PointDialog();

   // interrogation de l'interface
   getXYZW(float *, float *, float *, float *);

   // modification du contenu de l'interface
   setXYZW(float, float, float, float);
   setOkCallback(XtCallbackProc, XtPointer);
   setApplyCallback(XtCallbackProc, XtPointer);
   setCancelCallback(XtCallbackProc, XtPointer);

   // ouverture de l'interface
   popup();
   // fermeture de l'interface
   popdown();

private :
   Widget textFieldX, textFieldY, textFieldZ, textFieldW;
   Widget topShell;
   Widget okButton, applyButton, cancelButton;

};
```

Nous insistons sur le rôle de serveur de cette classe, qui définit uniquement des méthodes `get()` et `set()`. De plus, toutes ces méthodes reçoivent en paramètre des objets de type élémentaire (`float`, `int`, `char`).

Le constructeur de `PointDialog` reçoit en paramètre toutes les valeurs requises pour initialiser correctement l'interface. Il construit immédiatement l'interface, en ne conservant dans ses variables membres que les "widgets" nécessaires pour répondre à une requête d'un client : les différents "TextField" (qui contiennent la valeur textuelle d'une coordonnée), le "Shell" parent de toute l'interface (nécessaire pour la destruction de l'interface) et les boutons (nécessaires pour mettre à jour les fonctions de rappel). Le constructeur de `PointDialog`, bien qu'il crée l'interface, ne l'affiche pas. C'est suite à un appel à `popup()` que l'interface s'affiche.

Il existe deux méthodes sous *Motif* pour faire apparaître à l'écran une boîte de dialogue. Une de ces méthodes utilise la fonction `XtPopup()`, l'autre `XtManageChild()`. Le client de l'interface ne doit pas être concerné par ce détail d'implémentation. En effet, peu importe la façon de le faire, le résultat est le même, la boîte de dialogue s'affiche à l'écran. C'est pourquoi toutes les classes d'encapsulation d'une interface doivent définir la méthode `popup()`, ainsi que `popdown()`. Cette dernière ferme la boîte de dialogue sans la détruire.

Le client appelle le destructeur de `PointDialog` lorsqu'il n'a plus besoin de l'interface. Le destructeur a donc la responsabilité de détruire l'interface, ainsi que toute

la mémoire allouée dynamiquement par l'objet d'encapsulation. Dans l'exemple précédent, un simple appel à `XtDestroyWidget(topShell)` suffit.

Pour obtenir les valeurs entrées par l'utilisateur, le client appelle la méthode `getXYZW()`. À l'interne, des fonctions *Motif* sont utiliséees pour obtenir les valeurs textuelles contenues dans les widgets de texte, puis ces chaînes de caractères sont converties en nombres réels, et retournées au client.

La mise-à-jour de l'interface est demandée par le client à l'aide de la méthode `setXYZW()`. Si l'interface est ouverte, les modifications sont visibles immédiatement par l'utilisateur. Il s'agit donc d'une modification dynamique du contenu des "TextField". Il est aussi possible de changer dynamiquement les fonctions de rappel associées aux boutons. Une méthode telle que `setOkCallback()` remplace tous les "callbacks" associés au bouton Ok par le "callback" passé en paramètre. À l'interne, la méthode `setOkCallback()` ressemblera à ceci :

```
PointDialog::setOkCallback(XtCallbackProc callback, XtPointer
clientData) {

   XtRemoveAllCallbacks(okButton, XmNactivateCallback);
   XtAddCallback(okButton, XmNactivateCallback, callback,
      clientData);

}
```

La présence de méthodes `set()` dans la déclaration d'une classe d'interface indique à l'utilisateur de cette classe qu'elle peut être utilisée de façon persistante, c'est-à-dire créée au début de l'application et détruite à la fin. Entre-temps, le contenu de l'interface

peut être mis-à-jour à l'aide des méthodes `set()`, les "callbacks" peuvent changer, et l'interface peut être interrogée avec les méthodes `get()`. Mais ce type d'interface peut aussi être utilisé de façon temporaire, c'est-à-dire créé au moment où il est requis, et détruit aussitôt après. Les deux types d'utilisation sont possibles grâce à l'existence des méthodes `setOkCallback()` et `setCancelCallback()`, qui laissent la possibilité au client de détruire ou non l'interface lorsque l'utilisateur a terminé l'édition.

Le choix du mode d'utilisation de l'interface dépend de la complexité de l'interface, de la persistance du contenu et de considérations sur la consommation de la mémoire vive. En effet, une hiérarchie de "widgets" occupe en mémoire un espace non négligeable, et c'est pourquoi on ne peut se permettre d'utiliser toutes les interfaces d'une application de façon persistante. Par contre, la persistance a ses avantages : il est généralement plus rapide de mettre à jour une interface que de la reconstruire. Il y a donc un facteur ergonomique à considérer : une interface complexe, qui doit être ouverte régulièrement après un délai non négligeable pour l'utilisateur, doit être utilisée de façon persistante afin de minimiser le délai d'ouverture qui, à la longue, exaspère les utilisateurs.

Mais il y a un autre facteur très important qui justifie l'utilisation d'une interface en mode persistant : celui de la persistance des données qu'elle contient. En effet, si une hiérarchie de "widgets" n'est pas détruite au moment de la fermeture de l'interface, son contenu demeure intact jusqu'au prochain appel d'une méthode `set()`. Une boîte de dialogue pour la sélection d'un fichier, interface standardisée sous *Motif*, constitue un

bon exemple d'une interface qui devrait toujours être utilisée de façon persistante. Ainsi, chaque fois que l'utilisateur doit sélectionner un fichier, il se trouve initialement dans le dernier répertoire qu'il a utilisé, et en général c'est celui qui l'intéresse.

Le modèle utilisé pour traiter les fonctions de rappel est important. La consistance est nécessaire afin de minimiser l'effort de maintenance. Les fonctions de rappel qui nous intéressent ici sont celles que spécifient le client de l'interface. En général, une telle fonction est appelée suite à la pression du bouton "Ok", "Apply", ou "Cancel" sur une boîte de dialogue. Le contrôle est retourné à l'objet client, afin qu'il puisse récupérer le contenu de la boîte de dialogue. Pour ce faire, on utilise une fonction intermédiaire qui a pour seul rôle de retourner le contrôle à l'objet client. Reprenons l'exemple de `PointDialog`. La classe client de `PointDialog` - appelons-la `CurveEditor` - demande à être avertie lorsque l'utilisateur appuie sur le bouton "Ok" :

```
void CurveEditor::popupPointDialog() {

   pointDialog->setOkCallback(pointDialogOk_CBwrap, this);
   pointDialog->popup();

}
```

La fonction de retour `pointDialogOk_CBwrap` ne fait que retourner le contrôle à une instance de `CurveEditor` :

```
void pointDialogOk_CBwrap(Widget widget, XtPointer clientData,
        XtPointer callData) {

   CurveEditor *editor = (CurveEditor *) clientData;
```

```
    editor->pointDialogOk_CB(widget, clientData, callData);
}
```

Finalement l'éditeur de courbes, sachant que l'utilisateur a appuyé sur le bouton "Ok", récupère le contenu de l'éditeur de points auquel il est lié :

```
void CurveEditor::pointDialogOk_CB(Widget, XtPointer, XtPointer)
{
    float x, y, z, w;
    pointEditor->getXYZW(&x, &y, &z, &w);
    pointEditor->popdown();
}
```

4.3 Visualisateur de courbes

4.3.1 Spécifications

Définissons une *vue* comme étant une sous-région rectangulaire du plan X-Y. Un visualisateur de courbes affiche cette vue (c'est-à-dire un système d'axes la représentant graphiquement) ainsi que les courbes visibles dans cette vue. Le visualisateur peut contenir plusieurs courbes; les courbes paramétriques coexistent avec les courbes explicites. Cependant, les courbes explicites ont la particularité suivante : leur représentation graphique dépend des limites en X de la vue courante.

1. L'usager peut sélectionner un objet et modifier ses attributs graphiques.

2. L'usager peut modifier la sous-région du plan X-Y qu'il souhaite visualiser par l'intermédiaire de boîtes de dialogue. Trois types de modifications de la vue sont supportés :

2.1. Spécification des limites en X de la vue (x min. et x max.) :

Dans ce mode, l'usager force le visualisateur à afficher la vue spécifiée en X. Toutefois, la plage à afficher en Y est déterminée par le visualisateur.

2.2. Spécification complète de la vue (x min, y min., x max., y max.) :

Dans ce mode, l'usager force le visualisateur à afficher une certaine vue. Il est possible que les courbes soient peu ou prou visibles dans cette région.

2.3. Ajustement automatique de la vue en fonction de la scène :

Dans ce mode, l'usager n'a rien à spécifier. Le visualisateur détermine lui-même une région d'intérêt.

4.3.2 Implémentation

La classe `IcViewer` de `libicurve` définit un visualisateur de courbes. La figure 5.5 montre une instance d'un visualisateur de courbes, utilisé dans le contexte du visualisateur de cinétique chimique de Dataflot.

Le visualisateur de courbes contient deux listes : une liste de courbes graphiques (`IcCurve`) et une liste de points de contrôle graphiques (`IcPoint`). La seule méthode qui devrait intéresser le client de cette classe est `addCurve()`, qui permet l'ajout d'une courbe au visualisateur :

```
void IcViewer::addCurve(CrvCurve *curve);
```

À l'interne, la méthode `addCurve()` crée une instance de `IcCurve`, à laquelle elle associe la courbe reçue en paramètre. Le visualisateur ajoute la courbe graphique à sa liste. Si la nouvelle courbe est définie par des points de contrôle, le visualisateur crée une instance de `IcPoint` pour chacun de ces points de contrôle, et les ajoute à sa liste.

4.3.3 Modification de la vue

Trois méthodes portant le même nom, `resetView`, mais de signature différentes, implémentent les trois modes de calcul de la vue :

- `void IcViewer::resetView(float xmin, float xmax, float ymin, float ymax);`

  Les paramètres reçus déterminent complètement la vue, peu importe que les courbes soient visibles ou non dans cette vue.

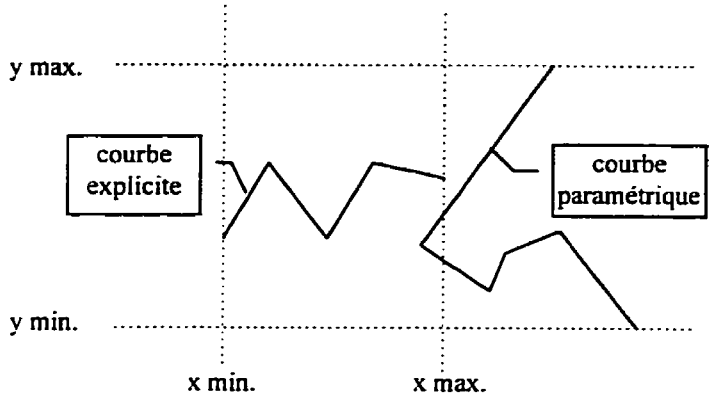


Figure 4.3 Vue 2D dont l'étendue selon l'axe Y est faussée par une courbe paramétrique.

- `void IcViewer::resetView(float xmin, float xmax);`

Les paramètres reçus déterminent seulement les limites en X de la vue. Les limites en Y sont calculées par l'algorithme suivant :

1. Déterminer la vue *VI*, définie par le domaine X spécifié et la fusion des limites en Y de toutes les courbes **explicites**, sur ce domaine.

2. Pour chaque courbe **paramétrique**, si ses limites en X sont comprises en tout ou en partie dans *VI*, fusionner ses limites en Y avec les limites en Y de *VI*.

3. Retourner *VI*.

Note : l'étape 2 peut fausser le calcul de la vue si la courbe paramétrique atteint ses extrema en Y à l'extérieur du domaine X spécifié, tel que montré sur la figure 4.3.

On voit sur cette figure que la vue calculée est inutilement étendue en Y à cause de la courbe paramétrique.

- `void IcViewer::resetView();`

  Cette méthode doit déterminer une vue "idéale" à partir de l'information contenue dans le visualisateur (courbes, points de contrôle). Voici l'algorithme utilisé pour arriver à cette vue idéale.

  1. Déterminer *V1*, la vue englobant toutes les courbes paramétriques.
  2. Déterminer *V2*, la vue englobant les points de contrôle des courbes explicites.
  3. Déterminer *V3*, la vue résultant de la fusion de *V1* et *V2* (plus petite vue englobant *V1* et *V2*).
  4. Déterminer *V4*, la vue englobant toutes les courbes explicites (définies par des points de contrôle ou non) évaluées sur le domaine X de *V3*.
  5. Calculer *V5*, la fusion de *V3* et *V4*.
  6. Retourner *V5*.

4.3.4 Édition des attributs graphiques

Bien que le visualisateur de courbes ne permette pas d'éditer les courbes qu'il contient (rôle qui est laissé à l'éditeur de courbes), il permet de les sélectionner et de modifier leurs attributs graphiques.

Puisqu'une instance de la classe `IcViewer` est conçue pour être utilisée dans une interface plus complexe, elle n'est pas dotée d'une barre de menu. Cependant, l'usager peut faire apparaître un menu en appuyant sur le bouton droit de la souris. L'option **Edit Graphical Attributes** de ce menu ouvre une boîte de dialogue permettant d'éditer l'objet graphique sélectionné.

4.4 Visualisateur de surfaces

4.4.1 Spécifications

Un visualisateur de surfaces permet à l'usager de visualiser les fonctions explicites de deux variables ainsi que les fonctions paramétriques 3D de deux variables. Ces deux classes d'objets peuvent être présentes simultanément dans le visualisateur. L'usager peut faire des rotations, des translations et des mises à l'échelle de la scène qu'il observe, afin de mieux l'interpréter. Ces modifications du point de vue de l'observateur se font dynamiquement, avec la souris.

Définissons une *vue* comme étant une sous-région rectangulaire 3D de l'espace. Le visualisateur de surfaces affiche les surfaces explicites et paramétriques visibles dans cette vue.

L'usager dispose de trois façons pour modifier la vue courante:

- spécification complète de la vue (xmin, xmax, ymin, ymax, zmin, zmax),
- spécification partielle de la vue (xmin, xmax, ymin, ymax),
- détermination automatique par le visualisateur d'une vue intéressante.

Le visualisateur de surfaces permet aussi à l'usager de modifier les attributs graphiques des objets qu'il observe. Pour ce faire, il sélectionne un objet avec la souris et ouvre une boîte de dialogue en choisissant l'option **Edit Graphical Attributes** du menu accessible par le bouton droit de la souris.

4.4.2 Implémentation

La classe `IsViewer` de `libisurf` définit une visualisateur de surfaces. Cette classe dérive de la classe `WinXY` (chapitre 3).

La classe `IsViewer` est la généralisation aux surfaces du visualisateur de courbes, défini par la classe `IcViewer`. Les mêmes fonctionnalités du point de vue client sont offertes. La méthode `addSurface()` permet au client d'ajouter une surface à la scène.

Tout comme dans le cas du visualisateur de courbes, nous présentons les algorithmes utilisés pour déterminer la sous-région à visualiser :

- `resetView(float xmin, float xmax, float ymin, float ymax, float zmin, float zmax);`

  Il s'agit du cas trivial où l'usager spécifie en entier la vue qui l'intéresse. Le visualisateur affiche cette vue sans se soucier du fait qu'il n'y a peut-être rien à visualiser dans cette sous-région.

- `resetView(float xmin, float xmax, float ymin, float ymax);`

  Les paramètres déterminent seulement la composante XY de la vue. On détermine sa composante Z avec l'algorithme suivant :

  1. Soit *VI*, la vue englobant toutes les surfaces explicites sur le domaine XY spécifié.

  2. Pour chaque surface paramétrique, si ses limites XY sont comprises en tout ou en partie dans les limites XY de *VI*, fusionner la composante Z de ses limites avec la composante Z de *VI*.

  3. Retourner *VI*.

- `resetView();`

Pour cette méthode, on doit déterminer une vue "idéale" à partir des surfaces contenues dans le visualisateur ainsi que de leurs points de contrôle. Les surfaces paramétriques ont priorité dans l'établissement de cette vue idéale. Les limites XY des points de contrôle des surfaces explicites fournissent aussi une information importante :

1. Déterminer *V1*, la vue englobant toutes les surfaces paramétriques.
2. Déterminer *V2*, la vue englobant les points de contrôle des surfaces explicites.
3. Déterminer *V3*, la vue résultant de la fusion de *V1* et *V2*.
4. Déterminer *V4*, la vue englobant toutes les surfaces explicites (définies par des points de contrôle ou non) évaluées sur la composante XY de *V3*.
5. Calculer *V5*, la fusion de *V3* et *V4*.
6. Retourner *V5*.

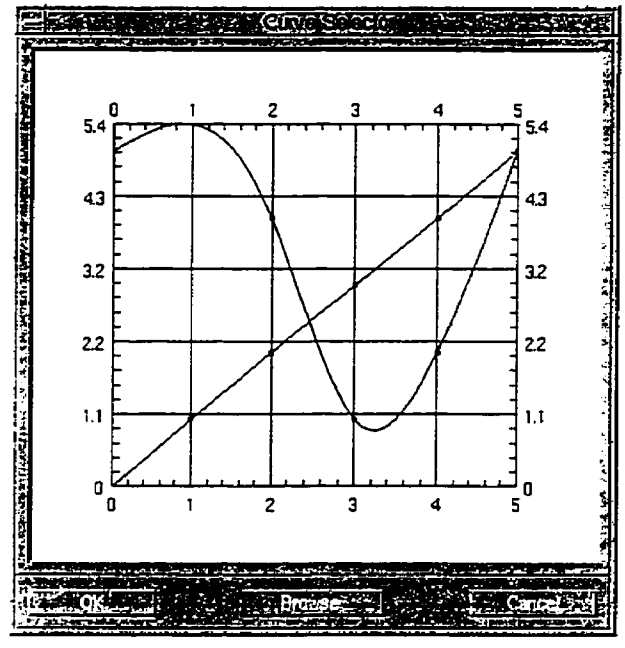


Figure 4.4 Interface graphique permettant la sélection d'une courbe parmi plusieurs

4.5 Sous-système de chargement d'une courbe contenue dans un fichier

4.5.1 Formulation du problème

La situation suivante se présente souvent dans le contexte de Dataflot : l'utilisateur doit associer une courbe à un paramètre de Dataflot et la définition de cette courbe est contenue dans un fichier. Il est possible que ce fichier contienne plusieurs courbes et l'usager doit alors en sélectionner une avec la souris.

4.5.2 Implémentation

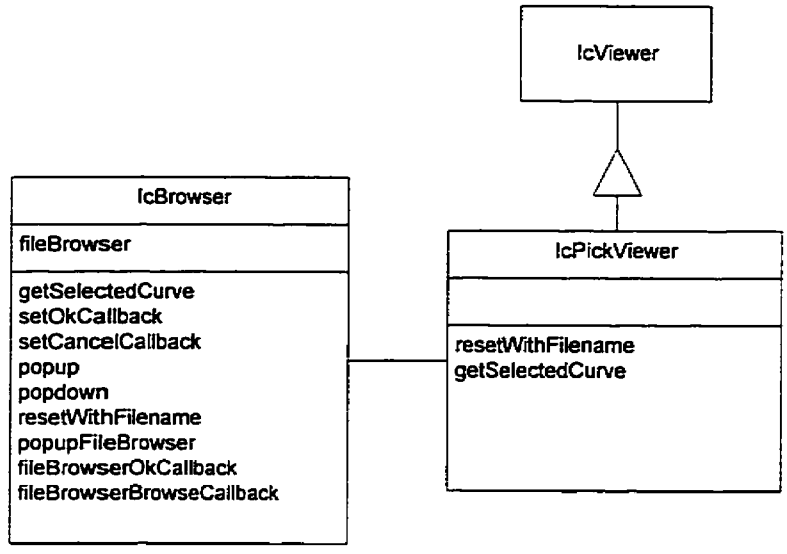


Figure 4.5 Sous-système de chargement d'une courbe contenue dans un fichier

La classe `IcBrowser` est la classe d'encapsulation de l'interface montrée à la figure 4.4 ainsi que la classe d'interface du sous-système. La figure 4.5 montre les classes impliquées dans ce sous-système. La fenêtre OpenGL de cette interface graphique est une instance de la classe `IcPickViewer`. Cette classe dérive du visualisateur de courbes (classe `IcViewer`) et lui ajoute essentiellement une méthode pour le chargement des courbes contenues dans un fichier (`resetWithFilename()`). La classe `IcBrowser` gère une boîte de dialogue de sélection de fichier que l'usager ouvre à l'aide du bouton **Browse**. La sélection d'un fichier charge les courbes qu'il contient par l'appel de `IcPickViewer::resetWithFilename()`.

Le client de `IcBrowser`, sur réception du "callback" associé au bouton "Ok", récupère la courbe sélectionnée par un appel à `getSelectedCurve()`.

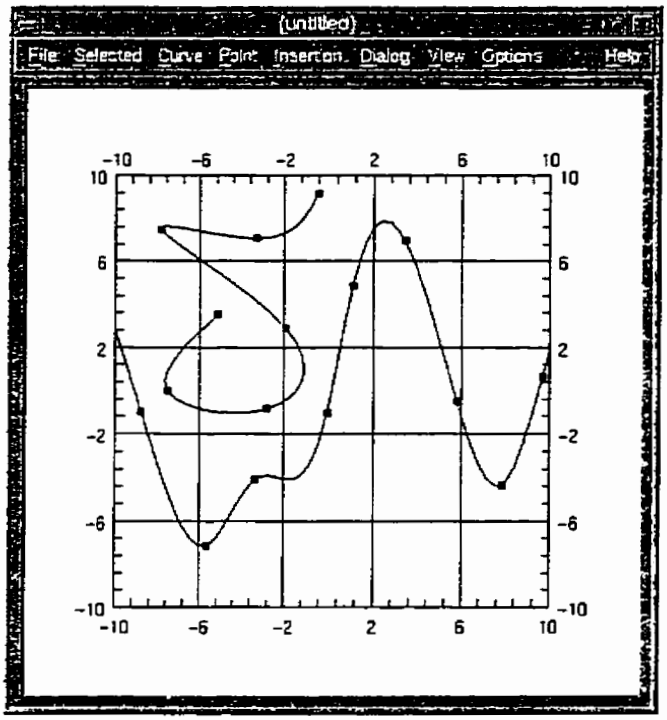


Figure 4.6 Éditeur de courbes

4.6 Éditeur de courbes

4.6.1 Formulation de problème

L'éditeur de courbes permet de visualiser et de modifier simultanément plusieurs courbes paramétriques et explicites. Les opérations qu'il supporte sont :

- Importation/exportation d'une courbe sur disque.

- Construction d'une nouvelle courbe.

- Déplacement, destruction, ajout de points de contrôle.

- Édition des attributs d'une courbe (par exemple, coefficients pour les polynômes, dérive, covariance, et normalisation pour les courbes krigées.).

L'éditeur de courbes permet la construction d'une instance de toutes les classes dérivant de `CrvCurve` (`libcurve`).

Deux formats de fichiers d'entrée et sortie sont supportés :

- Un format très simple, donnant la liste, sur deux colonnes, des coordonnées (x, y) de chaque point de contrôle. Ceci permet l'importation de données brutes provenant de systèmes d'acquisition de données.

- Un format compatible avec Dataflot, fondé sur la représentation en arbre des objets complexes.

L'éditeur ne supporte pas la création de groupes d'objets graphiques. En tout temps, un seul objet graphique peut être sélectionné.

Pour déplacer un objet, l'usager le sélectionne, puis il déplace la souris en tenant le bouton gauche enfoncé. Si l'objet sélectionné est une courbe, une translation est appliquée à tous les points de contrôle. Si la courbe sélectionnée n'a pas de points de contrôle, son déplacement n'a aucun effet.

Quatre options sont disponibles pour l'ajout d'un point de contrôle (menu **Insertion**) :

- Insertion en tête de la liste.
- Insertion en queue de liste.
- Insertion "avant" le point de contrôle présentement sélectionné, c'est-à-dire à la position qu'occupe le point sélectionné dans la liste de points de contrôle.
- Insertion "après" le point de contrôle présentement sélectionné, c'est-à-dire à la position suivante dans la liste de points de contrôle.

La pression de CTRL-bouton gauche déclenche l'ajout d'un point de contrôle à la position courante de la souris. Pour l'insertion en tête et en queue de liste, si plusieurs courbes sont présentes dans l'éditeur, l'usager doit d'abord sélectionner la courbe concernée par l'opération.

Notons que le mode d'insertion n'a aucun effet lorsque la courbe concernée est de classe explicite. Seules les courbes paramétriques sont concernées par le mode d'insertion.

Si l'usager tente d'ajouter un point de contrôle à une courbe qui n'est pas définie par des points de contrôle, l'opération est ignorée.

La destruction d'un objet (point de contrôle ou courbe) peut se faire de deux manières différentes :

- En sélectionnant d'abord l'objet à détruire, puis en choisissant l'option **Delete** du sous-menu **Edit**.

- En pointant l'objet à détruire avec la souris (il n'est pas nécessaire de le sélectionner) puis en appuyant sur la touche **Delete** du clavier.

L'édition des coordonnées d'un point de contrôle se fait à l'aide d'une boîte de dialogue. L'usager ouvre cette boîte de dialogue avec l'option **Edit** du sous-menu **Points**. Elle demeure ouverte tant que l'usager n'appuie pas sur le bouton *Ok* ou *Cancel*. Par la suite, lorsque l'usager sélectionne un point de contrôle, le contenu de la boîte de dialogue est mis à jour. La pression du bouton *Apply* transfère le contenu de la boîte de dialogue au point de contrôle sélectionné. Cette interface est donc de type non modal, ce qui permet à l'utilisateur de continuer à travailler sur d'autres fenêtres lorsqu'elle est ouverte.

L'édition des attributs d'un type spécifique de courbe se fait aussi par l'intermédiaire d'une boîte de dialogue. Par exemple, pour éditer une courbe krigée, l'usager sélectionne d'abord la courbe puis il choisit l'option **Edit** du menu **Curve**. Ceci a pour effet d'ouvrir une boîte de dialogue contenant trois menus à option (dérive, covariance, normalisation) ainsi qu'une zone de saisie de texte pour la distance d'influence. L'usager transfère le contenu de la boîte de dialogue à la courbe en appuyant sur *Apply*. Le type de la boîte de dialogue est modal, ce qui signifie que l'usager n'a pas accès aux autres fenêtres de l'application.

4.6.2 Implémentation

L'éditeur de courbes (classe `CurveEditor`) dérive du visualisateur de courbes (`IcViewer`). Une instance de la classe `CurveEditor` est construite à l'initialisation de l'application ; on l'attache ensuite à l'interface graphique de la figure 4.6. L'instance de la classe `CurveEditor` est en fait, sur cette figure, la fenêtre blanche OpenGL contenant une grille et une courbe.

Toutes les fonctionnalités d'édition sont implémentées par la classe `CurveEditor`. L'interface graphique autour d'une instance de cette classe ne sert qu'à lui envoyer des messages. Toutes les fonctionnalités de visualisation de l'éditeur sont héritées du visualisateur de courbes. Il ne nous reste qu'à définir un ensemble de méthodes permettant l'édition interactive des objets graphiques, ainsi que la création de nouvelles courbes.

4.6.2.1 Sélection d'un objet

La classe `CurveEditor` participe au protocole de sélection d'objets graphiques tel que défini dans WinGL. Lorsque l'usager sélectionne un objet, la méthode virtuelle `processHits()` est appelée. Le rôle premier de `processHits()` est de déterminer quel objet graphique a été sélectionné. Il est possible que plusieurs objets graphiques se chevauchent, et `processHits()` doit en retenir un seul. C'est aussi dans `processHits()` que nous déterminons quelle est la classe de l'objet sélectionné. De plus, selon l'état de l'éditeur et le type de l'objet sélectionné, on doit réaliser

certaines opérations. Par exemple, si l'objet sélectionné est un point de contrôle et que l'éditeur de points de contrôle est ouvert, on doit mettre à jour son contenu.

L'usager a la possibilité de sélectionner seulement quatre types d'objets graphiques : courbe, point de contrôle, tangente, effet de pépite. Une variable membre importante de `CurveEditor` est initialisée dans `processHits()` ; elle nous permettra éventuellement de transmettre des messages à l'objet graphique sélectionné :

```
GraphicObject *selectedObject;
```

4.6.2.2 Conception "idéale" de l'éditeur de courbes

Idéalement, l'éditeur de courbes serait simplement un objet permettant de rediriger un événement se produisant sur l'interface graphique, vers l'objet graphique concerné. Par exemple, lorsque l'usager déplace la souris sur la fenêtre de l'éditeur, celui-ci en est informé et il redirige simplement le message `Translate()` vers l'objet sélectionné.

Malheureusement, la situation n'est pas aussi simple. Supposons que le message `Delete()` soit envoyé à une instance de `IcCurve`. Une courbe est-elle capable de s'autodétuire ? Même si c'était possible, il semble plus logique que ce soit l'objet qui la possède, en l'occurence l'éditeur de courbes, qui le fasse. La méthode `Delete()` de `IcCurve` devrait donc demander à l'éditeur de la détruire. Cependant, nous rejetons ce protocole puisqu'il crée une association bidirectionnelle. En effet, l'éditeur de courbes connaît nécessairement la classe `IcCurve`. Or, si `IcCurve` connaît aussi l'éditeur,

celà crée une association bidirectionnelle. Un objet devient à la fois client et serveur d'un même objet. L'expérience nous incite à rejeter ce type d'association, autant que possible. La figure 4.7 montre que, pour un point de contrôle, le message `Delete()` peut être envoyé directement à l'objet graphique, sans associations bidirectionnelles. Avant de transmettre `Delete()` à un objet graphique, on vérifie sa classe avec un appel à `isPoint()`, par exemple. Si cette méthode indique que l'objet sélectionné est un point de contrôle, on lui transmet le message `Delete()`. S'il s'agit plutôt d'une courbe, c'est l'éditeur qui prend en charge sa destruction.

La figure 4.7 montre également que le message `Translate()` peut être transmis directement à l'objet graphique sélectionné, peu importe son type, sans associations bidirectionnelles.

4.7 Éditeur de surfaces

Au moment de la rédaction de ce mémoire, seul un prototype d'éditeur de surfaces interactif a pu être développé, faute de temps. Malgré tout, nous pouvons formuler le problème ainsi que certaines idées concernant sa conception.

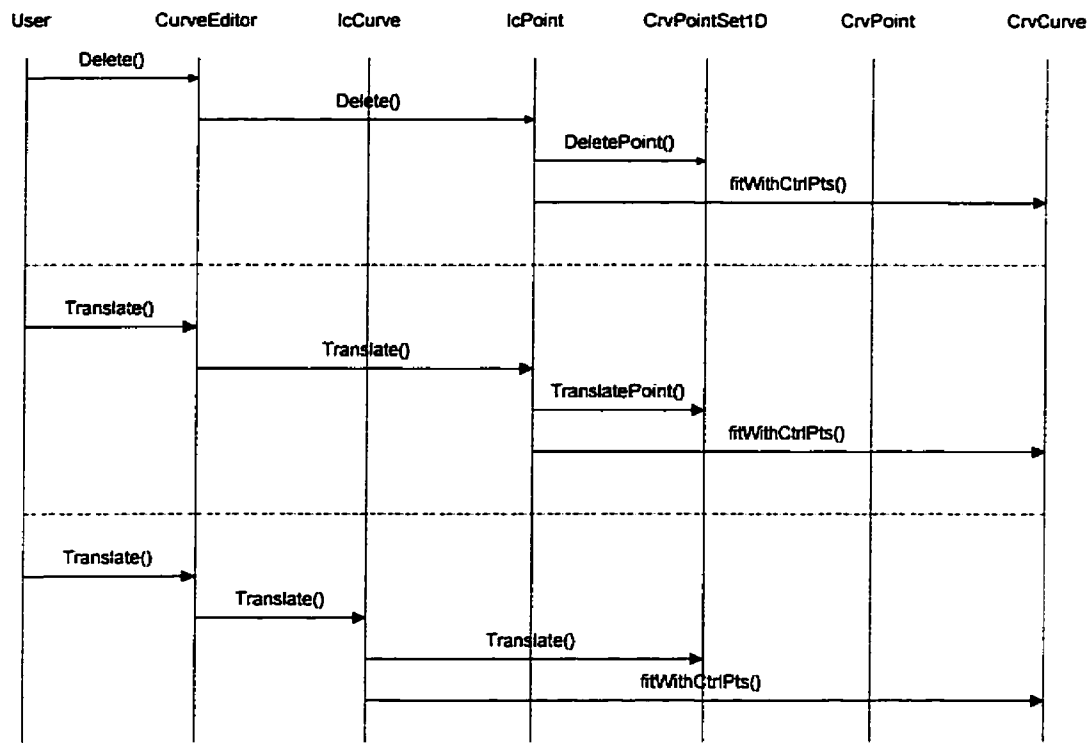


Figure 4.7 Diagramme d'événements pour la translation et la destruction d'un point de contrôle dans l'éditeur de courbes

4.7.1 Formulation du problème

L'éditeur de surfaces a pour premier rôle la construction d'instances des différentes classes de la bibliothèque de surfaces. Ces surfaces peuvent ensuite être enregistrées dans un fichier texte, afin de les utiliser dans d'autres logiciels (FLOT, Dataflot). Il permet aussi la visualisation dynamique de ces surfaces.

Lorsqu'une surface n'est pas définie par des points de contrôle, l'usager a la possibilité d'ouvrir une boîte de dialogue pour éditer ses coefficients.

Lorsqu'une surface est définie par des points de contrôle, l'usager peut modifier leur position dans l'espace de deux façons :

- Par translation d'un point de contrôle avec la souris (ceci nécessite un mécanisme, simple à spécifier pour l'usager, de transformation d'une translation 2D en translation 3D);

- en ouvrant une boîte de dialogue contenant les coordonnées X, Y, Z, W du point de contrôle.

4.7.2 Conception

Les principaux points à considérer dans la conception de l'éditeur de surfaces sont :

- la sélection en 3D d'un objet graphique,

- la transformation d'une translation 2D, sur l'écran, en translation 3D.

Nous avons discuté de la sélection d'un objet graphique avec OpenGL au chapitre 3. Afin de simplifier la sélection d'objets graphiques, nous proposons de limiter la sélection au cas ponctuel. Autrement dit, le regroupement de points de contrôle n'est pas possible. Nous avons vu à la section 3.3 que la sélection ponctuelle en 3D est beaucoup plus facile à traiter que la sélection étendue.

La figure 4.8 illustre le problème de l'interprétation d'une translation de la souris. Le vecteur translation de la souris sur l'écran correspond à une infinité de vecteurs dans la scène. Nous proposons deux méthodes pour lever cette ambiguité.

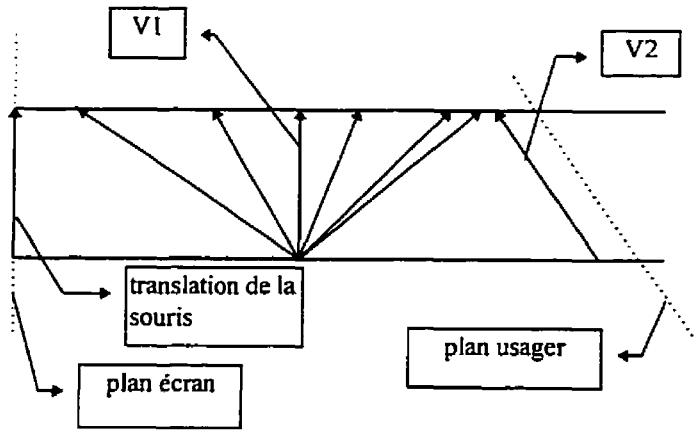


Figure 4.8 Interprétation d'une translation de la souris sur l'écran

La méthode la plus simple utilise uniquement la matrice de projection courante de la fenêtre dans laquelle la translation se produit. On considère que le déplacement de la souris s'effectue sur le plan Z=0, par exemple, en coordonnées fenêtre. À l'aide d'appels à la méthode `screenToWorld()`, on obtient le vecteur de translation correspondant en coordonnées universelles. Sur la figure 4.8, cette méthode retourne le vecteur V1. Si l'utilisateur déplace un objet sur une des vues orthogonales des plans XY, YZ ou XZ, il peut arriver assez facilement à positionner ses objets dans l'espace avec cette méthode. C'est la solution que nous avons retenue pour le prototype développé.

La seconde solution utilise un plan dans l'espace, défini par l'utilisateur. On cherche le vecteur sur ce plan dont la projection à l'écran se confond avec le vecteur déplacement

de la souris. Toujours sur la figure 4.8, le vecteur V2 est retourné par cette méthode. Elle présente cependant le désavantage de produire des singularités, lorsque la normale du plan usager est parallèle au plan de l'écran.

Chapitre 5

Aperçu sur le logiciel Dataflot

Dataflot est le logiciel d'édition des paramètres de simulation du calculateur FLOT. Son rôle consiste à générer un fichier texte qui sera par la suite utilisé par FLOT pour initialiser la simulation (voir l'annexe A pour un exemple).

Dataflot se présente à l'écran comme un panneau principal (figure 5.1) à partir duquel on active des panneaux secondaires. Aucune édition n'est permise dans le panneau principal. Il ne fait qu'afficher un résumé de l'information contenue dans les panneaux secondaires. C'est à partir du panneau principal que l'usager active les fonctionnalités d'entrée/sortie. Le bouton **Load** charge le fichier spécifié, tandis que **Save** enregistre sur disque le contenu courant du logiciel. Le bouton **Reset** remet à leurs valeurs par défaut l'ensemble des paramètres du logiciel.

Le panneau des paramètres généraux de simulation (figure 5.2) contient les paramètres liés aux fichiers d'entrée, fichiers de sortie, ainsi que quelques paramètres contrôlant l'arrêt de la simulation. C'est dans ce panneau que l'on spécifie le temps de pré-injection (pour le préchauffage du moule) et de post-injection (cuisson).

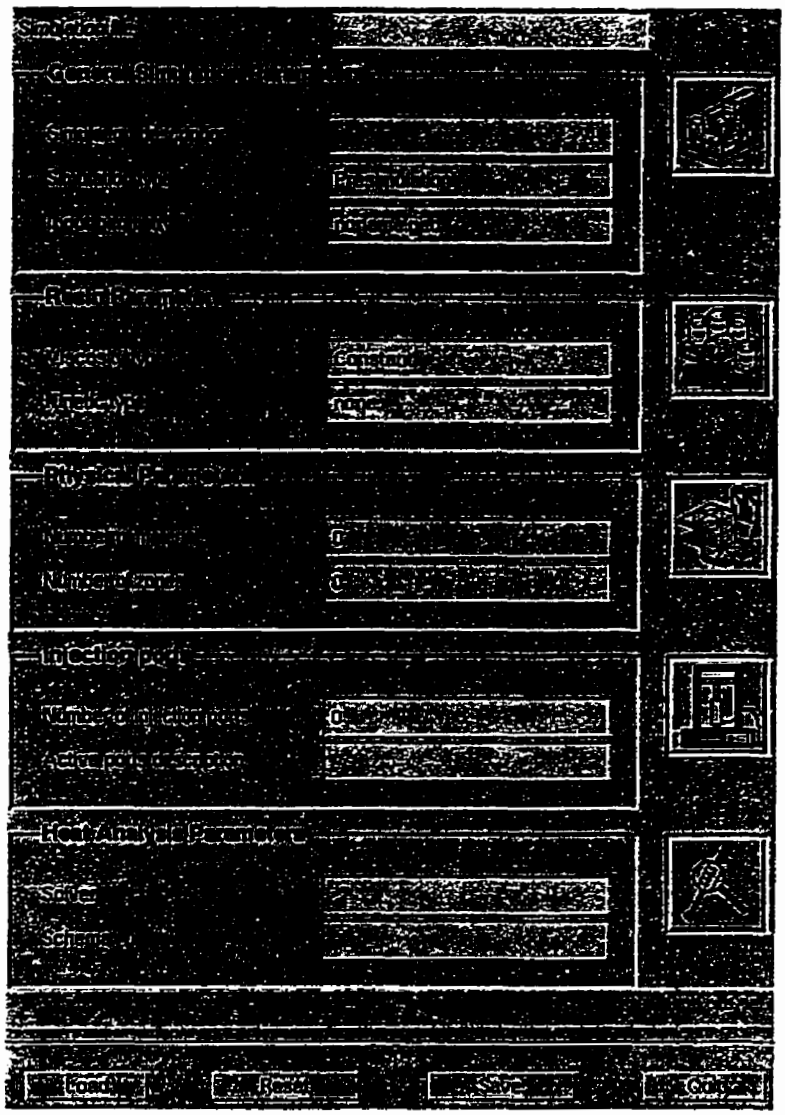


Figure 5.1 Panneau principal de Dataflot

Le panneau d'édition des points d'injection (figure 5.3) permet d'associer des attributs à un point d'injection créé à l'étape du maillage. Le bouton **Scan Geo** démarre la recherche des points d'injection contenus dans le fichier de géométrie. Le panneau affiche tous les points d'injection ainsi trouvés. Il est ensuite possible de copier, détruire ainsi que créer un nouveau point d'injection. Un point d'injection a une régulation en

pression ou en débit. Cependant, dans la version courante du logiciel, tous les points d'injection reçoivent le même type de régulation. Parmi les attributs spécifiques d'un point d'injection on retrouve son activité (un point d'injection peut être défini, mais non actif), son numéro d'identification (utilisé pour faire le lien avec le fichier de maillage) et sa valeur de régulation. Notons que cette dernière peut être une simple constante ou une courbe dont la variable indépendante est le temps.

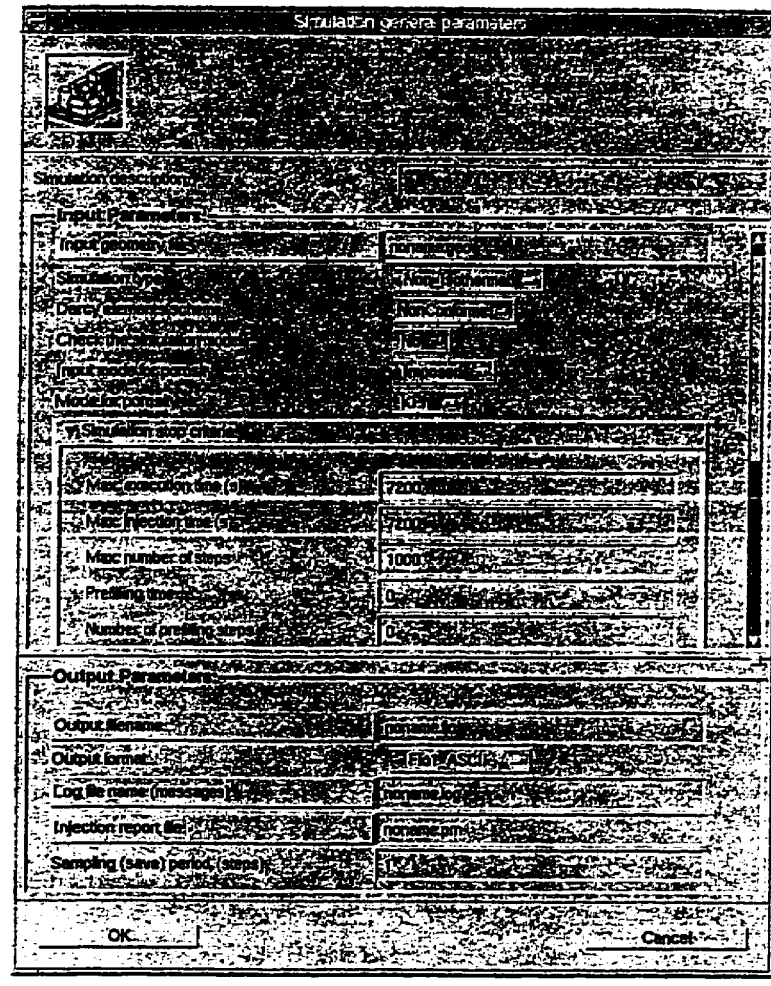


Figure 5.2 Paramètres généraux de simulation

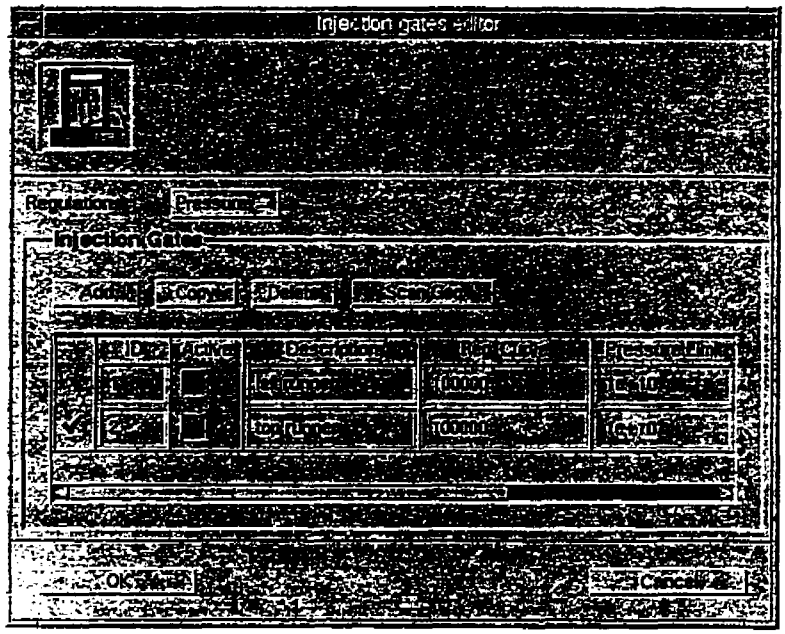


Figure 5.3 Panneau des points d'injection

Nous abordons maintenant l'aspect base de données de Dataflot. Celle-ci contient les propriétés physiques de matériaux (moules, tissus, tubes chauffants), ainsi que la définition de résines.

La figure 5.4 montre le gestionnaire de base de données utilisé dans le contexte de l'édition de résines. On voit dans la partie gauche du panneau une liste de résines ainsi que les boutons **Import**, **Export**, **Copy**, **Create**, et **Delete**. Lorsque ce panneau est ouvert pour la première fois, cette liste contient uniquement la résine courante, c'est-à-dire celle qui est associée au panneau principal. L'utilisateur peut ajouter de nouvelles résines à la liste de différentes façons :

- Le bouton **Import** récupère les résines contenues dans un fichier et les insère dans la liste.

- Après avoir sélectionné une résine de la liste, on appuie sur le bouton **Copy**.

- En appuyant sur le bouton **Add**, on crée une résine par défaut, dont on peut modifier les caractéristiques pour définir une nouvelle résine.

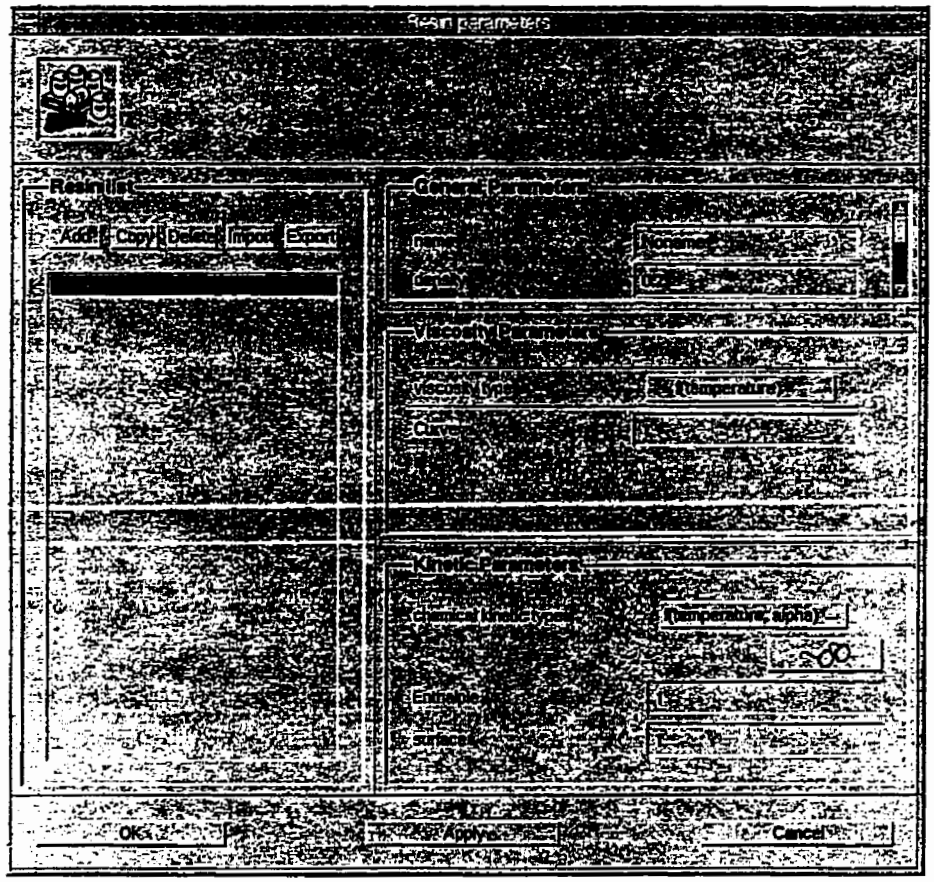


Figure 5.4 Gestionnaire de base de données des résines

Lorsque l'usager sélectionne une résine dans la liste, la portion droite de l'interface, qui contient la description de la résine courante, se met à jour dynamiquement. Cette partie droite de l'interface est divisée en 3 sections :

- paramètres généraux de la résine (nom, C_p, etc.),
- fonction de viscosité,
- cinétique de polymérisation.

En ce qui concerne la fonction de viscosité, quatre modèles sont disponibles :

- constant,
- fonction générale de la température, $f(T)$,
- fonction générale de la température et du degré de conversion, $f(T, \alpha)$,
- fonction générale de la tempéarture et du temps, $f(T, t)$.

Le modèle constant est évidemment un cas particulier des trois autres modèles, mais il permet d'optimiser le calcul.

En ce qui concerne l'équation différentielle de cinétique chimique, deux types de modèles sont disponibles :

- Kamal-Sourour,

- fonction générale de la température et du degré de polymérisation, $f(T, \alpha)$.

L'utilisateur peut rapidement vérifier qualitativement la validité de son modèle de cinétique chimique en appuyant sur le bouton **Preview** (bouton à lunettes), ce qui ouvre le visualisateur de la figure 5.5. Cette interface lui permet d'observer l'évolution de la réaction chimique de polymérisation sur la plage de température et de temps qui l'intéresse.

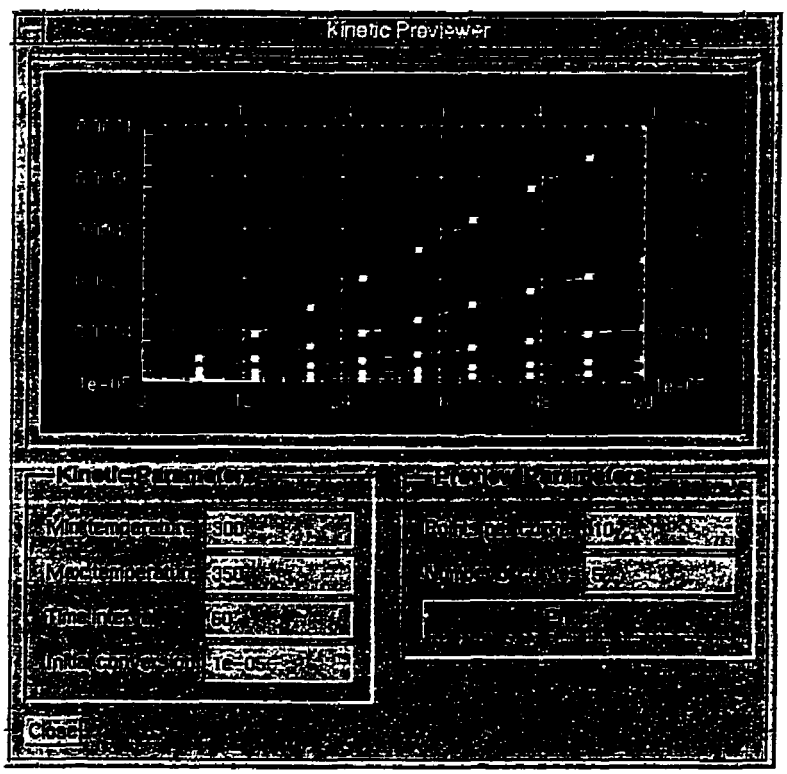


Figure 5.5 Interface de visualisation d'une cinétique chimique

L'association de propriétés physiques à une zone du maillage se fait avec la fenêtre montrée à la figure 5.6. Chaque élément fini doit appartenir à une zone physique.

Puisque, en général, on ne connaît pas les identificateurs numériques de chaque zone du maillage, on utilise le bouton **Scan Geo**, qui parcourt le fichier de géométrie et en extrait les différentes zones.

Chaque ligne du panneau des zones physiques possède un bouton permettant d'ouvrir l'éditeur de matériaux, qui apparaît à la figure 5.7. Le lecteur remarquera que ce panneau a la même apparence que le panneau d'édition des résines, ainsi que les mêmes fonctionnalités (**Import**, **Export**, etc.). La seule différence se situe dans la partie droite de l'interface, celle où l'on édite les propriétés du matériau. Le menu à options **material type** change dynamiquement les paramètres visibles dans cette partie de l'interface. Les options disponibles sont : *Fabric*, *Mold*, *Tube*. Par exemple, sur la figure 5.7, le menu à options **material type** indique *Fabric* et seules les propriétés physiques d'une préforme sont visibles dans l'interface. Finalement, lorsque l'utilisateur appuie sur le bouton **Ok**, le logiciel transfère le contenu du panneau à la zone physique à partir de laquelle l'éditeur de matériaux a été ouvert.

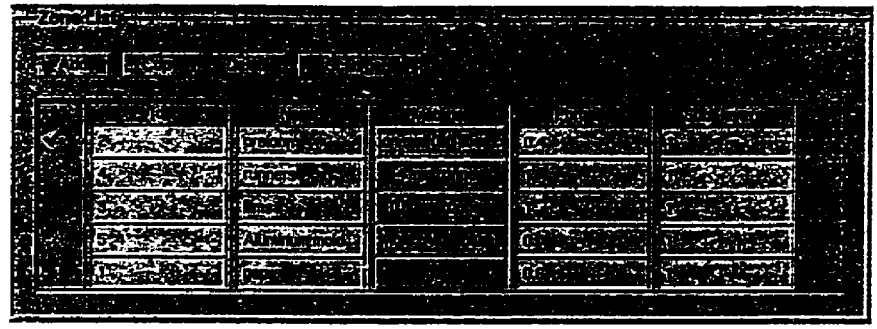


Figure 5.6 Liste des zones physiques

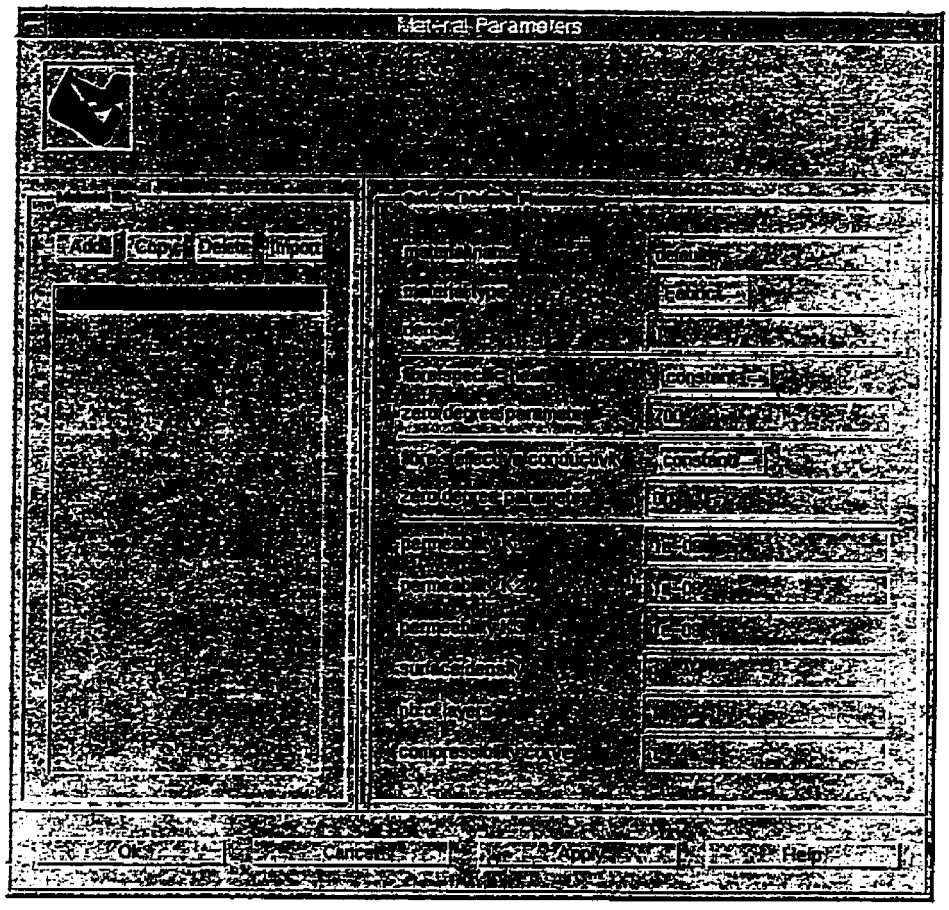


Figure 5.7 Éditeur des propriétés physiques d'une préforme

La figure 5.8 montre le panneau d'édition des paramètres thermiques de Dataflot. Il permet l'édition de paramètres contrôlant la résolution numérique du transfert de chaleur. Notons la présence des sections **Prefilling, Filling** et **Curing** (cette dernière n'est pas visible sur la figure) qui servent à imposer des conditions frontières thermiques variables dans le temps par l'intermédiaire de courbes.

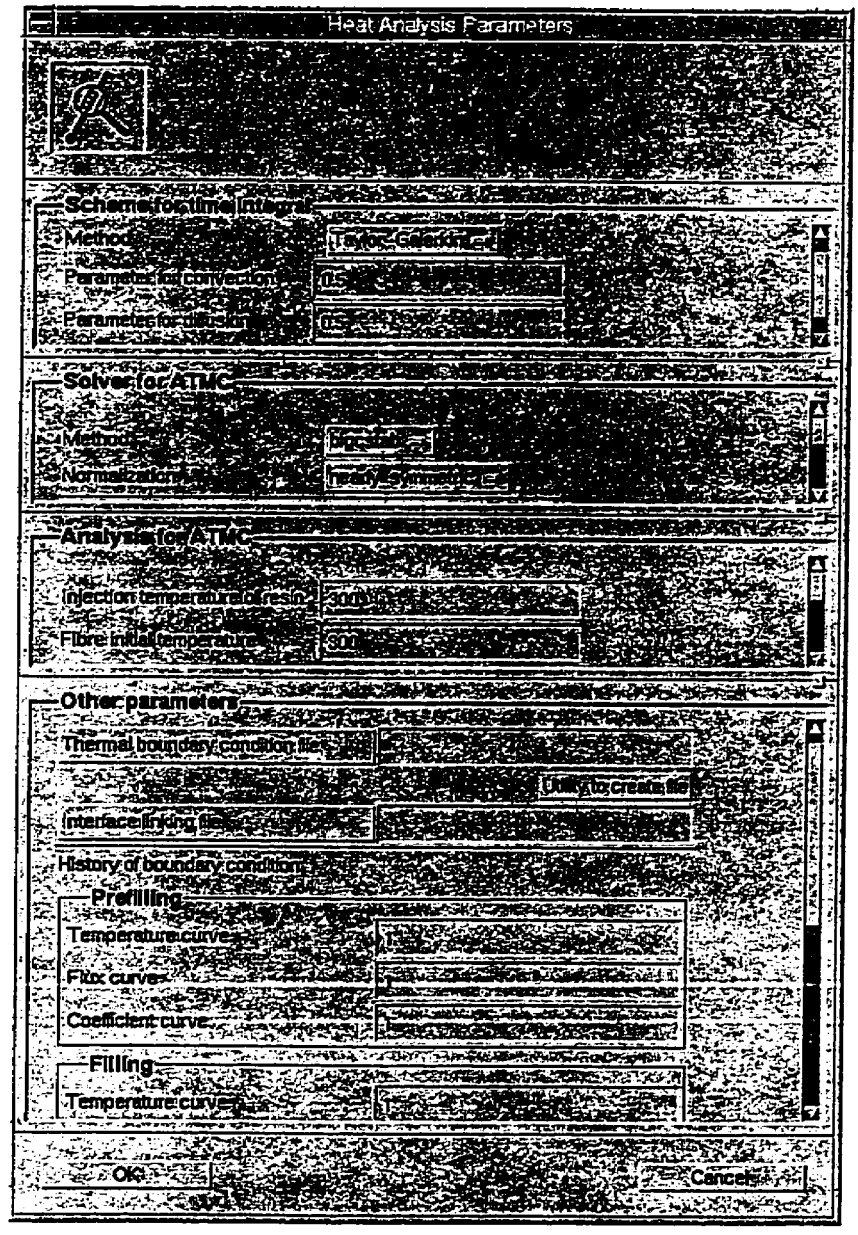


Figure 5.8 Panneau d'édition des paramètres thermiques

Une interface graphique particulière est utilisée pour associer une courbe ou une surface à un paramètre de simulation. Par défaut, ces courbes et ces surfaces sont constantes et l'utilisateur peut en changer la valeur en modifiant le contenu d'un champ textuel. Pour

associer une fonction non-constante au paramètre de simulation, l'utilisateur fait apparaître un menu en pointant la souris dans le champ textuel, puis en appuyant sur le bouton droit de la souris, tel qu'illustré à la figure 5.9. Il y a deux façons de modifier le *type* de la fonction courante :

- avec le bouton **Browse** du sous-menu **Change Model**, ce qui permet de récupérer une fonction définie dans un fichier texte;
- en sélectionnant un des modèles pré-définis du sous-menu **Change Model.**

Pour éditer les coefficients d'un modèle paramétrique à partir de Dataflot, l'utilisateur sélectionne l'option **Edit**. Pour éditer une fonction définie à partir de points de contrôle, il doit plutôt utiliser l'éditeur de courbes ou de surfaces. On charge une telle fonction dans Dataflot avec l'option **Browse** du sous-menu **Changle Model**. L'option **View** permet la visualisation de la fonction courante associée à un paramètre de simulation.

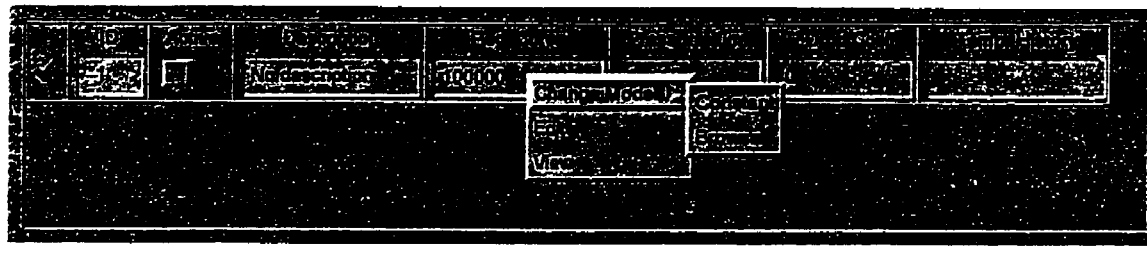


Figure 5.9 Interface permettant l'association d'une courbe à un point d'injection

Chapitre 6

Utilisation des courbes et des surfaces dans Dataflot et FLOT

Dans ce chapitre, nous décrivons d'abord le regroupement des nombreux paramètres de simulation du procédé RTM en objets. Dataflot permet l'édition de ces objets et FLOT les utilise en cours de simulation. Ces explications sont nécessaires pour comprendre comment le code de calcul utilise la bibliothèque de courbes et de surfaces décrite au chapitre 2. Nous étudierons principalement l'utilisation des courbes et des surfaces dans l'évaluation des fonctions de viscosité et de cinétique de polymérisation d'une résine. La possibilité d'associer une courbe ou une surface à un paramètre de simulation au niveau calcul se manifeste nécessairement par l'apparition d'une nouvelle interface graphique dans Dataflot. Nous expliquerons la conception de l'interface graphique développée pour associer une courbe ou une surface à un paramètre de simulation. L'architecture est importante dans ce chapitre puisqu'elle nous permet de montrer la facilité avec laquelle nous introduisons un nouveau modèle paramétrique à Dataflot et FLOT, ce qui constitue un des buts ultimes de ce travail. C'est pourquoi nous présentons à la section 6.9 un exemple détaillant les étapes de l'ajout d'un modèle paramétrique à Dataflot et FLOT. Finalement, nous étudierons la gestion de base de données dans Dataflot et nous proposerons une généralisation de ce module.

6.1 Classes de calcul

Dataflot et FLOT se partagent la connaissance de certaines classes. C'est cette connaissance commune qui permet à Dataflot d'éditer une objet qui sera plus tard utilisé par le code de calcul. Les *classes de calcul* contiennent les paramètres de simulation dont a besoin FLOT pour procéder au calcul. Le seul rôle de Dataflot est l'édition interactive d'instances de ces classes :

- `GeneralParameters`
- `ThermalParameters`
- `InjectionPort`, `InjectionPortList`
- `PhysicalZone`, `PhysicalZoneList`
- `Resin`
- `Material`

Les classes `GeneralParameters` et `ThermalParameters` contiennent des paramètres simples (valeurs numériques, chaînes de caractères). Plus précisément, `GeneralParameters` contient les paramètres généraux de simulation (nom du fichier de géométrie, nom du fichier de résultat, type d'analyse à effectuer, etc.). `ThermalParameters` contient un ensemble de paramètres reliés à l'analyse

thermique (méthode numérique à utiliser, nom du fichier de conditions frontières, courbes d'évolution des conditions frontières dans le temps, etc).

La classe `InjectionPort` contient les attributs d'un point d'injection. Les principaux attributs d'un point d'injection sont son statut (actif ou non), le type de régulation utilisée (pression ou débit), ainsi qu'une courbe décrivant la valeur dans le temps de la pression ou du débit. Un point d'injection contient aussi une valeur limite de pression pour les injections à débit imposé. La classe `InjectionPortList` définit une liste de points d'injection, car une simulation en utilise parfois plusieurs.

La classe `Material` contient la fusion de toutes les propriétés physiques d'un moule, d'un type de renfort (tissu ou mat) ou d'un tube chauffant. Chaque zone du maillage éléments finis est associée à une instance de la classe `Material`. Chaque élément fini possède un numéro le liant à une zone. Ainsi, indirectement, chaque élément fini est associé à une instance de la classe `Material`. L'attribut `materialType` prend une des valeurs suivantes : `Fabric`, `Mold`, `Tube`, selon le type de matériau représenté.

La classe `Resin` sert d'interface pour calculer l'avancement de la réaction de polymérisation d'une résine et sa viscosité.

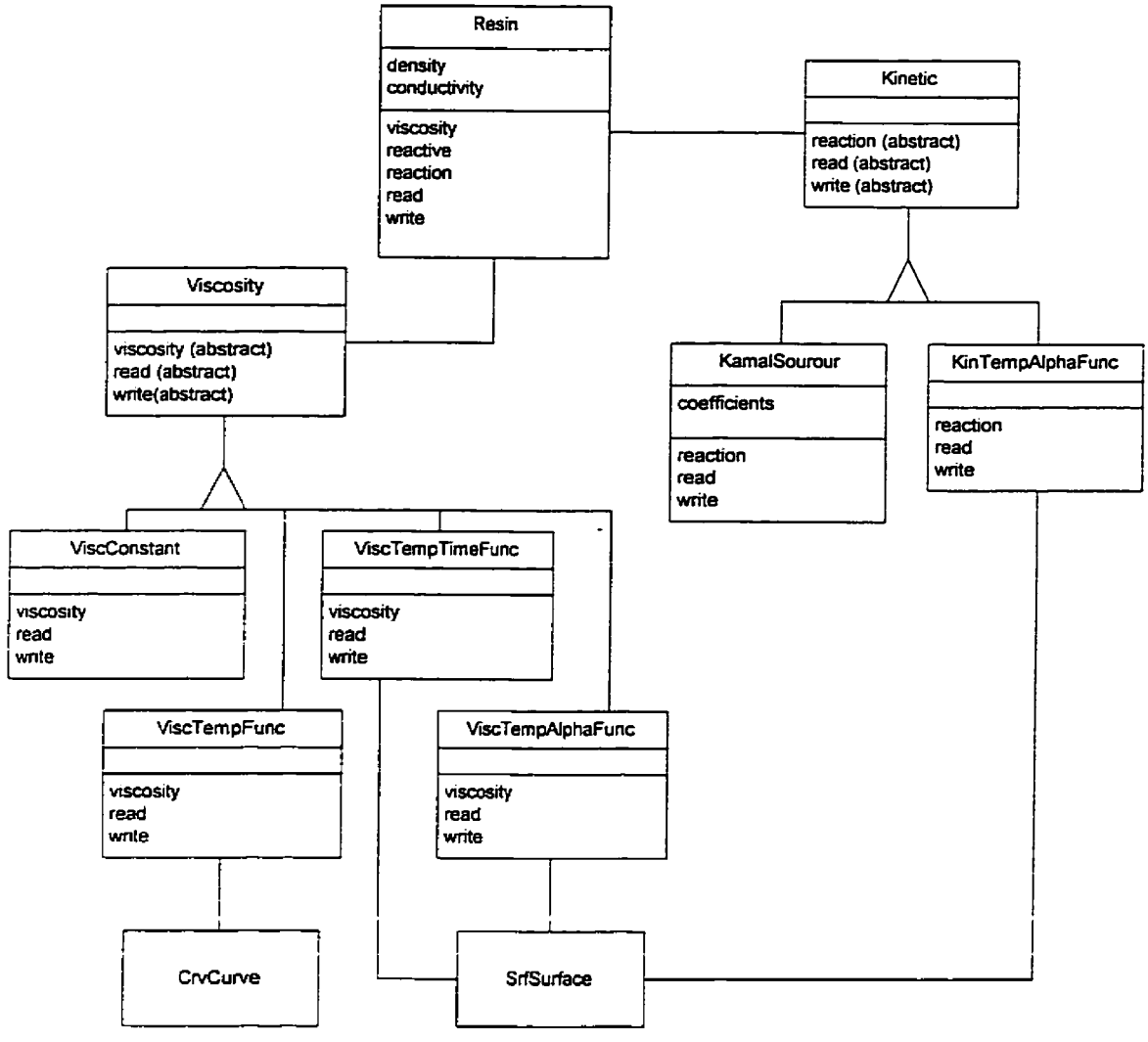


Figure 6.1 Classes impliquées dans la définition d'une résine

6.2 Utilisation des courbes et des surfaces dans la définition d'une résine

Une résine est fonctionnellement plus complexe que les autres paramètres physiques de FLOT. On fait appel au polymorphisme pour évaluer la fonction de viscosité et le degré d'avancement de la réaction chimique de polymérisation.

---

classe :

`Resin`

rôle :

fournir une interface à l'évaluation de la viscosité d'une résine et à sa réaction chimique de polymérisation.

principales opérations :

- `reaction` :
  retourne la chaleur dégagée pendant un intervalle de temps *dt*, pour une température constante et un degré de réaction donnés.
- `viscosity` :
  évaluation de la fonction de viscosité générale, *f(Temperature, α, temps)*.

On voit sur la figure 6.1 le modèle objet des résines. La classe d'interface, c'est-à-dire la seule que connaît le code de calcul, est `Resin`. Elle fournit principalement les opérations `reaction()` et `viscosity()`, lesquelles sont déléguées aux classes `Kinetic` et `Viscosity`, respectivement.

La signature de l'opération `viscosity()` est :

```
float Viscosity::viscosity(float temperature, float alpha,
float time);
```

Il s'agit donc d'une interface générale polymorphe. Le code de calcul appelle cette opération sans avoir connaissance du modèle de viscosité associé à la résine. Il transmet à la résine toutes les variables indépendantes qui peuvent intervenir dans un modèle de viscosité. C'est le modèle lui-même qui sélectionnera les variables qui le concernent. La classe `Resin` ne fait que déléguer l'évaluation de la fonction de viscosité à la classe `Viscosity` qui lui est associée :

```
float Resin::viscosity(float temperature, float alpha, float
time) {

   // myViscosity est une variable membre de type Viscosity *
   return myViscosity->viscosity(temperature, alpha, time);

}
```

Dans cet exemple de code, la variable membre `myViscosity` pointe à une instance de la classe `Viscosity`. Voici deux exemples d'implémentation de `viscosity()`, qui utilisent les courbes et les surfaces :

```
float ViscosityTempFunc::viscosity(float temperature, float
        alpha, float time) {

   float value;
   // myCurve est une variable membre de type CrvCurve *
   myCurve->evaluate(&temperature, 1, &value);
   return value;

}
```

```
float ViscosityTempAlphaFunc::viscosity(float temperature, float
alpha, float time) {

   float value;
   float pos[2];
   pos[0] = temperature;
   pos[1] = alpha;
   // mySurf est une varaible membre de type SrfSurface *
   mySurf->evaluate(&pos, 1, &value);
   return value;

}
```

La classe `ViscosityTempFunc` fait appel à l'opération polymorphe `evaluate()` des courbes, tandis que les classes `ViscosityTempAlphaFunc` et `ViscosityTempTimeFunc` sont associées à une surface. Ainsi n'importe quelle courbe ou surface définies dans ces bibliothèques peut être utilisée dans la représentation d'une viscosité.

Puisque très souvent l'utilisateur de Dataflot se contente d'une viscosité constante, nous avons créé la classe `ViscosityConstant`. Il s'agit en fait d'une optimisation de calcul, puisqu'il serait aussi possible d'utiliser la classe `ViscosityTempFunc` en lui associant une courbe constante. Cependant, en implémentant `viscosity()` comme le montre l'exemple suivant, on évite l'appel à la méthode virtuelle `evaluate()` de la courbe :

```
float ViscosityConstant::viscosity(float temperature, float
alpha, float time) {

   // myConstantValue est une variable membre de type float
   return myConstantValue;

}
```

Parlons maintenant de la polymérisation d'une résine. La classe `Kinetic` spécifie la signature de l'opération `reaction()` :

```
float reaction(float *alphaI, float temperature, float *alphaF, float dt);
```

Cette opération retourne la chaleur dégagée par la réaction chimique sur l'intervalle de temps `dt`, en supposant une `temperature` constante. Elle doit donc résoudre l'équation différentielle $\dot{\alpha} = f(T,\alpha)$. Le tableau `alphaI` contient les valeurs de α initiales requises pour la résolution de l'équation différentielle. Les valeurs de α finales sont retournées dans le tableau `alphaF`.

La classe `KamalSourour` implémente `reaction()` en résolvant l'équation différentielle de Kamal et Sourour (section 1.9) avec la méthode d'Euler modifiée, ce qui donne une précision locale de $O(dt^3)$.

La classe `PolymTempAlpha` fait appel à la bibliothèque de surfaces pour évaluer $\dot{\alpha} = f(T,\alpha)$. Bien que $f(T, \alpha)$ puisse être quelconque, en pratique on utilise surtout une surface krigée explicite, qui est très bien adaptée à ce genre de problème. La méthode `reaction()` de cette classe emploie elle-aussi la méthode d'Euler modifiée pour résoudre l'équation différentielle.

6.3 Généralisation des fonctions de viscosité et de polymérisation d'une résine

Supposons qu'un jour nous souhaitions ajouter à Dataflot et FLOT un nouveau modèle de viscosité qui tienne compte du taux de cisaillement de la résine ($\dot{\gamma}$), par exemple. On devra alors rajouter un paramètre à l'opération `viscosity()` des classes `Resin` et `Viscosity` ainsi qu'aux sous-classes de `Viscosity`. Par exemple, on aura :

```
float Resin::viscosity(float temperature, float alpha,
float time, float shear);
```

De même, pour implémenter la fonction de viscosité $\eta = f(T,\alpha,\dot{\gamma})$ on devra définir une nouvelle classe dérivant de `Viscosity`, par exemple `ViscosityTempAlphaShear`, qui fait appel à l'évaluation d'une fonction générale explicite de trois variables indépendantes. Notons que ces fonctions ne sont pas disponibles pour l'instant. Elles peuvent cependant être développées au besoin en très peu de temps.

Nous constatons toutefois une certaine inélégance de cette architecture. Le lecteur remarquera qu'une classe telle que `ViscosityTempAlpha` n'existe que parce qu'elle sait a priori quels paramètres de l'opération `viscosity()` constituent la température et le degré de conversion ; elle peut donc les retransmettre dans le bon ordre à la fonction *f(x, y)* qui lui est associée. On peut se poser la question suivante : devrons nous

implémenter une classe dérivant de `Viscosity` pour chaque permutation des variables indépendantes (ex : `ViscosityTimeAlpha`, `ViscosityTimeTempAlpha`, `ViscosityTimeTemp`) ? Ces permutations ne sont certainement pas toutes intéressantes (nous pensons d'ailleurs avoir implémenté dans ce projet les plus importantes). Cependant, une certaine généralisation s'impose.

Pour ce faire, nous introduisons une classe que nous nommons `ViscosityGeneral`, et qui dérive de la classe `Viscosity`. Sa méthode `viscosity()` ne fait que retransmettre les paramètres qu'elle reçoit à l'opération `evaluate()` de la classe `Function` qui lui est associée (voir la section 2.10 pour une description de la classe `Function`). La classe `Function` sert d'interface pour évaluer une fonction de N variables. Puisqu'une classe dérivée de `Function` possède M variables indépendantes ($M \leq N$), celle-ci utilise pour l'évaluation une table de correspondance qui associe à chacune de ses M variables une des N variables. Par défaut, la table de correspondance d'une fonction de M variables est $\{0, 1, 2, 3, \ldots M\text{-}1\}$. C'est l'utilisateur de Dataflot qui doit éditer cette table. Cependant, pour y arriver, il doit connaître un détail d'implémentation, c'est-à-dire l'ordre dans lequel se présentent les paramètres de l'opération `viscosity()`. Présentement, on retrouve dans l'ordre : T, α, t. Ainsi, à partir d'une fonction de eux variables $f(x, y)=P(x, y)$, par exemple, l'usager associe t à x et T à y pour évaluer $f(t, T)=P(t,T)$ en spécifiant la table de correspondance $\{2, 0\}$. Cette table indique à la fonction de remplacer sa "première" variable (x) par la troisième

variable (t) de la liste de paramètres et de remplacer sa "seconde" variable (y) par la première de la liste (T).

6.4 Fichier d'entrée de Flot

L'annexe 1 présente un exemple de fichier généré par Dataflot que FLOT utilise en entrée. Ce fichier contient la description textuelle de l'état de chaque objet de calcul au moment de l'enregistrement. On remarque immédiatement que le fichier est divisé en cinq objets principaux, instances des cinq classes de calcul principales : `GeneralParameters`, `Resin`, `PhysicalZoneList`, `ThermalParameters`, `InjectionPortList`.

6.5 Classes d'interface

Les *classes d'interface* sont utilisées pour l'édition d'un objet de calcul. Elle sont connues uniquement de Dataflot. Elles ont pour rôle d'encapsuler une interface utilisateur. Par exemple, la classe `IgeneralParameters` encapsule l'interface servant à éditer la classe de calcul `GeneralParameters`. Elle construit l'interface graphique de la figure 5.2. Les autres classes d'interface sont :

- `ImainPanel` (figure 5.1)
- `IthermalParameters` (figure 5.8)

- `IinjectionPortList` (figure 5.3)
- `IphysicalZoneList` (figure 5.6)
- `Iresin` (figure 5.4)
- `Imaterial` (figure 5.7)

6.6 Architecture de Dataflot

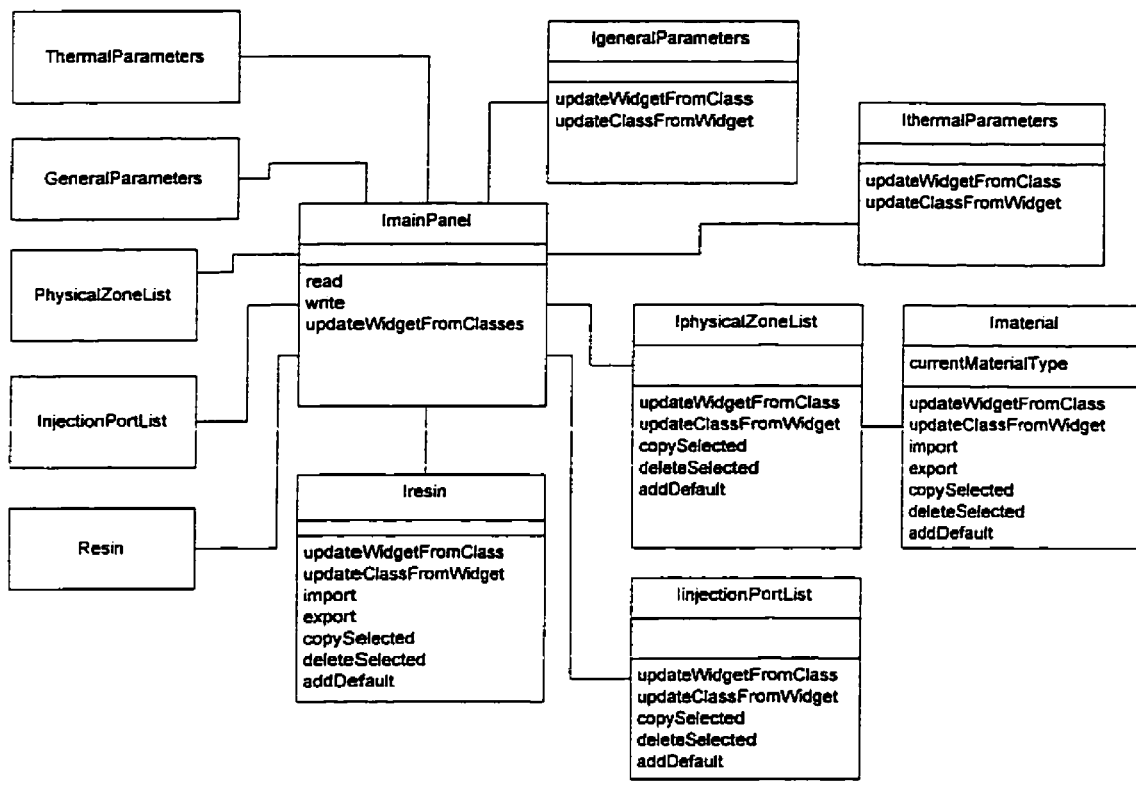


Figure 6.2 Modèle objet de Dataflot

La figure 6.2 illustre l'architecture générale de Dataflot. On remarque que le logiciel est bâti autour de la classe `ImainPanel`, classe d'interface qui encapsule le panneau

principal de Dataflot. Une instance de `ImainPanel` possède donc une instance de toutes les classes de calcul définies par FLOT ainsi qu'une instance des classes d'interface requises pour les éditer.

Lorsque l'utilisateur demande l'ouverture d'un panneau secondaire, `ImainPanel` communique avec la classe d'interface concernée. Celle-ci met à jour son interface graphique à partir de l'objet de calcul qu'elle représente et qu'elle demande au panneau principal. Lorsque l'usager a terminé l'édition, l'objet d'interface a deux possibilités :

- Transmettre le contenu de l'interface à l'objet de calcul du panneau principal, un champ à la fois, en utilisant les opérations `set()` de l'objet de calcul.
- Construire un nouvel objet de calcul à partir de l'interface et le transmettre au panneau principal, qui détruit l'objet de calcul désuet.

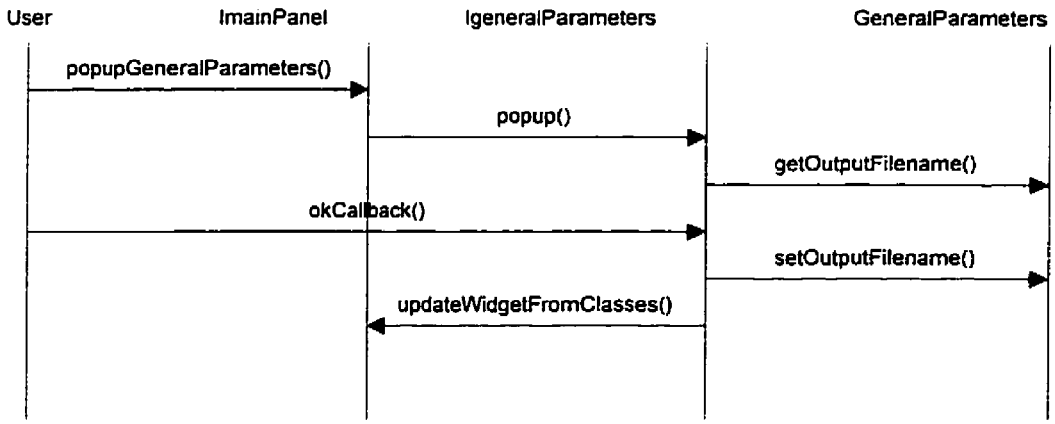


Figure 6.3 Échange de messages lors de l'ouverture d'un panneau et de l'enregistrement des modifications

Pour les interfaces simples, nous préférons adopter la première approche. Pour les interfaces plus complexes (celles qui gèrent une liste, `IinjectionPortList`, `IphysicalZoneList`), on utilise la deuxième approche. Les paragraphes suivants décrivent sommairement les classes d'interface. Nous nous limitons aux principales opérations qu'implémente chaque panneau.

---

classe :

`ImainPanel`

rôle :

classe d'encapsulation du panneau principal de Dataflot.

principales méthodes :

- `read` :
  chargement du fichier Dataflot spécifié par l'usager.
- `write` :
  sauvegarde sur disque du contenu courant du programme.
- `updateClassFromWidget` :
  mise-à-jour de l'interface graphique du panneau principal (résumé de l'information contenue dans les panneaux secondaires) à partir de l'état courant des objets de calcul qu'il contient.
- `reset` :
  destruction des objets de calcul et reconstruction d'objets par défaut.

La classe `ImainPanel` encapsule l'interface graphique du panneau. À sa construction, elle crée une instance par défaut de chacune des classes de calcul suivantes : `GeneralParameters`, `ThermalParameters`, `InjectionPortList`, `Resin`, `ZoneList`. Ainsi, lorsque Dataflot est lancé, il contient des valeurs par défaut valides

pour la plupart des paramètres de simulation. Ce sont les objets de calcul eux-mêmes qui connaissent ces valeurs par défaut.

Cette classe gère aussi le chargement d'un fichier Dataflot, la remise à défaut du programme ainsi que la sauvegarde du contenu courant de Dataflot. Elle est responsable aussi de l'ouverture des panneaux secondaires.

---

classe :

`IgeneralParameters`

rôle :

classe d'encapsulation du panneau des paramètres généraux de simulation.

principales méthodes :

- `updateWidgetFromClass :`
  mise-à-jour du contenu de l'interface graphique à partir de l'objet de calcul de classe GeneralParameters que contient le panneau principal.
- `updateClassFromWidget :`
  transfère le contenu courant de l'interface à l'objet de calcul du panneau principal par ses méthodes set().

---

classe :

`IthermalParameters`

rôle :

classe d'encapsulation du panneau des paramètres thermiques de Dataflot.

principales méthodes :

- `updateWidgetFromClass :`
  mise-à-jour du contenu de l'interface graphique à partir de l'objet de calcul GeneralParameters que contient le panneau principal.
- `updateClassFromWidget :`

transfère le contenu courant de l'interface à l'objet de calcul du panneau principal par ses méthodes set().

Les classes `IthermalParameters` et `IgeneralParameters` sont fonctionnellement identiques. Il s'agit d'interfaces simples en ce sens qu'elles contiennent uniquement des menus à options et des champs textuels. La mise-à-jour dynamique de ces interfaces est donc triviale.

classe :

`IinjectionPortList`

rôle :

classe d'encapsulation du panneau des points d'injection de Dataflot.

principales méthodes :

- `updateWidgetFromClass` :
  mise-à-jour dynamique du contenu de l'interface graphique à partir de l'objet de calcul de classe `InjectionPortList` que contient le panneau principal.
- `updateClassFromWidget` :
  création d'une nouvelle instance de `InjectionPortList` initialisée avec le contenu courant de l'interface, puis tranfert au panneau principal de cette nouvelle liste.
- `copySelected` :
  ajout en queue de liste d'une copie du point d'injection courant.
- `deleteSelected` :
  retrait du point d'injection courant.
- `addDefault` :
  ajout en queue de liste d'un nouveau point d'injection initialisé à défaut.

L'interface graphique de la classe `IinjectionPortList` est relativement complexe à implémenter puisqu'elle gère dynamiquement l'ajout, la copie, la destruction de points

d'injection. Il est donc plus simple de détruire l'interface sur réception du message `updateWidgetFromClass()` et de la reconstruire que d'esssayer de la mettre à jour en préservant certains éléments d'interface.

Une fonctionnalité importante de ce panneau est "Scan Geo", qui parcourt le fichier de maillage (dont le nom a été spécifié dans le panneau des paramètres généraux) à la recherche de points d'injection. Ce traitement est implémenté par la méthode `scanGeo()`. Pour chaque point d'injection ainsi trouvé, le panneau rajoute un élément à sa liste (interface et calcul), dont les attributs sont initialisés à défaut.

---

classe :

`IphysicalZoneList`

rôle :

classe d'encapsulation du panneau des zones physiques de Dataflot.

principales méthodes :

- `updateWidgetFromClass` :
  mise-à-jour du contenu de l'interface graphique à partir de l'objet de calcul de classe PhysicalZoneList que contient le panneau principal.
- `updateClassFromWidget` :
  création d'une nouvelle instance de la classe PhysicalZoneList et transfert de cet objet au panneau principal.
- `copySelected` :
  copie la zone courante.
- `deleteSelected` :
  détruit la zone courante.
- `addDefault` :
  ajoute une zone par défaut.

L'interface gérée par `IphysicalZoneList` contient une liste des différentes zones physiques définies dans le fichier de maillage. Par défaut, une zone est associée à un tissu. Les fonctionnalités de gestion d'une liste chaînée sont identiques à celles de la classe `IinjectionPortList`. La méthode `scanGeo()` parcourt le fichier de géométrie à la recherche des identificateurs de zones physiques. Pour chaque identificateur distinct, elle ajoute une zone à l'interface ainsi qu'à l'instance de la classe de calcul `PhysicalZoneList` associée à l'interface.

---

classe :

`Iresin`

rôle :

classe d'encapsulation du gestionnaire de bases de données de résines de Dataflot.

principales méthodes :

- `updateClassFromWidget` :
  transfert de la résine courante au panneau principal.
- `updateWidgetFromClass` :
  cette méthode ajoute au gestionnaire la résine du panneau principal, si elle ne s'y trouve pas déjà. Dans tous les cas, l'objet courant devient celui du panneau principal.
- `import` :
  chargement de toutes les résines contenues dans un fichier.
- `export` :
  enregistrement dans un fichier de tous les objets contenus dans le gestionnaire.
- `copySelected` :
  création d'une nouvelle instance de résine, copie conforme de la résine courante.
- `deleteSelected` :
  destruction de la résine courante.
- `addDefault` :
  ajout d'une nouvelle instance, initialisée à défaut.

La clase `Iresin` gère une base de données de résines. Son interface est utilisée de façon persistante. Ainsi, il n'est pas nécessaire que l'usager enregistre ses modifications à chaque fois qu'il ferme le panneau.

---

classe :

`Imaterial`

rôle :

classe d'encapsulation du gestionnaire de bases de données pour la définition des propriétés physiques d'un matériau.

principales méthodes :

- `updateClassFromWidget` :
  transfert du matériau courant à la zone physique qui a ouvert le panneau.
- `updateWidgetFromClass` :
  cette méthode ajoute au gestionnaire le matériau associé à la zone physique qui a demandé l'ouverture du gestionnaire, s'il ne s'y trouve pas déjà. Dans tous les cas, l'objet courant devient celui du panneau principal.
- `import` :
  chargement de toutes les définitions de matériaux contenues dans un fichier.
- `export` :
  enregistrement dans un fichier de tous les objets contenus dans le gestionnaire.
- `copySelected` :
  création d'une nouvelle instance de matériau, copie conforme du matériau courant.
- `deleteSelected` :
  destruction du matériau courant.
- `addDefault` :
  ajout d'une nouvelle instance de matériau, initialisée à défaut.

L'interface graphique de la classe `Imaterial` est très semblable à celle de la classe `Iresin`. Les fonctionnalités *Import*, *Export*, *Add*, *Copy*, *Delete*, etc. sont identiques.

Cependant, cette interface se spécialise dans l'édition des propriétés physiques des matériaux pouvant être associés à une zone du maillage (tissu, moule, tube).

6.7 Commentaires sur l'architecture de Dataflot

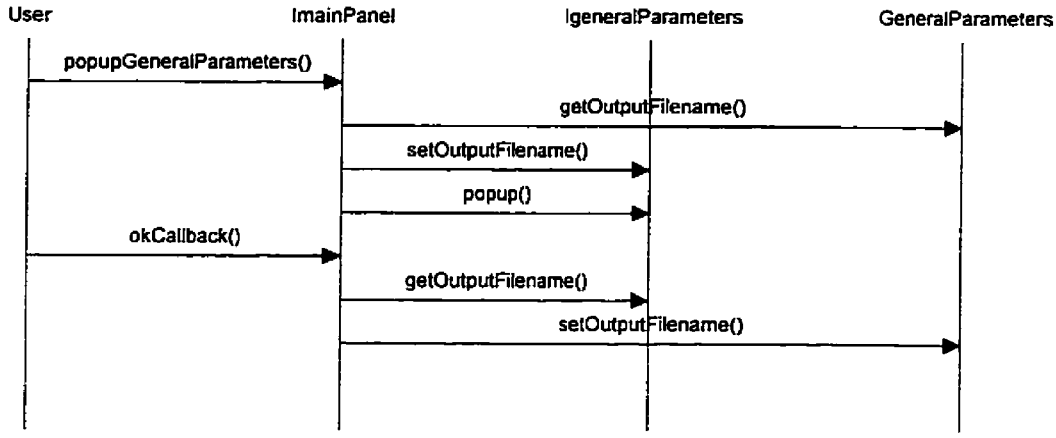


Figure 6.4 Modification proposée à l'échange de messages lors de l'ouverture et de l'enregistrement du contenu d'un panneau

La figure 6.3 montre la séquence actuelle de l'échange de messages, pour une interface typique, lorsque l'utilisateur demande l'ouverture d'un panneau de Dataflot. Nous pensons que ce modèle complique l'entretien du programme, car les messages circulent dans tous les sens. Il ne correspond pas au modèle "idéal" d'interface que nous avons développé à la section 4.2. En effet, la classe `IgeneralParameters` joue à la fois le rôle de client et de serveur de la classe `ImainPanel`. Nous proposons plutôt le modèle illustré par la figure 6.4, où les panneaux secondaires jouent le rôle de serveurs, conformément à notre modèle "idéal". On voit sur cette figure que tous les messages

partent du panneau principal. Contrairement à la figure 6.3, c'est le panneau principal qui reçoit le message d'enregistrement des modifications de l'usager (`okCallback()`) plutôt que le panneau secondaire.

Avec le recul, nous constatons que les panneaux des zones physiques et des points d'injection sont inutilement complexes. En effet, l'utilisateur n'a pas besoin des fonctionnalités *Add*, *Copy*, *Delete* de ces panneaux. Pourquoi y aurait-il moins de points d'injection que ceux définis dans le fichier de maillage ? Pourquoi laisser la possibilité à l'utilisateur d'oublier d'associer un matériau à une zone du maillage ? Nous pensons que seule la fonctionnalité `scanGeo()` est importante, et elle devrait se faire automatiquement à l'ouverture du panneau. Bien sûr, si la liste de zones ou de points d'injection était très longue, la fonctionnalité *Copy* serait aussi justifiée, mais en pratique, ce n'est pas le cas.

6.8 Association d'une fonction à un paramètre de simulation

6.8.1 Formulation du problème

Plusieurs paramètres de simulation sont modélisés par des fonctions d'une ou deux variables indépendantes. La viscosité de la résine, par exemple, peut être une courbe qui dépend de la température. Un modèle plus complexe de viscosité fait intervenir la température et le degré de conversion comme variables indépendantes. La régulation en pression d'un point d'injection est un paramètre que l'on peut modéliser par une courbe

dans le temps. Nous décrivons dans cette section le sous-système développé pour construire une instance de courbe à partir de Dataflot et l'associer à un paramètre de simulation. Un sous-système similaire a été développé pour associer une surface à un paramètre de simulation.

Tout d'abord, on doit distinguer deux types d'utilisation des courbes dans Dataflot. Dans un premier cas, on associe une courbe à un paramètre physique, la viscosité de la résine en fonction de la température, par exemple. Il existe plusieurs modèles paramétriques décrivant la viscosité. Nous voulons laisser à l'utilisateur la possibilité de choisir parmi un ensemble de modèles courants prédéfinis ou, dans le cas où aucun modèle ne lui convient, une courbe d'interpolation qu'il aura construite à partir de données expérimentales. Pour certains types de paramètres de simulation, la pression dans le temps par exemple, il nous est impossible de prédéfinir des courbes, car celles-ci sont trop liées au contexte de simulation. Nous avons dans cette situation deux choix à proposer à l'usager : l'utilisation d'une courbe constante ou le chargement d'une courbe d'interpolation.

Les modèles paramétriques ont l'avantage d'être simples à éditer : l'usager entre la valeur numérique de chaque coefficient. Par contre, l'édition d'une courbe d'interpolation est plus complexe puisque l'usager doit en éditer les points de contrôle. Lorsqu'il s'agit d'éditer un modèle paramétrique, l'usager doit pouvoir le faire dans

Dataflot, sans passer par l'éditeur de courbes. Pour éditer une courbe d'interpolation, l'usager la construit dans l'éditeur de courbes et l'importe dans Dataflot.

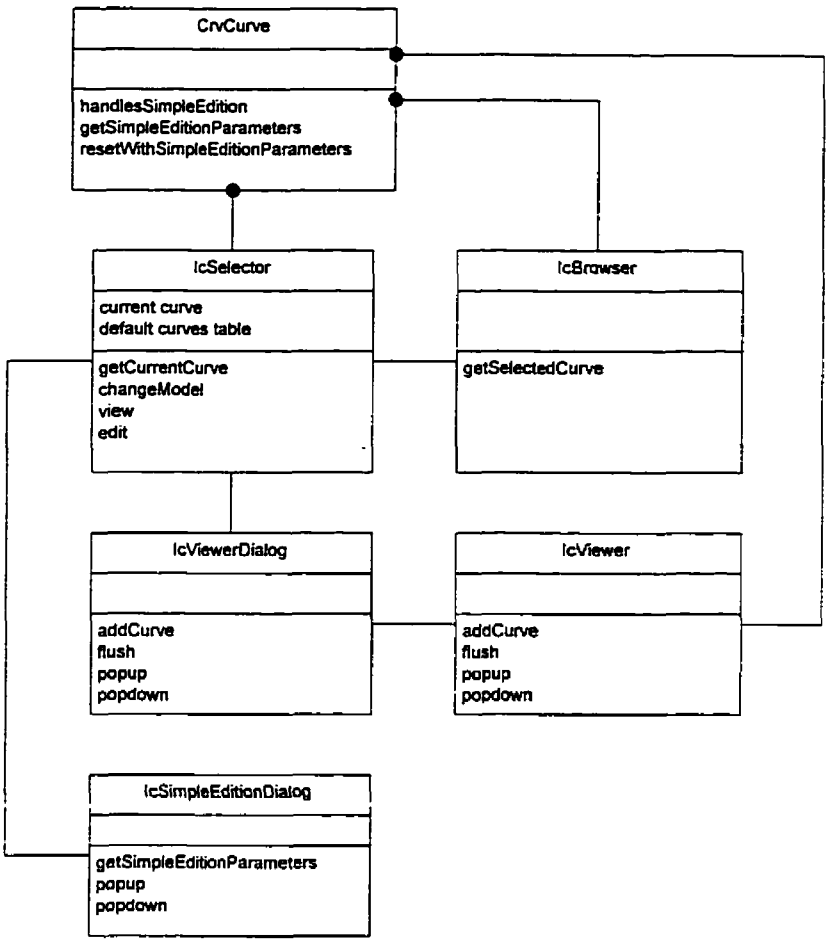


Figure 6.5 Classes impliquées dans le système d'association interactive d'une courbe à un paramètre de simulation

La solution retenue pour répondre à ces besoins est montrée par la figure 6.6. Il s'agit de l'interface d'édition d'un point d'injection. On remarque la présence du champ **Regulation Curve**, qui indique à l'usager qu'une courbe peut être associée à ce paramètre. La courbe courante liée au point d'injection est une constante, et sa valeur

est 100000 Pa.. L'usager, en appuyant sur le bouton droit de la souris, fait apparaître un menu (visible sur la même figure), don les différentes options sont:

- *Change Model*
- *Edit*
- *View*

Change Model permet à l'usager de changer la courbe courante liée à un paramètre. Le sous-menu qui apparaît en sélectionnant *Change Model* contient une liste de courbes prédéfinies (modèles paramétriques). Dans cet exemple, seule une courbe constante est disponible. Si l'usager souhaite simuler une rampe de pression au point d'injection (courbe trop spécifique au problème étudié pour faire partie des courbes prédéfinies), il a la possibilité de créer une courbe linéaire par morceaux dans l'éditeur de courbes et de la charger dans Dataflot en sélectionnant l'option *Browse* de *Change Model*. La fenêtre montrée à la figure 4.4 apparaît. Il s'agit de l'interface de chargement d'une courbe contenue dans un fichier décrite à la section 4.5. L'item *View* du menu sert à visualiser la courbe courante. L'option *Edit* permet l'édition dans Dataflot des coefficients d'un modèle paramétrique. Elle ouvre la boîte de dialogue de la figure 6.6.

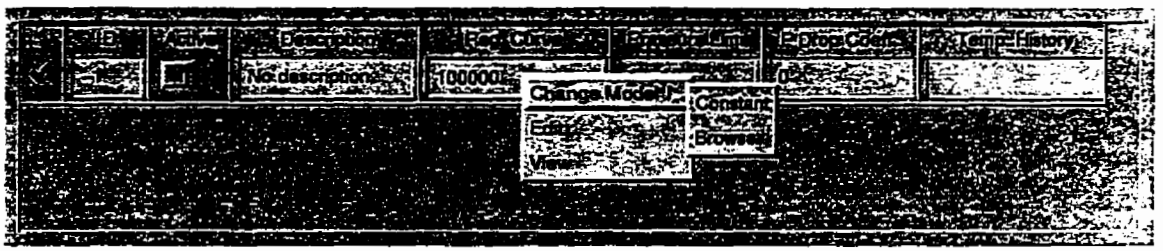


Figure 6.6 Interface d'édition d'un point d'injection

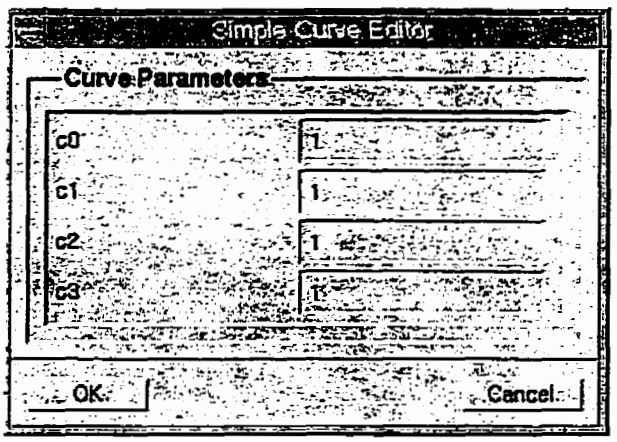


Figure 6.7 Édition d'une courbe polynômiale du troisième degré

6.8.2 Implémentation

La classe servant d'interface au sous-système se nomme `IcSelector`. La figure 6.5 illustre les différentes associations des classes impliquées dans ce sous-système.

Supposons que l'on souhaite rajouter à Dataflot un nouveau paramètre qui peut être modélisé par une courbe. Prenons l'exemple de la chaleur massique de la résine en fonction de la température. On veut laisser à l'utilisateur le choix entre certains modèles prédéfinis (constant, linéaire ou quadratique), ainsi que la possibilité d'aller chercher la

courbe de son choix. L'exemple de code suivant montre comment construire l'objet `IcSelector` dans ce contexte :

```
Widget textW;      // le widget texte qui recevra le sélecteur
float coefs[3] = ...  // la valeur des coefficients est sans
                      // importance ici
CrvCurve *curves[3];
curves[0] = new CrvPoly(&coefs[0], 1); // une constante
curves[1] = new CrvPoly(&coefs[0], 2); // une droite
curves[2] = new CrvPoly(&coefs[0], 3); // une parabole
IcSelector *selector = new IcSelector(textW, &curves[0], 3, 1);
```

L'utilisation de la classe `IcSelector` est donc simplifiée à l'extrême. Le client de cette classe se contente de construire un tableau de courbes qu'il transmet au constructeur de `IcSelector`. Ces courbes seront disponibles à l'utilisateur en tant que modèles prédéfinis. Le constructeur de `IcSelector` reçoit aussi le "widget" de texte auquel sera accroché un menu. L'objet `IcSelector` gère ensuite les différentes options du menu (*Change Model*, *Edit*, *View*). Il fait appel à la classe `IcViewerDialog` pour la visualisation de la courbe courante, à la classe `IcBrowser` pour la récupération d'une courbe contenue dans un fichier et à la classe `IcSimpleEditionDialog` pour l'édition des coefficients d'un modèle paramétrique. Notons que seules les courbes possédant la propriété d'édition "simple" (paragraphe 2.7.1.2) peuvent être éditées à partir de l'objet `IcSelector`.

Le client de `IcSelector` récupère la courbe courante associée au paramètre de simulation par l'appel suivant :

```
CrvCurve *selectedCurve = selector->getCurrentCurve();
```

6.9 Ajout d'un modèle paramétrique à Dataflot et FLOT

Cette section vise à démontrer la facilité avec laquelle on peut ajouter un nouveau modèle à Dataflot et FLOT. On nous demande d'ajouter à ces deux programmes le modèle de compaction de Gauvin (Gauvin et al, 1986, 1988), qui relie le taux de fibres d'une préforme avec la pression appliquée. Nous devons d'abord ajouter une nouvelle classe à la bibliothèque de courbes, que nous appelerons `CrvCompGauvin`. Rappelons que le modèle de Gauvin nécessite 5 coefficients constants, soit V_0, A_0, A_1, A_2, A_3, et que le taux de fibres vf est défini en fonction de la pression appliquée P par

$$v_f(P) = \frac{10v_0}{10A_0P + A_1\ln(10P) + \frac{A_2}{10P} + A_3}.$$

Spécifions d'abord la classe `CrvCompGauvin`. Il s'agit d'une courbe explicite, nous dérivons donc `CrvCompGauvin` de `CrvExplicit`.

```
class CrvCompGauvin : public CrvExplicit {

public :

   // constructeur
   CrvCompGauvin(float v0, float a0, float a1, float a2,
                 float a3);
   // constructeur de copie virtuel
```

```
    virtual CrvCurve* copyCurve() {
       return new CrvCompGauvin(*this);}
    }

    // evaluation
    int evaluate(const float *x, int nbPos, float *values,
           int order=0);

    // methodes pour l'edition simple de la courbe
    int handlesSimpleEdition(void) {return 1;}
    void getSimpleEditionParams(char ***params, float **values,
            int *nbParams);
    void resetWithSimpleEditionParams(float *params);

    // lecture et écriture dans un arbre
    int read(IONode* node);
    Liste<IONode>* write();

    // identification du nom de la classe
    virtual char *getTypeString() {return "Compaction_Gauvin";}
private :

    float myCoefs[5];
    char *myCoefsDescr[5];

};
```

Le constructeur de `CrvCompGauvin` ne fait qu'initialiser un tableau de coefficients.

ainsi que leur description textuelle :

```
CrvCompGauvin::CrvCompGauvin(float v0, float a0, float a1, float
a2, float a3) {

    myCoefs[0] = v0;
    myCoefs[1] = a0;
    myCoefs[2] = a1;
    myCoefs[3] = a2;
    myCoefs[4] = a3;
    myCoefsDescr[0] = "v0";
    myCoefsDescr[1] = "a0";
    myCoefsDescr[2] = "a1";
    myCoefsDescr[3] = "a2";
    myCoefsDescr[4] = "a3";

}
```

La méthode la plus importante est bien entendu `evaluate()`. Voici son implémentation :

```
int CrvCompGauvin::evaluate(const float *x, int nbPos, float
*values, int order) {

   if (order != 0) return 1;  // le calcul des dérivées
                              // n'est pas implémenté

   for (int i=0; i<nbPos; i++) {
      float v0 = myCoefs[0];
      float a0 = myCoefs[1];
      float a1 = myCoefs[2];
      float a2 = myCoefs[3];
      float a3 = myCoefs[4];
      values[i] = 10.*v0/(10.*a0*x[i] + a1*log(10.*x[i]) +
                  a2/(10.*x[i]) + a3;
   }

   return 0;

}
```

Les méthodes suivantes concernent l'édition des coefficients de la classe `CrvCompGauvin`. Tout d'abord, on permet à un client d'obtenir l'information textuelle et numérique décrivant l'instance :

```
void CrvCompGauvin::getSimpleEditionParams(char ***coefsDescr,
float **coefs, int *nbCoefs) {

   *coefsDescr = &myCoefsDescr[0];
   *coefs = &myCoefs[0];
   *nbCoefs = 5;

}
```

Le post-constructeur d'édition copie simplement le tableau de coefficients reçu en paramètre :

```
void CrvCompGauvin::resetWithSimpleEditionParams(float *coefs) {

    myCoefs[0] = coefs[0];
    myCoefs[1] = coefs[1];
    myCoefs[2] = coefs[2];
    myCoefs[3] = coefs[3];
    myCoefs[4] = coefs[4];

}
```

En ce qui concerne la représentation graphique de cette classe, on peut se contenter des méthodes fournies par la classe `CrvExplicit`.

Les méthodes de lecture et d'écriture dans un arbre sont les suivantes :

```
void CrvCompGauvin::read(IONode *node) {

    IONode *na0, *na1, *na2, *na3, *nv0;

    na0 = FindToken(node->content, "A0");
    na1 = FindToken(node->content, "A1");
    na2 = FindToken(node->content, "A2");
    na3 = FindToken(node->content, "A3");
    nv0 = FindToken(node->content, "V0");

    myCoefs[0] = ReadFloat(nv0);
    myCoefs[1] = ReadFloat(na0);
    myCoefs[2] = ReadFloat(na1);
    myCoefs[3] = ReadFloat(na2);
    myCoefs[4] = ReadFloat(na3);

}

List<IONode>* CrvCompGauvin::write() {

    List<IONode>* list = new List<IONode>;
    IONode nodeFamily, nodeType, nodeA0, nodeA1, nodeA2, nodeA3,
           nodeV0;

    nodeFamily = WriteString("FAMILY", "Curve");
    nodeType = WriteString("TYPE", "CompGauvin");
    nodeV0 = WriteFloat("V0", myCoefs[0]);
    nodeA0 = WriteFloat("A0", myCoefs[1]);
    nodeA1 = WriteFloat("A1", myCoefs[2]);
    nodeA2 = WriteFloat("A2", myCoefs[3]);
```

```
    nodeA3 = WriteFloat("A3", myCoefs[4]);

    // insertion des jetons en fin de liste
    list->insertEnd(nodeFamily);
    list->insertEnd(nodeType);
    list->insertEnd(nodeV0);
    list->insertEnd(nodeA0);
    list->insertEnd(nodeA1);
    list->insertEnd(nodeA2);
    list->insertEnd(nodeA3);

return list;

}
```

Pour que cette classe soit disponible dans Dataflot et FLOT, on doit de plus procéder aux modifications suivantes :

- Ajouter un test dans la fonction de lecture générale d'une courbe `ReadCurve()` (voir le paragraphe 2.4.2).

- L'utiliser comme courbe par défaut dans les sélecteurs de courbes (voir le paragraphe 6.8) concernés de Dataflot (panneau des propriétés physiques des matériaux).

- Ajouter une option pour cette courbe dans le menu **New** de l'éditeur de courbes.

6.10 Résumé sur l'utilisation présente et à venir de la bibliothèque de fonctions

La figure 6.8 montre une vue d'ensemble de l'utilisation présente des courbes et des surfaces dans Dataflot et FLOT. On remarque sur cette figure que chaque classe d'interface utilisant une courbe ou une surface est associée à la classe `IcSelector` ou

`IsSelector`. Ces classes ont été décrites à la section 6.8. Elles gèrent l'interface graphique permettant l'association d'une courbe ou d'une surface à un paramètre de simulation.

Nous avons discuté de l'intégration des courbes et des surfaces aux résines à la section 6.2. Elles permettent l'évaluation polymorphe de la fonction de viscosité de la résine et de son taux de polymérisation.

La compaction d'une préforme est une courbe reliant la pression appliquée au taux de fibre. Cependant, cette courbe dépend du nombre de plis de la préforme. Lemenn (1994) montre que l'influence du nombre de plis sur la courbe de compaction n'est pas négligeable, pour un nombre de plis assez faible. En pratique, la courbe de compaction tend rapidement vers une certaine limite en fonction du nombre de plis. C'est pourquoi, bien que nous devrions théoriquement utiliser une surface afin d'évaluer $v_f=f(P, n)$, où v_f est la fraction volumique de fibres, P est la pression appliquée et n est le nombre de plis, nous pensons qu'il est plus approprié d'avoir recours à une courbe. D'abord la définition d'une surface est beaucoup plus complexe pour l'utilisateur que la définition d'une courbe. Ensuite, le nombre de plis ne varie pas en cours de simulation. En fait, l'utilisation d'une fonction de deux variables est intéressante pour l'utilisateur lorsqu'elle provient d'une base de données. Par contre, s'il doit lui-même la construire, la tâche lui apparaîtra trop lourde. Notons qu'au moment de la rédaction de ce mémoire,

le code de calcul FLOT n'a pas encore été modifié pour tenir compte de la compaction de la préforme.

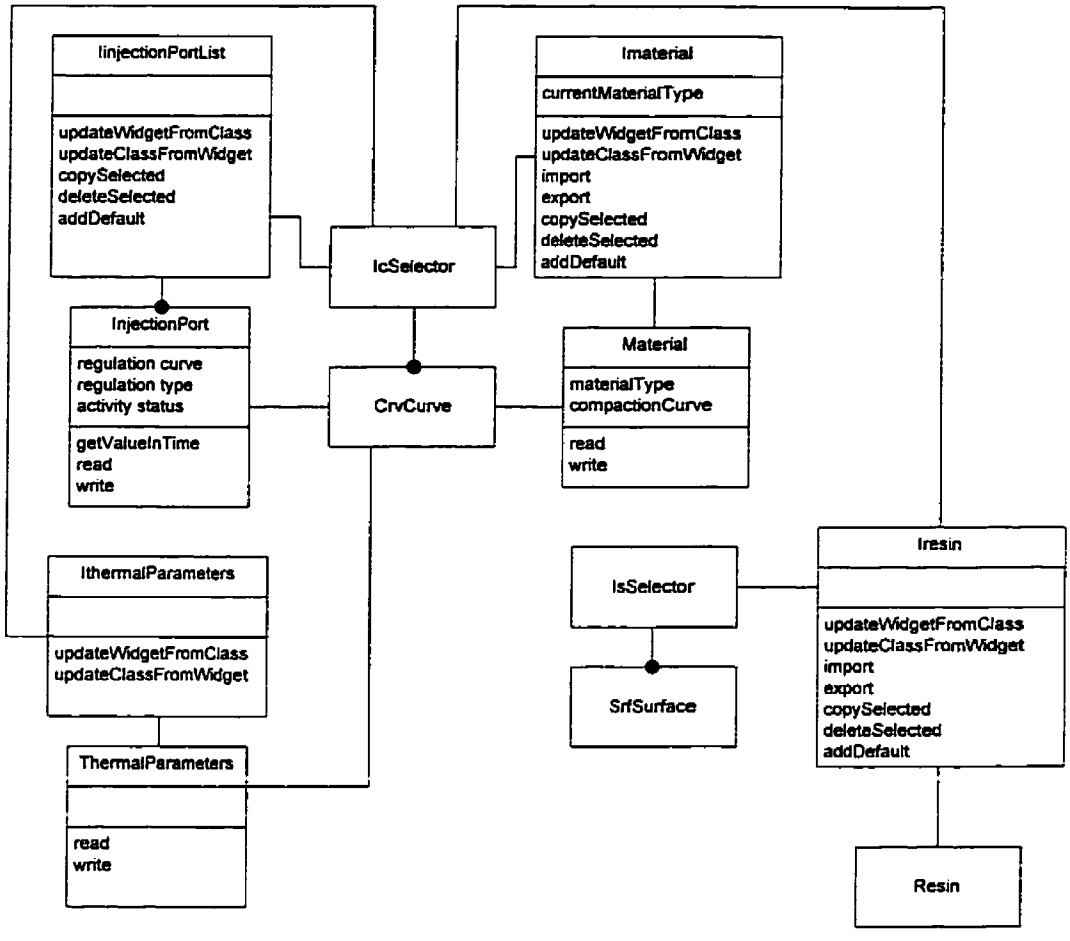


Figure 6.8 Vue d'ensemble de l'utilisation présente des courbes et des surfaces par Dataflot et Flot

Un point d'injection est aussi associé à une courbe, ce qui permet de décrire l'évolution de la pression imposée ou du débit imposé dans le temps. Bien que ces courbes soient générales, en pratique nous avons recours le plus souvent aux fonctions linéaires par morceaux et constantes par morceaux.

La simulation thermique utilise elle-aussi la bibliothèque de courbes. On remarque sur la figure 5.8 la présence de trois zones intitulées **Prefilling**, **Filling** et **Curing** (non visible sur la figure). Chaque zone contient trois courbes décrivant l'évolution dans le temps des conditions frontières thermiques de la cavité (température, flux et coefficient d'échange). Ainsi, il est désormais possible de simuler la polymérisation de la résine sous l'influence de rampes thermiques.

Parmi les utilisations futures des bibliothèques de courbes et de surfaces, on retrouve la possibilité d'associer une courbe "binaire" à l'activité d'un point d'injection. Une telle courbe se construit rapidement à partir d'une courbe constante par morceaux. On pourra ainsi programmer l'ouverture et la fermeture de points d'injection dans le temps.

Plusieurs paramètres physiques d'importance "secondaire" peuvent être modélisés par une courbe. On pense, par exemple, à la chaleur spécifique d'une résine, à la chaleur spécifique des fibres, à la conductivité effective des fibres, fonctions de la température qui sont présentement limitées aux polynômes de degré 0, 1, et 2. Dans l'implémentation courante de Dataflot, ces polynômes ne proviennent pas de la bibliothèque de courbes. On pourra éventuellement les remplacer par des courbes générales, mais pour l'instant, nous ne voyons pas d'urgence à une telle généralisation.

6.11 Reconception de la gestion de base de données dans Dataflot

Avec le recul, nous constatons qu'un travail de reconception des panneaux d'édition des propriétés physiques des matériaux (moule, tube, tissu) et du panneau des résines s'impose. Rappelons qu'il s'agit présentement de deux panneaux distincts, mais qui possèdent essentiellement les mêmes fonctionnalités de gestion d'une base de données (importation, exportation, copie d'un objet, destruction d'un objet, etc.). Seul le type d'objet considéré diffère. Dans un cas, il s'agit d'instances de la classe `Resin`, dans l'autre d'instances de la classe `Material`. L'implémentation actuelle présente certains inconvénients pour le futur : chaque nouveau type d'objet que l'on souhaitera gérer à l'aide de ces mêmes fonctionnalités de base de données nécessitera la définition d'un nouveau panneau, ce qui demande un effort de programmation non négligeable et qui augmente inutilement la taille du code exécutable. De plus, la duplication de fonctionnalités est très souvent source d'erreurs.

Il est toutefois envisageable de fusionner la gestion de tous les objets de base de données en un seul panneau. Tout ce que nous présentons par la suite dans cette section n'a pas encore été implémenté. Il s'agit d'un travail de conception.

Tout d'abord, l'interface du gestionnaire généralisé hypothétique est très semblable à la version actuelle (figure 5.4). Elle consiste en deux zones : la partie gauche de l'interface sert toujours à sélectionner un objet dans une liste, tandis que la partie droite permet

d'éditer l'objet sélectionné. Cependant, puisque le gestionnaire traite les résines, les propriétés physiques des zones et éventuellement de nouvelles classes d'objets, nous rajoutons un menu à options nommé **Filter**, permettant de filtrer par classe les objets de la liste. Les options de ce menu sont donc pour l'instant `Resin`, `Mold`, `Tube` et `Fabric`.

La fonctionnalité **Import** charge dans le gestionaire tous les objets contenus dans un fichier spécifié par l'usager, et reconnus par le gestionnaire. La fonctionnalité **Export** enregistre sur disque tous les objets que contient le gestionnaire. Notons que ces fonctionnalités s'appliquent à toutes les classes d'objets reconnus par le système. Autrement dit, l'état du filtre n'a pas d'influence sur la classe des objets lus ou écrits.

Afin d'obtenir un gestionnaire général et facile à maintenir, il ne doit pas manipuler des classes spécifiques telles que `Resin` ou `Material`. Plutôt, il doit "voir" des objets aux propriétés identiques. C'est pourquoi nous introduisons une nouvelle classe que nous appellerons `DBobject` ("database object"). Le gestionnaire traite une liste d'instances de `DBobject`, sans savoir s'il s'agit d'une résine ou d'un autre objet. La figure 6.9 donne une vue d'ensemble des classes impliquées dans cette nouvelle architecture.

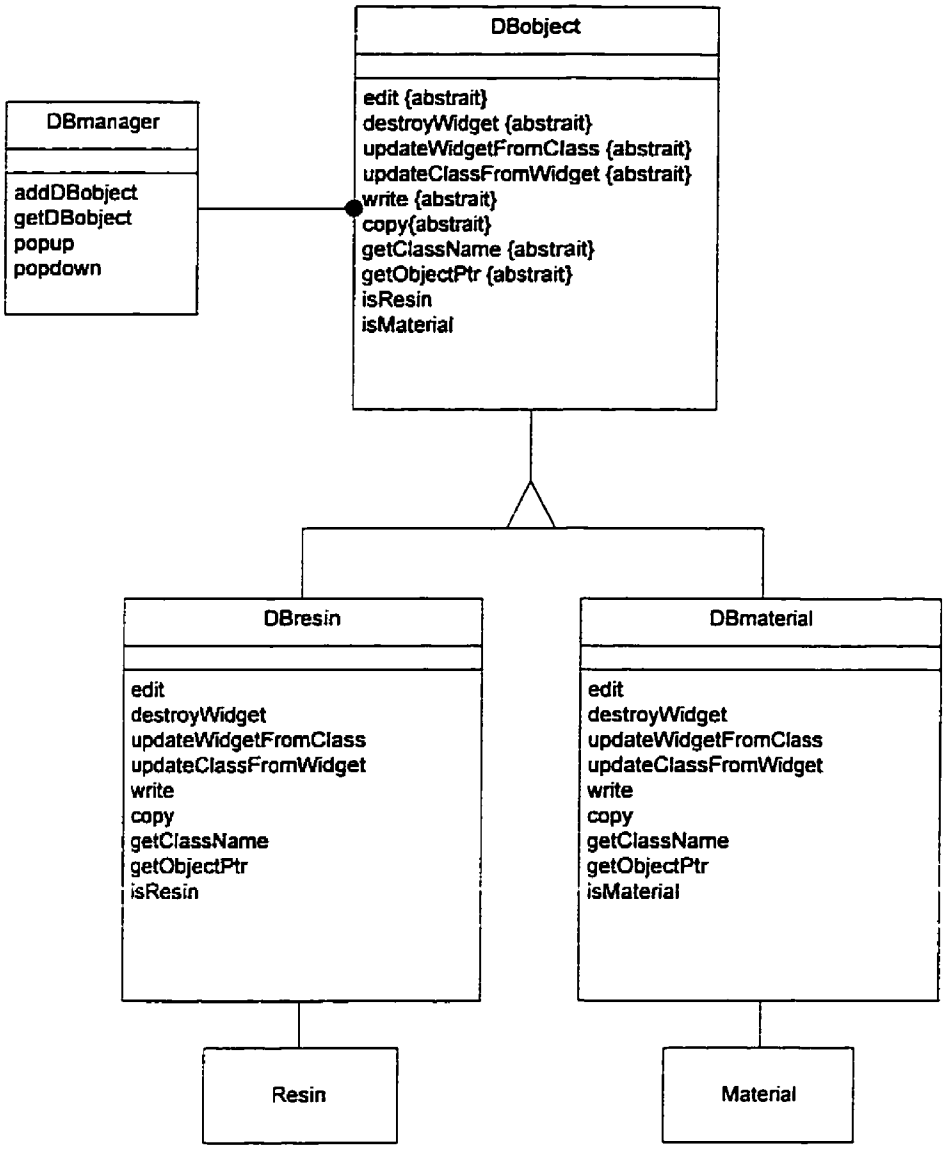


Figure 6.9 Architecture proposée pour la généralisation du gestionnaire de base de données

classe :

`DBobject`

rôle :

classe d'interface fournissant les opérations dont a besoin le gestionnaire de base de données de Dataflot pour permettre l'édition des attributs d'un objet de calcul, le copier et l'enregistrer sur disque.

principales opérations :

- `edit(scroll window)` :
  affichage (construction au besoin) de l'interface graphique ce cet objet. Le paramètre "scroll window" est utilisé comme "parent" de l'interface.
- `destroyWidget` :
  détruit l'interface graphique.
- `updateClassFromWidget` :
  mise-à-jour de l'objet de calcul à partir du contenu de l'interface.
- `write` :
  écriture de l'objet de calcul sur disque.
- `copy` :
  retourne un "clone" d'une instance de `DBobject`.
- `getClassName` :
  retourne une chaîne de caractères identifiant la classe de calcul.
- `getObjectPtr` :
  retourne un pointeur à l'objet de calcul.
- `isResin, isMaterial, ...` :
  identification de l'objet de calcul associé à `DBobject`.

Ainsi, chaque instance de `DBobject` construit et possède sa propre interface graphique.

Les descendants de `DBobject` sont `DBmaterial` et `DBresin`, qui redéfinissent pratiquement toutes les opérations de `DBobject`.

La partie droite de l'interface du gestionnaire est un "scroll window". Une instance de `DBobject` peut donc créer son interface en utilisant ce "scroll window" comme parent,

sans trop se soucier de la taille de la fenêtre à l'écran. Une instance de `DBobject` doit cependant construire une interface graphique "raisonnable", c'est-à-dire que sa largeur devrait être compatible avec celle du "scroll window" qui la contiendra.

Le bouton **Apply** de l'interface du gestionnaire transfère le contenu de l'interface à l'objet de calcul. Pour ce faire, le gestionnaire appelle simplement la méthode `updateClassFromWidget()` de l'objet courant.

Lorsque l'utilisateur sélectionne un objet dans la partie gauche de l'interface, le gestionnaire appelle d'abord la méthode `destroyWidget()` de l'objet courant et appelle ensuite la méthode `edit()` de l'instance sélectionnée, ce qui affiche l'interface graphique de l'objet sélectionné.

Pour lire une base de données, le gestionnaire utilise la fonction `readDBobjects()`, qui retourne une liste de `DBobject`. Cette fonction reconnaît les objets de calcul qui peuvent être associés à une classe dérivée de `DBobject`. Elle construit donc des objets de calcul qu'elle associe à une instance de `DBobject`. Par exemple, si elle détecte une résine dans un fichier, elle construit une instance de la classe `Resin` qu'elle associe aussitôt à une instance de la classe `DBresin`.

---

classe :

`DBmanager`

rôle :

encapsuler l'interface graphique du gestionnaire de base de données de Dataflot.

principaux attributs :

`objectList` : liste d'instances de la classe `DBobject`.

principales opérations :

- `addDBobject` :
  ajout d'un objet, qui devient l'objet courant.
- `getDBobject` :
  retourne une copie de l'objet courant.
- `setClientOkCallback, setClientCancelCallback` :
  permet à un client de spécifier la fonction associée au bouton Ok ainsi qu'au bouton Cancel.
- `popup(), popdown()` :
  ouvre, ferme l'interface graphique.

D'un point de vue client, l'utilisation de la classe `DBmanager` est relativement simple. Supposons que le panneau principal de Dataflot soit client de `DBmanager` pour l'édition d'une résine. Le panneau principal crée d'abord une instance de `DBresin`, à laquelle il associe sa résine courante. Puis il appelle `DBmanager::addDBobject()` avec l'objet `DBresin` passé en paramètre. Lorsque l'usager a terminé l'édition de la résine, le panneau principal en est informé et il récupère l'objet courant du gestionnaire avec `getDBobject()`.

La généralisation du gestionnaire de base de données de Dataflot n'est pas urgente, puisqu'il traite pour l'instant uniquement les résines et les matériaux. Cependant lorsque de nouveaux objets seront ajoutés au système, les concepts développés dans cette section pourront servir de point de départ pour un travail de reconception.

Chapitre 7

Simulation du remplissage et de la cuisson d'une pièce moulée par RTM

Nous présentons dans ce chapitre une simulation thermique simple illustrant certains concepts présentés dans ce mémoire. Nous développons un modèle de cinétique de polymérisation krigée à partir de données expérimentales. La viscosité de la résine est décrite par un modèle paramétrique faisant intervenir la température et le degré de polymérisation. Enfin, nous simulons la cuisson de la pièce sous l'influence d'une rampe de température. L'annexe I présente le fichier Dataflot utilisé pour réaliser cette simulation.

7.1 Paramètres de simulation

7.1.1 Géométrie et conditions frontières

Nous simulons le moulage d'une plaque rectangulaire dont les dimensions sont 0,3x0,15x0,04 mètres. La température est imposée sur toutes les faces de la pièce. Puisque le problème a deux plans de symétrie, nous maillons seulement le quart de la pièce, tel que montré sur la figure 7.1. L'injection est du type « ligne », sur le plan y=0. Le maillage est constitué de 4 couches d'éléments dans l'épaisseur et de 30x30 éléments

dans le plan x-y. Le débit d'injection constant de 6E-7 m^3/s permet le remplissage du domaine maillé en 5 minutes, compte tenu d'une porosité de 40%.

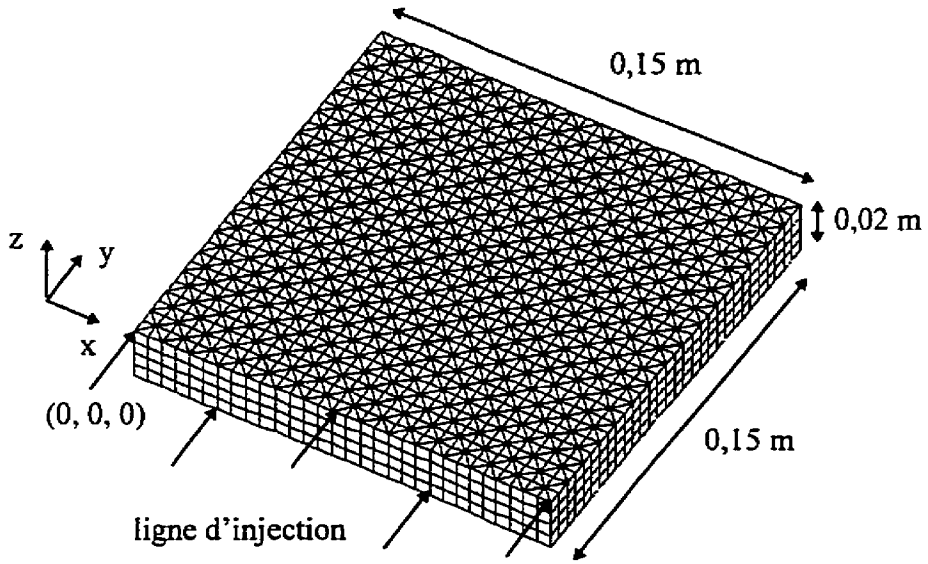


Figure 7.1 Maillage du quart de la cavité

7.1.2 Résine

La résine utilisée est de type époxy. Nous disposons de mesures de DSC (« Differential Scanning Calorimetry ») pour modéliser sa cinétique de polymérisation. Ceci nous permet de définir un modèle de cinétique chimique interpolé par krigeage. Nous utilisons les courbes de DSC montrées à la figure 7.2 que nous intégrons numériquement. La figure 7.3 montre le résultat de cette intégration.

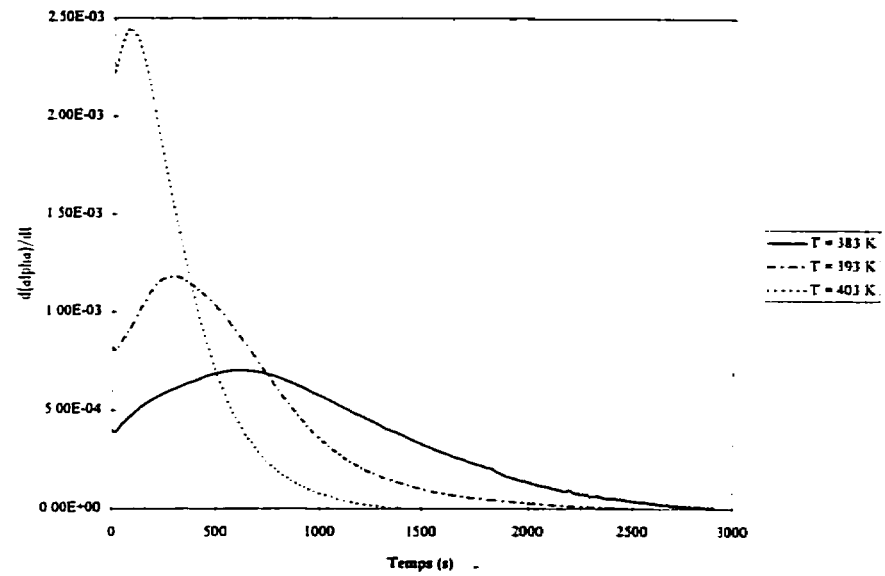


Figure 7.2 Taux de conversion en fonction du temps

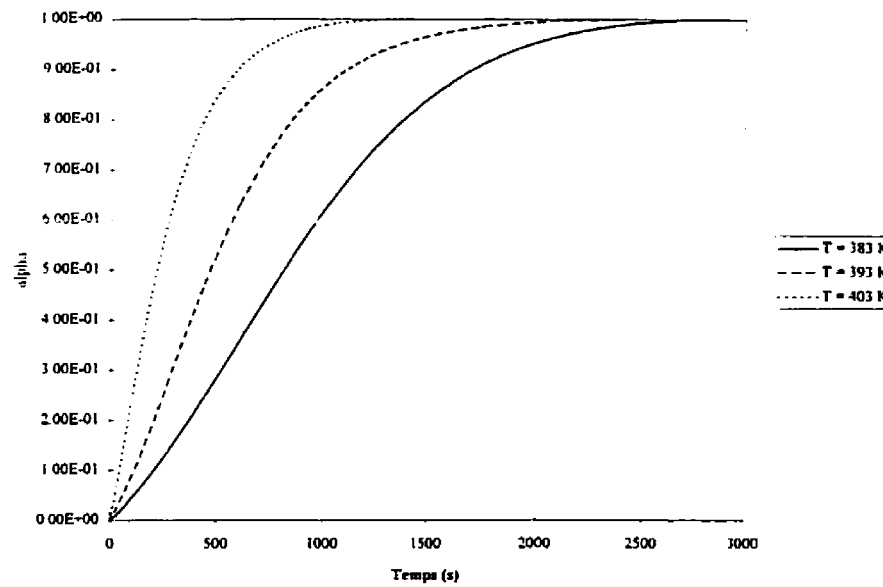


Figure 7.3 Degré de conversion en fonction du temps

À partir des courbes de DSC et des courbes intégrées, nous échantillonnons un certain nombre de points, de coordonnées (température, degré de conversion, vitesse de conversion). La vitesse de conversion est la variable dépendante. La définition en format Dataflot de la surface krigée résultant de cet échantillonnage est la suivante :

```
DSC_SURFACE {
   family : Surface;
   type : Krig_E;
   control_points {
      family : PointSet;
      type : PointSet1D;
      Points {
         dimension : 3;
         number : 30;
         data : \
            383       0.          4.0E-4 \
            383       0.163       6.16E-4 \
            383       0.367       7.05E-4 \
            383       0.592       5.93E-4 \
            383       0.844       3.22E-4 \
            383       0.955       1.34E-4 \
            383       0.993       4.03E-5 \
            383       1.          7.71E-6 \
            393       0.          8.18E-4 \
            393       0.124       1.02E-3 \
            393       0.212       1.15E-3 \
            393       0.297       1.18E-3 \
            393       0.439       1.11E-3 \
            393       0.604       9.23E-4 \
            393       0.883       3.11E-4 \
            393       0.930       1.89E-4 \
            393       1.          2.4E-6 \
            403       0.          2.23E-3 \
            403       0.0337      2.35E-3 \
            403       0.177       2.44E-3 \
            403       0.402       2.16E-3 \
            403       0.733       1.16E-3 \
            403       0.908       3.72E-4 \
            403       0.988       7.39E-5 \
            320       0.          0. \
            320       0.2         0. \
            320       0.4         0. \
            320       0.6         0. \
            320       0.8         0. \
```

```
                320         1.0         0.;
        }
    }

}
```

En ce qui concerne la viscosité de la résine, le fabricant suggère le modèle paramétrique suivant, qui tient compte du degré de polymérisation :

$$\eta(T,\dot{\gamma},\alpha) = f_2(\dot{\gamma}, f_1(T)) \cdot f_3(\alpha), \tag{7.1}$$

$$f_1(T) = Be^{(T_b/T)} = \eta_0, \tag{7.2}$$

$$f_2(\dot{\gamma}) = \frac{\eta_0}{1+\left[\frac{\eta_0\dot{\gamma}}{\tau^*}\right]^{(1-n)}}, \tag{7.3}$$

$$f_3(\alpha) = \left[\frac{\alpha_{gel}}{\alpha_{gel}-\alpha}\right]^{C_1+C_2\alpha}, \tag{7.4}$$

où η_0 est la viscosité à la température T avec une contrainte de cisaillement nulle, α_{gel} est le degré de conversion au gel. Les constantes du modèle sont :

B	=	7,54E-10	(Pa.s)
T_b	=	6517	(K)
C_1	=	2,611	
C_2	=	-1,057	
α_{gel}	=	0,7152	
τ^*	=	1E+17	(Pa)

On considère nul le taux de cisaillement $\dot{\gamma}$ de la résine pour les faibles vitesses d'injection utilisées dans le procédé RTM. Les équations 7.1 à 7.4 se résument donc à :

$$\eta(T,\alpha) = Be^{(T_b/T)}\left[\frac{\alpha_{gel}}{\alpha_{gel}-\alpha}\right]^{C_1+C_2\alpha} \quad (7.5)$$

La bibliothèque de surfaces définit la classe `SrfExpo5`, qui implémente l'évaluation de l'équation 7.5. Sa définition dans le format Dataflot est :

```
EA9150_Viscosity_Model {
   family : Surface ;
   type : Expo_5 ;
   K1 : 7.54E-10 ;
   K2 : 6517 ;
   K3 : 0.7152 ;
   K4 : 2.611 ;
   K5 : -1.057 ;
}
```

Les paramètres physiques suivants complètent la définition de cette résine :

- masse volumique : 1250 kg/m^3
- chaleur spécifique : 2000 J/kg.K
- conductivité thermique : 2,325 W/m.K

7.1.3 Préforme

La préforme est constituée de mats unidirectionnels de carbone. Sa porosité est de 40% lorsque le moule est fermé. Les perméabilités K_x et K_y utilisées dans la simulation sont

respectivement de 2,25E-11 m^2 et 1,4E-11 m^2 pour cette porosité. Les paramètres physiques des fibres sont :

- masse volumique : 1800 kg/m^3
- conductivité thermique : 7,788 W/m.K
- chaleur spécifique : 1088,5 J/kg.K

7.1.4 Paramètres thermiques

On injecte la résine à la température de 322 K dans un moule préchauffé à 338 K. La température sur toute la surface extérieure de la pièce est maintenue constante à 338 K pendant le remplissage. On nous demande ensuite d'imposer le cycle de température suivant pendant la cuisson de la pièce :

- Montée uniforme de la température sur la surface extérieure de la pièce, de 338 K à 394 K en 25 minutes.
- Température de 394 K maintenue constante pendant une heure.
- Descente uniforme de 394 K à 310 K en 15 minutes.

La spécification de ce cycle thermique dans Dataflot est réalisée à l'aide des deux courbes suivantes, qui peuvent être construites à partir de l'éditeur de courbes. La première est associée à la phase de remplissage et la seconde à la phase de cuisson.

```
fil_temp_curve {
   FAMILY : Curve ;
   TYPE : Constant ;
   VALUE : 1.09 ;
}

post_temp_curve {
   FAMILY : Curve ;
   TYPE : Piecewise_Linear ;
   Control_Points {
      FAMILY : PointSet;
      TYPE : PointSet1D ;
      POINTS {
         DIMENSION : 2 ;
         NUMBER : 4 ;
         DATA : \
            0          1.09 \
            1500       1.27 \
            5100       1.27 \
            6000       1;
      }
   }
}
```

Le lecteur remarquera que la valeur de la courbe constante ainsi que les points de contrôle de la courbe linéaire par morceaux ne se trouvent pas dans l'intervalle de température spécifié, soit 338 K à 394 K. Il s'agit d'une particularité de la verion actuelle du logiciel. Ces courbe *multiplient* les condition frontières spécifiées dans un fichier externe. Dans notre cas, cette valeur est de 310 K. La courbe constante dont la valeur est 1.09 multiplie 310 K pendant le remplissage. La condition frontière de température imposée est donc 338 K pendant le remplissage.

7.2 Résultats de simulation

Les figures 7.4 à 7.9 montrent quelques résultats de simulation obtenus à partir des paramètres décrits à la section précédente. Les figures 7.4 à 7.8 ont été générées sur un

support de visualisation situé sur la section médiane dans l'épaisseur de la pièce (z=0,02 m). La figure 7.4 montre le champ de température et le remplissage après 235 secondes d'injection. Notons sur cette figure le retard du front thermique sur le front de résine, ainsi que la présence d'oscillations numériques à proximité de la paroi chauffée de la pièce. La figure 7.5 montre que le degré de polymérisation est négligeable à la fin de l'injection. Les figures 7.6 à 7.8 illustrent la variation de la température sur la paroi extérieure du moule pendant la cuisson. La figure 7.7 a été obtenue au cours de la période de chauffe tandis que la figure 7.8 a été obtenue pendant la phase de refroidissement de la pièce. Finalement, la figure 7.9 montre le champ de température dans l'épaisseur de la pièce pendant la cuisson. On note une différence de température de 30 K entre la paroi et le centre de la pièce.

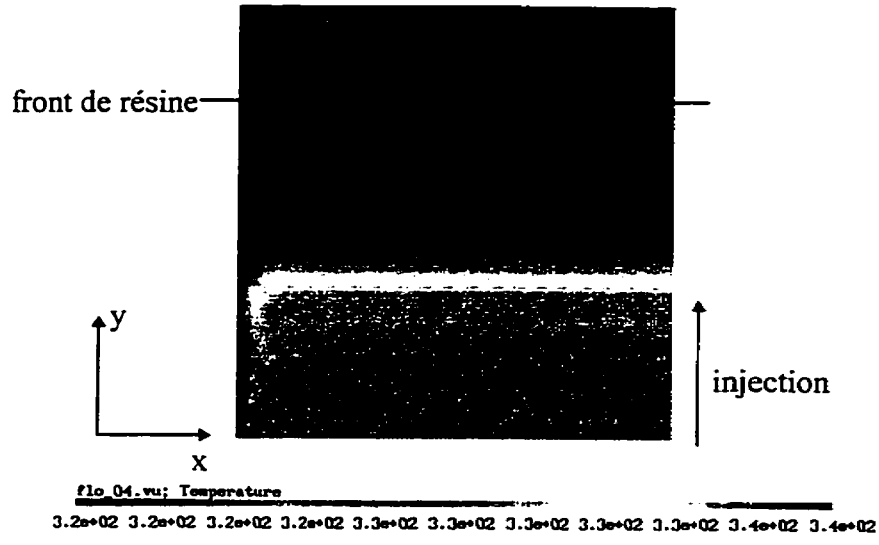


Figure 7.4 Température à 235 s. (z=0,02)

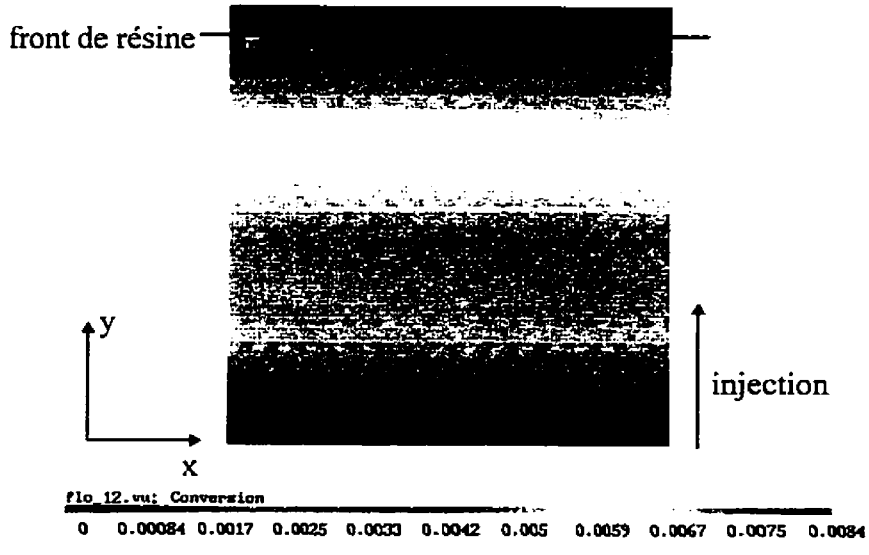


Figure 7.5 Conversion à 285 s. (z=0,02)

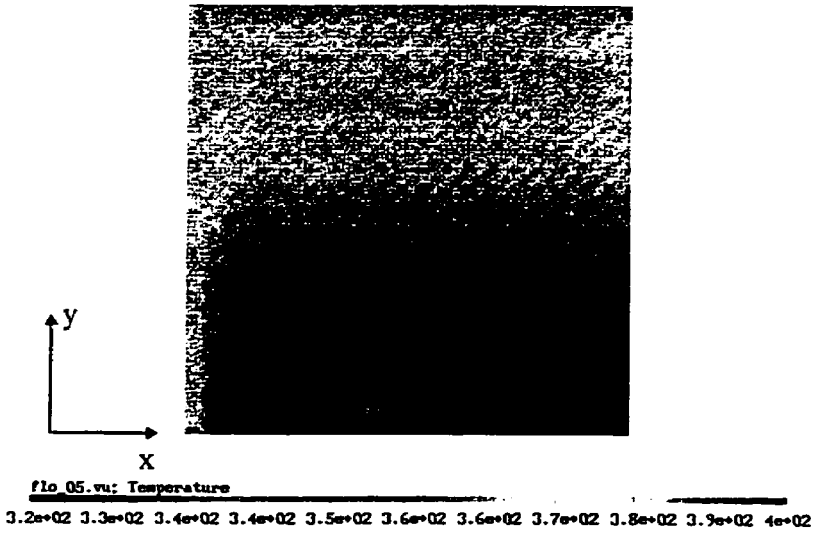


Figure 7.6 Température à t=1020 s. (z=0,02)

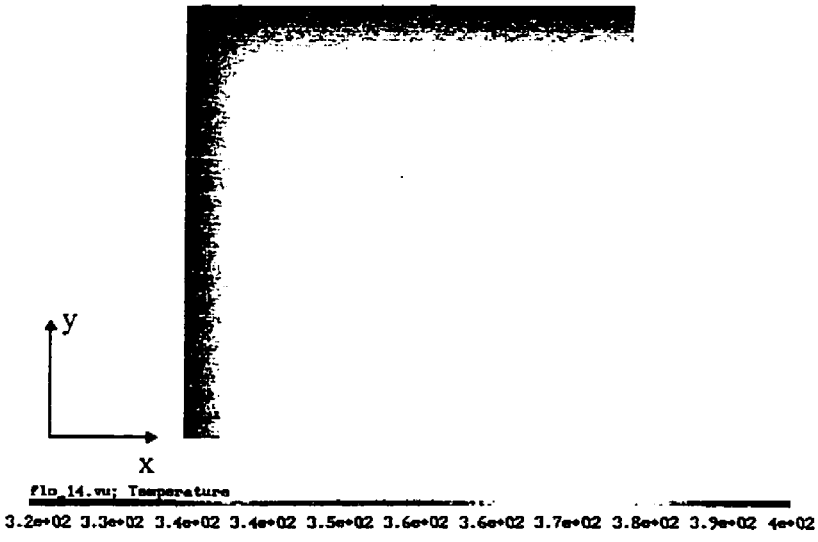


Figure 7.7 Température à t=2820 s. (z=0,02)

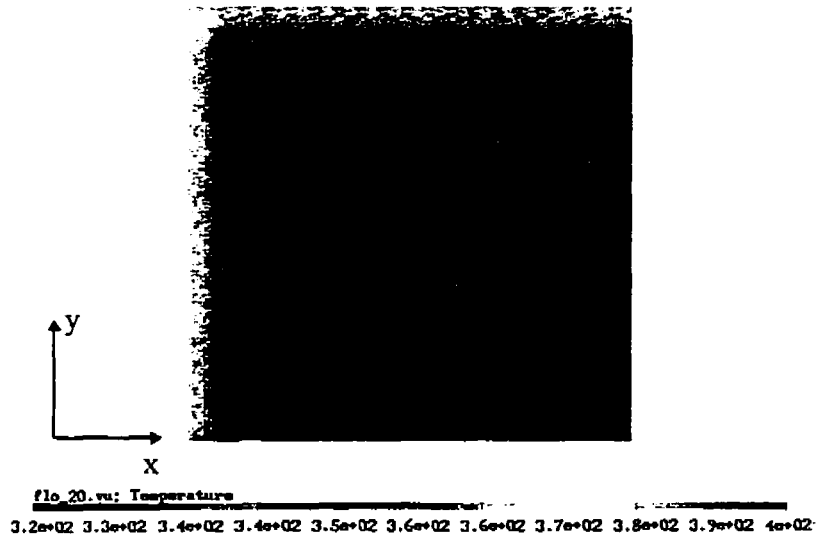


Figure 7.8 Température à t=5850 s. (z=0,02)

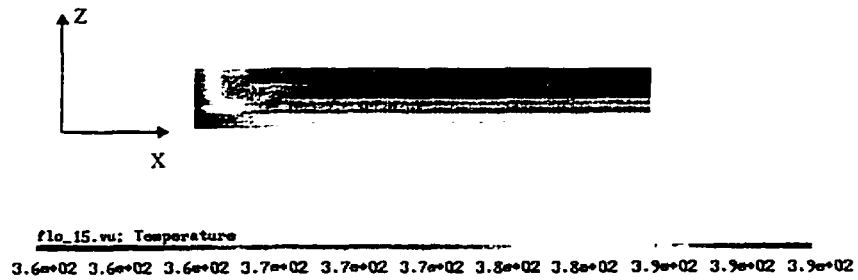


Figure 7.9 Température à t=2100 s. (y=0)

7.3 Discussion

Le résine injectée est très peu réactive dans la plage de température de 322 K à 338 K. Ceci nous empêche de mettre en évidence les différences de polymérisation induites par le front thermique pendant le remplissage. Malgré tout, il nous semble anormal d'obtenir un champ de conversion stratifié comme celui de la figure 7.5, puisque la résine qui longe la paroi extérieure du moule est soumise à une température élevée plus longtemps que la résine du centre de la pièce.

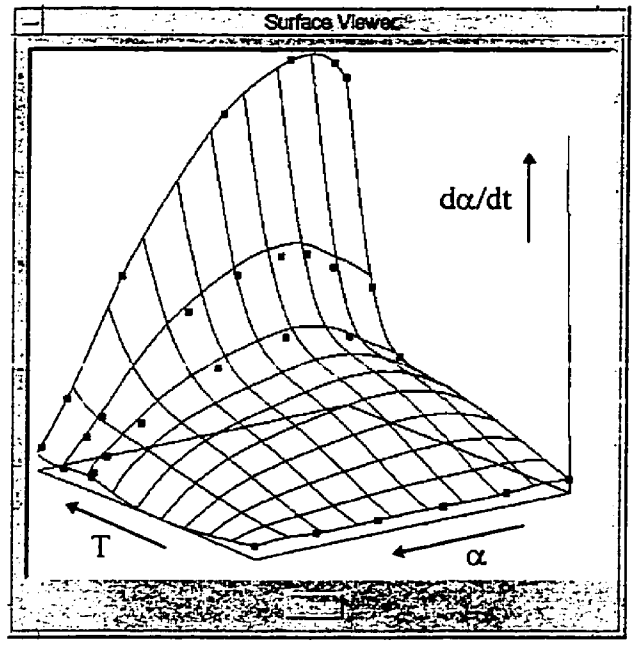


Figure 7.10 Cinétique krigée

Le krigeage de la cinétique de polymérisation nécessite quelques précisions. Nous n'avions à notre disposition que trois courbes de DSC, pour des températures de 383 K, 393 K et 403 K. Ceci pose un problème puisque la température du moule est de 340 K,

ce qui signifie que la vitesse de réaction doit être extrapolée pendant toute la phase de remplissage. Nous avons constaté la présence d'oscillations numérique causées par cette extrapolation. Pour résoudre ce problème, nous avons ajouté au modèle krigé une courbe isotherme T = 320 K sur laquelle nous imposons une vitesse de réaction nulle. Ainsi toute la plage de températures est couverte par la surface krigée et nous évitons l'extrapolation. La figure 7.10 montre le modèle de cinétique krigée résultant (surface $\dot{\alpha} = f(T,\alpha)$). Sur cette figure, l'échelle que l'on devrait lire sur les différents axes est $320K \leq T \leq 393K$, $0 \leq \alpha \leq 1$ et $0 \leq \dot{\alpha} \leq 2.44E-3$. La figure 7.11 montre 5 courbes de cinétique isothermes pour des températures comprises entre 320 K et 340 K. Ceci nous permet de constater visuellement, à l'aide du visualisateur de cinétique de Dataflot, l'absence d'oscillations sur cette plage de température.

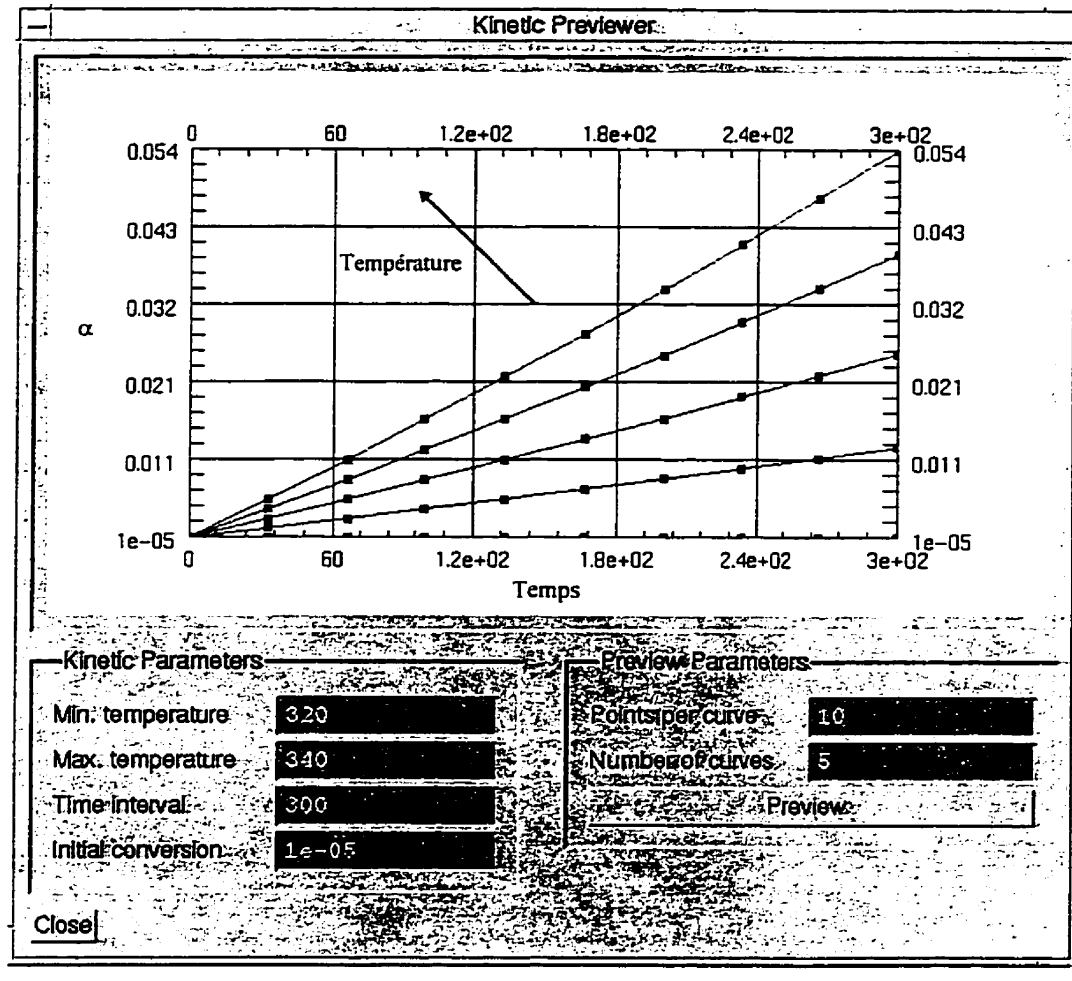


Figure 7.11 Cinétique chimique pendant la phase de remplissage

Conclusion

Nous avons d'abord présenté dans ce mémoire quelques modèles paramétriques tirés de la littérature et qui décrivent les principaux paramètres physiques impliqués dans la simulation numérique du procédé RTM. Des modèles de compaction d'une préforme, de viscosité et de cinétique de polymérisation d'une résine ont été décrits. Nous avons aussi mentionné les variables importantes dans la modélisation de la perméabilité et les conséquences du drapage sur la porosité. Cette revue nous amène à la constatation qu'il existe un grand nombre de modèles paramétriques mais qu'ils sont tous relativement coûteux à caractériser. L'interpolation constitue alors une alternative intéressante à ces modèles.

Afin d'augmenter le niveau de réalisme des simulations, nous avons conçu une bibliothèque de courbes et de surfaces pouvant être associées à une multitude de paramètres de simulation. Ces fonctions implémentent différents modèles paramétriques ainsi que des modèles interpolés généraux, tels que le krigeage dual. Les utilisateurs du logiciel peuvent donc choisir de lier à un paramètre de simulation un modèle paramétrique de forme prédéfinie, ou encore une fonction interpolée. Parmi les avantages de la conception orientée objet de cette bibliothèque, nous insistons sur la facilité avec laquelle on peut désormais intégrer de nouveaux modèles paramétriques au logiciel.

Nous avons ensuite décrit la bibliothèque graphique, fondée sur le langage graphique OpenGL, que nous avons développée en réponse aux nombreux besoins de visualisation 3D de ce projet. Cette bibliothèque définit entre autres un visualisateur de scène 3D. Elle fait appel à OpenGL pour la sélection d'objets graphiques. La sélection étendue en 3D s'est avérée particulièrement difficile avec OpenGL à cause du problème de la sélection involontaire d'objets cachés. Nous avons présenté quelques solutions à ce problème. Un éditeur de courbes et de surfaces a été développé à partir de cette bibliothèque.

Nous avons présenté la structure en arbre utilisée pour la lecture et l'écriture d'objets complexes. Tout objet complexe sait comment se reconstruire lui-même à partir de cet arbre. Ce protocole original nous a mené au développement d'une base de données orientée objet de paramètres de simulation.

La bibliothèque de courbes est présentement utilisée dans le code de calcul FLOT pour la spécification de la pression ou du débit d'injection dans le temps, pour la définition de la fonction de viscosité d'une résine, ainsi que pour l'imposition de conditions frontières thermiques variables dans le temps. La bibliothèque de surfaces est présentement utilisée pour définir une fonction de vitesse de polymérisation interpolée ainsi que pour définir des fonctions de viscosité dépendantes de la température et du degré de polymérisation de la résine.

Plusieurs développements sont envisageables à partir de la structure logicielle que nous avons présentée. Nous proposons l'utilisation d'une courbe temporelle afin de programmer l'ouverture et la fermeture des points d'injection et des évents. Nous devrons aussi tenir compte éventuellement du phénomène de compaction de la préforme autour du point d'injection qui se manifeste lorsque la résine est injectée sous haute pression. La structure logicielle est en place pour recevoir des courbes de compaction statique et des surfaces de compaction dynamique, mais un travail de conception important sera nécessaire au niveau du code de calcul.

Bibliographie

AUDET, M. (1996). Simulation numérique tridimensionnelle du transfert de chaleur dans les moules d'injection pour matériaux composites. Mémoire de maîtrise, École Polytechnique de Montréal, Canada.

BATCH, G.L. et CLIMSKEY, S. (1990). Multilayer compaction and flow in composites processing. 45th annual conference, composites institute, the society of the plastics industry. February 12-15, session 9-A.

BENOIT, Y. (1994). Rapport de stage. Document produit suite au stage à la société SNECMA, France.

BOOCH, G. (1994). Object-oriented analysis and design. Addison-Wesley, second edition.

CAMARERO, R. et GRANGER, L. (1990). Introduction à la conception assistée par ordinateur et à l'infographie. École polytechnique de Montréal.

ECKEL, B. (1993). C++ inside & out. McGraw-Hill.

FERGUSON, P.M. (1994). Motif reference manual. O'Reilly & Associates.

FERLAND, P. (1994). Simulation numérique non-isotherme du procédé de moulage par transfert de résine. Mémoire de maîtrise, École Polytechnique de Montréal, Canada.

FOUNTAIN, R., HAAS, T.W. (1975). Application of infrared spectroscopy to the cure of polyimide laminates. J. Appl. Polym. Sci., 19, 1767-1770.

GALLAGHER, R. S. (1995). Computer Visualization. CRC Press.

GAUVIN, R., CHIBANI, M. et LAFONTAINE, P. (1986). The modeling of pressure distribution in resin transfer molding. 41st annual conference, reinforced plastics/composites institute, the society of the plastics industry. January 27-31.

GAUVIN, R. et CHIBANI, M.. (1988). Modelization of the clamping force and mold filling in resin transfer molding. 43rd annual conference, composites institute, the society of plastics industry. February 1-5, session 22-C.

GERALD, C. et WHEATLEY, P. (1989). Applied numerical analysis. Addison-Wesley, fourth edition.

GONZALEZ-ROMERO, V.M. et CASILLAS, N. (1989). Isothermal and temperature programmed kinetic studies of thermosets. Polym. Eng. Sci., 29, 295-301.

HELLER, D. et FERGUSON, P.M. (1994). Motif programming manual. O'Reilly & Associates, second edition.

KAMAL, M.R., et SOUROUR, S. (1973). Kinetics and thermal characterization of thermoset cure. Polym. Eng. Sci., 13, 59-64.

KAMAL, M.R. (1974). Thermoset characterization for moldability analysis. Polym. Eng. Sci., 14, 231-239.

LEMENN, Y. (1994). Étude de la compressibilité et de la perméabilité des renforts directionnels. Mémoire de maîtrise, École Polytechnique de Montréal, Canada.

McMINDS, D. L. (1993). Mastering OSF/Motif widgets. Addison-Wesley, second edition.

NEIDER, J., DAVIS, T. et WOO, M. (1993). OpenGL programming guide. Addison-Wesley.

PARRETTE, W. A. (1993). Motif programming in the X Window system environment. McGraw-Hill.

TAYLOR, D. W. (1948). Fundamentals of soil mechanics. John Wiley and Sons, New York.

RUMBAUGH, J., BLAHA, M., EDDY, F., PREMERLANI, W. et LORENSEN, W. (1991). Object oriented modeling and design. Prentice Hall, Englewood Cliffs.

SCHACH, S. R. (1993). Software engineering. Aksen Associates, second edition.

SCHWARTZ, M. (1992). Composite materials handbook. McGraw-Hill, second edition.

SEDGEWICK, R. (1990). Algorithms in C. Addison-Wesley.

THOMAS, G., et FINNEY, R. (1988). Calculus and analytic geometry. Addison-Wesley, seventh edition.

TROCHU, F. (1993). Presentation of a contouring program based on dual kriging interpolation. Engineering with Computers. 9, 160.

TROCHU, F., HAMMAMI, A. et BENOIT, Y. (1996). Prediction of fibre orientation and net shape definition of complex composite parts. Composites Part A. 27, 319-328.

VAN DER WEEN, F. (1991). Algorithms for draping fabrics on doubly curved surfaces. Int. J. Num. Meth. Eng. 31, 1415.

WANG, K.J., HUANG, Y.J. et LEE, L.J. (1990). Reaction injection molding of polyureas. II: Rheo-kinetic changes and model simulation. Polym. Eng. Sci. 30, 654-664.

WATT, A., et WATT, M. (1992). Advanced animation and rendering techniques. Addison-Wesley.

WATT, A. (1989). Fundamentals of three-dimensional computer graphics. Addison-Wesley.

WERNECKE, J. (1994). The Inventor mentor. Addison-Wesley.

WIRFS-BROCK, R., WILKERSON, B. et WIENER, L. (1990). Designing object-oriented software. Prentice-Hall.

YOUSEFI, A. (1996). Cure analysis of promoted polyester and vinylester reinforced composites and heat transfer in RTM molds. Thèse de doctorat, École Polytechnique de Montréal, Canada.

Annexe A

Exemple de fichier Dataflot

Un fichier Dataflot contient 5 blocs d'information, correspondant aux 5 panneaux du logiciel. On retrouve dans l'ordre : paramètres généraux de simulation, résine, propriétés physiques des zones, paramètres thermiques et points d'injection :

```
General_Parameters {
      description : "";
      output_file : "these.flo";
      log_file : "these.log";
      inj_report_file : "these.prn";
      geometry_file : "these.geo";
      sim_type : Non_Isothermal;
      output_format : Flo1_Bin;
      darcy_elements : NonConforme;
      test_model : No;
      input_porosity_mode : Imposed;
      porosity_mode : KOnly;
      sampling_period : 5;
      sor_factor : 1.4;
      conv_factor : 0.001;
      overfill_mode : Gradient;
      overfill_factor : 1.001;
      Stop_Criteria {
            max_total_steps : 1000;
            nb_prefill_steps : 0;
            nb_postfill_steps : 40;
            prefilling_time : 0;
            postfilling_time : 6000;
            max_conversion : 1;
            max_machine_time : 7200;
            max_sim_time : 7200;
      }
      compensate_error : 1;
      save_velocity_field : 0;
      fixed_dt_max : 3600;
}
```

```
Resin {
      family : "Resin";
      density : 1250;
      cp : constant;
      cp_0_const : 2000;
      cp_0_lin : 1;
      cp_0_quad : 3;
      cp_1_lin : 2;
      cp_1_quad : 4;
      cp_2_quad : 5;
      thermal_conductivity : 2.325;
      name : "Dexter/Hysol EA9150";
      VISCOSITY {
            FAMILY : Viscosity;
            TYPE : Expo_5;
            K1 : 7.54E-10 ;
            K2 : 6517 ;
            K3 : 0.7152 ;
            K4 : 2.611 ;
            K5 : -1.057 ;
      }
      KINETIC {
            FAMILY : Kinetic;
            TYPE : f(temperature, alpha);
            Hr : 237.682;
            Surface {
                  FAMILY : "Surface";
                  TYPE : "Krig_E";
                  CONTROL_POINTS {
                        FAMILY : "PointSet";
                        TYPE : "PointSet1D";
                        POINTS {
                              DIMENSION : 3;
                              NUMBER : 30;
                              DATA :
383, 0, 0.0004, 383, 0.163, 0.000616, 383, 0.367, 0.000705, 383,
0.592, 0.000593, 383, 0.844, 0.000322, 383, 0.955, 0.000134,
383, 0.993,
4.03e-05, 383, 1, 7.71e-06, 393, 0, 0.000818, 393, 0.124,
0.00102,
393, 0.212, 0.00115, 393, 0.297, 0.00118, 393, 0.439, 0.00111,
393,
0.604, 0.000923, 393, 0.883, 0.000311, 393, 0.93, 0.000189, 393,
1,
2.4e-06, 403, 0, 0.00223, 403, 0.0337, 0.00235, 403, 0.177,
0.00244,
403, 0.402, 0.00216, 403, 0.733, 0.00116, 403, 0.908, 0.000372,
403,
0.988, 7.39e-05, 320, 0, 0, 320, 0.2, 0, 320, 0.4,
```

```
0, 320, 0.6, 0, 320, 0.8, 0, 320, 1, 0;
                        }
                    }
                }
        }
}
```

```
Physical_Parameters {
      Zone_List {
            {
                  family : "Zone";
                  material_name : "TPI4398";
                  zone_id : 1;
                  zone_name : "cavity";
                  zone_porosity : 0.4;
                  nb_sublayers : 1;
            }
      }
      Material_List {
            {
                  family : "Material";
                  material_name : "TPI4398";
                  type : Fabric;
                  density : 1800;
                  ml_cp_f : constant;
                  ml_cp_f0_const : 1088.5;
                  ml_cp_f0_lin : 1;
                  ml_cp_f0_quad : 3;
                  ml_cp_f1_lin : 2;
                  ml_cp_f1_quad : 4;
                  ml_cp_f2_quad : 5;
                  ml_lambda_f : constant;
                  ml_lambda_f0_const : 7.788;
                  ml_lambda_f0_lin : 0.2;
                  ml_lambda_f1_lin : 0.3;
                  permeability_K1 : 2.25e-11;
                  permeability_K2 : 1.4e-11;
                  permeability_K3 : 1e-20;
                  compressibility_curve : 1;
                  nb_layers : 8;
                  surface_density : 286;
                  tensor_conductivity_K1 : 0.3;
                  tensor_conductivity_K2 : 0.3;
                  tensor_conductivity_K3 : 0.3;
                  thermal_dispersion_K1 : 0;
                  thermal_dispersion_K2 : 0;
```

```
                tube_type : Dirichlet;
                tube_temp : 0;
                tube_coef : 0;
            }
        }
}
```

```
Heat_analysis_parameters {
      solver : ATMC;
      schema : Taylor-Galerkin;
      theta1 : 0.5;
      theta2 : 0.5;
      resol_sdp : conjugate_gradient;
      norm_sdp : symmetric;
      precond_sdp : ssor;
      param_sdp : 1.2;
      stoptol_sdp : 1e-06;
      maxit_sdp : 300;
      resol_def : bigcstab;
      norm_def : nearly_symmetric;
      precond_def : none;
      param_def : 1.2;
      stoptol_def : 1e-06;
      maxit_def : 300;
      fic_init : 300;
      fic_sol_init : "";
      fic_sol_fin : "";
      fic_zone_coef : true;
      fic_source : 0;
      fic_vecteur :
0, 0, 0;
      fic_tenseur :
0, 0, 0, 0, 0, 0, 0, 0, 0;
      ml_temp_moule_init : 300;
      ml_temp_fibre_init : 338;
      ml_sol_fin : "";
      ml_temp_resine_inj : 322;
      ml_sol_init : "";
      ml_type_cfr_inj : tucker;
      cond_fr : "temp_bound.ifr";
      raccord : "";
      fich_dump : "these";
      periode_dump : 1;
      pre_temp_curve {
            FAMILY : "Curve";
            TYPE : "Constant";
```

```
            VALUE : 1;
      }
      pre_flux_curve {
            FAMILY : "Curve";
            TYPE : "Constant";
            VALUE : 1;
      }
      pre_coef_curve {
            FAMILY : "Curve";
            TYPE : "Constant";
            VALUE : 1;
      }
      fil_temp_curve {
            FAMILY : "Curve";
            TYPE : "Constant";
            VALUE : 1.09;
      }
      fil_flux_curve {
            FAMILY : "Curve";
            TYPE : "Constant";
            VALUE : 1;
      }
      fil_coef_curve {
            FAMILY : "Curve";
            TYPE : "Constant";
            VALUE : 1;
      }
      post_temp_curve {
            FAMILY : "Curve";
            TYPE : "Piecewise_Linear";
            Control_Points {
                  FAMILY : "PointSet";
                  TYPE : "PointSet1D_G";
                  POINTS {
                        DIMENSION : 2;
                        NUMBER : 4;
                        DATA :
0, 1.09, 1500, 1.27, 5100, 1.27, 6000, 1;
                  }
            }
      }
      post_flux_curve {
            FAMILY : "Curve";
            TYPE : "Constant";
            VALUE : 1;
      }
      post_coef_curve {
            FAMILY : "Curve";
            TYPE : "Constant";
```

```
            VALUE : 1;
      }
}
```

```
injection_list {
      InjGate {
            Boundary_Condition {
                  family : "Boundary_Condition";
                  bc_label : "No description";
                  bc_type : 1;
                  bc_id : 1;
                  bc_value_in_time {
                        FAMILY : "Curve";
                        TYPE : "Constant";
                        VALUE : 6e-07;
                  }
            }
            family : "InjGate";
            inj_active : 1;
            inj_length : 0;
            inj_lin_drop_coeff : 0;
            inj_temp_history : 0;
            inj_pressure_max : 1e+10;
      }
}
```

Annexe B

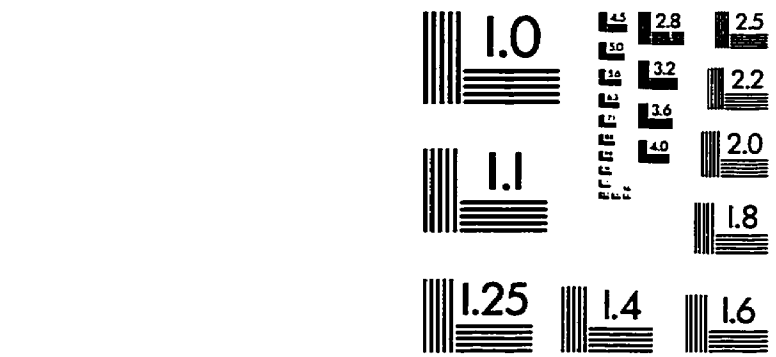
Exemple d'un fichier de simulation avant Dataflot

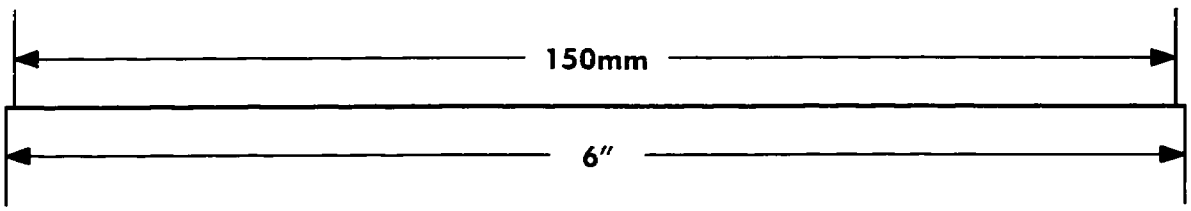
Cette annexe montre le format de fichier qui était utilisé avant Dataflot. Nous insistons sur la nature non-structurée de ce fichier.

```
!PARAMETRES_FLOT_V1.0

! Parametres generaux
fichier_sortie:"simulation1.flo"
injection_log:"simulation1.prn"
fichier_geometrie:"simulation1.geo"
tester_modele:false

max_etapes:1000
max_exe_time:240
max_inj_time:7200
alpha_max:1.01
frequence:10
binflo:true
modele_ecoulement:avec_darcy
modele_thermique:c_3
modele_cinetique:complet
type_debordement:gradient_alpha
tolerance_debordement:1.0
id_zones:1
porosites_de_zones:0.5
permeabilites_k1:1.0e-9
permeabilites_k2:1.0e-9
permeabilites_k3:1.0e-9
cp_zones:1.0
kth_zones:1.0
rho_zones:1.0
type_injection:debit_constant
id_injection:1
valeurs_injectees:100.0e-6

!Parametres addditionnels


! Parametres avec Darcy
viscosite:1.0
densite:1.0
```

```
type_resine:parametree2
resine_a:1.0
resine_b:-1.0
resine_k:0.0
resine_krig:""
tol:1e-3
sor:1.5

! Parametres thermiques
kth_resine:1.0
cp_resine:1.0
t_front:300
t_injection:325
etapes_chaleur:25
schema:0.66

! Parametres cinetiques
hr_ks:200
c_ks:1.0,1.0,1.0
a_ks:20000,20000,20000
e_ks:7000,7000,7000
m_ks:0.5,0.5,0.5
p_ks:0.75,0.75,0.75
```

IMAGE EVALUATION
TEST TARGET (QA-3)





APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989