SYSTÈME D'ÉVALUATION ET DE GESTION DES RISQUES D'INONDATION EN MILIEU FLUVIAL

PROJET SEGRI

Rapport de recherche 2004

Rapport de recherche No R-720-b *Janvier 2005*

Système d'**É**valuation et de **G**estion des **R**isques d'**I**nondation en milieu fluvial

Projet SEGRI

Rapports de recherche 2004

Présenté au

Fonds des Priorités Gouvernementales en Science et en Technologie – volet Environnement (FPGST-E)

19 janvier 2005

Équipe de réalisation

Institut National de la Recherche Scientifique – Eau, Terre et Environnement

Yves Secretan	Professeur, PhD
Eric Larouche	Ingénieur en informatique

Collaborateurs

Maude Giasson	Stagiaire, étudiante à la maîtrise
Cédric Caron	Stagiaire
Benjamin Behaghel	Stagiaire
Daniel Nadeau	Stagiaire
Stéphane Lévesque	Stagiaire
Sébastien Labbé	Stagiaire

Pour les fins de citation : **Secretan Y., Larouche E. & coll. (2004)**.

Système d'Évaluation et de Gestion des Risques d'Inondation en milieu fluvial (SEGRI) : Rapports de recherche 2004. Québec, INRS-Eau, Terre & Environnement. Pagination multiple. (INRS-Eau, Terre & Environnement, rapport de recherche 720 b)

Pour: Fonds des Priorités Gouvernementales en Science et en Technologie – volet Environnement (FPGST-E).

ISBN: 2-89146-524-5

Liste des rapports

RAPPORT #1 : Module Éditeur

RAPPORT #2 : Géométries et SRChampVector

RAPPORT #3 : Modèle Numérique de Terrain (MNT)

RAPPORT #4 : Module de validation

RAPPORT #5 : Module de filtres

RAPPORT #6 : Module de partition de maillage

RAPPORT #7 : Module projet

RAPPORT #8 : Algorithme de maillage

RAPPORT #9 : Validation du modèle de température sous H2D2

RAPPORT # 1 : Module Éditeur

Rapport de développement logiciel

Rapport sur l'éditeur

Présenté par :

Benjamin Behaghel

18 janvier 2005

Versions du document

Date	Auteur	Commentaires	Version
26/03/2004	Benjamin Behaghel	Version initiale	V 1.0
15/04/2004	Benjamin Behaghel	Modifications structurelles	V 1.1
19/04/2004	Benjamin Behaghel	Corrections	V 1.2
22/04/2004	Benjamin Behaghel	§5.2.2.3, 6.2 et 8.1 (diagramme)	V 1.3

SOMMAIRE

1. Introduction

1.1 Objectifs

Le but de ce rapport est de présenter les différentes composantes de l'éditeur de Modeleur 2.0 (Module_Editeur) et de résumer les différentes décisions prises lors du travail d'analyse, de design et d'implantation. Ce rapport déterminera aussi les futures étapes de développement de l'éditeur ou liées à l'éditeur.

Ce rapport ne prétend pas suivre chronologiquement toutes les étapes de la réflexion sur l'éditeur mais relate les décisions qui ont finalement été prises et leurs raisons. Il explique aussi les principes de fonctionnement du module *Module_Editeur*.

1.2 Contexte

Modeleur est un système d'information géographique (SIG) dédié à l'hydraulique fluviale. Plusieurs activités de Modeleur ont besoin de pouvoir éditer des objets géométriques (notamment les activités MNT et validation de données). C'est pourquoi un éditeur permettant la manipulation de ce type de données est nécessaire. Modeleur 1.0 offrait cette possibilité, Modeleur 2.0 doit donc au moins proposer les mêmes fonctionnalités et améliorer celles qui ont lieu de l'être.

1.2.1 Références

Ce rapport fait suite au travail d'analyse et de design réalisé autour de l'éditeur. Ces travaux ont fait l'objet des rapports suivants:

- Résumé de la réunion du 25-02-2004 - Mise en commun MNT et semis de points & Diagrammes de séquence. 18 pages.
- Résumé de la réunion du 02-03-2004 - Sur l'éditeur. 12 pages.
- Résumé de la réunion du 09-03-2004 – Sur l'éditeur #2. 6 pages

Ces rapports s'appuient sur les spécifications fonctionnelles de Modeleur 2 :

- Spécifications – Modeleur 2.0. Québec, INRS-Eau, Terre & Environnement. 68 pages.

Ces travaux s'appuient aussi sur les SRChampsVector :

- Rapports sur SRChampsVector : « *//Caxipi/green/Rapports/Algorithmes, SRChampVector, Isolignes et Isosurfaces/* »

1.3 Structure du document

Plusieurs concepts ont été abordés au cours de l'étude et de la réalisation de l'éditeur. Chacun de ces concepts est repris et expliqué dans la suite de ce rapport.

Ce rapport fait une présentation générale du Module Editeur avec explication du rôle de chacune des composantes de ce module et sur la nature des objets manipulés dans le cadre de l'édition.

Ce rapport indique les points importants de conception du module :

- indépendance de l'éditeur vis-à-vis des structures à éditer ;
- généricité des structures à éditer ;
- communication entre les structures à éditer et l'éditeur.

Ce rapport donne des explications sur la manière dont fonctionnent les différentes classes du Module Editeur et la façon de les utiliser.

Ce rapport donne quelques explications sur VTK et son utilisation dans l'éditeur.

En annexes, ce rapport présente quelques idées ou concepts qui n'ont finalement pas été retenus.

2 Notions de bases pour l'éditeur

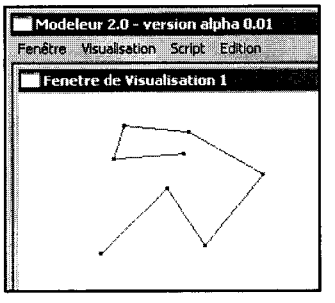
2.1 Géométries

2.1.1 Point

Un point est une géométrie de dimension 0. Un point est représenté par ses 3 coordonnées x, y et z. Dans le présent rapport, le terme « point libre » sera utilisé pour désigner un point qui n'appartient ni à une polyligne ni à un polygone.

2.1.2 Polyligne

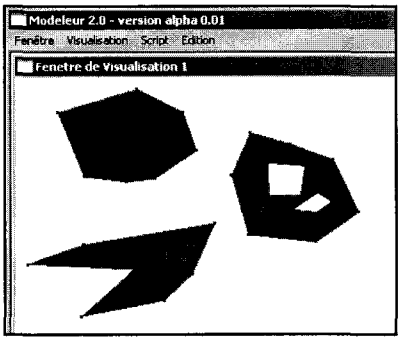
Une polyligne est un ensemble ordonné de plus de 2 points reliés 2 à 2 par un segment.



Exemple de polyligne dans Modeleur 2.0

2.1.3 Polygone

Un polygone est une surface plane avec 0 à n trous. L'intérieur d'un polygone forme un ensemble connexe de points. Un trou est représenté par un polygone dans le polygone.



Exemple de polygones dans Modeleur 2.0

2.1.4 Multi

Les multis font référence aux multipolygones, multipolyligne et multipoints. Ils servent à désigner plusieurs entités. Par exemple, un multipolygone désigne 1 à n polygones.

3 Le module éditeur

Au cours du développement du logiciel Modeleur 2.0, il s'est fait ressentir le besoin d'un éditeur pour certains modules, notamment les modules MNT et Semi de points. Dans la suite du projet, d'autres modules pourront aussi avoir besoin d'un éditeur.

Lors de la mise en commun du travail effectué de part et d'autre sur les semis de points et le MNT, il s'est avéré que les besoins vis-à-vis de l'éditeur sont les mêmes. Le choix a alors été fait de réaliser un éditeur générique au détriment d'un éditeur spécialisé pour chaque module.

3.1 Fonctionnement du Module Editeur

Le module éditeur (annexe 8.1) est composé de 3 parties :

- un gestionnaire d'événements (*EDGestionnaire*) : il permet d'analyser les événements afin d'indiquer à l'éditeur les actions à effectuer pour y répondre ;
- un éditeur (*EDEditeur*) : il maintient à jour toute la partie graphique et permet d'interagir avec ce qui est édité ;
- une interface liée à l'édité (*EDInterfaceContainer*) : elle permet à l'éditeur d'agir sur ce qui est édité. C'est cette interface qui indique à l'éditeur si une action a réussi ou non afin que ce dernier puisse maintenir la cohérence entre ce qui est affiché et l'objet édité.

Le module éditeur est un outil dont vont se servir d'autres modules. Il fonctionne de la façon suivante : lorsqu'un module a un besoin en édition, il demande au gestionnaire d'événements de l'éditeur de créer un nouvel éditeur auquel il spécifie un EDInterfaceContainer (contenant la structure à éditer) et un espaceVTK indiquant où réaliser l'affichage graphique. L'éditeur affiche alors la structure à éditer et permet une interaction avec celle-ci par le biais d'outils graphiques.

3.2 Le rôle d'EDGestionnaire

Le gestionnaire d'éditeur (*EDGestionnaire*) est la classe qui gère les événements d'édition. Le gestionnaire d'éditeur permet d'avoir plusieurs éditeur fonctionnant simultanément. Le gestionnaire récupère les événements arrivant sur le bus et appelle les fonctions correspondant à ces actions de l'éditeur (*EDEditeur*) pour l'éditeur concerné.

Le gestionnaire d'éditeur gère aussi les différents modes dans lesquels peut se trouver l'éditeur (sélection, ajout, déplacement, ...). Certains modes de l'éditeur peuvent être implicites. Par exemple en mode sélection aussi longtemps que le bouton gauche est enfoncé, on est en mode « implicite » de déplacement.

Les modes de l'éditeur peuvent aussi être explicites, c'est-à-dire imposés par l'utilisateur. Par exemple le mode ajout de polygone qui permet de créer de nouveaux polygones doit être activé par l'utilisateur au moyen d'un événement (cet événement pourra être lancé à partir d'un menu par exemple).

Le gestionnaire d'éditeur gère 2 types d'événements :

- les événements de souris qui concernent toutes les actions de l'utilisateur sur la fenêtre d'édition ;
- les événements de commande qui sont utilisés pour définir le mode d'édition (ajout, sélection, insertion) ou effectuer des actions directes telles que supprimer les objets sélectionnés.

3.3 Le rôle d'EDEditeur

Le rôle d'*EDEditeur* est de réaliser l'affichage de l'édité et de permettre des interactions graphiques avec ce dernier. Il connaît aussi un ensemble de fonctions permettant de notifier les modifications à l'édité. Les fonctionnalités définies dans les spécifications et auxquelles il doit répondre sont les suivantes :

- afficher ;
- créer des objets (points, polylignes ou polygones) ;
- sélectionner par sélection directe ou par définition d'une zone de sélection ;
- détruire un objet complet ou une partie d'un objet ;
- déplacer un objet complet ou une partie d'un objet ;
- unir des polygones ;
- insérer des points dans des polylignes ;
- insérer des polylignes dans des polygones.

Toutes ces fonctionnalités sont connues du gestionnaire d'événements de l'éditeur qui sait lesquelles appeler en fonction des besoins qu'on lui a exprimés.

3.4 Le rôle d'EDInterfaceContainer

Le but d'EDInterfaceContainer est de proposer une interface accessible à l'éditeur et permettant de notifier à l'édité les modifications graphiques effectuées. Le but de cette interface est de proposer l'ensemble minimal des fonctionnalités nécessaires au fonctionnement de l'éditeur. Ces fonctionnalités sont les suivantes :

- ajout : permet d'ajouter un point, une polyligne ou un polygone à la structure éditée ;
- déplacement : permet de déplacer des objets de l'édité ;
- insertion : permet de compléter des objets de l'édité ;

- suppression : permet de supprimer des objets de l'édité ;
- itération : permet de parcourir les objets de l'édité.

4 Détails sur l'éditeur

Cette partie présente l'éditeur tel qu'il est implanté au final. Pour connaître plus en détails les raisons de chaque décision, se reporter aux différents rapports sur l'éditeur.

4.1 Type de l'édité

L'éditeur utilisé dans le Module Editeur est un éditeur de type géométrique, c'est à dire qu'il ne connaît que les informations nécessaires à son fonctionnement et à l'affichage. Cela le sépare donc de ce qu'il édite puisque ce qui est édité peut contenir d'autres informations. Par exemple on peut souhaiter éditer des objets de type SRChampVector, ces objets sont intéressants car ils permettent d'associer à chaque point de l'information (température, pression, type de substrat ...).

Le Module Editeur permet donc d'éditer des objets de type géométrique ou des objets de type SRChampVector, tout en offrant un éditeur qui reste indépendant de ce qu'il édite puisque n'étant que géométrique. Pour qu'un type d'objet soit éditable, il faut qu'il implémente une interface (*EDInterfaceContainer*). Cette interface propose des fonctions qui permettent à l'éditeur d'indiquer à l'édité les modifications à apporter. Ces modifications doivent nécessairement être accomplies par l'édité car il est le seul à pourvoir garantir sa propre intégrité.

4.1.1 Responsabilités éditeur - édité

Il est nécessaire pour le Module Editeur de posséder des fonctionnalités permettant de modifier les objets édités. Les fonctionnalités minimales nécessaires que la structure à éditer doit fournir ont été regroupées dans *EDInterfaceContainer,* une classe virtuelle dont doivent hériter ceux qui désirent se faire éditer (voir section 5.2). C'est la classe *EDEditeur* qui utilisera les fonctionnalités de *EDInterfaceContainer* pour mettre à jour l'édité en fonctions des actions effectuées dans l'éditeur.

Pour construire cette classe, il a été nécessaire de bien faire la part entre ce qui est de la responsabilité de l'éditeur et ce qui est sous la responsabilité de l'édité. En définitive, l'éditeur ne doit avoir affaire qu'aux fonctions d'affichage. Il ne doit pas connaître la composition de ce qui est édité. Il doit juste savoir afficher ce qu'on lui demande d'afficher et déplacer, supprimer, modifier graphiquement tout ce qui est affiché.

Au cours de ces opérations, l'éditeur doit avertir l'édité des modifications apportées. Celui-ci lui dira si ces opérations sont autorisées. L'édité fait alors les modifications qui s'imposent si l'opération est autorisée et l'éditeur de son côté doit s'assurer du maintien de sa cohérence avec l'édité.

4.2 Communication avec l'éditeur

Le module éditeur est un module indépendant et connecté sur le bus. Il est donc générique puisque n'importe qui peut utiliser ses fonctionnalités en envoyant les bons événements. Il communique avec l'édité par le biais d'appels de fonctions.

4.3 Communication entre les structures à éditer et l'éditeur

Il est nécessaire pour l'éditeur de pouvoir « lire » les objets à éditer (conversion objet géométrique vers objet VTK) et de pouvoir « écrire » de nouveaux objets (conversion d'un objet VTK vers un objet géométrique). Sur ce point là, la réflexion a été poussée pour rendre l'éditeur le plus indépendant possible de la structure à éditer.

4.3.1 Ecriture

Pour « l'écriture » l'idée la plus simple a été mise en œuvre, c'est-à-dire, qu'on décrit les objets géométriques sous forme de coordonnées 3D (GOCoordonneesXYZ). Dans le cas d'un point, on fournit juste une coordonnée, dans le cas d'une polyligne on fournit un itérateur sur un conteneur de points. Enfin, dans le cas d'un polygone, il faut fournir une liste d'itérateurs représentant les différentes polylignes du polygone.

4.3.2 Lecture & callback

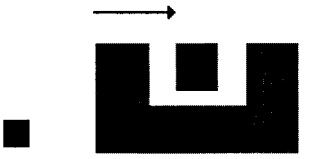
Pour la « lecture » on utilise[1] le principe des callback. Dans un système avec callback, l'éditeur demande à itérer sur un objet géométrique en précisant quelle fonction appeler pour chaque niveau de cet objet. Par exemple, dans le cas d'une polyligne, l'éditeur demande une itération sur une polyligne en passant une fonction (de l'éditeur) à appeler pour chaque point de cette polyligne. Cette technique de callback est particulièrement intéressante puisque *EDEditeur* n'a pas à connaître la structure des objets sur lesquels il itère et *EDInterfaceContainer* n'a pas besoin de connaître VTK ni la manière dont sont construits les acteurs.

4.4 Unité des opérations

Il s'est avéré au cours de la réflexion sur l'éditeur que les différentes opérations de base n'étaient pas divisibles en sous opération, c'est-à-dire par exemple que le déplacement de 2 polygones ne peut pas être vu comme le déplacement d'un polygone suivi du déplacement d'un second polygone. Cela s'explique par le fait que certaines opérations sont invalides et donc doivent être interdites par le conteneur édité. Prenons par exemple un cas ou 2 polygones n'auraient pas le droit de se chevaucher, nous pouvons alors obtenir la situation suivante :

[1] Une autre solution basée sur un héritage d'itérateurs avait été proposée. Cette solution qui n'a pas été retenue est présentée en annexe (8.3).

Supposons que l'on désire déplacer les 2 polygones suivant le vecteur de déplacement représenté par la flèche noire



Si l'on déplace les 2 polygones en même temps, le déplacement est valide, c'est-à-dire qu'à l'arrivée, aucun polygone n'en chevauche un autre.

Si l'on déplace le petit en premier, le déplacement est invalide puisque le petit polygone chevauche le gros.

Si l'on déplace le gros polygone en premier, le déplacement est invalide car le gros polygone chevauche le petit. Cet exemple montre par la même occasion qu'on ne peut pas espérer trouver à coup sûr un ordre dans lequel le déplacement des polygones un à un est correct.

On retrouve le même problème d'unité des opérations lorsqu'on essaye de déplacer des parties d'objets (par exemple plusieurs points d'un polygone) ou que l'on essaye de supprimer plusieurs parties d'un objet. Il est en effet impossible de supprimer un point à une polyligne n'en comportant que 2 puisqu'une polyligne n'est affichable que si elle comporte au moins 2 points. Il est par contre possible de lui supprimer les 2 points, cela aura pour effet de supprimer la polyligne.

Pour l'insertion le problème ne se pose pas puisqu'il a été décidé de n'autoriser l'insertion que d'un seul élément à la fois (une polyligne dans un polygone ou un point dans une polyligne).

5 Fonctionnement et utilisation

5.1 EDEditeur

Cette partie a pour vocation d'expliquer le fonctionnement et le comportement de la classe *EDEditeur*. Y sont décrites les structures importantes qui sont utilisées par les classes de ce module et les fonctions pour un utilisateur externe (API).

5.1.1 Comportement des fonctions d'*EDEditeur*

EDEditeur, la classe centrale de l'éditeur, comprend de nombreuses fonctions qui seront appelées par *EDGestionnaire*[2] en fonction des événements que ce dernier aura reçus. Les fonctions de l'éditeur ont été définies en recherchant les actions minimales auxquelles l'éditeur doit savoir répondre. Ces fonctions sont les suivantes :

- recharge : détruit et recrée tous les objets VTK à partir d'un EDInterface-Container passé en paramètre ;
- asgType : permet de signifier sur quel **type** d'objets on travaille : points, polylignes, polygones.
- ajoutePoint : ajoute un point. Cela permet de construire un point, une polyligne ou un polygone suivant le **type** en cours ;
- selectionne : permet de sélectionner un objet ou une partie d'objet VTK ;
- dessineCadre : permet de dessiner un cadre définissant une **zone** de sélection ;
- selectionneZone : permet de demander la sélection de tous les objets ou parties d'objets se trouvant dans la **zone** de sélection ;
- deplaceSelectionTemp : déplace graphiquement seulement les objets sélectionnés suivant un vecteur de déplacement ;
- deplaceSelection : déplace les objets sélectionnés suivant un vecteur de déplacement. deplaceSelection notifie le déplacement au conteneur qui l'effectuera si celui est valide. S'il est effectivement valide, on déplacera aussi graphiquement les objets correspondant de l'éditeur ;
- supprimeSelection : demande la suppression des éléments sélectionnés ;
- insereTrou : construit un trou de polygone ;
- scinde : insère un point entre 2 sommets d'un segment ;
- arreteActionEnCours : permet d'annuler une action qu'on est en train d'effectuer mais qu'on ne désire pas terminer ;

[2] *EDEditeur* ne répond pas directement aux événements car on veut pouvoir avoir plusieurs éditeurs qui fonctionnent simultanément. Il faut donc un gestionnaire chargé de repartir entre les éditeurs les différentes actions à effectuer en réponse à un événement. C'est le rôle de *EDGestionnaire*.

- deselectionneTout : permet de désélectionner tout ce qui est sélectionné, par exemple quand on clique dans le vide ou quand on finit un déplacement ;
- unis : permet d'unir 2 polygones.

L'*EDEditeur* se doit de maintenir la cohérence entre lui-même et l'*EDInterfaceContainer* tout en affichant les différents objets de ce dernier. Il est nécessaire de faire la distinction entre 3 types d'actions de l'éditeur :

- les actions temporaires de l'éditeur, i.e. qui n'influent pas sur *EDInterface-Container* ;
- les actions parallèles, i.e. qui sont faites dans *EDEditeur* et dans *EDInterface-Container ;*
- les actions qui ne sont effectuées que du côté d'*EDInterfaceContainer.*

5.1.1.1 Les actions temporaires

Ce sont toutes les actions qui ne sont pas terminées et dont le déroulement n'influe par sur l'*EDInterfaceContainer.*

On retrouve dans les actions temporaires les actions suivantes :

- la création d'un polygone ou d'une polyligne. En effet, ce n'est que lorsqu'on pose le dernier point d'une polyligne ou d'un polygone que celui-ci est ajouté au conteneur. Tant que la création n'est pas terminée, les modifications n'apparaissent que dans l'éditeur ;
- l'insertion d'une polyligne dans un polygone pour les mêmes raisons ;
- la sélection qui modifie l'aspect graphique uniquement dans *EDEditeur.*

Le déplacement est aussi considéré comme une action temporaire puisque le choix[3] a été fait de faire du déplacement statique dans le cadre de l'éditeur.

Le déplacement statique consiste à appliquer les modifications effectives à la fin du déplacement seulement. Cette méthode permet de ne vérifier le respect des contraintes vis à vis de tous les objets présents qu'à la fin du déplacement et permet ainsi un gain de performance. Alors que le déplacement dynamique empêchait l'utilisateur d'effectuer des déplacements invalides, le déplacement statique, de son côté, laisse l'utilisateur libre d'effectuer des déplacements invalides, mais annule le déplacement une fois celui-ci effectué s'il est effectivement non valide.

5.1.1.2 Les actions parallèles

Quand une action se termine, il faut si cela est nécessaire appeler les fonctions correspondantes de *EDInterfaceContainer* et mettre à jour l'affichage. La première solution pour le faire est d'effectuer la modification simultanément dans *EDInterfaceContainer* et dans *EDEditeur.*

[3] *voir rapport de réunion du 02/03/04 sur l'éditeur #1*

Dans ce cas, on ne détruit pas l'objet VTK, on ne fait que le modifier. Dans les cas de déplacement par exemple, il est très facile de déplacer l'objet dans l'éditeur tout en restant cohérent avec l'*EDInterfaceContainer*. De même dans le cas de l'ajout, on dispose déjà de l'acteur correspondant à ce qu'on ajoute puisqu'on l'a crée progressivement. Dans ces 2 cas là il est inutile de détruire l'objet VTK pour le reconstruire. Par contre, il faut faire très attention à ce que les mêmes modifications soient apportées dans *EDEditeur* et *EDInterfaceContainer*.

5.1.1.3 Les actions effectués que dans EDInterfaceContainer

Une autre solution quand une action est terminée est de détruire dans *EDEditeur* l'objet VTK qui a subi une modification (déplacement, suppression, insertion) puis de le redessiner complètement. On peut même pousser le système à l'extrême et à chaque action terminée, tout détruire et tout réafficher (ce qui est fait par la méthode recharge). C'est la solution la plus générale, mais elle peut être parfois un peu lourde inutilement.

Il est possible d'utiliser indifféremment l'une ou l'autre des 2 méthodes présentées en 5.1.1.2 et 5.1.1.3 dans chacun des cas. Cela doit être transparent et n'avoir aucun d'impact sur *EDInterfaceEditeur*. Cependant il est plus intéressant de jongler avec les 2 principes et d'utiliser le plus simple au cas par cas. A ce jour, seuls les déplacements et les ajouts sont gérés en parallèle à *EDInterfaceContainer*.

5.1.2 Structures de l'éditeur

L'éditeur a besoin de maintenir un lien entre les objets graphiques (VTK) qu'il crée et les objets du conteneur que les objets VTK représentent. On utilise donc une structure associative (par exemple un *std::map*) pour pouvoir retrouver un acteur à partir d'une référence d'un objet du conteneur et une autre pour pouvoir faire l'opération inverse. Le n-ième point d'un acteur représente le n-ième point de l'objet du conteneur associé a cet acteur, de même pour le n-ième polyligne et le n-ième polygone.

L'éditeur utilise aussi une structure qui n'a aucune influence ou incidence sur l'extérieur et qui lui permet de connaître les objets qui sont sélectionnés ainsi que leurs couleurs originales.

```
struct StructSelection
{
  std::vector<CellId>      selections; // Parties d'acteur sélectionnées
  std::vector<Couleur>     couleursOriginalesSelections; // Couleurs des parties sélectionnées
};
std::map<vtkActor*, StructSelection>  m_selections; /* Structure contenant tout ce qui est
sélectionné dans l'éditeur */
```

L'éditeur dispose aussi d'une classe qui lui sert à conserver des informations permettant de construire un acteur (8.2) lors des appels de fonctions callback (4.3.2) de

EDInterfaceContainer. Les éléments de cette classe ne sont modifiables que par EDEditeur.

```
Entier          cptPoints;          // Sert à compter un nombre de points à mettre dans l'acteur
Entier          cptPolylignes;      // Sert à compter un nombre de lignes à mettre dans l'acteur
vtkPoints*      pointsP;            // Les points de l'acteur
vtkFloatArray*  tabP;               // Les couleurs des cell de l'acteur
vtkCellArray*   vertsP;             // Les points libres
vtkCellArray*   linesP;             // Les polylignes de l'acteur
vtkCellArray*   polysP;             // Les polygones de l'acteur
```

5.2 Utilisation de EDInterfaceContainer

L'interface *EDInterfaceContainer* doit être implémentée par toute classe désirant être éditée. Il est important de bien comprendre le fonctionnement de cette interface (*EDInterfaceContainer*) et les différentes structures qui y sont utilisées avant de se lancer dans sa réalisation.

5.2.1 Les structures utilisées dans *EDInterfaceContainer*

Pour les communications entre *EDEditeur* et *EDInterfaceContainer* plusieurs structures différentes sont utilisées ou ont été mises en place :

- L'unité choisie pour représenter un point est le GOCoordonneesXYZ. On est aussi souvent amené à traiter des listes de points (*std::vector<GOCoordonneesXYZ>*) et donc des *std::vector<GOCoordonneesXYZ>::const_iterator*.
- Une enum est utilisée pour savoir quand on a affaire à un polygone, une polyligne ou un point.
- Une structure (StructGeomPrimitif) est utilisée pour pouvoir décrire les différents objets à manipuler ou les parties de ces objets. C'est la structure servant à désigner dans un langage commun à *EDEditeur* et à *EDInterfaceContainer* les objets manipulés par l'éditeur.

```
struct StructGeomPrimitif
  {
    Type    type;
    void*   reference ;
    Entier  indice1;
    Entier  indice2;
  };
```

La signification de cette structure est récapitulée dans le tableau suivant (et disponible dans *EDInterfaceContainer.h*) :

Objet à	type	reference	indice1	indice2

représenter				
Polygone entier	EDPOLYGONE	pointeur au polygone	-1	-1
i-ème polyligne d'un polygone	EDPOLYGONE	pointeur au polygone	i	-1
i-ème sommet de la j-ème polyligne d'un polygone	EDPOLYGONE	pointeur au polygone	j	i
Polyligne entière	EDPOLYLIGNE	pointeur à la polyligne	-1	-1
i-ème sommet d'une polyligne	EDPOLYLIGNE	pointeur à la polyligne	i	-1
i-ème point	EDPOINT	pointeur au multipoint	i	-1

5.2.2 Fonctions d'EDInterfaceContainer

Le but recherché pour *EDInterfaceContainer* est de proposer l'interface minimale pour faire fonctionner l'éditeur. *EDInterfaceContainer* est donc défini de telle manière qu'il n'y apparaisse que les fonctions essentielles au bon fonctionnement de l'éditeur. Ces fonctionnalités[4] sont présentées dans les sous paragraphes suivants.

5.2.2.1 Lecture

Il y a les fonctions dites de « lecture » qui permettent de créer les acteurs correspondant aux objets du conteneur. Ces fonctions fonctionnent sur le principe des callback. L'éditeur demande à afficher tous les polygones en appelant une fonction d'*EDInterfaceContainer* et en lui passant la fonction pour créer graphiquement un polygone. Pour chaque polygone, *EDInterfaceContainer* appelle cette fonction en lui passant la référence du polygone à dessiner. Cette fonction appellera le niveau suivant (polyligne) et le procédé se répète jusqu'à atteindre le point qui est l'unité de base.

5.2.2.2 Ajoute

Il y a les fonctions permettant d'ajouter des objets à *EDInterfaceEditeur*. Elles sont au nombre de 3, une pour ajouter un point, une pour ajouter une polyligne et la dernière pour ajouter un polygone. Les 3 fonctionnent sur le même principe. Elles prennent un itérateur sur un ensemble de points ordonnés et renvoient une référence (void*) sur l'objet[5] crée dans l'*EDInterfaceContainer*.

5.2.2.3 Déplace

Il y a une fonction de déplacement. Elle est unique car un ensemble de déplacement ne peut pas être décomposé par l'éditeur en sous déplacements indépendants les uns des autres (§4.4). Cette fonction prend donc un vecteur de déplacement ainsi qu'un itérateur

[4] Une autre solution avec moins de fonctionnalités avait été imaginée, mais elle ne fonctionnait pas pour le cas présent. Voir §8.3.2

[5] On peut renvoyer autre chose que la référence à l'objet créer puisqu'on demande un void*, mais on doit être capable par la suite d'identifier grâce à cette « référence » l'objet alors créé.

sur un ensemble de StructGeomPrimitif (§5.2.1) qui représente l'ensemble des éléments à déplacer.

5.2.2.4 Supprime

Il y a une fonction de suppression. Elle est unique pour les mêmes raisons que celle de déplacement, à savoir qu'une partie de suppression peut être invalide alors que l'ensemble de la suppression est valide. Ici l'enjeu est un peu plus important que pour le déplacement puisqu'une fois un objet supprimé, il est impossible de le recréer avec la même référence et les mêmes valeurs portées. Elle prend en paramètre un itérateur sur une liste de d'entités à supprimer décrites chacune sous la forme d'un StructGeomPrimitif (§5.2.1).

5.2.2.5 Unis

La fonction d'union prend en entrée 2 polygones et essaye de les unir. Cette fonction ressort les polygones créés dans le vecteur passé en entrée.

5.2.2.6 Insere

Les fonctions d'insertion prennent en paramètre l'endroit où il faut insérer sous la forme d'un StructGeomPrimitif (§5.2.1) et l'objet à insérer.

Par exemple, dans le cas d'une insertion d'un point dans une polyligne, l'objet à insérer serait le point en question (sous forme de GOCoordonneesXYZ) et l'endroit où insérer serait l'indice du sommet de la polyligne avant lequel se trouvera le point inséré.

5.3 L'éditeur et VTK

L'éditeur fonctionne avec VTK, il est donc important de connaître cette librairie ou du moins la manière dont elle est utilisée ici avant de poursuivre. Des explications sur VTK sont présentées en annexes (8.2).

Dans le cadre de l'éditeur de Modeleur 2.0, on doit pouvoir manipuler 3 types d'objets simultanément :

- des points ;
- des polylignes ;
- des polygones.

Ces différents objets sont représentés de la manière suivante dans l'éditeur :

- Points : Les points libres sont représentés par un ou plusieurs acteurs (vtkActor : 8.2.7). On peut utiliser un acteur par point mais ce n'est pas recommandé car le

nombre d'acteurs que VTK peut gérer simultanément est trop limité. Les points sont identifiés par leur id de cell[6].

- Polylignes : on représente chaque polyligne par un acteur. Chaque point de la polyligne est représenté et identifié par une cell (vtkVerts) contenant uniquement ce point. La polyligne est représentée et identifiée par une cell (vtkLines) contenant les points de la polyligne dans l'ordre.
- Polygones : on représente chaque polygone par un acteur. Chaque point du polygone est représenté par une cell vtkVerts. Chaque polyligne du polygone est représentée et identifiée par une cell vtkLines qui ici, sert à exprimer un contour. Chaque polygone est représenté et identifié par une cell vtkPolys qui permet d'exprimer sa surface.

Cette méthode[7] de représentation facilite grandement l'association entre les objets VTK et les objets géométriques. Il est important de pouvoir associer les éléments de l'éditeur à celui de l'édité pour pouvoir effectuer correctement les actions demandées d'un bord comme de l'autre.

Cette méthode[8] de représentation permet de plus de pouvoir sélectionner pour chaque type d'objet chacune de ses composantes indépendamment des autres. Par exemple, il est ainsi possible de sélectionner :

- un point d'un polygone ;
- une polyligne de polygone ;
- un polygone au complet.

On ne peut sélectionner et donc effectuer des actions telles que la suppression ou le déplacement que sur un type d'objet à la fois[9] (point, polyligne ou polygone). Il est par contre possible de déplacer un point de polygone en même temps qu'un point de polyligne et un point simple.

Lorsqu'il s'agit d'insérer un point ou une polyligne ou de détruire une partie d'un objet, la solution retenue dans l'éditeur est de détruire complètement l'acteur correspondant et d'en recréer un nouveau. Une exception cependant, lorsqu'on ajoute un point *libre* il n'est pas nécessaire de détruire l'acteur représentant l'ensemble des points, on l'ajoute simplement à la fin de ce dernier. (voir aussi 5.1.1.2)

---

[6] Cell : une cell est un objet VTK qui représente un ensemble ordonné constitué d'un ou plusieurs points. Il existe différents type de cell. Dans l'éditeur 3 sont utilisés les **vtkVerts** qui sont des cell à un seul point permettant de représenter des points libres, des **vtkLines** qui servent à représenter les polylignes et les **vtkPolys** qui servent à représenter les polygones.

[7] La décision d'utiliser un acteur par objet géométrique (sauf pour les points libres) fait suite à la réunion du 09/03/04 sur l'éditeur #2

[8] Il faut cependant faire attention à la manipulation des indices quand on communique avec *EDInterfaceContainer* : si *EDInterfaceContainer* nous parle de la N-iéme polyligne d'un polygone, il faut comprendre qu'il parle de la cell numéro (*nombre de points du polygone* + *N*) puisqu'il faut tenir compte des points qui sont eux aussi chacun représenté par une cell.

[9] Cette décision a été prise lors de la réunion du 02/03/04 sur l'éditeur pour ne pas alourdir inutilement le fonctionnement de l'éditeur.

6 Conclusion

6.1 Résumé

L'éditeur est indépendant des structures à éditer, il ne connaît que leur partie géométrique.

Pour utiliser l'éditeur il faut disposer d'une implémentation de l'interface *EDInterface-Container (5.2)*.

L'éditeur doit maintenir un lien entre les objets VTK et les objets édités. Ce lien s'appuie sur les références des objets et sur les indices de leurs constituants.

C'est le gestionnaire d'événements de l'éditeur qui indique à l'éditeur quelles actions effectuer. Par conséquent, pour utiliser l'éditeur, il faut communiquer avec son gestionnaire d'événements.

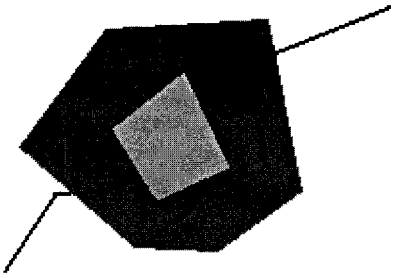
6.2 Perspectives futures

L'éditeur est maintenant bien avancé et les fonctionnalités de bases sont implémentées. Il reste encore à ce jour à lui faire subir une procédure de tests automatisée. Les fonctions de l'éditeur peuvent encore être optimisées. Il serait aussi intéressant de voir son comportement dans un cas réel d'utilisation.

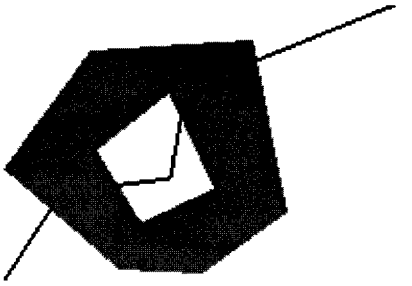
Il reste encore à savoir comment sera déterminée la couleur des différents objets à afficher. Pour le moment, cette couleur est définie par l'éditeur.

Quelques points sont encore non résolus comme :

- Comment sélectionner un point d'une polyligne ou d'un polygone qui se trouve superposé à un autre ? Si deux points sont superposés, c'est toujours celui du dessus qui sera sélectionné. L'utilisateur n'a pour le moment d'autre choix que de déplacer le point du dessus pour pouvoir sélectionner le point du dessous.
- Comment gérer la transparence des trous ? Les trous doivent normalement permettre de voir ce qu'il y a dedans. Actuellement, ce n'est pas le cas. La technique de rendre les trous transparents ne fonctionne pas, car ce qu'on voit alors au travers, c'est le polygone contenant le trou. De plus, il ne nous est pas permis de découper le polygone puisqu'il est nécessaire garder la correspondance entre les sommets des objets VTK et ceux de *EDInterfaceContainer.*



Situation actuelle



Situation idéale

- Quel comportement donner au copier-coller ? Le problème du copier coller se situe dans la partie coller. En effet, certaines positions sont interdites, il n'est alors pas évident de savoir où coller ce qu'on a copié. Une idée serait de faire plutôt un cloner déplacement-copie.
- Bugs d'affichage de VTK. Quand on dessine un polygone à plus de 4 sommets avec un angle rentrant, VTK dessine un voile noir indésirable.
- Pour le moment le conteneur dans lequel sont les points est un *std::vector* construit pour l'occasion mais on pourrait imaginer d'implanter un itérateur directement sur un objet VTK.

7 Glossaire

- **Callback :** principe de programmation qui s'exprime par le fait de passer une fonction à une autre fonction pour que cette dernière l'appelle au moment opportun (voir exemple en 5.2.2).
- **Conteneur :** le conteneur est la classe éditée (*EDInterfaceContainer* étant l'interface de ce conteneur). On peut aussi parler d'édité. On trouvera aussi parfois la notation SRChamp ou SRChampVector, cela parce qu'*EDInterface-Container* est actuellement implanté avec un SRChampVector (*EDContainer-SRChampVector*).
- **Editeur :** on parle d'éditeur pour désigner suivant le contexte la classe EDEditeur contenant les méthodes d'édition ou le module éditeur.
- **Edité :** on parle d'édité pour désigner la structure que l'on veut représenter et sur laquelle on désire interagir graphiquement.
- **Itérateur :** structure permettant de parcourir toutes les parties d'un objet (par exemple : chaque élément d'une liste). il est sous entendu ici que lorsqu'on passe un itérateur en paramètre à une fonction on passe en réalité l'itérateur de début et celui de fin (pour savoir quand arrêter l'itération).
- **Multipoint, multipolyligne, multipolygone :** servent à désigner plusieurs entités. Par exemple, une multipolyligne désigne de 1 à n polylignes.
- **Objet géométrique :** désigne les objets que l'on désire éditer sans toutes les informations supplémentaires. Désigne un ensemble de coordonnées ordonnées suivant un ordre particulier
- **Objet VTK :** désigne les acteurs (vtkActor) représentants les objets géométriques.
- **Point :** Un point est une géométrie de dimension 0. Un point est représenté par ses 3 coordonnées x, y et z. Dans le présent rapport, le terme « point libre » sera utilisé pour désigner un point qui n'appartient ni à une polyligne ni à un polygone.
- **Polygone :** Un polygone est une surface plane avec 0 à n trous. L'intérieur d'un polygone forme un ensemble connexe de points. Un trou est représenté par un polygone dans le polygone.
- **Polyligne :** Une polyligne est un ensemble ordonné de plus de 2 points reliés 2 à 2 par un segment de droite.
- **Vecteur de déplacement :** Un vecteur définit 3 paramètres : un sens, une direction et une longueur. Un vecteur peut être représenté par 2 coordonnées ordonnées. Dans l'éditeur il n'est représenté que par sa coordonnée d'arrivée car on considère que la coordonnée de départ est l'origine.
- **VTK :** Visualisation ToolKit. Ensembles de librairies graphiques. Se reporter à la partie 8.2.

8 Annexes

8.1 Diagramme du Module Editeur

Le module Editeur (*Module_Editeur)* est composé de 3 classes principales dont une virtuelle pure :

- La classe *EDGestionnaire* qui gère les événements et appelle en réponse les fonctions concernées de l'éditeur. Cette classe hérite de *TEModuleBase* afin de pouvoir recevoir des événements.
- La classe *EDEditeur* qui effectue les différentes actions de base de l'éditeur.
- La classe virtuelle pure *EDInterfaceContainer* qui définit une interface minimum à implémenter pour les classes qu'on désire éditer.

Il contient aussi les classe descendants de *EDEven* (*EDEvenEditionSouris* et *EDEven-EditionMenu*) et qui définissent les événements propres à l'éditeur. La classe *EDEven* descend de *TEEvenement*.

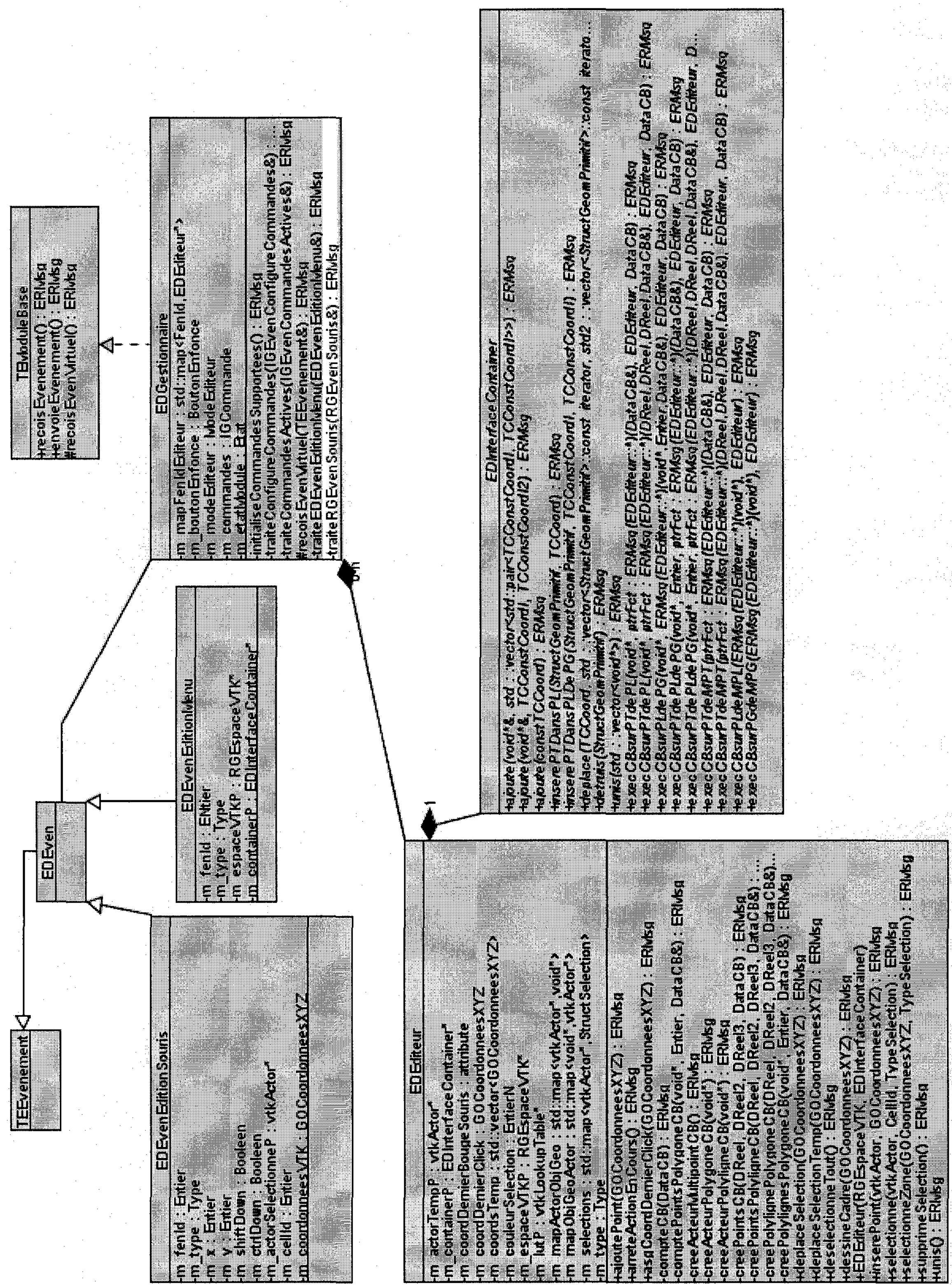


Diagramme de classes de l'éditeur

8.2 VTK

VTK (Visualisation ToolKit) est un ensemble de librairie permettant de faire du traitement graphique.

Le site officiel de VTK : http://public.kitware.com/VTK

On peut y trouver la réponse à de nombreuses questions, des exemples et toutes les classes officielles disponibles.

Toutes les fonctionnalités de VTK ne sont pas utilisées dans l'éditeur. Les quelques connaissances à avoir sont reprises dans les paragraphes suivants.

8.2.1 vtkPoints

Une liste de points. Cette classe est utile pour définir les sommets des différentes figures à dessiner. Tout ce qui est dessiné dans l'éditeur s'appuie là dessus.

8.2.2 VtkCell

une cell est un objet VTK qui représente un ensemble ordonné constitué d'un ou plusieurs points. Les cells permettent de représenter un point, une polyligne ou un polygone sans trous.

8.2.3 VtkCellArray

Tableau de listes de points. Permet de définir comment seront reliés les vtkPoints. Dans l'éditeur ils seront identifiés par les noms suivants : verts pour les vtkCell ne contenant qu'un seul point, lines pour les vtkCell formant un polyligne (2 à n points) et polys pour les vtkCell formant des polygones (3 à n points).

8.2.4 vtkFloatArray

Tableau de valeurs. Dans l'éditeur on s'en sert pour définir la couleur des vtkCell

8.2.5 vtkPolyData

Entité qui regroupe les vtkPoints, les vtkCellArray et vtkFloatArray (pour la couleur).

```
vtkPolyData* polyDataP = vtkPolyData::New(); // Crée un nouveau vtkPolyData
polyDataP->SetPoints(pointsP);          // affecte des points au PolyData
pointsP->Delete();                      // Libère la mémoire
polyDataP->SetVerts(vertsP);            // Définit les points libres du PolyData
vertsP->Delete();                       // Libère la mémoire
polyDataP->GetCellData()->SetScalars(tabP); //Définit les couleurs des différentes cell du PolyData
tabP->Delete();                         // Libère la mémoire
```

8.2.6 vtkPolyDataMapper

Entité représentant ce qui sera réellement affiché à l'écran. Le vtkPolyDataMapper est construit à partir d'un vtkPolyData.

```
vtkPolyDataMapper* mapperP ;
mapperP = vtkPolyDataMapper::New();      // Crée un nouveau vtkPolyDataMapper
mapperP->SetInput(polyData);             // définit le VtkPolyData utilisé pour ce mapper
polyData->Delete() ;                     // Libère la mémoire
mapperP->ScalarVisibilityOn();           // Les valeurs définies dans le vtkPolyData seront utilisées
mapperP->SetScalarRange(0., m_lutP->GetNumberOfColors()); // Définit le range des couleurs utilisées
mapperP->SetLookupTable(m_lutP);         // Définit la table de couleurs utilisée
```

8.2.7 vtkActor

C'est l'unité la plus haute de représentation graphique. Il désigne un ensemble de formes géométriques ainsi que leurs propriétés.

```
vtkActor* actorP = vtkActor ::New() ;    // Crée un nouvel acteur
actor.SetProperty(propertyP);            // Définit les propriétés de cet acteur
propertyP->Delete();                     // Libère la mémoire
actor.SetMapper(mapperP);                // Définit le vtkPolyDataMapper de cet acteur
mapperP->Delete();                       // Libère la mémoire
actorP->PickackeOn();                    // Définit l'acteur comme sélectionnable
```

8.2.8 vtkProperty

Permet de définir certaines propriétés de l'acteur comme la taille des points ou l'épaisseur des lignes.

```
vtkProperty* propertyP = vtkProperty::New();   // Crée de nouvelles propriétés
propertyP->SetPointSize(3.);                   // Définit la grosseur des points
propertyP->SetLineWidth(2.);                   // Définit l'épaisseur des lignes
vtkActor* actorP = vtkActor ::New();           // Crée un nouvel acteur
actorP->SetProperty(propertyP);                // Affecte les nouvelles propriétés
propertyP->Delete() ;                          // Libère la mémoire
```

8.2.9 vtkLookupTable

Sert à définir les couleurs qui sont utilisées dans l'éditeur. Exemple d'utilisation :

```
vtkLookupTable  lutP;
lutP = vtkLookupTable::New();            // Crée une nouvelle table de couleur
lutP->SetNumberOfTableValues(2);         // Définit le nombre de couleurs
lutP->SetTableValue(0, 0., 0., 0.);      // Noir
lutP->SetTableValue(1, 1., 0., 0.);      // Rouge
```

8.2.10 Autre

Quelques formules se retrouvent souvent au long de l'éditeur :

- La fonction *Allocate* : sert à définir est à allouer la mémoire nécessaire :

```
vtkPoints* pointsP = vtkPoints::New();
pointsP->Allocate(10); // Alloue la mémoire pour 10 points
```

- Les fonctions *GetPointsCell* et *GetCellsPoint* qui permettent respectivement de récupérer tous les points appartenant à une cell et toutes les cell contenant un point particulier.
- La fonction *GetMapper* qui permet de récupérer le vtkPolyDataMapper d'un acteur et la fonction *GetInput* qui permet de récupérer le vtkPolyData d'un vtkPolyDataMapper.

```
// Récupère le PolyDataMapper d'un vtkActor (actor)
vtkPolyDataMapper* pdmP = dynamic_cast<vtkPolyDataMapper*> (actor.GetMapper());
vtkPolyData* pdP = NULL;
if (pdmP)
pdP = dynamic_cast<vtkPolyData*> (pdmP->GetInput()); // récupère le PolyData d'un PolyDataMapper
```

- Les fonctions *GetPoints, GetCell* (ou *GetVerts*, *GetLines* et *GetPolys*) et *GetScalars* qui permettent de récupérer respectivement les points d'un vtkPolyData, les différentes vtkCell d'un vtkPolyData et les valeurs portées par ces points (pour définir les couleurs).

8.3 Solutions non retenues

8.3.1 Pour lire les objets édités

Avant de choisir une solution s'appuyant sur un système de callback, un héritage d'itérateur avait été imaginé. Cette solution présentant quelques difficultés et n'ayant pas de réel avantage a donc été abandonnée.

Avec ce principe, le module *Module_Editeur* devait définir des classes virtuelles d'itérateurs qui devaient être implémentées par toute classe voulant implémenter *EDInterfaceContainer*. Cela permettait alors à l'éditeur d'itérer sur les différentes dimensions (polygone, polyligne, point) des objets du container et à partir de cela de construire les objets VTK correspondant. Cette méthode présentait l'inconvénient d'imposer de nombreuses classes virtuelles d'itérateurs, ce qui obligeait ceux qui implémentent un *EDInterfaceContainer* à implémenter de nombreux itérateurs. De plus, le design de cette méthode ne fonctionnait pas car il prévoyait que l'éditeur

n'aurait à connaître que la classe mère de toutes les classe d'itérateur, or il devait lui-même instancier des itérateurs et il est impossible d'instancier des classes virtuelles pures.

8.3.2 Pour les fonctions minimales de EDInterfaceConteneur

Une autre idée pour avoir un ensemble de fonction minimum pour EDInterfaceContainer aurait été de n'avoir que 2 fonctions : ajoute, détruis. Dans ce cas, *EDEditeur* ferait tout le travail et demanderait l'ajustement d'*EDInterfaceContainer* en demandant la suppression des objets puis en les recréant.

Cette solution n'a pas été retenue car outre les problèmes de performance qu'elle peut poser, il s'avère que l'on ne serait pas capable de conserver les informations non géométriques, ce qui entre en contradiction avec le choix d'un Module Editeur pouvant éditer toute structure.

RAPPORT # 2 : Géométries et SRChamp Vector

Rapport de développement logiciel

Géométries et SRChampVector

Présenté par :

Maude Giasson

18 janvier 2005

Versions du document

Date	Auteur	Commentaires	Version
11 03 2004	Maude Giassson	Version initiale	V1
21 04 2004	Maude Giasson	Modifications suite aux commentaires d'Éric; Ajout d'une dizaine de pages	V3
23 04 2004	Maude Giasson	Modifications suite à la relecture d'Éric	V4
28-04-2004	Maude Giasson	Révision suite aux notes d'Éric et M.Secretan	V5

1. Introduction

1.1 Objectifs

Ce rapport se veut une introduction aux classes servant à représenter des géométries ainsi qu'au SRChampVector, un champ particulier pouvant faire porter des valeurs aux géométries.

Les principales contraintes, idées et applications ayant été étudiées lors de la conception de ces classes seront décrites ici.

1.2 Référence

Rapports locaux :

[1] Isolignes et isosurfaces – 5-02-2004 – vx.doc
[2] Isolignes et isosurfaces – 19-02-2004 – vx.doc
[3] Isosurfaces – 26-02-2003 – vx.doc
[4] Réunion 2 – isolignes et isosurfaces – 10-02-2004.doc
[5] SRChampVector – 12-02-2004 – vx.doc
[6] SRChampVectorII – 17-02-2004 – vx.doc
[7] SRChampVector porteur de valeur – 5- 04-2004- vx.doc
[8] Rapport d'analyse – Classes de géométries – 07-04-2004 – vx.doc
[9] Rapport de développement de logiciel Champs-Séries

Références externes :

[10] OpenGIS Simple Features Specification For SQL
[11] JTS Topology Suite, Technical Specifications
[12] Site web de TerraLib : http://terralib.dpi.inpe.br/home.html
[13] Site web de GEOS : http://geos.refractions.net/
[14] Site web de JTS : http://www.vividsolutions.com/jts/JTSHome.htm

1.3 Contexte

Lors d'un stage précédent, les champs et séries ont étés développés. Un champ supporte une région et, pour chacun des points de cette région, le champ peut donner une valeur correspondant à ce point. La manière d'obtenir la valeur en un point donné de la région varie pour chacun des types de champ.

À partir de certains champs, il est possible d'effectuer un calcul d'isolignes et isosurfaces. Une isoligne est une ligne de niveau. Par exemple, une carte topographique représente les

isolignes d'altitude sur une région donnée. Pour leur part, les isosurfaces sont les surfaces séparant ces isolignes.

Un calcul d'isolignes et d'isosurfaces pouvant être long, il est intéressant de concevoir un objet intermédiaire permettant d'enregistrer le résultat. Par exemple, avec un résultat d'isolignes enregistré dans une structure quelconque, il devient possible d'effectuer différents affichages sans avoir à refaire tout le calcul, ou même de déplacer certaines isolignes.

Modéliser une structure pouvant représenter des isolignes et isosurfaces revient, si on généralise, à concevoir une structure capable de représenter des géométries (telles les polylignes et polygones) et de leur faire porter de l'information. Le travail de généralisation devient donc très intéressant car le cadre d'utilisation d'une telle structure peut largement dépasser le cas très particulier des isolignes et isosurfaces.

Par exemple, un éditeur de géométries aurait certainement besoin d'utiliser une structure capable de représenter des géométries similaires à celles développées pour les isolignes et isosurfaces.

Ce rapport de design se concentre sur le SRChampVector, classe capable de faire porter de l'information à des géométries. À cela s'ajoute la conception des diverses classes pouvant représenter les géométries.

1.4 Structure

Ce document débute par une introduction à des notions géométriques de base et aux spécifications GIS. Ensuite, une librairie GIS nommée TerraLib est étudiée et ce, dans le but d'introduire les classes de géométries servant de proxy entre TerraLib et Modeleur 2.0. Finalement, le contenu porte sur le design du SRChampVector.

2 Notions de géométrie

Cette section a pour objectif de définir des notions de bases de géométrie, non pas dans un but de rigueur excessive, mais plutôt pour introduire des concepts revenant souvent dans ce texte. Les définitions GIS plus complètes et rigoureuses se trouvent dans le document [1].

Polyligne :
Une polyligne (*lineString* en GIS) repose sur une liste ordonnée de coordonnées. La polyligne est l'union des segments de droite compris entre chacune des paires de coordonnées consécutives dans cette liste. Les coordonnées représentent les **sommets** de la polyligne.

Les divers segments de droite formant la polyligne peuvent se recouper ou se superposer sans aucune restriction.

La **frontière** d'une polyligne est constituée des deux sommets aux extrémités de la polyligne. Dans le cas où ces sommets sont les mêmes, la polyligne a une frontière vide et est dite **fermée**.

Une polyligne est **simple** si elle ne passe pas deux fois sur le même point, à part peut-être sur sa frontière. Un type de polyligne particulièrement important est la polyligne **simple et fermée** (*linearRing* en GIS).

MultiPolyligne :
Dans ce texte, la multiPolyligne est définie comme une collection de polylignes sans conditions. Toutefois, la définition GIS d'une multiPolyligne impose certains critères. Par exemple, les polylignes formant une multiPolyligne GIS ne peuvent se croiser qu'en leur frontière.

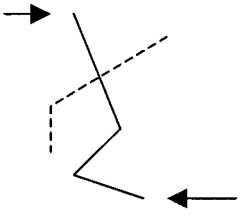


Figure 1 :
Ces deux polylignes ne se croisent pas en leur frontière; elles forment une multiPolyligne valide pour ce document, mais invalide selon la définition GIS.

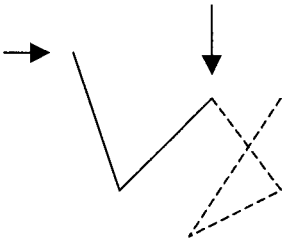


Figure 2 :
Ces deux polylignes se croisent en leur frontière. Elles forment une multiPolyligne valide pour ce document et selon la définition GIS

Polygone :
Un polygone est une surface plane, définie par une frontière extérieure et 0 ou plusieurs frontières intérieures (des trous) respectant certaines assertions. Pour une liste exhaustive

de ces assertions, voir [10] section 2.1.10. Parmi les plus importantes, notons que les frontières d'un polygone forment un ensemble de polylignes simples et fermées, ne se croisant pas et que l'intérieur de tout polygone forme un ensemble connexe de points.

On dit d'un ensemble de points qu'il est **connexe** s'il est possible de relier toute paire de points de l'ensemble par une polyligne continue ne sortant pas de l'ensemble.

Les **sommets** d'un polygone correspondent aux sommets des polylignes simples et fermées formant ses frontières.

MultiPolygone :
Dans ce texte, le multiPolygone est défini comme une collection de polygones sans conditions. Toutefois, la définition GIS d'un multiPolylgone impose certains critères. Par exemple les polygones formant un multiPolygone GIS ne peuvent s'intersecter qu'en un nombre fini de points.

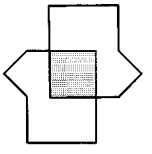


Figure 3 :
Il y a une infinité de points dans l'intersection de ces deux polygones. Ils ne forment donc pas un multiPolygone valide selon les définitions GIS, mais forment un multiPolygone valide selon le présent document.

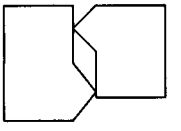


Figure 4 :
Il y a 2 points dans l'intersection de ces deux polygones. Ils forment donc un multiPolygone valide selon les définitions GIS et selon le présent document aussi.

Point :
Un point est une géométrie ne recouvrant qu'une seule coordonnée dans l'espace.

3 TerraLib

TerraLib est une librairie GIS. Elle permet de représenter des structures géométriques, d'effectuer certains calculs sur ces dernières et de stocker les résultats dans une base de données. L'objectif n'est pas ici de détailler TerraLib dans son ensemble, mais plutôt d'introduire les outils de TerraLib utilisés dans les classes de géométrie développées. Seul le travail en mémoire de TerraLib sera considéré, laissant ainsi de côté toutes les fonctionnalités concernant l'interaction avec la base de donnée.

Il sera d'abord question de l'architecture des classes de géométrie de TerraLib pour ensuite décrire les copies dans TerraLib. Finalement, une section est consacrée aux algorithmes de TerraLib.

3.1 Architecture des classes de géométrie

Les trois principales géométries qui seront utilisées sont le polygone, la polyligne et le point. TerraLib possède des classes modélisant chacune de ces structures. Il s'agit respectivement de TePolygon, TeLine2D et TePoint. Ces structures ont en commun d'être des géométries. Voici un diagramme représentant la structure hiérarchique de ces classes.

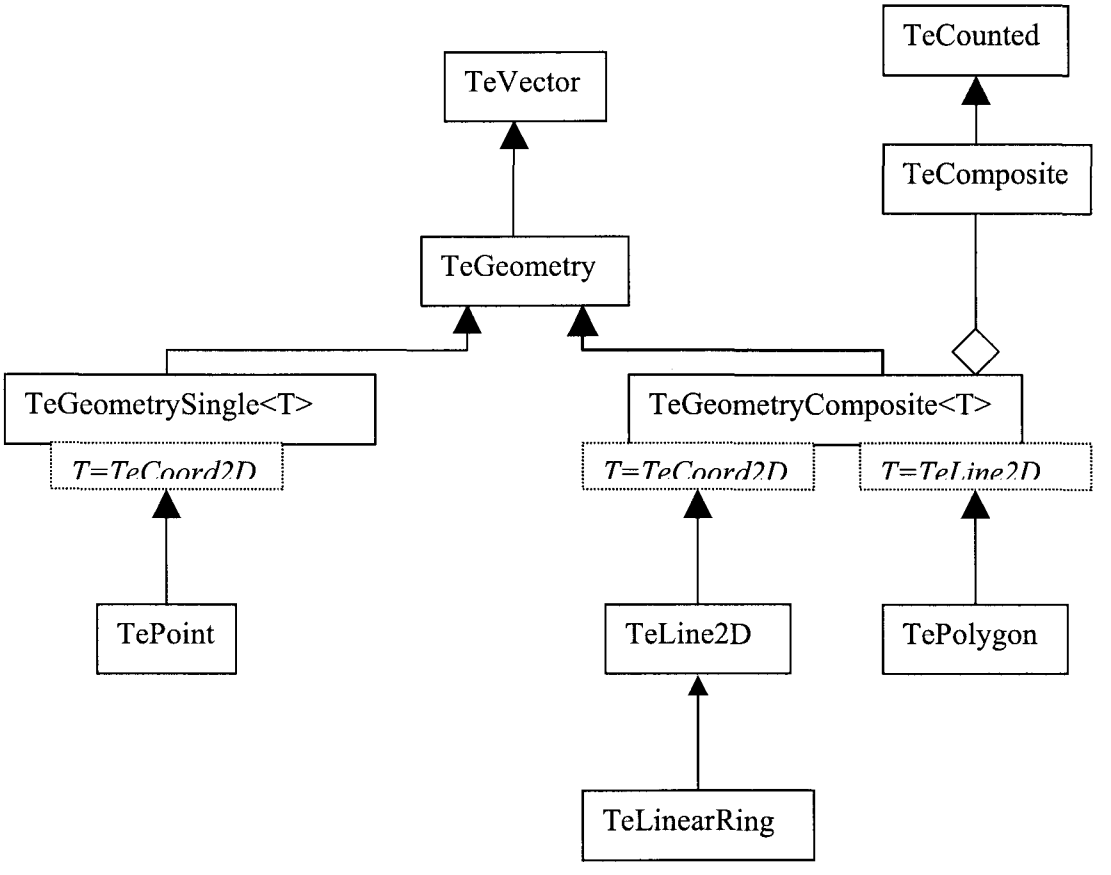


Figure 5 : Architecture des classes de géométries dans TerraLib

3.1.1 TeGeometry

La classe TeGeometry représente une gamme de fonctions et d'attributs qui sont propres à toutes les géométries Parmi les attributs propres à n'importe quelle géométrie, on retrouve la boîte (box) et l'id.

La boîte maintenue dans la classe TeGeometry représente le plus petit rectangle pouvant contenir la structure géométrique au complet. Cette boîte est utilisée dans certains algorithmes de calcul.

La classe TeGeometry, possède des méthodes permettant d'obtenir la boîte ou de la définir, d'obtenir le type de géométrie courante ou un identifiant de la géométrie.

La classe TeGeometry possède deux sous classes, toutes deux templates. Il s'agit de TeGeometrySingle et TeGeometryComposite. Soit T le paramètre template de ces classes. Comme l'indiquent clairement leur nom, les TeGeometrySingle sont formées d'une seule instance de T alors que les TeGeometryComposite nécessitent plusieurs instances de T.

3.1.2 TeGeometrySingle

Cette classe permet de modifier et d'obtenir l'unique élément formant cette géométrie.

3.1.3 TeGeometryComposite

Afin de regrouper les diverses instances de T, la classe TeGeometyComposite possède un attribut pImpl_ qui est un pointeur vers un TeComposite. La classe TeComposite est un container de T qui, grâce à sa classe de base TeCounted, compte le nombre de références pointant sur l'objet courant.

La classe TeGeometryComposite permet entre autre d'ajouter un élément au composite, d'obtenir des itérateurs sur les éléments du composite, de détruire un ou tous les éléments du composite, d'obtenir le nombre d'éléments qui le forme et de copier les éléments d'un composite dans un autre.

3.1.4 Points/polylignes/polygones

Les points (TePoint) sont un TeGeometrySingle contenant une seule coordonnée (TeCoord2D). Les polyligne (TeLine2D) sont un composite de plusieurs de TeCoord2D alors que les polygones (TePolygon) sont formés de plusieurs polylignes fermées (TeLinearRing).

Le polygone, la polyligne et le point possèdent très peu de méthodes qui leur sont propres. La plupart proviennent directement de l'héritage de TeGeometry et TeGeometryComposite/Single. Par exemple, la seule méthode propre à la classe TeLine2D est la méthode isRing() qui permet de savoir si la TeLine2D est en réalité une TeLinearRing.

3.2 Mode référence

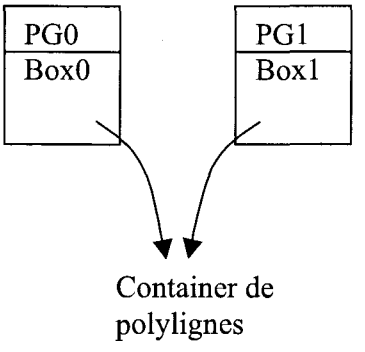
Dans le but de minimiser le nombre de copies devant être faites, TerraLib travaille principalement avec des références aux objets. Il est nécessaire de bien comprendre le contrat que TerraLib nous passe en travaillant de la sorte.

Un exemple reflétant l'importance de considérer le comportement « par référence » de TerraLib lors du développement de nos classes est l'ajout d'une polyligne à un polygone.

En effet, on peut ajouter une TeLine2D à un TePolygon (container de TeLinearRing). Si la TeLine2D est fermée (premier sommet égal au dernier) elle formera une frontière valide et sera simplement ajoutée au polygone. Toutefois, si la TeLine2D n'est pas fermée, TerraLib ajoutera un sommet à la polyligne afin de la fermer avant d'ajouter la polyligne au polygone. Puisque TerraLib travaille en mode référence, un tel ajout de point à la polyligne a des influences non seulement sur le polygone, mais aussi sur la polyligne ayant servie à la construction du polygone. Ce comportement doit être bien compris avant d'utiliser les classes de TerraLib afin de s'assurer que le comportement correspond à celui souhaité. La plupart des difficultés proviennent du fait que TerraLib utilise principalement le mode référence pour copier ses données.

3.2.1 Constructeur de copie

Tout d'abord, considérons le constructeur de copie du TeGeometryComposite. C'est ce constructeur de copie qui est utilisé lors de la copie d'un polygone ou d'une polyligne. Afin de rendre plus claire l'explication, considérons un polygone PG1 à créer en copiant un autre polygone PG0.



Le pointeur au container (TeComposite) de polylignes de PG0 est donné à PG1 et la box de PG0 est copiée dans PG1. Ainsi, PG0 et PG1 possèdent la même liste de polylignes mais des box différentes (quoique égales pour l'instant). Toute modification aux polylignes de PG0 aura un effet direct sur PG1 et vice-versa. Or, ces modifications ne mettent à jour la box que sur l'objet où elles ont été faites. Ainsi, si PG1 est construit en copiant PG0 et qu'ensuite une polyligne est ajoutée à PG1, les deux polygones (PG1 et PG0) auront une nouvelle polyligne, mais seule la box de PG1 sera à jour.

De la même manière, la méthode copyElement de TerraLib ne copie pas les éléments internes des composites un à un, comme on pourrait s'y attendre avec le nom de la méthode. Deux polygones, dont le premier a été copié dans le second à l'aide de la méthode copyElement, auront exactement les mêmes polylignes. Une éventuelle modification à l'un des deux polygones aura des impacts sur l'autre.

3.2.2 Ajout de polyligne au polygone

Un polygone est construit par ajouts successifs de polylignes. C'est la méthode add (de TeGeometryComposite) qui ajoute une polyligne à un polygone. Cette méthode ajoute la polyligne au composite du TeGeometryComposite et met à jour la box. L'ajout d'une polyligne à un TeComposite se fait en copiant la polyligne à ajouter. Or la copie se fait par appel au constructeur de copie des polylignes qui, comme nous l'avons vu, garde la même référence à la liste des sommets. Cette façon de faire doit être bien comprise afin d'éviter d'entraîner des erreurs de cohérence dans les objets utilisés. Un exemple concret clarifiant la problématique lors de l'ajout d'une polyligne à un polygone est détaillé en annexe (9.1) .

3.3 Algorithmes

TerraLib implante certaines fonctions pouvant être utilisées sur des géométries. On retrouve entre autre des fonctions servant à déterminer les relations topologiques, des fonctions pour calculer l'aire d'une géométrie et certaines fonctions auxiliaires.

3.3.1 Relations topologiques

Les fonctions représentant des relations topologiques servent à déterminer si deux géométries sont égales, disjointes, s'intersectent, se touchent, se croisent, se contiennent ou se recouvrent. Dans chacun des cas le résultat du test est un booléen. TerraLib ne permet pas de retrouver, par exemple, le résultat de l'intersection entre deux ou plusieurs polygones.

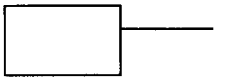
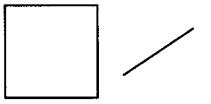
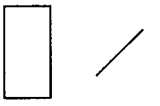
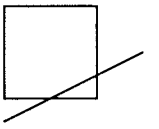
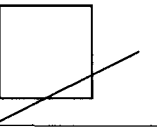
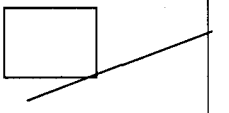
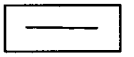
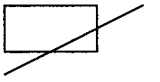
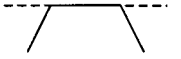
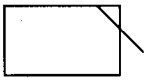
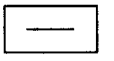
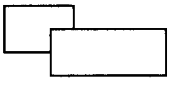
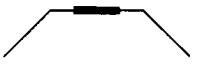
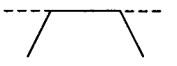
Le tableau 7 présente les diverses relations topologiques pouvant unir deux géométries. La plupart des relations s'appliquent à tous les couples (x,y) dans

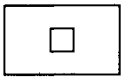
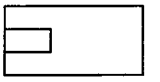
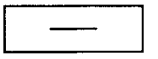
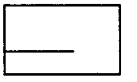
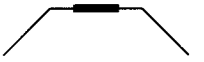
{TePoint, TeLine2D, TePolygon, TePointSet, TeLineSet, TePolygonSet} X
{TePoint, TeLine2D, TePolygon, TePointSet, TeLineSet, TePolygonSet}
sauf dans les cas mentionnés.

Dans la colonne Test se trouve une expression représentant les cas où la relation est vérifiée. Les symboles qui y sont utilisés sont les suivants :

- I : intérieur
- E : extérieur
- B : frontière (boundary en anglais)
- ^ : et
- v : ou
- inter : intersection

Voir annexe (9.2) pour une définition claire de intérieur extérieur et frontière.

Tableau 7 : Relations topologiques pouvant être testées avec TerraLib			
Relation	Test	Exemple Faux	Exemple Vrai
TeEquals(x,y)	Vérifie si une géométrie est égale à l'autre		
TeDisjoint(x,y)	(I(x) inter I(y) = false) ^ (I(x) inter B(y) = false) ^ (B(x) inter I(y) = false) ^ (B(x) inter B(y) = false)		
TeIntersect(x,y)	!TeDisjoint(x,y)		
TeTouches(x,y) S'applique à tous les cas sauf (PT,PT)	(I(x) inter I(y) = false) ^ ((B(x) inter I(y) = true) v (I(x) inter B(y) = true) v (B(x) inter B(y) = true))		
TeCrosses(x,y) S'applique aux 2 cas suivants :	*L'intersection doit former des points et pas des lignes. Dans le cas 1, la ligne doit avoir une partie à l'extérieur du polygone*		
Cas 1 : x est TeLine2D et y est TePolygon	(I(x) inter I(y) = true) ^ (I(x) inter E(y) = true)		
Cas 2 : x est TeLine2D et y est TeLine2D	dim(I(x) inter I(y)) = 0		
TeWithin(x,y)	(I(x) inter I(y) = true) ^ (I(x) inter E(y) = false) ^ (B(x) inter E(y) = false)		
TeContains(x,y)	TeWithin(y,x)		
TeOverlaps(x,y) S'applique aux 2 cas suivants :	*Chacune des composantes (ligne, polygone) possède une partie qui n'est pas dans l'autre*		
Cas 1 : x = TePolygon y = TePolygon	(I(x) inter I(y) = true) ^ (I(x) inter E(y) = true) ^ (E(x) inter I(y) = true)		
Cas 2 : x = TeLine2D y = TeLine2D	(dim(I(x) inter I(y)) = 1) ^ (I(x) inter E(y) = true) ^ (E(x) inter I(y) = true)		

TeCovers(x,y) S'applique aux 3 cas suivants :	*x est complètement contenu dans la frontière et l'intérieur de y et les frontières de x et y se touchent.*		
Cas 1 : x = TePolygon y = TePolygon	TeWithin(y, x) ^ ((B(x) inter B(y) = true))		
Cas 2 : x = TePolygon y= TeLine2D	TeWithin(y, x) ^ ((B(x) inter B(y) = true) v (B(x) inter I(y) = true))		
Cas 3 : x = TeLine2D y = TeLine2D	TeWithin(y, x) ^ ((B(x) inter B(y) = true))		
TeCoveredBy(x,y) S'applique à (PG,PG), (PL,PG) et (PL,PL)	TeCovers(y,x)		

3.3.2 Fonctions auxiliaires

Les fonctions auxiliaires permettent de retrouver quelle relation(voir tableau 7) unit 2 objets et de faire certaines opérations géométriques de base, comme retrouver la distance entre 2 points. Les diverses fonctions de TerraLib sont regroupées en [12] sous la section « Modules ».

4 Besoins face aux géométries

Cette section présente les besoins que nous avons face aux géométries afin de comprendre les choix qui ont motivés la conception des classes de ToolKit_Geometrie.

Trois types de géométries doivent exister, soient le point, la polyligne et le polygone.

La polyligne et le polygone correspondent respectivement à la lineString et au polygon définis en [1]. Les géométries doivent permettre d'effectuer certaines opérations de base, d'assurer leur validité et de répondre à des algorithmes plus élaborés. Elles doivent aussi être suffisamment flexibles pour étendre leur utilisation au cas du SRChampVector. C'est à dire qu'une stratégie permettant d'associer une valeur à chacun des sommets des géométries doit pouvoir être élaborée.

4.1 Opérations de base

Il doit être possible d'ajouter, de retirer et déplacer un ou plusieurs sommets à une polyligne. Cette modification se fait à partir de l'indice du sommet dans la polyligne.

Concernant un polygone, il doit être possible d'ajouter, de retirer ou de déplacer des trous. On travaille à partir de l'indice du trou. Concernant la frontière extérieure d'un polygone, elle peut être déplacée en tout temps mais ne peut être retirée que si le polygone n'a pas de trou. On doit aussi pouvoir ajouter, retirer et déplacer des sommets du polygone. Les sommets du polygone sont identifiés à l'aide de deux indices. Le premier correspond à la frontière (extérieure ou trou) où se trouve le sommet et le second à la position du sommet dans la frontière.

4.2 Validité

Le polygone, la polyligne et le point doivent être en mesure de garantir leur validité. Voici en quoi cela consiste pour chacun de ces types de géométrie.

4.2.1 Point

La validité du point est automatique.

4.2.2 Polyligne

Des erreurs sur la validité peuvent survenir lorsque le nombre de sommets que possède la polyligne entraîne une incohérence. Par exemple, une polyligne à 1 seul sommet n'est pas souhaitable car peu cohérente avec la définition d'une polyligne. Il reste à décider si une polyligne à 0 sommet est tolérée ou non.

4.2.3 Polygone

C'est pour le polygone que la validité est la plus difficile à assurer. Les contraintes décrivant un polygone valide sont décrites en [10]. Il faut, entre autre, pouvoir s'assurer que :

- Les trous sont à l'intérieur de la frontière extérieure du polygone et ne touchent pas à cette frontière;
- Les frontières du polygone forment des polylignes simples et fermées;
- Le polygone est connexe.

Il ne sera pas nécessaire de faire ces vérifications après chaque opération géométrique, l'important étant d'avoir accès à une méthode permettant de vérifier si un polygone est valide ou non. D'autres façons d'effectuer ces vérifications pourraient être acceptables, l'important, c'est d'avoir une stratégie permettant d'assurer la validité d'un polygone.

4.3 Opérations avancées

On doit être en mesure d'effectuer certaines opérations à partir des géométries. D'abord, on peut souhaiter savoir si une coordonnée se trouve sur une géométrie ou non. Une coordonnée est un point intérieur d'une polyligne si elle est sur un de ses segments de ligne et sur un polygone si elle touche sa surface. On doit aussi pouvoir trouver l'intersection, l'union et la différence entre 2 polygones. Dans le cas où le résultat s'exprime avec plus d'un polygone, on doit être en mesure d'itérer sur les divers polygones du résultat.

4.4 Regroupement, porter des valeurs

Les géométries seront utilisées dans divers contextes. Elles seront en particulier à la base d'une structure représentant les isolignes et isosurfaces.

Il faudra donc être en mesure d'avoir une structure capable de représenter à la fois diverses géométries et de faire porter des valeurs aux sommets de ces géométries.

Les géométries doivent donc offrir suffisamment de flexibilité pour qu'on soit en mesure d'associer une valeur à chacun des sommets des géométries.

5 Conception des géométries

La librairie TerraLib est utilisée dans le développement des géométries. Il a toutefois été considéré comme préférable de ne pas retrouver du code TerraLib partout dans Modeleur. Des classes servant de proxy entre Modeleur et TerraLib ont donc été élaborées. En plus de simuler le travail de TerraLib, ces classes permettent d'adapter le comportement de TerraLib au comportement souhaité dans Modeleur et d'ajouter certaines fonctionnalités.

5.1 GOPoint, GOPolyligne, GOPolygone

À la base, nous retrouvons un GOPoint, une GOPolyligne et un GOPolygone. Bien que ces classes contiennent respectivement un TePoint, une TeLine2D et un TePolygon, l'objectif est qu'un client puisse avoir l'impression qu'un GOPoint contient une GOCoord, qu'une GOPolyligne contienne un container de GOCoord et qu'un GOPolygone contienne un container de GOPolyligne.

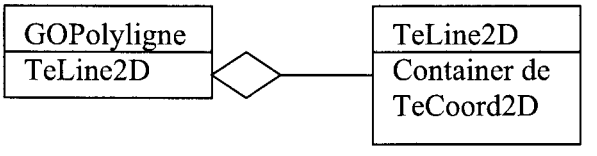


Figure 8 :
La GOPolyligne n'est pas un container de GOCoord; elle contient plutôt une TeLine2D

5.1.1 Opérateur [ ]

Les opérateurs [] ont donc dû être modifiés de manière à simuler le comportement souhaité chez les GO. Soit une GOPolyligne pL0. Lorsqu'on fait pL0[0] on s'attend à recevoir en sortie le premier sommet de cette polyligne. Or, la GOPolyligne ne garde pas en mémoire des GOCoord correspondant à ses sommets (voir figure 8). Il est donc nécessaire de créer GOCoord localement et de la renvoyer par copie.

De la même manière, une GOPolyligne créée localement est retournée par copie lorsqu'on fait appel à l'opérateur [] sur un GOPolygone. Cela justifie la présence d'un constructeur de copie dans la classe GOPolyligne.

L'objet retourné par l'opérateur [] des classes GOPolyligne et GOPolygone ne peut pas servir à modifier la polyligne ou le polygone en question. Il s'agit d'un mode lecture seulement.

5.1.2 Itérateurs

Les classes d'itérateurs ont été construites dans la même optique. Lorsqu'on déréférence un GOConstPolygoneIterateur, c'est une GOPolyligne que l'on obtient même si, en réalité, un GOPolygone ne contient aucune GOPolyligne. Cette astuce est rendue possible par la présence dans les classes d'itérateurs d'une donnée membre permettant de conserver en mémoire l'objet sur lequel on aimerait pointer. Mais, en réalité, un itérateur de GOPolygone ou GOPolyligne n'itère par sur un objet GO. Il contient simplement un itérateur TerraLib sur lequel il redirige toutes les requêtes qui lui sont passées. À l'aide de l'objet TerraLib sur lequel son itérateur TerraLib pointe, l'itérateur GO peut mettre à jour sa donnée membre GO au moment d'être déréférencé.

Bien entendu, cette manière de faire ne fonctionne que pour des itérateurs constants car l'objet pointé par l'itérateur ne peut pas réellement être modifié. La restriction aux itérateurs constants était aussi nécessaire pour assurer la validité d'un polygone. En effet, seul le polygone lui-même est en mesure d'assurer sa validité. S'il devient possible d'itérer sur les polylignes d'un polygone et les modifier individuellement, il devient impossible d'assurer la validité du polygone car ce dernier n'est pas nécessairement informé des changements ayant eu lieu sur ses polylignes.

5.1.3 Modification du contenu

Les itérateurs et l'opérateur[] ne permettant pas de modifier le contenu des GOPolylignes et GOPolygones, des fonctionnalités effectuant diverses modifications sur ces objets ont dues être élaborées. Le besoin de modifier les objets de plusieurs manières différentes est en particulier venu de l'utilisation du SRChampVector (et donc des géométries) dans un contexte d'édition. Plusieurs opérations géométriques ont alors été définies et ces dernières ont des répercussions directes sur les fonctions devant être implantées dans les géométries. Voici la liste des opérations maintenant possibles sur les géométries et les particularités de certaines d'entre elles :

- Polygone
    - Ajouter/déplacer/retirer un sommet à une polyligne du polygone.
        - On doit connaître l'indice de la polyligne à laquelle on souhaite altérer un sommet et l'indice du sommet dans la polyligne.
    - Ajouter une polyligne à un polygone.
        - Si le polygone est vide, cette polyligne est une frontière extérieure. Sinon, c'est un trou dont l'indice est égal au nombre de polylignes dans le polygone après son insertion.
    - Retirer une polyligne à un polygone.

        - On doit connaître l'indice de la polyligne à retirer. Si c'est la frontière extérieure (indice 0), le polygone doit n'avoir aucun trou.
    - Déplacer une polyligne d'un polygone.
        - On doit connaître l'indice de la polyligne à déplacer dans le polygone.
    - Déplacer le polygone en entier.
- Polyligne
    - Ajouter/déplacer/retirer un sommet à la polyligne.
        - On doit connaître l'indice du sommet que l'on souhaite altérer dans la polyligne.
    - Déplacer la polyligne en entier.
- Point
    - Déplacer le point.

Diverses méthodes ont été implantées dans les géométries afin d'effectuer les opérations définies ci-haut. On a pris soin de mettre à jour le box TerraLib après chacune des opérations. Dans plusieurs cas, cela implique la reconstruction totale du box de la géométrie. Ces méthodes ne vérifient que très partiellement la validité de l'objet qu'elles modifient. Il est donc possible d'effectuer des opérations qui invalident un objet. TerraLib n'étant pas en mesure d'assurer et de vérifier la validité d'un objet géométrique, en particulier d'un polygone, il faudra développer une stratégie permettant de répondre à ce besoin.

5.1.4 Copies

Il a été question en 3.2 du mode référence utilisé par TerraLib. Dans la mise en place des GO, des constructeurs par copies effectuant de réelles copies ont été implantés. Ils n'appellent donc pas directement le constructeur de copie de TerraLib. Ainsi, il devient plus facile pour un éventuel client de suivre la trace de ses objets. De la même manière, lorsqu'une GOPolyligne est ajoutée à un GOPolygone, cette dernière est réellement copiée. Ainsi, la polyligne originale demeure intacte.

5.1.5 Problèmes en suspend

5.1.5.1 Validité des polygones

Malgré le fait qu'elle fut identifiée en (4.2.3) comme un besoin face aux géométries, la validité d'un polygone n'est pas vérifiée dans les GO. Il est donc possible d'obtenir un polygone invalide. Par exemple, une frontière d'un trou peut être déplacée à l'extérieur du polygone et ce, sans générer d'erreur. Ce problème de gestion de la validité dans un polygone devra être réglé dans le futur.

5.1.5.2 Gestion des arrondis

De la même manière, la gestion des arrondis n'a pas été considérée. Pour l'instant, seule la méthode pointInterieur des GOPolyligne et GOPolygone est susceptible de souffrir de cette mauvaise gestion.

Cette méthode prend en paramètre une coordonnée et vérifie si cette coordonnée se trouve sur l'objet courant. Par exemple, avec une polyligne, pointInterieur doit retourner

vrai si la coordonnée passée est sur une arête de la polyligne ou si elle est un de ses sommets. Pour ce faire, la méthode pointInterieur appelle la méthode TeIsOnLine de TerraLib.

```
bool TeIsOnLine(  const TeCoord2D& c,
                  const TeLine2D& l,
                  const double& tol = 0.0)
```

Vérifie si la coordonnée c est sur la frontière ou à l'intérieur de la ligne l.

Lors de la phase de test, une polyligne a été construite à partir de quatre sommets. Ensuite, la méthode pointInterieur de la polyligne fut appelée avec un de ces quatre sommets en paramètre. On aurait pu s'attendre à ce que cette méthode retourne « vrai », ce qui ne fut pas le cas.

Voici l'exemple en question. Il s'agit de vérifier si la coordonnée (0.4 , 0.7) se trouve à l'intérieur de la polyligne suivante :

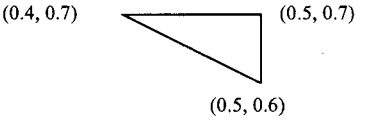


Figure 9
Exemple de polyligne où pointInterieur donne un résultat erroné

Évidemment, la réponse théorique est oui puisque, en plus d'être sur la polyligne, la coordonnée (0.4 , 0.7) en est un sommet.

Pour avoir la réponse du programme, on appelle la méthode pointInterieur de GOPolyligne qui, à son tour, appelle la méthode TeIsOnLine de TerraLib avec une tolérance nulle. Après avoir vérifié que la coordonnée est bien dans le box de la polyligne, la méthode TeIsOnLine itère sur les segments de la polyligne et, à chaque fois, appelle la méthode TeIsOnSegment de TerraLib.

```
bool TeIsOnSegment(  const TeCoord2D &c,
                     const TeCoord2D& c1,
                      const TeCoord2D& c2,
                      const double &tol=0.0)
```

Vérifie si la coordonnée c est sur le segment reliant c1 et c2; le segment est fermé

Après avoir vérifié si la coordonnée c appartient au box du segment, la méthode TeIsOnSegment trouve l'équation du segment reliant c1 et c2 et regarde si c vérifie cette équation ou non.

C'est dans la construction de cette équation que se glisse une erreur d'arrondi. En effet, alors que l'équation, sous forme ax + by + c = 0, devrait être 0.1y – 0.07 = 0, on obtient plutôt 0.099999999999999978y -0.070000000000000007. Bien qu'il ne s'agisse là que d'une infime erreur, cette différence n'est pas tolérée lorsque le facteur de tolérance est de 0.0. Voici, étape par étape comment se construit cette équation lorsqu'on appelle TeIsOnSegment avec : c = (0.4, 0.7), c1 = (0.5 , 0.7), c2 = (0.4 , 0.7) et tol = 0.0.

```
Code C++                                          Valeur assignée

xref = c.x();                                     0.40000000000000002
yref = c.y();                                     0.69999999999999996
xi = c1.x();                                      0.50000000000000000
yi = c1.y();                                      0.69999999999999996
xf = c2.x();                                      0.40000000000000002
yf = c2.y();                                      0.69999999999999996

a = yf - yi;                                      0.00000000000000000
b = xi - xf;                                      0.099999999999999978
c = xf * yi - xi * yf;                           -0.070000000000000007

d = fabs(a * xref + b * yref + c);                2.7755575615628914e-017
if(d <= tol)
      return true; //true n'est pas retourné alors qu'il devrait l'être
```

Cet exemple démontre que lorsque pointInterieur appelle TeIsOnLine, le facteur de tolérance assigné ne devrait pas être de 0 comme il l'est présentement. Une première idée de solution serait de définir un chiffre assez petit, mais pas 0, correspondant à un seuil de tolérance minimal. Dans l'exemple plus haut, une tolérance de l'ordre de 10^{-15} aurait suffit. Or, dans un autre cas, cette tolérance pourrait ne pas être suffisante.

Bref, la gestion des arrondis est un problème en suspend. Il faudrait arriver à trouver une manière d'adapter le seuil de tolérance à chacun des cas que l'on pourrait rencontrer.

5.2 Les Multi

Les clients des géométries auront souvent à interagir avec un ensemble contenant plusieurs géométries à la fois. Ces géométries pourront toutes être du même type ou bien être de types différents. Le SRChampVector est un exemple de client des géométries qui contient à la fois un ensemble de polygones, un ensemble de polylignes et un ensemble de points.

Il a donc fallu créer des classes capables de contenir plusieurs géométries à la fois, soient les classes GOMultiPoint, GOMultiPolyligne et GOMultiPolygone. Chacune des classes GOMultiXX contient un vecteur de pointeurs vers des objets de type XX. Il devait être possible d'itérer sur les divers objets que contient un GOMultiXX. Or, lorsque l'itérateur se voit déréférencé, c'est l'objet (de type XX) que l'on veut avoir et non un pointeur vers ce dernier. Ainsi, des classes d'itérateurs sur les GOMulti ont été développées de manière à reprendre le comportement des itérateurs sur les vecteurs mais à modifier le comportement de l'opérateur de déréférencement de manière à ce qu'il retourne une référence (constante) à l'objet plutôt qu'un pointeur à l'objet.

Pour l'instant, seuls les itérateurs constants ont été créés. En fait, il serait possible de construire un itérateur non constant sur les GOMulti. Il deviendrait ainsi possible d'itérer sur un GOMulti et de modifier ses diverses composantes. Pour l'instant, cela ne causerait pas de problèmes. Toutefois, il se peut qu'à long terme le rôle des multi soit modifié. Par exemple, les multiPolygones pourraient être spécialisés afin d'ajouter les contraintes permettant de les définir de la même manière que les multiPolygones GIS (s'intersectant en un nombre fini de points). Les itérateurs sur des multiPolygones devraient alors absolument être constants afin d'assurer la validité du multiPolygone en tout temps.

5.3 GOGeometrie

La classe GOGeometrie sert à regrouper ensemble diverses géométries. Elle contient donc à la fois un ensemble de points, un ensemble de polylignes et un ensemble de polygones. À part donner accès à ses géométries, la GOGeometrie ne possède aucune fonctionnalité. Elle ne sert qu'à effectuer un regroupement. Ce regroupement est entre autre utile pour le SRChampVector qui contient une seule GOGeometrie au lieu d'avoir à maintenir simultanément un GOMultiPolygone, un GOMultiPolyligne et un GOMultiPoint.

6 SRChampVector

Le SRChampVector est une structure contenant à la fois des polygones, des polylignes et des points. Ce champ est template d'un TTValeur représentant le type des valeurs pouvant être portées par les sommets des géométries qu'il contient.

La section suivante décrit la conception interne du SRChampVector. On y apprend entre autre que le SRChampVector hérite du SRChamp et de la SAAlgèbre. Les sections qui suivent permettent de comprendre comment le SRChampVector agit en tant que champ et en tant qu'algèbre. Finalement, les fonctionnalités propres à ce nouveau type de champ seront introduites.

6.1 Structure d'un SRChampVector

Tout d'abord, le SRChampVector doit pouvoir contenir plusieurs polygones, polylignes et points. Ces derniers sont tous regroupés sous une GOGeometrie. Le SRChampVector contient donc une GOGeometrie servant à représenter toutes ses géométries.

La stratégie permettant d'associer une valeur à chacun des sommets du SRChampVector n'a pas encore été mise au point. Il se peut que la GOGeometrie portée pas le SRChampVector soit plus tard remplacée par un autre type de géométrie, cette fois porteuse de valeurs en ses sommets. Quoi qu'il en soit, toutes les valeurs portées par les sommets d'un SRChampVector devront être du même type. Ce type est défini par le paramètre template TTValeur.

Il peut être intéressant d'interpoler dans un SRChampVector. Par exemple, on peut souhaiter estimer la valeur en un point à l'intérieur d'un polygone. Le SRChampVector

n'associe pas directement de valeur à ce point mais pourrait interpoler pour estimer cette valeur. Cette capacité de fournir une valeur en tout point en fait un champ. Il hérite donc du SRChamp.

On pourrait souhaiter placer ce type de champ en série de manière à obtenir des informations intermédiaires. Pour ce faire, le SRChampVector devra répondre aux conditions imposées par la SAAlgebre dont il hérite.

Les deux sections suivantes permettent de comprendre ce qu'implique le fait que le SRChampVector hérite de la SAAlgebre et du SRChamp.

6.2 Le SRChampVector en tant que SAAlgèbre

Pour pouvoir considérer le SRChampVector comme une algèbre, il doit être possible d'additionner deux SRChampVector, de multiplier ou diviser un SRChampVector par un scalaire et obtenir un autre SRChampVector comme résultat de ces opérations.

Considérons les deux SRChampVector suivants ayant la même région mais portant des géométries différentes.

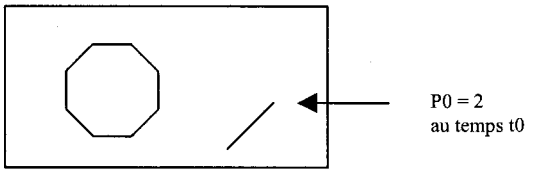


Figure 10 :
S0 représente
l'état au temps t0

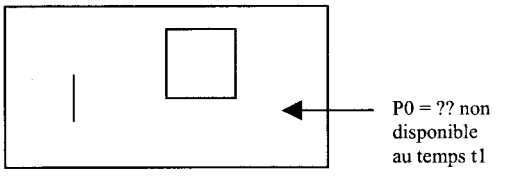


Figure 11 :
S1 représente l'état au
temps t1

On imagine mal comment additionner ces deux champs. Plusieurs confondront l'addition de champ avec l'union de leur géométrie. Voici pourquoi cette façon de faire donne très peu de sens. Supposons que S0 + S1 donne le champ S2 illustré à la figure 12.

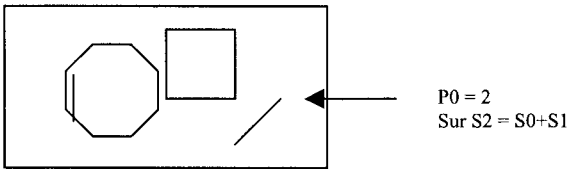


Figure 12 :
S2 = S0 + S1 avec
+ défini comme
l'union géométrique

Imaginons que S0 correspond au temps t0 = 0 alors que S1 représente l'état au temps t1 = 1. Pour trouver le champ au temps t1/2 = 0,5, il faudrait faire (S0+S1)/2, ou plutôt S2 * ½. On en revient à un problème de définition de la multiplication par un scalaire dans un champ. Alors que l'addition correspond à l'union, à quoi peut correspondre la multiplication ?

Une solution serait de multiplier chacune des valeurs des sommets par le scalaire en question. Mais cette opération se combine très mal avec l'opération d'addition (l'union géométrique) choisie. Par exemple, pour multiplier S2 par ½, on multiplierait chacun des sommets de S2 par ½. On obtiendrait donc S3 qui est la moyenne entre S0 et S1. Or ce résultat est tout à fait insensé. Par exemple, on obtient que P0 vaut 1 au temps t1/2 alors qu'on ne sait même pas ce qu'il vaut au temps t1. La définition de l'opération d'addition sur le SRChampVector comme l'opérateur d'union géométrique n'a donc pas de sens.

La condition pour pouvoir additionner deux SRChampVector est que ces derniers aient exactement la même géométrie et portent le même type de valeurs. Cette condition assure une cohérence dans les résultats. Reprenons l'exemple précédent avec, cette fois, S0 et S1 ayant la même géométrie, les valeurs portées étant réelles.

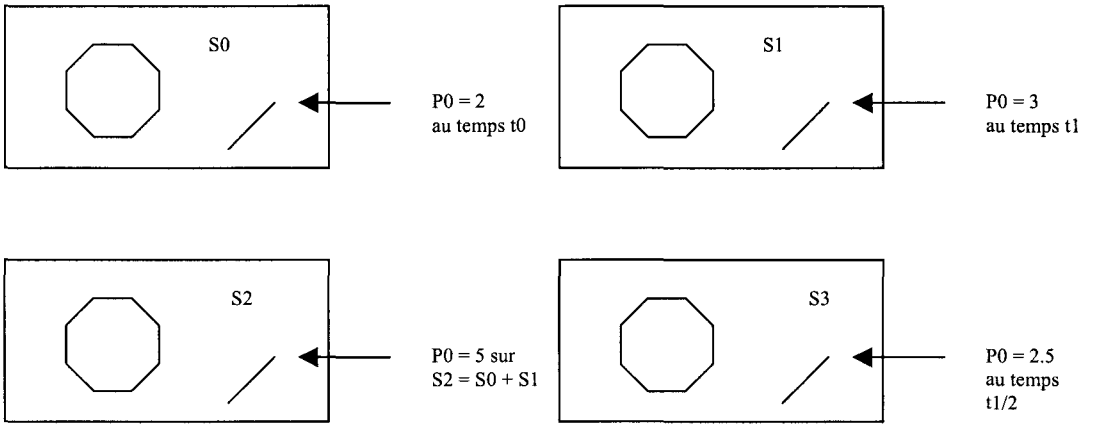


Figure 13 :
S0 représente l'état au temps t0,
S1 représente l'état au temps t1,
S2 = S1 + S0
S3 est un estimé de l'état au temps t1/2

Ainsi, le résultat de l'addition sera un nouveau SRChampVector contenant la même géométrie que les deux autres, mais ayant sur ses sommets des valeurs correspondant à l'addition des valeurs portées par les sommets des deux autres champs. Ainsi, les TTValeurs portées par le champ doivent former un groupe, de manière à être fermées pour l'addition. De cette manière, le résultat de l'addition de deux SRChampVector<TTValeur> sera aussi un SRChampVector<TTValeur>.

Lorsqu'un SRChampVector<TTValeur> est multiplié par un scalaire, ce sont toutes les valeurs portées par les sommets du SRChampVector<TTValeur> qui sont multipliées une à une par ce scalaire. Encore une fois, le résultat doit demeurer une TTValeur afin que le champ résultant soit bien un SRChampVector<TTValeur>. Pour ce faire, le type TTValeur doit non seulement définir un groupe, mais aussi être un espace vectoriel sur **R**. Par exemple, les TTValeurs peuvent être des réels, des vecteurs de réels, des matrices de réels, mais pas des entiers tels qu'on les connaît. Toutefois, si la multiplication d'un réel par un entier est définie et donne un entier, alors les TTValeurs pourront être des entiers.

Les opérations algébriques du champ n'ont pas encore été implantées mais pourront l'être facilement dans le futur à condition que les champs respectent les conditions définies plus haut.

Bref, les SRChampVector peuvent former une algèbre à condition de ne faire interagir entre eux que des champs ayant la même géométrie et portant le même type de valeurs en leurs sommets. Le type des valeurs portées doit répondre à une structure d'espace vectoriel par rapport à **R**.

6.3 Le SRChampVector en tant que SRChamp

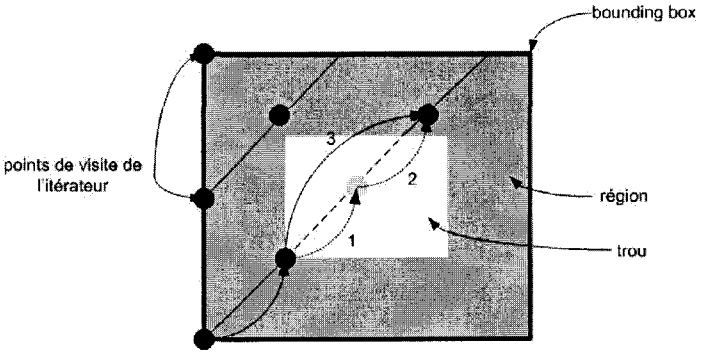
Un SRChamp est une structure capable de calculer une valeur associée à chacun des points de sa région. Le SRChampVector contient donc une région. C'est sur cette dernière que se retrouvent ses géométries. Il ne peut toutefois pas calculer une valeur pour n'importe quel point de la région, mais seulement pour les points se trouvant à l'intérieur d'une des géométries qui le compose. La méthode pointInterieur du SRChamp est donc redéfinie pour retourner « vrai » non pas pour les points se trouvant sur la région, mais pour les points se trouvant sur une des géométries de la région seulement.

C'est la méthode reqValeur qui permet de calculer la valeur associée à un des points d'un champ. Dans le cas d'un SRChampVector, cette méthode *retourne un message d'erreur (à faire)* si le point passé en paramètre ne se trouve pas à l'intérieur d'une des géométries du champ. Dans le cas où le point est valide, une interpolation doit avoir lieu. Le travail d'interpolation n'a pas encore été implanté. Il a toutefois été convenu que ce travail aura lieu au sein d'un algorithme d'interpolation. La méthode reqValeur construira donc un algorithme d'interpolation pour ensuite appeler la méthode principale de cet algorithme. Cette méthode devra trouver à quelle géométrie appartient le point en question et appeler la méthode (toujours dans l'algorithme) permettant d'interpoler sur la géométrie trouvée.

6.4 Itérer sur un SRChampVector

Les champs possèdent une région sur laquelle il est possible d'itérer. Pour ce faire, des classes représentant des itérateurs de région ont été développées dans le passé (voir [9]). Le travail de ces itérateurs consiste à parcourir la région d'un champ, c'est à dire de donner successivement plusieurs points appartenant à cette région.

Une région est un ensemble de polygones. Voici un exemple repris de [9] permettant d'illustrer l'itération sur la région d'un champ :



Dans le cas d'un SRChampEF ou d'un SRChampAnalytique, il est en effet intéressant de parcourir de manière uniforme la région. Or, dans un SRChampVector, l'information est contenue uniquement sur les géométries. Ainsi, les points obtenus lors des itérations ne se trouvant sur aucune géométrie ne sont pas intéressants. Seuls ceux se trouvant sur au moins une géométrie sont pertinents.

Il faudra donc revoir les itérateurs pour adapter le travail au cas particulier du SRChampVector.

Une première approche serait de laisser les itérateurs effectuer leur travail sur la région comme ils le font déjà, mais en éliminant les points ne se trouvant sur aucune géométrie. Or, il est bien possible de parcourir la région d'un SRChampVector ayant de nombreuses géométries sans ne jamais tomber sur ces dernières. Imaginons par exemple un SRChampVector contenant un million de points. Les seuls points valides lors d'une itération sur la région sont ce million de points. Or, considérant qu'il y a une infinité de points dans le champ, même si une itération sur la région tente de nombreux points, il est très peu probable de tomber sur un des points du champ.

Une autre solution serait de faire des itérateurs permettant d'itérer sur l'ensemble des points, puis sur l'ensemble des polylignes et finalement sur l'ensemble des polygones. L'itération sur les polygones ressemblerait à l'itération déjà en place dans les SRChamp; ce serait un scan de leur surface. L'itération sur les polyligne serait plutôt un scan sur 1 dimension alors que l'itération sur les points ne ferait que retourner un point sur x; x étant un certain pas d'itération. Cela permettrait donc de scanner l'ensemble des géométries du SRChampVector.

Peu de réflexions ont eu lieu sur ce sujet. Il s'agit donc d'un dossier toujours ouvert.

6.5 Fonctionnalités propres au SRChampVector

6.5.1 Porter des valeurs

Il faudra être capable d'associer une valeur à chaque point, à chaque sommet de polyligne et à chaque sommet de polyligne de polygone du champ. Ces entités (points et sommets) sont toutes représentables par des coordonnées. Or, sur la même coordonnée peuvent se retrouver les sommets de deux polygones différents et ces deux sommets peuvent porter

des valeurs différentes. Si c'est la tâche du champ de conserver les valeurs, il faudrait, par exemple, que chaque point/sommet de polyligne/sommet de polygone ait un identifiant unique. Le champ pourrait alors associer une valeur à cet identifiant. Idéalement, cet identifiant ne doit pas être supprimé ni modifié lors de l'ajout/la suppression/ le déplacement d'un ou plusieurs points/sommets de polyligne/sommets de polygone.

Une autre manière d'avoir un champ où les géométries portent des valeurs en leurs sommets serait de créer des classes de géométries porteuses de valeurs. L'idéal serait d'arriver à récupérer le code des géométries simples (sans valeurs) pour les géométries porteuses de valeurs. On pourrait imaginer des classes géométriques capables de prendre soit des coordonnées pures, soit des coordonnées couplées avec une valeur et d'effectuer le même travail dans les deux cas.

Pour l'instant, le SRChampVector construit est purement géométrique. C'est à dire qu'aucune valeur n'est associée aux sommets des géométries. Aucune méthode intéressante permettant d'associer une valeur à chacun des sommet n'a encore été trouvée. C'est donc un sujet qui n'est pas clos.

6.5.2 Fonctionnalités géométriques

L'utilisation du SRChampVector dans un contexte d'édition a amené la nécessité d'implanter de nombreuses fonctionnalités permettant de faire des opérations sur les géométries du champ. Il s'agit en fait de fonctionnalités implantant les opérations géométriques définies en 5.1.3. Étant utilisé dans un contexte d'édition, le SRChampVector se trouve forcé de définir toutes ces opérations. En fait, il ne peut pas tout simplement donner sa géométrie et ne faire que des opérations de champ (retourner la valeur en un point), car cette géométrie est intimement liée à des valeurs portées par les sommets. Toute modification géométrique doit donc être accompagnée d'une mise à jour par le champ des valeurs aux sommets.

6.5.3 Itération

Le champ fournit aussi des itérateurs sur ses géométries. Il ne faut pas confondre les itérateurs sur les géométries avec les itérateurs dont il est question en 6.4. Alors qu'il était question en 6.4 de scanner l'intérieur des géométries, il est maintenant question de pouvoir parcourir, une à une les géométries du champ.

Les fonctions reqMultiXXIterateur retournent un GOMultiXXConstIterateur qui pointe sur un objet de type XX. Par exemple, reqMultiPolygneIterateur() permet d'obtenir un itérateur sur le multipolygone du SRChampVector. Lorsqu'on déréférence cet itérateur, on obtient un des polygones du multipolygone.

Bien entendu, cet itérateur ne donne l'accès aux structures qu'en lecture seulement car, les modifications doivent passer par le SRChampVector afin de s'assurer que les valeurs portées par les sommets sont mises à jour.

7 Phase de test

Une phase de test a eu lieu pour certaines géométries. En particulier des fichiers de test ont été crées pour les classes GOPoint, GOPolyligne et GOPolygone.

Cette phase fut très pratique. Elle permit en particulier de découvrir certaines particularités discutées en 3.2 et en 5.1.5.2. Pour l'instant, ces tests ne sont pas sur CVS. Les classes de géométries sont toujours en développement. L'entretien et la mise à jour de ces tests lors des modifications aux géométries est toujours assez long. Pour l'instant, la version test et la version des géométries ne correspondent pas. *Cette situation devrait être rectifiée.*

8 Conclusion

8.1 Résumé

En résumé ce rapport couvre deux principaux volets, soient la conception des classes de géométries et celle du SRChampVector.

Les géométries de bases devant être implantées ont d'abord été définies. Par la suite, une analyse des besoins face aux géométries a eu lieu. Elle a permis de faire ressortir que les géométries devaient être modifiables (ajout de sommets, déplacement) et assez flexibles pour permettre la création d'un SRChampVector où des valeurs seraient associées aux sommets des géométries. Finalement, les géométries doivent permettre d'exécuter certains algorithmes comme l'intersection et doivent assurer leur validité.

C'est TerraLib qui a été choisie comme librairie pour l'implantation des géométries. Les classes de GO sont donc un proxy entre le code de Modeleur et celui de TerraLib. Avant d'entrer dans la conception des GO, certaines particularités de la librairie TerraLib sont décrites. Cela permet, de mieux comprendre les choix ayant été faits lors de la conception des GO, comme celui de redéfinir le constructeur de copie.

Finalement, le SRChampVector est décrit. On peut remarquer que, dans un contexte d'édition, le SRChampVector est très pratique. Sous leur format actuel, les classes de géométries, permettent de bien faire le travail. Par contre, lorsqu'on en arrive plus particulièrement au travail précis du champ, comme l'interpolation, le scan des géométries ou l'association des valeurs aux sommets, le développement est beaucoup moins avancé. Bref, il y a encore du travail à faire sur ce dossier.

8.2 Perspectives futures

Ce rapport dresse un portrait du statut actuel des classes conçues tout en recensant les problématiques n'ayant pas encore été résolues.

Il a été clairement décrit qu'on devait pouvoir faire des intersections et des unions sur des polygones. TerraLib n'implante pas de telles fonctionnalités. Il serait assez lourd de réimplanter ces fonctionnalités, et, quoi que plausible, l'idée d'utiliser une autre librairie pour faire ce travail n'est pas très intéressante puisque cela irait alourdir

considérablement le travail d'un futur programmeur voulant s'initier aux classes de géométries. Il semble donc que le choix de TerraLib comme librairie géométrique mène à un cul de sac à ce sujet.

De même TerraLib ne semble pas offrir une bonne flexibilité permettant de faire porter des valeurs aux sommets des géométries. En effet, le lien avec le template est brisé au niveau des classes TeLine2D et TePolygon ce qui vient figer ces deux dernières. Le type de coordonnées qu'elles portent semble difficile à altérer. En conséquence, le SRChampVector ne porte toujours pas de valeur sur ses sommets.

Finalement TerraLib ne permet pas de vérifier la validité d'un polygone. Ces trois faiblesses dans TerraLib ont conduit à un questionnement au sujet de la pertinence du choix de cette librairie comme outil géométrique. Une autre librairie, GEOS, a été analysée et s'est avérée plus complète sur certains domaines. Elle permet entre autre d'effectuer des intersections entre les polygones et de vérifier la validité d'un polygone. Aussi, les coordonnées portées par les sommets des polygones et polylignes semblent pouvoir être spécialisées. Mais GEOS a une faiblesse considérable, elle ne permet pas de modifier une structure existante. Il est donc impossible d'ajouter ou de retirer un sommet à une polyligne ou un polygone existant sans avoir à reconstruire toute la structure.

Bref, les géométries ne répondant pas bien aux besoins ayant été définis, le SRChampVector se retrouve devant une impasse. Il faudra donc revoir une partie de la conception des géométries pour qu'elles puissent bien remplir le rôle qu'on attend d'elles.

9 Annexes

9.1 Ajout d'une polyligne à un polygone dans TerraLib

Soit PG0 un polygone et PL0 une polyligne. Nommons BoxGone0 la box du polygone et BoxLigne0 celle de la polyligne. Lors de l'ajout de PL0 à PG0, le constructeur de copie de polyligne est appelé. C'est donc PL0', une copie de PL0 qui est ajoutée au polygone. Il est important de noter que, par la nature particulière du constructeur de copie, bien que PL0' ait sa propre box, elle partage la même liste de sommets que PL0. Suite à l'ajout de PL0 à PG0, la box de PG0 est mise à jour. Or, toute modification ultérieure aux sommets de PL0 entraînera une modification aux sommets de PL0' sans toutefois mettre à jour sa box, et encore moins celle du polygone PG0. Ainsi, la cohérence de la box n'est pas gardée. Pour assurer la cohérence, une polyligne ajoutée à un polygone ne doit pas être modifiée par la suite.

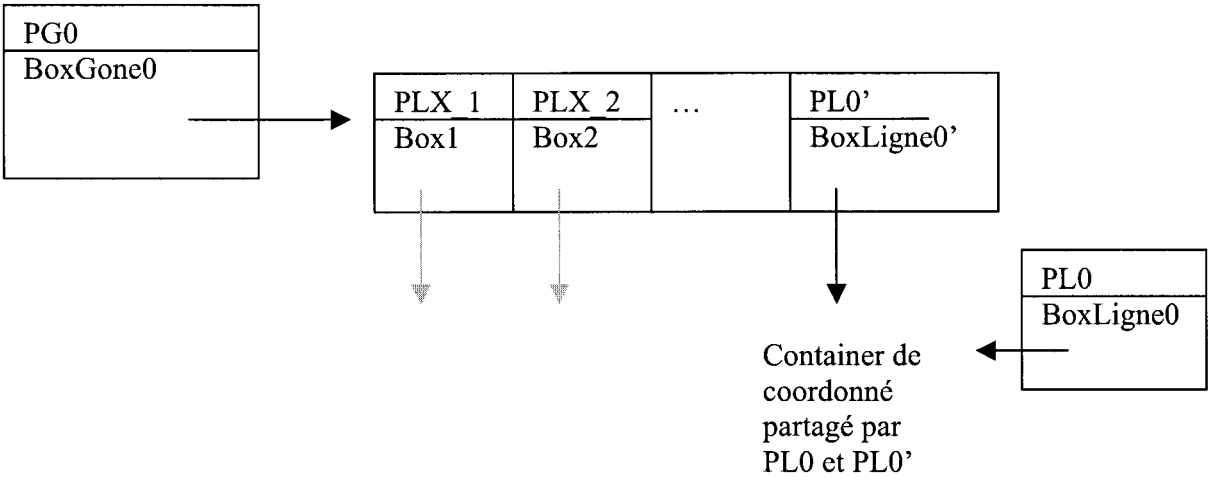


Figure 15 : Une copie de PL0, soit PL0' est ajoutée à PG0. Un ajout d'une coordonnée à PL0 entraînera une mise à jour de BoxLigne0 de PL0 mais pas de mise à jour de la box BoxLigne0' de PL0' ni de BoxGone0 de PG0.

9.2 Intérieur, frontière et extérieur des géométries TerraLib

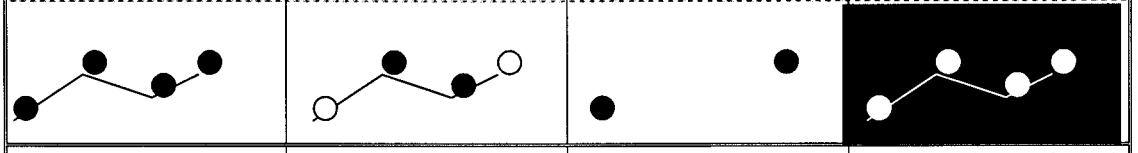
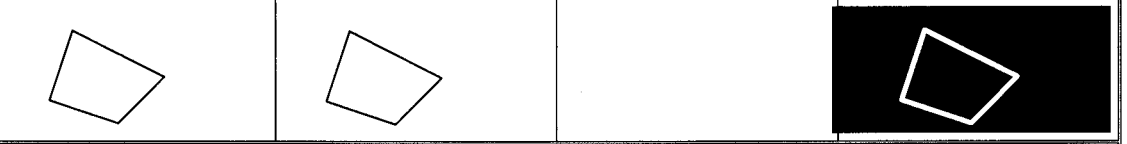
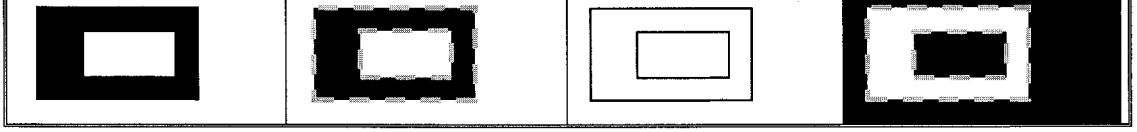
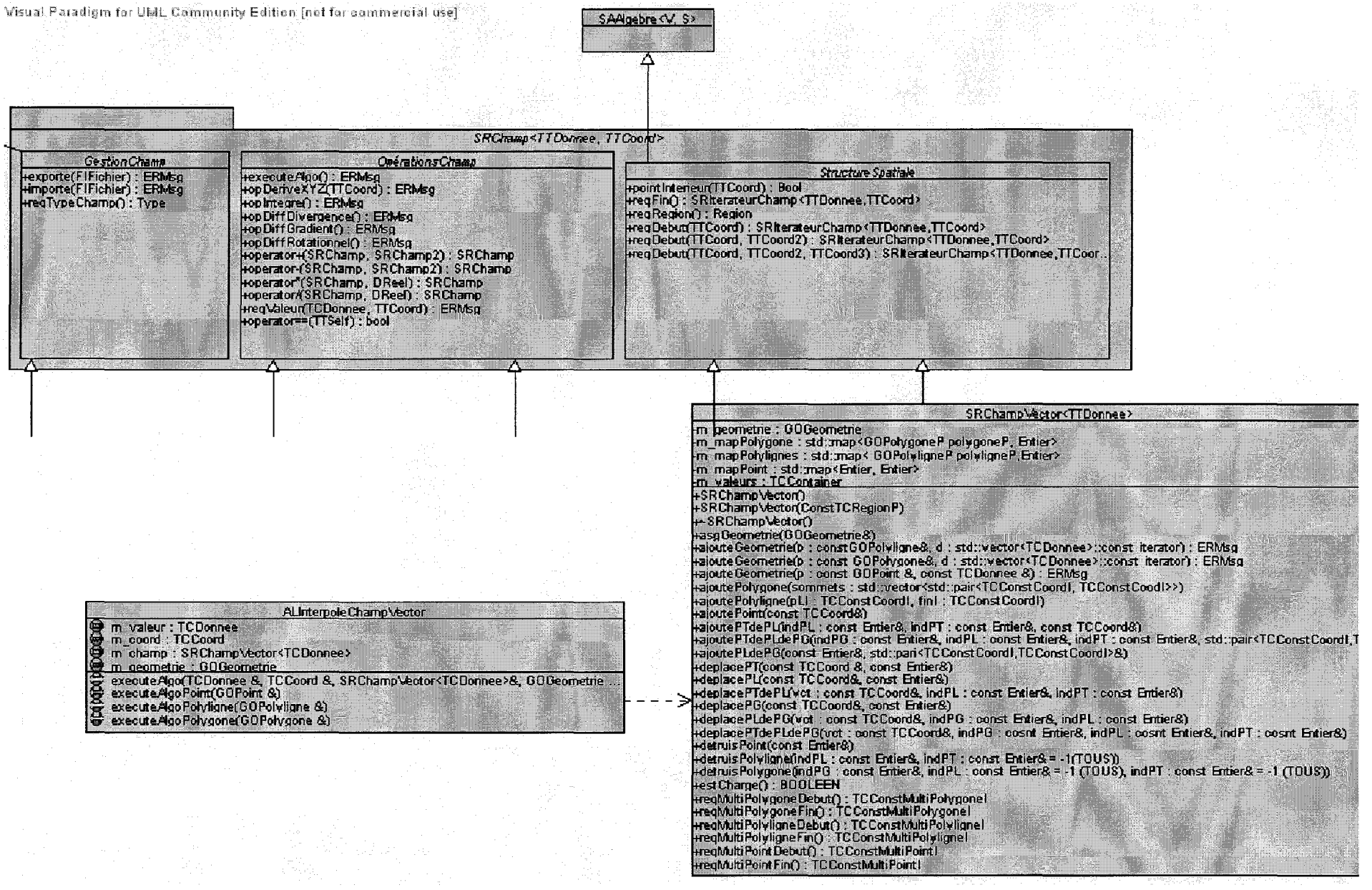
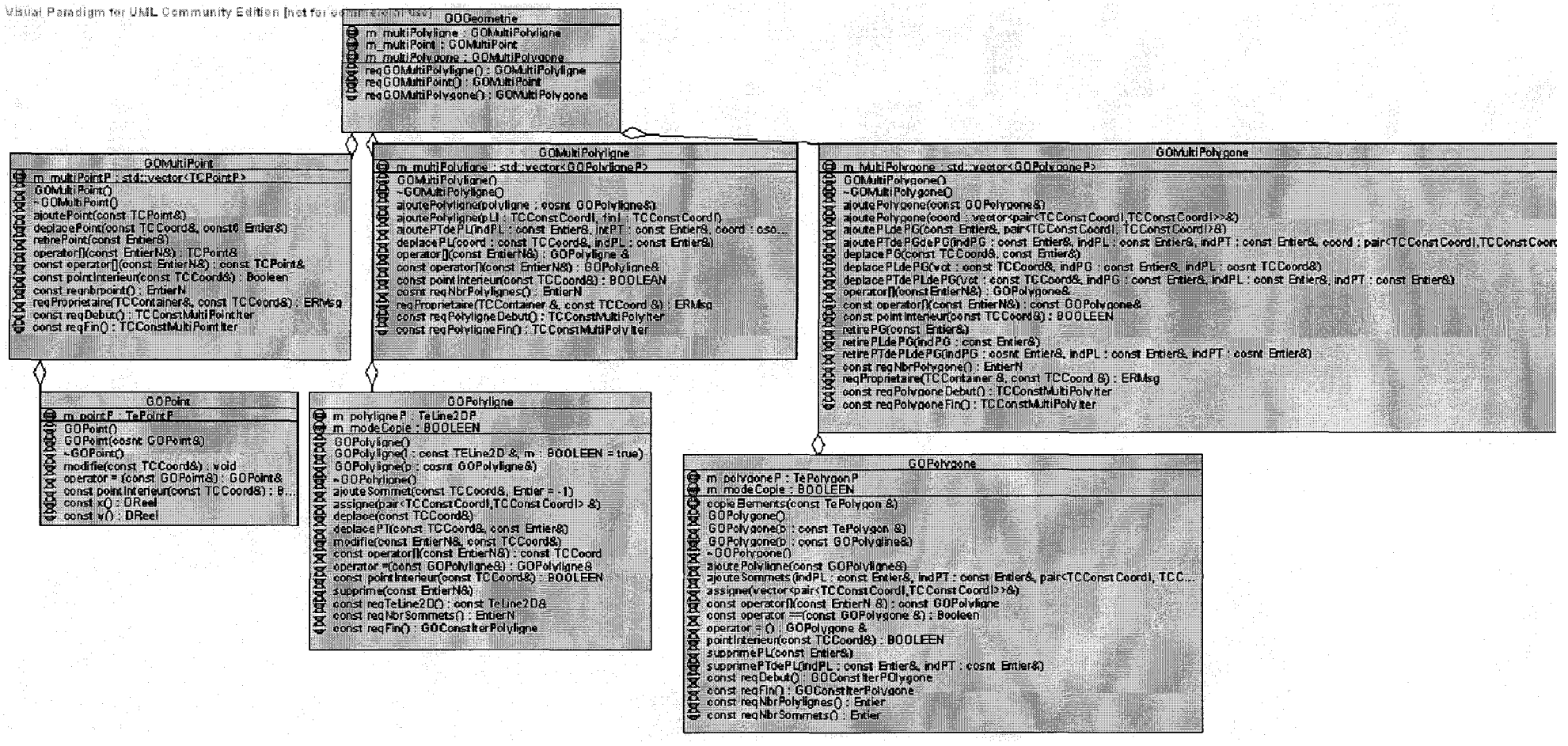
GÉOMÉTRIE	INTERIEUR	FRONTIÈRE	EXTERIEUR
TePoint	Le point même	Vide	Tout sauf ce qui est à l'intérieur ou sur les frontières
			
TeLine2D	Tous les points de la ligne sauf la frontière	Le premier et dernier point	Tout sauf ce qui est à l'intérieur ou sur les frontières
			
TeLinearRing	Tous les points	Vide	Tout sauf ce qui est à l'intérieur ou sur les frontières
			
TePolygon	Tout ce qui est entre les frontières	Toutes les frontières (polylignes)	Tout sauf ce qui est à l'intérieur ou sur les frontières
			

Diagramme de classe : SRChampVector


Visual Paradigm for UML Community Edition [not for commercial use]
SAAlgebre<V, S>
SRChamp<TTDonnee, TTCoord>
GestionChamp
+exporte(FIFichier) : ERMsg
+importe(FIFichier) : ERMsg
+reqTypeChamp() : Type
OpérationsChamp
+executeAlgo() : ERMsg
+opDeriveXYZ(TTCoord) : ERMsg
+opIntegre() : ERMsg
+opDiffDivergence() : ERMsg
+opDiffGradient() : ERMsg
+opDiffRotationnel() : ERMsg
+operator+(SRChamp, SRChamp2) : SRChamp
+operator-(SRChamp, SRChamp2) : SRChamp
+operator*(SRChamp, DReel) : SRChamp
+operator/(SRChamp, DReel) : SRChamp
+reqValeur(TCDonnee, TTCoord) : ERMsg
+operator==(TTSelf) : bool
StructureSpatiale
+pointInterieur(TTCoord) : Bool
+reqFin() : SRIterateurChamp<TTDonnee,TTCoord>
+reqRegion() : Region
+reqDebut(TTCoord) : SRIterateurChamp<TTDonnee,TTCoord>
+reqDebut(TTCoord, TTCoord2) : SRIterateurChamp<TTDonnee,TTCoord>
+reqDebut(TTCoord, TTCoord2, TTCoord3) : SRIterateurChamp<TTDonnee,TTCoor...
SRChamp\Vector<TTDonnee>
-m_geometrie : GOGeometrie
-m_mapPolygone : std::map<GOPolygoneP polygoneP, Entier>
-m_mapPolylignes : std::map< GOPolyligneP polyligneP,Entier>
-m_mapPoint : std::map<Entier, Entier>
-m_valeurs : TCContainer
+SRChamp\Vecton()
+SRChamp\Vecton(ConstTCRegionP)
+~SRChamp\Vector()
+asgGeometrie(GOGeometrie&)
+ajouteGeometrie(p : constGOPolyligne&, d : std::vector<TCDonnee>::const iterator) : ERMsg
+ajouteGeometrie(p : const GOPolygone&, d : std::vector<TCDonnee>::const iterator) : ERMsg
+ajouteGeometrie(p : const GOPoint &, const TCDonnee &) : ERMsg
+ajoutePolygone(sommets : std::vector<std::pair<TCConstCoordI, TCConstCoodI>>)
+ajoutePolyligne(pLI : TCConstCoordI, finI : TCConstCoordI)
+ajoutePoint(const TCCoord&)
+ajoutePTdePL(indPL : const Entier&, indPT : const Entier&, const TCCoord&)
+ajoutePTdePLdePG(indPG : const Entier&, indPL : const Entier&, indPT : const Entier&, std::pair<TCConstCoordI,T
+ajoutePLdePG(const Entier&, std::pari<TCConstCoordI,TCConstCoordI>&)
+deplacePT(const TCCoord &, const Entier&)
+deplacePL(const TCCoord&, const Entier&)
+deplacePTdePL(vct : const TCCoord&, indPL : const Entier&, indPT : const Entier&)
+deplacePG(const TCCoord&, const Entier&)
+deplacePLdePG(vct : const TCCoord&, indPG : const Entier&, indPL : const Entier&)
+deplacePTdePLdePG(vct : const TCCoord&, indPG : cosnt Entier&, indPL : cosnt Entier&, indPT : cosnt Entier&)
+detruisPoint(const Entier&)
+detruisPolyligne(indPL : const Entier&, indPT : const Entier& = -1(TOUS))
+detruisPolygone(indPG : const Entier&, indPL : const Entier& = -1 (TOUS), indPT : const Entier& = -1 (TOUS))
+estChange() : BOOLEEN
+reqMultiPolygoneDebut() : TCConstMultiPolygoneI
+reqMultiPolygoneFin() : TCConstMultiPolygoneI
+reqMultiPolyligneDebut() : TCConstMultiPolyligneI
+reqMultiPolyligneFin() : TCConstMultiPolyligneI
+reqMultiPointDebut() : TCConstMultiPointI
+reqMultiPointFin() : TCConstMultiPointI
ALInterpoleChamp\Vector
m_valeur : TCDonnee
m_coord : TCCoord
m_champ : SRChamp\Vector<TCDonnee>
m_geometrie : GOGeometrie
executeAlgo(TCDonnee &, TCCoord &, SRChamp\Vector<TCDonnee>&, GOGeometrie ...
executeAlgoPoint(GOPoint &)
executeAlgoPolyligne(GOPolyligne &)
executeAlgoPolygone(GOPolygone &)

9.3 Diagramme de classe : GOGeometrie, GOPolyligne, GOPolygone, GOPoint, GOMultiPolyligne, GOMultiPolygone et GOMultiPoint


Visual Paradigm for UML Community Edition [not for commercial use]
GOGeometrie
m_multiPolyligne : GOMultiPolyligne
m_multiPoint : GOMultiPoint
m_multiPolygone : GOMultiPolygone
reqGOMultiPolyligne() : GOMultiPolyligne
reqGOMultiPoint() : GOMultiPoint
reqGOMultiPolygone() : GOMultiPolygone
GOMultiPoint
m_multiPointP : std::vector<TCPointP>
GOMultiPoint()
~GOMultiPoint()
ajoutePoint(const TCPoint&)
deplacePoint(const TCCoord&, const Entier&)
retirePoint(const Entier&)
operator[](const EntierN&) : TCPoint&
const operator[](const EntierN&) : const TCPoint&
const pointInterieur(const TCCoord&) : Booleen
const reqnbrpoint() : EntierN
reqProprietaire(TCContainer&, const TCCoord&) : ERMsg
const reqDebut() : TCConstMultiPointIter
const reqFin() : TCConstMultiPointIter
GOMultiPolyligne
m_multiPolyligne : std::vector<GOPolyligneP>
GOMultiPolyligne()
~GOMultiPolyligne()
ajoutePolyligne(polyligne : const GOPolyligne&)
ajoutePolyligne(pLI : TCConstCoordI, finI : TCConstCoordI)
ajoutePTdePLI(indPL : const Entier&, intPT : const Entier&, coord : cso...
deplacePLI(coord : const TCCoord&, indPL : const Entier&)
operator[](const EntierN&) : GOPolyligne &
const operator[](const EntierN&) : GOPolyligne&
const pointInterieur(const TCCoord&) : BOOLEAN
const reqNbrPolylignes() : EntierN
reqProprietaire(TCContainer &, const TCCoord &) : ERMsg
const reqPolyligneDebut() : TCConstMultiPolyIter
const reqPolyligneFin() : TCConstMultiPolyIter
GOMultiPolygone
m_multiPolygone : std::vector<GOPolygoneP>
GOMultiPolygone()
~GOMultiPolygone()
ajoutePolygone(const GOPolygone&)
ajoutePolygone(coord : vector<pair<TCConstCoordI,TCConstCoordI>>&)
ajoutePLdePG(const Entier&, pair<TCConstCoordI, TCConstCoordI>&)
ajoutePTdePGdePG(indPG : const Entier&, indPL : const Entier&, indPT : const Entier&, coord : pair<TCConstCoordI,TCConstCoor
deplacePG(const TCCoord&, const Entier&)
deplacePLdePG(vot : const TCCoord&, indPG : const Entier&, indPL : cosnt TCCoord&)
deplacePTdePLdePG(vot : const TCCoord&, indPG : const Entier&, indPL : const Entier&, indPT : const Entier&)
operator[](const EntierN&) : GOPolygone&
const operator[](const EntierN&) : const GOPolygone&
const pointInterieur(const TCCoord&) : BOOLEEN
retirePG(const Entier&)
retirePLdePG(indPG : const Entier&)
retirePTdePLdePG(indPG : cosnt Entier&, indPL : const Entier&, indPT : cosnt Entier&)
const reqNbrPolygone() : EntierN
reqProprietaire(TCContainer &, const TCCoord &) : ERMsg
const reqPolygoneDebut() : TCConstMultiPolyIter
const reqPolygoneFin() : TCConstMultiPolyIter
GOPoint
m_pointP : TePointP
GOPoint()
GOPoint(cosnt GOPoint&)
~GOPoint()
modifie(const TCCoord&) : void
operator = (const GOPoint&) : GOPoint&
const pointInterieur(const TCCoord&) : B...
const xQ : DReel
const yQ : DReel
GOPolyligne
m_polyligneP : TeLine2DP
m_modeCopie : BOOLEEN
GOPolyligne()
GOPolyligne(: const TELine2D &, m : BOOLEEN = true)
GOPolyligne(p : cosnt GOPolyligne&)
~GOPolyligne()
ajouteSommet(const TCCoord&, Entier = -1)
assigne(pair<TCConstCoordI,TCConstCoordI> &)
deplace(const TCCoord&)
deplacePTI(const TCCoord&, const Entier&)
modifie(const EntierN&, const TCCoord&)
const operator[](const EntierN&) : const TCCoord
operator =(const GOPolyligne&) : GOPolyligne&
const pointInterieur(const TCCoord&) : BOOLEEN
supprime(const EntierN&)
const reqTeLine2D() : const TeLine2D&
const reqNbrSommets() : EntierN
const reqFin() : GOConstIterPolyligne
GOPolygone
m_polygoneP : TePolygonP
m_modeCopie : BOOLEEN
copieElements(const TePolygon &)
GOPolygone()
GOPolygone(p : const TePolygon &)
GOPolygone(p : const GOPolygone&)
~GOPolygone()
ajoutePolyligne(const GOPolyligne&)
ajouteSommets(indPL : const Entier&, indPT : const Entier&, pair<TCConstCoordI, TCC...
assigne(vector<pair<TCConstCoordI,TCConstCoordI>>&)
const operator[](const EntierN &) : const GOPolyligne
const operator ==(const GOPolygone &) : Booleen
operator = () : GOPolygone &
pointInterieur(const TCCoord&) : BOOLEEN
supprimePL(const Entier&)
supprimePTdePL(indPL : const Entier&, indPT : cosnt Entier&)
const reqDebut() : GOConstIterPOlygone
const reqFin() : GOConstIterPolygone
const reqNbrPolylignes() : Entier
const reqNbrSommets() : Entier

9.4 Diagramme de classe : itérateurs sur les GO : à venir

RAPPORT # 3 : Modèle Numérique de Terrain (MNT)

Rapport final

Modèle numérique de terrain (MNT)

Présenté par :

Cédric Caron

18 janvier 2005

Versions du document

Date	Auteur	Commentaires	Version
23/03/2004	Cédric Caron	Version initiale	1.00
13/04/2004	Cédric Caron	Révision	1.01
15/04/2004	Cédric Caron	Révision	1.02
23/04/2004	Cédric Caron	Révision	1.03

Table des matières

1 Introduction

1.1 Objectifs

Le but de ce rapport est de présenter les différents composants de l'activité MNT de Modeleur 2.0 et de discuter du développement de ceux-ci. Le rapport résume le travail d'analyse, de design et d'implantation effectué jusqu'ici et tente de déterminer les prochaines étapes du développement de l'activité.

1.2 Références

- Spécifications – Modeleur 2.0. Québec, INRS-Eau, Terre & Environnement. 68 pages.
- Rapports de réunions – Réunion 1 à 5 sur le MNT, Québec, INRS-Eau, Terre & Environnement.
- Rapport de réunion – Mise en commun MNT et semis de points, Québec, INRS-Eau, Terre & Environnement. 18 pages.
- Rapport de réunion – Rapport de réunion 1 et 2 sur l'Éditeur, Québec, INRS-Eau, Terre & Environnement. 5 pages.

2 Description de l'activité MNT

Le MNT (Modèle Numérique de Terrain) est une activité dans le cadre de Modeleur 2.0. Celle-ci permet de faire la description de chacun des aspects du terrain sur lequel l'usager désire bâtir son étude. Le MNT est construit de partitions (partitions de topographie, de vent, de glace, etc.) qui sont utilisées pour spécifier les jeux de données à utiliser à chaque section du domaine. De la sorte, il est possible de construire un modèle numérique de terrain qui spécifie les données que l'on voudra éventuellement porter vers un maillage.

Dans la première version de Modeleur, l'essentiel du concept de l'outil fut développé et mis en oeuvre. Certaines fonctionnalités ont été revues afin d'améliorer la convivialité du logiciel. La principale fonctionnalité qui a été modifiée a trait à la construction de la partition. Dans la nouvelle version, une partition est constituée à l'aide d'un assemblage de différentes couches de données. Ces dernières pourront être associées à des jeux de données différents et possiblement concurrents.

Les couches de données pourront ainsi être superposées en certains endroits du domaine de validité introduisant un concept de priorité (défini par l'usager) entre celles-ci pour lever les ambiguïtés dans les zones partagées. Cela permettra de raffiner l'étude à certains endroits, sur des sections particulières définies sur la partition sans avoir à définir un nombre élevé de sous domaines inter reliés comme ce l'était le cas dans Modeleur 1.0.

2.1 Les partitions

Un MNT contient à priori un nombre non déterminé de partition car c'est à l'utilisation qu'est déterminé de quoi sera construit ce MNT.

Il existe deux ensembles de partitions; les partitions spécialisées et les partitions génériques.

2.1.1 Les partitions spécialisées

Les types de partitions spécialisées sont les suivants: topographie, substrat, glace, vent, frottement de Manning. Il important de noter que cette liste pourrait s'allonger.

La gestion des partitions spécialisées est basée sur un principe de superposition de couches de données en fonction de leur priorité. Cela est expliqué au point 2.2.

2.1.2 Les partitions génériques

Comme son nom l'indique, une partition générique n'est pas associée à un aspect déterminé d'analyse. Elle sert plutôt à porter de l'information utilisateur. Cette information sera interpolée si l'information est portée aux sommets des sous-domaines et ne le sera pas si l'information est portée par les sous-domaines en tant que tel. À la création de la partition, on doit spécifier si l'information sera portée aux sommets ou par

la surface des sous-domaines. On doit aussi définir le type d'information que la partition hébergera. L'information portée par la partition générique doit être définie en spécifiant une structure de données. De plus, si on a spécifié que l'information est portée aux sommets, on doit définir les règles d'interpolation de la structure de données en question.

2.2 Les couche de données

Une couche de données est constituée d'un ou plusieurs sous-domaines qui se voient assignés un jeu de données commun. Le ou les sous-domaines d'une couche doivent obligatoirement se trouver dans les limites de validité du jeu de données.

2.2.1 Concept de priorité de couche

On doit accorder une priorité aux couches de données. Plus la couche de données possède une priorité élevée, plus elle se retrouve à l'avant-plan dans l'affichage de la partition. Les jeux de données associés aux couches qui se retrouvent à l'avant plan sont ceux qui sont utilisés lorsque l'information est portée sur un maillage. Si deux couches de données se superposent en certains endroits, ce sont les données de la couche la plus prioritaire qui sont privilégiées pour ces endroits.

2.3 Le squelette du MNT

Un concept de zone commune à toutes les partitions d'un MNT, appelé « squelette du MNT », sera mis en oeuvre. Celui-ci pourra servir lors de la construction d'un maillage afin de s'assurer que le maillage ne dépasse pas la zone des données partagées.

2.4 Interaction usager – MNT

Voici une section qui introduit l'interaction entre l'usager et l'outils de gestion de MNT.

2.4.1 Création, ouverture d'un MNT

À la création ou à l'ouverture du MNT, l'usager se voit présenter une fenêtre contenant le gestionnaire de partitions avec une arborescence des partitions contenues dans ce MNT (voir Figure 1). On peut y voir toutes les partitions du MNT, et ainsi en ajouter, en enlever, en enregistrer.

Il y a également une vue avec onglets qui permet de visiter chacune des partitions du MNT. L'usager peut donc modifier chacune de ses partitions et naviguer d'une partition à l'autre. Quand il change d'onglet, les changements faits à la partition en cours sont gardés en mémoire, mais ne sont pas sauvegardés.

2.4.2 Modification et sauvegarde d'un MNT

Lorsque l'usager a créé ou a ouvert un MNT, il peut y ajouter une partition. L'usager doit spécifier le type de partition qu'il désire ajouter. Un même type de partition peut être

présent plusieurs fois dans un MNT. L'ajout d'une partition a pour effet d'insérer un nouvel onglet et un nouvel item dans l'arborescence du MNT.

L'usager peut aussi ajouter une couche de données à une partition. Cette couche devra être du même type que la partition à laquelle elle est ajoutée. L'ajout d'une couche a pour effet d'insérer un nouvel item d'arborescence dans l'arbre de la partition. Symétriquement, il est possible de retirer toutes les composantes du MNT.

Finalement, il est possible de sauvegarder individuellement chacune des entités d'un MNT une à la fois et de façon distincte. Il est aussi possible de sauvegarder tout le MNT et toutes ces composante du même coup.

2.4.3 Navigation et changement de focus

Voici un aperçu de l'interface utilisateur proposée à l'usager :

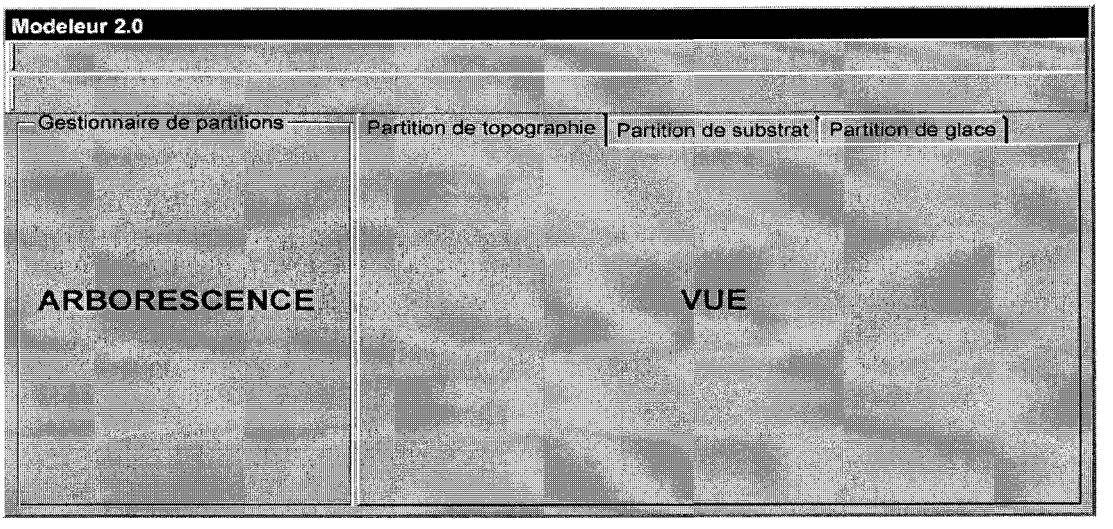


Figure 1-Sections de l'interface du MNT

Pour naviguer à l'intérieur d'un MNT, l'usager doit cliquer sur les différents items de l'arborescence de ce dernier. Ainsi, il déplace le focus du système et peut consulter le MNT à différents niveaux.

L'action d'effectuer un double-clic sur une couche ou une partition dans l'arborescence a pour effet de changer le focus dans le MNT et, par le fait même, de changer le contenu affiché dans la fenêtre « vue ».

Si le focus est dirigé vers une partition, le contenu de cette dernière est affichée. C'est à dire que toutes ses couches sont superposées l'une sur l'autre en ordre de priorité croissante. Si l'usager n'est pas satisfait du résultat obtenu, il peut changer l'ordre de priorité des couches, ajouter une nouvelle couche ou modifier une couche existante. Lorsque le focus est dirigé vers une couche, cette dernière est affichée en premier plan dans la fenêtre « vue » et le système entre en « mode édition de couche». Cela permet de modifier la couche à l'aide d'outils d'édition de sous-domaines.

3 Analyse et état actuel

3.1 Les grands acteurs et leurs rôles

Les cinq modules qui ont un rôle à jouer dans la mise en œuvre de l'activité MNT sont le module MNT, la partie GUI du MNT en python, le module visualisation (VTK), le module gestionnaire BD et le module éditeur. Voici une brève description qui fait ressortir le rôle de chacun d'entre eux au sein de l'activité.

3.1.1 Le module MNT

Le module MNT regroupe toutes les classes mettant en œuvre les structures de données et les fonctionnalités propres au MNT. En plus de contenir une classe gestionnaire connectée au bus d'événement, ce module met en œuvre les structures fonctionnelles du MNT, des partitions et des couches de données.

Le module met en œuvre la construction de couches d'affichage VTK. Cela a pour le but de gérer l'affichage statique des couches de données des partitions.

Le module comprend une classe gestionnaire, une classe d'algorithme, des classes informationnelles et des classes conteneurs de données.

3.1.2 Le GUI du MNT

Pour supporter les interactions entre l'usager et le gestionnaire MNT, on fait appel à une interface graphique(GUI) codée en python. L'interface graphique du MNT, consiste en une fenêtre composée d'une arborescence et d'une série d'onglets hébergeant l'affichage de chacune des partitions du MNT. La Figure 1 expose l'allure globale de cette interface graphique.

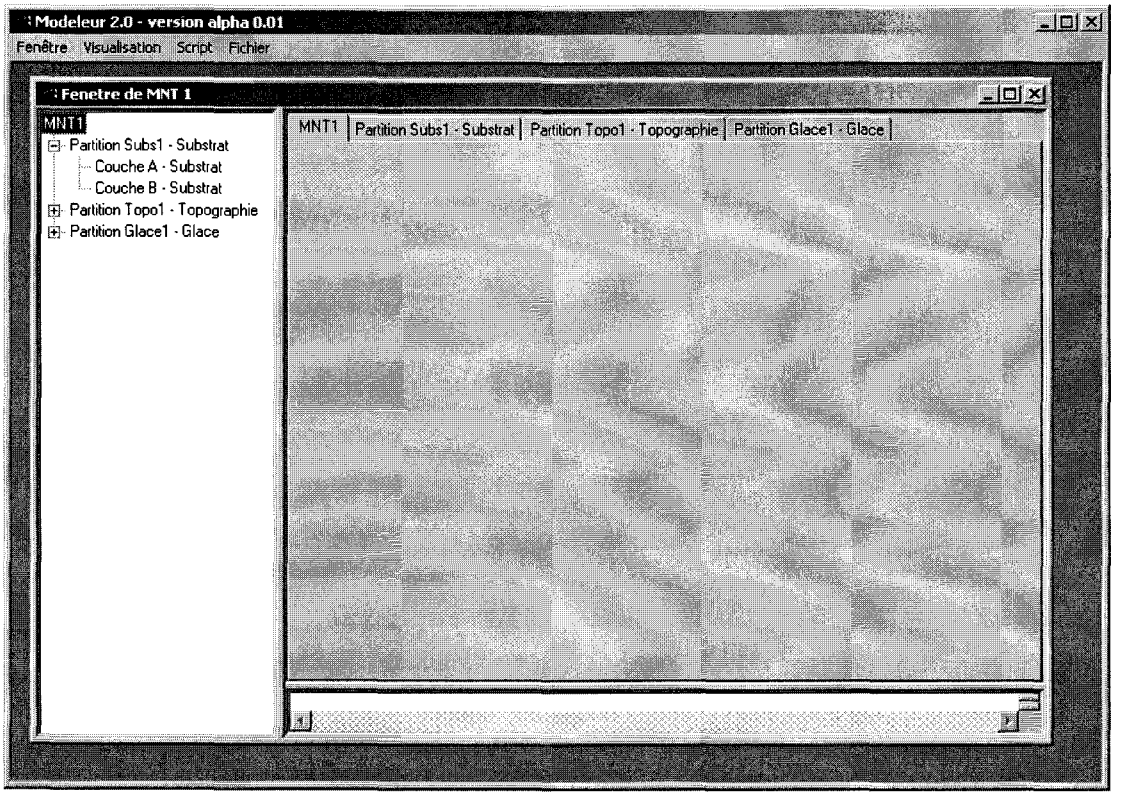


Figure 2- Allure actuelle de l'interface graphique du MNT (GUI)

La fenêtre permet la gestion d'un seul MNT à la fois. Ces fenêtres sont générées à l'ouverture ou à la création d'un MNT.

Le GUI est un module Python qui est étroitement lié au module MNT. Son rôle est de faire le pont entre le GUI de Modeleur, l'usager et le gestionnaire du module MNT.

3.1.3 Le module de visualisation VTK

Le module de visualisation VTK est utilisé dans les premiers prototypes de Modeleur 2.0 pour afficher des nœuds et des maillages.

Ce module sert également pour l'activité MNT afin d'afficher de façon statique toutes les couches d'une partition dans l'espace défini par les onglets.

La partie de travail faite par le module de visualisation sera minimale. C'est à dire que celui-ci ne devra s'occuper que de la mise à jour de l'affichage à la réception d'événements allant dans ce sens.

Comme il a été explique au point 3.1.1, la responsabilité de construire les couches d'affichage VTK reviendra au module gestionnaire de MNT.

3.1.4 Le module gestionnaire bases de données

Le gestionnaire de base de données est un module actuellement implanté dans les premiers prototypes de Modeleur 2.0. Celui-ci s'occupe de tous ce qui a trait aux échanges et au maintient des structures de données mémoires de Modeleur 2.0.

Quelques ajouts à ce module ont donc été nécessaires afin de gérer les nouvelles structures de données MNT qui sont stockées dans les tables la base de données.

3.1.4.1 Mode référence et mode copie

Il existe deux modes d'opération possible qui pourraient définir la façon dont les données seront gérées : un mode référence et un mode copie.

- **Le mode référence :** permettrait d'avoir une référence sur la structure de données en mémoire, et de pouvoir la modifier directement (par exemple, si une même partition appartient à plusieurs MNT et qu'elle est modifiée dans un de ces MNT, elle le sera aussi dans tous les autres);
- **Le mode copie :** prendrait une copie de l'objet pour chacune de ses associations (par exemple, l'ajout d'une partition existante à un MNT serait effectué en faisant une copie complètement indépendante de cette partition).

Il faut noter que le choix de l'un de ces deux modes d'opération reste toujours en suspend. La question est de définir quel mode d'opération sera utilisé pour Modeleur 2.0. Le mode « copie » ou le mode « référence » . Il faut réfléchir à savoir lequel des deux est plus avantageux puisqu'il est pratiquement impensable de laisser le choix à l'utilisateur. Ces deux modes d'opération sont définis plus en détails dans le document *Spécifications - Modeleur 2.0* .

Le module MNT maintient des pointeurs à ses structures en mémoire. De là, le design du module MNT ne repose pas sur la gestion de ces données qui sont maintenues à l'extérieur de celui-ci (du côté de la BD). En conséquence, la décision sur le choix du mode copie ou du mode référence n'a pas d'impacts majeurs pour la poursuite du projet en ce qui concerne le MNT.

3.1.4.2 Ouverture multiple de MNT

Lors de l'analyse, il a été déterminé que l'ouverture multiple d'un même MNT ne serait pas supportée dans Modeleur. Par contre, il faut que l'ouverture multiple d'une partition ou d'une couche soit supportée. Cela doit être fait parce que les même couches et les même partitions peuvent apparaître dans plusieurs MNT différents et donc être ouverts plusieurs fois simultanément.

Afin d'éviter de charger plusieurs fois les même données, le gestionnaire de base de données ne maintient qu'une seule instance d'une même structure en mémoire. De là, le gestionnaire MNT devra être capable de distinguer les différentes ouvertures de la même structure en mémoire.

3.1.5 Le module éditeur

Le module éditeur est un module complètement indépendant qui met en œuvre toutes les fonctionnalités d'édition des structures géométriques de Modeleur. L'interface du module éditeur a été prévue pour être très générique afin de pouvoir aisément le réutiliser au profit des autres modules de Modeleur.

Le MNT interface avec ce module éditeur pour assurer les fonctionnalités d'édition des sous-domaines composant les couches. L'éditeur se voit passer un conteneur d'objets géométriques (il est prévu que ce soit un conteneur de géométries TerraLib) représentant les sous domaines de la couche MNT à éditer. Comme il a été dit précédemment, la couche à éditer est celle qui détient le focus dans le MNT.

Le module éditeur a la responsabilité de superviser les demandes de création, de suppression et de modification des objets géométries du conteneur. De plus, il doit gérer l'affichage des géométries dans la couche interactive du module visualisation.

Le module éditeur doit en tout temps assurer la cohérence entre la vue des structures et leurs données associées. Le conteneur étant passé par référence, on assure cette synchronisation en tout temps.

3.2 Les diagrammes de séquences du MNT

La deuxième phase de l'analyse fut de mettre en place les diagrammes de séquences de l'activité MNT. Afin de représenter les différents scénarios d'usage du module MNT, cette série de diagrammes de séquences explique la répartition des tâches entre les différents acteurs de l'activité. Les diagrammes principaux sont exposés en annexe de ce document au point 7.1.

La construction de ses diagrammes a permis de bien définir la répartition du travail à accomplir entre les modules ainsi que les différentes interactions entre ceux-ci. En plus d'avoir l'avantage de former une vision globale du contrôle au sein de l'activité, cela a permis d'avoir une bonne idée des différentes structures de données qui allaient être nécessaires à l'étape de l'implantation.

Voici les quelques types de scénarios qui ont été étudiés. Cela permet de faire ressortir les points d'intérêts trouvés à chacun de ceux-ci.

- **Scénario d'édition :** Permet de voir la répartition du travail entre le module MNT, le module Editeur et le module Visualisation dans le cadre de l'initialisation de l'édition d'une couche MNT. Notez que c'est l'éditeur qui s'occupe de la couche interactive d'affichage;
- **Scénario de changement de focus :** Permet de voir la répartition du travail entre le module MNT, le GUI et le module Visualisation dans le cadre d'un changement de focus dans le système. Il

permet de mettre en évidence l'ordre des étapes à effectuées;

- **Scénario d'ouverture d'un MNT :** Permet de voir la répartition du travail entre le module MNT, le GUI et la BD dans le cadre de l'ouverture d'un MNT existant. Il permet de mettre en évidence le fait que le GUI interroge lui-même la BD sur la liste de MNT présents dans la BD;
- **Scénarios d'ajout :** Ce type de scénario englobe l'ajout d'une structure existante où d'une nouvelle structure à l'élément courant du système. Permet de faire ressortir la symétrie dans tous les cas d'ajout de structures ;
- **Scénarios de retranchement :** Ce type de scénario englobe tous les retranchements de structures composant un MNT. Permet de faire ressortir la symétrie dans la suite des étapes d'un tel travail;
- **Scénario de sauvegarde :** Permet de voir la répartition du travail entre le module MNT, le GUI et la BD dans le cadre de la sauvegarde de l'élément courant dans le système. Il permet de mettre en évidence la symétrie de la sauvegarde au sein de l'activité MNT;
- **Scénario de sauve sous :** Permet de voir la répartition du travail entre le module MNT, le GUI et la BD dans le cadre de la sauvegarde sous un nouveau nom de l'élément courant dans le système. Il permet de mettre en évidence la symétrie de la sauvegarde au sein de l'activité MNT;

De plus, il est à remarquer que sur les diagrammes, on a séparé la partie « menu » de la partie « fenêtre » du GUI. Ces deux parties de l'interface sont différenciées dans le but de détailler un peu plus les schémas. Il y a donc le GUI « menu » et le GUI « fenêtre ». Le GUI menu est constitué des différents items de menu spécialisés du MNT, c'est à dire qu'il s'occupe de recevoir les actions de l'usager provenant de ces menus. Il y aussi la partie « GUI fenêtre » qui est constituée de l'arborescence et des onglets sur lesquels repose l'espace pour visualiser/éditer les partitions et les couches MNT. Celui-ci est

l'hôte de manipulations directes sur les structures du MNT allant du simple changement de focus jusqu'à l'édition des sous domaines.

4 Design et implantation

La première étape du design du module MNT fut de construire le diagramme des classes du MNT. Ce diagramme présente les interfaces et les attributs des différentes classes du module. Cela a permis de pousser plus loin la conception du module MNT.

Voici une série de points qui tentent d'expliquer les différents concepts entourant le design et l'implantation du module MNT. On y retrouve l'explication du design ainsi que la raison d'être des classes du diagramme.

4.1 Le diagramme de classes du MNT

Le diagramme de classes du MNT est présenté en annexe au point 7.2 du présent document.

4.1.1 La classe *MTGestionnaire*

La classe *MTGestionnaire* est en quelque sorte la classe maîtresse du module. Comme son nom l'indique, son rôle est de gérer le fonctionnement du module, tout en assurant les interactions avec le reste de l'application.

Voici le détail des rôles principaux de la classe :

4.1.1.1 Envoi/réception d'événements

La classe *MTGestionnaire* a pour rôle de recevoir et d'envoyer des événements sur le bus.

Pour ce faire, la classe hérite de la classe *TEModuleBase*. Elle est branchée directement sur le BUS d'événement et elle contient les méthodes qui permettent d'émettre et de recevoir des événements.

Les événements qui ont trait au MNT sont décrit au point 4.1.6 de ce document.

4.1.1.2 Traitement des événements

La classe *MTGestionnaire* a aussi pour rôle de contrôler les traitements par rapport aux événements reçus. Elle doit effectuer la distinction entre ces derniers pour finalement appeler les bonnes fonctions et ainsi effectuer les traitements appropriés.

Pour cela, une série de méthodes privées, appelées *traitexxx()*, permettent de mettre en œuvre la série de traitements à effectuer pour chacun des types d'événements qu'il est possible de recevoir dans le gestionnaire. Une méthode *traiteMTEvenement* permet de distinguer la bonne méthode à appeler selon le contexte.

Pour le module MNT, dès que le focus est porté sur une couche MNT il y a réception d'un événement de changement de focus, lequel contient un lien vers une couche MNT. Le gestionnaire doit alors envoyer cette couche en édition. Il faut donc vérifier si l'événement indique que le focus est dirigé sur une couche, sur une partition ou sur le

MNT. Si le focus est sur une couche, cette mécanique se termine par la génération d'un événement d'édition de cette couche.

4.1.1.3 Maintenir les liens entre les structure de données

La classe *MTGestionnaire* a aussi la responsabilité de maintenir le lien entre les différentes structures d'un MNT. Elle maintient le lien entre les « Id » de chacune des fenêtres GUI ouvertes et les structures du MNT qui les concernent.

Pour ce faire, elle contient un attribut qui est une « Map ». Cette dernière garde chacun des « Id » de fenêtre en lien avec un objet informationnel *MTInfoMnt*, lequel objet est utilisé par le GUI. Ces objets sont définis en détails au point 4.1.3 de ce document.

4.1.2 Les classes conteneurs

Sur le diagramme de classe, on peut noter la présence des classes *MTMnt*, *MTPartition* et *MTCouche*. Ces classes sont des classes d'objets conteneurs. On les appelle ainsi puisqu'elles servent essentiellement à contenir les données en mémoire. Voici la raison d'être de ces classes :

- elles permettent d'obtenir un découpage beaucoup plus clair des classes en isolant les données des informations;
- elles permettent de représenter et de gérer la hiérarchie au sein du module. C'est à dire qu'un objet de type *MTMnt* contient une liste d'objets *MTPartition* et qu'un objet *MTPartition* contient une liste d'objets *MTCouche*.
- De plus, la priorité des couches au sein d'une partition, ainsi que la priorité des partitions au sein d'un MNT, sont maintenues à l'aide des listes d'objets. L'ordre des entrées dans la liste correspond à l'ordre de priorité. Pour changer l'ordre de priorité, on doit changer l'ordre dans la liste.

Toutes les classes conteneurs sont composées d'une liste de pointeurs à d'autres conteneurs référant aux structures qu'elles contiennent. À titre d'exemple, un objet *MTPartition* contient une liste de pointeurs à des objets *MTCouche*.

Les classes *MTPartition* et *MTCouche* possèdent un attribut définissant le type de données portées. De plus, la classe *MTCouche* contient le « Id » du jeu de données qu'elle représente.

Puisque chaque type d'aspect du terrain (substrat, glace, topographie, etc.) implique un type de données différent, les classes *MTCouche* deviendront très certainement des classes virtuelles et permettront de générer des héritiers spécialisés pour chacun de ces cas de figure.

4.1.2.1 MTMnt

Cette classe *MTMnt* est le conteneur associé à un MNT. Voici l'énoncé des attributs de celle-ci suivi d'une brève description :

- *listePartition* : liste de pointeurs à des objets conteneurs *MTPartition*;

4.1.2.2 MTPartition

Cette classe est le conteneur associé à une partition. Voici l'énoncé des attributs de celle-ci suivi d'une brève description :

- *listeCouche* : liste de pointeurs à des objets conteneurs *MTCouche*;
- *typePartition* : attribut définissant le type de la partition (topographie, substrat, glace, ...) .

4.1.2.3 MTCouche

Cette classe est le conteneur associé à une couche.

Les couches du MNT sont elle même constituées d'un conteneur de géométries. Ce conteneur répond à une interface dérivée de la classe *EDInterfaceConteneur* (voir Rapport de réunion – Deuxième réunion sur l'éditeur) et assure le respect des contraintes d'édition des sous domaines de couche MNT.

Voici l'énoncé des attributs de la classe suivit d'une brève description :

- *sousDomainesP* : pointeur au conteneur de géométries;
- *typeCouche* : attribut définissant le type de la couche (topographie, substrat, glace, ...) ;
- *jeuDonneeId* : attribut contenant l'identifiant unique (Bd) du jeu de données associé à la couche MNT.

4.1.3 Les classes informatives (préfixe MTInfo)

Les classes du module MNT dont le nom commence par *MTInfo* sont des classes dites informatives. Ces classes d'objets permettent de contenir toutes les données d'information et d'opération concernant les composantes de MNT. Cela permet d'éviter la pollution des classes conteneurs par de l'information reliée au contrôle du MNT. D'ailleurs, ces classes *MTInfo* contiennent toutes comme attribut un pointeur vers l'objet conteneur qui les concerne.

Il y a trois classes informatives dans le module. Il y a la classe *MTInfoMnt*, la classe *MTInfoPartition* et la classe *MTInfoCouche*. Dans l'ordre, celles-ci servent à contenir toutes les informations sur un MNT, sur une partition et sur une couche.

Au même titre que pour les classes conteneurs, ces classes informatives sont organisées afin de représenter la hiérarchie de données informatives du module. Un objet de type *MTInfoMnt* contient une liste d'objets *MTInfoPartition* et un objet *MTInfoPartition* contient une liste d'objets *MTInfoCouche*.

Voici le détail de chacune de ces classes informatives :

4.1.3.1 MTInfoMnt

La classe *MTInfoMnt* contient l'information se rapportant à un MNT en particulier.

Voici la définition des structures et des attributs qui la compose:

- *mntP* : pointeur à l'objet conteneur MTMnt;
- listInfoPartition : liste de pointeurs à des objets informatifs MTInfoPartition;
- nom : attribut contenant le nom du MNT;
- elementsActifs : structure contenant les pointeurs aux objets MTInfoPartition et MTInfoCouche afin de maintenir les liens aux éléments qui ont le focus dans le MNT (voir le point 4.2.2 du présent document).

Notons qu'à l'initialisation, le GUI reçoit et maintient un lien vers un objet de cette classe. C'est donc cette classe qui constitue le lien de données entre les fenêtres GUI et leur MNT.

4.1.3.2 MTInfoPartition

Cette classe contient toute l'information se rapportant à une partition.

Voici la définition des attributs qui la compose:

- *partitionP* : pointeur à l'objet conteneur *MTPartition*;
- *listInfoCouche* : liste de pointeurs à des objets informatifs *MTInfoCouche*;
- *nom* : attribut contenant le nom de la partition.

4.1.3.3 MTInfoCouche

Cette classe contient toute l'information se rapportant à une couche et à son affichage.

Voici la définition des attributs qui la compose:

- *coucheP* : pointeur à l'objet conteneur *MTCouche*;
- *nom* : attribut contenant le nom de la couche;
- *couleur* : attribut indiquant la couleur d'affichage de la couche;
- *affichable* : drapeau indiquant si la couche est visible à l'affichage;
- *imprimable* : drapeau indiquant si la couche est visible à l'impression.

4.1.4 Les classes *MTCoucheVtk*

Lors du travail sur un MNT, le focus du système pourra être mis sur les différents composants de celui-ci. Si le focus du système pointe sur une partition, on doit voir le contenu de celle-ci s'afficher dans l'onglet du GUI qui lui est dédié. Il faut donc pouvoir afficher de façon statique toutes les couches de cette partition active selon leur ordre de

priorité. Pour cela le module MNT doit pouvoir générer la couche d'affichage d'une partition pour le module de visualisation.

Cela doit être fait de façon conditionnelle puisque si le focus est sur une des couches de la partition en question, on doit pouvoir bâtir la couche d'affichage statique, mais sans la couche active.

Pour ce faire, on a prévu une classe de couche VTK d'une partition (*MTCoucheVtk*) ainsi qu'une classe d'algorithme de construction des objets de cette classe (*MTAlgoCoucheVtk*).

La classe *MTCoucheVtk*, qui est une classe héritière de la classe *RGCouche* (voir rapport sur le module de visualisation), met en place les structures qui permettent au module visualisation d'afficher le contenu d'une partition. Elle permet donc de faire l'interface entre le module MNT et le module visualisation.

4.1.5 La classe *MTAlgoCoucheVtk*

La classe *MTAlgoCoucheVtk* met en place l'algorithme qui permet de construire et de remplacer la *MTCoucheVtk*.

Le module MNT fait appel à cet algorithme à chaque changement de focus dans le MNT puisque la couche d'affichage statique doit rester représentative des données en tout temps. L'algorithme reconstruit une nouvelle couche d'affichage pour remplacer l'ancienne à chaque fois.

4.1.6 Les classes d'événements

Les classes d'événements qui concernent le gestionnaire de MNT sont définies dans un fichier d'entête (*MTEvenement.h*). Pour plus de clarté, celles-ci n'apparaissent pas dans le diagramme de classes.

Voici la définition de chacune des classes d'événements:

4.1.6.1 La classe MTEvenement

Les événements spécifiques adressés au MNT et impliquant des traitements sur un MNT sont regroupés dans la classes d'événements *MTEvenement*.

Voici les attributs d'un objet *MTEvenement* :

- *typeAction* : attribut permettant d'indiquer le type de traitement à effectuer dans le gestionnaire MNT (membre de l'énumération *typeMNTAction* : AJOUTE, AJOUTE_NOUVEAU, SUPPRIME, RETIRE, SAUVE, SAUVE_SOUS, CHANGE_PRIORITE et CHANGE_FOCUS);
- *typeItem* : attribut permettant d'indiquer le type d'item MNT concerné par l'événement (membre de l'énumération *typeMNTItem* : MNT, PARTITION et COUCHE);
- *fenId* : attribut contenant le Id de la fenêtre GUI concernée;

- *itemId* : attribut indiquant l'identifiant unique (Id base de données) d'un item de MNT (MNT, partition, couche);
- *infoPartitionP* : pointeur à un objet *MTInfoPartition*;
- *infoCoucheP* : pointeur à un objet *MTInfoCouche*.

4.1.6.2 La classe MTEvenConfigureCommandesIG

Comme son nom l'indique, ce type d'événement permet de configurer les commandes de menus qui concernent le MNT. Ce type d'événement ne contient aucun attribut.

4.1.6.3 La classe MTEvenEditeCouche

Ce type d'événement est généré par le gestionnaire MNT et sert essentiellement à indiquer au module Éditeur d'initialiser l'édition d'un conteneur de sous domaines de la couche MNT active.

Un seul attribut, le pointeur au conteneur des sous domaines, est associé à l'événement :

- *sousDomainesP* : pointeur au conteneur de géométries (voir le point 4.1.2.3);

4.1.6.4 La classe MTEvenRafraichisGui

Ce type d'événement est généré par le gestionnaire MNT et sert essentiellement à indiquer au module GUI de rafraîchir la mise en forme et l'affichage d'une de ses fenêtres GUI. Un seul attribut, le Id de la fenêtre en question, est associé à l'événement :

- *fenId* : attribut contenant le Id de la fenêtre GUI concernée.

4.2 Structures du diagramme

Le diagramme de classes du MNT contient diverses composantes de données informatives. Cela rend le diagramme plus clair et permet au lecteur de s'y retrouver plus facilement.

Certaines de ces composantes étant affiliées à un même contexte ont été regroupées à l'intérieur de structures de données. D'autres sont simplement des énumérations permettant de faciliter les traitements.

Voici la description de chacune d'elles :

4.2.1 Structure *InfoMntCoucheVtk*

Cette structure fait partie de la classe *MTGestionnaire*. Elle permet de contenir les pointeurs aux objets associés à un MNT.

- *infoMntP* : pointeur à l'objet informationnel du MNT;
- *coucheVtkP* : pointeur à la couche d'affichage VTK.

Puisque le gestionnaire reçoit des événements contenant le « Id » de la fenêtre active, celui-ci doit conserver le lien entre l'objet *MTInfoMnt* et cette fenêtre pour assurer la cohérence du système. De la même façon, le gestionnaire doit connaître le « Id » de la fenêtre pour envoyer des événements et les adresser à la bonne fenêtre. Celui-ci conserve donc cette structure *InfoMntCouche* dans un *Map* qui lui lie l'Id de la fenêtre.

4.2.2 Structure *ElementActif*

Cette structure se trouve dans la classe *MTInfoMnt*. Elle permet de contenir les pointeurs aux objets informatifs qui sont actifs (qui détiennent le focus) dans le MNT à un instant donné.

- *infoPartitionP :* pointeur sur la partition active;
- *infoCoucheP :* pointeur sur la couche active.

Ensemble, les deux pointeurs permettent de savoir exactement sur quel item du MNT le focus est maintenu à un instant donné. Trois cas de figure sont représentés selon la valeur de ces deux pointeurs :

- Si les deux pointeurs ne sont pas nuls, le focus est détenu par la couche pointée par *infoCoucheP*. On l'appellera la couche active. Pour sa part, le pointeur *infoPartitionP* nous dirige sur la partition parente de la couche active.
- Si le pointeur à la couche est nul et que le pointeur à la partition, lui, n'est pas nul, c'est que le focus est sur une partition. À ce moment, la partition pointée est la partition active;
- Si les deux pointeurs sont nuls, le focus est sur le MNT en tant que tel.

4.3 L'interface graphique du MNT (GUI)

L'interface graphique du MNT, ou GUI, consiste en un module indépendant mais étroitement lié au MNT (projet *PY_Module_MNT*). Le GUI est codé en python et est une extension de l'interface de Modeleur.

Le GUI doit s'occuper de la création des fenêtres, de la réception des événements de souris et de la mise en forme de l'arborescence et des onglets des partitions du MNT.

Il a été déterminé que celui-ci devait rester le plus simple possible. Ce n'est pas lui qui doit gérer la synchronisation avec les structures de données du MNT. C'est au gestionnaire MNT que revient la responsabilité d'aviser le GUI qu'une mise à jour est nécessaire en envoyant un événement de mise à jour (voir le point 4.1.6.4).

Le GUI doit ensuite voir à ce que cette mise à jour se fasse. Pour ce faire, le module conserve un lien à la structure informative du MNT à l'aide d'un pointeur à l'objet *MTInfoMnt* de ce dernier.

De plus, lorsque l'usager pose une action, le GUI a la responsabilité d'envoyer les bons événements au gestionnaire MNT afin que ce dernier dispose de toute l'information dont il a besoin pour effectuer ses traitements. Pour ce faire, il doit déterminer le ou les items du MNT qui sont concernées par l'action, déterminer le type d'événement à envoyer, joindre les pointeurs des items concernés à l'événement et l'envoyer sur le bus.

4.4 Prototype du module MNT

Le travail d'analyse sur l'activité MNT ayant été méticuleusement fait, on a pu passer à l'élaboration d'un prototype pour le module MNT et son GUI. Ce prototype a permis de mettre en place le module de façon itérative, c'est à dire une composante à la fois, et de permettre la vérification de chacune de celles-ci avec soin.

5 Conclusion

5.1 Résumé

Dans ce document, on a présenté l'activité MNT et on a résumé le travail d'analyse qui lui a été consacré. De plus, il a été question du design et de l'implantation effectuée jusqu'à maintenant en ce qui concerne le module MNT.

5.2 Perspectives futures

Au moment de la rédaction de ce document, le prototype en est à sa toute première ébauche.

En ce qui concerne le GUI, on dispose de la mécanique de création des fenêtres et de la mise à jour de celles-ci. On peut aussi envoyer des événements sur le BUS.

Pour le module MNT, les entêtes et les interfaces des classes principales ont été mises en place. De plus, la mise en place de la mécanique d'envoi/réception d'événements a été effectuée.

Finalement, quelques tests ont été effectués sur le GUI, le gestionnaire MNT et sur le module GestionnaireBD, pour vérifier que les événements circulent bien entre ceux-ci.

La suite logique des événements est de compléter le prototype du module en suivant les conclusions du travail d'analyse. Cela comprend entre autres de:

- terminer la mise en œuvre des fonctions du gestionnaire du MNT;
- terminer la classe *MTEvenement*;
- terminer la mise en œuvre du GUI Python;
- mettre en place la classe des couches d'affichage statiques VTK ainsi que l'algorithme de construction;
- Revisiter/modifier les tables reliées au MNT dans la BD;
- Revisiter et implanter la création/chargement/déchargement/sauvegarde des structures MNT (MNT, partitions, couches) dans le gestionnaire de BD

6 Glossaire

ACT : Action de l'usager sur le GUI fenêtre;

Arbo : L'arborescence du MNT;

BD : Base de données;

BUS : Bus d'événement;

DLG: Boîte de dialogue;

EV: Événement envoyé sur le BUS;

EV in : Paramètre d'entrée envoyé avec l'événement;

EV out : Paramètre de retour de l'événement;

JD : Jeu de données;

MAJ : Mise à jour;

MNU : Item de menu;

Req: Requête;

Ptr: Pointeur;

TRT: Traitements internes;

Couche MNT : Structure associée à un jeu de données, qui contient une liste de sous domaines (voir document « Spécifications Modeleur 2.0 »);

Couche VTK : Ensemble d'objets VTK destinés à l'affichage. Elle est ordinairement construite à partir des polygones d'une couche MNT;

Éléments actifs (EA): L'entité du MNT qui détient le focus dans le système à un moment donné. Cela peut être le MNT lui-même, une *Partition* du MNT ou encore une *Couche* MNT;

Partition courante : On dit de la partition qu'elle est courante à un certain moment si l'élément actif est l'une de ses couches MNT ou si la partition est elle-même l'élément actif;

Polygone : Objet géométrique tiré de la librairie TerraLib, décrivant une forme fermée. Il est constitué d'une ou de plusieurs polylignes qui, à leur tour, sont composées de sommets ainsi que des arrêtes les reliant entre eux;

7 Annexes

7.1 Diagrammes de séquences du MNT

7.1.1 Édition de la couche MNT active

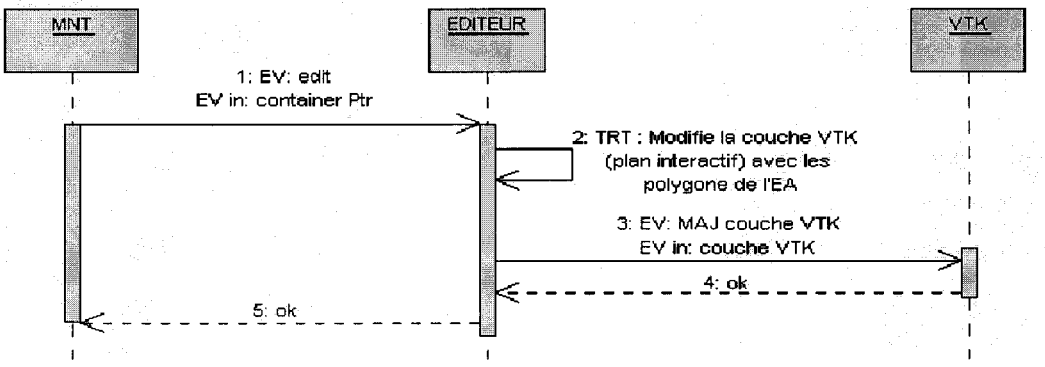


Figure 3 - Diagramme de séquence de l'édition de la couche MNT active

7.1.2 Changement de focus

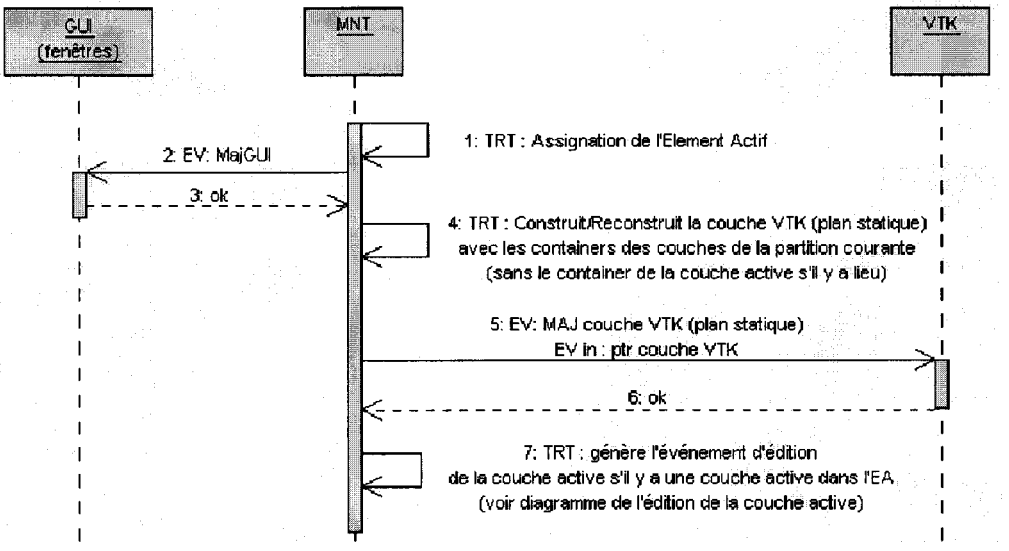


Figure 4 - Diagramme de séquence d'un changement de focus

7.1.3 Création d'un MNT

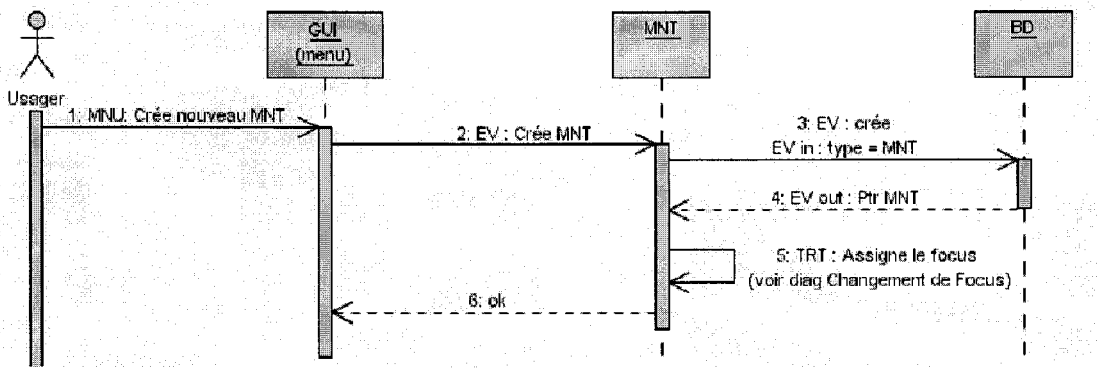


Figure 5 - Diagramme de séquence de la création d'un MNT

7.1.4 Ouverture d'un MNT

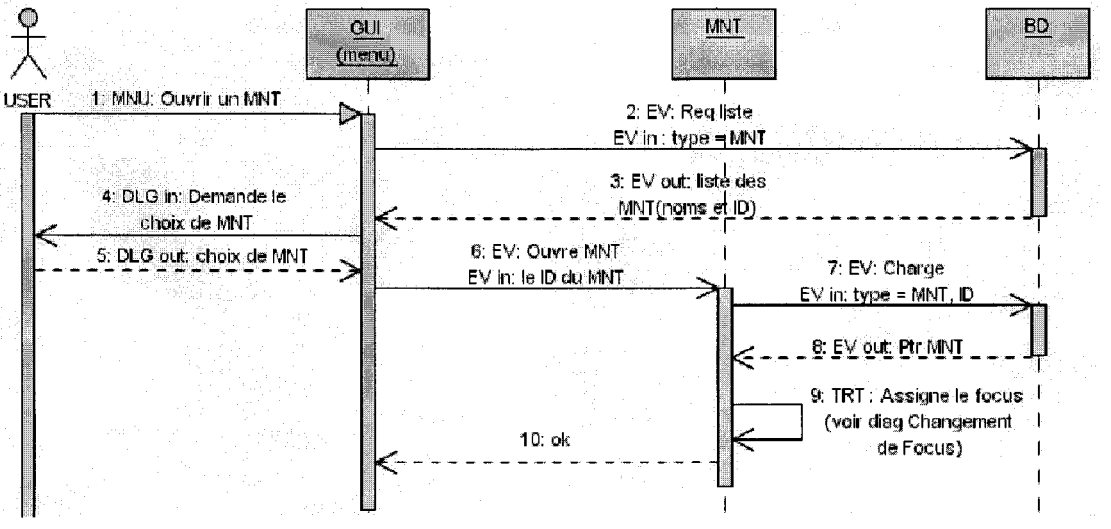


Figure 6 – Diagramme de séquence de l'ouverture d'un MNT

7.1.5 Sauvegarde d'un MNT

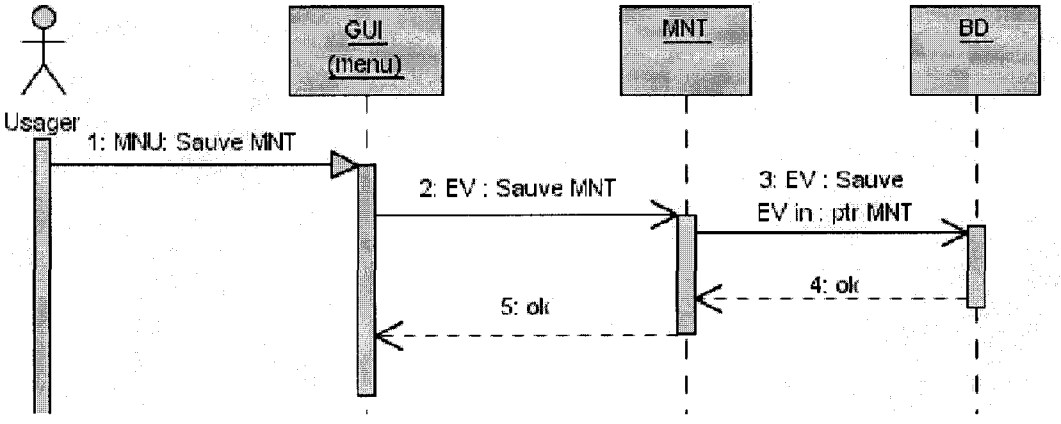


Figure 7 – Diagramme de séquence de la sauvegarde du MNT

7.1.6 Sauvegarde d'un MNT sous un autre nom

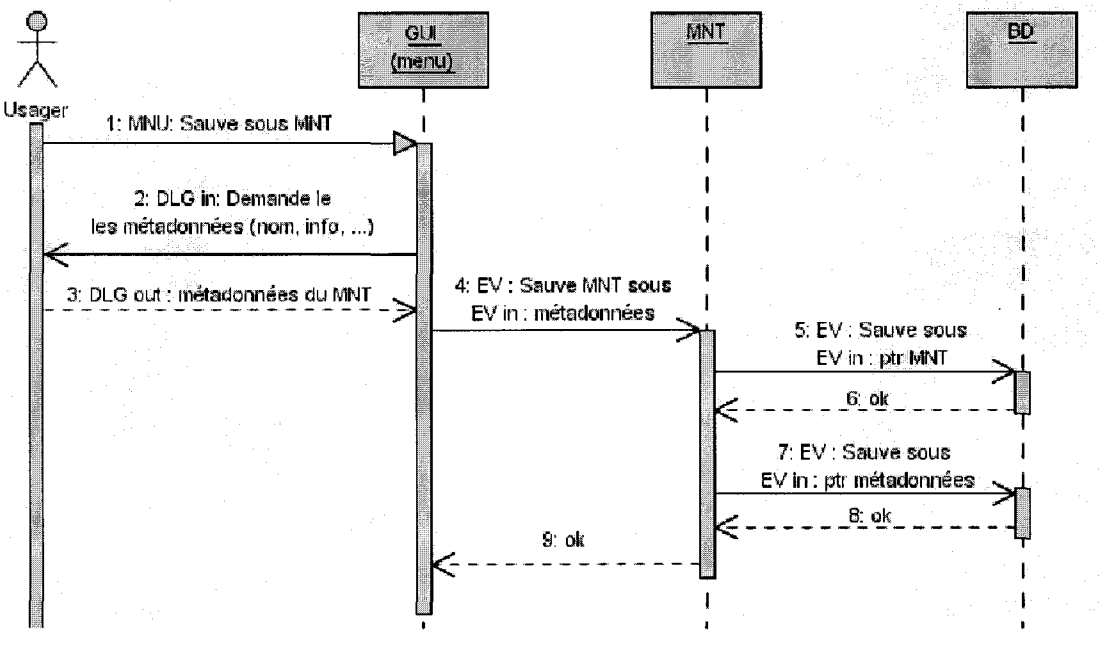


Figure 8 - Diagramme de séquence de la sauvegarde d'un MNT sous un autre nom

7.1.7 Ajout d'une couche existante à la partition courante

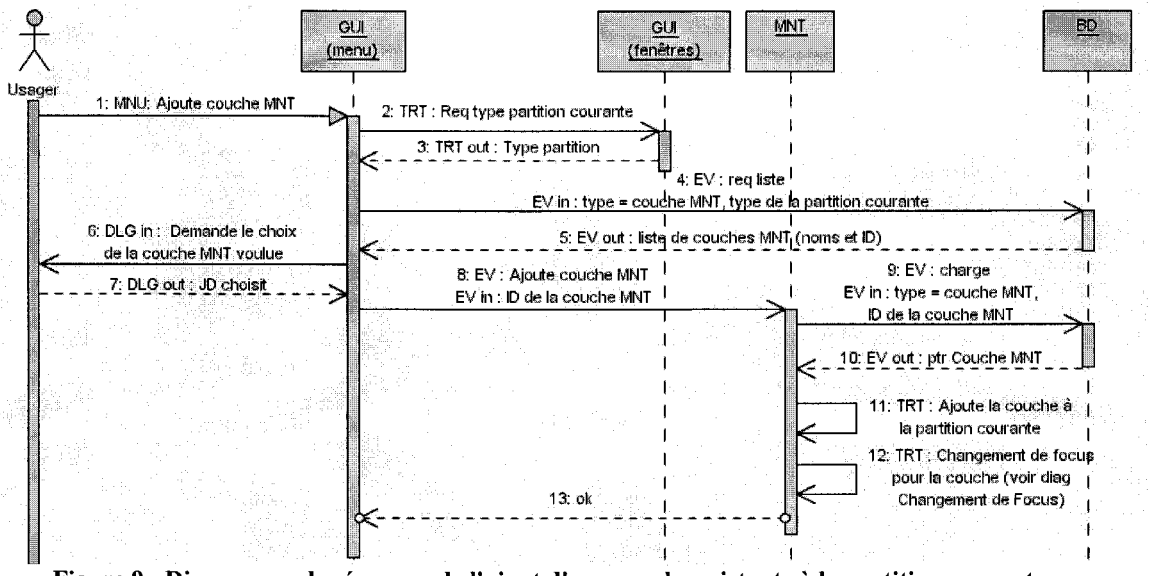


Figure 9 - Diagramme de séquence de l'ajout d'une couche existante à la partition courante

7.1.8 Ajout d'une nouvelle couche à la partition courante

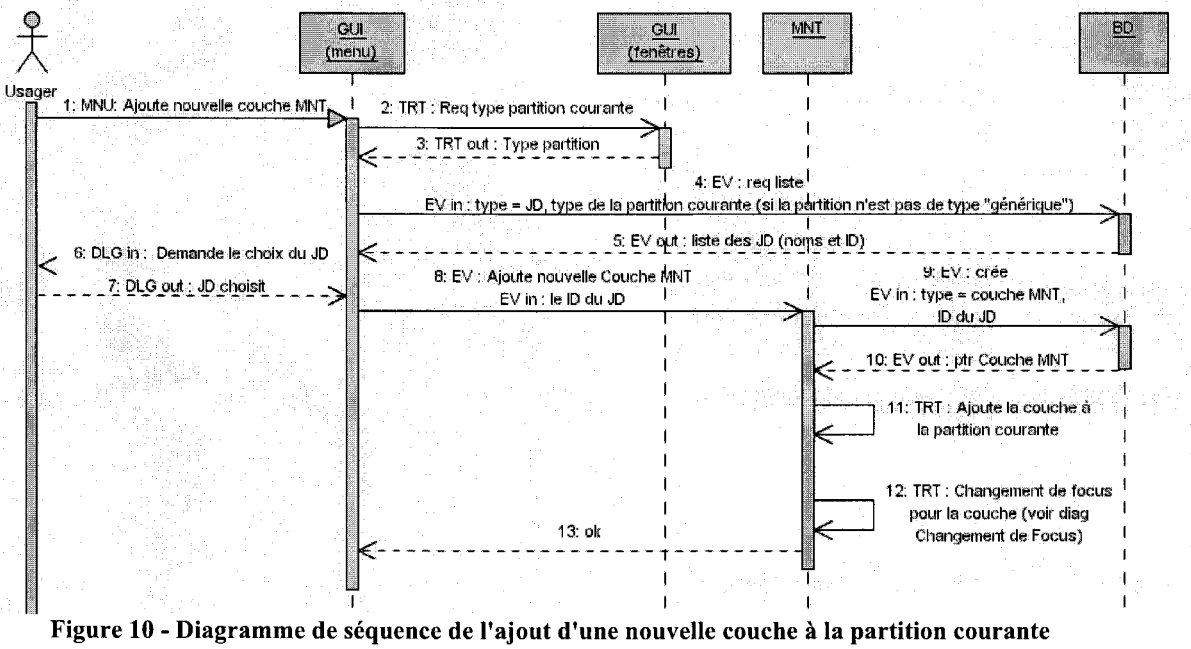


Figure 10 - Diagramme de séquence de l'ajout d'une nouvelle couche à la partition courante

7.1.9 Ajout d'une partition à un MNT

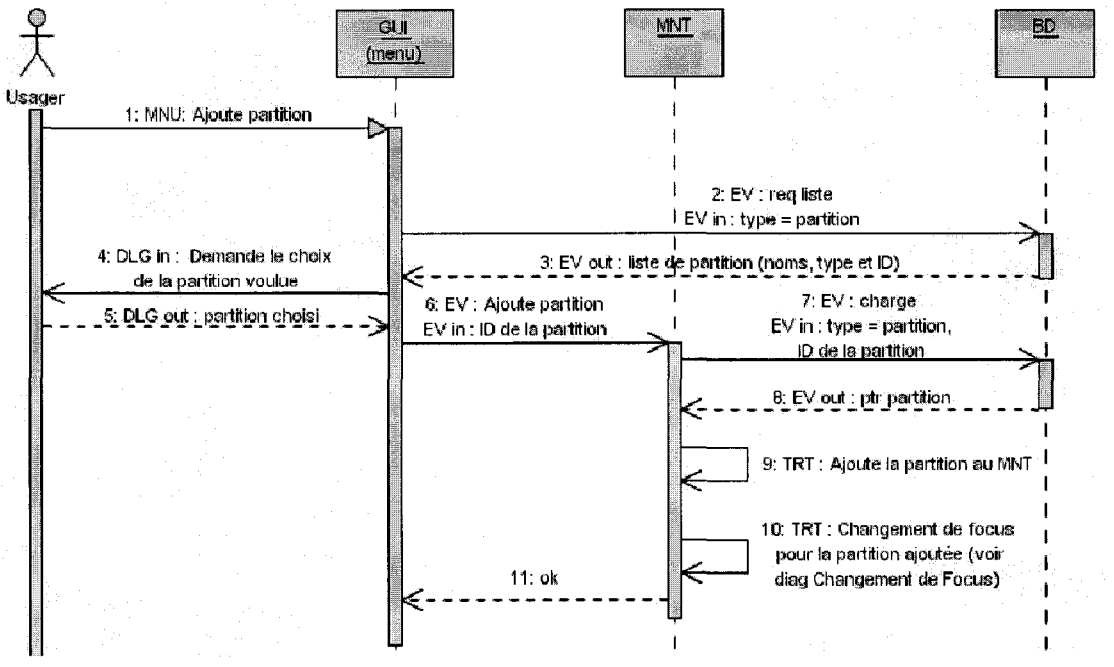


Figure 11 - Diagramme de séquence de l'ajout d'une partition à un MNT

7.1.10 Ajout d'une nouvelle partition à un MNT

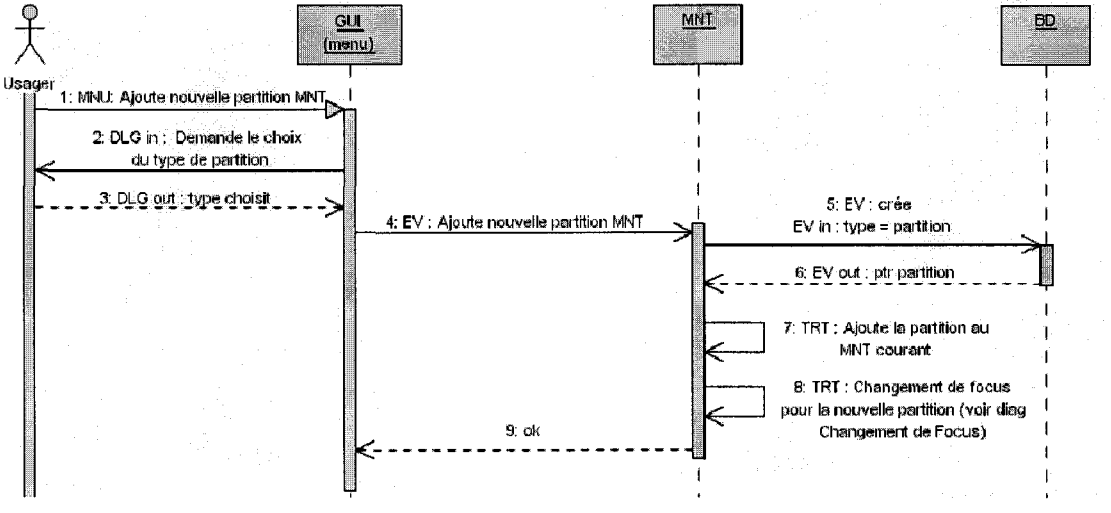


Figure 12 - Diagramme de séquence de l'ajout d'une nouvelle partition à un MNT

7.1.11 Retranchement d'une partition à un MNT

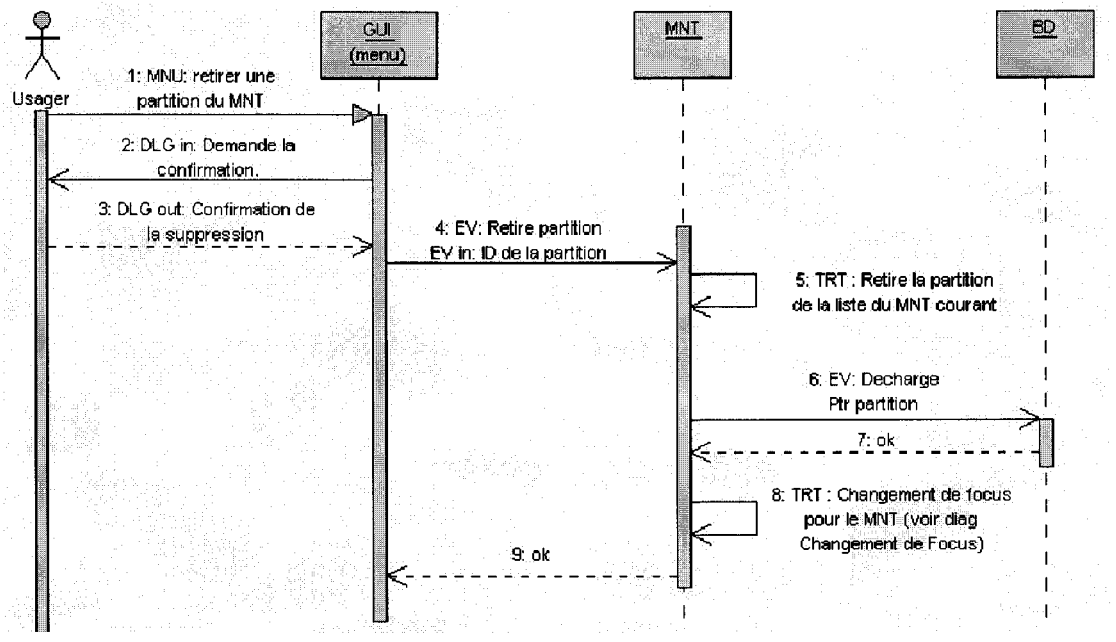


Figure 13 - Diagramme de séquence du retranchement d'une partition à un MNT

7.1.12 Retranchement d'une couche à une partition

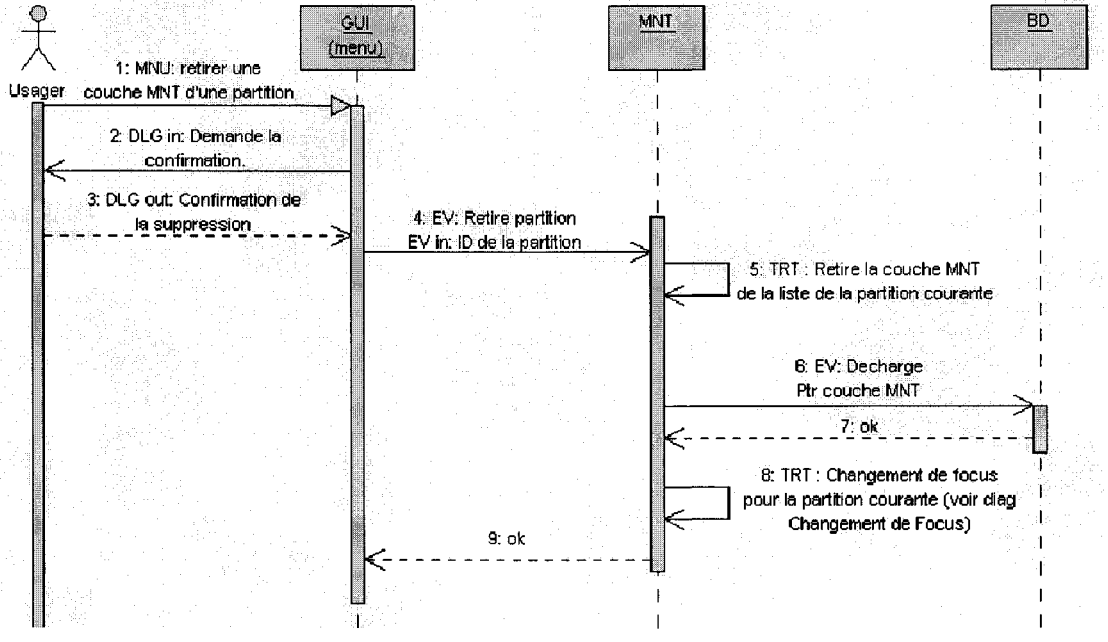
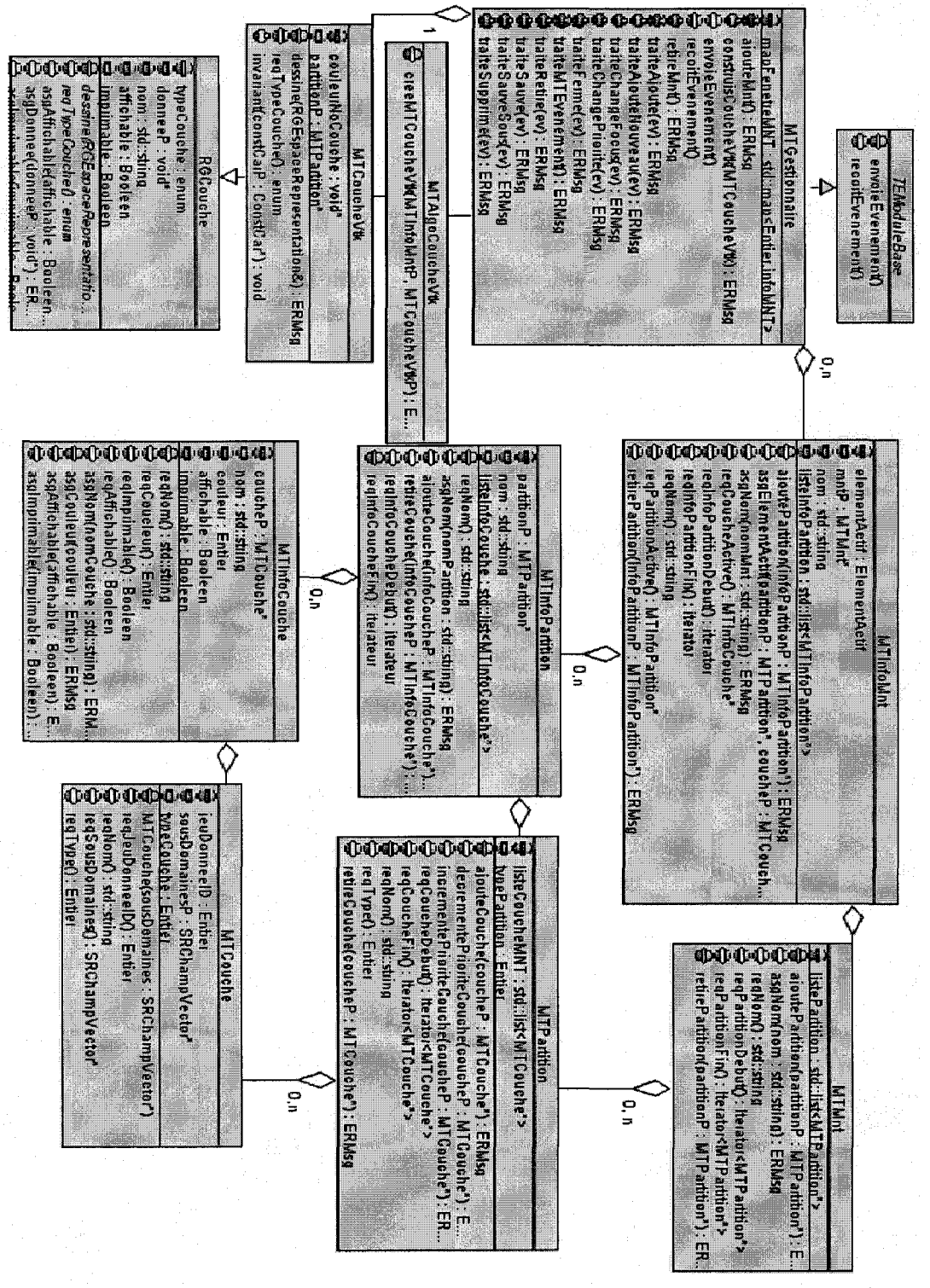


Figure 14 - Diagramme de séquence du retranchement d'une couche à une partition

7.2 Diagramme de classes du MNT



RAPPORT # 4 : Module de validation

Rapport de développement logiciel

Rapport sur le module de validation

Présenté par :

Benjamin Behaghel

13 août 2004

Versions du document

Date	Auteur	Commentaires	Version
15/06/2004	Benjamin Behaghel	Version initiale	V 1.0
22/06/2004	Benjamin Behaghel	Corrections et changements à partir du §3	V 1.1
28/06/2004	Benjamin Behaghel	Corrections	V 1.2
08/07/2004	Benjamin Behaghel	§ 4.2.2	

SOMMAIRE

Table des figures

1. Introduction

1.1 Objectifs

Le but de ce rapport est de présenter les différentes composantes du module de validation de données de Modeleur 2.0 et de résumer les différentes décisions prises lors du travail d'analyse, de design et d'implantation.

Ce rapport ne prétend pas suivre chronologiquement toutes les étapes de la réflexion sur le module de validation mais relate les décisions qui ont finalement été prises et leurs raisons. Il explique aussi les différents principes de fonctionnement de ce module.

1.2 Contexte

Modeleur est un système d'information géographique (SIG) dédié à l'hydraulique fluviale. La première étape lors de son utilisation est d'importer puis de manipuler des semis de points. Ces semis de points sont des relevés altimétriques du terrain concerné. Dans ces relevés, il y a un certain bruit, c'est à dire des points qui ne représentent pas le sol. Ces points sont par exemples des arbres, des toits de maison, des épis de maïs... C'est pourquoi il est nécessaire de pouvoir visualiser ces points afin de s'assurer de leur cohérence et de pouvoir ôter le bruit.

Il y deux parties dans le travail de validation : l'une permettant la visualisation et la manipulation manuelle des points via un outil graphique (voir rapport sur le module éditeur) et l'autre permettant le traitement automatique de ces points (voir rapport sur le module de filtres).

Le présent rapport concerne la partie sur l'affichage de ces points. En effet, le nombre de points que modeleur 2 sera amené à traiter est très grand, c'est pourquoi il est nécessaire de mettre en place et d'utiliser des structures de données particulières pour les manipuler.

1.2.1 Références

Ce rapport fait suite au travail d'analyse et de design réalisé autour du module validation. Ces travaux ont fait l'objet des rapports suivants:

- Semi de points & éditeur. Résumé de réunion - 14-04-2004. 8 pages.
- Module validation de données & Semi de points. Résumé de réunion - 27-04-2004. 9 pages

Ces rapports s'appuient sur les spécifications fonctionnelles de Modeleur 2 :

- Spécifications – Modeleur 2.0. Québec, INRS-Eau, Terre & Environnement. 68 pages.

Ces travaux s'appuient aussi sur le module éditeur :

- Rapport final sur l'éditeur-1.4. 29 pages

Les travaux effectués s'appuient et concernent aussi le module Gestionnaire de base de donnée :

- \\Caxipi\Green\Rapports\Base de données

1.3 Structure du document

Plusieurs concepts ont été abordés au cours de l'élaboration du module de validation et plusieurs modules sont impliqués dans sa réalisation. Chacun de ces concepts est repris et expliqué dans la suite de ce rapport.

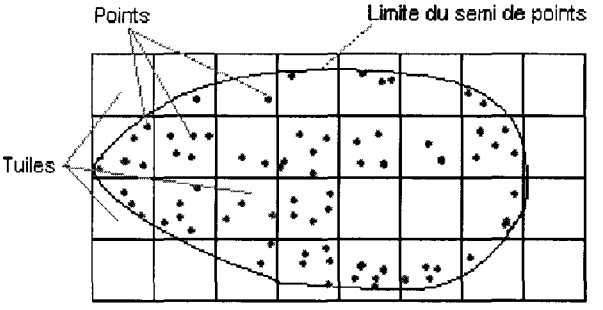
2 Notions de bases pour le module de validation

2.1 Semi de points

Un semi (de points) représente l'ensemble des points mesurés appartenant à un même projet.

2.2 Tuile

Une tuile est une partie d'un semi de point.



Exemple de semi de point et de tuiles

2.3 Point (type de)

Si tous les points d'un même semi sont de même type, les points des différents semis peuvent êtres de types différents. C'est à dire qu'ils peuvent porter différents types d'informations. Certains types de points pourront, par exemple, contenir une information de pression, de température ou d'altitude. On parlera souvent de points de topo. Ce sont des points représentant une altitude. C'est avec des jeux de données de points de topo que le module de validation a initialement été mis en place.

3 Le module Validation

Le but du module de validation est de permettre l'utilisation de grands jeux de points. Les principaux problèmes posés par la taille de ces jeux de données sont les suivants :

- il est impossible de charger tous les points en mémoire ;
- le temps de chargement de ces points dans la base de données peuvent être long ;
- on ne peut pas afficher tous les points d'un semis en même temps à l'écran.

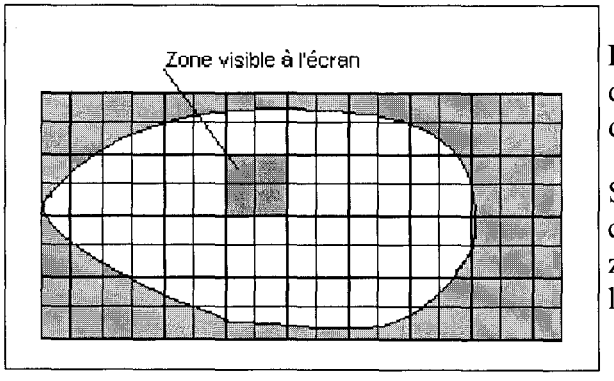
Le module validation va permettre de répondre à ces problèmes par la mise en place de solutions au maximum transparentes pour l'utilisateur.

3.1 Tuiles

Le module validation se base sur un système de tuiles. Ces tuiles découpent l'espace d'une manière prédéfinie. Tout point d'un semi appartient à une tuile.

Ces tuiles vont permettre d'accélérer le chargement des points dans la base de données puisque au lieu d'avoir à faire des requêtes spatiales[1] il sera possible d'effectuer des requêtes sur un index[2] de la table.

Ces tuiles permettent de faciliter les chargements et les déchargements de points dans la base de données. En effet les zones à charger/décharger sont toujours carrées[3] et il est relativement aisé de définir les tuiles à charger pour remplir l'écran (ou la zone dans laquelle dessiner).



Le dessin à gauche représente un semi dont la partie visible à l'écran est formée de 4 tuiles.

Sans le système de tuiles, la zone à charger serait dans ce cas une simple zone carrée. Mais cela se compliquera lorsqu'on déplacera la caméra.

[1] Notons cependant qu'il est assez facile de repasser à un système de requêtes spatiales tout en conservant le principe des tuiles. Il suffit de se débrouiller pour que les zones à charger soit les mêmes que celles définissant les tuiles pour retrouver exactement le même résultat.

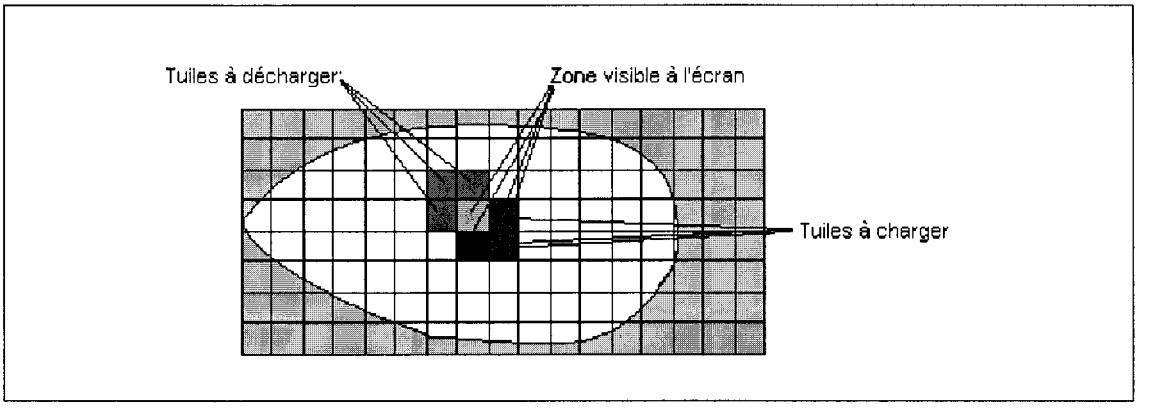
[2] Pour que les index soient pris en compte, ne pas oublier de faire un **VACCUM ANALYSE** sur la table des points.

[3] Les zones définies dans le module validation sont carrées ou rectangulaires car se sont les formes facilitant le plus le travail. Il serait possible de modifier le système pour avoir des zones d'une autre forme.

Dans l'image ci-dessous, on a déplacé la caméra vers le bas et vers la droite.

Avec un système de tuiles, il suffit alors de décharger 3 tuiles et d'en recharger 3.

Sans le système de tuiles il faudrait décharger une zone assez complexe formant un angle et charger une zone assez complexe formant, elle aussi un angle. Une autre solution serait de décharger tout et de recharger une zone carrée, mais on désire gagner du temps sur les chargements/déchargements de données dans la base de données



On peut grâce à cet exemple commencer à comprendre l'intérêt d'un système s'appuyant sur des tuiles.

Un autre aspect important est qu'il existe déjà un module gestionnaire de BD qui est chargé d'exécuter les requêtes et de gérer un compte de référence[4]. Le compte de référence se marie très bien avec le système de tuiles. En effet les tuiles permettent d'identifier proprement un groupe de points. De cette façon, on ne peut pas tomber sur un groupe de point à moitié chargé, chaque zone est soit chargée, soit non chargée. Il en irait tout autrement si l'on avait pas de tuiles prédéfinies, il serait alors impossible de faire coïncider exactement les zones à charger et à décharger et l'on serait donc à chaque déplacement de caméra obligé de décharger tous les points pour ensuite charger complètement la zone qui nous intéresse.

[4] Le compte de référence a été mis en place lors de l'implémentation du module de validation. Il consiste en une table d'association entre des chaînes de caractères identifiant la requête et des pointeurs correspondant aux données chargées avec cette requête. Ce compte de référence permet lorsqu'une requête a déjà été effectuée, de retourner le pointeur sur les données sans refaire la requête. Cela présente deux intérêts :

- Gain de temps puisque la requête n'a pas à être effectuée.
- Partage des données. Deux modules exécutant la même requête vont avoir un pointeur vers les mêmes données. Ainsi lorsque l'un modifiera ces données, elle seront à jour pour l'autre.

Attention cependant, lors de requêtes spatiales par exemple, mais dans n'importe quel cas en général, il est toujours possible de récupérer les mêmes données avec des requêtes différentes. Dans ce cas, le compte de référence ne pourra pas fonctionner et il existera plusieurs pointeurs vers les mêmes données. Cela pourrait poser des problèmes, notamment lors de commit, il pourrait y avoir des écrasements de données.

3.1.1 Tuiles et points utilisateurs

En suivant parfaitement le principe des tuiles, chaque point appartiendrait à une et unique tuile. Cependant, il existe une exception : les points utilisateurs. Ces points sont un peu particuliers en regard des autres, en effet, il est possible de les supprimer, de les déplacer et d'en ajouter de nouveaux (ces opérations ne sont pas permises pour les autres points).

Si l'ajout et la suppression de ces points ne posent pas de problèmes d'implantation, il n'en va pas de même pour le déplacement qui implique qu'un point peut changer de tuile. On a vu que le chargement des tuiles se faisait via un gestionnaire de BD qui utilise un compte de référence. Plusieurs problèmes peuvent alors se poser :

- Comment déplacer un point d'une tuile vers une tuile qui n'a pas été chargée ?
- Comment déplacer un point vers une tuile qui a été chargée par une autre partie de l'application ?
- Comment supporter le fait qu'un déplacement d'un groupe de points puisse répartir ces points dans plusieurs tuiles et le fait qu'il soit nécessaire de mettre à jour plus d'information lors de la sauvegarde des points dans la base de données ?

Pour ne pas avoir tous ces problèmes, il a été décidé puisque les points utilisateurs sont peu nombreux, de les séparer dans une tuile à part : la tuile des points utilisateurs. Tous les points utilisateurs, et uniquement eux, se retrouvent dans une seule et unique tuile qui englobe tout le semis. Si on déplace un point utilisateur il reste dans la tuile des points utilisateurs et si on ajoute un point, il est automatiquement ajouté dans cette tuile.

3.2 Chargements/déchargements

3.2.1 Semi

Les tuiles sont définies par une longueur et une largeur et par rapport à une tuile d'origine. Ces informations sont maintenues dans la table « semi » présentée ci-dessous.

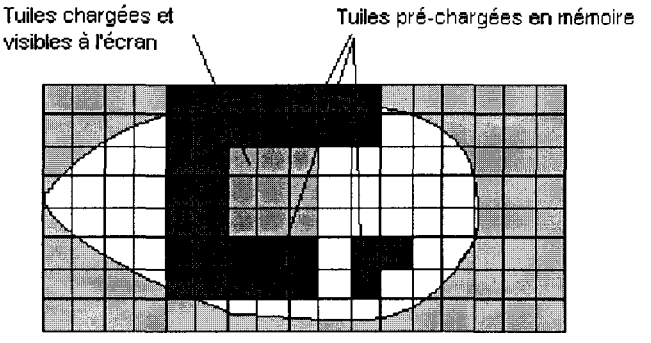
semi_topo
semi_topo_id (PK)
srid (FK)
information_id (FK)
position_tuile_origine_x
position_tuile_origine_y
longueur_tuile_x
longueur_tuile_y
premiere_tuile_i
premiere_tuile_j
derniere_tuile_i
derniere_tuile_j
nbPointsParTuile
niveauDetailMax

Table des semis de Topo

Ainsi, grâce à ces informations il va être possible de déterminer exactement quelles sont les tuiles qu'il faut charger pour l'affichage. Il est important de pouvoir répondre de façon précise à cette question car l'on veut que les chargements soient les plus rapides possibles et l'on ne veut pas, par conséquent, charger des zones qui ne seront pas affichées. Il est possible d'obtenir les limites de l'écran grâce à un événement[5] du Module d'affichage.

On connaît ainsi les tuiles nécessaires pour l'affichage. On va donc charger ces tuiles. Parmi celles-ci, certaines sont déjà chargées (puisqu'en général la caméra se déplace localement), grâce au compte de référence leur chargement est alors instantané.

Le déchargement des tuiles qui ne sont plus utiles est fait ensuite. Il est possible de garder les tuiles en mémoire dans un buffer au lieu de les décharger. Ainsi, si l'utilisateur fait des déplacements inverses à ceux qu'il vient d'effectuer ou si certaines zones sont plus souvent chargées que d'autres, le chargement des tuiles de ces zones sera immédiat. Ce buffer pourra être libéré au fur à mesure suivant des règles à déterminer[6] (dernière utilisation, distance à l'écran…)



Exemple de tuiles bufférisées

3.2.1.1 Tout visualiser

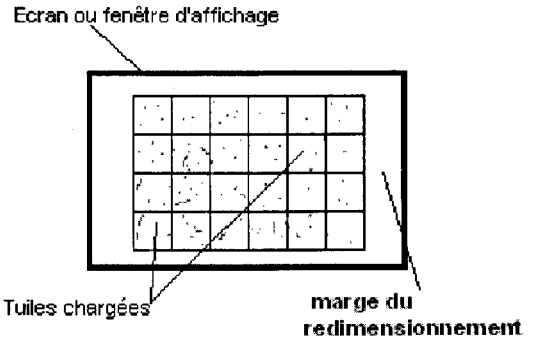
Les champs première/dernière_tuile_i/j de la table semi servent lorsqu'on désire visualiser le semi dans son ensemble. Quand cela arrive, on charge toutes les tuiles comprises entre ces bornes, sans tenir compte de la place disponible à l'écran, et on les affiche. Pour finir, on demande au module de visualisation de redimensionner l'espace pour permettre de voir l'ensemble des tuiles chargées et affichées[7].

Cette méthode présente toutefois un léger inconvénient. Lorsque l'espace d'affichage se redimensionne, il le fait avec une certaine marge (voir §4.2.1).

[5] Cet événement « *AFEvenTailleEspaceVTK* » permet de connaître les valeurs des bornes de l'écran. Il a été publié à l'occasion de la création du module validation. Il n'est pas vraiment destiné à être réutilisé par d'autres modules car il répond vraiment à un besoin spécifique et particulier.

[6] Actuellement aucune tuile n'est conservée en buffer, ce qui correspond à la règle pas plus de 0 tuile dans le buffer

[7] On fait en fait l'inverse de ce qui est fait habituellement à savoir : regarder la taille de l'écran pour ne charger que ce qui entre dedans.



exemple de marge suite à un redimensionnement

3.2.2 Création d'un semi

Lors de la création d'un nouveau semi, on s'écarte un peu du design du module gestionnaire de BD. En effet, pour pouvoir charger des points, il est nécessaire d'avoir un Id de semi.

L'id de semi est obtenu lors de l'appel de la fonction « creeDonnee » du module gestionnaire de BD. Cette fonction demande à la BD la prochaine valeur d'une séquence prédéterminée (*semi_topo_semi_topo_id_seq*) et la retourne. Normalement, cette valeur ne devrait être demandée que lors de la sauvegarde et le « creeDonnee » ne devrait faire qu'un « new ».

Cette manière de procéder implique que des numéros de séquence peuvent être « perdu » si l'on crée des semis mais qu'on ne les sauvegarde pas[8].

3.2.3 Points

Les points sont, eux, regroupés dans des tables en fonction de leur type. Les données présentes dans ces tables varient donc en fonction de ce même type mais certaines sont tout de même communes à tous les points. La table des semis de Topo est ici donnée à titre d'illustration.

Les tuiles ne sont pas nécessaires dans la table des points, elles ne sont présentes que pour des raisons d'optimisation. Elles sont destinées à être remplacée par un système de requêtes spatiales.

Notons que l'opération « création de tuiles » n'a pas de sens. Les tuiles sont un artifice utilisé pour accélérer le chargement. Certaines tuiles n'ont pas de points, mais on les charge de la même façon que celles qui en ont.

[8] Ceci n'a pratiquement aucune conséquence à moins de créer des semis par millions. Il est cependant intéressant de s'en rappeler pour comprendre au besoin les Id attribués aux semis.

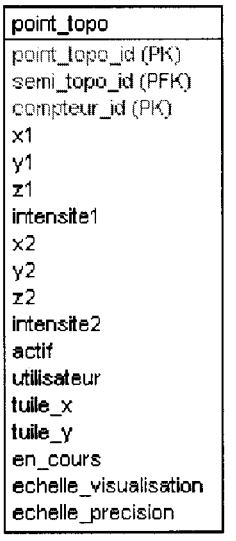


Table des points de Topo

Les caractéristiques communes à tous les points sont :

- point_topo_id : identifiant du point. Notons que cet identifiant n'est pas unique pour 2 raisons : des points de semis différents peuvent avoir le même Id et les copies temporaires (voir sauvegardes temporaires) d'un point ont le même Id ;
- compteur_id : sert à différencier un point de ses copies temporaires ;
- x1 : coordonnée x du point (dans le repère défini pour ce semi) ;
- y1 : coordonnée y du point (dans le repère défini pour ce semi) ;
- echelle_visualisation : permet de définir une distance à partir de laquelle on ne veut plus voir le point. En effet, tous les points ne pouvant être affichés à l'écran, il est inutile de tous les charger. L'échelle de visualisation est un nombre dont l'interprétation est laissée au module de validation ;
- echelle_precision : champ non encore utilisé ;
- actif : permet de savoir si un point est actif ou non (s'il entre dans le calcul des filtres ou non) ;
- utilisateur : identifie les points utilisateurs ;
- en_cours : permet de savoir quel est le point à afficher/charger parmi un point et ses copies temporaires ;

Les caractéristiques « tuile_x » et « tuile_y » sont communes à tous les types de points mais il est cependant possible de s'en passer puisqu'elles ne sont qu'une autre représentation du « x1 » et du « y1 ».

3.2.4 Calcul du niveau de détail

Du fait de leur grande quantité, les points ne peuvent pas tous êtres visibles à l'écran. Le niveau de détail définit quels sont ceux qui vont être chargés et affichés. Cela permet de diminuer le temps de chargement des tuiles.

Le calcul du niveau de détail se fait en résolvant l'équation suivante :

Nombre de points désirés pour l'affichage = nombre moyen de points par tuile * nombre de tuiles à charger * proportion.

La seule inconnue est la valeur proportion. Une fois la proportion obtenue, il ne reste plus qu'à faire le lien entre la proportion et le niveau de détail.

Exemple : on désire afficher environ 50000 points. Il y a en moyenne 5000 points par tuiles et nous avons 40 tuiles à charger : proportion = 50000 / (5000 * 40) = 0.25. Il faut donc afficher 1 point sur 4 dans ces tuiles pour avoir 50000 points. Si nous avons 10 niveaux de détail[9] cela correspond au niveau 2 ou 3.

3.2.5 Sauvegardes Temporaires et nettoyage de la BD

Au cours de la manipulation des points d'un semi, certaines tuiles sont chargées, déchargées, rechargées et ainsi de suite. Les modifications apportées aux points de ces tuiles doivent apparaître au cours des chargements ultérieurs. Pour cela, il faut s'arranger pour garder l'information quelque part. On ne peut pas garder en mémoire toutes les tuiles sur lesquelles des modifications ont été apportées puisque cela pourrait revenir à garder toutes les tuiles en mémoire et, d'un autre coté, on ne veut pas écraser les données initiales (les points d'origines sont identifiés grâce au champ *compteur_id* qui est à 0) de la base de donnée tant que l'utilisateur ne demande pas de sauvegarde.

Pour résoudre ce problème, lorsqu'une tuile est déchargée, ses points sont sauvegardés temporairement dans la base de données, dans la même table que les points originaux. Les champs « en_cours » et « compteur_id » (voir § 3.2.3) permettent de différencier les différents points. Lors de l'opération de sauvegarde, au lieu d'enregistrer des données, on va nettoyer la base en enlevant toutes les sauvegardes temporaires qui ne servent plus.

Notons cependant que les valeurs des points d'origine ne doivent jamais être supprimées de la base.

3.3 Design du module validation[10]

Le diagramme de classes du module de validation est présenté en annexes (§6).

[9] 10 niveaux de détail signifie : niveau 0 : tous les points ; niveau 1 : 9 points sur 10 ; niveau 2 : 8 points sur 10 et ainsi de suite.
[10] Identifiant du module : VD (Validation de Données)

3.3.1 VDGestionnaire

Le module de validation s'appuie sur le système récurrent d'un gestionnaire, ici le gestionnaire de validation (*VDGestionnaireValidation*). Celui-ci est chargé de gérer les événements et de « distribuer » le travail au container de tuiles. Il doit aussi maintenir une interface de container de tuile (*VDInterfaceContainerTuiles*) héritant du *EDInterfaceContainer* du module éditeur et permettant d'afficher les points des tuiles.

3.3.2 VDContainerTuiles

La pièce principale du module de validation est le container de tuiles (*VDContainerTuiles*). C'est lui qui a en charge la gestion des tuiles. Il contient une tuile de points utilisateurs et une liste de tuiles représentant les tuiles chargées et affichées. Pour sa gestion interne, il peut aussi maintenir une liste de tuiles chargées non affichées (tuiles non déchargées) qui sont les tuiles dites bufférisées. Cette classe contient les méthodes permettant de charger des tuiles, de les sauver et de les décharger. Elle permet aussi de récupérer les différentes informations portées par un point. C'est aussi ici que doit être déterminé le niveau de détail[11] à charger.

Le container de tuiles est template du type des points contenus dans les tuiles. Pour casser le template et de manière à ce que le gestionnaire de validation n'ait à connaître que des containers de tuiles de même type, il existe un container de tuiles générique (*VDContainerTuilesGenerique*) dont hérite *VDContainerTuiles*. Ce dernier possède en plus des infos sur un semi de points (classe *VDInfoSemi*).

3.3.3 VDInfoSemi

La classe VDInfoSemi reprend toutes les informations présentées dans le §3.2.1 afin de permettre le chargement des bonnes tuiles.

3.3.4 VDTuiles

Les tuiles (*VDTuiles*) sont template du type de points elles contiennent l'information commune aux points d'un même tuile (id du semi, tuile_x et tuile_y) ainsi qu'un ensemble de points (*VDPoints*).

3.3.5 VDPoints

Les points (*VDPoints*) sont templates du type de point. Cela s'explique par le fait qu'un certain nombre de paramètres (tels que les coordonnées x et y) sont communs à tous les points. Les points ont donc une partie commune qui provient de l'héritage de *VDPointGenerique* et une partie qui leur vient des points spécialisés, par exemple, les

[11] Le niveau de détail permet de ne pas charger tous les points afin de sauver du temps, il peut être déterminé, par exemple, à partir du nombre de tuiles.

points de topo (*VDTopo*). Les points possèdent une méthode leur permettant de retourner l'ensemble de leurs valeurs sous la forme d'une chaîne de caractères[12].

3.3.6 VDInterfaceContainer

Le *VDInterfaceContainer* quant à lui interface le module éditeur de manière à pouvoir réaliser l'affichage des points chargés. Notons qu'*EDInterfaceContainer*[13] présente des possibilités plus larges que celles requises dans le cas d'un semi de points, toutes les fonctionnalités concernant les polygones ou les polylignes n'ont donc pas été implémentées. Notons aussi que la plupart des opérations ne concernent que la tuile utilisateur puisque c'est la seule sur laquelle on peut ajouter/enlever/déplacer des points.

Le *VDInterfaceContainer* contient un pointeur aux différentes tuiles du *VDContainerTuile*. Cela lui permet de tenir à jour et d'être tenu à jour par le *VDContainerTuile*.

3.3.7 Événements

3.3.7.1 Événements reçus

Le module validation répond à plusieurs événements particuliers qui sont :

- VDEvenEditionMenu : Cet événement fait suite à une demande de l'utilisateur. Il se décompose en :
  - Charge : permet de charger un semi existant
  - Nouveau : permet de créer un nouveau semi
  - Sauve : permet de sauver le semi courant
  - Afficher tout : permet de visualiser le semi dans son ensemble
  - Afficher propriétés : permet de connaître les propriétés d'un point sélectionné
- AFEvenCameraModifie : cet événement indique que la zone visible à l'écran à changée, que la caméra a été déplacée. Quand le module de validation reçoit cet événement, il doit s'assurer que les tuiles présentement chargées couvrent bien toute la surface visible à l'écran. Dans le cas contraire il faut charger les tuiles manquantes.

[12] Cette chaîne pourrait être de type xml est sert, entre autres, lors de l'affichage ou de la modification des propriétés des points. Elle pourrait servir à générer une boite de dialogue associée. Voir le rapport « Filtre du module validation - 18-05-2004.doc ». Voir §4.3.1

[13] EdinterfaceContainer est l'interface qu'il faut implémenter pour pouvoir être édité par le module éditeur. Voir aussi le rapport final sur le module éditeur.

3.3.7.2 Événements envoyés

Le module validation au cours de son utilisation envoie différents événements sur le bus. Ces événements sont les suivants :

- des événements de chargement dans la base de données pour charger les infos du semi ou les points du semi (*GDEvenChargeDonnee*) ;
- des événements de déchargement dans la base de données pour décharger les infos du semi ou les points du semi (*GDEvenDeChargeDonnee*);
- des événements de sauvegarde dans la base de données pour sauver les infos du semi ou les points du semi (*GDEvenSauveDonnee*) ;
- des événements de création dans la base de données pour créer de nouveaux semis (*GDEvenCreeDonnee*) ;
- des événements pour redessiner ce qui se trouve dans le *VDInterfaceContainerTuile* ;
- des événements pour ajuster la taille de l'écran (*AFEvenResetCamera*);
- des événements pour connaître la taille de l'écran (*AFEvenTailleEspaceVTK*) ;
- des événements pour connaître les points sélectionnés par l'utilisateur (*EDEvenListeSelection*).

3.3.8 Fonctionnement général

Le module de validation fonctionne de la façon suivante :

- Il reçoit un ordre de chargement de semi, il charge ce semi dans un *VDContainerTuile* et crée un *VDInterfaceContainerTuile* pour lui permettre d'afficher les tuiles chargée.
- Par la suite, le module peut recevoir des événements lui indiquant que l'espace de représentation a changé (*AFEvenCameraModifie*). Dans ce cas, il charge et décharge des tuiles pour s'adapter au nouvel espace. En fonction de la taille de l'espace et donc du nombre de tuiles à charger, il va être déterminé automatiquement un niveau de détail qui permettra de ne pas charger tous les points des tuiles afin d'améliorer les performances du module.

4 Conclusion

4.1 Résumé

Le module de validation permet la manipulation de semis de points en s'appuyant sur un système à base de tuiles. Pour ce faire, il utilise le gestionnaire de BD et son compte de référence.

Le module de validation manipule des points templates. Si on veut utiliser autre chose que des points de Topo, il est toutefois nécessaire d'implémenter, outre le type de point désiré, le *GDAlgoIO*[14] correspondant à ce type de point et modifier le gestionnaire de validation pour qu'il puisse créer des containers de tuiles de ce type.

Le *VDInterfaceContainerTuile* permet l'affichage par le module éditeur des points chargés.

Deux nouveaux événements ont été ajoutés au module d'affichage : l'un pour connaître la taille de l'espace et l'autre pour demander un redimensionnement en fonction des objets affichés.

4.2 Problèmes & améliorations

4.2.1 Redimensionnement de l'espace

Ce problème est présentement résolu par le fait qu'une fois les limites (points extrêmes) du semi définies, on décide de ne pas charger les tuiles au delà de ces limites puisqu'elles ne ramèneraient aucun points. Ces limites doivent êtres définies dans la table « info_semi » .

Le problème de redimensionnement de la marge lors du redimensionnement de l'espace (§3.2.1.1) peut être résolu en chargeant immédiatement après le redimensionnement les tuiles manquantes pour compléter la marge. Le problème de temps pourrait être résolu en ne chargeant que les tuiles des 2 extrémités, redimensionnant ensuite pour après charger effectivement toutes les tuiles de l'espace affiché. Mais le nombre de tuiles chargées alors ne sera pas celui espéré puisqu'on aurait alors chargé les tuiles se trouvant dans la « marge », c'est à dire, au delà de celles indiquées.

Il existe peut-être des fonctionnalités de VTK permettant de réduire, voire de supprimer la marge liée au redimensionnement.

[14] Ce GDAlgoIO_NouveauTypeDePoint sera quasiment identique au GDAlgoIOTuileTopo qui peut alors servir d'exemple. Les seules modifications sont d'adapter les requêtes aux types de valeurs portées par les points.

Une autre approche serait de ne pas avoir les tuiles que l'on désire afficher, mais bien l'espace (sous forme de coordonnées géographique) que l'on désire afficher. Il faudrait alors disposer d'un événement permettant de modifier la taille de l'espace d'affichage puis charger toutes les tuiles présentes dans cet espace. Cette approche a déjà été mise en place mais n'a pas été conservée car d'une part le redimensionnement n'était pas bien précis et d'autre part, il a été exprimé qu'il pouvait être dangereux de laisser un événement permettant de redimensionner l'espace VTK à tous les modules

4.2.2 Requêtes spatiales

Il est intéressant de passer à une table de points permettant d'effectuer des requêtes spatiales. Cela permet de se débarrasser des indices de tuiles, il sera ainsi très facile de redimensionner les tuiles, il suffira en effet de changer les valeurs longueur_tuile_x et longueur_tuile_y de la table semi. La version courante sur CVS utilise les requêtes spatiales.

Les performances avec les requêtes spatiales sont nettement moins bonnes qu'avec les indices de tuiles mais il devrait être possible d'atteindre au moins les mêmes performances.

4.2.3 VDPoints & VDTuiles

Les *VDPoints* et *VDTuiles* sont présentement dans le module de validation. Or, ils sont aussi utilisés par le module de filtres. D'autres modules pourraient aussi être amenés à les utiliser. Il serait certainement intéressant d'isoler cette partie du module de validation dans un module indépendant.

4.3 Perspectives futures

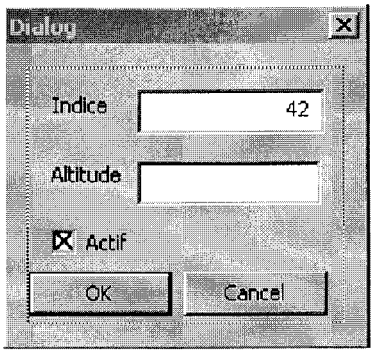
4.3.1 Boites de dialogue et XML

Utiliser un format spécifique tel que le format XML pour contenir de l'information permettrait de créer un module spécialisé dans l'affichage de ces informations sous forme de boîte de dialogue interactive ou non.

Ainsi, la chaîne de caractère XML ci-dessous nous donnerait la boîte de dialogue suivante :

<TITRE>**Dialog**</TITRE>
<ITEM>
<NOM>**indice**</NOM>
<INT>**42**</INT>
<COMMENTAIRE>indice du point</COMMENTAIRE>
</ITEM>
<ITEM>
<NOM>**altitude**</NOM>

<INT></INT>
<COMMENTAIRE>indiquer l'altitude désirée</COMMENTAIRE>
</ITEM>
<ITEM>
<NOM>**actif**</NOM>
<CHOIX>**vrai**</CHOIX>
<COMMENTAIRE>activer/désactiver le point</COMMENTAIRE>
</ITEM>



Boite de dialogue généré à partir d'une chaîne XML

Une fois le champ altitude rempli avec une valeur (disons 4808), quand l'utilisateur appuiera sur ok, il récupérera sa chaîne originale avec un item modifié :

<ITEM>
<NOM>altitude</NOM>
<INT>**4808**</INT>
<COMMENTAIRE>indiquer l'altitude désirée</COMMENTAIRE>
</ITEM>

S'il appuie sur « cancel », c'est la chaîne d'origine qui est retournée.

4.3.2 Support des différents types de points

Une des contraintes du module de validation est de pouvoir gérer différents types de points. Pour le moment seuls les points de topo ont été utilisés. Pour utiliser d'autres points, il faut suivre la procédure suivante :

- Exporter le nouveau type de points à partir du *VDGestionnaire* comme c'est fait pour les points de topo. Ex : template DLL_IMPEXP(MODULE_VALIDATION) VDPoint<VDTopo>;
- Ajouter dans le module gestionnaire de BD une classe GDAlgoIO-Nouveau_type_de_point en s'inspirant de GDAlgoIOTuileTopo et ajouter aussi un GDAlgoIOInfoSemiNouveau_type_de_point pour le nouveau type de point.
- Dans *VDGestionnaire* dans la fonction *traiteVDEvenEditionMenu*, faire la séparation entre les différents types de points.

- Dans la base de données, ajouter une table pour les points de type nouveau et une table info pour ces points.

4.3.3 Performances

Les performances pour le module de validation sont difficiles à définir. Plusieurs paramètres entrent en jeu :

- Performance de la machine sur laquelle est installée la base de données.
- Optimisation de la base de données. Par exemple entre une base sans index et avec index le temps de chargement des tuiles va du simple au triple. En faisant un vaccum analyse après avoir crée les index le temps de chargement peut être divisé par 10. D'autres optimisations existent sûrement, il peut être intéressant de chercher dans cette voie pour améliorer les performances de la BD.
- Taille des tuiles : avec des tuiles plus grandes, il y a moins de requêtes à exécuter et il est possible de gagner du temps. Par contre plus les tuiles sont grandes, plus on va charger de points qui ne seront pas affichés. Il est nécessaire de trouver un juste milieu qui dépendra de la densité du semi.
- Nombre de points chargés. Le niveau de détail est intéressant car il permet de ne pas charger tous les points. Il reste alors à définir le nombre de points avec lequel on désire travailler.
- Nombre de tuiles à charger. Le nombre de tuiles influe aussi sur la vitesse d'exécution. Il est plus long de récupérer quelque chose avec plusieurs requêtes qu'avec une seule requête. Il peut donc être intéressant de charger plusieurs tuiles ensemble ou de changer la taille des tuiles en cours de processus.

5 Glossaire

Point : position géographique à laquelle peut être affecté un ensemble de valeurs.

Points utilisateurs : les points utilisateurs sont des points crées par un utilisateur contrairement aux autres qui proviennent de relevés effectués de différentes manières.

Semi de points : ensemble de point d'une même projection.

Template : technique de génération automatique de code en paramétrant des fonctions ou des classes.

Tuile : sous partie d'un semi. Les tuiles sont un artifice destiné à améliorer les temps de chargement de points dans la base de données.

VTK : Visualisation ToolKit. Ensembles de librairies graphiques.

6 Annexes

6.1 Diagramme de classes

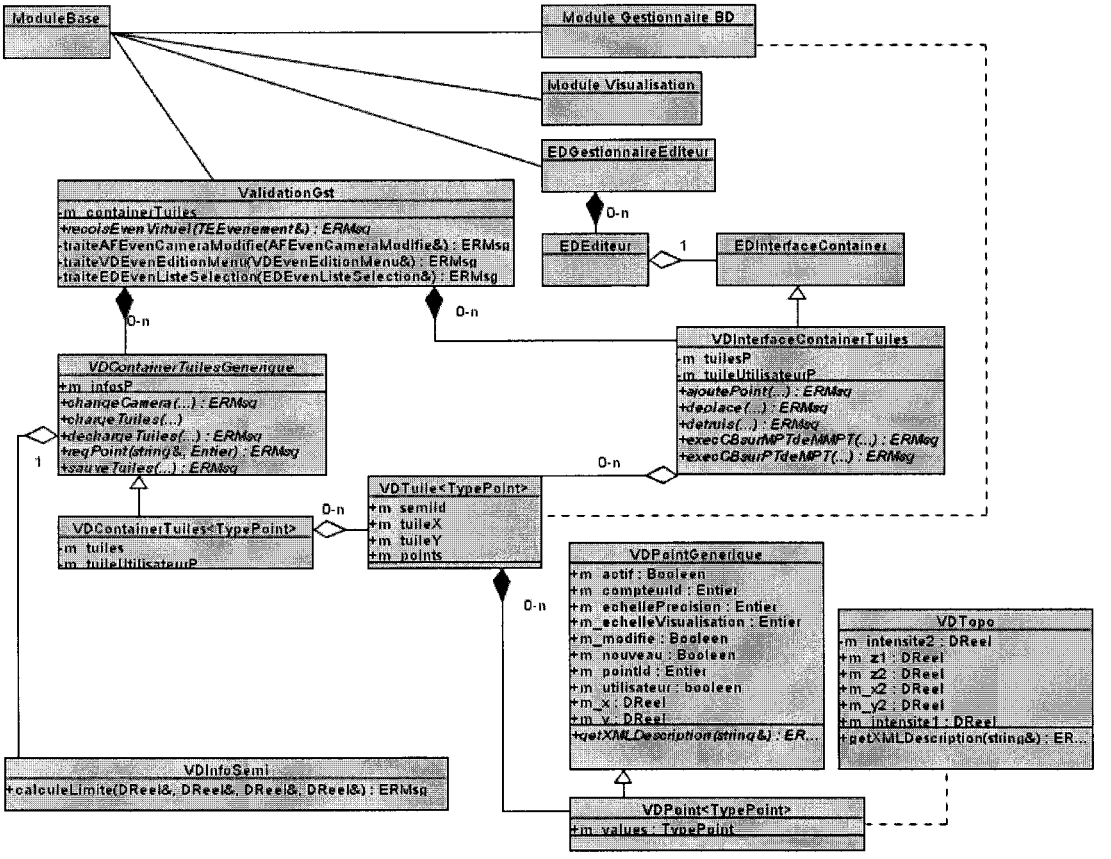


Diagramme de classes du Module Validation

RAPPORT # 5 : Module de filtres

Rapport de développement logiciel

Rapport sur le module de filtres

Présenté par :

Benjamin Behaghel

13 août 2004

Versions du document

Date	Auteur	Commentaires	Version
15/06/2004	Benjamin Behaghel	Version initiale	V 1.0
23/06/2004	Benjamin Behaghel	Correction à partir du §3.2	V 1.1
28/06/2004	Benjamin Behaghel	Corrections	V 1.2

SOMMAIRE

Table des figures

1. Introduction

1.1 Objectifs

Le but de ce rapport est de présenter les différentes composantes du module de filtrage de Modeleur 2.0 et de résumer les différentes décisions prises lors du travail d'analyse, de design et d'implantation.

Ce rapport ne prétend pas suivre chronologiquement toutes les étapes de la réflexion sur le module de filtrage mais relate les décisions qui ont finalement été prises et leurs raisons. Il explique aussi les différents principes de fonctionnement de ce module.

1.2 Contexte

Modeleur est un système d'information géographique (SIG) dédié à l'hydraulique fluviale. La première étape lors de son utilisation est d'importer puis de manipuler des semis de points. Ces semis de points sont des relevés altimétriques du terrain concerné. Dans ces relevés, il y a un certain bruit, c'est à dire des points qui ne représentent pas le sol. Ces points sont par exemples des arbres, des toits de maison, des épis de maïs... C'est pourquoi il est nécessaire de pouvoir visualiser ces points afin de s'assurer de leur cohérence et de pouvoir ôter le bruit.

Il y deux parties dans le travail de validation : l'une permettant la visualisation et la manipulation manuelle des points via un outil graphique (voir rapport dur le module éditeur) et l'autre permettant le traitement automatique de ces points (voir rapport sur le module de filtres).

Le présent rapport concerne la partie sur le filtrage de ces points.

1.2.1 Références

Ce rapport fait suite au travail d'analyse et de design réalisé autour du module de validation. Et plus particulièrement de la partie filtre de validation. Ces travaux ont fait l'objet des rapports suivants:

- Mise en place des filtres du module validation. Filtre du module validation - 18-05-2004.doc – 8pages
- Semi de points & éditeur. Résumé de réunion - 14-04-2004.doc. 8 pages.
- Module validation de données & Semi de points. Résumé de réunion - 27-04-2004.doc. 9 pages

Ces rapports s'appuient sur les spécifications fonctionnelles de Modeleur 2 :

- Spécifications – Modeleur 2.0. Québec, INRS-Eau, Terre & Environnement. 68 pages.

Ces travaux s'appuient aussi sur le module éditeur :

- Rapport final sur l'éditeur-1.4. 29 pages

Les travaux effectués s'appuient et concernent aussi le module Gestionnaire de base de donnée :

- \\Caxipi\Green\Rapports\Base de données

1.3 Structure du document

Plusieurs concepts ont été abordés au cours de l'élaboration du module de filtres et plusieurs modules sont impliqués dans sa réalisation. Chacun de ces concepts est repris et expliqué dans la suite de ce rapport.

2 Notions de bases pour le module de filtres

2.1 Semi de points

Un semi (de points) représente l'ensemble des points mesurés appartenant à un même projet.

2.2 Point (type de)

Si tous les points d'un même semi sont de même type, les points des différents semis peuvent êtres de types différents. C'est à dire qu'ils peuvent porter différents types d'informations. Certains types de points pourront, par exemple, contenir une information de pression, de température ou d'altitude. On parlera souvent de points de topo. Ce sont des points représentant une altitude. C'est avec des jeux de données de points de topo que le module de filtres a initialement été mis en place.

On parlera de points génériques lorsqu'on ne travaillera qu'avec les valeurs communes à tous les type de points (valeurs portées par la classe mère *VDPointGenerique*). On parlera de points spécialisés quand on travaillera avec les valeurs spécialisées des points. Par exemple, un point de topo est un point spécialisé.

3 Le module de filtres

Parmi les points dans les différents semis, nombreux sont ceux qui ne sont pas intéressants. Par exemple, dans le cadre des semis de topo les relevés incluent de nombreux points qui représentent autre chose que le sol, comme des maisons, véhicules arbres…

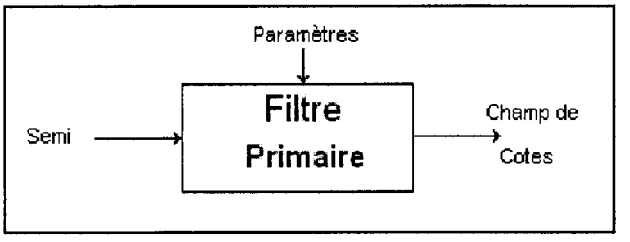
Il est nécessaire pour la suite des opérations de pouvoir distinguer les points intéressants du bruit. Les points étant de par la nature des relevés, très nombreux[1], il est alors impensable de laisser faire ce travail par l'utilisateur.

Le module de filtres propose une structure permettant d'accueillir des filtres qui seront chargés de distinguer ces points.

3.1 Filtres et combinaisons de filtre

Les filtres fonctionnent de la manière suivante : ils prennent en entrée un semi de points et ressortent une cote associée à chaque point du semi. Cette cote indique un niveau d'appartenance du point au sol.

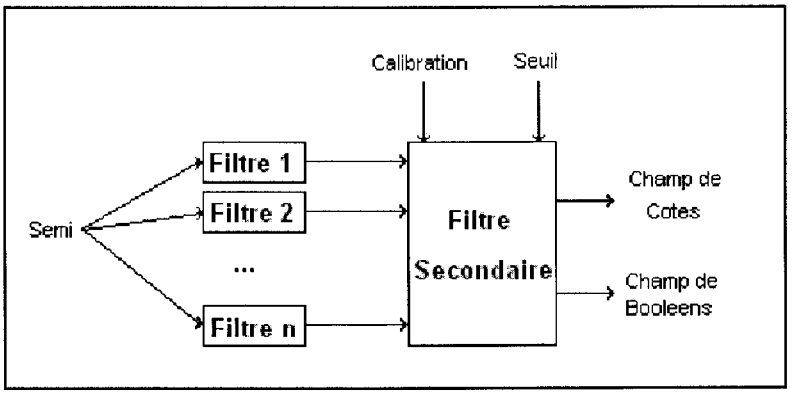
Une des idées importantes pour ce module est que l'on doit être capable de combiner les résultats des filtres entre eux. Cela est représenté par les 2 schémas suivants :



Exemple de filtre

Un filtre peut être quelque chose de simple, par exemple, il renvoie 0 pour tous les points d'altitude de plus de 1622 mètres[2] et 1 pour tous les autres. Les filtres peuvent être plus complexes notamment en se basant sur leur voisinage (les valeurs des points proches) et en combinant les différentes informations portées par les points.

[1] On parle en général de 1 point par mètre. Un rapide calcul nous montre que pour une zone de 10 km sur 10 km cela ne représente pas moins de 100 millions de points.
[2] le point culminant du Québec se trouve à 1622m (mont d'Iberville)



Exemple de combinaison de filtre

Les combinaisons de filtres permettent d'exécuter plusieurs filtres et de pondérer leurs sorties. Par exemple, un filtre marche bien pour les sous bois et que l'on est sur un terrain de type forêt on pourra lui donner plus d'importance qu'un autre filtre habituellement utilisé pour les terrains de type villes. Ces combinaisons peuvent êtres de simples pondérations ou peuvent être mises en place à partir de techniques plus évoluées comme de la logique floue, des réseaux de neurones, statistiques…

Les filtres primaires représentent les filtres de bases qui travaillent à partir de semis de points et les filtres secondaires sont les filtres qui vont travailler à partir des cotes, c'est à dire, du résultat des filtres primaires. Finalement, les filtres secondaires ressortiront un champ de booléens déterminé à partir des cotes finales et d'une valeur seuil.

3.2 Paramètres des filtres & XML[3]

Les filtres peuvent avoir besoin de paramètres, ces paramètres ne sont pas toujours de même type ni dans les mêmes quantités. Il est donc très difficile d'unifier les paramètres pour tous les filtres possibles.

Pour résoudre ce problème, il a été décidé de passer tous les paramètres sous forme d'une chaîne de caractère dans un format spécifique. Ce format pourrait être le format XML. Ce format permettrait entre autre de générer à la volée des boîtes de dialogues pour que l'utilisateur puisse choisir les différents paramètres et les différents filtres[4].

Voici un exemple de chaîne XML pour créer un filtre secondaire :

```
<FILTRESECONDAIRE>
<TYPEFILTRESECONDAIRE>DLL</TYPEFILTRESECONDAIRE>
<NOMFILTRESECONDAIRE>c:\\FiltreSecondaire.dll</NOMFILTRESECONDAIRE>
<NOMBREFILTRESPRIMAIRES>3</NOMBREFILTRESPRIMAIRES>

<FILTREPRIMAIRE>
      <TYPEFILTREPRIMAIRE>DLL</TYPEFILTREPRIMAIRE>
      <NOMFILTREPRIMAIRE>c:\\FPrimaire1.dll</NOMFILTREPRIMAIRE>
```

[3] Voir aussi rapport final sur le module de validation.
[4] C'est le principe expliqué dans le 1er rapport sur les filtres de validation.

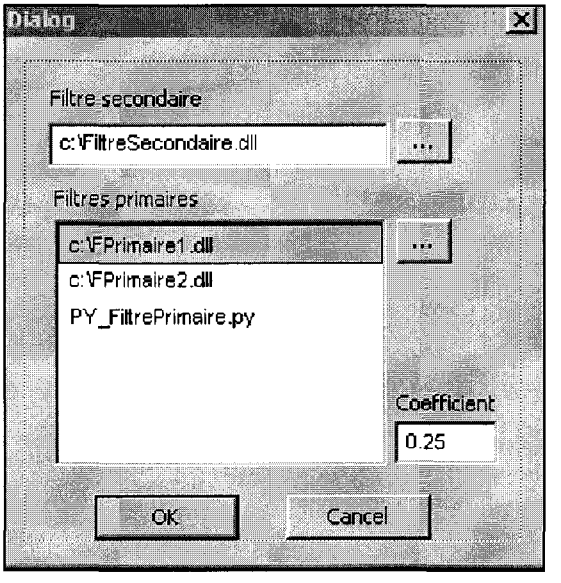
```
      <COEFFICIENT>0.25</COEFFICIENT>
</FILTREPRIMAIRE>

<FILTREPRIMAIRE>
      <TYPEFILTREPRIMAIRE>DLL</TYPEFILTREPRIMAIRE>
      <NOMFILTREPRIMAIRE>c:\\FPrimaire2.dll</NOMFILTREPRIMAIRE>
      <COEFFICIENT>0.75</COEFFICIENT>
</FILTREPRIMAIRE>

<FILTREPRIMAIRE>
      <TYPEFILTREPRIMAIRE>SCRIPTPYTHON</TYPEFILTREPRIMAIRE>
      <NOMFILTREPRIMAIRE>PY_FiltrePrimaire</NOMFILTREPRIMAIRE>
      <COEFFICIENT>0.3</COEFFICIENT>
</FILTREPRIMAIRE>

</FILTRESECONDAIRE>
```

Une telle chaîne pourrait être obtenue a partir d'une boîte de dialogue de ce type :



Exemple de boîte de dialogue pour l'application de filtres

3.3 Type des filtres

Les filtres doivent pouvoir être écrits par n'importe qui. Comme tout le monde ne dispose pas d'un compilateur C++, les filtres peuvent êtres écrits en python. Il est donc possible d'avoir des filtres qui sont des dll générés à partir de fichier C++ ou des scripts python.

Dans un cas comme dans l'autre le filtre doit hériter de *FVFiltrePrimaire* (ou *FVFiltreSecondaire*). La seule restriction est que ces filtres doivent implémenter la fonction *calcule()* qui est le point d'entrée et de sortie des filtres.

3.4 Design et fonctionnement du module de filtres

Le module de filtres s'appuie sur un double proxy[5] : un pour les filtres primaires et un pour les filtres secondaires.

Le principe de fonctionnement d'un proxy est le suivant : on dispose d'une classe mère dont héritent plusieurs classes dont une classe dite proxy. Cette classe proxy crée et contient une instance d'une des autres classes sous la forme classe mère. Ainsi, quand une des fonctions du proxy est appelé, il ne fait qu'appeler la fonction correspondante de la classe contenue.

Le but est ici d'avoir un proxy capable d'instancier des classes dérivant la classe mère et se trouvant dans des dll ou des fichiers de script python. Ainsi, il est possible de rajouter de nouveaux filtres sans recompiler Modeleur.

Le module de filtres utilise les structures de points du modules validation. Il utilise une tuile[6] spéciale qui représente tout l'espace sur lequel il désire travailler (cette tuile n'est pas forcément carrée ni forcément continue) et utilise les mêmes classes du gestionnaire de base de données pour charger cette tuile.

Le gestionnaire de ce module est chargé de créer un filtre secondaire puis de d'exécuter la fonction calcule permettant d'obtenir un résultat sous la forme d'un champ de cotes ou de booléens.

Il reste à faire la partie (interface utilisateur) permettant de construire la chaîne XML permettant d'instancier les différents filtres.

3.4.1 Filtres templates

Les filtres du module de filtres sont template d'un type de point. Les types de points sont les mêmes que ceux utilisés avec le module de validation.

Le fait que les filtres soient templates fait que les proxy sont en place une fois pour toutes. Seule la classe *FVGestionnaire* devrait subir des modifications par le futur.

3.4.2 Utiliser le module de filtres - Créer des filtres

Pour utiliser le module filtre, il faut avoir des filtres. Ceux-ci peuvent être des fichiers de script python ou des dll. Il faut au moins un filtre primaire et au moins un filtre secondaire, ceux-ci doivent hériter respectivement de *FVFiltrePrimaire* et de *FVFiltreSecondaire*.

Le filtre secondaire est chargé d'exécuter les filtres primaires, il peut ensuite effectuer différents calculs avec les résultats de ces filtres primaires.

[5] Voir le diagramme de classes disponible en annexes (§6).
[6] Voir le rapport final sur le module de validation.

Les filtres primaires sont chargés de calculer une cote pour chaque point d'un semi. Le semi est passé en paramètre lors de l'appel de la fonction calcule. La seule contrainte de cette classe est qu'elle doit retourner un vecteur de cotes de même taille que le vecteur de points.

3.5 Tests

Pour tester le module de filtre, 4 filtres ont été mis en place :

- un filtre secondaire en C++ qui ajuste les cotes en fonction d'un coefficient;
- deux filtres primaires en C++ qui affectent une cote en fonction de valeurs fixe ou de la valeur maximale d'altitude d'un semi ;
- un filtre primaire en python[7] qui affecte une cote en fonction de valeurs fixes.

Tous ces filtres sont disponibles sur CVS dans « Module_Filtres » dans le sous repertoire « filtre » et peuvent servir d'exemple pour la création de véritables filtres.

[7] Voir aussi l'exemple en annexe.

4 Conclusion

4.1 Résumé

Il y a 2 types de filtres : les filtres primaires travaillant à partir des semis et les filtres secondaires travaillant à partir des résultats des filtres primaires.

Les paramètres permettant d'instancier les filtres sont passés sous la forme d'une chaîne de caractères au format XML.

Les filtres peuvent êtres écrits par les utilisateurs en C++ ou en script python.

Le module de filtre suit le même design pour les filtres primaires que pour les filtres secondaires, à savoir le design du proxy.

4.2 Perspectives futures

4.2.1 Support des différents types de points

Une des contraintes du module de validation est de pouvoir gérer différents types de points. Pour le moment seuls les points de topo ont été utilisés. Pour utiliser d'autres points, il faut suivre la procédure suivante :

- Exporter le nouveau type de points à partir du *FVGestionnaire* comme c'est fait pour les points de topo. Ex : template DLL_IMPEXP(MODULE_FILTRES) VDPoint<VDTopo>;
- Ajouter dans le module gestionnaire de BD une classe GDAlgoIO-Nouveau_type_de_point en s'inspirant de GDAlgoIOTuileTopo et ajouter aussi un GDAlgoIOInfoSemi pour le nouveau type de point.
- Dans *FVGestionnaire* dans la fonction *traiteFVEvenEditionMenu*, faire la séparation entre les différents types de points.

Dans la base de données, ajouter une table pour les points de type nouveau et une table info pour ces points.

4.2.2 Filtres et Base de données

Une des réflexions qui reste à mener sur le module de filtres est son interaction avec la base de données.

- Comment sauvegarder les paramètres des filtres ?
- Comment assurer l'existence des filtres ?

Il peut être intéressant de sauvegarder les paramètres donnés à un filtre pour pouvoir, si ce sont de bons paramètres, les réutiliser plus tard.

Les filtres secondaires ont pour paramètres des noms de filtre primaires. Encore faut-il pouvoir assurer la présence de ces filtres sur la machine en cours d'utilisation.

Il doit être possible de sauvegarder les filtres dans la base de données, qu'ils soient sous forme de scripts python ou de dll. Il faudrait alors réussir à mettre en place un système qui fait qu'on ne peut pas supprimer un filtre si un autre dépend de lui.

5 Glossaire

Dll : Dynamic Linked Library. Bibliothèque de liens dynamiques. Ensemble de routines extraites d'un programme principal, pour être partagées par plusieurs programmes ou pour optimiser l'occupation de la mémoire (les DLL pouvant être chargées et déchargées à volonté). Utilisées essentiellement sous Windows désormais, autrefois utilisées par OS/2, elles peuvent aussi être exploitées dans certaines conditions sous Linux.

Filtre primaire : permet d'affecter une cote à chaque point d'un ensemble de points. Cette cote représente un degré d'appartenance au sol.

Filtre secondaire : permet de combiner les résultats de plusieurs filtres primaires entre eux afin d'affiner le résultat final.

Point : position géographique à laquelle peut être affecté un ensemble de valeurs.

Proxy : design pattern permettant de faire jouer le rôle d'un objet à un autre objet.

Python : langage de script, libre, cousin de Perl dans son esprit, mais beaucoup plus fortement orienté objet, dont le développement a débuté en 1990.

Semi : ensemble de point d'une même projection.

Template : technique de génération automatique de code en paramétrant des fonctions ou des classes.

XML : Extensible Markup Language. Langage de balises permettant de séparer le contenu de la présentation.

6 Annexes

6.1 Diagramme de classes

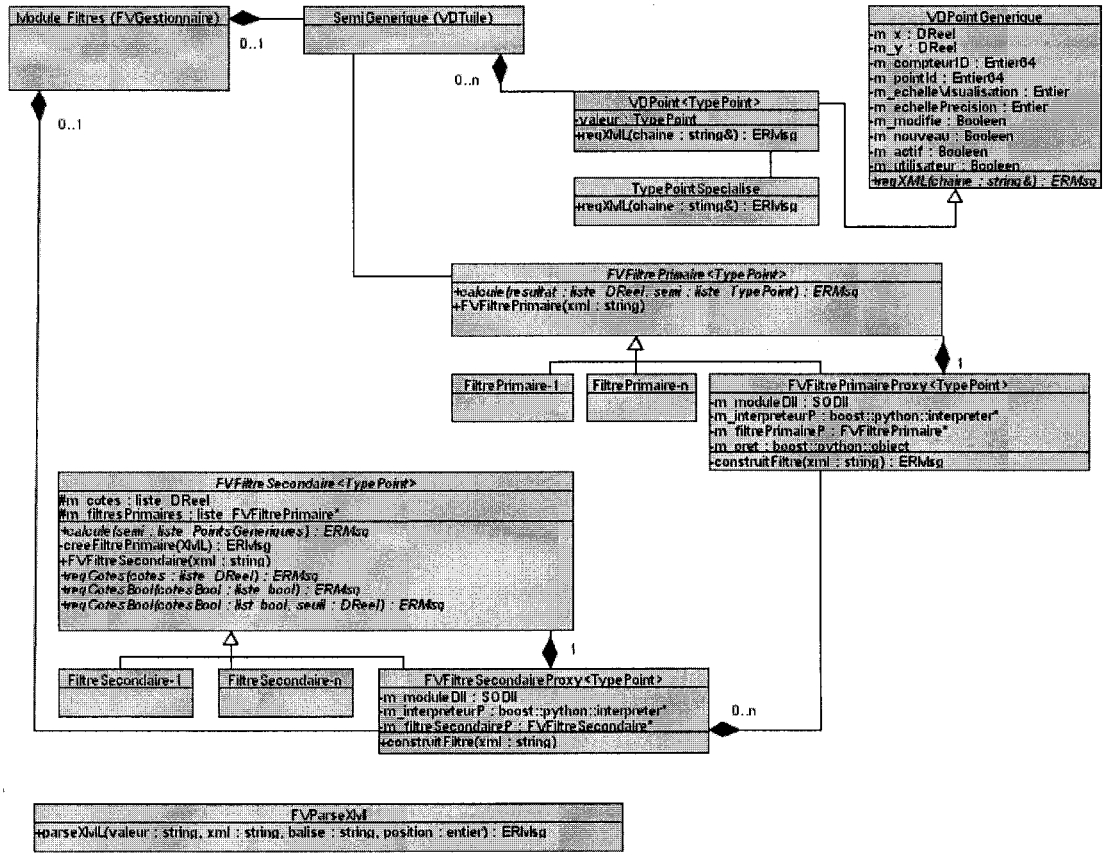


Diagramme de classes du Module de filtres

6.2 Exemple de filtre primaire en python

```
import PY_Module_Filtres_Points
from PY_Module_Filtres_Points import *
import PY_Erreur
from PY_Erreur import *
from Singleton import *
from wxPython.wx import *
import sys,os
import time
from string import *

# Point d'entrée du filtre
def creeFiltre(str):
```

```
  return PY_FiltrePrimaire(str)

# Filtre
class PY_FiltrePrimaire(FVFiltrePrimaireVDTopo):

  # Constructeur
  def __init__(self, str):
    FVFiltrePrimaireVDTopo.__init__(self)

  # Fonction calcule
  def calcule(self, cotes, points):
    print("Calcule du filtre Python")
    for point in points:
      tmp = point.m_values.m_z1
      if (tmp < 0):
        cotes.push_back(0.1)
      elif (tmp < 2000):
        cotes.push_back(0.7)
      else:
        cotes.push_back(0.05)
    retour = PY_Erreur.ERMsg(PY_Erreur.ERMsg.Type.OK)
    return retour
```

6.3 Classe VDTopo

La classe VDTopo contient les variable membres suivantes :

- DReel m_z1;
- DReel m_z2;
- DReel m_x2;
- DReel m_y2;
- DReel m_intensite1;
- DReel m_intensite2;

RAPPORT # 6 : Module de partition de maillage

Rapport de développement logiciel

Module partition de maillage

Présenté par :

Maude Giasson

18 janvier 2005

Versions du document

Date	Auteur	Commentaires	Version
11-08-2004	MG	Version initiale	V0

1. Introduction

1.1 Objectifs

Ce rapport a pour objectif d'introduire le module partition de maillage. Les principaux blocs formant ce module y sont expliqués, sans toutefois entrer dans les détails ni reproduire abusivement les informations déjà présentes dans les autres documents. Ce rapport a aussi comme objectif de relever les parties incomplètes et les faiblesses à améliorer pour parfaire ce module.

1.2 Contexte

Le module partition de maillage gère tout ce qui concerne l'activité de maillage. Au cœur du module, on retrouve les couches de maillage, qui portent l'information servant à mailler. Ces couches sont ordonnées au sein d'une partition de maillage. La partition de maillage et la couche de maillage sont des structures proches de la partition et de la couche déjà développées dans le cadre du MNT.

L'utilisateur peut construire une partition et lui ajouter des couches contenant divers paramètres de maillage. Ensuite, une ou plusieurs couches peuvent être maillées. Le résultat du maillage peut être affiché à l'écran ou enregistré dans la BD.

Les couches sont empilées selon un ordre de priorité. Les paramètres utilisés pour le maillage d'une section sont toujours pris sur la couche la plus prioritaire contenant la dite section. Il doit toutefois y avoir une continuité aux frontières entre les couches.

1.3 Structure

En premier lieu, ce rapport couvrira certains choix faits en cours de développement mais concernant plutôt l'interface. Il s'agit là d'un supplément aux spécifications fonctionnelles déjà existantes. En parallèle seront indiqués les choix ayant été mis en place et ceux n'étant toujours pas développés.

Ensuite, il sera question de la conception du module, en insistant sur les objets clés de ce dernier, du lien avec le module MNT, et des événements circulant à travers ce module.

Il sera plus loin question des principales classes constituant le module. On y retrouvera le détail des principaux algorithmes et méthodes.

Tout au long de ce document, des notes concernant l'état actuel du module et les améliorations à apporter ont été insérées. Toutefois, un document spécial [15] a été conçu afin de regrouper l'ensemble des notes particulières concernant le module partition de maillage et certains modules avec lesquels il interagit. La lecture de ce second document viendra donner un réel aperçu des prochaines étapes à franchir pour améliorer le module partition de maillage.

1.4 Références :

[1] Spécifications – Modeleur 2.0.
\\Caxipi\Green\Spécifications fonctionnelles\ Spécifications - Modeleur 2.0 - v1.35.pdf

[2] Rapport final : Modèle numérique de terrain (MNT)
\\Caxipi\Green\Rapports\MNT\Rapport final\ Rapport final2 - MNT.doc

[3] Rapport de réunion : Partition de maillage, 30-03-2004
\\Caxipi\Green\Rapports\Maillage\Partition de maillage - 30-03-2004 - v2.doc

[4] Rapport de réunion : Partition de maillage – réunion 2, 20-04-2004
\\Caxipi\Green\Rapports\Maillage\Partition de maillage - réunion 2 - 02-04-2004 - v2.doc

[5] Rapport de réunion : Partition de maillage – réunion 3, 19-04-2004
\\Caxipi\Green\Rapports\Maillage\Partition de maillage - réunion 3 - 19-04-2004 - v3.doc

[6] Rapport de réunion : Partition de maillage – réunion 4, 27-04-04
\\Caxipi\Green\Rapports\Maillage\Réunion 4\Partition de maillage - réunion 4 - 26-04-2004 - v2.doc

[7] Rapport de réunion : Partition de maillage – réunion 5, 03-04-04
\\Caxipi\Green\Rapports\Maillage\Partition de maillage - réunion 5 - 03-04-2004 - v2.doc

[8] Rapport de réunion : Partition de maillage – réunion 6, 10-04-04
\\Caxipi\Green\Rapports\Maillage\Réunion 6\Partition de maillage - réunion 6 - 10-04-2004 - v1.doc

[9] Rapport de réunion : Objets graphiques des MNT et des partitions de maillage, 14-06-04
\\Caxipi\Green\Rapports\Maillage\Réunion 7 - Objets graphiques des MNT et des partitions de maillage- 14-06-2004 - v3.doc

[10] Rapport de réunion : PMMailleurFactory, 18-06-2004
14-06-04
\\Caxipi\Green\Rapports\Maillage\Réunion 8 - PMMailleurFactory - 18-06-2004 - v2.doc

[11] Rapport de réunion : Nouvelles classes d'affichage, 07-07-2004
\\Caxipi\Green\Rapports\Maillage\Réunion 9\Réunion 9 - Nouvelles classes d'affichage - 7-06-2004 - v2.doc

[12] Rapport de réunion : Géométries – passage à GEOS, 21-06-2004
\\Caxipi\Green\Rapports\Algorithmes, SRChampVector, Isolignes et Isosurfaces\
Géométries - passage à GEOS - 21-06-2004 - v2.doc

[13] Rapport de développement logiciel : Rapport sur le module de filtres
\\Caxipi\Green\Rapports\Validation\Rapport final sur le module de filtres-1.2.pdf

[14] Rapport de développement logiciel : Rapport sur le module de validation
\\Caxipi\Green\Rapports\Validation\Rapport final sur le module validation-1.2.pdf

[15] Module partition de maillage : Les prochaines étapes, 13-08-2004
\\Caxipi\Green\Rapports\Maillage\Module Partition de maillage - Prochaines étapes - 13-08-2004.doc

[16] Ressource web : GEOS
http://geos.refractions.net/

2 Suppléments aux spécifications fonctionnelles

2.1 Partition de maillage

La partition de maillage est une structure permettant d'effectuer un travail de maillage sur une région. Elle est composée de diverses couches de maillages empilées selon leur niveau de priorité. Elle contient aussi des paramètres de maillage propres à tout le domaine comme le nom du mailleur.

Une partition de maillage doit pouvoir être sauvegardée, sauvegardée sous un autre nom ou effacée.

On peut ouvrir une partition de maillage existante ou bien créer une nouvelle partition. Il est aussi possible d'ajouter une nouvelle couche à la partition de maillage ou d'ajouter une couche déjà existante. Finalement, on peut demander de mailler la partition en entier et voir le résultat à l'écran. On doit aussi pouvoir mailler une partition et enregistrer le résultat.

Il peut y avoir plusieurs partitions d'ouvertes en même temps.

Il doit être possible de modifier les paramètres portés par une partition.

2.2 Couche de maillage

Une couche de maillage possède un domaine sur lequel elle définit des paramètres de maillage, tel un champ d'erreur et un niveau de tolérance.

Il est possible d'ouvrir une nouvelle couche ou de charger une couche existante. Dans les deux cas, la couche est ajoutée à la partition sélectionnée. La nouvelle couche est affichée ; son affichage ne concerne que le contour car à ce stade, elle n'a pas encore été maillée.

On doit pouvoir changer le niveau de priorité sur une couche, sauver une couche ou la sauver sous un autre nom.

On peut sélectionner une couche et demander que cette couche soit maillée. Dans ce cas, seule la partie visible de la couche (non recouverte par d'autres couches de la partition) est maillée. Ce maillage est fait en tenant compte des autres couches de la partition. Des nœuds sont forcés aux endroits (visibles) où la couche croise les autres couches et les paramètres tiennent compte des couches plus prioritaires adjacentes. Ainsi, si une couche A est maillée et une couche B est maillée par la suite, le maillage de B sera continu avec le maillage de A, et ce, sans que A n'aie à être remaillée.

Les couches de maillages sont en mode copie. La même couche ne peut pas se retrouver à la fois dans deux partitions différentes.

Lors d'un double-clic sur une couche, cette dernière passe en édition, il est alors possible de modifier sa géométrie. Une seule couche est éditée à la fois.

Il devra être possible de modifier les paramètres contenus par la couche.

2.3 Sélection

Il est possible de sélectionner une couche ou une partition. Les actions ont toujours lieu sur la sélection.

Par exemple, si l'on demande d'ajouter une couche, alors la couche sera ajoutée à la partition sélectionnée.

2.4 Affichage

2.4.1 GUI

Chaque partition de maillage s'affiche dans sa propre fenêtre. Toutefois, d'autres éléments graphiques, provenant d'autres modules, peuvent être affichés dans une fenêtre contenant une partition de maillage. Le GUI entretient un arbre indiquant les partitions ouvertes ainsi que les couches que ses dernières contiennent.

Lorsqu'une partition est sélectionnée, on voit l'ensemble des couches, empilées selon leur ordre de priorité. Il en est de même si une couche est sélectionnée.

2.4.2 Édition

Lors du double-clic sur une couche, cette dernière passe en édition. Sa couleur est alors modifiée et, visuellement, elle passe sur le dessus de la partition. Il est alors possible de modifier son domaine. En réalité, elle conserve sa vraie priorité. Elle ira reprendre sa place dans l'empilement de couches dès qu'elle perdra le focus.

2.4.3 Affichage des maillages

Aux domaines des couches se superposent les maillages. Lorsqu'une action « maille » a lieu sur une couche, le maillage résultant est affiché sur cette dernière. On conserve à l'écran deux maillages par couche, soit le précédent et le maillage valide, représentant les paramètres portés par la couche. Lorsqu'un maillage devient invalide, sa couleur est modifiée.

Il devra y avoir deux modes de travail. Un mode géométrique et un mode paramètres. Lors d'un travail en mode géométrique, on pourra modifier la géométrie des couches, en enlever et en ajouter, mais on ne pourra pas mailler, ni voir des maillages. En mode « paramètres », il sera possible de mailler des couches, de voir les maillages (valide et précédent) et de modifier les paramètres. Dans ce second mode, il sera possible de voir les effets des modifications sur les maillages à l'aide de l'affichage du maillage précédent.

2.4.4 Liste d'affichage

Les deux objets graphiques de maillage portés par chacune des couches se retrouveront dans la liste d'affichage. Il sera alors possible de mettre visibles ou invisibles certains maillages. Par exemple, pour certaines couches, on pourrait souhaiter retirer le maillage précédent pour ne voir que le maillage valide.

Concernant les domaines, une fenêtre contenant une partition de maillage contiendra un seul objet dans sa liste d'affichage représentant les contours des couches de la partition. Cet objet graphique pourra toutefois être explosé en divers petits objets (un par couche). Ainsi, il sera possible de faire une modification sur la visibilité de la partition entière ou bien sur la visibilité d'un élément de la partition.

2.4.5 Gestion de la caméra

La caméra devra être fixée selon le repère du projet et ne pouvoir être bougée que suite à une action explicite faite par l'utilisateur.

2.4.6 Ajout de nouvelle couche

Une nouvelle couche ajoutée ne comporte pas de domaine en réalité. Un domaine par défaut pourra être assigné aux nouvelles couches afin d'avoir une base sur laquelle construire. Ce pourra être, par exemple, un carré au centre du repère affiché par la caméra.

2.5 État actuel

2.5.1 Partitions et couches

Des partitions de maillage peuvent être créées ou chargées. Toutefois, au niveau de la base de données, le chargement hard-code la création d'une partition puisque actuellement, aucune partition n'est enregistrée dans la BD. De même, il est possible d'ajouter à la partition une nouvelle couche ou une couche existante. Toutefois, lors de l'ajout de couche existante, la couche ajoutée est hard-codée dans le gestionnaire de données.

Il peut y avoir plusieurs partitions d'ouvertes en même temps.

Le niveau de priorité d'une couche ne peut pas encore être changé.

Il n'est pas possible de modifier les paramètres d'une partition ou d'une couche, ni de sauvegarder ou supprimer une partition. Le seul mailleur 1D existant est trivial. Par mailleur 1D trivial, on entend que le seul travail fait par ce mailleur est de former un élément L2 à partir de l'arête à mailler. Le seul mailleur 2D est le mailleur frontal, n'utilisant ni champ d'erreur ni niveau de tolérance. Ce sont les mailleurs contenus par chaque partition et chaque couche.

2.5.2 Maillage et affichage du maillage

On peut demander de mailler la couche sélectionnée ou la partition en entier, mais seulement pour diriger le résultat vers l'écran. On n'enregistre pas de résultat de maillage dans la BD.

La structure pour gérer la continuité des maillages est en place dans l'algorithme. Certains cas échappent encore pour faute de précision. Il faudra donc renforcir les géométries.

On peut voir à l'écran le maillage valide et le maillage précédent pour chacune des couches. Les cas où le maillage est invalidé ne sont pas encore bien établis. La gestion des deux modes (géométrique, paramètres) n'est pas encore en place. En conséquence, on peut déplacer une couche qui porte un maillage. Le maillage n'est pas déplacé, ce qui alourdit la compréhension.

2.5.3 Sélection, GUI, édition, liste d'affichage et caméra

La gestion du GUI et la sélection fonctionnent bien. Les couches peuvent être éditées et y subir des modifications géométriques. Lors d'une série de clics rapides sur une même couche, cette dernière disparaît de l'écran ; il s'agit d'un bug à corriger.

Aucune liste d'affichage n'est en place.

La caméra est centrée sur les acteurs de la fenêtre. À chaque modification géométrique la caméra se replace automatiquement, ce qui n'est pas souhaitable.

Lors de l'ajout d'une nouvelle couche, un domaine est hard-codé à cette dernière dans le gestionnaire de BD. Ce n'est pas fonction de la zone d'affichage.

3 Lien avec le Module MNT

Le lien unissant le module partition de maillage au module MNT est particulier. Ces deux modules n'interagissent pas entre eux; ils ne s'échangent aucun événement. Toutefois, la conception de ces deux modules est très proche. Les classes d'événements et le gestionnaire sont similaires. La PMInfoPartition, la PMPartition, la PMInfoCouche et la PMCouche héritent respectivement de MTInfoPartition, MTPartition, MTInfoCouche et MTCouche.

Les classes du MNT ont été reprises dans la partition de maillage afin de ne pas dupliquer tout le travail. Ainsi, c'est au niveau des classes du MNT que se situent les containers, les PM n'étant là que pour ajouter les attributs supplémentaires devant être portés et certaines restrictions. Les restrictions servent à limiter le travail des MT. Par exemple, on ne peut pas assigner n'importe quelle MTPartition à une PMInfoPartition. La PMInfoPartition s'assure que la MTPartition assignée est, en réalité, une PMPartition.

Le fait de réutiliser les classes du MNT et ce, dans un exemple d'application plus concret a permis de faire ressortir les forces et les faiblesses de ce module.

Lors de la conception de la partition de maillage, le double chaînon

InfoPartition-Partition
InfoCouche-Couche

a suscité certaines interrogations.

D'abord, le discernement entre ce qui est « info » et ce qui ne l'est pas a dû être fait. Pour faire ce discernement, il faut comprendre la raison de cette séparation. Des partenaires externes pourraient souhaiter utiliser une partie du code développé dans Modeleur 2. Par exemple, ils pourraient souhaiter profiter de la structure du MNT. Afin de pouvoir échanger les classes du MNT, il est nécessaire d'avoir une structure contenant toute l'information vitale, nécessaire pour effectuer le travail de base d'un MNT, mais rien de plus. Cette structure devra, par exemple, être indépendantes des considérations d'affichage, qui elles sont fortement couplées à Modeleur 2.

C'est pour cette raison qu'une partition « pure » a été développée. Cette partition est un container de couches « pures ».

Au niveau de Modeleur 2, l'utilisation de ces couches et de ces partitions nécessite certaines informations supplémentaires. C'est pourquoi ces informations ont été regroupées dans le deuxième chaînon qu'est celui des « info ». Cette dualité permet d'avoir une structure pure, tout en conservant un surplus d'information dans les infos. Ainsi, tous les attributs n'étant pas vitaux pour le travail de la partition et de la couche doivent se retrouver au sein des infos.

Bien qu'elle ait passé le test qu'est celui de se faire spécialiser par la partition de maillage, cette conception à chaînon double comporte certains défauts. Par exemple, la symétrie entre le container d'infoCouche contenu par un infoPartition et le container de couche contenu par la partition est difficile à garantir.

En effet, l'infoPartition donne accès à sa partition par la méthode reqPartition(). À partir de là, il est possible d'ajouter une couche à la partition, sans avoir ajouté l'infoCouche correspondante à l'infoPartition. Cela a pour effet de dégénérer l'infoPartition.

Il y a eu un début de réflexion dans le but d'améliorer ce modèle. Cela n'a pas mené a des résultats tangibles. Il faudra revoir ce modèle afin de l'améliorer ou bien s'assurer que ses limites de validité et que son contexte d'utilisation sont bien définis

4 Liens avec les autres modules

Le module partition de maillage échange des événements intéressant particulièrement deux modules. Il s'agit du module gestionnaire de données et du module d'affichage. Il entretient aussi certaines discussions avec le module éditeur.

Cette section a pour objectif de faire ressortir les relations avec les autres modules. On peut se référer aux rapports de réunion pour avoir une idée de la manière dont circulent les événements.

4.1 Module gestionnaire de données

Le module partition de maillage envoie des événements pour le chargement et la création d'objets. Ces événements intéressent le module gestionnaire de données qui est responsable de créer, de charger et décharger les objets ainsi que d'entretenir un compte de référence sur ces derniers.

Cette section vise à indiquer les principaux objets que le module gestionnaire de données charge/créée et décharge pour le module partition de maillage, et, surtout, à préciser le moment où ces objets sont chargés/créés et déchargés.

Les principaux objets faisant partie de la partition de maillage sont décrits dans la section 5. Voici toutefois certaines informations qui aideront à comprendre la balance de la présente section :

- Une PMInfoPartition est un container de PMInfoCouche.
- La PMInfoCouche hérite de MTInfoCouche. MTInfoCouche contient un AFObjetGraphiqueSRChampVector représentant son contour.
- La PMInfoCouche contient deux PMObjetGraphiqueMaillageVariable.
- À l'ajout d'une partition, un AFObjetGraphiqueComposite est associé à la partition au sein du gestionnaire. Seront placés dans ce composite les AFObjetGraphiqueSRChampVector représentant les contours des domaines des couches de la partition.

4.1.1 Principaux gains de compte de référence

4.1.1.1 PMInfoCouche

Le module partition de maillage prend un compte de référence sur une PMInfoCouche dans deux cas :

- Lorsqu'il envoie un événement de chargement/de création de PMInfoCouche. Un tel événement est envoyé lorsqu'une PMInfoCouche est ajoutée à une PMInfoPartition.
- Lorsqu'il demande le chargement d'une PMInfoPartition et que cette PMInfoPartition contient des PMInfoCouche. En fait, il prend alors un compte de référence pour chacune des PMInfoCouche contenue par la PMInfoPartition.

4.1.1.2 PMInfoPartition

Le module partition de maillage prend un compte de référence sur une PMInfoPartition dans un seul cas :

- Lorsqu'il envoie un événement de chargement/de création de PMInfoPartition. Cela se produit à l'ouverture d'une PMInfoPartition.

4.1.1.3 PMObjetGraphiqueMaillageVariable

Le module partition de maillage prend un compte de référence sur un PMObjetGraphiqueMaillageVariable dans deux cas :

- Lorsqu'il envoie un événement de chargement/de création de PMInfoCouche. En fait, il prend alors un compte de référence pour chacun des deux

PMObjetGraphiqueMaillageVariable contenu par la PMInfoCouche. Cela se produit lorsqu'une PMInfoCouche est ajoutée à une PMInfoPartition.

- Lorsqu'il envoie un événement de chargement de PMInfoPartition. En fait, il prend alors un compte de référence pour chacun des deux PMObjetGraphiqueMaillageVariable contenu par chacune des PMInfoCouche de la PMInfoPartition chargée. Cela se produit lorsqu'une PMInfoPartition est ouverte.

4.1.1.4 AFObjetGraphiqueSRChampVector

Le module partition de maillage prend un compte de référence sur un AFObjetGraphiqueSRChampVector dans deux cas :

- Lorsqu'il envoie un événement de chargement/de création de PMInfoCouche. En fait, il prend alors un compte de référence sur l'AFObjetGaphiqueSRChampVector contenu dans la MTInfoCouche, classe de base de la PMInfoCouche. Cela se produit à l'ajout d'une PMInfoCouche à une PMInfoPartition.
- Lorsqu'il envoie un événement de chargement de PMInfoPartition. En fait, il prend alors un compte de référence sur chacun des AFObjetGrpahiqueSRChampVector contenu dans chacune des PMInfoCouche de la PMInfoPartition chargée. Cela se produit à l'ouverture d'une PMInfoPartition existante.

4.1.1.5 AFObjetGraphiqueComposite

Le module partition de maillage prend un compte de référence sur un AFObjetGraphiqueComposite dans un seul cas :

- Lorsqu'il envoie un événement de création d'AFObjetGraphiqueComposite. Un tel événement est envoyé à l'ajout d'une PMInfoPartition

4.1.2 Principales pertes de compte de référence

4.1.2.1 PMInfoCouche

Le module partition de maillage perd un compte de référence sur une PMInfoCouche dans deux cas :

- Lorsqu'il envoie un événement de déchargement de PMInfoCouche. Un tel événement est envoyé lorsqu'une PMInfoCouche est retirée à une PMInfoPartition.
- Lorsqu'il demande le déchargement d'une PMInfoPartition et que cette PMInfoPartition contient des PMInfoCouche. En fait, il perd alors un compte de référence pour chacune des PMInfoCouche contenue par la PMInfoPartition. Cela se produit à la fermeture d'une PMInfoPartition.

4.1.2.2 PMInfoPartition

Le module partition de maillage perd un compte de référence sur une PMInfoPartition dans un seul cas :

- Lorsqu'il envoie un événement de déchargement de la PMInfoPartition. Cela se produit à la fermeture d'une PMInfoPartition.

4.1.2.3 PMObjetGraphiqueMaillageVariable

Le module partition de maillage perd un compte de référence sur un PMObjetGraphiqueMaillageVariable dans deux cas :

- Lorsqu'il envoie un événement de déchargement de PMInfoCouche. En fait, il perd alors un compte de référence pour chacun des deux PMObjetGraphiqueMaillageVariable contenu par la PMInfoCouche. Cela se produit lorsqu'une PMInfoCouche est retirée d'une PMInfoPartition.
- Lorsqu'il envoie un événement de déchargement de PMInfoPartition. En fait, il perd alors un compte de référence pour chacun des deux PMObjetGraphiqueMaillageVariable contenu par chacune des PMInfoCouche de la PMInfoPartition déchargée. Cela se produit lorsqu'une PMInfoPartition est fermée.

4.1.2.4 AFObjetGaphiqueSRChampVector

Le module partition de maillage perd un compte de référence sur un AFObjetGraphiqueSRChampVector dans deux cas :

- Lorsqu'il envoie un événement de déchargement de PMInfoCouche. En fait, il perd alors un compte de référence sur l'AFObjetGaphiqueSRChampVector contenu dans la MTInfoCouche, classe de base de la PMInfoCouche. Cela se produit au retrait d'une PMInfoCouche à une PMInfoPartition.
- Lorsqu'il envoie un événement de déchargement de PMInfoPartition. En fait, pour chacune des PMInfoCouche contenue dans la PMInfoPartition, il perd un compte de référence l'AFObjetGraphiqueSRChampVector contenu la PMInfoCouche.

4.1.2.5 AFObjetGraphiqueComposite

Le module partition de maillage perd un compte de référence sur un AFObjetGraphiqueComposite dans un seul cas :

- Lorsqu'il envoie un événement de déchargement de AFObjetGraphiqueComposite. Un tel événement est envoyé suite à la fermeture d'une PMInfoPartition. L'AFObjetGraphiqueComposite envoyé à déchargé doit avoir été préalablement vidé de ses AFObjetGraphique par le gestionnaire de partition de maillage.

4.1.3 État actuel

Les méthodes concernant le retrait ne sont pas encore en place. Ainsi, on ne retire jamais de couches ou de partitions. En conséquence, on ne retire pas d'objets graphiques non plus.

La création de l'AFObjetGraphiqueComposite ne passe pas encore via le module gestionnaire de données.

Des GDAlgoIo ont été créés dans le gestionnaire afin de pouvoir répondre aux événements de création et de chargement de PMInfoCouche et PMInfoPartition. Toutefois, la dynamique interne de ces algorithmes n'est pas faite, on a plutôt recourt à du hard codage. Il faudra revoir le module gestionnaire de données pour bien créer et charger les objets utilisés par le module partition de maillage.

4.2 Module d'affichage

Presque toutes les actions ayant lieu sur le module partition de maillage impliquent une modification de l'affichage. Le module partition de maillage et le module d'affichage sont donc en constante communication.

4.2.1 Affichage/retrait de l'affichage

Il a été mentionné à la section précédente que lors du chargement ou de la création d'une infoCouche, celle-ci arrive au module partition maillage avec trois objets graphiques déjà créés/chargés. Bien qu'ils soient créés/chargés, ces objets graphiques ne sont pas encore affichés. Le PMGestionnaire doit donc s'occuper de demander l'affichage de ces trois objets graphiques.

Les deux maillages (précédent, valide) sont illustrés par des PMObjetGraphiqueMaillageVarialbe. Au départ, ces objets graphiques ne contiennent rien à afficher. Ils sont tous de même ajoutés au module d'affichage dès la création de l'infoCouche. Ils contiendront un affichage plus tard (lorsque la couche sera maillée).

Le contour de l'infoCouche est illustré par le AFObjetGraphiqueSRChampVector contenu dans la MTInfoCouche. Cet AFObjetGraphiqueSRChampVector n'est pas envoyé directement au module d'affichage. Il est plutôt ajouté à l'intérieur du AFObjetGraphiqueComposite associé à l'infoPartition.

Bref, seulement pour l'ajout d'une infoCouche, le PMGestionnaire envoie au module d'affichage trois événements. D'abord, il y a deux événements pour l'affichage des PMObjetGraphiqueMaillageVariable (les deux maillages), puis un événement pour ajouter l'AFObjetGraphiqueSRChampVector (représentant le contour) à l'AFObjetGraphiqueComposite associé à l'infoPartition où est ajoutée l'infoCouche.

Pour être symétrique, lorsqu'une infoCouche sera retirée, trois événements devront être envoyés afin que le PMGestionnaire retire les objets graphiques de l'affichage. Cela n'est pas encore fait.

Lorsqu'une infoPartition est créée ou chargée, plusieurs objets graphiques doivent aussi être affichés. Le travail est similaire à celui fait lors de l'ajout d'une infoCouche, mais avec répétitions.

4.2.1.1 État actuel

Lorsqu'il reçoit un objet graphique à afficher, le module d'affichage doit prendre un compte de référence sur cet objet. Ainsi, il doit charger cet objet via le module gestionnaire de données. Cette dynamique n'est pas encore en place mais devra nécessairement être faite pour assurer le bon fonctionnement lors du partage d'information entre le module d'affichage et le module partition de maillage.

4.2.2 Synchronisation

Bien qu'il ait été développé en parallèle au module partition de maillage, le concept de synchronisation relève définitivement du module affichage. Il ne sera donc pas question ici de ce concept. Pour plus de clarifications, voir [9] et [11].

4.2.2.1 Lors du maillage

Lors d'une action de maillage, les contours ne sont pas modifiés. Toutefois, les PMObjetGraphiqueMaillageVariable portés par les infoCouches sont modifiés. En effet, le maillage valide passe dans le maillage précédent pour dorénavant afficher le nouveau maillage obtenu. Le PMGestionnaire envoie donc des événements afin que le module d'affichage synchronise les objets graphiques concernés.

De la même manière, à chaque fois que le maillage d'une infoCouche est invalidé, les deux objets graphiques (maillage valide et maillage précédent) sont synchronisés. Pour l'instant, des invalidations de maillage ont lieu lors de l'ajout d'une infoCouche et lors du maillage. Il sera question plus amplement de l'invalidation de maillage dans la section 5.3.2 concernant le démailleur.

4.2.2.2 Lors de l'édition

Le cas de l'édition est particulier. Lors d'un double-clic sur une couche, cette dernière passe en édition. Il n'est pas souhaitable d'avoir deux affichage de la couche, soit celui contrôlé par l'éditeur (dynamique) et celui contrôlé par le module d'affichage (statique). Ainsi, lorsqu'une couche passe en édition, elle devient invisible pour le module d'affichage. Son affichage est en fait géré par l'éditeur seulement. De la même manière, lorsqu'une couche cesse d'être éditée, elle doit redevenir visible pour le module d'affichage. Le module d'affichage doit aussi synchroniser l'objet graphique afin qu'il représente bien le nouveau domaine de la couche. En effet, la couche a pu être modifiée lors de son passage en édition.

Bref, lorsqu'une couche gagne le focus, le PMGestionnaire met l'AFObjetGraphiqueSRChampVector associé à cette couche invisible. Il demande ensuite au module d'affichage de synchroniser cette couche, ce qui la rend effectivement invisible au niveau du module d'affichage.

Ensuite, lorsque la couche perd le focus, le PMGestionnaire met l'AFObjetGraphiqueSRChampVector associé à cette couche visible. Il demande au module d'affichage de synchroniser cette couche. Cette synchronisation a non seulement pour effet de rendre l'objet graphique à nouveau visible, mais permet aussi de reconstruire l'acteur en fonction de la nouvelle configuration du SRChampVector représentant le domaine de la couche.

Actuellement, le focus est principalement géré par le GUI. Ce sont les actions de souris dans l'arborescence qui impliquent l'envoi d'événements de perte et de gain de focus. Il serait bien de réfléchir à une manière plus fiable de gérer le focus, ou bien de garantir la fiabilité de la méthode actuelle. Par exemple, il faudrait considérer ce qui se passe lorsque

la même partition gagne et perd le focus ou bien lorsqu'on travaille dans plusieurs fenêtre à la fois.

4.2.2.3 Lors du changement de niveau de priorité

Le module partition de maillage ne permet pas encore de modifier le niveau de priorité d'une couche. Ainsi, les nouvelles couches sont toujours placées sur le dessus.

Lorsqu'il sera possible de modifier le niveau de priorité d'une couche, il faudra, en parallèle créer au sein du module d'affichage une structure permettant de définir un AFObjetGraphiqueComposite où les acteurs ont des priorités. Ainsi, lorsqu'une couche deviendra plus prioritaire, son AFObjetGraphiqueSRChampVector sera déplacé au sein de l'AFObjetGraphiqueComposite afin d'y être aussi plus prioritaire. Il faudra compléter le module d'affichage en conséquence. Il se peut que le module d'affichage aie à détruire et reconstruire sa liste d'acteurs afin de les réinsérer dans le nouvel ordre souhaité.

5 Principales classes du module partition de maillage

5.1 Gestion

5.1.1 Événements

Les classes d'événements sont toutes regroupées au sein du fichier PMEvenement. Les événements ont lieu sur une sélection simple ou une sélection multiple. Une sélection simple est un duo PMInfoPartitionP-PMInfoCoucheP. Pour sa part, la sélection multiple est tout simplement une liste de sélections simples.

Il y a trois principaux groupes d'événements. Les événements simples, les événements d'ajout et l'événement maille.

5.1.1.1 Événements simples

La classe PMEvenActionSimple représente un ensemble d'action de base pouvant avoir lieu sur les sélections simples. L'attribut *m_typeAction* d'identifier l'action devant avoir lieu sur la sélection. Les actions disponibles sont supprimer, retirer, sauver, gagner le focus, perdre le focus et changer de niveau de priorité.

5.1.1.2 Événements d'ajout

Il y a deux classes d'événements d'ajout, soient PMEvenAjouterCouche et PMEvenAjouterPartition. Dans les 2 cas, on peut ajouter un nouvel élément (créer) ou bien ajouter un élément existant (charger). L'attribut *m_typeAction* permet de discerner les événements de création des événements de chargement. Un attribut contenant l'id de l'objet à charger a été placé pour les cas de chargements. Pour sa part, la sélection simple représente la structure dans laquelle on souhaite ajouter un nouvel élément. Par exemple, si on souhaite ajouter une couche à une partition, la sélection représentera la partition à laquelle ajouter une couche.

Le cas d'ajout de partition est particulier. Qu'il s'agisse d'une création ou d'un chargement, la partition doit être associée à une fenêtre. L'événement contient l'id de fenêtre où ajouter la partition. De plus, le PMGestionnaire doit assigner à l'événement la nouvelle partition créée ou ajoutée. Cette dernière servira pour le rafraîchissement de l'arborescence du GUI.

5.1.1.3 Événement maille

L'événement de maillage est exprimé au sein de la classe PMEvenMailler. La sélection contenue par l'événement maille une sélection multiple représentant les couches à mailler ou la partition à mailler. Le PMEvenMailler fournit des méthodes permettant d'itérer sur les sélections simples comprises dans la sélection multiple.

On peut souhaiter mailler pour obtenir un résultat visuel à l'écran ou bien mailler pour enregistrer le résultat dans la BD. L'attribut *m_typeAction* permet de définir s'il faut afficher ou enregistrer le résultat.

Un attribut *m_maillageEnregistrementP* fait partie de l'événement maille. Il n'est utilisé que dans le cas d'un événement de maillage avec enregistrement. Dans ce cas, PMGestionnaire doit s'assurer d'enregistrer le maillage calculé dans l'événement.

5.1.2 Gestionnaire

Le PMGestionnaire est la classe responsable de recevoir les événements intéressant la partition de maillage, responsable de lancer les actions commandées par les événements et, au besoin, d'envoyer des événements sur le bus afin de se faire aider par d'autres modules dans la réalisation de son travail.

Le PMGestionnaire doit gérer plusieurs partitions de maillage à la fois. Il stocke donc ces partitions dans son attribut *m_mapFenetrePartition*. Ce map fait le lien entre un id de fenêtre et une structure d'information concernant la partition.

La structure d'information concernant la partition est la suivante :

```
 Struct InfoPartitionCoucheVtk
{
    PMInfoPartitionP          infoPartitionP ;
    PMInfoCoucheP             infoCoucheP ;
    AFObjetGraphiqueCompositeP objetVtkP ;
    EDContainerSRChampVectorP editionP ;
} ;
```

L'infoPartitionP de cette structure pointe vers la partition concernée par la structure. Pour sa part, l'infoCoucheP pointe vers la couche ayant le focus dans la partition. Ce pointeur est NUL si c'est la partition entière qui a le focus.

À chaque infoPartition est associé un objetVtkP qui est un AFObjetGraphiqueCompositeP. Ce composite contient un ensemble de

AFObjetGraphiqueSRChampVector où chaque AFObjetGraphiqueSRChampVector représente le domaine d'une des couches contenues par la partition.

L'EDContainerSRChampVectorP contient ce qui est présentement en édition pour cette partition.

5.2 Conteneurs

Il a déjà été question du double chaînon :

PMInfoPartition-PMPartition
PMInfoCouche-PMCouche

et de sa relation avec les classes du MNT. Ici, on se contente de discuter de certaines particularités propres à ces classes.

5.2.1 PMInfoPartition

La PMInfoPartition ne définit pas de nouveaux attributs par rapport à sa classe de base (MTInfoPartition). Toutefois, elle redéfinit le constructeur afin de s'assurer que la MTPartition contenue par la PMInfoPartition est bien une PMPartition et non n'importe quelle MTPartition.

5.2.2 PMPartition

La PMPartition contient plusieurs attributs concernant les mailleurs. Ces attributs ont été placés dans la partition plutôt que dans l'infoPartition car le rôle premier d'une partition de maillage est, évidemment, de mailler. Les informations concernant les mailleurs sont donc des informations vitales. La PMPartition contient le nom des deux mailleurs, soit le mailleur 1D et le mailleur 2D. Elle contient aussi, pour chaque mailleur, une string représentant les paramètre de maillage. Finalement, deux booleen servent à indiquer si les mailleurs sont des *dll* ou non. La section sur les mailleurs (5.4) explique l'utilisation de chacun des ces paramètres. Le point important réside dans le fait que la partition contient de l'information sur les mailleurs et des méthodes permettant d'assigner ou de lire ces informations. Toutefois, la partition ne se sert pas directement de ces paramètres. Elle ne fait que les conserver et les fournir au moment opportun.

5.2.3 PMInfoCouche

La PMInfoCouche possède deux PMObjetGraphiqueMaillageVariable. Il s'agit d'objets graphiques servant à représenter le maillage valide ainsi que le maillage précédent. La PMInfoCouche fournit des méthodes permettant de lire et d'assigner ces objets graphiques. De plus, la PMInfoCouche fournit une méthode *invalideMaillage()* . Le travail de cette méthode consiste à rendre invalide le maillage actuel(valide) en le plaçant dans le maillage invalide pour ensuite vider le contenu du maillage valide. Cette méthode est appelée par le gestionnaire lorsque la PMInfoCouche a été désignée comme devant être démaillée.

5.2.4 PMCouche

La PMCouche conserve le champ d'erreur et le niveau de tolérance. Le mailleur se servira de ces deux informations pour mailler le domaine de la couche. La couche

contient aussi le nom des mailleurs 1D et 2D à utiliser ainsi que des paramètres de maillage propres à la couche. Les mailleurs 1D et 2D doivent être les mêmes pour la couche et pour la partition dont elle fait partie. Il n'a pas encore été confirmé que les couches auront besoin de paramètres de maillage supplémentaires à ceux définis par la partition. Cela se produira si on souhaite avoir des paramètres de maillage variants d'une couche à l'autre.

5.2.5 PMObjetGraphiqueMaillageVariable

Le PMObjetGraphiqueMaillageVariable est un objet graphique représentant un maillage. C'est un objet graphique à simple acteur. Contrairement aux objets graphiques plutôt constants (ou statique) le PMObjetGraphiqueMaillageVariable ne contient pas explicitement l'information qu'il affiche, c'est à dire qu'il ne contient pas de pointeur au maillage qu'il cherche à afficher.

À la construction du PMObjetGraphiqueMaillageVariable, un acteur est construit et assigné à cet objet graphique. Il s'agit toutefois d'un acteur vide, n'illustrant rien. L'acteur sera peu à peu rempli par des appels successifs à la méthode *ajouteMaillage(...)* du PMObjetGraphiqueMaillageVariable. Chaque appel à *ajouteMaillage(...)* a pour effet de dessiner le petit bout de maillage passé en paramètre dans l'acteur du PMObjetGraphiqueMaillageVariable et cela, sans effacer ce qui est déjà contenu par cet acteur.

La méthode *videActeur()* permet de vider l'acteur du PMObjetGraphiqueMaillageVariable.

L'attribut *m_ombre,* indique le niveau de couleur que le maillage doit avoir. On accepte deux niveaux de couleur, soit le niveau normal, et le niveau ombragé, c'est à dire une couleur moins vive. Un exemple de PMObjetGraphiqueMaillageVariable ombragé est le m_*maillagePrecedent* contenu par la PMInfoCouche. Une modification de *m_ombre* n'a pas d'impact immédiat sur l'acteur. C'est plutôt lors de la synchronisation de l'acteur que ce dernier sera mis à jour par rapport à cette variable.

Il ne faut pas confondre le PMObjetGraphiqueMaillageVariable avec le VIObjetGraphiqueMaillageConstant. Le premier sert au module partition de maillage ; il ne conserve pas de pointeur au maillage qu'il affiche et il peut être modifié, rabouté, et vidé. Le second objet graphique est tout le contraire. Il est utilisé par le module visualisation pour la visualisation d'un maillage statique ; il conserve un pointeur au maillage qu'il affiche et ne peut plus être modifié.

La seule similarité entre ces deux objets réside dans le fait qu'ils utilisent le même algorithme, soit TRTraceElement, pour remplir leur acteur. Le VIObjetGraphiqueMaillageConstant appelle TRTraceElement qu'une seule fois. Pour sa part, le PMObjetGraphiqueMaillageVariable utilise TRTraceElement à chaque fois que sa méthode *ajouteMaillage(...)* est appelée.

5.3 Algorithmes

Pour effectuer son travail, la partition de maillage a recourt à différents algorithmes. Il sera ici question de ces algorithmes.

5.3.1 PMAlgoAssemble

Le PMAlgoMaille divise les couches en petites zones et maille une à une chacune de ces zones. Ensuite, les maillages de ces zones doivent être réunifiés.

Le PMAlgoMaille doit être indépendant de la manière dont ces maillages sont réunifiés. C'est pourquoi il utilise le PMAlgoAssemble qui lui est fourni lors de sa configuration.

Si l'on souhaite afficher le résultat (PMEvenMailler avec *m_typeAction* == MAILLER_VTK), le PMAlgoAssemble devra assembler les petits bouts de maillage dans les *m_maillageValide* de chaque couche. C'est le travail du PMAlgoAssembleVtk.

D'autre part, si l'on souhaite enregistrer le résultat (PMEvenMailler avec *m_typeAction* == MAILLER_ENREGISTER), alors le PMAlgoAssemble devra réunir tous les maillages, indépendamment de la couche contenant la zone, en un seul maillage. C'est le travail du PMAlgoAssembleEnregistre.

PMAlgoAssemble définit une interface à laquelle tous les algorithmes d'assemblage doivent se confirmer. C'est assez simple, l'interface contient une méthode *ajoute(...)* qui identifie le maillage à ajouter et la couche à laquelle ce maillage est associé.

5.3.2 PMAlgoDemaille

Le PMAlgoDemaille est l'algorithme chargé de trouver quelles sont les couches à démailler suite à une action. En fait, son travail est de calculer quelles sont les couches affectées par la dite action.

Cet algorithme est configuré à l'aide d'une PMPartition. C'est la partition sur laquelle le calcul aura lieu.

Après avoir configuré l'algorithme, le gestionnaire peut lui demander de calculer les couches affectées par une action. Quelle que soit l'action, l'algorithme ne doit connaître que le niveau de priorité de la couche ayant été modifiée. À partir de là, il pourra calculer les couches affectées par la modification de paramètres. Pour l'instant, le PMAlgoDemaille considère que toutes les couches de niveau inférieur doivent être démaillées. Ce n'est peut-être pas le comportement souhaité.

Le PMAlgoDemaille n'effectue aucun démaillage lui-même, il ne fait que calculer les couches affectées. Il remplit un vecteur contenant tous les niveaux des couches devant être démaillées et fournit des itérateurs permettant de parcourir ce vecteur.

Ainsi, après avoir appelé le PMAlgoDemaille, le PMGestionnaire doit lire le contenu du vecteur du vecteur contenu dans le PMAlgoDemaille et invalider le maillage sur les infoCouche dont le niveau est indiqué dans ce vecteur.

C'est le PMGestionnaire qui se charge d'invalider les maillages car c'est lui qui a la responsabilité de demander au module d'affichage de se mettre à jour suite à ces modifications graphiques.

5.3.3 PMAlgoMaille

Le PMAlgoMaille est sans aucun doute l'algorithme le plus important de la partition de maillage. Son travail est aussi plus complexe.

Dans un premier temps, le gestionnaire configure le PMAlgoMaille. Pour ce faire, il lui transmet les paramètres suivants :

- Partition dans laquelle des couches seront maillées
- Algo d'assemblage à utiliser
- mailleur 1D
- mailleur 2D

Ensuite, l'algorithme est exécuté à l'aide de l'opérateur ( ). Le seul paramètre alors nécessaire est un vecteur de PMInfoCouche. Il s'agit des couches à mailler.

La section de pseudo-code à la page suivante exprime les diverses étapes permettant de mailler une sélection. Elle relate le travail fait par l'opérateur ( ) en supposant que le PMAlgoMaille est déjà configuré.

L'opérateur ( ) du PMAlgoMaille utilise plusieurs fonctions privées de service afin d'effectuer son travail. En particulier, les sections 2a, 2b du bloc principal et la section 1a du bloc secondaire demandent un certain travail géométrique. Il sera plus amplement question de ces parties de codes dans la section de ce rapport réservées aux considérations géométriques (6).

BLOC PRINCIPAL :
Connaît (puisque l'algo est configuré):
Partition à mailler, algoAssemble, mailleur 1D, mailleur 2D
Reçoit :
Vecteur de couches à mailler
Sortie (modifie) :
L'AlgoAssemble contient le résultat du maillage ou bien a modifié des structures pour qu'elles contiennent le résultat (i.e les objets graphique de infoCouches).

Pseudo code du bloc principal :

1. Configurer les mailleurs 1D et 2D à partir des paramètres XML contenus dans la partition à mailler

2. Pour chacune des couches à mailler, et de la plus prioritaire à la moins prioritaire
   a. Extraire les zones à mailler de cette coucheAMailler (i.e les zones visibles)
   b. Ajouter des sommets aux zones à mailler aux endroits où elles croisent des parties visibles de couches moins prioritaires
   c. Pour chaque zone à mailler
      i. Mailler le contour de zoneAMailler avec un mailleur 1D (voir **BLOC SECONDAIRE**)
      ii. Ordonner zoneAMailler dans le sens anti-horaire
      iii. Obtenir un premier maillage à partir du contour de la zoneAMailler
      iv. Envoyer la zoneAMailler au mailleur 2D avec les paramètres de la coucheAMailler
      v. Récupérer le maillage obtenu en iv. Demander à l'algoAssemble d'assembler ce bout de maillage.

BLOC SECONDAIRE :
Connaît (Puisque que l'algo est configuré):
Partition à mailler, algoAssemble, mailleur 1D, mailleur 2D
Reçoit :
zoneAMailler
coucheAMailler(couche à laquelle appartient la zoneAMailler)
Sortie (modifie) :
Des sommets sont ajoutés à zoneAMailler aux endroits où le mailleur 1D a placé des nœuds.

Pseudo code du bloc secondaire :

1. Parcourir les arêtes de la zoneAMailler et, pour chaque arête :
   a. Trouver la couche propriétaire de l'arête à mailler dans la partition ; c'est à dire la couche la plus prioritaire à laquelle appartient l'arête.
   b. Extraire les paramètres de maillage de coucheProprietaire
   c. Demander au mailleur 1D de mailler areteAMailler en utilisant les paramètres extraits
   d. Récupérer le résultat en c. Ajouter à zoneAMailler des sommets aux endroits où le maillage 1D récupéré contient des nœuds.

5.4 Mailleurs

Le module partition maillage se base sur les mailleurs pour effectuer la partie plus mathématique de son travail, soit mailler les domaines.

Pour mailler une partition donnée, deux mailleurs sont utilisés, soient le mailleur 1D et le mailleur 2D. La partition contient l'information permettant de créer et configurer ces deux mailleurs.

Il a été question de la création et de la configuration des mailleurs à plusieurs reprises dans les réunions précédentes. En particulier, [7] traite du PMMailleurProxy et [10] du PMMailleurFactory. Ces questions ne seront pas discutées dans le présent rapport. Seules les informations jugées essentielles mais n'ayant fait l'objet d'aucun rapport jusqu'à maintenant seront discutées ici.

Une interface générale a été conçue pour les mailleurs. Il s'agit de la PMInterfaceMailleur. Le mailleur 1D et le mailleur 2D partagent la même interface. Cela a été fait dans le but de n'avoir besoin que d'un seul PMMailleurProxy et d'un seul PMMailleurFactory. L'attribut *m_dimension* contenu dans la PMInterfaceMailleur permet de définir le type de mailleur (1D ou 2D) représenté par l'objet. Il doit être assigné dès la création de la PMInterfaceMailleur ou plutôt lors dès la création d'un objet dérivant de PMInterfaceMailleur car PMInterfaceMailleur est une classe abstraite.

La PMInterfaceMailleur est une classe abstraite. Elle contient deux méthodes abstraites soient *mailleLigne(...)* et *mailleZone(...)*. Des préconditions ont été placées afin de s'assurer que *mailleLigne(...)* n'est appelée que sur les mailleurs1D et *mailleZone(...)* que sur les mailleurs 2D. Ainsi, les classes dérivées de PMInterfaceMailleur doivent, dans la redéfinition de *mailleLigne(...)* et *mailleZone(...)*, appeler la méthode de la classe de base. Un mailleur 2D dérivant de PMInterfaceMailleur doit définir *mailleLigne(...)*, mais il peut se contenter d'appeler *mailleLigne(...)* du PMInterfaceMailleur, laquelle se chargera d'identifier la précondition non respectée..

À la rédaction de ce rapport, le seul mailleur 2D intégré à Modeleur 2 est le mailleur frontal. Le mailleur frontal a été intégré de deux manières, soit en tant que *dll* externe et en tant que librairie. Dans le premier cas, il peut être construit par l'intermédiaire du PMMailleurProxy. Dans le second cas, il peut être construit par l'intermédiaire de la PMMailleurFactory.

Toujours à la rédaction de ce rapport, aucun vrai mailleur 1D n'a été intégré à Modeleur2. Un mailleur 1D trivial a toutefois été construit afin de simuler la dynamique de création et d'utilisation des mailleurs. Ce mailleur 1D reçoit une arête à mailler, en fait un élément L2 et retourne un maillage ne contenant qu'un seul élément L2. Un tel mailleur a un avantage considérable lors des tests. En effet, avec ce mailleur, les seuls noeuds devant se trouver sur les arêtes des couches sont les sommets des couches et les points d'intersections avec d'autres couches. Ainsi, le mailleur 1D trivial permet de repérer rapidement les erreurs dans l'algoMaille car les nœuds indésirables sont faciles à identifier.

Les mailleurs ont été placés dans le répertoire \Module partition maillage\ Mailleurs\. Pour les mailleurs utilisés comme librairie, un projet a été créé et ajouté à la solution Modeleur. Des projets ont aussi été créés pour les mailleurs utilisés comme *dll* externes. Toutefois, ils n'ont pas été ajoutés à la solution Modeleur.

> *Attention : les dll des mailleurs ne sont pas sur CVS. Pour exécuter une commande « maille » il faut s'assurer d'avoir préalablement construit les dll des mailleurs.*

Il est à noter que les mailleurs actuels effectuent leur travail à partir de la géométrie seulement. Ils sont donc indépendants des paramètres portés par les couches. Bien que ces paramètres leur soient fournis, ils ne les utilisent pas.

6 Considérations géométriques

Il a déjà été question à la section 5.3.3 du PMAlgoMaille. Toutefois, les sujets géométriques n'avaient pas été approfondis.

Lors de l'exécution de la méthode principale de PMAlgoMaille, ce dernier a recours à plusieurs fonctions internes et privés pour effectuer un travail géométrique. Le travail effectué par ces fonctions repose particulièrement sur ToolKit_Geometrie qui, à son tour, renvoie le travail à GEOS.

Dans cette section, il sera question des fonctions géométriques existant dans PMAlgoMaille. Ce rapport ne concernant pas le design des géométries, on se contentera de citer les principales fonctions de ToolKit_Geometrie. On insistera en particulier sur les sections de code causant problème lors du maillage.

6.1 PMAlgoMaille ::trouveZonesVisibles(...)

Voici la signature complète de la fonction dont il est question :

```
ERMsg PMAlgoMaille::trouveZonesVisibles(
                            GOMultiPolygone&           zonesVisibles,
                            ConstPMCoucheP             coucheP,
                      const TCContainerConstPMCouche&  vectExceptions)
```

Le premier paramètre (zonesVisibles) est un paramètre de sortie. Il s'agit du multipolygone représentant l'ensemble des zones visibles de la couche. Le deuxième paramètre représente la couche pour laquelle on recherche la zone visible. Finalement, le dernier paramètre (facultatif) est un vecteur d'exceptions. Il s'agit d'un ensemble de couches, plus prioritaires que coucheP, mais qui seront considérés comme transparentes dans le calcul. C'est-à-dire qu'elles ne seront pas prises en considération lors du calcul, comme si elles ne recouvrent pas coucheP.

Soit les trois couches suivantes (de la plus prioritaire à la moins prioritaire) :

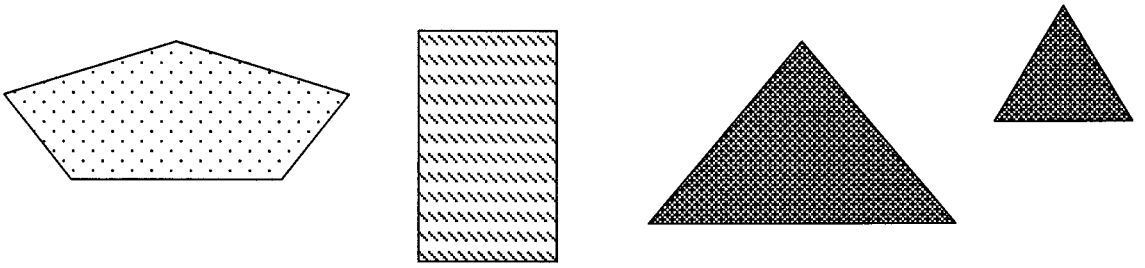


Figure 1 : Trois couches de la partition P

Au sein de la partition P, l'empilement de ces trois couches produit l'effet suivant :

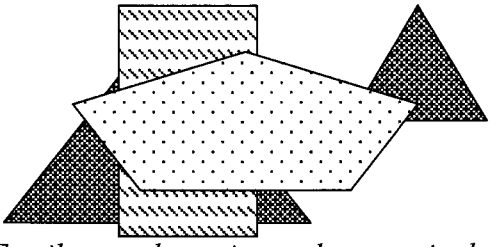


Figure 2 : Empilement des trois couches au sein de la partition P

Si on applique trouveZonesVisibles sur la couche 2 (les deux triangles) on obtient le résultat suivant :

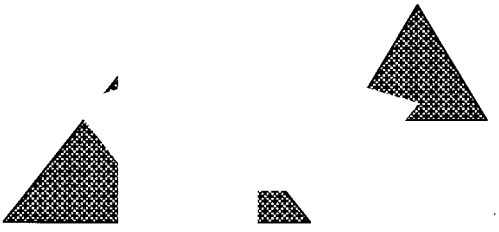


Figure 3 : Résultat de trouveZonesVisibles avec la couche 2 comme second paramètre

Toutefois, si on applique trouveZonesVisibles, mais cette fois en plaçant la couche 0 (le pentagone) dans le vecteur d'exceptions, on obtient le résultat suivant :

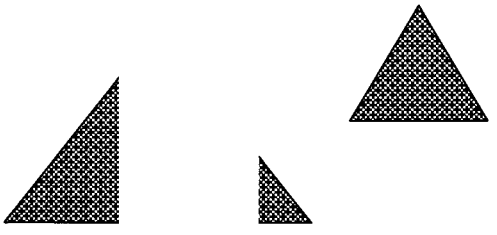


Figure 4 : trouveZoneVisible utilisant le vecteur d'exception

Le travail de la fonction *trouveZonesVisibles(...)* est assez simple. Cela repose principalement sur la méthode *difference(...)* de la classe GOMultiPolygone.

La méthode *difference(...)* de la classe GOMultiPolygone effectue la différence entre deux multiPolygones. Cela revient à « enlever » au multiPolygone courant les sections appartenant au multiPolygone passé en paramètre.

Ainsi, le travail de *trouveZonesVisibles(...)* consiste d'abord à placer le multiPolygone représentant *coucheP* (couche pour laquelle on cherche la zone visible, soit couche 2 dans l'exemple ci-haut) dans le multiPolygone de sortie nommé *zonesVisibles*. Sont ensuite enlevées, une à une, les zones étant recouvertes par les couches plus prioritaires et ce, à l'aide de la méthode *difference(...)* du GOMultiPolygone. À la fin, on obtient le résultat voulu, soit le multiPolygone *zonesVisibles* représentant les zones de la *coucheP* étant visibles dans l'empilement de la partition.

La fonction *touveZonesVisibles(...)* est utilisée à diverses reprises par l'operateur ( ) du PMAlgoMaille. En se référant au pseudo code en 5.3.3, on voit qu'il est question de partie visible en 2a et en 2b.

Pour l'instant, seul le PMAlgoMaille a besoin d'utiliser une méthode comme *trouveZonesVisibles(...)*. Il est toutefois important de noter que, bien qu'elle utilise des PMCouche en paramètre, le travail de la méthode *trouveZonesVisibles(...)* pourrait aussi bien s'appliquer aux MTCouche qu'aux PMCouche. Ainsi, si, dans un futur module, une telle fonction s'avère nécessaire, il pourrait être possible de déplacer cette fonction vers le module MNT où elle devrait être rendue publique.

6.2 PMAlgoMaille ::traiteAjouteSommet

Voici la signature complète de la fonction dont il est question :

```
ERMsg PMAlgoMaille::traiteAjouteSommet(
                            GOMultiPolygone&    mpgZonesAMailler,
                            ConstPMInfoCoucheP  infoCoucheAMaillerP)
```

Cette fonction ajoute au *mpgZonesAMailler* passé en paramètre des sommets aux endroits où il croise des parties visibles de couches moins prioritaires. Le *mpgZonesAMailler* correspond à la partie visible de *infoCoucheAMaillerP*. Le paramètre *infoCoucheAMaillerP* sert à déterminer la priorité de la couche et ce, afin de trouver quelles sont les couches moins prioritaires.

Considérons à nouveau l'exemple de la section précédente. Dans l'illustration ci-bas, il est clair que la couche 1 (le rectangle) aura un noeud en A lorsqu'il sera maillé. Afin d'obtenir un maillage continu, la couche 0 (le pentagone) doit aussi avoir un noeud en A. C'est pour cela qu'un sommet doit être ajouté à la couche 0 en A. L'ajout de ce sommet est justement le travail de la fonction *traiteAjouteSommet(...)*.

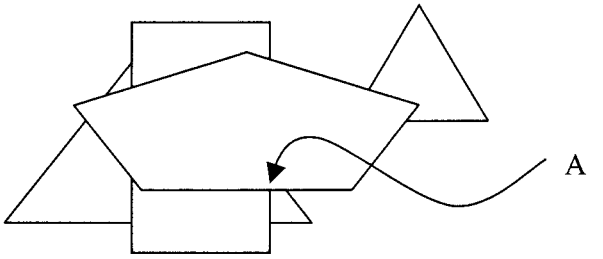


Figure 5 : Sommet A sera ajouté à la couche 0 par traiteAjouteSommet(...)

Il a été mentionné que des sommets sont ajoutés aux endroits **visibles** où la couche considérée croise des couches moins prioritaires. Par endroit visible, on entend un endroit n'étant pas recouvert par une autre couche. Dans la figure 6, on voit que la couche 2 croise la couche 0 en B. Or, il n'est pas souhaitable d'ajouter un sommet à la couche 0 à cet endroit. En effet, ce point de croisement n'étant pas visible, aucun ajout de sommet n'est nécessaire pour respecter les contraintes de continuité.

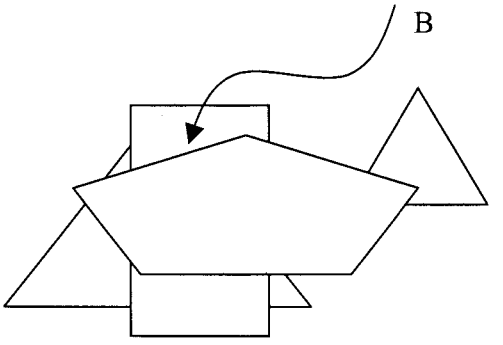


Figure 6 : Il ne doit pas y avoir de sommet d'ajouté au pentagone en B

La fonction *traiteAjouteSommet(...)* se fonde principalement sur la méthode *ajoutePointsCroisement(...)* du GOMultiPolygone pour effectuer son travail. Voici la signature de cette méthode :

ERMsg GOMultiPolygone ::ajoutePointsCroisements(const GOMultiPolygone&)

Cette méthode ajoute au multipolygone courant les points de croisement avec le multipolygone fournit en paramètre. Considérons les deux multipolygones de la page suivante :

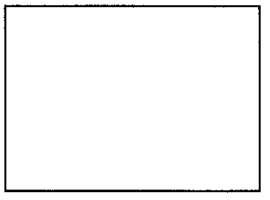


Figure 7 : Multipolygone courant auquel des sommets seront ajoutés

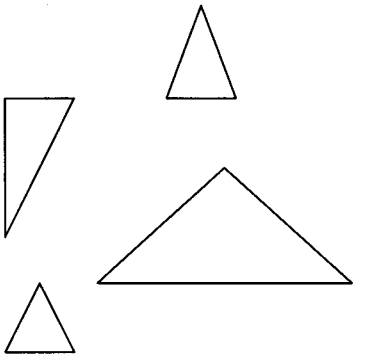


Figure 8 : Multipolygone passé en paramètre servant à trouver les sommets à ajouter

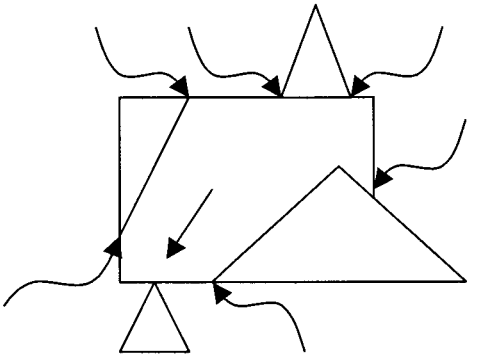


Figure 9 : Combinaison des deux multipolygones illustrant les sommets qui seront ajoutés au rectangle par GOMultiPolygone::ajoutePointsCroisements(...)

La méthode *GOMultiPolygone ::ajoutePointsCroisement(...)* ne prend en considération que deux multipolygones à la fois. De plus, elle ne se préoccupe pas des contraintes de « zone visible » comme doit le faire la fonction *PMAlgoMaille ::traiteAjouteSommet(...)*. Pour effectuer son travail, *traiteAjouteSommet(...)* doit parcourir les couches moins prioritaires que *infoCoucheAMaillerP* (voir signature de la fonction). Pour chacune de ces couches moins prioritaires, elle doit trouver et ajouter les points de croisements avec *mpgZonesAMailler* en appelant *ajoutePointsCroisement(...)*. Toutefois, elle doit prendre soin de ne pas ajouter des points superflus (voir figure 6). Pour ce faire, elle doit effectuer un travail avant d'appeler *ajoutePointsCroisement(...)*. Il sera donc question ici du travail à effectuer afin de ne pas ajouter de sommets superflus.

6.2.1 Extraire la zone visible

Reprenons l'exemple déjà introduit à la section 6.1. Cette fois, supposons que la couche à mailler est la couche 0, soit celle composée du pentagone uniquement. Afin d'ajouter les sommets au multipolygone de la couche 0, la fonction *traiteAjouteSommet(...)* est appelée avec les paramètres suivants :

coucheP : couche 0
mpgZoneAMailler : multipolygone de la couche 0 composé d'un pentagone

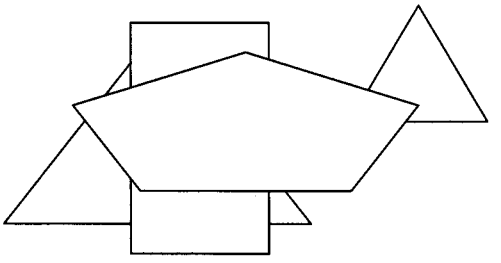


Figure 10 : Des sommets doivent être ajoutés au pentagone formant le multipolygone de la couche 0.

La fonction *traiteAjouteSommet(...)* parcourt les couches moins prioritaires que la couche 0. Pour chacune des couches rencontrées, elle effectue le travail suivant :

1. Copier le multipolygone de la couche rencontrée ;
2. Extraire la partie visible du multipolygone en *1*;
3. Ajouter à *mpgZonesAMailler* les points de croisement avec le multipolygone obenu en *2*.

Dans ce cas, c'est l'étape *2* qui assure que seuls des sommets visibles sont ajoutés à *mpgZonesAMailler*. En effet, avant de calculer les points de croisement entre les deux couches, on extrait la partie visible. Ainsi, les points de croisements trouvés seront nécessairement visibles.

Illustrons cet algorithme à l'aide de l'exemple développé. La première couche parcourue est la couche 2, soit celle composée de triangles.

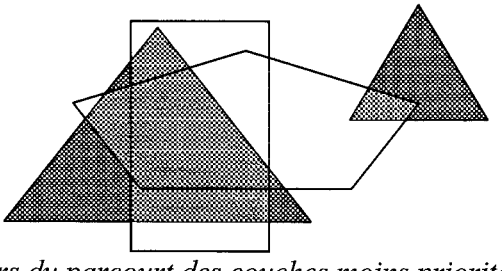


Figure 11 : Lors du parcourt des couches moins prioritaires que couche 0, la couche 2 est d'abord rencontrée.

Dans un premier temps, le multipolygone de la couche 2 est copié. Ensuite, sa partie visible est extraite à l'aide de *trouveZonesVisibles(...)*. On obtient la partie visible suivante :

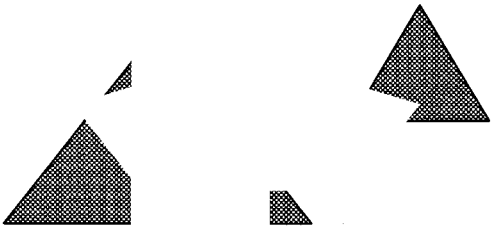


Figure 12 : Partie visible de la couche 2.

Finalement, à l'étape 3, on appelle *GOMultiPolygone ::ajoutePointsCroisement(...)* afin d'ajouter au *mpgZoneAMailler* les points de croisement avec le multipolygone obtenu à l'étape précédente. Les sommets ajoutés sont donc les suivants :

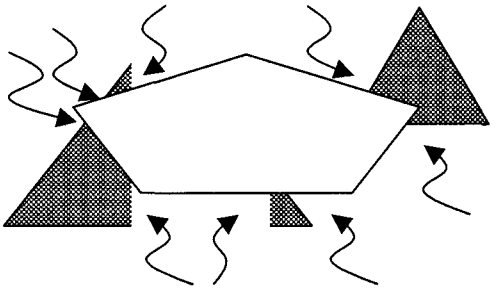


Figure 13 : Sommets ajoutés au pentagone par ajoutePointsCroisemets(...) lorsque le multipolygone passé en paramètre est la partie visible de la couche 2.

On peut comparer avec la figure 6 et remarquer que, comme désiré, aucun sommet n'est ajouté en B.

D'autres sommets doivent être ajoutés à la couche 0. Il s'agit des points de croisement avec la couche 1 (composée d'un rectangle seulement). Le détail ne sera pas fait à nouveau ici. On peut facilement voir, en consultant la figure 10 que quatre sommets supplémentaires seront ajoutés au multipolygone de la couche 0.

Cette façon de faire fonctionne théoriquement très bien. Elle a toutefois causé certains problèmes lorsqu'elle fut implantée.

Le problème semble provenir de la gestion de la précision suite aux opérations successives effectuées avec GEOS.

Dans l'exemple ci-haut, on extrait la partie visible du multipolygone composé de triangles. En effectuant ce travail, on réduit automatiquement ses zones de rencontre avec le pentagone à des rencontres se situant sur les arêtes communes. Or, un tout petit décalage entraînera l'omission d'un sommet.

Afin de simplifier la question, considérons la partie suivante du schéma :

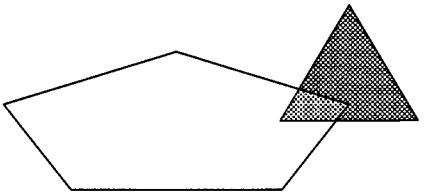


Figure 14 : Section du schéma de la figure 11.

En un premier temps, on extrait la partie visible du triangle, pour ainsi obtenir le résultat suivant :

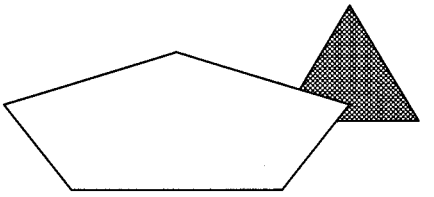


Figure 15 : Partie visible du triangle.

Toutefois, il se peut que les points de croisements trouvés ne se situent pas directement sur l'arête du pentagone. Dans ce cas particulier, le point de croisement en est très près mais n'est pas directement dessus. C'est une petite erreur de précision qui se glisse faisant en sorte que le point trouvé pour tronquer le triangle, bien qu'il soit calculé à partir des équations des arêtes du triangle et du pentagone, n'appartient pas exactement à ces arêtes. En zommant et en exagérant largement l'erreur, on pourrait imaginer que la figure représente plutôt le schéma suivant :

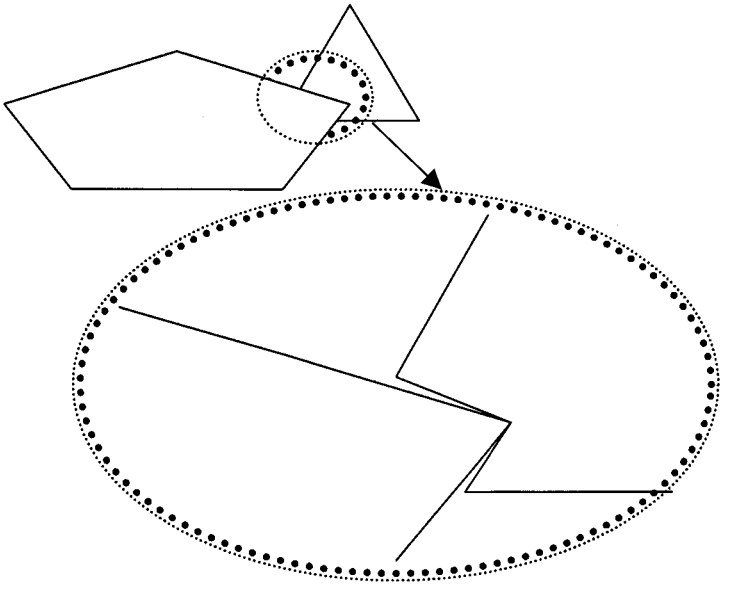


Figure 16 : Zoom exagérant l'erreur de précision.

Avec un tel résultat pour la zone visible du triangle, un problème survient nécessairement plus tard, à l'étape 3, lorsque l'on cherche les points d'intersectin entre cette zone visible et le pentagone. En effet, la petite impécision introduite a pour effet que les points d'intersections ne sont plus identifiés.

Bref, bien que ce premier algorithme soit théoriquement correct, il conduit à une erreur lors de son application. Ce n'est pas à tous les coups que des sommets sont oubliés. Or, ces omissions de sommet sont assez fréquente.

L'idéal sera d'arriver à bien gérer la précision avec GEOS et ensuite réintroduire cet algorithme. Il faudra de toute manière régler le problème de précision GEOS afin d'obtenir des résultats plus fiables et plus cohérents d'un à l'autre.

6.2.2 Extraire la zone visible en omettant la couche courante

Bien que le premier algorithme soit cohérent, il ne fonctionne pas très bien. Un des objectifs ciblé lors du développement de la partition de maillage était d'obtenir rapidement une partition la plus fonctionnelle possible. L'algorithme déficient devait donc être remplacé par un autre plus efficace.

Le second algorithme ici présenté est celui actuellement présent dans le code. Il devra toutefois être modifié car, bien qu'il donne en pratique des meilleurs résultats que le premier, il comporte une erreur de logique.

Pour comprendre la raison de cet algorithme, il faut bien voir que les erreurs d'imprécisions qui se produisent dans l'algorithme précédent proviennent toujours de points de croisements situés sur des arêtes partageant un segment. C'est dans ces cas que GEOS est plus sensible aux erreurs. Or, le calcul de partie visible avant l'extraction d'arête amène nécessairement ce genre de problème.

L'idée de base de ce second algorithme est de calculer la partie visible, mais en omettant la couche à mailler dans ce calcul. De cette manière, les points de croisement ne seront pas sur les arêtes partageant un segment.

Considérons encore une fois l'exemple introduit aux sections précédentes, avec la couche 0 (le pentagone) comme couche à mailler. Lorsqu'on itère sur les couches moins prioritaires pour trouver les sommets à ajouter, on rencontre d'abord la couche 2, soit celle composée de triangle.

Comme dans l'algorithme précédent, on extrait la partie visible de la couche 2. Toutefois, on fait cette extraction en omettant la couche 0, c'est-à-dire en considérant la couche 0 comme invisible. On obtient donc le résultat suivant pour la partie visible de la couche 2.

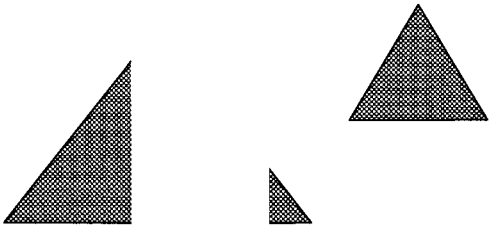


Figure 17 : Partie visible de la couche 2 en considérant comme invisible la couche à mailler (couche 0).

L'avantage d'omettre la couche 0 dans le calcul est que, à l'étape suivante, lorsque l'on vient pour trouver les points de rencontre, ces derniers sont plus facilement obtenus car les arêtes ne partagent plus de segment mais se croisent réellement.

Dans un second temps, on peut calculer les points à ajouter à la couche 0. Les points de croisement sont facilement trouvés et ce, sans erreur. On reprend ensuite le même travail avec la couche 1.

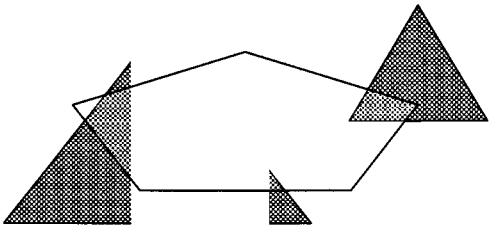


Figure 18 : Les points de croisements sont trouvés sans instabilité car il y a réellement croisement et non pas seulement un partage de segment d'arête.

Dans l'exemple illustré ci-haut, l'algorithme effectue le travail escompté. C'est d'ailleurs l'avantage de cet algorithme ; il produit les résultats attendus. Mais justement, ce n'est pas un bon algorithme car, dans certains cas, le résultat attendu est lui-même mauvais. En voici un exemple.

Soit les trois couches suivantes :

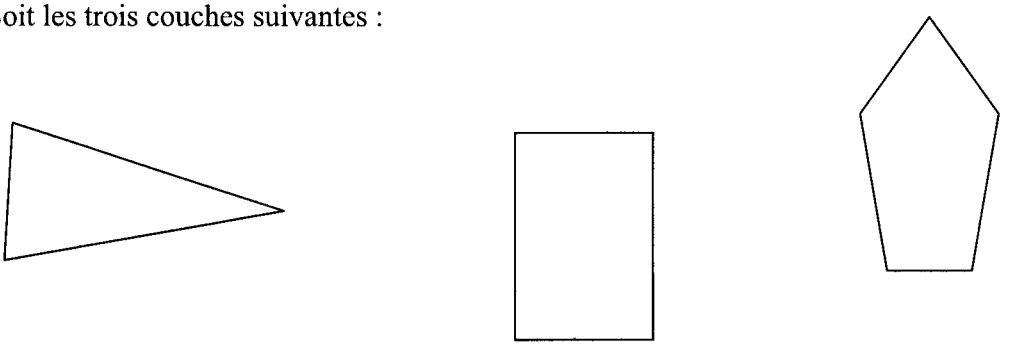


Figure 19 : Trois couches qui seront empilées dans une partition.

Dans la partition, ces couches sont empilées de la manière suivante :

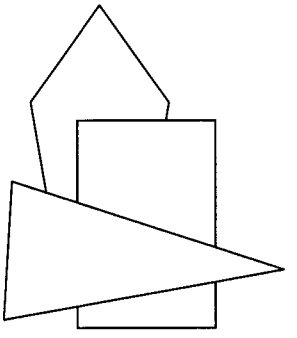


Figure 20 : Partition contenant les trois couches empilées selon leur priorité

Considérons que l'on souhaite mailler la couche du centre, soit celle composée d'un rectangle. Dans un premier temps, on trouve *mpgZoneAMailler* comme étant le multipolygone suivant :

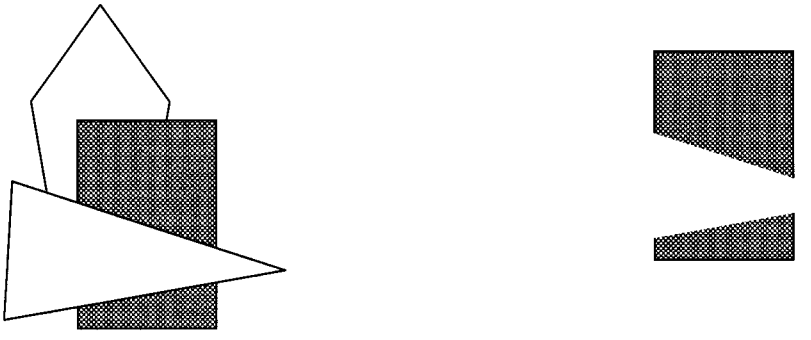


Figure 20 : La partie hachurée du rectangle, formée de deux quadrilatères, forme mpgZoneAMailler.

Ensuite, on trouve les sommets devant être ajoutés à *mpgZoneAMailler*. Pour ce faire on appelle *traiteAjouteSommet(...)* en lui fournissant *mpgZoneAMailler*. Cette fonction itère sur les couches moins prioritaires que couche 1 de la partition. Elle ne rencontrera que couche 2, soit la couche composée du pentagone. Ensuite, elle copie le multipolygone de couche 2 (le pentagone) et tente d'obtenir la partie visible de ce multipolygone, mais en omettant la couche 1 dans le calcul de partie visible. Ainsi, la partie visible calculée est la suivante :

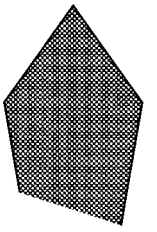


Figure 21 : Partie visible de la couche 2 calculée en considérant la couche 1 comme invisible.

Par la suite, des sommets sont ajoutés à *mpgZoneAMailler* aux endroits où il rencontre le multipolyone précédent.

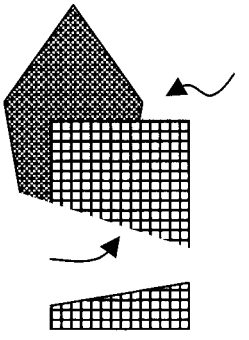


Figure 22 : Sommets ajoutés à mpgZoneAMailler

Reportons ces nouveaux sommets dans le dessin de la partition :

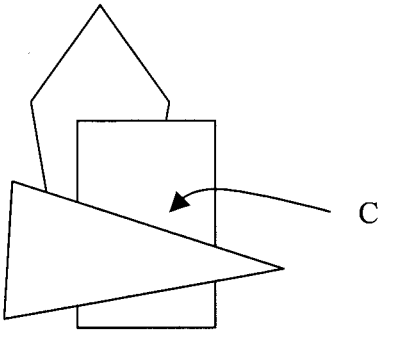


Figure 23 : Sommet ajouté à tord à la couche 1 (formée du rectangle en C).

On voit alors clairement qu'un sommet a été ajouté alors qu'il n'aurait pas dû l'être. Ce type d'exemple cause un problème sérieux puisque il vient briser la continuité du maillage. En effet, ce sommet est ajouté au multipolygone de la couche 1. Or, pour que le maillage soit continu, le même sommet devrait être ajouté au multipolygone de la couche 0, ce qui n'est pas le cas. En effet, lorsque l'ont souhaitera mailler la couche 0, on

trouvera la partie visible du pentagone (en considérant la couche 0 comme invisible) ce qui donnera le résultat suivant :

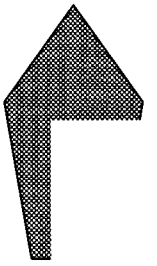


Figure 24 : Partie visible de la couche 2 en considérant la couche 0 comme invisible.

Ensuite, lorsque l'on calculera les points d'intersection entre ce multipolygone et le multipolygone de la couche 0, le point C de la figure 23 ne se retrouvera pas sur la couche 0, ce qui causera un incontinuité dans le maillage. En effet, un nœud se trouvera en C sur la couche 1 (voir figure 23) alors qu'il n'y aura pas de nœud en ce même endroit sur la couche 0 (voir figure 25).

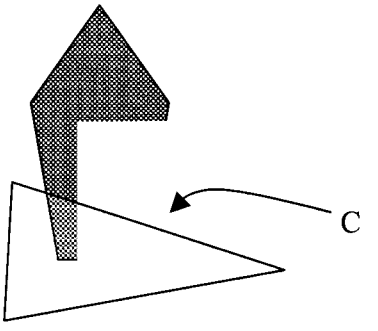


Figure 25 : Il n'y aura pas de nœud en C sur la couche 0.

Bref, le premier algorithme, théoriquement correct mais causant des erreurs à l'exécution, a été remplacé par ce second algorithme. Cela constitue une erreur car, quoi qu'il semble mieux fonctionner à l'exécution, les erreurs commises ne sont pas dues à des erreurs de précision mais bien à une défaillance claire dans l'algorithme. Il s'agit donc d'un mauvais choix de remplacement.

Il est à noter que le vecteur d'exceptions introduit dans la méthode *trouveZonesVisibles(...)* (voir 6.1) sert précisément à cette version de *traiteAjouteSommet(...)*. En effet, c'est dans cette version qu'on a besoin d'extraire une partie visible en considérant certaines couches comme invisibles. Puisque cette version de la fonction *traiteAjouteSommet(...)* devra être remplacée, il sera aussi souhaitable

d'enlever le vecteur d'exceptions inclus dans *trouveZonesVisibles(...)*. Cela viendra nettement simplifier la signature de cette dernière méthode.

6.2.3 Ajouter les sommets invisibles

Voici une dernière manière de traiter l'algorithme d'ajout de sommet. Il s'agit ici d'un compromis qui pourrait être utilisé temporairement en attendant de mieux comprendre GEOS. L'idée en est bien simple. On a déjà vu que ce qui cause problème, c'est le traitement des sommets « invisibles » qui ne doivent pas être enlevés. En fait, en enlevant cette condition, on risque d'enlever bien des problèmes.

Ainsi, on pourrait tolérer de forcer des sommets n'étant pas nécessaires si cela nous assure des résultats plus fiables. Bref, l'idée de ce dernier algorithme consiste à ajouter tous les points de croisement entre les couches, sans tenir compte du fait qu'ils soient des points visibles ou non. En éliminant ainsi l'appel à *trouveZonesVisibles(...),* on ne fera appel à GEOS qu'une seule fois, ce qui conduira sans doute à de meilleurs résultats.

6.3 PMAlgoMaille ::trouvePrincipalProprietaireArete

Avant de mailler l'intérieur d'une zone (mailler 2D), on doit d'abord mailler le contour de la zone à l'aide d'une mailleur 1D. Pour ce faire, on a recourt à la fonction suivante :

```
PMAlgoMaille ::maille1DContour  ( GOPolyligne&        contour,
                                  ConstPMInfoCoucheP  infoCoucheAMaillerP
                                ) const
```

La GOPolyligne passée en paramètre représente le contour d'une zone à mailler. Pour sa part, la ConstPMInfoCoucheP fournie est la couche dont le contour d'une des zone à mailler est représentée par la GOPolyligne.

La fonction *maille1DContour* a pour objectif de mailler chacune des arêtes de *contour.* Pour mailler ces arêtes, elle doit utiliser des paramètres de maillage. Les paramètres de maillage à utiliser sont donnés par la couche la plus prioritaire contenant l'arête à mailler.

On sait déjà que les arêtes appartiennent toutes à *infoCoucheAMaillerP.* C'est évident puisque c*ontour* est justement le contour d'une zone à mailler de *infoCoucheAMaillerP.* Toutefois, il est possible qu'une arête de *contour* appartienne non seulement à *infoCoucheAMaillerP,* mais aussi à d'autres couches. De plus, ces dernières peuvent être plus prioritaires que *infoCouheAMaillerP.* Cela se produit en particulier lorsque *infoCoucheAMaillerP* croise d'autres couches plus prioritaires et que son multipolygone a été tronqué pour former la zone à mailler.

Considérons à nouveau l'exemple illustré à la figure 20. Cette fois, supposons que la couche à mailler est la couche 2, soit celle dont le multipolygone est formé d'un unique pentagone.

La zone à mailler est celle illustrée à la figure 24. Le contour de la zone est donc maillé avant l'intérieur. Pour ce faire, *maille1DContour(...)* est appelée en passant en paramètre la GOPolyligne suivante :

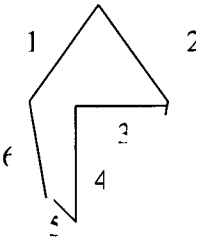


Figure 26 : GOPolyligne à mailler 1D où les arêtes sont numérotées.

À titre d'exemple, la couche 2 est propriétaire des arêtes 1, 2, et 6. Pour sa part, la couche 1 est propriétaire des arêtes 3 et 4. Finalement, la couche 0 est propriétaire de l'arête 5.

La fonction *maille1DContour(...)* parcourt les arêtes de la GOPolyligne. Elle doit mailler chacune d'elles. Pour ce faire, elle doit d'abord trouver la couche qui en est propriétaire afin d'en extraire les paramètres de maillage à utiliser. Par couche propriétaire, on entend la couche la plus prioritaire contenant l'arête en question. Trouver la couche la plus prioritaire contenant une arête, c'est le travail de *trouvePrincipalProprietaireArete(...)* dont la signature est la suivante :

```
ERMsg PMAlgoMaille::trouvePrincipalProprietaireArete     (
                      TCConstIterateurMTCouche&  proprietaireI,
                      ConstPMInfoCoucheP         infoCoucheAMaillerP,
                const GOPolyligne&               arete   ) const
```

Le premier paramètre de cette fonction (*proprietaireI*) est un paramètre de sortie. C'est en fait le propriétaire trouvé qui est retourné. Ensuite viennent deux paramètres d'entrée. Il s'agit de la couche à mailler et l'arête en question. La couche à mailler n'est pas tout à fait nécessaire. Toutefois, ce paramètre permet de vérifier que le bon propriétaire de l'arête a été trouvé. En effet, l'arête doit appartenir à *infoCoucheAMaillerP* ou bien à une couche moins prioritaire.

Le travail de la fonction *trouvePrincipalProprietaireArete(...)* est relativement simple. Il se base principalement sur *GOPolygone ::contient(...)* dont la signature est la suivante :

```
GOPolygone ::contient(Booleen& trouve, const GOPolyligne& arete)
```

Le paramètre *trouve* est un paramètre de sortie. Il indique si le polygone courant contient ou non l'arête *arete* passée comme paramètre constant d'entrée. Tout ce que fait la fonction *contient(...)*, c'est de demander à GEOS si le polygone courant contient ou non l'arête passée en paramètre.

Pour sa part, la fonction *PMAlgoMaille ::trouvePrincipalProprietaireArete(...)* itère sur les diverses couches de la partition, et ce, de la plus prioritaire à la moins prioritaire. Pour chacune d'elles, elle trouve le multipolygone de la couche, et itère sur chacun des polygones du multipolygone. Finalement, pour chacun de ces polygones, elle vérifie si *arete* appartient au polygone en question à l'aide de la fonction *GOPolygone ::contient(...).*

Théoriquement, *trouvePrincipalProprietaireArete(...)* devrait bien fonctionner. Or, certains problèmes, similaires à ceux rencontrés avec la fonction *traiteAjouteSommet(...,)* surviennent.

Par exemple, supposons que l'on cherche le propriétaire de l'arête 3 de la figure 26. Il faut comprendre que les sommets formant cette arête ont été créés plus tôt, soit au moment où la zone visible de la couche 2 fut extraite. À ce moment, la différence entre la couche 2 et la couche 1 a été faite, de manière à tronquer une partie de la couche 2 et créer l'arête 3 de la couche 2. Or, comme vu à la section 6.2.1, certains problèmes de précision peuvent survenir.

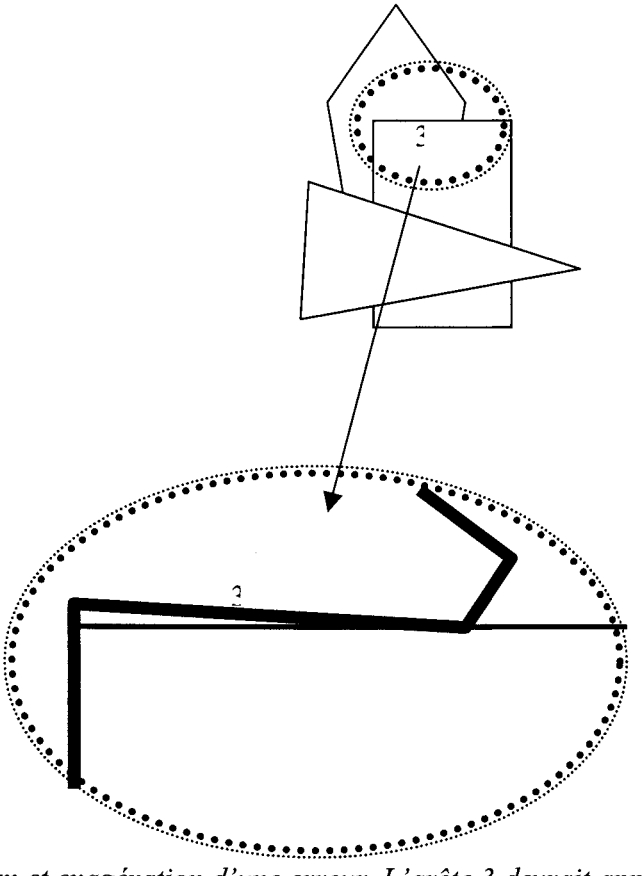


Figure 28 : Zoom et exagération d'une erreur. L'arête 3 devrait appartenir à la couche 1 ce qui n'est plus le cas.

Dans la figure 28 par exemple, on remarque que, lorsque le pentagone a été tronqué, un des sommets de l'arête 3 n'a pas tout a fait été placé à l'endroit souhaité. C'est une erreur de précision minime qui a été largement exagérée dans la figure. Toutefois, aussi minime qu'elle soit, cette petite différence a pour effet que, lorsque l'on cherche le propriétaire de l'arête 3, la couche 1 (le rectangle) n'est pas identifié comme en étant propriétaire.

Ce problème est très similaire à celui rencontré à la section 6.2.1. Ces deux problèmes viennent mettre une pression considérable sur les géométries actuelles. Il faudra absolument les fiabiliser en les renforcissant ou, au minimum, en établissant clairement leurs limites et en apprenant à travailler avec ces limites.

7 Conclusion

7.1 Résumé

Les grandes sections du module partition de maillage sont maintenant complétées. Il est possible d'ouvrir une partition, d'y ajouter des couches et de mailler les couches. On peut aussi modifier la géométrie d'une couche. Bien qu'elle ne soit pas encore tout à fait fonctionnelle, la partition de maillage permet présentement une interaction intéressante avec l'usager.

Les principaux blocs du casse tête sont donc en place. Ce présent document a permis de mieux connaître les différents blocs de cette structure et la manière dont ils interagissent. En particulier, on note des relations entre le module partition de maillage et le module MNT, le module gestionnaire de données et le module d'affichage.

Alors que les liens avec les autres modules ont été considérés en superficie, on est entrés plus en détail dans ce qui concerne le lien entre les différentes composantes internes du module partition de maillage. Parmi ces composantes, on note les événements, le gestionnaire, les couches et les partitions, les objets graphiques et, finalement, les algorithmes de maillage, de démaillage et d'assemblage.

Finalement, il fut grandement question des considérations géométriques. Ces dernières surviennent dans les méthodes privées du PMAlgoMaille et se basent principalement sur Toolkit_Geometrie. Cette section a permis de mieux comprendre les algorithmes géométriques utilisées et, surtout, les problématiques rencontrées. On y a ciblé l'importance de revoir la gestion de la précision dans les géométries afin d'éliminer certains erreurs qui ont un impact majeur au sein de la partition de maillage.

Le développement de la partition de maillage a permis de mettre à l'épreuve les ToolKit_Geometrie et, en particulier, GEOS. On peut maintenant affirmer avoir trouvé une librairie fiable, complète et, surtout, offrant beaucoup de support. Il reste évidemment du travail à faire dans ToolKit_Geometrie mais les futurs développeurs pourront compter sur le forum très dynamique de GEOS.

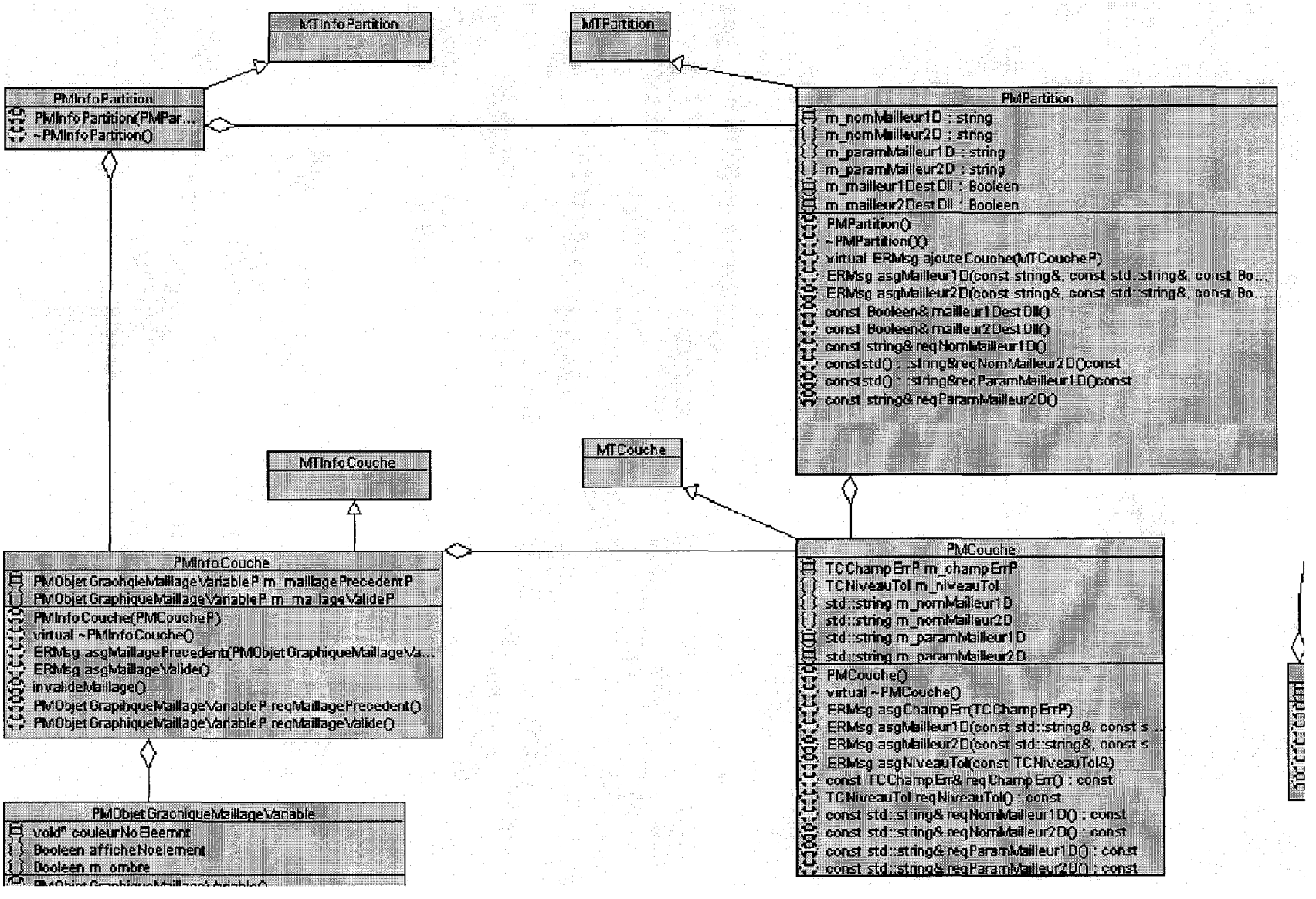
7.2 Perspectives futures

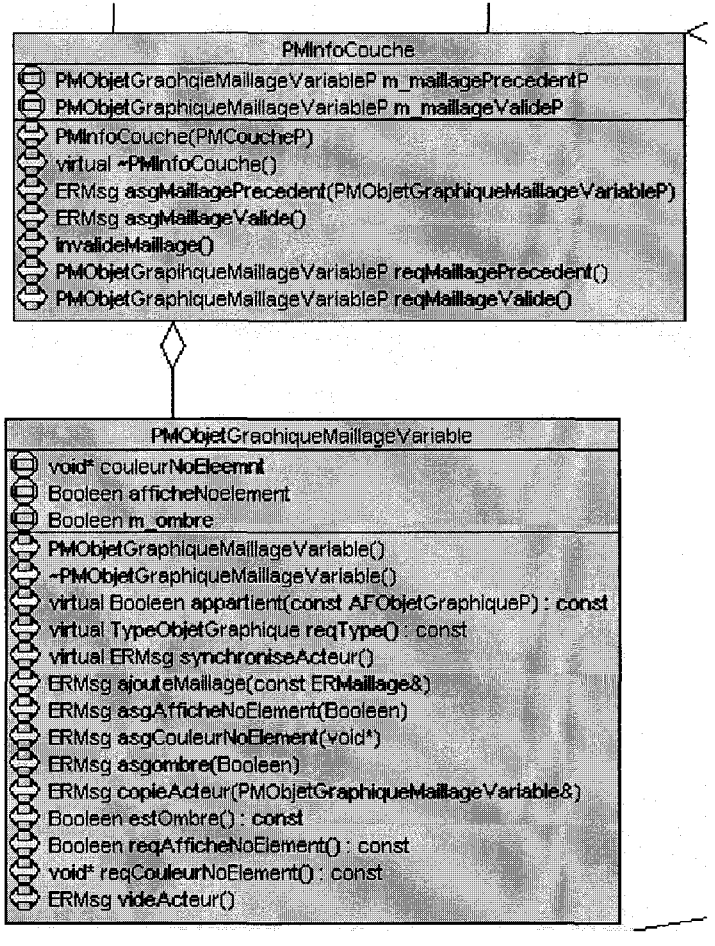
Ce rapport avait pour objectif principal de décrire l'état réel de la partition de maillage. Cela a permis de cibler plusieurs des étapes à franchir pour obtenir un module impeccable.

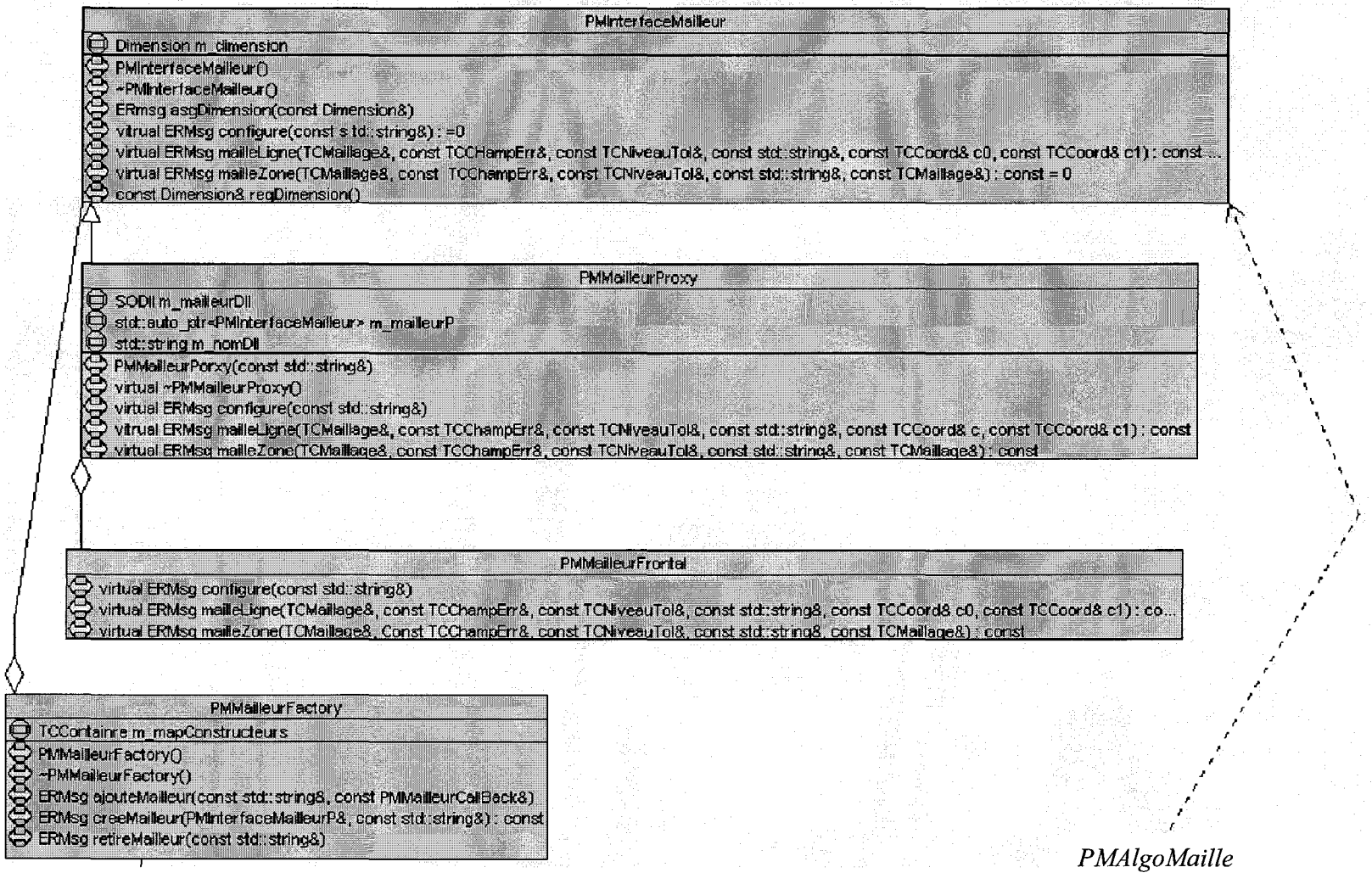
Tout au long du texte, on retrouve en parallèle la conception souhaitée pour la partition de maillage et l'état actuel de cette dernière. Dans plusieurs cas, les idées concernant la conception souhaitée devaient être élaborées plus largement. C'est pourquoi un second document [15] a été conçu. Ce dernier insiste principalement sur les trous de la partition de maillage et énumère donc les perspectives futures pour la partition de maillage.

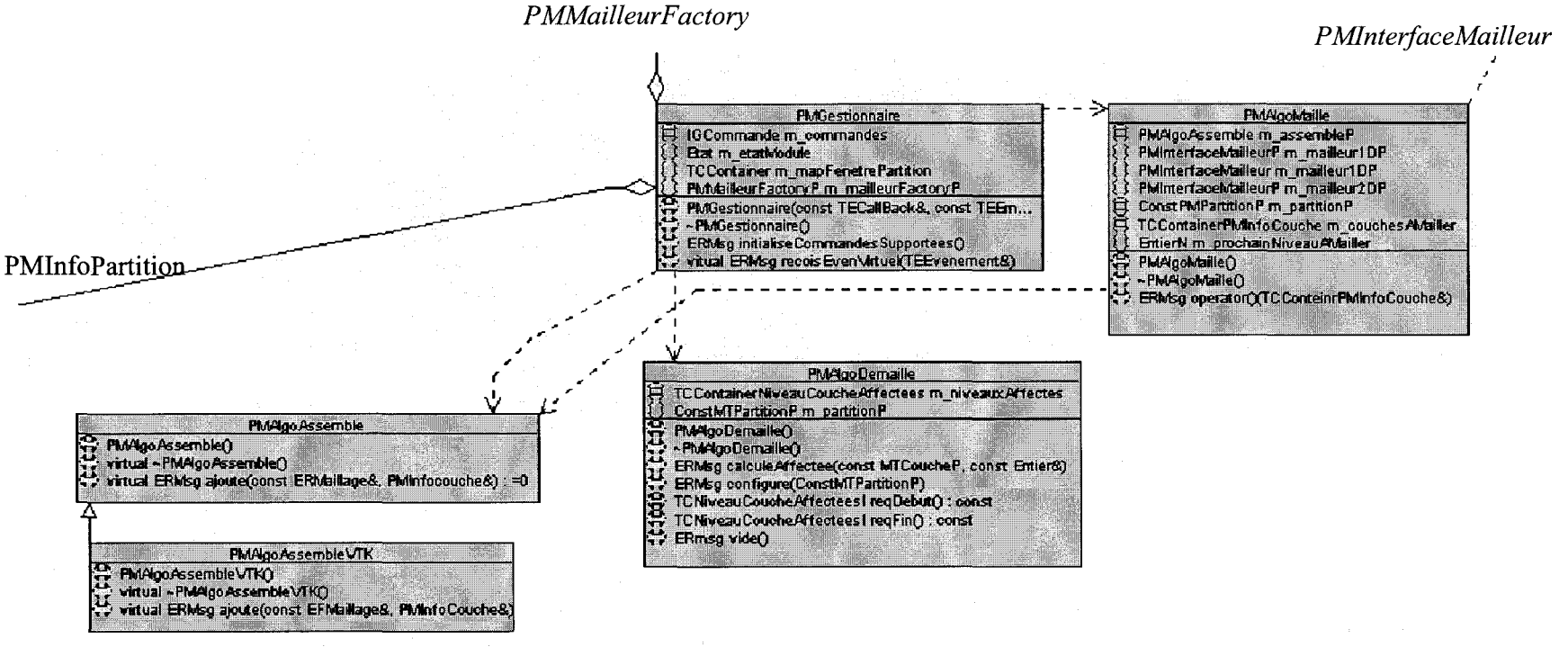
Parmi les principales améliorations à apporter, on note la séparation du travail de maillage en deux phases, soit une phase géométrique et une phase gestion des paramètres et maillage. Il faudra aussi voir à améliorer l'affichage, mais ces problèmes, tels la gestion de la caméra ou bien la création d'une liste d'affichage, relèvent principalement du module d'affichage. Finalement, plusieurs petites tâches de gestion restent à coder. Notons par exemple, la sauvegarde d'une couche ou bien le retrait d'une couche à une partition. Cela ne devrait toutefois pas représenter un effort considérable.

8 ANNEXE : Diagramme de classes


MTInfoPartition
MTPartition
PMInfoPartition
PMInfoPartition(PMPar...
~PMInfoPartition()
PMPartition
m_nomMailleur1D : string
m_nomMailleur2D : string
m_paramMailleur1D : string
m_paramMailleur2D : string
m_mailleur1DestDll : Booleen
m_mailleur2DestDll : Booleen
PMPartition()
~PMPartition()
virtual ERMsg ajouteCouche(MTCouche P)
ERMsg asgMailleur1D(const string&, const std::string&, const Bo...
ERMsg asgMailleur2D(const string&, const std::string&, const Bo...
oonst Booleen& mailleur1DestDll()
const Booleen& mailleur2DestDll()
const string& reqNomMailleur1D()
conststd() : :string&reqNomMailleur2D()const
conststd() : :string&reqParamMailleur1D()const
const string& reqParamMailleur2D()
MTInfoCouche
MTCouche
PMInfoCouche
PMObjetGraohqieMaillage\Variable P m_maillage Precedent P
PMObjetGraphiqueMaillage\Variable P m_maillage Valide P
PMInfoCouche(PMCouche P)
virtual ~PMInfoCouche()
ERMsg asgMaillage Precedent(PMObjetGraphiqueMaillage\Va...
ERMsg asgMaillage\Valide()
invalideMaillage()
PMObjetGrapihqueMaillage\Variable P reqMaillage Precedent()
PMObjetGraphiqueMaillage\Variable P reqMaillage\Valide()
PMCouche
TCChampErrP m_champErrP
TCNiveauTol m_niveauTol
std::string m_nomMailleur1D
std::string m_nomMailleur2D
std::string m_paramMailleur1D
std::string m_paramMailleur2D
PMCouche()
virtual ~PMCouche()
ERMsg asgChampErr(TCChampErrP)
ERMsg asgMailleur1D(const std::string&, const s...
ERMsg asgMailleur2D(const std::string&, const s...
ERMsg asgNiveauTol(const TCNiveauTol&)
const TCChampErr& reqChampErr() : const
TCNiveauTol reqNiveauTol() : const
const std::string& reqNomMailleur1D() : const
const std::string& reqNomMailleur2D() : const
const std::string& reqParamMailleur1D() : const
const std::string& reqParamMailleur2D() : const
PMObjetGraohqueMaillage\Variable
void* couleurNoEeemnt
Booleen afficheNoelement
Booleen m_ombre


PMInfoCouche
PMObjetGraohqieMaillageVariableP m_maillagePrecedentP
PMObjetGraphiqueMaillageVariableP m_maillageValideP
PMInfoCouche(PMCoucheP)
virtual ~PMInfoCouche()
ERMsg asgMaillagePrecedent(PMObjetGraphiqueMaillageVariableP)
ERMsg asgMaillageValide()
invalideMaillage()
PMObjetGraphqueMaillageVariableP reqMaillagePrecedent()
PMObjetGraphiqueMaillageVariableP reqMaillageValide()
PMObjetGraohiqueMaillageVariable
void* couleurNoEleemnt
Booleen afficheNoelement
Booleen m_ombre
PMObjetGraphiqueMaillageVariable()
~PMObjetGraphiqueMaillageVariable()
virtual Booleen appartient(const AFObjetGraphiqueP) : const
virtual TypeObjetGraphique reqType() : const
virtual ERMsg synchroniseActeur()
ERMsg ajouteMaillage(const ERMaillage&)
ERMsg asgAfficheNoElement(Booleen)
ERMsg asgCouleurNoElement(void*)
ERMsg asgombre(Booleen)
ERMsg copieActeur(PMObjetGraphiqueMaillageVariable&)
Booleen estOmbre() : const
Booleen reqAfficheNoElement() : const
void* reqCouleurNoElement() : const
ERMsg videActeur()


PMInterfaceMailleur
Dimension m_dimension
PMInterfaceMailleur()
~PMInterfaceMailleur()
ERmsg asgDimension(const Dimension&)
vitrual ERMsg configure(const s td::string&) : =0
virtual ERMsg mailleLigne(TCMaillage&, const TCChampErr&, const TCNiveauTol&, const std::string&, const TCCoord& c0, const TCCoord& c1) : const ...
virtual ERMsg mailleZone(TCMaillage&, const TCChampErr&, const TCNiveauTol&, const std::string&, const TCMaillage&) : const = 0
const Dimension& regDimension()
PMMailleurProxy
SODll m_mailleurDll
std::auto_ptr<PMInterfaceMailleur> m_mailleurP
std::string m_nomDll
PMMailleurProxy(const std::string&)
virtual ~PMMailleurProxy()
virtual ERMsg configure(const std::string&)
vitrual ERMsg mailleLigne(TCMaillage&, const TCChampErr&, const TCNiveauTol&, const std::string&, const TCCoord& c, const TCCoord& c1) : const
virtual ERMsg mailleZone(TCMaillage&, const TCChampErr&, const TCNiveauTol&, const std::string&, const TCMaillage&) : const
PMMailleurFrontal
virtual ERMsg configure(const std::string&)
virtual ERMsg mailleLigne(TCMaillage&, const TCChampErr&, const TCNiveauTol&, const std::string&, const TCCoord& c0, const TCCoord& c1) : co...
virtual ERMsg mailleZone(TCMaillage&, Const TCChampErr&, const TCNiveauTol&, const std::string&, const TCMaillage&) : const
PMMailleurFactory
TCContainre m_mapConstructeurs
PMMailleurFactory()
~PMMailleurFactory()
ERMsg ajouteMailleur(const std::string&, const PMMailleurCallBack&)
ERMsg creeMailleur(PMInterfaceMailleurP&, const std::string&) : const
ERMsg retireMailleur(const std::string&)
PMAlgoMaille


PMMailleurFactory
PMInterfaceMailleur
PMInfoPartition
PMGestionnaire
IGCommande m_commandes
Etat m_etatModule
TCContainer m_mapFenetrePartition
PMMailleurFactoryP m_mailleurFactoryP
PMGestionnaire(const TECallBack&, const TEEm...
~PMGestionnaire()
ERMsg initialiseCommandesSupportees()
virtual ERMsg recoisEvenVirtuel(TEEvenement&)
PMAlgoMaille
PMAlgoAssemble m_assembleP
PMInterfaceMailleurP m_mailleur1DP
PMInterfaceMailleur m_mailleur1DP
PMInterfaceMailleurP m_mailleur2DP
ConstPMPartitionP m_partitionP
TCContainerPMInfoCouche m_couchesAMailler
EntierN m_prochainNiveauAMailler
PMAlgoMaille()
~PMAlgoMaille()
ERMsg operator()(TCConteinrPMInfoCouche&)
PMAlgoDemaille
TCContainerNiveauCoucheAffectees m_niveauxAffectes
ConstMTPartitionP m_partitionP
PMAlgoDemaille()
~PMAlgoDemaille()
ERMsg calculeAffectee(const MTCoucheP, const Entier&)
ERMsg configure(ConstMTPartitionP)
TCNiveauCoucheAffecteesI reqDebut() : const
TCNiveauCoucheAffecteesI reqFin() : const
ERmsg vide()
PMAlgoAssemble
PMAlgoAssemble()
virtual ~PMAlgoAssemble()
virtual ERMsg ajoute(const ERMaillage&, PMInfocouche&) : =0
PMAlgoAssembleVTK
PMAlgoAssembleVTK()
virtual ~PMAlgoAssembleVTK()
virtual ERMsg ajoute(const ERMaillage&, PMInfoCouche&)

RAPPORT # 7 : Module Projet

Rapport de développement logiciel

Rapport final sur le Module Projet

Présenté par :

Stéphane Lévesque

18 janvier 2005

Versions du document

Date	Auteur	Commentaires	Version
11-08-04	Stéphane Lévesque	Version Initiale	V1.0

1. Introduction

1.1 Objectifs

Le but de ce rapport est de présenter les différentes composantes du Module Projet et ses implications sur la base de données, les métadonnées, les liens externes, le Module Gestionnaire BD et l'interface graphique.

Ce rapport explique les principes de fonctionnement du Module Projet, de son interface graphique, des métadonnées ainsi que les modifications portées sur les structures existantes, soient la base de données et le Module Gestionnaire BD.

1.2 Référence

- Spécifications – Modeleur 2.0, Québec, INRS-ETE.
- Rapport de réunions – Réunion 1 du Module Projet, Québec, INRS-ETE.
- Rapport de réunions – Réunion 2 du Module Projet, Québec, INRS-ETE.
- Rapport de réunions – Réunion 3 du Module Projet, Québec, INRS-ETE.

1.3 Contexte

La première version de Modeleur fonctionnait avec des fichiers. Chaque entité avait un ou plusieurs fichiers pour le décrire. Un projet était un répertoire contenant tous les fichiers des entités.

Modeleur 2.0 fonctionne maintenant avec une base de données qui contient plusieurs projets et leurs entités. Cette conversion a entraîné le développement d'une base de données adaptée aux nouveaux besoins, d'un Module Gestionnaire BD pour les transferts entre la base de données et la mémoire, ainsi qu'un traducteur pour importer dans la base de données les anciens projets de Modeleur 1.0.

Le partage de certaines entités entre les projets était une fonctionnalité importante pour Modeleur. Son implantation touche à la fois la structure de la base de données et le Module Gestionnaire BD qui veille à son bon fonctionnement.

La centralisation de ces données demande une manipulation et un entretien particulier pour gérer les projets et leurs entités. Ces tâches sont la responsabilité du Module Projet et de ses annexes dans le Module Gestionnaire BD.

Aussi, ces manipulations requéraient parfois des informations provenant de l'extérieur. Le Module Projet interagit donc avec l'utilisateur au moyen d'une interface graphique axée surtout autour de la complétion de formulaire.

Enfin, il a été jugé utile de conserver des informations de type générique sur les entités que contiennent les projets. Le principe de métadonnées, inspiré du standard « FGDC » utilisé par Environnement Canada, nous permet donc d'enregistrer certaines informations tel que le libellé, la description, la date de création et la date de modification sur les entités principales de Modeleur.

1.4 Structure

Ce document commence avec une présentation de la base de données développée pour Modeleur 2.0 ainsi que des explications quant aux modifications qui y ont été apportées pour supporter la séparation en schémas. Ensuite, le fonctionnement du Module Projet et son interface graphique sera expliqué, suivit de ses diagrammes de séquences. Par la suite, le Module Gestionnaire BD est présenté brièvement, suivi des ajouts qui lui ont été fait pour prendre en charge les projets, les liens externes et les métadonnées.

2 Structure de la base de données

2.1 Introduction à la base de données

La base de données contient les principaux types de données qui seront supportés par Modeleur 2.0. On y retrouve les partitions, les maillages, les espaces de représentation, les séries, les plans, les semis de topographies, les modèles numérique de terrain (MNT), les couches et les objets graphiques. Ces entités principales de Modeleur 2.0 sont les seules à avoir des métadonnées d'associées et à pouvoir être exportable sous forme de lien externe. Chaque type se voit attribuer une table principale contenant des informations utiles à son traitement. Rattachées à cette dernière se trouvent des tables secondaires pour spécialiser un type ou pour former une agrégation de plusieurs entités. Dans ces cas, un lien est créé entre la table principale et la table secondaire.

Les liens entre les tables sont faits grâce à leur identifiants (id) qui est un entier sur 64 bits, offrant 9 223 372 036 854 775 807 possibilités. Les versions précédentes de la base de données utilisaient des entiers sur 32, avec 2 147 483 647 possibilités, ce qui était suffisant pour la plupart des tables. Cependant, la centralisation de tous les projets au sein d'une même base de données augmentait considérablement le nombre d'entrées dans certaines tables, en particulier les points de topographie et les nœuds de maillage. La conversion vers les entiers sur 64 bits évite de devoir gérer le nombre de possibilités restantes lorsque les projets contiennent de nombreuses entités. Afin de standardiser le type des identifiants, toutes les tables associées au type de Modeleur 2.0 ont été révisées pour avoir des clés primaires et des clés étrangères sur 64 bits.

2.2 Séparation des projets dans la base de données

2.2.1 Séparation abstraite

Dans les versions précédentes de la base de données, la séparation des projets se faisait de façon abstraite. Une table 'projet' contenait les identifiants et les informations des projets de la base de données et chaque type important contenait dans sa table principale un champ pour faire le lien avec le projet auquel il était associé.

Séparée de cette manière, la base de données n'offrait pas de sécurité quant aux entités des autres projets. N'importe qui ayant un accès à la base de données pouvait lire, modifier ou supprimer des entités n'appartenant pas à son projet.

De plus, avec l'utilisation de GIS, les tables se voient régulièrement ajouter des colonnes. En effet, à chaque fois qu'une entité est dans une nouvelle projection, GIS nécessite la création d'une nouvelle colonne pour effectuer ses requêtes. Lorsque tous les projets utilisent les mêmes tables, le nombre de ces colonnes augmente rapidement, ce qui exige temps et espace.

2.2.2 Séparation en schémas

Pour contrer les problèmes cités précédemment, nous avons opté pour la solution de séparation en schémas : pour chaque projet, un schéma lui est associé.

Un schéma est une division de la base de données qui permet de contenir des tables, des séquences et des autorisations différentes des autres schémas. Cette indépendance était toute désignée pour résoudre le problème de la séparation des projets. Ainsi divisée, la base de données peut limiter l'accès aux entités d'un projet en modifiant les autorisations du schéma lui étant associé. Aussi, en ayant des tables séparées des autres projets, le nombre des colonnes GIS se trouve limité au nombre de projections que contient le projet du schéma, et non plus à l'ensemble des projets contenus dans la base de données.

Il est à noter que les schémas ne sont pas pris en charge par tous les systèmes de gestion de bases de données. Cette solution nous empêche donc d'être compatible avec de tels SGBD, dont MS Access.

2.2.2.1 Schéma 'public'

Pour mettre en place cette structure, un schéma accessible à tous, nommé 'public', contient les informations communes à chaque projet. Sa création est faite au moyen du script SQL 'creerSchemaPublic.sql' disponible dans le répertoire BD.

C'est dans ce schéma que se trouve la table 'spatial_ref_sys' contenant les localisations spatiales et les tables 'projection_base' et 'projection_interne' faisant le lien entre les projections et les projets. Le schéma public contient aussi une table nommée 'projet' détenant tous les identifiants et les noms des schémas associés aux projets de la base de données. Une table 'geometry_columns', créé automatiquement par le script GIS pour gérer ses colonnes, appartient aussi au schéma 'public'.

Le schéma 'public' contient aussi des séquences pour gérer les identifiants de ses tables. Une séquence est un objet qui permet de délimiter des nombres entiers. Elle permet de spécifier le minimum, le maximum, l'incrémentation et la valeur de départ. La fonction 'currval()' permet de savoir à quelle valeur est rendue une séquence tandis que la fonction 'nextval()' incrémente la séquence et retourne cette nouvelle valeur.

Chaque identifiant se voit attribuer une séquence portant le nom de la table de l'identifiant, le nom du champ de l'identifiant et une terminaison '_seq'. Par exemple, la séquence de l'identifiant des projets, 'projet_id', porte le nom 'projet_projet_id_seq'. Ces séquences permettent de savoir en tout temps la valeur d'un identifiant, ce qui permet, entre autres, de récupérer l'id de la dernière entité pour un type donné ou d'éviter d'utiliser deux fois la même valeur pour un identifiant.

2.2.2.2 Schéma de projet

Les schémas de projets contiennent toutes les tables et les séquences nécessaires à la création d'un projet de Modeleur 2.0. Sa création est faite au moyen du script SQL 'creerSchemaProjet.sql' disponible dans le répertoire BD.

La création d'un schéma est faite à chaque fois qu'un nouveau projet est créé. Toutes les tables et les séquences relatives aux entités sont alors créés. Une table 'parametre_projet' est aussi créée dans ce schéma. Cette table ne contient qu'une seule entrée qui conserve les paramètres du projet associé au schéma, soit la projection de base et la projection interne, le décalage sur les X et sur les Y, ainsi que les coordonnées pour la zone d'affichage.

2.2.2.3 Implantation des liens externes

L'implantation des schémas vient compliquer un peu la notion de liens externes. Bien qu'un projet contienne des entités, il doit parfois faire référence à des entités appartenant à d'autres projets. Au lieu de copier cette entité dans son schéma, il est plus avantageux de lui donner le droit d'utiliser cette entité sans la modifier. C'est alors qu'interviennent les liens externes.

Un lien externe est un lien avec une entité d'un autre projet, une entité externe, créé de telle sorte que le projet puisse l'utiliser tout comme s'il s'agissait d'une entité normale. Son implantation modifie quelque peu la structure du schéma de projet.

Premièrement, dans chaque schéma de projet se trouve une table nommée 'lien_externe' permet de recenser toutes les entités externes. Elle contient le type de l'entité, un identifiant local, ainsi que l'identifiant du projet d'où l'entité origine et l'identifiant qu'elle a dans ce projet. Toutes ces données sont nécessaires à la redirection des requêtes, expliquée plus en détails au point 5.2.3.

L'identifiant local est l'identifiant que le projet utilise à l'interne. Cet identifiant prend la valeur incrémentée de la séquence associée à son type, tout comme les autres entités normales. Une entrée est aussi ajoutée dans la table associée à son type, une entrée 'par défaut'. Cette entrée est une copie de l'entrée ayant l'identifiant 0 dans la même table. L'entrée 0 est créé dans le script de création du schéma de projet et tous ses champs sont mis à une valeur par défaut. Les informations de l'entrée 'par défaut' n'ont pas d'importance puisque les vraies informations sont contenues dans le projet externe que l'on peut retrouver dans la table 'lien_externe'.

Ce procédé permet d'utiliser des entités en référence tout en empêchant que les valeurs des identifiants se croisent et en contournant les problèmes d'intégrités référentielles dus aux clés étrangères.

3 Module Projet et son interface graphique

3.1 Rôles du Module Projet

Le Module Projet permet de faire le pont entre les fonctions de gestion des projets et l'usager de Modeleur 2.0. Ce module permet aussi aux autres modules du programme d'interfacer avec l'utilisateur au moyen de boîtes de dialogues programmées en python. Les images des boîtes de dialogues préliminaires comportant quelques annotations sont disponibles en annexes.

3.2 Détails des fonctions

La fonction 'traiteEvenNouveau' permet de créer un nouveau projet dans la base de données. Cette fonction demande la liste de tous les identifiants des projections GIS, liste utile pour que l'usager spécifie une projection. Elle affiche ensuite la boîte de dialogue 'Informations du projet' pour que l'usager puisse remplir les paramètres utiles à la création. Ces paramètres sont les suivants :

- le nom du projet;
- la projection de base;
- le décalage sur les X et les Y;
- la zone d'affichage.

Si les informations sont remplies correctement, la fonction envoie l'événement de création d'un projet avec ces paramètres.

La fonction 'traiteEvenChoisir' permet de choisir un projet existant parmi tous ceux que contient la base de données. Cette fonction demande la liste des projets existants et affiche la boîte 'Ouvrir projet' pour que l'usager puisse choisir un projet. Si un projet est choisi, la fonction retourne l'identifiant du projet choisis, sinon, l'identifiant est 'ABSENT_BD' ou –1.

La fonction 'traiteEvenOuvrir' permet d'ouvrir un projet existant parmi tous ceux que contient la base de données. Cette fonction utilise 'traiteEvenChoisir' pour obtenir l'identifiant d'un projet et envoie l'événement d'ouverture d'un projet avec cet identifiant.

La fonction 'traiteEvenFermer' permet de fermer le projet ouvert dans Modeleur 2.0. Cette fonction envoie l'événement d'ouverture d'un projet avec l'identifiant 'ABSENT_BD' ou –1. Prochainement, cette fonction vérifiera si les entités en mémoire doivent être sauvegarder avant de fermer le projet.

La fonction 'traiteEvenChoisirEntite permet de choisir une entité du projet, limitée à un type ou non. Cette fonction demande la liste des entités du projet selon un certain type. Si ce type est 'TOUS', toutes les entités du projet seront listées. Elle affiche ensuite la boîte de dialogue 'Choisir une entité' pour que l'usager puisse faire son choix. Si une entité est choisie, la fonction retourne l'identifiant et le type de l'entité choisie, sinon, l'identifiant est 'ABSENT_BD' ou –1.

La fonction 'traiteEvenGestionEntite' permet de gérer les entités du projet. Cette fonction demande la liste de toutes les entités du projet et affiche la boîte de dialogue 'Gestion des entités'.

Le bouton 'Créer un lien externe' permet de créer un nouveau lien externe dans le projet. En cliquant sur le bouton, une boîte de dialogue 'Ouvrir projet' permet de choisir dans quel projet l'entité d'origine se trouve. Si l'usager ne choisit aucun projet ou s'il prend le même projet, un message d'erreur lui est affiché. Après avoir choisi un projet valide, une boîte de dialogue 'Choisir une entité' permet de sélectionner l'entité d'origine vers laquelle le lien externe sera créé. Si une entité valide est choisie, le lien externe sera créé.

Le bouton 'Modifier les informations' permet de modifier les métadonnées associées à une entité du projet. En cliquant sur le bouton, une boîte de dialogue 'Modifier les informations' permet à l'usager de saisir les nouvelles métadonnées. Si aucune entité n'est choisie ou si un lien externe est sélectionné, un message d'erreur lui est affiché. Présentement, seuls le libellé et la description sont modifiables. Si l'usager valide ses modifications en appuyant sur 'OK', les métadonnées sont mis à jour dans la mémoire.

Le bouton 'Supprimer une entité' permet d'effacer une entité de la base de données. En cliquant sur le bouton, un message apparaît à l'usager pour confirmer la suppression. Si aucune entité n'est choisie ou si un lien externe est sélectionné, un message d'erreur lui est affiché.

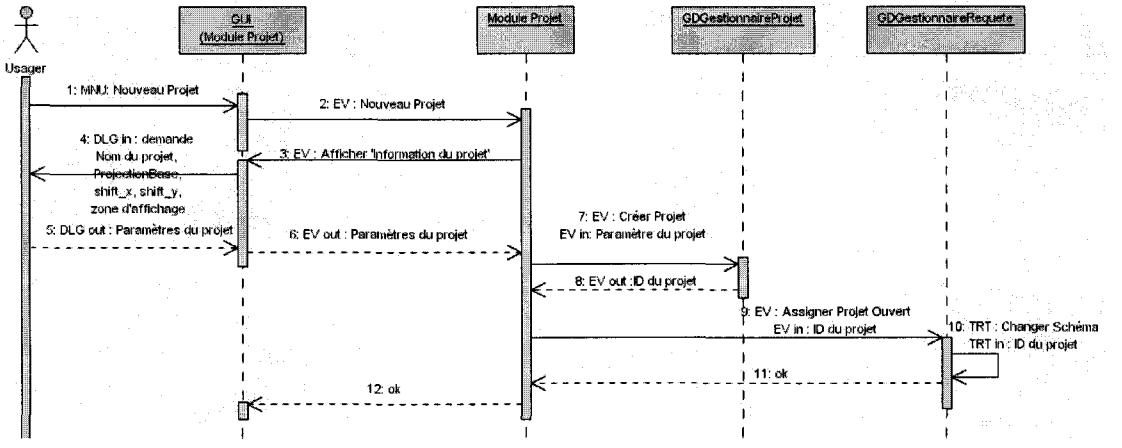
4 Communications entre les intervenants

4.1 Diagrammes de séquences

Les diagrammes de séquences permettent de bien représenter la répartition du travail entre les modules impliqués par une action Les diagrammes suivants schématisent différents scénarios faisant intervenir l'usager de Modeleur 2.0 et le Module Projet.

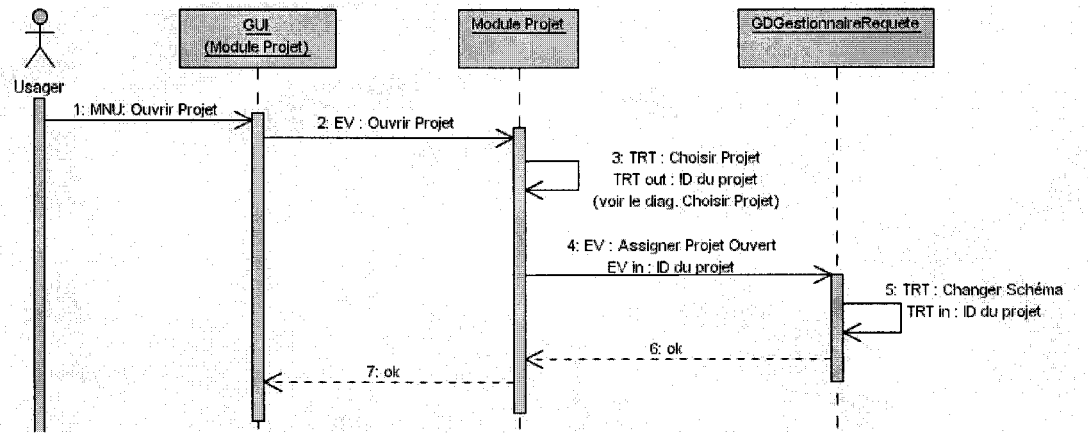
4.1.1 Nouveau Projet

Décrit les communications entre les modules et l'usager lorsque ce dernier crée un nouveau projet. Le projet créé devient par la suite le projet ouvert. À noter que le traitement 'Changer Schéma' redirige les requêtes sur le schéma associé à l'ID du projet reçu en paramètre.



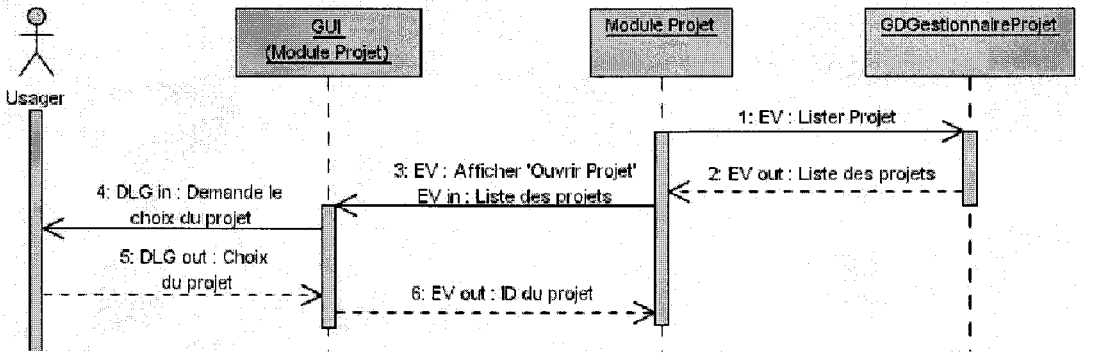
4.1.2 Ouvrir Projet

Décrit les communications entre les modules lorsqu'un projet doit être ouvert. À noter que cette séquence fait appel à la séquence **Choisir Projet** pour déterminer le projet à ouvrir. À noter que le traitement 'Changer Schéma' redirige les requêtes sur le schéma associé à l'ID du projet reçu en paramètre.



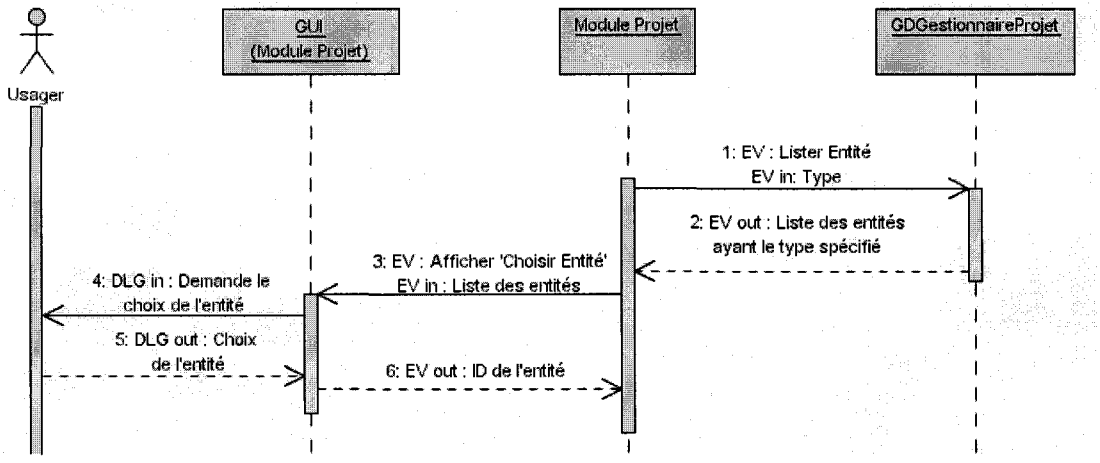
4.1.3 Choisir Projet

Décrit les différentes communications lorsque l'usager doit choisir un projet de la BD, par exemple pour l'ouvrir ou pour créer un lien externe. .



4.1.4 Choisir Entité

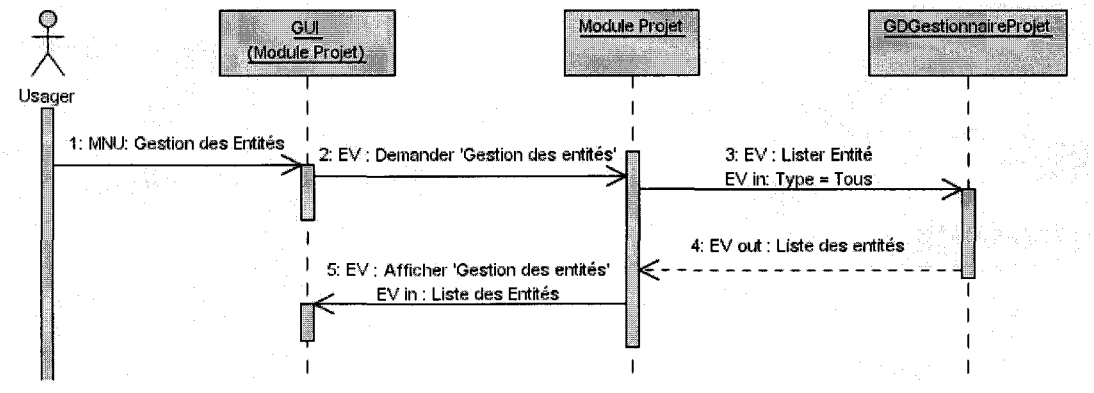
Décrit les communications entre le Module Projet, sa partie GUI et la partie GDGestionnaireProjet du Gestionnaire BD lorsque l'usager doit choisir une entité d'un certain type, par exemple lors de l'ouverture d'un maillage.



4.1.5 Gestion des entités

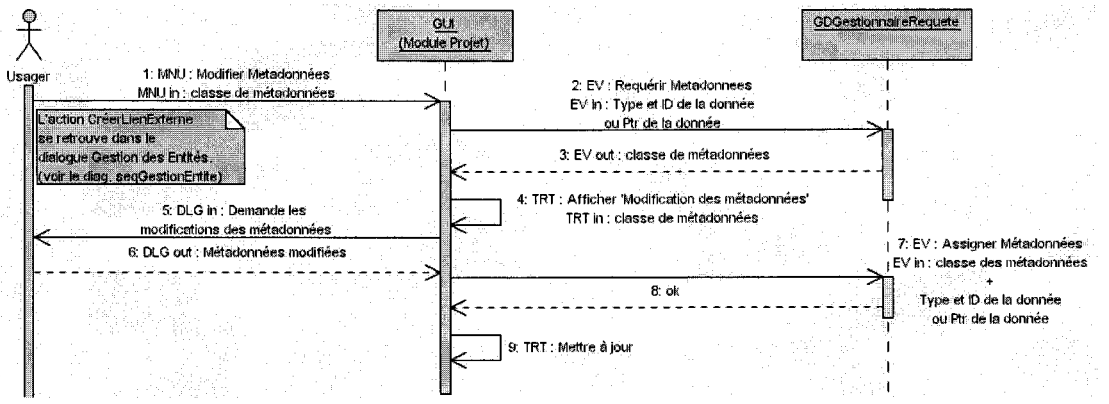
Décrit les communications entre le Module Projet, sa partie GUI et la partie GDGestionnaireProjet du Gestionnaire BD lors de l'ouverture de la boîte de dialogue 'Gestion des entités'. Cette partie permet les actions suivantes :

- créer un lien externe;
- supprimer une entité;
- modifier les métadonnées associées à une entité.



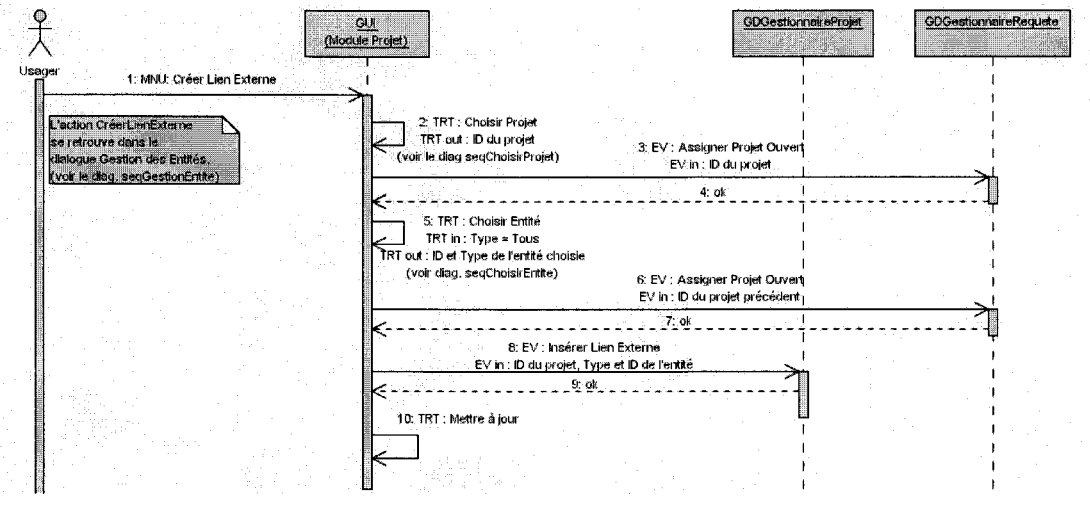
4.1.6 Modifier Métadonnées

Décrit les communications entre les différents modules et l'usager lorsque ce dernier modifie les métadonnées associées à une donnée du projet ouvert. Cette action est disponible dans la boîte de dialogue 'Gestion des entités'.



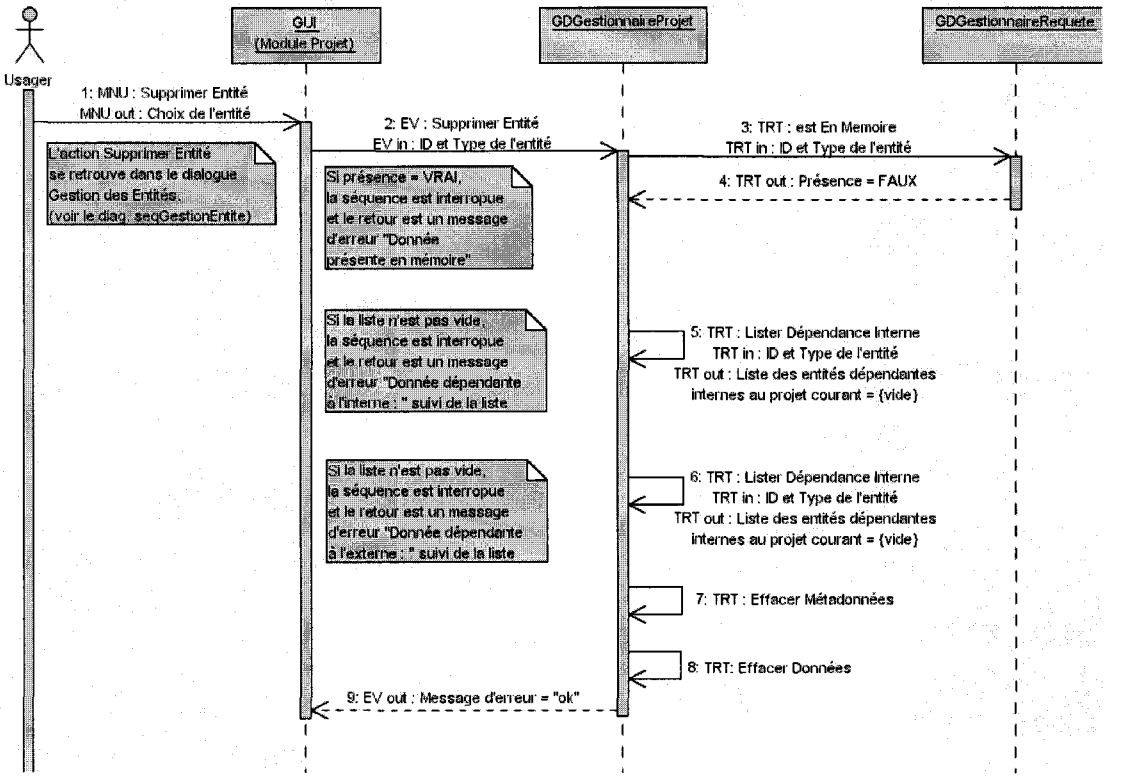
4.1.7 Créer Lien Externe

Décrit les communications entre le Module Projet, le Gestionnaire BD et l'usager lorsque ce dernier crée un lien externe pour utiliser une entité d'un autre projet. L'événement 'Changer Schéma' redirige les requêtes sur le schéma associé à l'ID du projet reçu en paramètre. Cette action est disponible dans la boîte de dialogue 'Gestion des entités'. À noter que le traitement pour mettre à jour n'est pas encore implanté.



4.1.8 Supprimer Entité

Décrit les communications entre le Module Projet, le Gestionnaire BD et l'usager lorsque ce dernier supprime de la BD une entité du projet actif. Le Module Projet doit s'assurer qu'il n'y a aucune dépendance, autant interne qu'externe, et que la métadonnée n'est pas en mémoire, puisqu'il la supprime en même temps que la donnée. Cette action est disponible dans la boîte de dialogue 'Gestion des entités'. À noter que les dépendances ne sont pas encore implantées.



4.2 Événements et structures de données

4.2.1 Module Projet

L'événement 'PJEvenNouveau' est intercepté par le Module Projet pour appeler la fonction 'traiteEvenNouveau'. Cet événement n'a aucun paramètre.

L'événement 'PJEvenOuvrir' est intercepté par le Module Projet pour appeler la fonction 'traiteEvenOuvrir'. Cet événement n'a aucun paramètre.

L'événement 'PJEvenChoisir' est intercepté par le Module Projet pour appeler la fonction 'traiteEvenChoisir'. Cet événement contient en paramètre l'identifiant du projet choisi par l'usager.

L'événement 'PJEvenChoisirEntite' est intercepté par le Module Projet pour appeler la fonction 'traiteEvenChoisirEntite'. Cet événement contient en paramètre le type d'entité à choisir par l'utilisateur. Les paramètres de sortie sont l'identifiant et le type de l'entité choisi.

L'événement 'PJEvenFermer' est intercepté par le Module Projet pour appeler la fonction 'traiteEvenFermer'. Cet événement n'a aucun paramètre.

L'événement 'PJEvenEnregistrer' est intercepté par le Module Projet pour appeler la fonction 'traiteEvenEnregistrer'. Cet événement n'a aucun paramètre.

L'événement 'PJEvenGestionEntite' est intercepté par le Module Projet pour appeler la fonction 'traiteEvenGestionEntité. Cet événement n'a aucun paramètre.

4.2.2 Module Gestionnaire BD

La structure de données 'infoProjet' permet de transiger des informations sur les projets de Modeleur 2.0. Cette structure contient l'identifiant du projet, le nom du schéma associé et l'identifiant de la projection de base.

La structure de données 'infoEntite' permet de transiger des informations sur les entités de Modeleur 2.0. Cette structure contient l'identifiant de l'entité, son type, son libellé et un booléen spécifiant s'il s'agit d'un lien externe.

La structure de données 'parametreProjet' permet de transiger les paramètres d'un projet de Modeleur 2.0. Cette structure contient l'identifiant du projet, le nom du projet, les projections et leurs identifiants, le décalage et la zone d'affichage.

L'événement 'GDEvenMetadonnees' est intercepté par la classe 'GDGestionnaireRequete' pour appeler différentes fonctions relatives aux métadonnées.

Cet événement contient en paramètre le type d'opération à effectuer. Il peut s'agir d'une assignation, d'une requête, du chargement ou du déchargement. Les autres paramètres sont un pointeur sur la classe GDMetadonnees, l'identifiant, le type et le pointeur d'une entité.

L'événement 'GDEvenChangeProjetActif' est intercepté par la classe 'GDGestionnaireRequete' pour appeler la fonction 'asgProjetActif'. Cet événement contient en paramètre l'identifiant d'un projet.

L'événement 'GDEvenReqProjetActif' est intercepté par la classe 'GDGestionnaireRequete' pour appeler la fonction 'reqProjetActif'. Cet événement contient en paramètre l'identifiant d'un projet.

L'événement 'GDEvenChangeProjetOuvert' est intercepté par la classe 'GDGestionnaireRequete' pour appeler la fonction 'asgProjetOuvert'. Cet événement contient en paramètre l'identifiant d'un projet.

L'événement 'GDEvenReqProjetOuvert' est intercepté par la classe 'GDGestionnaireRequete' pour appeler la fonction 'reqProjetOuvert'. Cet événement contient en paramètre l'identifiant d'un projet.

L'événement 'GDEvenSupprimeEntite' est intercepté par la classe 'GDGestionnaireProjet' pour appeler la fonction 'supprimeEntite'. Cet événement contient en paramètre l'identifiant et le type d'entité à supprimer.

L'événement 'GDEvenCreeLienExterne' est intercepté par la classe 'GDGestionnaireProjet' pour appeler la fonction 'creeLienExterne'. Cet événement contient en paramètre d'entrée l'identifiant et le type de l'entité d'origine et l'identifiant du projet d'origine. L'identifiant du lien externe créé est en paramètre de sortie.

L'événement 'GDEvenCreeProjet' est intercepté par la classe 'GDGestionnaireProjet' pour appeler la fonction 'creeProjet'. Cet événement contient en paramètre un pointeur sur la structure 'parametreProjet'.

L'événement GDEvenListeEntite' est intercepté par la classe 'GDGestionnaireProjet' pour appeler la fonction 'listeEntite'. Cet événement contient en paramètre le type de l'entité et un vecteur pour contenir la liste.

L'événement GDEvenListeProjet' est intercepté par la classe 'GDGestionnaireProjet' pour appeler la fonction 'listeProjet'. Cet événement contient en paramètre un vecteur pour contenir la liste.

L'événement GDEvenListeSrid' est intercepté par la classe 'GDGestionnaireProjet' pour appeler la fonction 'listeSrid'. Cet événement contient en paramètre un vecteur pour contenir la liste.

5 Module Gestionnaire BD

5.1 Introduction au Module Gestionnaire BD

Le gestionnaire de données sert en quelque sorte d'interface entre les différents modules de Modeleur 2.0 et les données contenues dans la base de données. Ce module a pour rôle de répondre aux requêtes sur les données de la BD, de gérer les entités et leurs métadonnées et de veiller à la redirection entre les schémas.

Ce module est constitué de plusieurs classes. La plupart sont des spécialisations de la classe 'GDAlgoIo' pour effectuer les traitements de création, de chargement et de sauvegarde sur une entité en particulier. Par exemple, GDAlgoIoMaillage est une classe dérivée de GDAlgoIo qui se fait appeler par la classe GDGestionnaireRequete lorsqu'une opération sur les maillages est demandée.

La classe GDGestionnaireRequete reçoit la plupart des événements du Module Gestionnaire BD. Une de ses fonctions est de rediriger les traitements de création, de chargement et de sauvegarde vers le bon algorithme. Elle s'occupe aussi de garder des traces des entités déjà en mémoire.

La classe BDConnection s'occupe principalement de la connexion ODBC avec la base de données, bien qu'elle effectue aussi quelques requêtes sur les identifiants et les colonnes GIS.

5.2 Modifications apportées au Module Gestionnaire BD

Plusieurs changements ont été apportés au Module Gestionnaire Projet. Comme ce module est très proche de la base de données, un changement dans sa structure a souvent des répercussions sur le code du Gestionnaire BD.

5.2.1 Modifications relatives aux identifiants sur 64 bits

Un de ces changements importants est l'adoption des identifiants de 64 bits. Modeleur 2.0 doit pouvoir se faire compiler sur différents compilateurs, dont GCC 3.3, Intel 7.1 et Visual Compiler 7.0 et 7.1. Son code doit être le plus standard possible pour éviter les conflits entre compilateurs.

Un problème avec les entiers sur 64 bits est qu'ils ne sont pas encore standards sur tous les compilateurs. En effet, bien que la plupart des compilateurs utilisent le type 'signed long long int' pour définir un entier sur 64 bits, d'autres compilateurs comme celui de Borland et de Microsoft utilise le type '__int64'. Pour contrer ce problème, lors de la définition des types dans le fichier 'sytypes.h', une vérification est faite pour savoir si le compilateur provient de Microsoft. Si tel est le cas, Entier64 est défini comme '__int64' et EntierN64 comme 'unsigned __int64', sinon, ils sont définis comme 'signed long long int' et 'unsigned long long int'.

La librairie OTL utilise quant à elle un type qu'elle nomme 'bigint'. Afin de compiler correctement la librairie, il faut s'assurer de définir OTL_BIGINT au bon type d'entier sur 64 bits avant d'inclure le fichier 'otlv4.h'. Ce type est présentement défini dans le fichier 'BDConnection.h'.

5.2.2 Modifications relatives à la séparation en projet

L'avènement des projets a aussi nécessité l'ajout de plusieurs fonctions dans le Module Gestionnaire BD.

La classe 'GDGestionnaireRequete' a été passablement modifiée et la classe 'GDGestionnaireProjet' a dû être créée. Afin de bien cibler les changements et les ajouts faits dans ces classes, chacune des fonctions est expliquée sans toutefois trop entrer dans les détails.

5.2.2.1 Modifications dans la classe 'GDGestionnaireRequete'

La fonction 'changeSchema' permet de changer le schéma dans lequel les requêtes sur la base de données sont effectuées. Cette fonction recherche le nom du schéma associé à l'identifiant de projet reçu en paramètre en faisant une requête sur la table 'projet' dans le schéma 'public'. Elle change ensuite la variable 'search_path' de la base de données pour inclure ce nom de schéma dans la recherche. Si l'identifiant de projet est égal à ABSENT_BD ou –1, la variable 'search_path' est changée pour ne contenir que le schéma 'public'.

Les fonctions 'asgProjetActif' et 'reqProjetActif' permettent de déterminer dans quel projet les requêtes sur la base de données auront lieu. La fonction 'asgProjetActif' change le schéma actif pour celui reçu en paramètres. Ces fonctions sont utiles entre autres pour l'ouverture d'un projet et pour les redirections causées par les liens externes (voir le point 5.2.3 pour plus de détails).

Les fonctions 'asgProjetOuvert' et 'reqProjetOuvert' permettent de déterminer quel est le projet ouvert dans Modeleur 2.0. C'est dans le schéma associé à ce projet uniquement que des modifications (des sauvegardes d'entités) peuvent avoir lieu. Si la fonction 'asgProjetOuvert' est appelée avec l'identifiant de projet égal à 'ABSENT_BD' ou –1, le Module Projet procède à la fermeture du projet.

Temporairement, les données restantes dans le projet à sa fermeture d'un projet sont déchargées de la mémoire. Normalement, il ne devrait rester aucun donnée en mémoire car les modules responsables de ces données auraient dû les décharger. Toutefois, les procédures de déchargement ne sont implantées que dans certains modules actuellement.

5.2.2.2 Création de la classe 'GDGestionnaireProjet'

Afin de ne pas alourdir la classe 'GDGestionnaireRequete', la plupart des fonctions de gestion des projets et des entités ont été regroupées dans une nouvelle classe 'GDGestionnaireProjet'

La fonction 'creeProjectionInterne' permet d'insérer une nouvelle entrée dans la table 'spatial_ref_sys' du schéma 'public'. Cette fonction extrait les informations d'une projection déjà existante et crée une nouvelle entrée avec celles-ci, en spécifiant un identifiant supérieur à 1 000 000 et en inscrivant comme nom d'auteur 'INRS'. Cette mesure sert à différencier les projections réelles de celles ajoutées pour le fonctionnement interne de Modeleur 2.0.

La fonction 'creeSchema' permet de créer un nouveau schéma et de l'associer à un projet. Cette fonction crée le schéma dans la base de données, redirige de façon temporaire les requêtes au moyen de la fonction 'asgProjetActif' et remplit le schéma en exécutant le script SQL 'creerschemaprojet.sql'. Ce fichier doit se trouver dans le même répertoire que l'exécutable.

La fonction 'creeProjet' permet de créer un projet ainsi qu'un schéma qui lui sera associé. Cette fonction effectue une vérification dans base de données pour savoir si le nom du projet existe déjà. En effet, présentement, le nom de projet sert aussi de nom de schéma. Par conséquent, il souffre des mêmes limitations que les noms de schéma. Il doit donc commencer par une lettre, ne comporter aucun espace ou caractère spécial, ne doit pas dépasser 30 caractères et il doit être exclusif. Ceci pourrait être changé en cessant d'utiliser le même nom pour le projet et le schéma.

Si le nom est valide, la fonction poursuit en créant une projection interne à l'aide de la fonction 'creeProjectionInterne'. Ensuite, elle crée des entrées dans la table 'projection_base' et 'projection_interne' et 'projet' du schéma 'public'. Si tout s'est bien déroulé, elle crée un schéma portant le nom du projet à l'aide de la fonction 'creeSchema' et remplit la table 'parametre_projet' du nouveau schéma avec les paramètres de projet qu'elle a reçus en argument.

La fonction 'supprimeEntite' permet de supprimer une entité du projet ouvert. Cette fonction vérifie que les entités ne sont pas déjà en mémoire avant de les supprimer. Prochainement, lorsqu'il sera possible de déterminer les dépendances, un traitement plus intelligent pourra être fait afin de déterminer si une entité peut être supprimée ou bien si elle est référée ailleurs.

La fonction 'listeProjet' permet de sortir la liste de tous les projets que contient la base de données. Cette fonction permet d'extraire une liste contenant pour chaque projet, son identifiant, son nom et sa projection.

La fonction 'listeEntite' permet de sortir la liste de toutes les entités du projet actif. Cette fonction permet d'extraire une liste contenant pour chaque entité, son identifiant, son type, son libellé ainsi qu'une variable indiquant s'il s'agit d'un lien externe. Il est possible d'appeler cette fonction en spécifiant le type d'entité devant se retrouver dans la liste. Si aucun type n'est spécifié, toutes les entités du projet actif seront listées.

La fonction 'listeSrid' permet de sortir la liste de toutes les projections de la base de données.

Les fonctions 'liste' qui viennent d'être explicitées sont utiles, entre autres, pour l'interface graphique du Module Projet.

5.2.3 Modifications relatives aux liens externes

L'implantation des liens externes a nécessité la modification et l'ajout de plusieurs fonctions dans le Module Gestionnaire BD. Les principales classes touchées sont 'GDGestionnaireProjet' et les classes dérivées de 'GDAlgoIo'.

5.2.3.1 Modifications dans la classe 'GDGestionnaireProjet'

La fonction 'estLienExterne' permet de déterminer si une entité est un lien externe ou une entité locale. Cette fonction recherche dans la table 'lien_externe' pour voir s'il existe une entrée avec le type et l'id passés en paramètres. Si une telle entrée existe, la fonction retourne VRAI ; ce qui signifie que la donnée est un externe, sinon, elle retourne FAUX ; ce qui signifie que la donnée est locale.

La fonction 'donneeOrigine' permet de déterminer le projet d'origine d'une entité ainsi que son identifiant dans ce projet. Cette fonction recherche dans la table 'lien_externe' pour voir s'il existe une entrée avec le type et l'id passés en paramètres. Si une telle entrée existe, elle extrait l'identifiant du projet d'origine et l'identifiant de l'entité d'origine et les retourne à la fonction appelante. Si aucune entrée n'est trouvée, la fonction retourne l'identifiant du projet ouvert et l'identifiant de l'entité reçue en paramètre.

La fonction 'creeLienExterne' permet de créer un lien externe vers une entité (ex : maillage) d'un autre projet. Dans le projet ouvert (projet en cours), une nouvelle entrée 'par défaut' est créée dans la table associée au type de l'entité ciblée (ex : table 'maillage'). On extrait ensuite l'identifiant de cette nouvelle entité pour pouvoir l'insérer dans la table 'lien_externe' afin de mettre en lien les informations de l'entité d'origine avec cet identifiant. Après cette étape, la table 'lien_externe' contient donc une nouvelle entrée associant le type de la donnée, l'identifiant qui vient d'être créé, l'id du projet externe et l'identifiant de la donnée externe.

5.2.3.2 Modifications dans les classes dérivées de 'GDAlgoIo'

La structure des classes dérivées de 'GDAlgoIo' a subi quelques modifications pour accueillir les liens externes. Maintenant, la fonction 'chargeDonnee' de tous les héritiers 'GDAlgoIo' (ex : GDAlgoIoMaillage) ne fait qu'exécuter la fonction 'chargeDonnee' de leur classe mère.

La classe mère s'occupe de gérer l'aspect de redirection des requêtes vers des liens externes s'il y a lieu de le faire. C'est également elle qui s'occupe de vérifier le compte de références pour déterminer si une donnée doit être chargée ou non. Cette nouvelle façon de faire permet de centraliser les traitements communs à un seul endroit.

La fonction 'chargeDonnee' de 'GDAlgoIo' vérifie donc si l'entité qu'elle doit charger est un lien externe. Si c'est le cas, elle redirige les requêtes vers le projet externe approprié à l'aide de la fonction 'asgProjetActif'.

Par la suite, si la donnée n'est pas déjà chargée en mémoire, il y a appel de la fonction 'traiteChargement' de la classe héritière (ex : GDAlgoIoMaillage). Ceci permet de procéder à l'étape concrète du chargement de l'entité ciblée (ex : maillage). Si l'entité n'est pas un lien externe, aucune redirection ne fut nécessaire et la fonction 'traiteChargement' est appelée directement.

Après l'appel à 'traiteChargement', s'il y a eu redirection, il doit y avoir réassignation du projet actif pour qu'il pointe à nouveau vers le « projet ouvert ». Finalement, dans les deux cas, le pointeur sur l'entité qui vient d'être chargée en mémoire est retourné.

5.2.4 Modifications relatives aux métadonnées

L'implantation des métadonnées a nécessité la modification et l'ajout de plusieurs fonctions dans le Module Gestionnaire BD. En plus de la création d'une nouvelle classe, détaillée au point 5.2.4.2, les classes 'GDGestionnaireRequete', 'GDGestionnaireProjet' et les classes dérivées de 'GDAlgoIo' ont aussi été modifiées.

Son implantation a aussi nécessité la création d'une nouvelle classe pour contenir et gérer ces informations. Présentement, la classe 'GDMetadonnees' ne contient que quatre métadonnées, le libellé, la description, la date de création et la date de modification.

Il est très probable que d'autres métadonnées s'ajouteront prochainement à cette classe, mais il faut garder à l'esprit que ces métadonnées doivent être générales car elles sont attribuées à plusieurs entités n'ayant pas nécessairement de liens ensemble. Il faut aussi prendre note que toutes les métadonnées d'un projet sont conservées en mémoire, il faut donc veiller à minimiser l'espace que prend cette classe.

5.2.4.1 Modifications dans la classe 'GDGestionnaireRequete'

La fonction 'chargeMetadonnees' permet de charger toutes les métadonnées du projet actif. Cette fonction, appelée après l'ouverture d'un projet, parcourt toutes les tables des entités principales de Modeleur 2.0 et charge en mémoire les métadonnées qui leurs sont associées. Si une entité se trouve à être un lien externe, les requêtes sont redirigées de façon temporaire le temps d'extraire les informations du projet d'origine.

La fonction 'sauveMetadonnees' permet de mettre à jour dans la base de données les métadonnées qui aurait été modifiées au cours des opérations dans le projet. À noter que les métadonnées des liens externes ne sont pas mises à jour car ces entités et leurs métadonnées associées ne peuvent être modifiées.

La fonction 'dechargeMetadonnees' permet de décharger toutes les métadonnées de la mémoire. Elle sauvegarde aussi les métadonnées qui ont été modifiées. Cette fonction est appelée avant la fermeture d'un projet.

Les fonctions 'asgMetadonnees' et 'reqMetadonnees' permettent d'assigner et de recevoir les métadonnées associées à une entité. Ces fonctions peuvent trouver une entité à partir de son identifiant et son type ou à partir de son pointeur si l'entité est déjà chargée en mémoire. Il est impossible d'assigner les métadonnées d'un lien externe puisque ces derniers ne sont pas modifiables.

La fonction 'creeMetadonneesBD' permet d'ajouter une entrée dans la table 'information', table qui conserve toutes les métadonnées du projet. Cette fonction crée une entrée avec des valeurs par défaut pour en récupérer l'identifiant. Cet identifiant sur les métadonnées est nécessaire lors de la sauvegarde dans la base de données des entités principales de Modeleur 2.0.

La fonction 'sauveDonnee' a été modifiée afin de mettre à jour les champs automatiques des métadonnées associées à l'entité sauvegardée. Présentement, seul le champ 'date de modification' est mis à jour.

La fonction 'ajouteDonnee' a aussi été modifiée pour faire le lien avec les métadonnées lors de l'ajout d'une entité en mémoire. Si l'entité ajoutée est un lien externe, on lui donne l'identifiant de son lien externe provenant du projet ouvert plutôt que son identifiant réel provenant de son projet d'origine. S'il s'agit d'une nouvelle entité, une métadonnées est créé en mémoire.

La fonction 'modifieDonneeId' a été modifiée pour mettre à jour le lien données / métadonnées lorsqu'une entité se voit changer son identifiant. Cette fonction est normalement appelée lorsqu'une entité nouvellement créée (son identifiant est donc égal à –1 ou ABSENT_BD) est sauvegardée pour la première fois et se voit attribuer un identifiant par la base de données.

La fonction 'enleveDonnee' a été modifiée pour effacer les métadonnées associées à une entité si cette dernière n'as pas été sauvegardée dans la base de données.

5.2.4.2 Création de la classe 'GDMetadonnees'

Les fonctions 'asgLibelle' et 'reqLibelle' permettent d'assigner et de recevoir la métadonnée 'libellé'. Cette métadonnée sert à identifier une entité en lui donnant un nom significatif. L'assignation vérifie que le libellé ne dépasse pas 256 caractères.

Les fonctions 'asgDescription' et 'reqDescription' permettent d'assigner et de recevoir la métadonnée 'description'. Cette métadonnée sert à documenter une entité en inscrivant des renseignements jugés pertinents. L'assignation vérifie que la description ne dépasse pas 1024 caractères.

La fonction 'verifieDateHeure' permet de vérifier qu'une chaîne de caractères contenant une date soit du bon format. Cette fonction retourne VRAI si le format est bon, FAUX sinon. Le format de la date est le suivant :

- 25/01/2004-12:01:01

Les fonctions 'asgDateCreation' et 'reqDateCreation' permettent d'assigner et de recevoir la métadonnée 'date de création' sous la forme d'une chaîne de caractères. Cette métadonnée sert à déterminer la date et l'heure de la création d'une entité. L'assignation vérifie que la date respecte le format. À noter que la fonction 'asgDCHeureCourante' permet d'assigner à la métadonnée 'date de création' la date et l'heure au moment de son appel.

Les fonctions 'asgDateModification' et 'reqDateModification' permettent d'assigner et de recevoir la métadonnée 'date de modification' sous la forme d'une chaîne de caractères. Cette métadonnée sert à déterminer la date et l'heure à laquelle une entité a été modifiée pour la dernière fois. L'assignation vérifie que la date respecte le format. À noter que la fonction 'asgDMHeureCourante' permet d'assigner à la métadonnée 'date de modification' la date et l'heure présent au moment de son appel.

L'opérateur '=' permet d'assigner toutes les informations d'une classe 'GDMetadonnees' à une autre. Cet opérateur retourne une référence sur son objet pour permettre des opérations en cascade.

6 Conclusion

6.1 Résumé

En résumé, ce rapport explique tous les changements qui ont été nécessaires à l'implantation des projets dans Modeleur 2.0 et de certaines autres fonctionnalités telles que les identifiants sur 64 bits et les métadonnées d'entités.

La structure de la base de données fut la plus touchée par l'implantation des projets. Son design a dû être corrigé pour supporter les schémas et les scripts SQL ont dû être réécrits.

Le Module Gestionnaire BD a eu plusieurs modifications et ajouts dans son code. En plus de la création d'une nouvelle classe pour gérer les projets et les entités, la plupart de ses classes existantes se sont faites modifier pour supporter les schémas, la redirection des liens externes et les métadonnées.

La classe GDMetadonnees est aussi une autre classe ajoutée au Module Gestionnaire BD. Son rôle est de contenir et de valider les métadonnées qui sont associées aux différentes entités d'un projet.

Enfin, le Module Projet, quant à lui, a eu comme rôle d'interagir entre l'utilisateur et toutes ces nouvelles fonctionnalités au moyen de son interface graphique, programmé en Python.

6.2 Perspectives futures

6.2.1 Métadonnées

Présentement, seules les métadonnées libellé, description, date de création et date de modification sont implantées dans Modeleur 2.0.

Pour ajouter d'autres métadonnées, par exemple pour suivre le standard 'FGDC' utilisé chez Environnement Canada, il suffit d'ajouter dans la classe GDMetadonnees une variable pour contenir l'information et des opérateurs pour l'assignation et la requête. Il faut aussi modifier le script python 'AfficheDlgModifierInfo' pour permettre à l'usager de les modifier.

6.2.2 Droits d'accès dans la base de données

La séparation de projets en schémas permet d'ajouter une notion de sécurité aux projets de Modeleur 2.0. En modifiant les droits d'accès du schéma associé à un projet, il est possible de restreindre la lecture et l'écriture du schéma, et même des tables, à certains usagers de la base de données. Ainsi protégé, le projet pourrait être ouvert en lecture seule par différents usagers sans risquer de contaminer les données. Aussi, les liens externes seraient limités aux projets dont l'usager a le droit de lecture.

6.2.3 Dépendances

L'implantation des dépendances n'étant pas faite, quelques fonctionnalités ne sont pas complètes.

Présentement, la suppression d'une entité ne donne aucun avertissement lorsqu'une autre entité dépend de celle-ci, qu'elle soit interne ou externe au projet. Aussi, la suppression de l'entité n'entraîne pas la suppression des autres entités qui la compose, par exemple les nœuds d'un maillage. La complétion des dépendances devrait donner les outils pour régler ces problèmes. Une solution possible pour les dépendances entre lien externe est détaillée dans le rapport de réunion 2 du Module Projet.

Aussi, dans la boîte de dialogue 'Gestion des entités', la liste des entités n'est affichée qu'en tant qu'arborescence à un seul étage. Les dépendances devraient permettre d'afficher l'arborescence des entités selon les dépendances qui les lient entre elles.

6.2.4 Utilisation de base de données externe

La notion de lien externe pourrait être améliorée afin de supporter le partage d'entité avec d'autres bases de données, en prenant pour acquis que la structure est identique. En ajoutant à la table de lien externe un champ pour contenir le nom du lien ODBC, il serait possible de créer une connexion pour acquérir les informations lorsque nécessaire.

Cependant, la structure actuelle de la base de données ne permettrait pas d'avertir le propriétaire d'une entité qu'elle dépend d'un projet externe à la base de données. La notion de balayage détaillée dans le rapport de réunion 2 est acceptable pour parcourir les projets de sa propre base de données, mais pas ceux de bases de données externes. La création d'une table de dépendance, locale à chaque projet et accessible en écriture aux usagers emprunteurs, permettrait de garder une trace des dépendances avec les autres projets des autres bases de données. Cette table pourrait aussi servir à noter les dépendances entre projets d'une même base de données si désiré.

6.2.5 Utilisation par les autres modules

Pour qu'un autre module puisse utiliser les fonctionnalités du Module Projet, il faut inclure le fichier 'PJEvenement.h'. Les événements qu'il contient permettent d'interagir avec l'usager pour obtenir différentes informations, par exemple l'identifiant d'une entité à charger. Un exemple de chargement d'un maillage est disponible dans le prototype de Modeleur 2.0.

Si le module désire faire des opérations sans l'aide de l'usager, il doit utiliser les événements disponibles dans le fichier 'GDEvenement.h'. Ceux-ci effectuent directement les opérations sur la base de données sans l'aide de l'usager si des informations valides sont fournies.

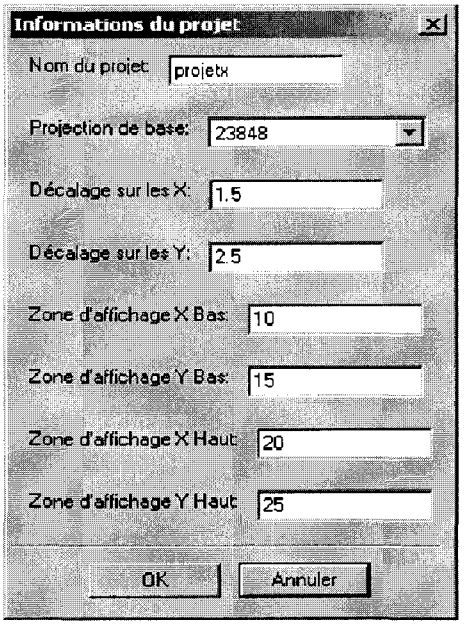
6.2.6 Parties incomplètes

Certaines parties du Module Projet ne sont pas encore implantées. Par exemple, la modification des paramètres d'un projet existant. Cette partie peut utiliser le script python 'AfficheDlgInfoProjet' pour permettre à l'usager de saisir les informations, mais il manque toujours une fonction dans la classe 'GDGestionnaireProjet' pour faire les changements dans la base de données.

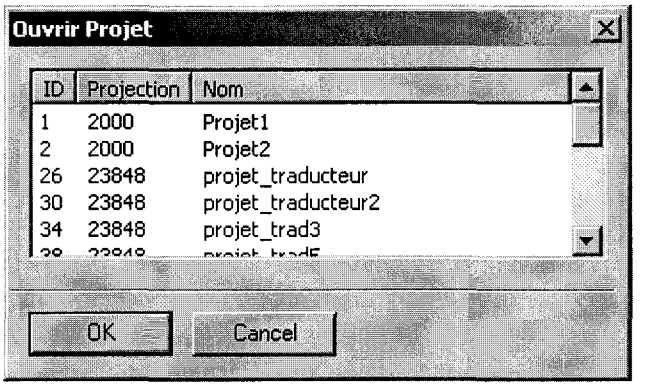
Aussi, une autre fonctionnalité qui pourrait être utile serait de trier et filtrer les entités dans les boîtes de dialogue 'Gestion des entités' et 'Ouvrir Entité'. Une documentation détaillée de la notion de filtrage est disponible dans les spécifications fonctionnelles de Modeleur 2, au point 5.6 Filtre de sélection.

7 Annexes

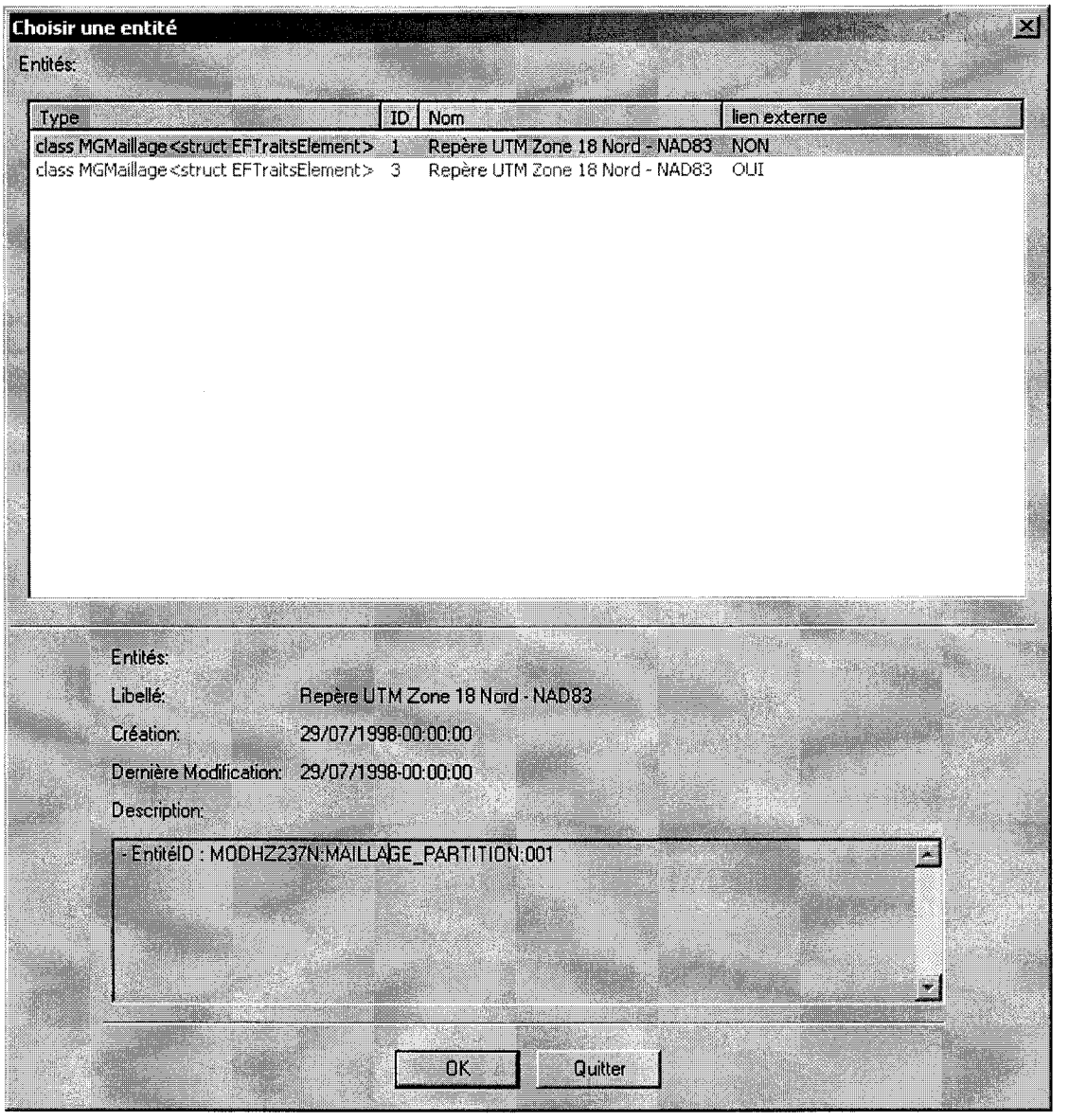
7.1 Boîtes de dialogues préliminaires



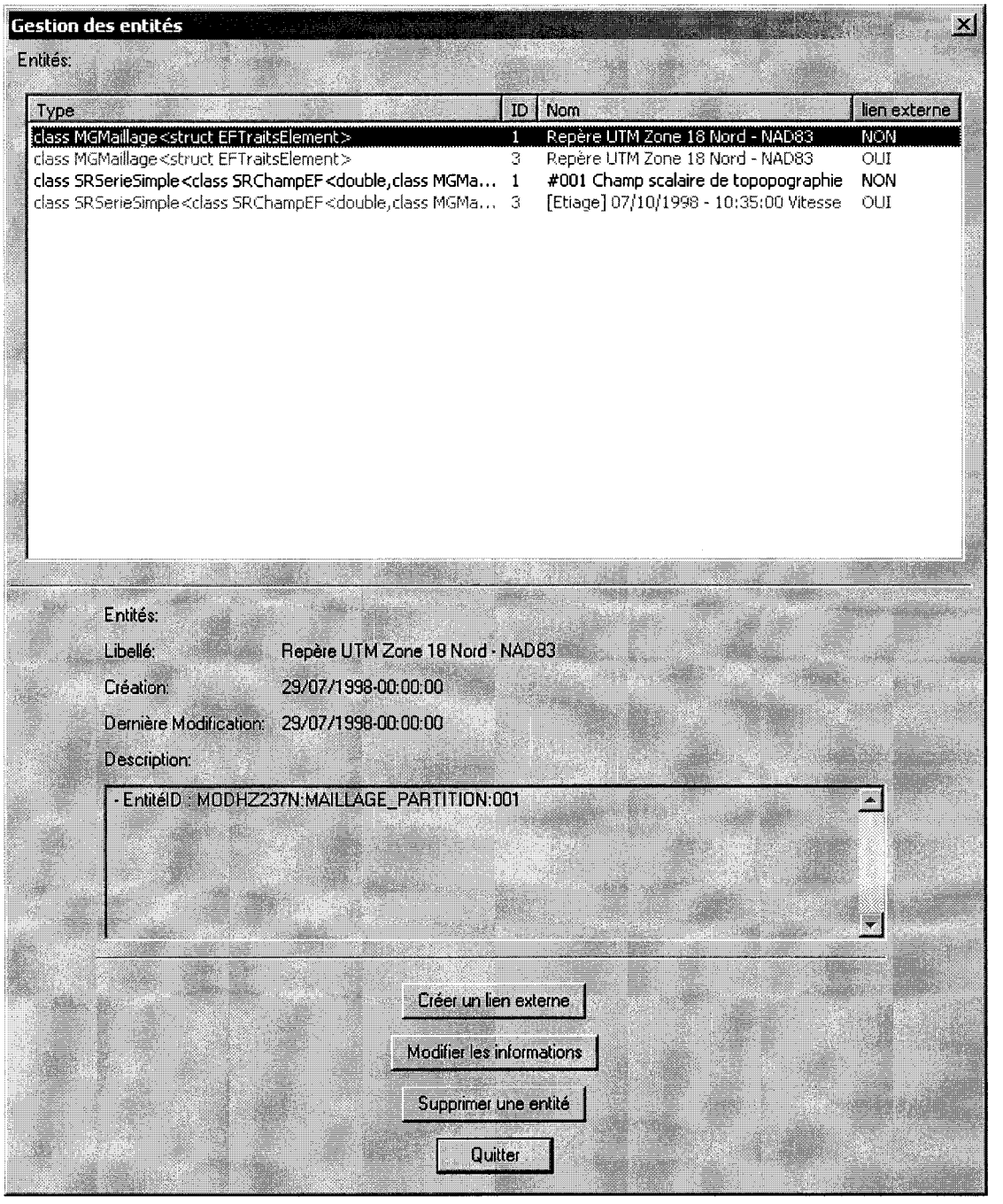
Appelée par la fonction 'traiteEvenNouveau' du Module Projet, cette boîte de dialogue permet à l'usager d'entrer les paramètres du projet qu'il veut créer. Après avoir remplis les champs et choisis la projection de base, l'usager confirme en appuyant sur le bouton 'OK'. Dans une version future, le nom de la projection sera affiché à l'usager au lieu de son identifiant.



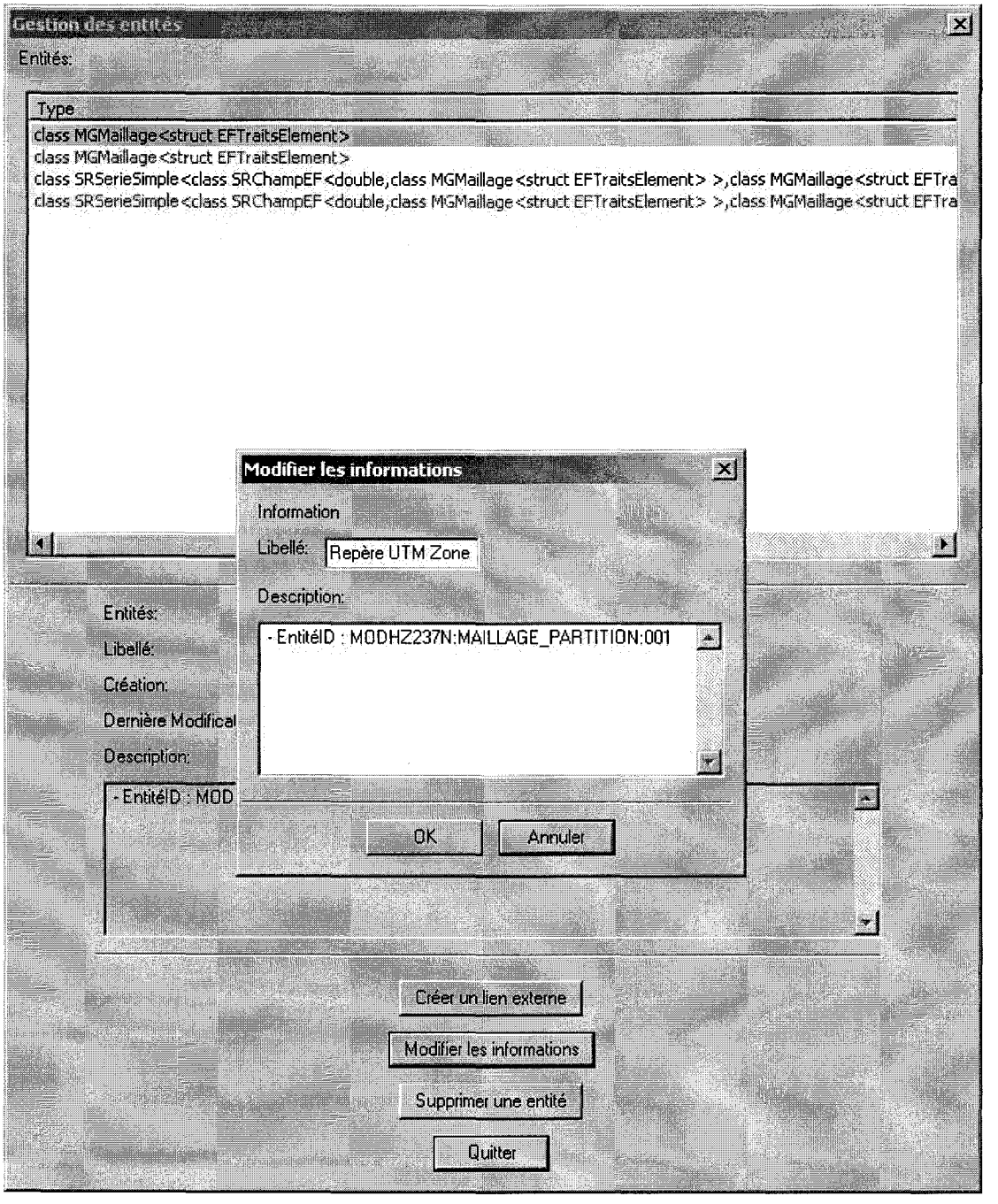
Appelée par la fonction 'traiteEvenChoisir' du Module Projet, cette boîte de dialogue permet à l'usager de choisir le projet qu'il désire ouvrir. Après avoir cliqué sur un projet, l'usager confirme son choix en appuyant sur le bouton 'OK'.



Appelée par la fonction 'traiteEvenChoisirEntite' du Module Projet, cette boîte de dialogue affiche toutes les entités du projet ou seulement celles d'un certain type. Les entités écrites en rouge sont les liens externes. Les métadonnées associées à l'entité sélectionnée sont affichées dans le bas de la boîte. Après avoir cliqué sur un projet, l'usager confirme son choix en appuyant sur le bouton 'OK'



Appelée par la fonction 'traiteEvenGestionEntite' du Module Projet, cette boîte de dialogue affiche toutes les entités du projet. Les entités écrites en rouge sont les liens externes. Les métadonnées associées à l'entité sélectionnée sont affichées dans le bas de la boîte. Après avoir cliqué sur un projet, l'usager choisit l'action qu'il désire effectuer en appuyant sur le bouton approprié.



Appelée par la fenêtre 'Gestion des entités', cette boîte de dialogue affiche les métadonnées modifiables par l'utilisateur. Après avoir effectué ses modifications, l'usager confirme en appuyant sur 'OK'

RAPPORT # 8 : Algorithme de maillage

Rapport de développement logiciel

Algorithme de maillage

Présenté par :

Sébastien Labbé

18 janvier 2005

Versions du document

Date	Auteur	Commentaires	Version
29-12-04	Sébastien Labbé	Version initiale	V0
18-01-05	Yves Secretan	Révision mineure	

Table des matières

1. Introduction

1.1 Objectifs

L'algorithme de maillage sert à créer un maillage de plusieurs couches superposées tel que décrit dans les spécifications fonctionnelles de Modeleur2. Le maillage obtenu doit être continu sur les frontières visibles de ses couches. De plus, le maillage d'une ligne frontière doit être fait selon les paramètres de la couche de priorité la plus élevée.

L'objectif de ce rapport est de présenter des améliorations aux algorithmes déjà existants.

1.2 Contexte

Un premier algorithme faisant ce travail a été développé. Or, dû à des problèmes de précision des géométries, il ne réussit pas à remplir la condition de continuité des maillages. Une modification lui permet d'éviter ces problèmes de précision. Par contre, cette modification rend l'algorithme logiquement incorrect. En effet, il ajoute des points à des endroits non désirés et ne répond donc pas à la condition de continuité. De plus, dans les deux cas, on utilise un algorithme géométrique de détermination du propriétaire de chaque arête. Or, il s'avère encore que les problèmes de précision empêchent de trouver correctement le propriétaire d'une arête dans plus de la moitié des cas.

Ces algorithmes sont expliqués dans la dernière section du

> *Rapport de développement logiciel – Module partition de maillage*, Maude Giasson, 20 septembre 2004.

Avant de poursuivre la lecture, il est recommandé de connaître ce rapport ou d'être familier avec le problème et le vocabulaire utilisé.

Ici, on propose un algorithme logiquement correct qui évite les problèmes de précision.

2. L'algorithme

Considérons l'exemple ci-contre. La couche 0 est de priorité minimum (début de la liste). La couche 5 est de priorité maximum (fin de la liste). Supposons que l'on désire mailler la couche 3. Les points d'intersection D, E et G doivent être des nœuds de ce maillage afin de satisfaire la condition de continuité. Toutefois, on doit éviter d'ajouter le point F, car la couche 1 cache ce point de croisement.

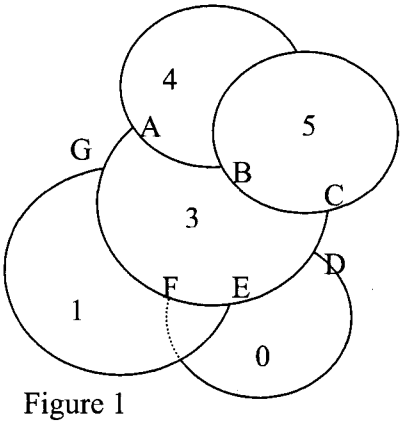


Figure 1

De plus, en parcourant les sommets de la région visible de la couche 3, on voudrait que chaque sommet connaisse la couche propriétaire de la prochaine arête. C'est-à-dire que dans le sens horaire, on aurait :

Sommets	A	B	C	D	E	F	G
Propriétaires	4	5	3	3	3	ND	3

Puisque le point F ne doit pas faire partie des sommets de la partie visible de la couche 3, sa couche propriétaire est non déterminée. On aurait pu parcourir la région dans le sens anti-horaire pour obtenir des résultats tout aussi valables.

Une fois que les régions visibles d'une couche ont été extraites, la première étape est de mailler le contour de celle-ci selon les paramètres des couches propriétaires des arêtes. L'idée est que chaque sommet connaisse sa couche propriétaire. Il suffirait ainsi de demander l'information au sommet plutôt que de la déterminer à l'aide d'un algorithme géométrique.

Il suffit donc de construire un algorithme qui ajoute des points de continuité aux bons endroits et qui assigne à chaque sommet, dès leur création, les informations sur les couches adjacentes.

L'idée générale de l'algorithme proposé est la suivante :

Soit un polygone simple et fermé d'une couche de priorité p.

1. Ajouter les points de croisement visibles avec les couches de priorité inférieure (voir section 3).
2. Soustraire les régions cachées par des couches de priorité supérieure en assignant à chaque nouveau sommet créé la couche propriétaire (voir section 4).
3. Mailler chacun des polygones possiblement disjoints obtenus (voir section 5).

De plus, on trouvera par la suite des sections sur la vérification de l'algorithme et sur le traitement des trous.

3. Interaction avec les couches inférieures

Ici, on ignore les couches de priorité supérieure à la couche à mailler. L'objectif est d'ajouter tous les points de croisement visibles avec les couches de priorité inférieure.

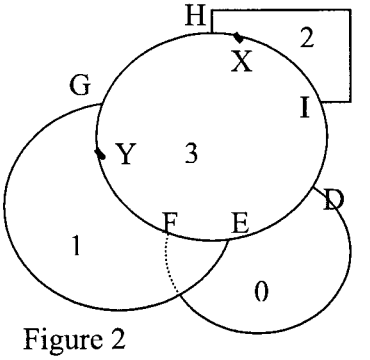


Figure 2

3.1 Fonction

La figure 2 a été obtenue à partir de la figure 1 en ignorant les couches 4 et 5. La couche 2 était précédemment cachée. On devra donc ajouter les points D, E, G, H et I. Les points X et Y représentent des sommets appartenant déjà à la couche 3.

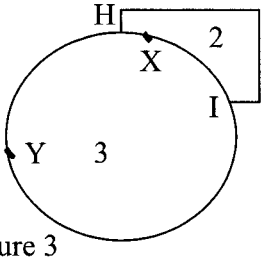


Figure 3

On traite les couches par ordre décroissant de priorité. L'idée est de se souvenir des parties cachées du périmètre sous lesquelles les points d'intersection avec des couches de priorité plus faibles ne sont pas visibles.

Sur la figure 3, on ajoute à la couche 3 les points d'intersection avec la couche 2. Il y a deux points de croisements : H et I. On réalise que la partie HXI est particulière. En effet, la couche 2 cache les éventuels points d'intersection avec la partie HXI.

Sur la figure 4, on ajoute les points d'intersection avec la couche 1. Les nouveaux points G et E ne sont pas sous la section cachée HXI. On se retrouve maintenant avec une nouvelle section cachée EYG.

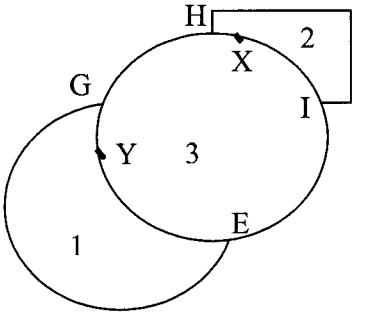


Figure 4

Voyons maintenant ce qui arrivera lors de la dernière étape celle où on ajoutera les points d'intersection avec la couche 0. D'abord, on ajoute comme précédemment les points d'intersection avec la couche 0 (voir la figure 5). C'est ensuite que l'on vérifie si ceux-ci, D et F, se situent sous des sections cachées. On réalise que c'est le cas pour le point F que l'on supprime. On se retrouve avec deux régions cachées : HXI et DEYG.

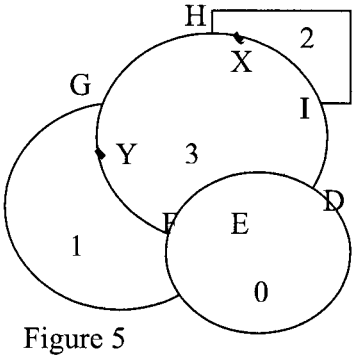


Figure 5

3.2 Structure

Le moyen utilisé pour maintenir l'information sur les parties visibles et non visibles est d'utiliser un *map* de la librairie standard. On utilise les indices de sommets comme clés et les entiers non signés qui représente la priorité de la couche adjacente pour leurs valeurs. Il aurait été suffisant pour notre algorithme d'utiliser un booléen (VISIBLE, INVISIBLE) comme valeur. Or, on a opté pour l'entier de la couche adjacente qui nous donne plus d'informations.

Ainsi, supposons que l'on travaille sur la couche de priorité p. On commence par initialiser les valeurs de tous les sommets à p. Par la suite, si pendant le traitement, la valeur en un sommet est strictement inférieure à p, l'arête qui suit est cachée. Alors que si la valeur en un sommet est toujours égale à p, l'arête qui suit est visible.

On commence à percevoir l'algorithme. En commençant par la couche de priorité p-1,

- on ajoute les points de croisements avec la couche p ;
- on parcourt les polygones obtenus en supprimant les points d'intersection cachés et en ajoutant au *map* d'adjacence les autres points d'intersection.

Le lecteur attentif aura remarqué qu'il n'y a toujours pas de moyen de reconnaître les points d'intersection. Pour ce faire, nous utilisons une hypothèse de nos fonctions géométriques. Une d'entre elles, celle qui ajoute des points de croisement à un polygone, garantit que les indices de chacun des points ajoutés sera strictement supérieur au maximum des indices des sommets qui sont déjà sur le polygone. Ce moyen ne dépend aucunement de GÉOS ou de n'importe quelle géométrie utilisée. De plus, il ne provoque aucun problème de précision.

Ainsi, avant d'ajouter des points d'intersection, on note l'indice maximum. Cela nous permet par la suite de dissocier les points d'intersection.

On a encore besoin d'une autre information. En effet, sur la figure 3, il nous faut un moyen de savoir si c'est bien la section HXI qui est cachée et non la section IYH. C'est pourquoi on commence à parcourir le polygone à un sommet qui n'est pas un point d'intersection. Ensuite, on demande à GÉOS si ce

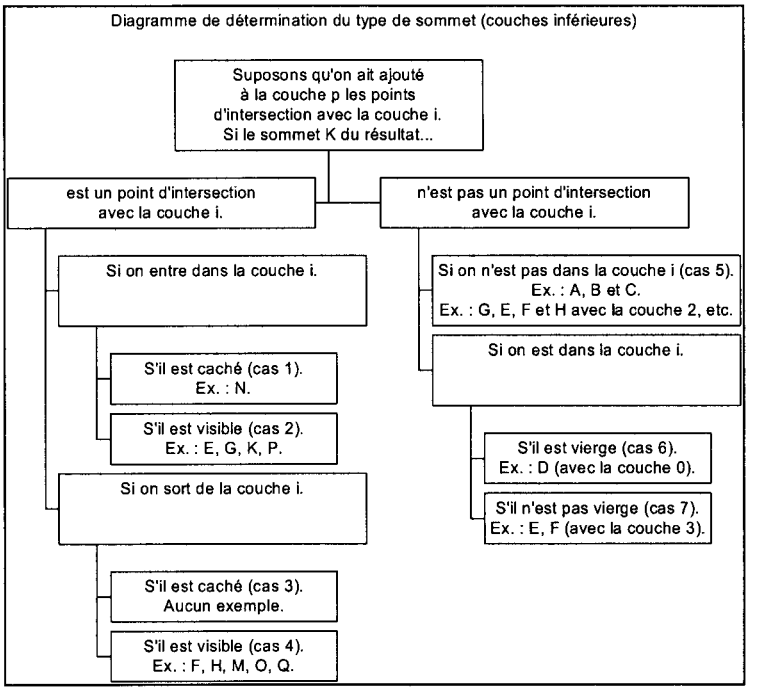


Diagramme 1

sommet initial est à l'intérieur de la couche adjacente. Cette information nous permet de nous situer par rapport aux points d'intersection.

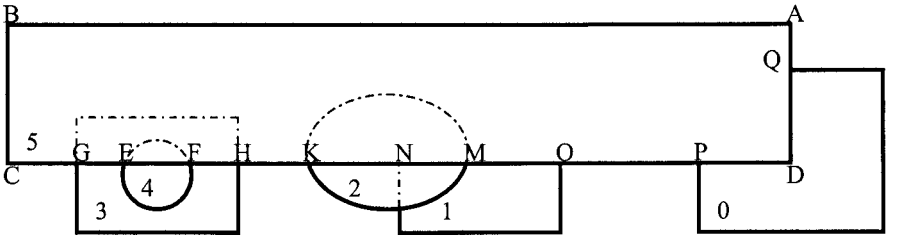


Figure 6

Parmi tous les types de sommet, on différencie sept cas qui se présentent. Ceux-ci sont présentés dans le diagramme 1, à la page précédente. Les exemples réfèrent à la figure 6 ci-dessus. Les principes suivants sont respectés :

- On détermine si le premier sommet est à l'intérieur de la couche au début du parcours. Par la suite, on alterne cette valeur à chaque fois que l'on croise un point d'intersection.
- On détermine si un point est caché en connaissant la valeur de la couche adjacente du sommet précédent. En effet, si ce nombre est inférieur à la priorité de la couche traitée (p), alors ce point ajouté est caché. On doit le supprimer du polygone.
- Lorsqu'un sommet n'est pas un point d'intersection, on ne fait que mettre à jour la valeur de la couche adjacente précédente sauf...
- Lorsque ce point est vierge (c'est-à-dire que son adjacence indique encore p) et que ce point est à l'intérieur de la couche i. On doit mettre à jour la valeur de la couche adjacente de ce sommet à la priorité de la couche i. Par exemple, le sommet D lors du traitement de la couche 5 avec la couche de priorité 0.

4. Interaction avec les couches supérieures

Cette section de l'algorithme remplit deux objectifs. Le premier est de soustraire les régions d'une couche qui sont cachées par des couches de priorités supérieures, car on maillera seulement ce qui est visible. Le deuxième est d'assigner la couche propriétaire à chaque arête créée. Cette information sera utilisée plus tard lors du maillage des régions.

4.1 Fonction

Il serait possible de faire l'union de toutes les couches supérieures et par la suite soustraire ce résultat à la couche à traiter. Or, il devient compliqué de déterminer le propriétaire de chacune des arêtes. C'est pourquoi on procède une couche à la fois. (Notons que l'ordre des couches supérieures n'est pas important.) De cette façon, on peut plus facilement déterminer le propriétaire des arêtes. En effet, il n'y a que deux possibilités. Supposons qu'on soustrait la couche i à la couche p. Soit que l'arête existait avant la soustraction et que son propriétaire ne change pas, soit que l'arête a été créée par

la soustraction et que son propriétaire est forcément la couche i. Ainsi, on obtiendra de façon itérative les informations en chaque sommet.

Poursuivons l'exemple utilisé dans les deux sections précédentes. Nous avions obtenu la couche 3 décrite à la figure 5 par les sommets X, I, D, E, Y, G, H, X. Initialement, chacun de ces sommets appartient à la couche 3. La première étape est d'effectuer la soustraction d'une couche de priorité supérieure. Commençons par la couche 4. On obtient donc une nouvelle couche décrite à la figure 7. On a que seule la section AJ appartient à la couche 4.

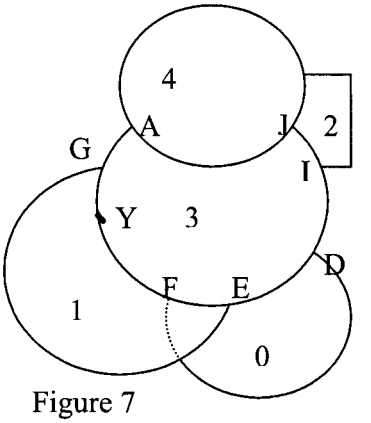


Figure 7

Ensuite, on refait les même étapes pour la couche de priorité 5. On peut observer le résultat à la figure 1 et on obtient les informations suivantes :

Sommets	A	B	C	D	E	Y	G
Propriétaires	4	5	3	3	3	3	3

En sortie, on obtient une couche à laquelle on a enlevé toutes les parties non visibles. De plus chacun des sommets connaît sa couche propriétaire. Elle est prête à être maillée.

4.2 Structure

Le moyen utilisé pour maintenir l'information sur les couches propriétaires est encore d'utiliser un *map* de la librairie standard. On utilise les indices de sommets comme clés et les entiers non signés qui représente la priorité de la couche propriétaire pour leurs valeurs. Au départ, on initialise les valeurs des couches propriétaires à p.

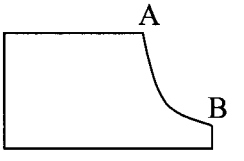


Figure 8

À l'image de la section précédente, on doit être en mesure de reconnaître certains types de points. Par exemple, à la figure 8, on a une couche 0 rectangulaire cachée par une couche 1 circulaire. Afin de pouvoir affirmer que la section courbée AB obtenue appartient à la couche 1, il faut être capable de reconnaître les points A et B. Or, ceux-ci sont exactement les points d'intersection entre les deux couches. Pour déterminer ces points, on ne peut utiliser la même astuce qu'à la section précédente, car la soustraction utilise une fonction de GÉOS. Or, cette fonction nous donne le résultat dans un troisième polygone ayant des indices de sommets indépendants. C'est pourquoi il faut reconstruire le *map* d'appartenance à chaque fois que l'on fait une action de soustraction.

Bref, pour reconnaître les points d'intersection, on procède de la façon suivante. Supposons toujours que l'on a la couche p à laquelle on soustrait la couche i.

- On crée *copieCi*, une copie du polygone de la couche i. On ajoute à *copieCi* les points de croisement avec la couche p.
- On crée *partieVisible*, une copie du polygone actuel *Cp* de la couche p. Au polygone actuel, on ajoute les points de croisement avec la couche i. L'objet *partieVisible* servira à effectuer la soustraction avec la couche i.
- Au polygone *partieVisible*, on soustrait le polygone de la couche i.

Après ces étapes, le résultat est contenu dans *partieVisible*. Toutefois, on se servira des deux autres copies (qui représentent l'ajout des points de croisement de l'un à l'autre) pour déterminer les points d'intersection et ensuite les couches propriétaires.

Le traitement des différents types de sommets rencontrés est représenté dans le diagramme 2, ci-contre.

Actuellement, le quatrième cas n'est pas codé. Jusqu'à maintenant, il n'est jamais arrivé qu'un sommet n'est pas été retrouvé dans les deux autres polygones. Cela étant possible, il faudrait compléter la partie manquante.

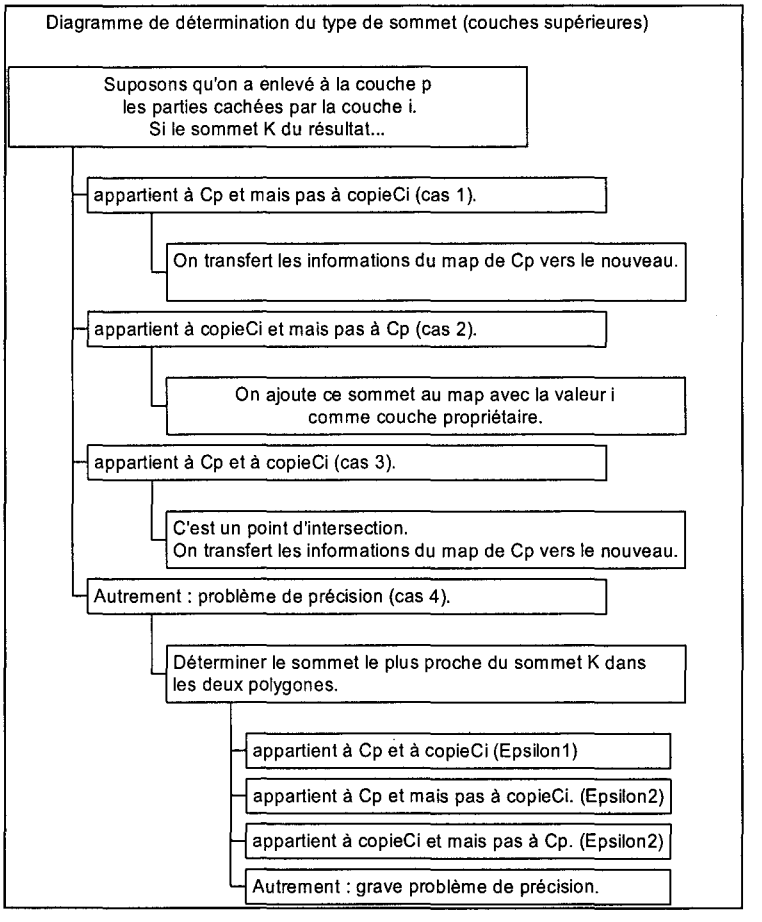


Diagramme 2

La fonction qui ajoute les points de croisements aux deux polygones est presque identique à celle qui est utilisée pour les couches inférieures. En effet, on doit aussi maintenir un *map* sur chacun des sommets. Par contre, on n'a pas à supprimer des points, car les parties non visibles ne s'appliquent pas dans ce cas.

5. Maillage d'un polygone

Cette partie de l'algorithme n'a pas été modifiée, à différence que maintenant le propriétaire de chaque arête n'est pas trouvé. En effet, cette information est maintenue par le sommet. Il est possible d'obtenir la documentation sur cette section de l'algorithme en utilisant la référence présentée en introduction.

6. Hypothèses exigées de la géométrie

L'idée de cet algorithme est de réduire au minimum la dépendance des calculs géométriques. Cette dépendance n'a pas été complètement effacée. Voici les hypothèses toujours exigées de la géométrie utilisée.

Traitement avec les couches inférieures :

- Chaque sommet doit posséder un indice unique.
- Lorsqu'on ajoute des points de croisement sur un polygone, les nouveaux points doivent posséder un indice supérieur à l'indice maximum alloué pour les sommets du polygone initial. C'est ainsi qu'on distingue les points d'intersection.

Traitement avec les couches supérieures :

- Si A et B sont deux polygones dont l'intersection est non vide. Alors, les sommets de la différence entre A et B appartiennent à A, à B ou sont les points d'intersection entre A et B. Bref, les fonctions d'ajout des points de croisement et de différence sont cohérentes entre elles.

Bien sûr, il reste des problèmes d'imprécision. Par exemple, déterminer si un point est à l'intérieur d'un polygone ou non, lorsque ce point est situé sur le périmètre du polygone. Ce problème n'est pas provoqué par l'algorithme mais plutôt indirectement par l'utilisateur par le positionnement des polygones. C'est pourquoi des mesures de robustesse ont été ajoutées un peu partout dans l'algorithme.

7. Robustesse

7.1 Cas de tangence

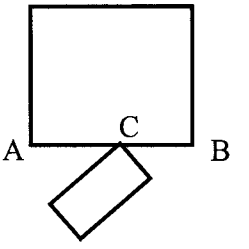


Figure 9

L'algorithme a été conçu initialement sous l'hypothèse que les cas de tangence sont inexistants (voir figure 9). En effet, l'idée utilisée lors du parcours d'un polygone est de demander qu'une seule fois au départ si on se trouve à l'intérieur d'un polygone. Par la suite, en connaissant le nombre de points de croisements rencontrés, on peut savoir si on se trouve à l'intérieur ou à l'extérieur. C'est ce principe qui est illustré à la figure 10. Le chiffres représentent le nombre de point de croisement rencontrés. On remarque que (s'il n'y a pas de point de tangence) les nombres impairs sont à l'intérieur tandis que les nombres pairs sont à l'extérieur.

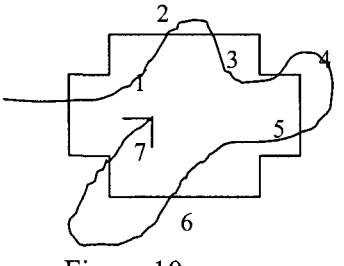


Figure 10

C'est pourquoi lors du parcours d'un polygone, on a ajouté des moyens de reconnaître ce type de cas. Par exemple, à chaque fois qu'on rencontre un point d'intersection, on s'assure que le point milieu de l'arête qui suit se trouve à l'intérieur du polygone si on se

trouvait auparavant à l'extérieur ou inversement. À la figure 9, on rencontre le point d'intersection C dans le sens anti-horaire. Avant de continuer, on s'assure que le point milieu du segment CB appartient à l'intérieur du polygone (le rectangle oblique), ce qui n'est pas le cas. On vient de détecter un problème. Jusqu'à présent, cette astuce a plus souvent servi à reconnaître le cas où on tente de mailler deux polygones identiques superposés.

7.2 Cas à problème

Que ce soit un message d'erreur ou un cas de tangence détecté, on tente du mieux possible d'ignorer les couches causant problèmes et continuer le traitement avec les autres couches. Ensuite, on transmet les informations concernant les problèmes à l'utilisateur. Il serait possible d'améliorer la performance de l'algorithme en ajoutant à la géométrie la possibilité de copier intégralement (indices identiques) un multipolygone. De cette manière, il serait toujours possible de revenir en arrière quelque soit l'étape effectuée.

8. Prochaines améliorations

8.1 Traitement des cas d'imprécision

Lors du traitement des couches supérieures, nous avons vu que nous parcourons les sommets d'un multipolygone à plusieurs reprises. Chacun des sommets est recherché dans deux autres multipolygones (voir le diagramme 2). En théorie, trois cas sont possibles. Toutefois, en pratique, dû à des problèmes d'imprécision, il est possible qu'un sommet ne soit pas trouvé dans aucun des deux multipolygones et cette partie n'a pas été codée. Dans ce cas, il suffit de rechercher dans les deux multipolygones le sommet qui minimise la distance au point en question. Ensuite, on définit une distance epsilon sous laquelle deux points sont considérés égaux. Selon les deux distances obtenues, on distingue trois cas :

- Les deux distances sont inférieures à epsilon ;
- Seule la première distance est inférieure à epsilon ;
- Seule la deuxième distance est inférieure à epsilon.

Autrement, c'est-à-dire si les distances sont supérieures à epsilon, il faudra se poser de sérieuses questions quant au epsilon choisi ou à la géométrie utilisée. On traite ces cas de façon identique à ceux qui ont été décrits à la section 4.

De plus, il serait intéressant d'ajouter au rapport de maillage les approximations effectuées sur les points, car ce cas semble se produire très rarement.

La majorité du travail à effectuer pour cette amélioration se trouve dans les géométries. Notamment, il faudrait ajouter la possibilité de trouver le point d'un multipolygone qui minimise la distance à un point donné.

8.2 Traitement des trous

Lors de l'extraction des parties visibles d'une couche, il arrive qu'une autre couche de priorité supérieure soit entièrement contenue dans la première. Cette situation provoque un trou dans la région visible de la première couche. Or, le maillage de polygone troués nécessite un traitement différent qui n'est actuellement pas fait. Plusieurs solutions sont possibles à ce problème. Initialement, on parlait de traiter ce cas directement dans le fichier PMAlgoMaille.cpp, alors qu'il serait mieux de le faire traiter ailleurs.

Premièrement, il faut savoir si le mailleur utilisé est capable de mailler des polygones avec des trous. Dans le cas positifs, il y aurait quelques modifications à apporter au fichier PMAlgoMaille.cpp :

- Dans le traitement des couches supérieures, il faudrait ajouter les sommets des trous au *map* d'appartenance de la même façon qu'on le fait pour les sommets actuellement. S'assurer que le traitement est vraiment le même.
- Dans l'opérateur(), avant d'envoyer la zone au mailleur, il faudrait lui ajouter les sommets des trous selon la manière que le mailleur les demande.

Dans le cas où le mailleur ne permet pas de mailler les trous, il faut développer une autre approche où tout serait effectué dans la partie géométrique. Présentement, la fonction de soustraction de deux polygones *retourne* une liste de polylignes en cas de trou, le premier élément de la liste étant le contour du polygone et où les suivants sont des trous. Or, il pourrait y avoir une autre fonction qui appelle celle-ci et qui effectue un traitement en cas de trou. Ainsi, si on veut soustraire B de A,

NotreFonction(A, B)

- Demander à la fonction actuelle de différence de soustraire B de A.
- Si le résultat A ne possède pas de trou,
  - On *retourne* A.
- Modifier A, une liste de polylignes, en une liste de polygones en joignant les trous au contour.
- *Retourner* A.

Comme un exemple visuel est toujours plus explicite, observons la figure 11. Un polygone initial (une liste formée d'un rectangle, d'un cercle et d'un triangle) peut être transformé en deux polygones sans trou. Il faut s'assurer qu'au moins deux coupes se rendent à chaque trou. Autrement, il existerait un polygone possédant un même sommet deux fois, ce qu'il est préférable d'éviter. Ainsi, on ajoute des lignes tant que chaque trou n'a pas été relié au moins deux fois en respectant les principes suivants :

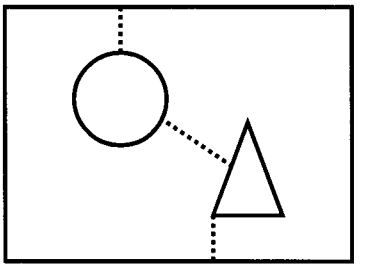


Figure 11

- Une ligne ne doit être ajoutée que dans la partie intérieure du polygone.
- Une ligne est possible si elle relie deux points qui se voient l'un et l'autre, c'est-à-dire que cette ligne ne croise pas le contour et les trous.
- Une ligne plus courte est choisie avantageusement à une autre plus longue.

L'inconvénient de cette technique, est que les lignes ajoutés seront immuables dans le maillage.

8.3 Vérification de l'appartenance

Au cours de l'utilisation de cet algorithme dans le cadre du logiciel Modeleur2, il pourrait s'avérer que, pour certaines arêtes, les propriétaires soient inexacts. Par exemple, cette manifestation pourrait être observée par la discontinuité du maillage en ces arêtes. Dans ce cas, il faudrait d'abord ajouter un algorithme géométrique de vérification de l'appartenance des arêtes. On pourrait ajouter ce résultat dans le rapport de maillage, qui s'affiche dans une boîte de dialogue à la fin du maillage, sous la forme suivante : « Parmi les 345 arêtes obtenues, 342 ont passé le test de vérification de l'appartenance des arêtes. » Voici un test proposé.

Après le traitement des couches supérieures, on obtient un multipolygone.
Pour chacun de ses sommets :

- Obtenir le propriétaire selon le *map* d'appartenance de ce sommet ;
- Obtenir le point milieu de l'arête en question ;
- Vérifier que ce point milieu est situé sur le périmètre de la couche propriétaire.

8.4 Robustesse

Des mesures de robustesse ont été ajoutées pour reconnaître des situations problématiques. Toutefois, ces tests sont des conditions suffisantes mais non nécessaires pour reconnaître un problème. Par conséquent, il serait possible d'améliorer cette robustesse (voir la section 7).

9. Conclusion

9.1 Résumé

Bref, nous avons exploré tous les aspects cruciaux importants de l'algorithme de maillage. Nous avons vu comment l'algorithme a été séparé en différentes parties ayant chacune leurs fonctions. D'abord, on a étudié la façon que l'on traite les couches de priorité inférieure. Il y a un moyen d'ajouter seulement les points de croisement visibles avec celles-ci. Ensuite, nous avons vu comment le traitement des couches supérieures s'effectue. Encore là, nous procédons de façon itérative en traitant une couche à la fois et où le résultat intermédiaire est en tout temps cohérent. Après, nous avons présenté les diverses hypothèses que nous demandons de la géométrie. Finalement, nous avons montré les mesures de robustesse qui ont été ajoutées à l'algorithme.

9.2 Perspectives futures

Quatre améliorations ont été décrites à la section 8. Rappelons-les :

- Compléter le code dans le traitement des couches supérieures (le cas d'imprécision).
- Ajouter la capacité de mailler les couches comportant des trous.
- Ajouter une vérification du propriétaire des arêtes.
- Améliorer la robustesse.

RAPPORT # 9 : Validation du modèle de température sous H2D2

Rapport final

Validation du modèle de température sous H2D2

Présenté par :

Daniel Nadeau

19 janvier 2005

REMERCIEMENTS

Je tiens à remercier très sincèrement les personnes suivantes, puisqu'elles toutes collaborées, de près ou de loin, à faire progresser le projet :

Yves, *pour sa rigueur scientifique, ses précieux conseils et sa confiance.*

Paul, *pour sa grande expertise, sa rigueur, son sens critique et son aide;*

Philippe, *pour m'avoir fait connaître la réalité physique derrière la simulation numérique*;

Jean et Olivier; *pour leur aide et leur disponibilité.*

RÉSUMÉ

Le logiciel de calcul H2D2 permet l'étude de la température de l'eau par le biais de simulations hydrauliques sur des zones d'intérêt documentées. Afin de valider la modélisation 2D de la température de l'eau avec H2D2, deux études ont été réalisées (en 2001 et 2003) sur le fleuve Saint-Laurent. Les résultats de ces études montrent bien la capacité des modèles utilisés à reproduire la gamme des températures observées. Or, par moments, les températures simulées par H2D2 ne concordaient pas bien avec celles mesurées. Il a donc été décidé, afin d'améliorer la qualité des résultats produits par le logiciel de calcul, de réaliser une étude de validation centrée sur la contribution du bilan thermique à l'établissement de la température de l'eau.

Afin de pouvoir valider le modèle de température utilisé par le logiciel de calcul H2D2, nous avons procédé à plusieurs simulations sur un maillage éléments finis adapté au segment 4 du marais aménagé de Saint-Barthélemy. Ce dernier étant bien documenté au plan des données environnementales, il a par la suite été possible de mettre en parallèle les températures simulées et mesurées par les thermistors présent sur le site lors de la période allant du 10 avril 2002 au 6 mai 2002 afin de pouvoir étudier l'impact de diverses combinaisons de paramètres dans le cadre du processus de résolution.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

ÉQUIPE DE RECHERCHE

Planification, conception et rédaction :

Daniel Nadeau[1]
Yves Secretan[1]
Paul Boudreau[1]

Production des données topométriques :

Sylvain Martin[2]
Olivier Champoux[2]

Production et assemblage des données météorologiques :

Jacques-Yves Lapierre[3]
Annie Duhamel[3]

Mesures de températures, de niveaux de surface, de turbidité de l'eau et autres renseignements relatifs au segment 4 du marais de Saint-Barthélemy :

Philippe Brodeur[4]
Marc Mingelbier[4]
Jean Morin[2]
Olivier Champoux[2]

---

[1] Institut National de la Recherche Scientifique – Eau, Terre et Environnement (INRS-ETE)

[2] Hydrologie, Service Météorologique du Canada, Environnement Canada

[3] Québec Climat, Services Climatologiques, Suivi et Adaptation en Climat, Service Météorologique du Canada, Environnement Canada

[4] Écologie des Systèmes Aquatiques, Société de la Faune et des Parcs du Québec, Direction de la Recherche sur la Faune

INTRODUCTION

Le Groupe de Recherche et d'Étude en Éco-Hydraulique Numérique (GREEN) se spécialise dans le développement d'un logiciel de simulation hydraulique par éléments finis. Le logiciel développé s'appelle Modeleur et il possède deux modules externes, nommés Hydrosim et Dispersim. Ils permettent respectivement d'effectuer des simulations hydrodynamiques et des simulations de transport-diffusion (qui comprend notamment l'étude de la température de l'eau). De son côté, Modeleur est un logiciel qui permet de construire des maillages, de même que de visualiser et effectuer des opérations sur des champs scalaires et vectoriels. Modeleur existe depuis quelques années et est fonctionnel. Or, il y a un peu plus d'un an, le GREEN met en branle le développement d'une deuxième version. Parallèlement, le projet H2D2 (logiciel de calcul), visant à réunir les deux modules externes en un seul pour les inclure dans Modeleur 2, est lancé.

Afin de valider la modélisation 2D de la température de l'eau avec H2D2, deux études sont menées (en 2001 et 2003) sur le fleuve Saint-Laurent, dans la région des Îles de Boucherville. Les résultats de ces études montrent bien la capacité des modèles utilisés à reproduire la gamme des températures observées. Or, par moments, les températures simulées par H2D2 ne concordent pas bien avec celles mesurées, plus particulièrement au niveau de l'amplitude. Il est donc décidé, afin d'améliorer la qualité des résultats produits par le logiciel de calcul, de réaliser une étude de validation concentrée uniquement sur la contribution du bilan thermique aux fluctuations de la température de l'eau.

Pour ce faire, il faut en premier lieu trouver un site bien documenté (topographie, données météorologiques, niveaux de surface, températures de l'eau ...) pourvu d'une activité hydrodynamique nulle. À cet effet, le segment 4 du marais aménagé de Saint-Barthélemy est choisi. En effet, dans le cadre d'une étude sur les poissons venant y frayer en 2002, le segment 4 reçoit une attention particulière de la part des chercheurs. En outre, par le biais de trois thermistors, des mesures de température de l'eau sont notamment effectuées sur une base horaire.

L'objectif de la présente étude est donc de tester la validité du modèle de température de l'eau utilisé au sein du processus de calcul. Il s'agit par exemple de voir si certains phénomènes météorologiques peuvent être la cause du mauvais fonctionnement des simulations via H2D2. L'effet de la méthode numérique de résolution se doit aussi d'être testé. Dans cette voie, plusieurs simulations éléments finis (en variant les combinaisons de paramètres d'entrée utilisés) sont lancées afin de pouvoir comparer les valeurs simulées à celles obtenues par les thermistors situés à l'intérieur du marais.

Le présent rapport détaille d'abord le modèle déterministe de température en traitant les divers phénomènes d'échange de chaleur qui y sont impliqués (chapitre 1). Ensuite, ces phénomènes sont présentés sous forme mathématique (chapitre 2). Une description du site de simulation, soit le marais de Saint-Barthélemy est ensuite réalisée (chapitre 3). Puis, les informations relatives à la méthodologie utilisée sont présentées (chapitre 4), suivi des détails sur les données environnementales (chapitre 5). Finalement, une étude des résultats des simulations de température est accomplie (chapitre 6).

1 MODÈLE DE TEMPÉRATURE

Dans les cours d'eau naturels, la température de l'eau T est principalement contrôlée par les paramètres climatologiques comme la radiation solaire, la température de l'air, la pression atmosphérique, l'humidité, les précipitations et la vitesse du vent (Heniche *et al.* 2002). Les équations aux dérivées partielles de transport–diffusion forment la base du modèle de transport de température. On leur ajoute des capacités de simulation de la température de l'eau en établissant un bilan thermique sur la colonne d'eau. Ceci consiste à établir la liste des sources et les puits de chaleur, à les modéliser et à les introduire dans les équations (Morin *et al.* 2002).

1.1 Bilan thermique

Ainsi, le flux thermique total S qui contribue à réchauffer ou à refroidir la colonne d'eau s'exprime (Morin *et al.* 2002) :

$$S = S_a + S_b + S_i , \qquad [1.1]$$

où S_a, S_b et S_i sont respectivement les flux thermiques d'échange avec l'atmosphère, d'échange avec le fond et d'échange avec le couvert de glace.

Le flux thermique avec l'atmosphère S_a représente normalement la contribution la plus importante au budget thermique global S. Il compte lui-même des composantes de radiation solaire, de radiation infrarouge, d'évaporation, de convection et des précipitations (Heniche *et al.* 2002) :

$$S_a = (H_s - H_l - H_e - H_c - H_p) f_{extinc} , \qquad [1.2]$$

amortie par une fonction d'extinction f_{extinc}.

Dans le cadre de la présente étude, il s'avère essentiel de s'attarder au développement des équations relatives au bilan thermique atmosphérique total.

Le flux de radiation solaire est modéré par deux facteurs liés à l'environnement : la fraction de couvert végétal et l'albédo (ou réflectivité) de l'eau, soit (Heniche *et al.* 2002)

$$H_S = H_{SI}(1-SF)(1-r) . \qquad [1.3]$$

Par souci de simplicité, il s'avère avantageux de se baser sur les mesures des radiations solaires incidentes H_{SI} (en W/m²) effectuées par les stations météorologiques. Quant à lui, le facteur de couvert végétal SF varie entre 0 (aucune végétation) à 1 (obstruction végétale complète). L'albédo de l'eau r est de 0 pour de l'eau claire et peut varier entre 0.06 et 0.12 pour des eaux turbides.

Le terme H_l (en W/m²) fait référence au phénomène de radiation infrarouge qui s'opère entre les deux fluides en contact, soit l'air et l'eau. Tout dépendant de la température extérieure, il peut s'agir d'un terme puits ou d'un terme source. L'expression s'écrit donc :

$$H_l=\sigma[\varepsilon_w(273.15+T)^4-\varepsilon_a(273.15+T_a)^4], \qquad [1.4]$$

où σ, la constante de Stefan-Bolztmann, a pour valeur $5.67051\text{x}10^{-8}$ W/m²·K⁴, alors que l'émissivité de l'eau, ε_w, est de 0.97. D'autre part, plusieurs équations représentant l'émissivité atmosphérique, ε_a, existent. Nous allons procéder à la description de celles qui sont proposées à l'utilisateur de H2D2, soit les formules de Brocard, Marceau, Paily et Sinokrot (équations extraites du fichier **meteo.f**[1]). Dans tous les cas, ε_a dépend de la couverture nuageuse C_c (qui varie entre 0 et 1).

En premier lieu, la formule de Brocard s'exprime ainsi :

$$\varepsilon_a=\varepsilon_{ac}(1+0.17C_c^2), \qquad [1.5]$$

où l'émissivité pour un ciel exempt de nuages, ε_{ac}, dépend de la température de l'air (T_a, en °C) tel que le prédit la formule de Swinbank :

$$\varepsilon_{ac}=0.937x10^{-5}(273.15+T_a)^2. \qquad [1.6]$$

La formule proposée par Marceau reprend l'équation 1.5, excepté que ε_{ac} implique la pression de vapeur de l'air e_a (en mmHg) dans son calcul (voir équations 1.15 et 1.16) :

$$\varepsilon_{ac}=0.97\left(0.74+\frac{0.0065e_a}{133.322387415}\right). \qquad [1.7]$$

L'émissivité atmosphérique telle que décrite par Paily nécessite la nébulosité C_c, l'altitude des nuages Z_n (en m) et la pression de vapeur de l'air e_a (en mb). Deux résultats intermédiaires doivent d'abord être obtenus, comme le décrivent les équations 1.8 et 1.9 :

$$INT\_1=0.74+0.025(100C_c)\exp(-1.92x10^{-4}(Z_n)) \qquad [1.8]$$

et

$$INT\_2=4.9x10^{-3}-(5.4x10^{-4})(100C_c)\exp(-1.97x10^{-4}(Z_n)). \qquad [1.9]$$

Finalement, on obtient l'expression suivante :

$$\varepsilon_a=INT\_1+(INT\_2)(e_a). \qquad [1.10]$$

[1] Fichier disponible sur le CD d'accompagnement.

La dernière formule proposée, celle de Sinokrot, reprend l'équation 1.5, où l'émissivité pour un ciel exempt de nuages est représentée ainsi :

$$\varepsilon_{ac}=1-\left[0.261\left(\exp\left(-7.77x10^{-4}(T_a)^2\right)\right)\right]. \qquad [1.11]$$

Le flux de chaleur associé à l'évaporation H_e contribue à diminuer la température de la colonne d'eau. On choisit la loi de Dalton afin de modéliser ce phénomène sous forme mathématique. Elle combine l'action du vent, représentée par $f(W_z)$, de même que la différence entre la pression de vapeur saturée e_s et la pression de vapeur e_a. De façon générale, on a donc (Heniche *et al.* 2002) :

$$H_e=f(W_z)(e_s-e_a). \qquad [1.12]$$

Nombreuses sont les formules qui existent afin de représenter la fonction du vent $f(W_z)$. Cependant, un nombre limité de ces dernières sont insérées dans le logiciel de calcul H2D2 : Barnwell, Brown, Bowles, Harleman, Mohseni et Morin (équations extraites du fichier **meteo.f**[2]).

Puisque les stations météorologiques mesurent généralement la vitesse du vent à 10 m au-dessus du sol, une transformation doit être effectuée en se basant sur le phénomène des couches limites afin d'obtenir la vitesse du vent à 2 mètres au-dessus du sol W_2. Ainsi, on a

$$W_2=W_{10}\left[\frac{\ln\left(\frac{2}{z_0}\right)}{\ln\left(\frac{10}{z_0}\right)}\right], \qquad [1.13]$$

qui s'avère être une expression dérivée de la théorie des couches limites, où z_0 est défini par

$$z_0=\frac{10}{\exp\left(\frac{\kappa}{\sqrt{10^{-3}(1+0.07W_{10})}}\right)} \qquad [1.14]$$

et $\kappa = 0.4$ est la constante de Von Karman. En raison du phénomène de couches limites, la vitesse du vent à 2 m au-dessus du sol est inférieure à celle à 10 m au-dessus du sol.

D'autre part, la formule de Magnus-Tetton permet d'identifier la valeur de la pression de vapeur saturée e_s (en Pa) en fonction de la température de l'air T_a (en °C) (Heniche *et al.* 2002) :

$$e_s=610.78\exp\left(\frac{17.26939T_a}{T_a+237.29}\right). \qquad [1.15]$$

[2] Fichier disponible sur le CD d'accompagnement.

De son côté, la pression de vapeur e_a (en Pa) est reliée à e_s via l'humidité relative RH (en %), de telle sorte que :

$$e_a=(\frac{RH}{100})e_s . \qquad [1.16]$$

Nous pouvons maintenant procéder aux détails des différentes fonctions de vent retenues. La formule de Barnwell appelle la vitesse du vent à 2 mètres au-dessus du sol, soit W_2 (en m/s) :

$$f(W_2) = 0.25(1.83483+0.32483W_2). \qquad [1.17]$$

Similairement, la fonction du vent de Brown est ainsi développée:

$$f(W_2)=3.9x10^{-2}W_2 . \qquad [1.18]$$

De son côté, la formulation proposée par Harleman est beaucoup plus complexe, puisqu'elle fait intervenir davantage de variables. Cette fonction du vent fait appel aux équations 1.19 à 1.21 :

$$T_{vs}=\frac{T_a}{1-0.378\left(\frac{e_s}{P_{atm}}\right)}, \qquad [1.19]$$

$$T_{va}=\frac{T_a}{1-0.378\left(\frac{e_a}{P_{atm}}\right)} \qquad [1.20]$$

et

$$DT_{1\_3}=(T_{vs}-T_{va})^{\frac{1}{3}} . \qquad [1.21]$$

En définitive, on a :

$$f(W_2)=1x10^{-2}\left(3.2W_2+2.7DT_{1\_3}\right). \qquad [1.22]$$

Quant à elle, la formule de Mohseni considère la vitesse du vent mesurée à 10 m par rapport au niveau de surface, W_{10}.

$$f(W_{10})=\frac{0.622(2.45x10^6)(1.225)(0.4)^2W_{10}}{P_{atm}\left[\log\left(\frac{10}{z_0}\right)\right]} \qquad [1.23]$$

Il est à noter dans cette dernière expression que P_{atm} (en Pa) représente la pression atmosphérique.

La dernière fonction du vent, d'une grande simplicité, est décrite par Morin à l'équation 1.24 :

$$f(W_2)=\frac{3.795x10^3 W_{10}}{P_{atm}}. \qquad [1.24]$$

En poursuivant dans la description du budget atmosphérique total, la convection à l'interface eau-air est modélisée par la loi de Bowen, exprimée ainsi (Heniche *et al.* 2002) :

$$H_c=RH_e \qquad [1.25]$$

où R dénote le coefficient de Bowen,

$$R=6.1x10^{-4}P_{atm}\left(\frac{T-T_a}{e_s-e_a}\right). \qquad [1.26]$$

Le flux thermique lié aux précipitations doit pouvoir considérer les chutes de neige et de pluie. Une simple loi de transfert thermique peut être appliquée pour décrire l'interaction air-eau due aux précipitations de pluie (Heniche *et al.* 2002) :

$$H_p=p_r c_p(T-T_a), \qquad [1.27]$$

où p_r dénote le flux de précipitations de pluie (en kg/m²·s) et c_p la chaleur spécifique de l'eau (en J/kg·°C).

En ce qui concerne les chutes de neige, l'expression de transfert thermique tient compte de trois différents phénomènes physiques: échange de chaleur dans la phase solide jusqu'à ce que la température de fusion (T_m) soit atteinte, énergie impliquée dans le changement de phase solide-liquide de même que le flux thermique dans la phase liquide. Le terme H_p devient alors :

$$H_p=p_s(c_{pi}(T_m-T_a)+L_i+c_p(T-T_m)), \qquad [1.28]$$

où p_s, c_{pi}, L_i et T_m représentent respectivement le flux de précipitations de neige (en kg/m²·s), la chaleur spécifique de la glace (en J/kg·°C), l'énergie latente de fusion de la glace (en J) et le point de fusion de l'eau (en °C).

Une petite précision mérite d'être apportée au sujet de la neige (Sergent, 1998). Avec la réduction des structures dendritiques, la neige déposée voit ses distances inter granulaires diminuer, et ceci a pour conséquence l'augmentation de sa masse volumique. Ainsi une neige tombée sans vent à une température assez basse de l'ordre de -15°C a en moyenne une masse volumique de l'ordre de 20 à 50 kg/m³, alors qu'avec un vent de l'ordre de 10 m/s et une température de -5°C, elle peut atteindre des valeurs de 150 à 200 kg/m3. Mais on constate qu'en moyenne, au moment de la précipitation, la neige a une masse volumique de l'ordre de 100 kg/m³, ce qui a en outre l'avantage de permettre d'appliquer la correspondance : 10 cm de neige/10 mm d'eau.

Le flux thermique avec le fond peut être important, surtout dans les zones peu profondes (Morin *et al.* 2002). En pratique, la détermination de S_b est difficile. Comme la température souterraine peut être considérée constante, nous utilisons un loi de convection thermique pour représenter les échanges thermiques avec le fond :

$$S_b = -K_b(T - T_b), \qquad [1.29]$$

où K_b (en W/m^2·°C) et T_b (en °C) sont respectivement le coefficient d'échange thermique et la température souterraine.
La prise en compte du flux thermique d'échange avec le couvert de glace est obligatoire dans les régions froides. Une estimation du flux thermique est donnée par la relation suivant, en faisant l'hypothèse que la concentration de frasil est négligeable (Heniche *et al.* 2002) :

$$S_i = -K_i(T - T_m), \qquad [1.30]$$

où K_i (W/m^2·°C) est le coefficient d'échange thermique associé à la glace.

1.2 Linéarité du bilan thermique

En regroupant les expressions relatives aux différents termes puits et sources, la balance thermique globale peut être écrite (Heniche *et al.* 2002) sous la forme

$$S = a(t)T + b(t), \qquad [1.31]$$

où $a(t)$ et $b(t)$ sont des coefficients transitoires dépendant des conditions météorologiques.

Afin d'obtenir un modèle linéaire en T, trois hypothèses doivent être posées :

1) Les propriétés physiques de l'eau (tels ρ et c_p) sont constantes en tout temps
2) La fonction du vent $f(W_2)$ est indépendante de la température de l'eau (T)
3) L'expression T^4 dans le terme de flux thermique de radiation infrarouge (H_l) est approximée par une courbe de tendance linéaire telle que

$$\sigma\varepsilon_w(273.15+T)^4 \approx \sigma\varepsilon_w(A_l T + B_l), \qquad [1.32]$$

où dans notre cas, $A_l = 1.06545\text{x}10^8$ K^3 et $B_l = 5.37180\text{x}10^9$ K^4, avec un coefficient de corrélation de 0.9951.

Attardons-nous maintenant à la description de chacun des termes participant aux coefficients $a(t)$ et $b(t)$. En se basant sur ce qui a été développé dans le budget thermique total, nous sommes à même de constater que

$$a(t)_{partiel} = -(\sigma\varepsilon_w A_l f_{extinc}) - (6.1x10^{-4} P_{atm} f(W_2) f_{extinc}) - K_i - K_b . \qquad [1.33]$$

La présence de précipitations vient directement influencer la valeur de *a(t)*. C'est pourquoi, en cas de pluie on a

$$a(t)=a(t)_{partiel}-c_p p_r f_{extinc}, \qquad [1.34]$$

alors qu'en cas de neige, on retrouve

$$a(t)=a(t)_{partiel}-c_p p_s f_{extinc}. \qquad [1.35]$$

En ce qui a trait à *b(t)*, l'expression va comme suit :

$$b(t)_{partiel}=\left[H_{SI}(1-SF)(1-r)-f(W_2)(e_s-e_a)-\sigma\varepsilon_w B_l+\sigma\varepsilon_a(273.15+T_a)^4+6.1x10^{-4}P_{atm}f(W_2)\right]f_{extinc}+K_b T_b. \qquad [1.36]$$

Selon la même trame que précédemment, en cas de pluie on a

$$b(t)=b(t)_{partiel}+c_p T_a p_r, \qquad [1.37]$$

tandis que s'il y a neige, l'expression devient :

$$b(t)=b(t)_{partiel}-(-c_{pi}T_a+L_i)p_s f_{extinc}. \qquad [1.38]$$

Il est important de mentionner que les équations 1.33 à 1.38 sont exactement identiques à celles inscrites au sein du code de calcul de H2D2 (selon la version 1.8 du fichier **cd2d_tmp_prcprno.for**[3]).

[3] Fichier disponible sur le CD d'accompagnement.

2 MODÈLE MATHÉMATIQUE

L'équation de transport-diffusion, bidimensionnelle, moyennée sur la profondeur, est à la base du modèle de température (Morin *et al.* 2002). En faisant l'hypothèse de conditions hydrodynamiques stationnaires, l'équation différentielle partielle en formulation non conservative est donnée par :

$$H\frac{\partial T}{\partial t} + H(u\frac{\partial T}{\partial x} + v\frac{\partial T}{\partial y}) - \frac{\partial}{\partial x}H(D_{xx}\frac{\partial T}{\partial x} + D_{xy}\frac{\partial T}{\partial y}) - \frac{\partial}{\partial y}H(D_{yx}\frac{\partial T}{\partial x} + D_{yy}\frac{\partial T}{\partial y}) + QH(T - T^0) - \frac{S}{\rho c_p} = 0 \qquad [2.1]$$

où t et (x, y) sont le temps et les coordonnées cartésiennes ; T, l'inconnue, représente la température moyenne de l'eau sur la profondeur ; H est la profondeur ; u et v sont les composantes de la vitesse ; Q est le débit par unité de volume provenant de tributaires ou de sources souterraines, et T_0 sa température. Dans le dernier terme, S, ρ et c_p sont le flux thermique présenté à la section précédente, la densité et la chaleur spécifique de l'eau. Les coefficients de dispersion D_{xx}, D_{yy}, D_{xy}=D_{yx} sont dérivés de l'hydrodynamique et ajustés afin de reproduire au mieux les mesures.

Généralement, le système est résolu dans le temps, après avoir spécifié des conditions aux limites et fourni une solution initiale. Sur une frontière d'entrée, la température est prescrite comme fonction du temps. Sur une frontière de sortie, on prescrit une condition de sortie libre ($\partial T/\partial \boldsymbol{n} = 0$). De plus, il faut tenir compte des zones asséchées, dans lesquelles la profondeur est mise à zéro et la température fixée à celle du fond.

2.1 Simplification de l'équation de transport-diffusion

Dans le cas de notre étude, puisque nous nous intéressons uniquement au bilan thermique et que l'aspect hydrodynamique est mis de côté, l'équation de transport-diffusion 2D se voit grandement simplifiée.

En assumant des coefficients de dispersion nuls ($D_{xx} = D_{yy} = D_{xy} = D_{yx} = 0$), de même que les composantes de la vitesse ($u = v = 0$) et en négligeant toute contribution provenant de tributaires ($Q = 0$), l'équation 2.1 devient :

$$H\frac{\partial T}{\partial t} = \frac{S}{\rho c_p} \qquad [2.2]$$

En insérant l'équation 1.31, on trouve :

$$\frac{\partial T}{\partial t} - \frac{a(t)T}{H\rho c_p} = \frac{b(t)}{H\rho c_p} \qquad [2.3]$$

Il incombe par la suite au logiciel de calcul H2D2 de résoudre cette équation, d'abord de façon stationnaire pour le temps initial, puis en transitoire pour chacun des pas de temps selon le bilan thermique.

Selon Greenberg (1998), pour une équation différentielle linéaire de la forme

$$y' + p(x)y = q(x), \qquad [2.4]$$

comme c'est le cas pour l'équation 2.3, nous obtenons une solution de la forme

$$y(x) = e^{-\int p(x)dx}\left(\int e^{\int p(x)dx} q(x)dx + C\right). \qquad [2.5]$$

Conséquemment, nous pouvons conclure que

$$T(t) = e^{\frac{1}{H\rho c_p}\int a(t)dt}\left(\frac{\int e^{-\frac{1}{H\rho c_p}\int a(t)dt} b(t)dt}{H\rho c_p} + C\right). \qquad [2.6]$$

En somme, en tenant compte de nos hypothèses de départ, nous constatons que la solution obtenue à l'équation de transport-diffusion simplifiée est de forme exponentielle.

2.2 Résolution numérique par les méthodes d'Euler[4]

La discrétisation de l'équation 2.3, qui constitue un problème non stationnaire du premier ordre, par la méthode des éléments finis, conduit à l'équation différentielle suivante :

$$\{\dot{T}\} + [K]\{T\} = \{F\} \text{ pour } t > t_0 \qquad [2.7]$$

et

$$\{T(t_0)\} = \{T_0\} \qquad [2.8]$$

Puisque nous sommes dans un système linéaire, [K] et { F }sont indépendants de { T } et de ses dérivées.

[4] Selon Dhatt et Touzot (1981).

Nous pouvons écrire, en premier lieu :

$$\{\dot{T}\}=\{f(\{T\},t)\} \qquad [2.9]$$

Ce qui donne, dans notre cas :

$$\{f\}=\{F\}-[K]\{T\}. \qquad [2.10]$$

2.2.1 *Méthode d'Euler explicite* (α = 0)

Nous pouvons maintenant discrétiser $\dot{T}$ par la formule des différences finies décentrée à gauche :

$$\{T(t)\}=\{\dot{T}_t\}\approx\frac{1}{\Delta t}(\{T_{t+\Delta t}\}-\{T_t\}). \qquad [2.11]$$

En utilisant cette relation pour discrétiser [2.9] à l'instant t, on obtient :

$$\{T_{t+\Delta t}\}=\{T_t\}+\Delta t\{f(\{T_t\},t\}. \qquad [2.12]$$

Puisque $\{f\}$ ne fait pas intervenir l'inconnue $\{T_{t+\Delta t}\}$, cette forme de récurrence de type Euler est dite explicite. Bien que cette méthode numérique soit simple, elle est inconditionnellement instable et peu précise, donc peu employée. La méthode d'Euler explicite possède une erreur du premier ordre.

2.2.2 *Méthode d'Euler implicite* (α = 1)

Cette méthode consiste à écrire [2.9] à l'instant $t + \Delta t$, de même qu'à utiliser la formule de différences finies décentrée à droite :

$$\{\dot{T}_{t+\Delta t}\}\approx\frac{1}{\Delta t}(\{T_{t+\Delta t}\}-\{T_t\}). \qquad [2.13]$$

D'où la formule de récurrence d'Euler implicite :

$$\{T_{t+\Delta t}\}=\{T_t\}+\Delta t\{f(\{T_{t+\Delta t}\},t+\Delta t)\}. \qquad [2.14]$$

Cette formule est ainsi dénommée puisqu'elle fait intervenir le vecteur inconnu $\{T_{t+\Delta t}\}$. La méthode d'Euler implicite, bien que très stable, est diffusive, c'est-à-dire qu'elle a tendance à amortir la solution, en n'y détectant pas tous les petites variations. La méthode d'Euler implicite possède une erreur du premier ordre.

2.2.3 *Méthode d'Euler semi-implicite* (0 ≤ α ≤ 1)

Cette méthode consiste à écrire l'équation 2.9 à l'instant $t + \alpha\Delta t$, avec $0 \leq \alpha \leq 1$. On peut ensuite exprimer la formule d'Euler ainsi :

$$\{T_{t+\Delta t}\}=\{T_t\}+\Delta t\{f(\{T_{t+\alpha\Delta t}\},t+\alpha\Delta t)\}, \qquad [2.15]$$

où

$$\{T_{t+\alpha\Delta t}\}=\alpha\{T_{t+\Delta t}\}+(1-\alpha)\{T_t\}. \qquad [2.16]$$

Lorsque $\alpha = 0.5$, nous sommes en présence de l'algorithme de Crank-Nicholson. À mi-chemin entre le Euler explicite et implicite, cet algorithme possède une erreur du deuxième ordre.

3 DESCRIPTION DU SITE

3.1 Localisation géographique

Le marais aménagé de Saint-Barthélemy est situé en bordure de l'autoroute 40, à proximité du village de Saint-Barthélemy, entre Louiseville et Berthierville. Il est localisé à l'ouest du lac Saint-Pierre, tout juste au nord des Îles de Sorel. Le marais est divisé en portions numérotées de 1 à 6, comme nous l'illustre la figure 3.1.

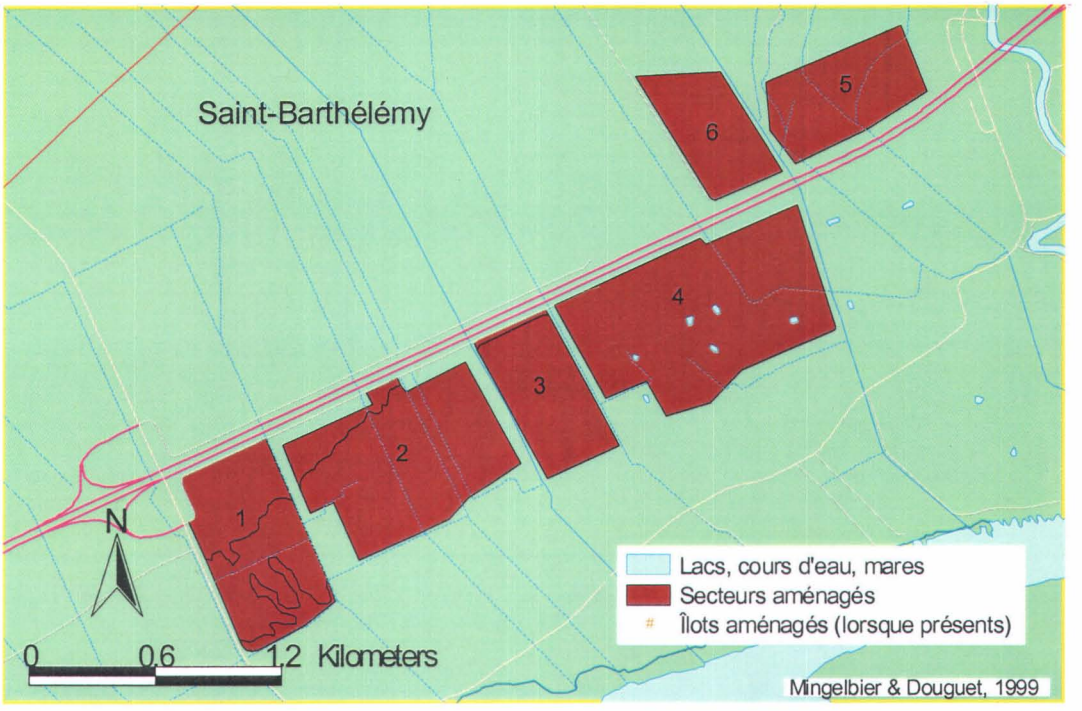


Figure 3.1 – Illustration des 6 segments composant le marais aménagé de Saint-Barthélemy

Sur cette dernière figure, on remarque en rose l'autoroute 40. C'est le segment 4 qui occupe la plus grande superficie, alors que le segment 6 semble le plus petit des secteurs aménagés.

Ainsi, uniquement le segment 4 est modélisé, puisque c'est le seul possédant des thermistors dans ses eaux. La figure 3.2 indique donc l'emplacement du segment 4, au moyen d'une flèche noire.

Figure 3.2 – Localisation du segment 4 du marais aménagé de Saint-Barthélemy

Une photographie aérienne est présentée à la figure 3.3. Le segment 4 y est encerclé. En portant attention à l'illustration, on y constate la présence d'arbres, principalement concentrés dans deux zones distinctes.

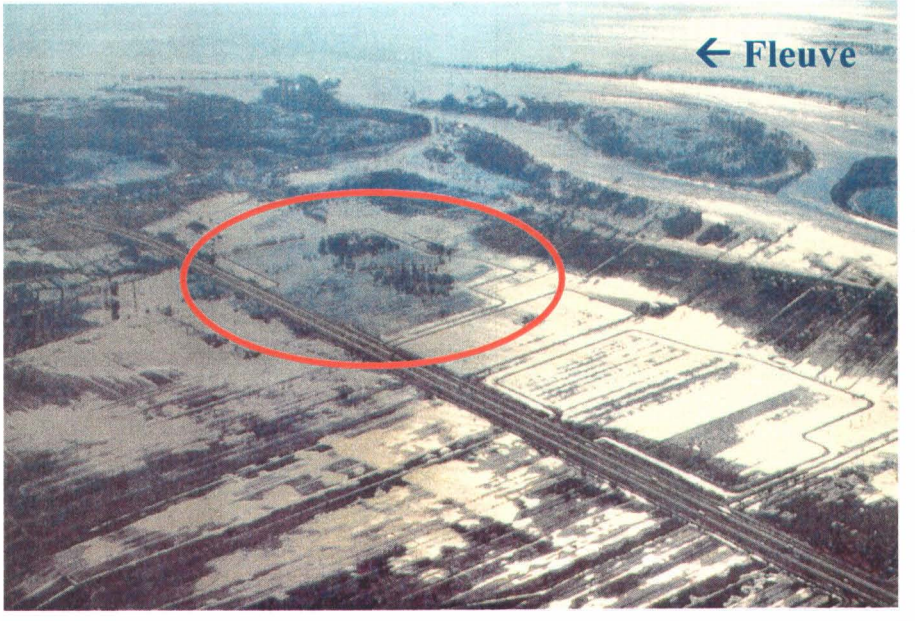


Figure 3.3 – Photographie aérienne présentant les différents secteurs du marais de Saint-Barthélemy

Au centre de la figure ci-haut, on peut observer la présence de l'autoroute 40. Au nord, on distingue une importante masse d'eau : c'est à cet endroit que le fleuve Saint-Laurent devient le lac Saint-Pierre.

3.2 Aménagement du marais

Le segment 4 est un ancien champ d'agriculteur. Cependant, plus aucune activité agricole n'y a cours depuis son achat, en 1997, par la fondation Canards Illimités. Depuis, il s'agit d'un marais aménagé pour les poissons, qui viennent y frayer.

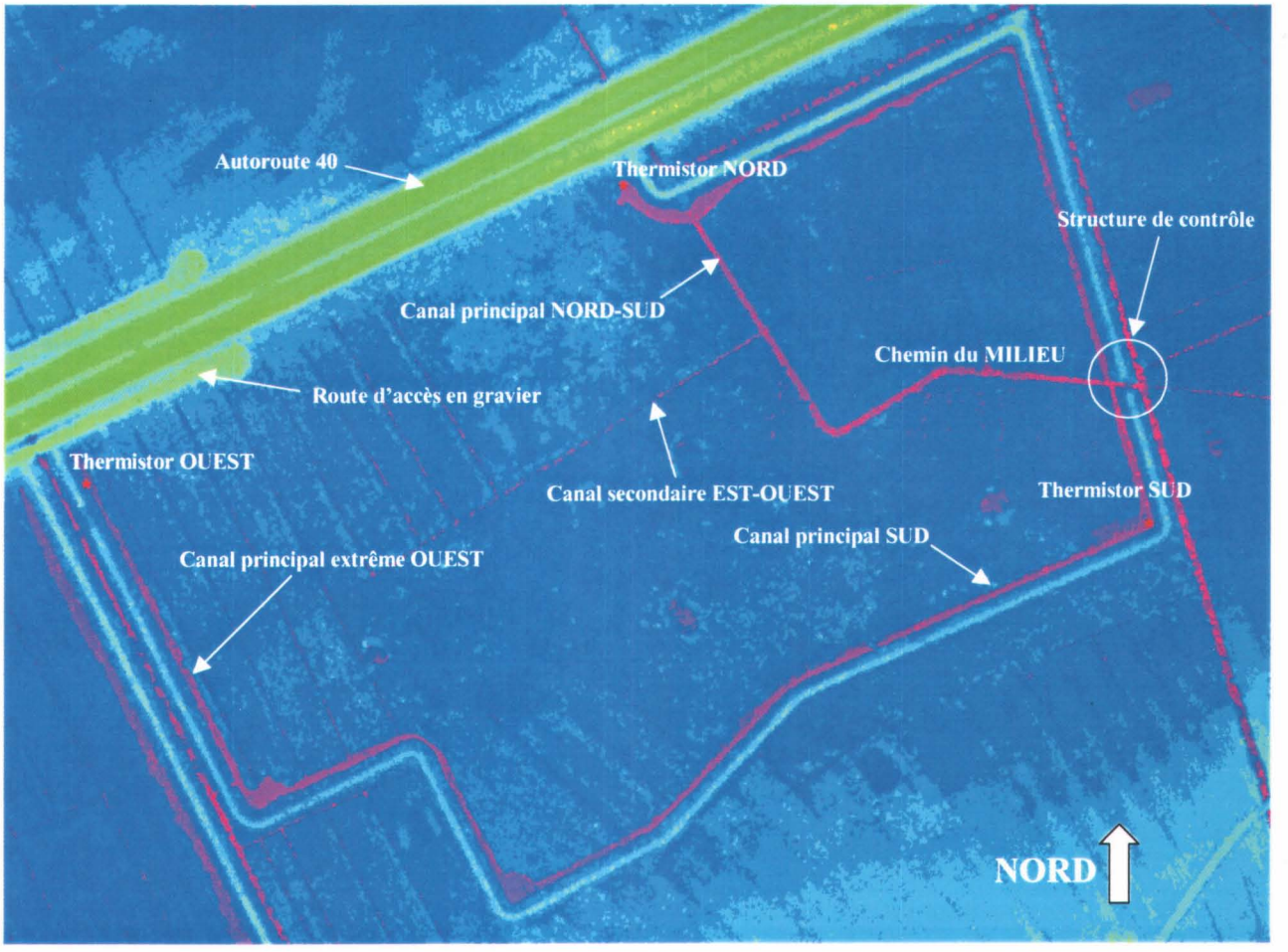


Figure 3.4 – Schéma détaillant l'arrangement des canaux et la position des thermistors dans le segment 4

À la figure précédente, on remarque la présence de multiples canaux à l'intérieur du marais, qui est pourvu d'une digue de ceinture (en turquoise) possédant une seule ouverture, soit la structure de contrôle. Afin de se retrouver dans tout ce labyrinthe de canaux, des noms ont été décernés aux plus importants. L'emplacement de trois thermistors intéressés à la température de l'eau en moyenne horaire y est aussi indiqué.

3.3 Régularisation des entrées / sorties d'eau

En tout temps, à l'exception des précipitations et d'un léger ruissellement sur les digues vers l'intérieur du marais, toute entrée ou sortie d'eau transite par la structure de contrôle (voir figure 5.10), qui dispose d'une porte d'environ 1 m² laissant entrer l'eau du fleuve. Lors de l'année 2002, du 1er avril au 9 mai, la vanne est ouverte, ce qui alloue l'entrée d'eau provenant du fleuve dans le segment 4. À partir du 9 mai 2002, la vanne se voit fermée. À la fin du mois de juillet, une vidange complète est effectuée, i.e. toute l'eau contenue dans le marais est évacuée. L'élévation des canaux principaux et secondaires est pensée de façon à ce qu'ils puissent se vider complètement lors de l'opération, qui dure généralement une dizaine de jours.

Lors du mois d'avril 2002 et au début de mai 2002, le marais est en contact avec le fleuve, par le biais de la structure de contrôle ouverte. Certes, l'eau du fleuve est plus froide que celle déjà présente dans le marais. On peut alors comprendre que le thermistor SUD, situé à proximité de la vanne, soit affecté par cette contribution d'eau froide. Cependant, il est raisonnable d'assumer que les mesures prises par les thermistors NORD et OUEST ne soient pas perturbées par ce phénomène. Dans un autre ordre d'idées, le courant associé à l'entrée d'eau du fleuve vers le marais est jugé négligeable.

4 MÉTHODOLOGIE

La température de l'eau est simulée en utilisant le logiciel de calcul H2D2 (annexe G) adapté au secteur du segment 4 du marais aménagé de Saint-Barthélemy.

Les étapes du processus complet de modélisation sont les suivantes :

1) monter le modèle numérique de terrain et le maillage de calcul
2) à l'aide des données topographiques, de niveaux de surface et l'exécutable **profondeur.exe**[5], obtenir les profondeurs.
3) à l'aide d'un modèle de transport-diffusion 2D pour la température, réaliser des simulations préliminaires afin de calibrer le modèle.
4) à l'aide du même modèle de transport-diffusion, réaliser une simulation sur toute la période de disponibilité des données, soit du 10 avril 2002 au 6 mai 2002 en pas horaire (26 jours * 24 heures = 624 itérations).
5) utiliser le signal de température et le comparer aux données horaires mesurées par les thermistors sur le terrain, placés à trois différents sites à raison d'un thermistor par site.

La modélisation de la température intègre tous les paramètres météorologiques (11) importants dans l'équilibre des températures de l'eau.

4.1 Implémentation numérique

4.1.1 Maillage

Le maillage éléments finis contient les informations nécessaires au calcul de la température de l'eau. Il comporte 80 121 nœuds et 157 999 éléments, sur une grille non structurée et adaptée aux divers besoins de précision spatiale des simulations (voir annexe B). La taille des mailles varie entre 1.5 m et 20 m de côté, et la densité des mailles est plus importante dans les régions où sont situés les thermistors de même que lorsqu'il y a présence de sillons et de fossés. Le logiciel Modeleur a été utilisé pour la création du maillage, puis pour son exportation en fichier compatible avec H2D2.

4.1.2 Propriétés nodales

À l'exception de la profondeur, les données hydrodynamiques sont fixées à zéro en tout temps et en tout lieu de l'espace :

1) vitesse en x : 0 m/s ;
2) vitesse en y : 0 m/s ;
3) **profondeur variable dans le temps et dans l'espace**;
4) diffusivité verticale : 0 m^2/s ;
5) diffusivité horizontale : 0 m^2/s.

[5] Fichier disponible sur le CD d'accompagnement.

4.1.3 Propriétés globales

Les propriétés globales sont des informations contenues en chaque nœud qui ne varient ni dans le temps ni d'une solution à l'autre. Voici les valeurs prises par les propriétés globales dans notre cas (selon l'ordre de lecture de H2D2) :

1) diffusivité moléculaire : **déterminée à l'issue des tests préliminaires** (section 6.1);
2) coefficient de pondération de la diffusivité verticale : 1.00000;
3) coefficient de pondération de la diffusivité horizontale : 1.00000;
4) coefficient de pondération de la diffusivité longitudinale : 1.00000;
5) masse volumique de l'eau : 1.00000 x 10^3 kg/m^3;
6) chaleur spécifique de l'eau : 4.18160 x 10^3 J/kg·°C;
7) masse volumique de la glace : 9.17000 x 10^2 kg/m^3;
8) chaleur spécifique de la glace : 2.07500 x 10^3 J/kg·°C;
9) chaleur latente de fusion de la glace : 3.34000 x 10^5 J;
10) pression atmosphérique : 1.01260 x 10^5 Pa;
11) température du fond : **déterminée à l'issue des tests préliminaires** (section 6.1);
12) coefficient de linéarisation A_l : 1.06297142669025 x 10^8 K^3;
13) coefficient de linéarisation B_l : 5.33908718317524 x 10^9 K^4.

Il est à remarquer que les coefficients de pondération des 3 types de diffusivité sont tous fixés à 1, alors les coefficients de dispersion sont nuls pour leur part. Par conséquent, aucun phénomène de dispersion n'est considéré lors de la simulation.

4.1.4 Conditions limites

Partout sur les frontières, nous sommes en présence de conditions de sortie libre.

4.1.5 Paramètres de résolution

La correction maximale allouée entre chacune des itérations est de 50°C. La méthode d'Euler est utilisée pour les calculs en régime non-stationnaire. Le paramètre alpha est déterminé au terme des tests préliminaires (section 6.1). Principalement, il s'agit de voir quel méthode, semi-implicite ($\alpha = 0.5$) ou implicite ($\alpha = 1$), engendre les meilleurs résultats.

5 DONNÉES ENVIRONNEMENTALES

5.1 Données météorologiques

5.1.1 *Provenance des données*

Les données météorologiques utilisées sont des données horaires moyennées, qui proviennent des stations Lac Saint-Pierre (station 701LP0N), Nicolet (station 7025442) et Dorval (station 7025250). Elles sont extraites de bases de données (annexes A et E) du Service Météorologique du Canada d'Environnement Canada pour la période allant du 1 mars 2002 au 31 août 2002.

Pour chacune des variables mesurées, selon la disponibilité des données, la station la plus près du domaine de simulation est utilisée. Ces variables sont jugées représentatives pour l'ensemble du domaine de simulation. Les variables directement extraites concernent :

- la couverture nuageuse (station 7025250);
- l'humidité relative (station 7025442);
- les précipitations totales (station 7025442);
- la pression atmosphérique (station 7025442);
- la radiation solaire (station 7025442);
- la température de l'air (station 701LP0N);
- la vitesse du vent (station 701LP0N).

Il est à remarquer qu'aucune donnée de couverture nuageuse n'est disponible dans la région du Lac Saint-Pierre pour la période d'intérêt. Pour cette raison, nous devons avoir recours aux mesures effectuées à l'aéroport de Dorval, qui est situé à environ 100 km à l'ouest du marais de Saint-Barthélemy. Tout décalage potentiel entre les données est négligé.

Les variables extraites sont combinées pour fournir les 11 variables réparties (ici présentées dans l'ordre requis par H2D2), données d'entrée au modèle de température et nécessaires à la simulation :

- la radiation solaire;
- le couvert végétal;
- la réflectivité de l'eau;
- la température de l'air;
- l'émissivité de l'atmosphère;
- la valeur de la fonction du vent;
- l'humidité relative;
- les précipitations;
- le coefficient de transfert thermique avec la glace;
- la fonction d'extinction;
- le coefficient de transfert thermique avec le fond.

Il est important de souligner que la direction du vent n'a aucune importance en ce qui concerne le traitement du vent effectué par H2D2.

Bien que cela ne cause pas un enjeu important dans cette étude-ci en raison de leur rareté, il peut arriver qu'il y ait des données manquantes au sein des fichiers de données météorologiques[6]. Advenant une telle situation, dans la base de données fournie par le Service de Climatologie d'Environnement Canada, le code `-99999M` est inscrit. Lorsque la phase d'extraction des données a lieu, le temps associé à la donnée manquante (appelons-le t_M) n'est pas inscrit dans le fichier de données extraites, ce qui fait qu'une interpolation doit être effectuée entre le temps précédant et suivant t_M.

Il s'avère intéressant d'aborder chacun des paramètres météorologiques sous forme graphique afin d'étudier leur comportement lors de la période à l'étude (sections 5.1.2 à 5.1.8).

5.1.2 *Température de l'air*

Les moyennes quotidiennes de température de l'air pour la période à l'étude sont décrites à la figure 5.1.

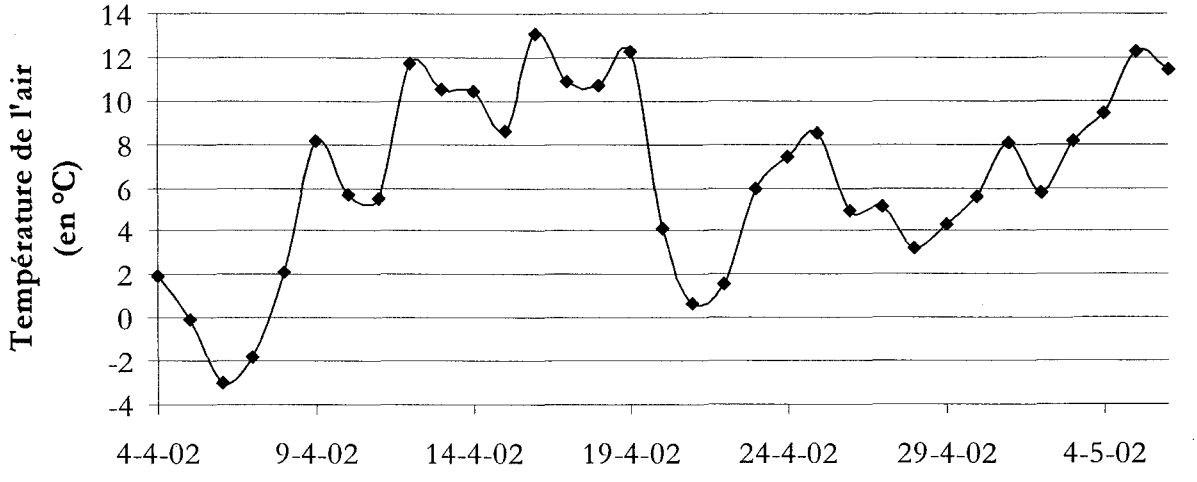


Figure 5.1 – Température de l'air (station Lac Saint-Pierre) pour la période d'intérêt

On peut constater que la moyenne de la température de l'air est sous la barre des zéros pour la période allant du 5 au 7 avril. La présence de glace et de neige est alors probable. Ensuite, les températures augmentent pour culminer lors de la journée du 16 avril, où la moyenne quotidienne dépasse légèrement les 13 degrés. Un important refroidissement sur une durée de 3 jours s'opère ensuite à partir du 20 avril. Il est à noter que le 21 avril, la température moyenne de l'air est tout près du point de congélation. Ensuite, on constate un réchauffement rapide, suivi d'une baisse progressive de température. À partir du 28 avril cependant, les températures augmentent graduellement, bien certaines oscillations dans les valeurs soient présentes.

5.1.3 *Couverture nuageuse*

Les moyennes quotidiennes de couverture nuageuse pour la période à l'étude sont présentées à la figure 5.2.

[6] Disponibles sur le CD d'accompagnement.

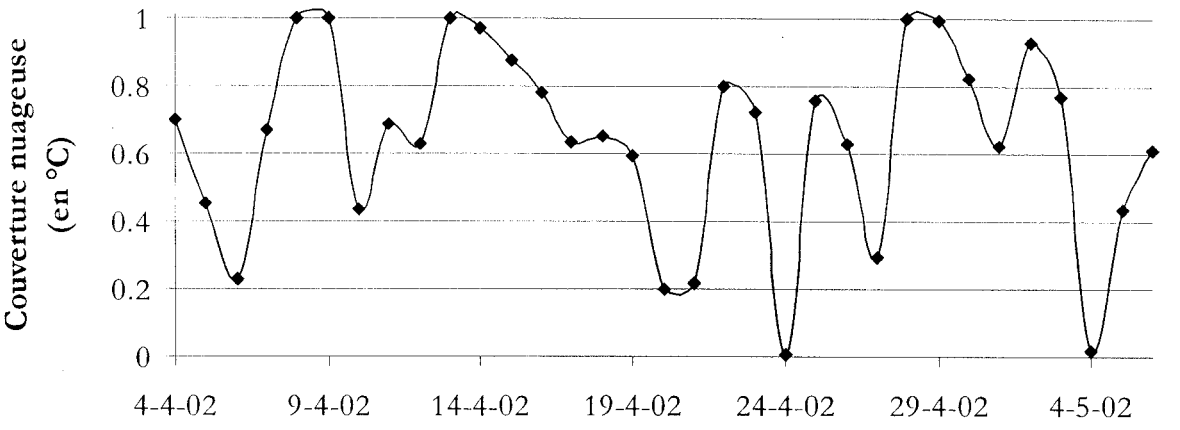


Figure 5.2 –Couverture nuageuse (station Dorval) pour la période d'intérêt

Il est difficile d'identifier une tendance dans les valeurs prises par la couverture nuageuse entre le 4 avril et le 6 mai. Néanmoins, on peut observer que les oscillations sont nombreuses. Les journées entières de ciel dégagé sont rares : seulement les 24 avril et 4 mai. D'autre part, on distingue 5 journées totalement nuageuses : les 8, 9, 13, 28 et 29 avril. En somme, on constate que la tendance, sur l'ensemble de la période, est au ciel partiellement dégagé.

5.1.4 Humidité relative

Les moyennes quotidiennes d'humidité relative pour la période à l'étude sont illustrées graphiquement à la figure 5.3.

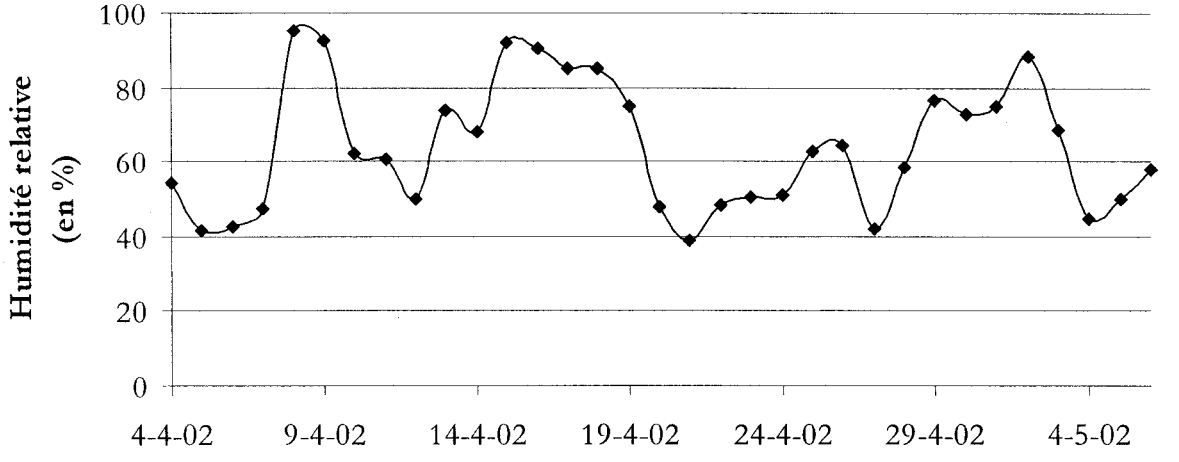


Figure 5.3 – Humidité relative (station Nicolet) pour la période d'intérêt

Tout au cours de la période, l'humidité relative oscille de part et d'autre d'une valeur moyenne qu'on pourrait approximer à 60 %. L'humidité est maximale lors des journées du 8 et 9 avril, qui possèdent respectivement des valeurs de 95 % et 92 %. Une seule fois on tombe sous la barre des 40 % de moyenne, soit le 21 avril (38.8 %).

5.1.5 Précipitations totales

Les précipitations totales quotidiennes (neige + pluie) pour la période à l'étude sont présentées à la figure 5.4.

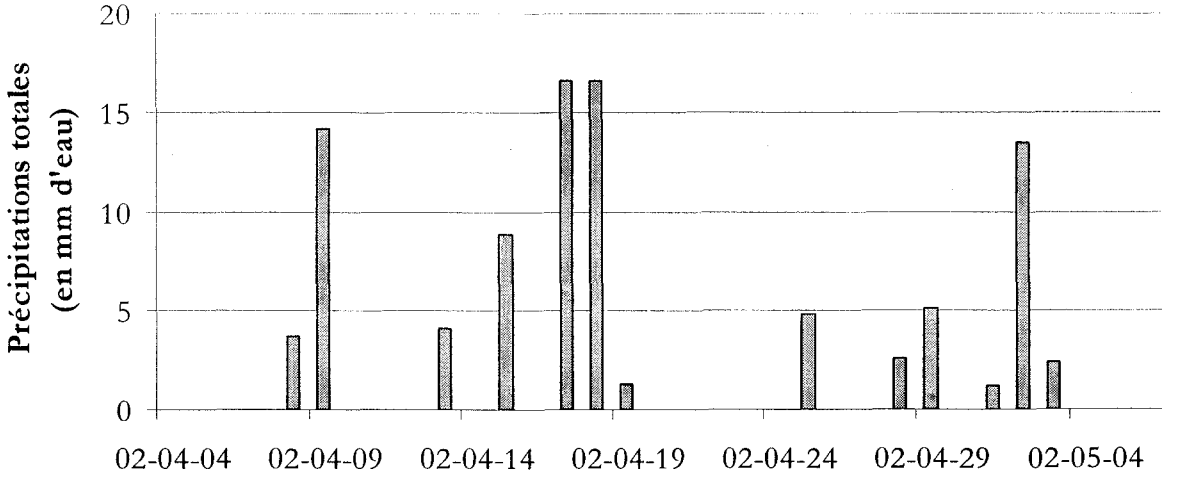


Figure 5.4 – Précipitations totales quotidiennes (station Nicolet) pour la période à l'étude

60 % des journées comprises entre le 4 avril et le 6 mai 2002 n'ont pas connu de précipitations. D'autre part, du 40 % résiduel, un près de la moitié des journées pluvieuses se sont situées sous la barre des 5 mm de précipitations. En somme, seulement 4 journées peuvent donc être considérées très pluvieuses : les 9 avril (14.2 mm), 17 avril (16.6 mm), 18 avril (16.6 mm) et 2 mai (13.5 mm).

5.1.6 Vitesse du vent

Les moyennes de vitesse du vent quotidiennes pour la période à l'étude sont illustrées à la figure 5.5.

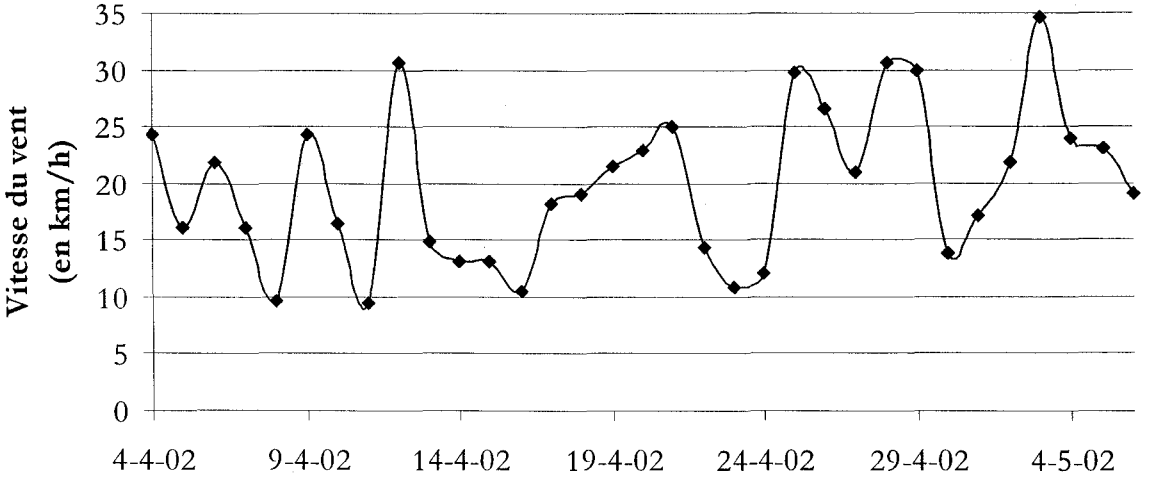


Figure 5.5 – Vitesse du vent (station Lac Saint-Pierre) pour la période à l'étude

Dans la portion gauche du graphique, les valeurs moyennes de vitesse du vent oscillent avec une amplitude croissante. À partir du 17 avril, progressivement pendant 5 jours, la vitesse du vent est augmentée jusqu'à ce qu'elle atteigne 25 km/h. Ensuite, il y a une importante diminution puis d'autres fluctuations qui vont même au-dessus de la barre des 30 km/h. À ce sujet, seulement 3 jours possèdent une moyenne quotidienne de vitesse du vent supérieure à 30 km/h, soit le 12 avril (30.67 km/h) et le 28 avril (30.71 km/h) ainsi que le 3 mai (34.63 km/h). À l'inverse, les journées les moins venteuses sont les 8 (9.54 km/h), 11 (9.46 km/h), 16 (10.54 km/h) et 23 avril (10.83 km/h). La moyenne pour la période se situe dans les environs de 20 km/h.

5.1.7 Radiation solaire

Les moyennes quotidiennes de radiation solaire pour la période à l'étude sont illustrées à la figure 5.6.

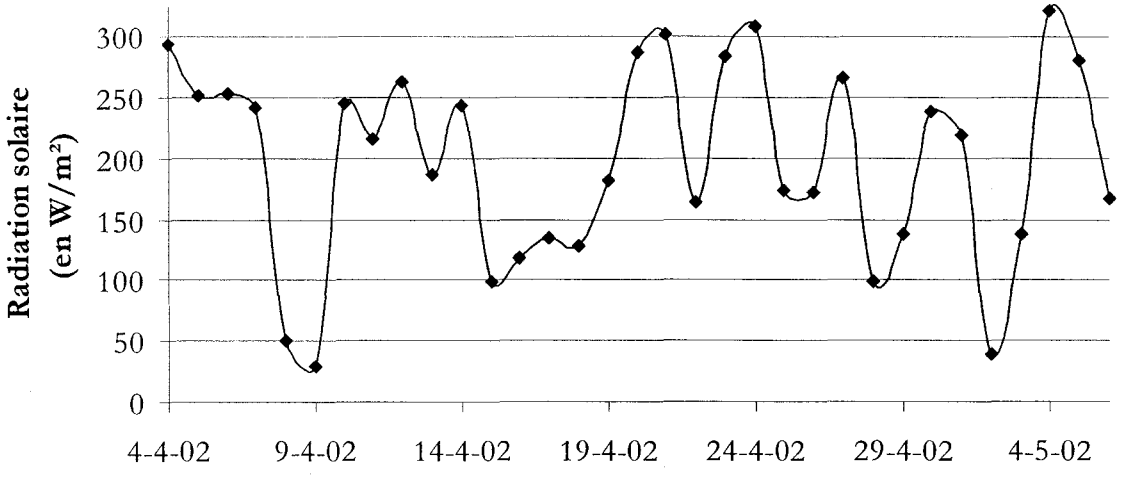


Figure 5.6 – Radiation solaire (station Nicolet) pour la période à l'étude

La moyenne sur cette période se situe dans les 200 W/m². Plusieurs fluctuations sont observées entre le 4 avril et le 6 mai, notamment dans la portion droite du graphique où elles sont quelque peu régulières. À noter les journées où les radiations solaires se sont faites plus discrètes : 8 avril (50.5 W/m²), 9 avril (28.8 W/m²) et 2 mai (39.8 W/m²). On observe une corrélation nette entre ces dates et les journées pluvieuses et nuageuses (voir figures 5.2 et 5.4). D'autre part, les journées de très fortes radiations ont lieu passée la mi-avril. Il s'agit plus particulièrement des 21 (302.2 W/m²) et 24 avril (308.6 W/m²) ainsi que du 4 mai (322.3 W/m²).

5.1.8 Pression atmosphérique

Les moyennes quotidiennes de pression atmosphérique pour la période à l'étude sont présentées à la figure 5.7.

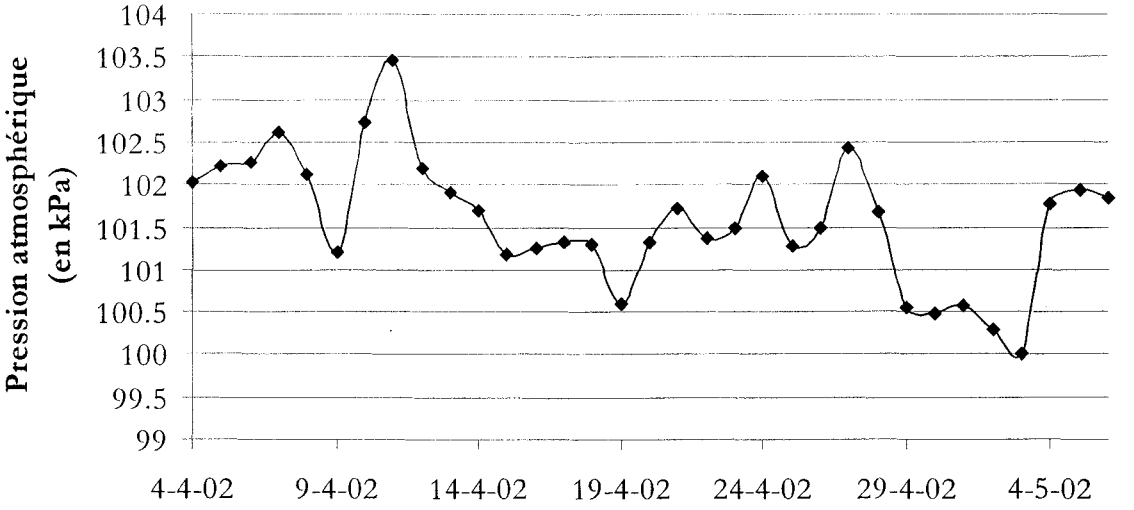


Figure 5.7 – Pression atmosphérique (station Nicolet) pour la période à l'étude

La pression atmosphérique n'est pas extrêmement variable sur la période d'intérêt, oscillant de part et d'autre d'une valeur moyenne d'environ 101.6 kPa. La moyenne quotidienne la plus basse est atteinte le 3 mai, avec 100 kPa. À l'opposé, le 11 avril, elle culmine à près de 103.5 kPa.

5.2 Données topographiques

Les données topographiques sont obtenues à partir de la base de données topométriques assemblée par le Service Météorologique du Canada. Un sondage par laser aéroporté est effectué en octobre 2001. À ce moment, la vanne d'eau est ouverte, donc le marais est en phase avec le fleuve. Le niveau fluvial est jugé bas au moment des mesures et vu l'élévation des canaux permettant un vidange complète (section 3.3), il est justifié de négliger toute présence d'eau résiduelle dans le marais. L'annexe C présente le champ scalaire de topographie sous Modeleur.

5.3 Données de température de l'eau

5.3.1 Détails techniques sur les thermistors

La température de l'eau est mesurée à l'aide de trois thermistors (OUEST, NORD et SUD) placés à environ 20 cm du fond dans le segment 4 du marais aménagé de Saint-Barthélemy. Chaque thermistor est en réalité contenu dans un dispositif cylindrique d'environ 2 cm de diamètre par 10 cm de longueur. Une fois programmé en mode horaire, l'appareil est maintenu au plafond d'une cage en plastique (voir figure 5.8) dotée d'une brique à sa base. Chaque thermistor dispose d'une autonomie énergétique d'environ 5 années. Cependant, au terme d'environ un an de stockage de données horaires, l'espace mémoire devient saturé et il faut venir récolter les mesures de températures, pour finalement réinitialiser l'appareil. Chaque cage en plastique est attachée à un piquet par le biais d'une corde (figure 5.9). En plus de stabiliser sa position, le piquet permet d'identifier au loin le site du thermistor.

Figure 5.8 – Cage en plastique dans laquelle est installé le thermistor (photo prise le 10 juin 2004)

Figure 5.9 –Piquet de soutien du thermistor NORD (photo prise le 10 juin 2004)

5.3.2 *Localisation des thermistors*

Les coordonnées de l'emplacement de chacun des thermistors sont d'abord relevées à l'aide d'un GPS. Or, lors de l'introduction de ces données dans le projet Modeleur, nous pouvons clairement distinguer la divergence entre les positions à l'écran et celles observées sur les lieux, en étudiant minutieusement le plan du terrain via le champ scalaire de topographie. Il est donc par la suite convenu que l'erreur associée aux GPS est considérable et qu'il faut trouver une façon plus précise de localiser les thermistors. Dans cette optique, Philippe Brodeur situe sur une carte du segment 4, par l'intermédiaire d'une croix, l'emplacement de chacun des trois thermistors. Ensuite, en tentant de situer le plus fidèlement possible la position de chacune de ces croix, les

coordonnées sont extraites grâce à la sonde dans Modeleur et au champ scalaire de topographie. Cette localisation manuelle des thermistors s'avère beaucoup plus près de la réalité qu'en utilisant les coordonnées GPS directement.

Enfin, pour obtenir une estimation de l'élévation de chaque thermistor, nous utilisons la sonde dans Modeleur pour relever la cote topographique à chacun des sites. Ensuite, 20 cm sont ajoutés à cette valeur, compte tenu de la position du thermistor dans la cage de plastique (suspendu au plafond), ladite étant déposée au fond du marais.

Les coordonnées des thermistors et leur élévation sont contenues dans le tableau 5.1.

	Coordonnées MTM zone 8 NAD 83		Estimation de la cote d'élévation NMM (en m)
Thermistor OUEST	338782	5114448	4.75
Thermistor NORD	339435	5114810	4.20
Thermistor SUD	340073	5114400	4.35

Tableau 5.1 – Position (MTM, zone 8, NAD 83) des trois thermistors présents au segment 4 (2002)

5.4 Données de niveaux de surface

La disponibilité des données de niveaux de surface s'étend du 4 avril au 6 mai 2002. Il s'avère important de comprendre comment sont exécutées ces mesures.

Toutes les données sont prélevées à côté de la vanne. La cote topographique du « U » formé par la structure de contrôle (voir figure 5.10) est connue, à 5.8 m. Il suffit d'abord de mesurer la différence entre la surface de l'eau et la surface horizontale de la structure. En soustrayant ce résultat à 5.8 m, le niveau de surface dans le marais peut alors être obtenu.

Figure 5.10 – Structure de contrôle régissant les entrées et sorties d'eau

Les données de niveaux de surface sont illustrés graphiquement à la figure 5.11.

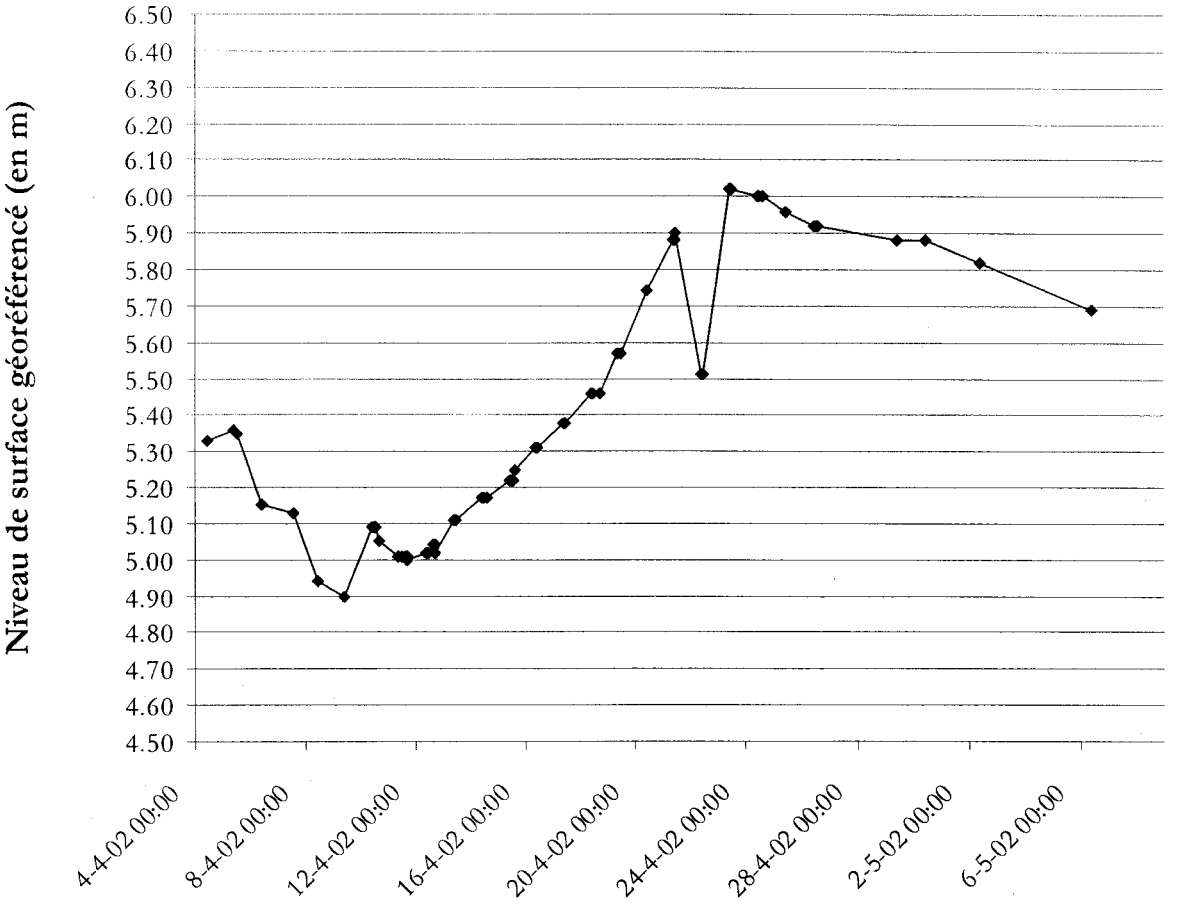


Figure 5.11 – Niveaux de surface géoréférencés pour la période de simulation sur le segment 4 du marais aménagé de Saint-Barthélemy

À remarquer l'augmentation progressive du niveau d'eau entre le 9 et le 21 avril, pour ensuite diminuer brusquement le 22 et gagner un niveau de surface au-dessus de la barre des 6 mètres lors de la matinée du 23 avril. Passée cette date, la quantité d'eau dans le bassin diminue faiblement de façon régulière. On peut aussi constater que la densité des données s'amenuise vers la fin du mois d'avril. Les deux dernières mesures ont lieu les 2 et 6 mai 2002.

5.5 Données sur la qualité de l'eau

Une importante turbidité implique la présence d'une grande quantité de matières en suspension dans la colonne d'eau. Ces petites particules participent à disperser davantage la radiation solaire incidente. Par conséquent, un certain pourcentage de l'énergie est réfléchi, ce qui entraîne une température de l'eau légèrement plus froide que pour des eaux totalement claires (Lilga M.C. 2003).

À ce sujet, quelques mesures de la turbidité de l'eau du marais effectuées au printemps 2002 sont présentées (tableau 5.2).

Site de mesure	Date	Coordonnées UTM zone 18		Turbidité	
				NTU	Approximation disque de Secchi
Près de la structure de contrôle	15 avril 2002	651004	5114803	219	0.078484174
Thermistor NORD	15 avril 2002	650435	5115016	58.45	0.200537288
Thermistor OUEST	15 avril 2002	649778	5114654	23.7	0.380731864
Thermistor NORD	24 avril 2002	650435	5115016	45.5	0.239577377
Thermistor OUEST	24 avril 2002	649733	5114670	260	0.069478659

Tableau 5.2 – Données de turbidité de l'eau

Il nous est possible de convertir les unités NTU afin d'obtenir un équivalent approximatif en mètres associés à l'expérience du disque de Secchi. On se rappellera que le disque de Secchi, comportant des quadrants blancs et noirs, permet, une fois plongé dans l'eau, d'en déterminer sa transparence en notant la profondeur à partir de laquelle il n'est plus visible. L'équation 5.1 (selon le fichier **secchi vs ntu.doc**[7], fourni par Philippe Brodeur) propose une correspondance entre les unités NTU et Secchi (m) :

$$y = 3.6055x^{-0.7102}, \qquad [5.1]$$

où y représente la valeur Secchi (en m) et x la turbidité en NTU. C'est grâce à cette relation, qui possède une coefficient de détermination de 0.7863, que la colonne de droite dans le tableau 5.2 est obtenue. On peut constater que l'eau du marais est très turbide par endroits. Par surcroît, au terme d'une visite sur le terrain le 10 juin 2004, il est observé que la grande présence de matière organique (notamment des feuilles mortes) au fond des canaux du marais confère à l'eau une couleur noirâtre, qualifiée de tannique.

[7] Fichier disponible sur le CD d'accompagnement.

6 RÉSULTATS DES SIMULATIONS DE TEMPÉRATURE

Il est important de noter que tous les fichiers de commande (format **.inp**) appelant le logiciel H2D2 utilisés lors des simulations de même que tous les fichiers techniques concernant la convergence numérique et la lecture des fichiers d'entrée (format **.out**) sont présents sur le CD d'accompagnement.

6.1 Résultats des tests préliminaires

Il importe de tester quelle combinaison de paramètres implémentés dans le processus de solution est à l'origine des meilleurs résultats de simulation.

Plusieurs simulations sont d'abord menées, en conservant des paramètres sensiblement identiques à ceux utilisés lors de l'étude de la batture Thaillandier (Morin *et al.* 2002). Cependant, en étudiant les champs de températures simulées, nous remarquons que la solution ne possède pas de sens physique en de multiples endroits, plus particulièrement à proximité des digues de ceinture, la où les zones de faible profondeur sont présentes en grande quantité.

Afin d'identifier la cause de cette perte de sens physique de la température (phénomène étant tout de même limitée par la correction maximale allouée de 50 °C à chaque itération), l'intégrité des fichiers de données météorologiques, de conditions limites et d'hydrodynamique est alors vérifiée, ne révélant rien d'anormal. Au terme de multiples tests, il est identifié que la cause de ces données de température de l'eau erronées en de multiples endroits est la diffusivité moléculaire de l'eau. Plusieurs tests sont ensuite menés afin d'identifier la valeur que doit posséder ce paramètre (section 6.1.1).

Aussi, l'impact de la turbidité de l'eau du marais de Saint-Barthélemy est quantifié (section 6.1.2). Ensuite, plusieurs simulations de quelques jours sont réalisées afin d'identifier quelle méthode de résolution, entre Euler implicite (α =1) et Euler semi-implicite (α = 0.5), est à privilégier. Aussi, afin d'ajuster le niveau du signal, trois différentes valeurs de température du fond du marais sont simulées (section 6.1.3).

6.1.1 Détermination de la diffusivité moléculaire de l'eau

Initialement fixée à une valeur de 5.00 x 10^{-2} m²/s (valeur utilisée lors de l'étude de la batture Thaillandier), la diffusivité moléculaire peut provoquer le décrochement de la solution en de multiples endroits.

En pratique, la diffusivité moléculaire de l'eau possède une valeur de l'ordre de 10^{-9} m²/s. Or, dans le cadre d'un calcul utilisant la méthode des éléments finis, vu la taille des mailles (dans ce cas-ci, variant entre 1.5 et 20 m), cette valeur de diffusivité doit être rehaussée pour que le phénomène soit perceptible au final. Étant donné que dans notre cas l'hydrodynamique est fixé à zéro, il s'agit en réalité de l'unique phénomène de diffusion qui est incorporé au système.

Voici les valeurs de diffusivité moléculaires testées :

1) 5.00×10^{-4} m²/s;
2) 5.00×10^{-5} m²/s;
3) 5.00×10^{-6} m²/s;
4) 5.00×10^{-8} m²/s.

Pour chacun des tests, la solution stationnaire pour la journée du 16 avril 2002 à 12:00 est obtenue. Les champs scalaires de températures sont ensuite comparés sous Modeleur, particulièrement dans les zones de faibles profondeurs. Des suites de cette analyse, il est possible de conclure que **la valeur de diffusivité générant les meilleurs résultats est de 5.00×10^{-6} m²/s**.

6.1.2 *Impact de la variation de la turbidité de l'eau*

Afin de rendre la simulation plus réaliste, compte tenu de la forte turbidité de l'eau impliquée dans le marais à l'étude, il est décidé d'effectuer un test réparti sur quelques jours en fixant une valeur de réflectivité de l'eau à 9 % (moyenne des deux valeurs de réflectivité de l'eau mentionnées à la section 1.1 pour des eaux turbides). Pour la même période, une simulation témoin est aussi exécutée, avec une valeur de réflectivité fixée à 0 %.

En comparant les résultats des deux tests, il est possible de conclure que le niveau de la courbe associée à $r = 0.09$ est tout juste inférieur à celui de la courbe témoin. Cela confirme qu'au sein du modèle thermique, pour une plus grande turbidité, les valeurs de température obtenues sont plus faibles. Cependant, la différence de température pour des eaux turbides et des eaux claires est si petite (≤ 0.5 °C) qu'**il est justifié de négliger ce phénomène**, du moins dans ce cas-ci.

6.1.3 *Ajustement du niveau des températures simulées*

En premier lieu, nous nous intéressons à l'identification d'un paramètre dans la simulation permettant l'ajustement du niveau des températures simulées, i.e. la « hauteur » de la courbe simulée. Il est préalablement connu que la turbidité de l'eau possède cette propriété de faire varier (à la baisse) le niveau des températures calculées, or, comme nous l'avons mentionné plus haut, son impact global est très minime. En se basant sur un bref test conduit pour étudier l'impact de la variation de la température du fond T_b (impliquée dans l'équation 1.36), il s'avère que ce paramètre affecte directement le niveau des températures résultant du calcul de H2D2.

Conséquemment, dans l'optique de calibrer le modèle à l'aide de ce paramètre et en se basant sur les mesures fournies par les thermistors, trois tests sont lancés, avec une diffusivité moléculaire à 5.00×10^{-6} m²/s et le paramètre α fixé à 1), sur la période allant du 20 avril 2002 0:00 au 23 avril 2002 0:00. Une longue simulation est préalablement effectuée, du 16 avril 2002 au 21 avril 2002, afin de permettre l'initialisation des données le 20 avril 2002 à 0:00. Voici les trois températures du fond T_b testées:

- simu1 : $T_b = 12$ °C
- simu2 : $T_b = 6$ °C
- simu3 : $T_b = 4$ °C

Sous forme graphique, les résultats sont illustrés aux figures 6.1 à 6.3.

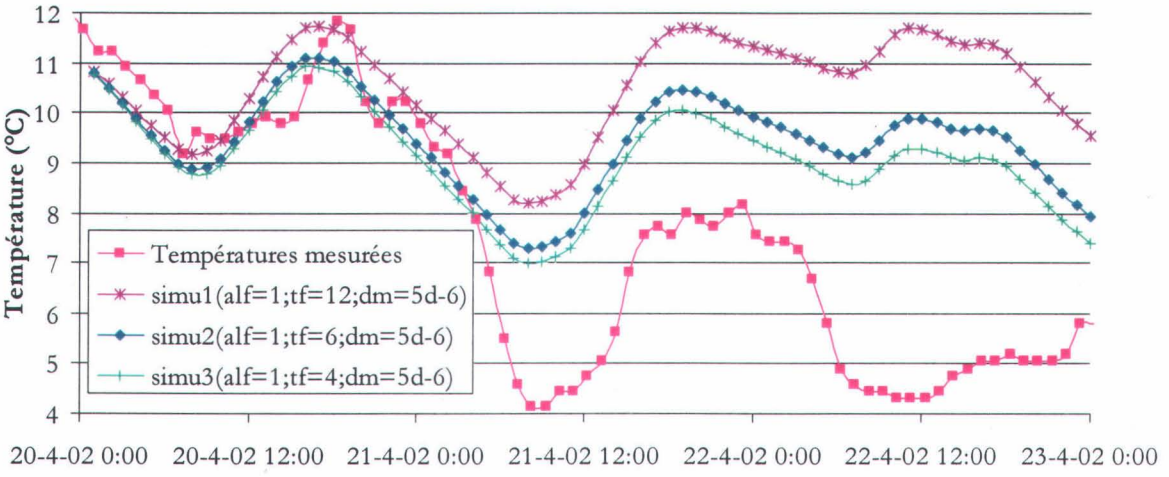


Figure 6.1 – Impact de la variation de la température du fond au thermistor NORD

Comme nous l'avons mentionné plus tôt, il est facile de constater qu'une diminution de la température du fond entraîne une baisse du niveau de la courbe des températures simulées. En ce qui concerne la région du thermistor NORD (figure 6.1), bien qu'au début les trois simulations semblent bien concorder au signal mesuré, un important refroidissement de l'eau est observé dans la nuit du 21 avril : la température mesurée diminue de près de 6 degrés en quelques heures seulement. À partir de ce point, c'est la courbe de la simulation # 3 qui présente la plus faible différence avec le signal mesuré. Le tableau 6.1 présente les valeurs moyennes de différence entre les températures mesurées et simulées en considérant tous les pas des temps compris dans la période de test.

	Différence moyenne entre $T_{simulée}$ et $T_{mesurée}$ (en °C)
simu1(alf=1;tf=12;dm=5x10^{-6})	3.18
simu2(alf=1;tf=6;dm=5x10^{-6})	2.27
simu3(alf=1;tf=4;dm=5x10^{-6})	2.00

Tableau 6.1 – Différences moyennes entre les températures simulées et mesurées pour le thermistor NORD

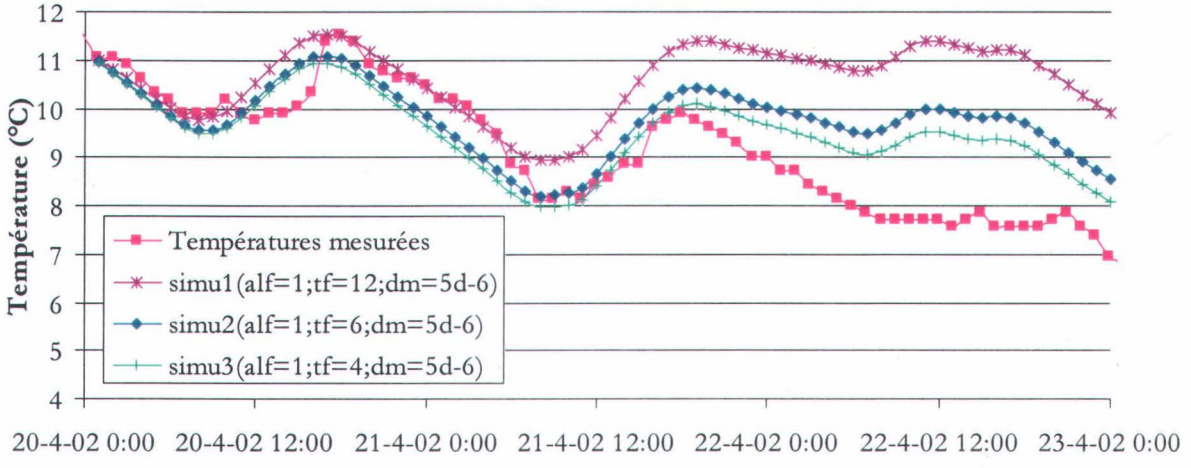


Figure 6.2 – Impact de la variation de la température du fond au thermistor SUD

La figure 6.2 illustre l'influence des différentes températures du fond testées. Jusqu'en fin de journée le 21 avril, les trois courbes simulées correspondent bien aux températures mesurées. Cependant, à partir de ce moment, les trois courbes simulées décrochent du signal du thermistor SUD, ce dernier diminuant lentement sans présenter d'oscillations notables. Comme c'était le cas à la figure 6.1, c'est la courbe de la simulation # 3 qui semble présenter la différence la plus faible avec les données captées par le thermistor SUD. C'est aussi ce que nous indique les données contenues au tableau 6.2.

	Différence moyenne entre $T_{simulée}$ et $T_{mesurée}$ (en °C)
simu1(alf=1;tf=12;dm=5x10^{-6})	1.50
simu2(alf=1;tf=6;dm=5x10^{-6})	0.92
simu3(alf=1;tf=4;dm=5x10^{-6})	0.76

Tableau 6.2 - Différences moyennes entre les températures simulées et mesurées pour le thermistor SUD

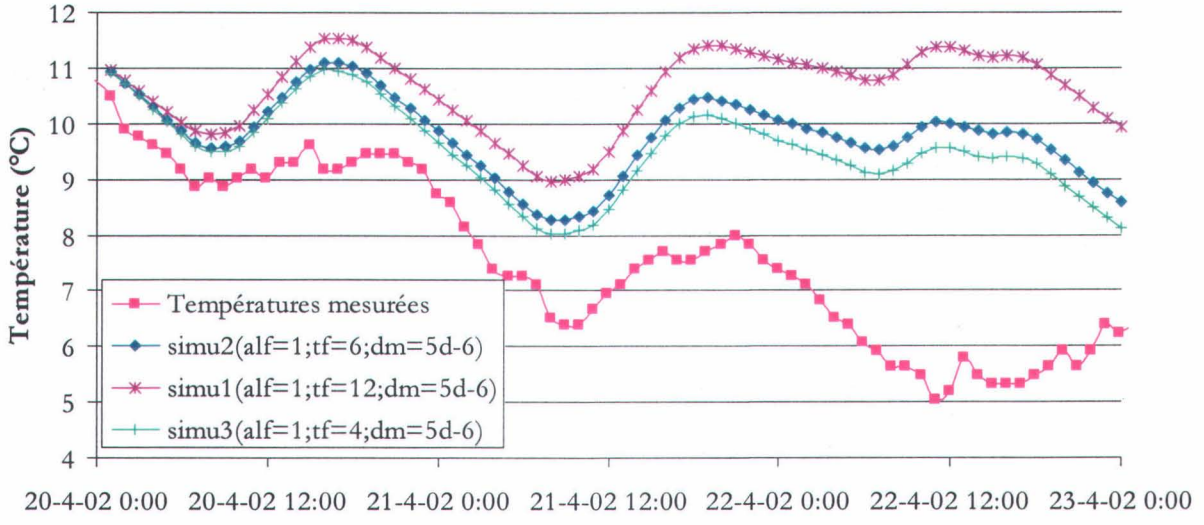


Figure 6.3 – Impact de la variation de la température du fond au thermistor OUEST

À partir de la figure 6.3, nous distinguons que tous les signaux simulés possèdent un niveau au-dessus de la courbe des températures mesurées au thermistor OUEST. Une fois de plus, c'est la courbe de la simulation # 3 qui possède la plus petite différence moyenne, comme nous le confirme le tableau suivant.

	Différence moyenne entre $T_{simulée}$ et $T_{mesurée}$ (en °C)
simu1(alf=1;tf=12;dm=5x10^{-6})	3.07
simu2(alf=1;tf=6;dm=5x10^{-6})	2.25
simu3(alf=1;tf=4;dm=5x10^{-6})	1.98

Tableau 6.3 - Différences moyennes entre les températures simulées et mesurées pour le thermistor OUEST

En somme, **nous pouvons conclure que le niveau le plus adéquat du signal simulé est atteint lorsque la température du fond est fixée à 4 °C**.

Finalement, il reste alors à déterminer quelle est la valeur optimale de α au sein du processus de résolution numérique avec la méthode d'Euler.

6.1.4 *Détermination de la valeur de α*

Trois différents tests sont menés afin de déterminer si une valeur de $\alpha = 0.5$ permet une réponse plus dynamique, concordant davantage avec les résultats mesurés par les thermistors. Les tests se déroulent sur la même période que les simulations 1 à 3 présentées dans la dernière section. La diffusivité moléculaire est fixée à 5.00×10^{-6} m²/s et les températures du fond sont aussi identiques à la section précédente. Seul le paramètre α, cette fois-ci, au lieu d'être égal à 1, vaut 0.5. Ainsi, il est possible d'évaluer, pour chacun des thermistors et pour chacune des températures du fond, la qualité des résultats selon la méthode de résolution numérique. En guise de synthèse, les simulations sont décrites ci-dessous :

- simu4 : $T_b = 12$ °C et $\alpha = 0.5$
- simu5 : $T_b = 6$ °C et $\alpha = 0.5$
- simu6 : $T_b = 4$ °C et $\alpha = 0.5$

Il s'avère, en étudiant les résultats de chacun de ces trois tests aux sites des thermistors et en les comparant aux courbes témoins (simulations 1 à 3), que la valeur du α n'influence pas sur l'aspect dynamique de la réponse obtenue. En effet, les courbes simu1 et simu4 se chevauchent parfaitement, et il en va de même pour les couples de courbes simu2 & simu5 ainsi que simu3 & simu6 (il y a certes un différence minime entre les résultats, souvent de l'ordre du centième de degré). Cependant, à d'autres endroits sur le maillage éléments finis, des différences notables existent entre les valeurs obtenues avec $\alpha = 0.5$ et $\alpha = 1$.

Une comparaison des champs scalaires de température obtenus avec $\alpha = 0.5$ (soit les simulations 4 à 6) permet d'observer qu'en de multiples lieux dans le marais, la solution est dénudée de sens physique, alors que ce n'est pas le cas pour les températures obtenues avec $\alpha = 1$ (soit les simulations 1 à 3). Conséquemment, puisqu'elle génère des solutions qui possèdent un sens physique en tout point du maillage, **il est préférable d'opter pour la méthode de résolution d'Euler implicite (où $\alpha = 1$)**. En effet, l'algorithme de Crank-Nicholson ($\alpha = 0.5$) est trop instable pour être utilisé, en plus de ne revêtir aucun attrait quant à la détection des changements qui auraient été amortis par le caractère diffusif de la méthode d'Euler implicite.

6.2 Simulation réalisée sur toute la période

Maintenant que tous les paramètres optimaux sont établis (**$\alpha = 1$, $T_b = 4$ °C et diffusivité moléculaire = 5.00×10^{-6} m²/s**), en considérant tous les détails mentionnés à la section 4.1, nous pouvons procéder à une simulation sur toute la période (voir le fichier de commandes à l'annexe F) .

6.2.1 Période de simulation

La période de simulation s'étend du 10 avril 2002 0:00 au 6 mai 2002 0:00. Il s'agit d'un total de 624 itérations, réalisées en pas de temps horaire. Selon un document fourni par Philippe Brodeur (fichier **glace.xls**[8]), il y a présence de glace dans le marais au moins jusqu'au 10 avril 2002. Voilà pourquoi la période de simulation utilise cette date de départ du calcul. De plus, au-delà du 6 mai 2002, il n'y a plus de données de niveaux de surface disponibles.

6.2.2 Initialisation de la simulation

La simulation est initialisée le 10 avril 2002 à minuit avec une valeur de 10 °C. Ainsi, pour le premier pas de temps, en chacun des nœuds sur le domaine, la température de l'eau est fixée à cette valeur de 10 °C.

6.2.3 Présentation des signaux simulées et mesurées aux thermistors

- Les figures 6.4 à 6.6 présentent, sur toute la période de simulation, les températures de l'eau obtenues par H2D2 et celles mesurées, à chacune des positions associées aux thermistors. Au sein de chaque graphique, une courbe jaune indique quelle est la profondeur de l'eau au site du thermistor.

- Les figures 6.7 à 6.15 présentent, sur toute la période de simulation, les signaux de températures simulés/mesurés pour chacun des thermistors, en insérant en parallèle les données météorologiques en moyenne horaire. L'objectif étant de pouvoir déduire, en observant ces graphiques, si certaines valeurs adoptées par les paramètres météo peuvent être à l'origine d'une divergence entre les températures obtenues par H2D2 et celles captées par les thermistors.

Le champ scalaire de température pour le dernier pas de temps (6 mai 2002 0 :00) est présenté à l'annexe H.

[8] Disponible sur le CD d'accompagnement.

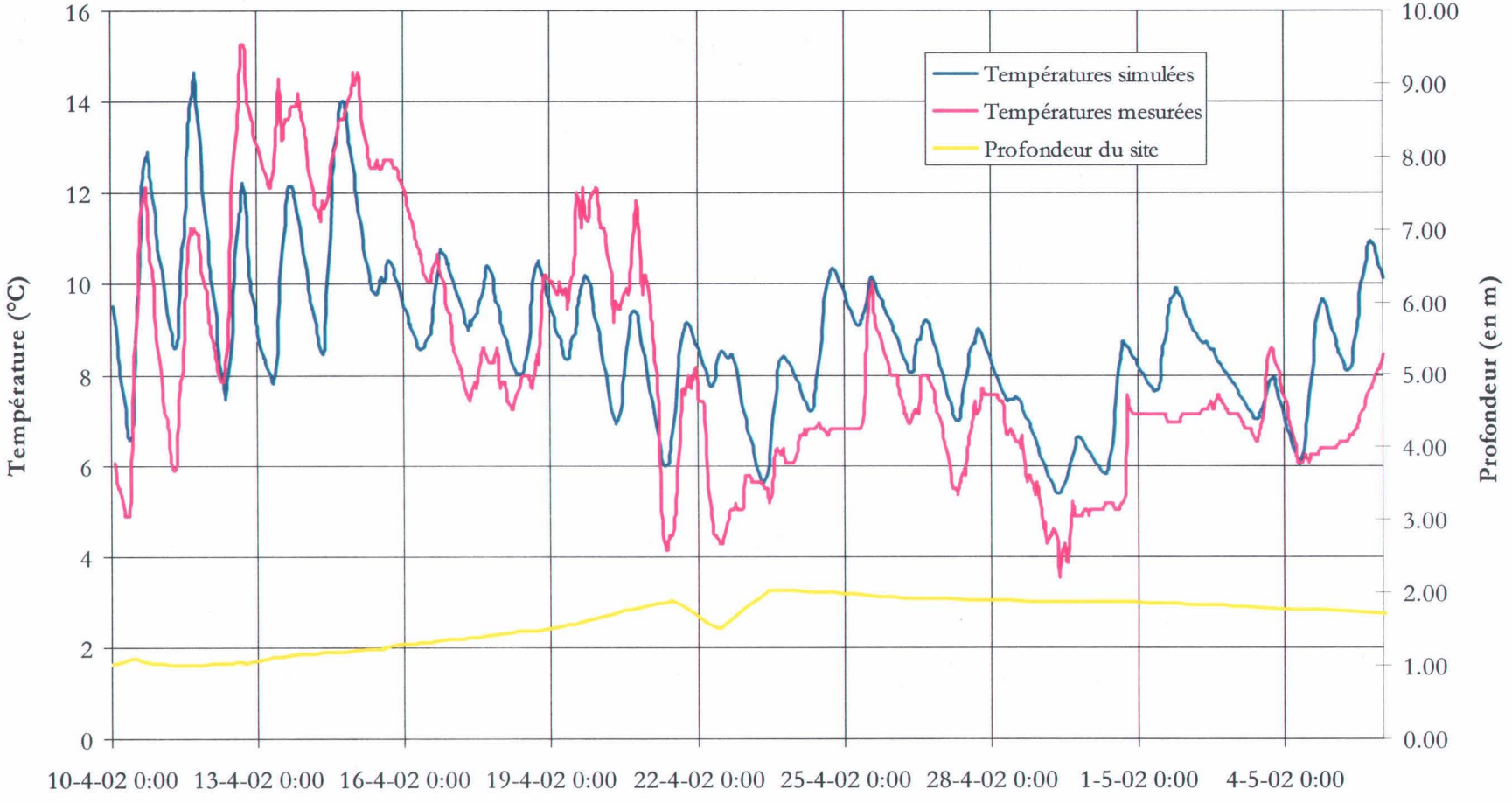


Figure 6.4 – Signal au thermistor NORD en moyenne horaire

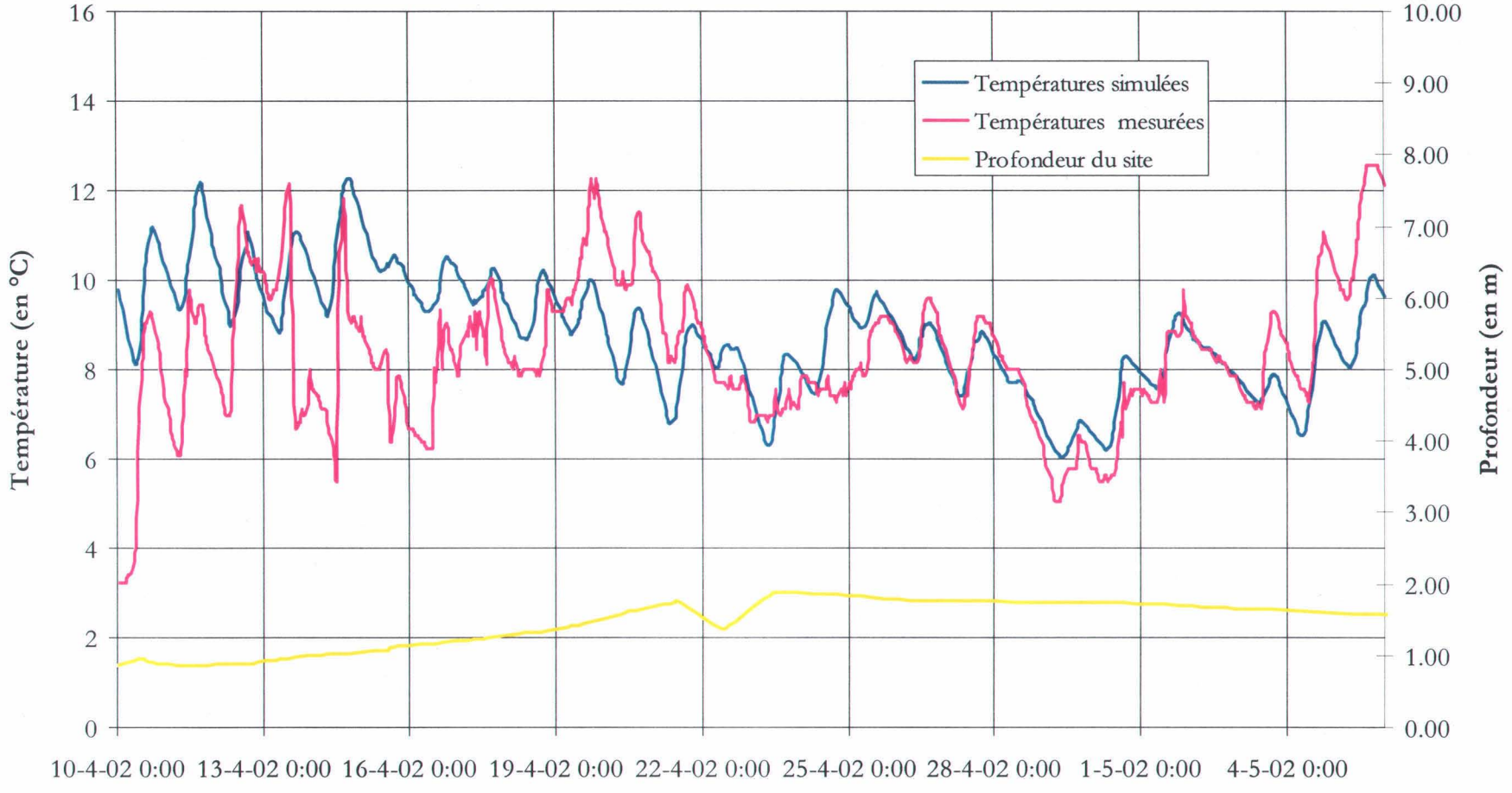


Figure 6.5 – Signal au thermistor SUD en moyenne horaire

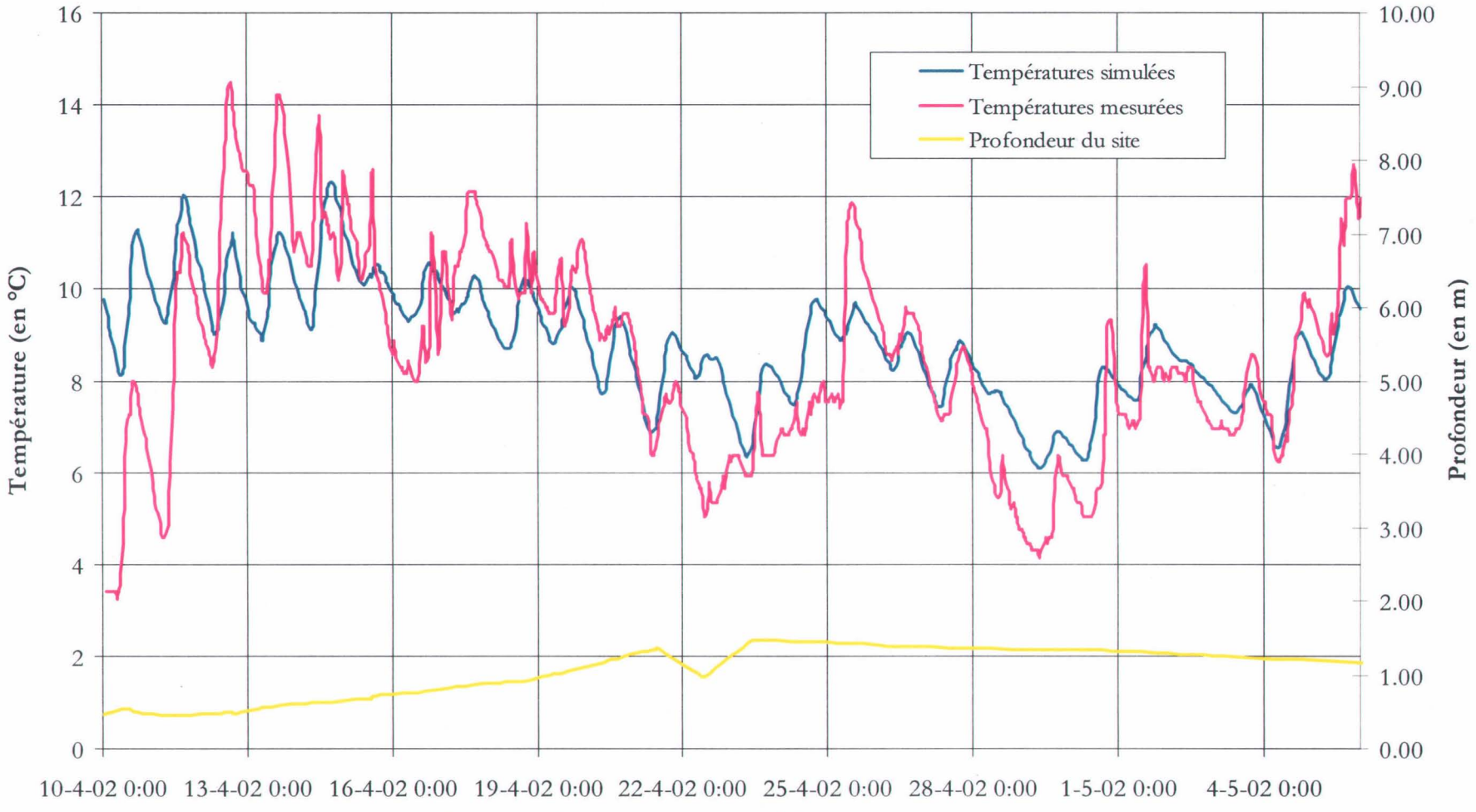


Figure 6.6 – Signal au thermistor OUEST en moyenne horaire

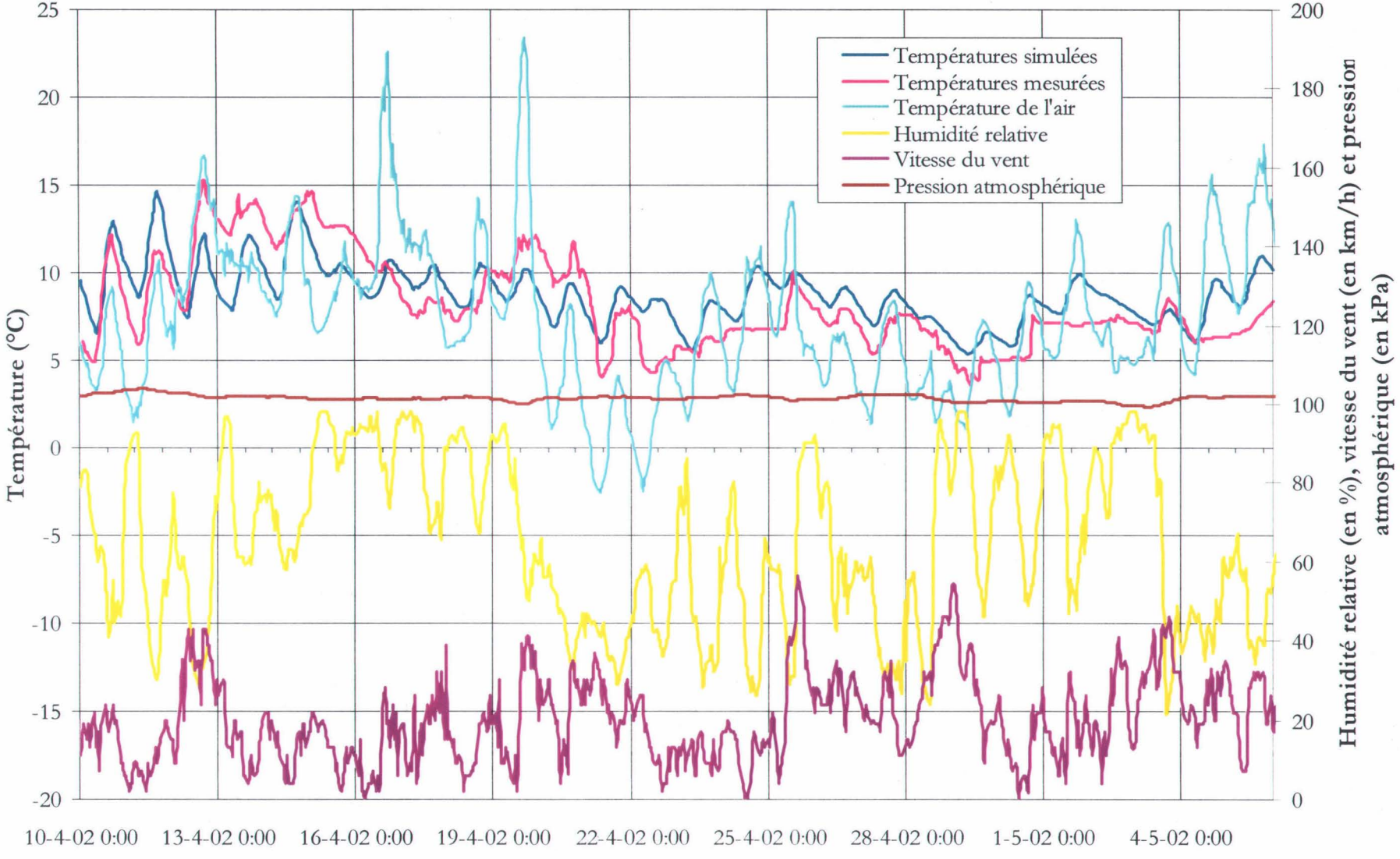


Figure 6.7 – Mise en parallèle de certains paramètres météo en moyenne horaire (T_a, RH, W_2 et P_a) avec les signaux mesuré/simulé au thermistor NORD

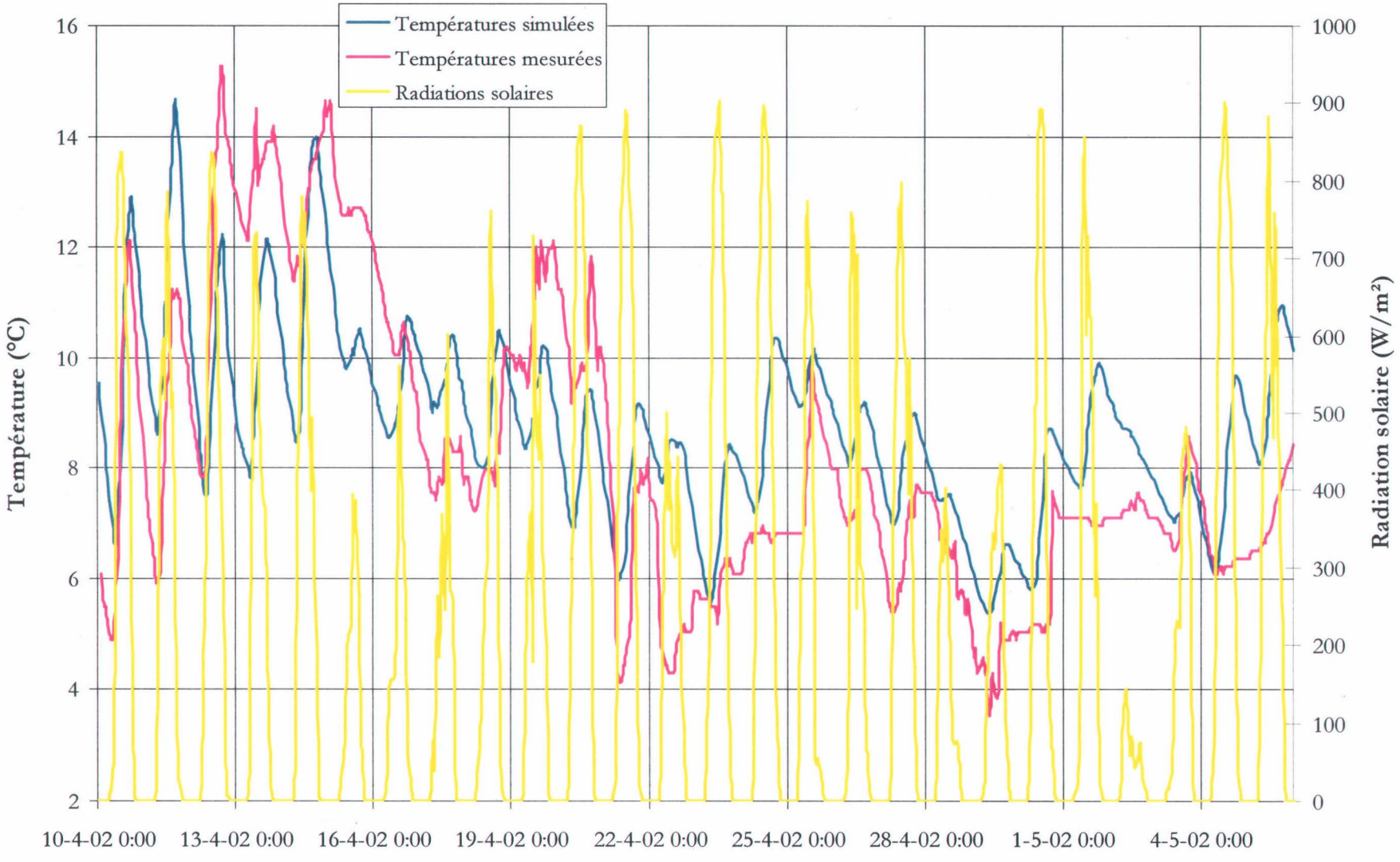


Figure 6.8 – Mise en parallèle de la radiation solaire horaire avec les signaux mesuré/simulé au thermistor NORD

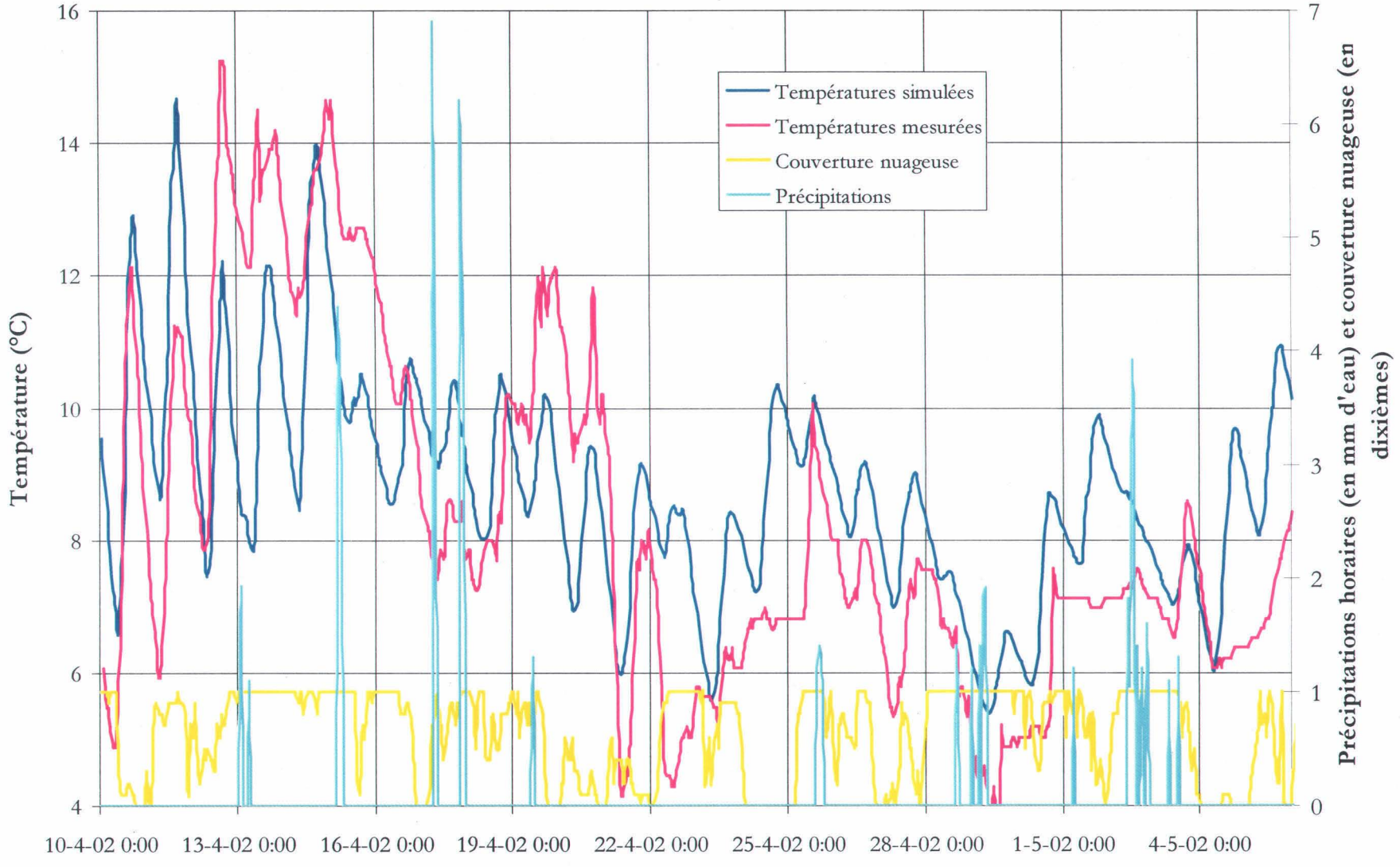


Figure 6.9 – Mise en parallèle des précipitations horaires et de couverture nuageuse horaire avec les signaux mesuré/simulé au thermistor NORD

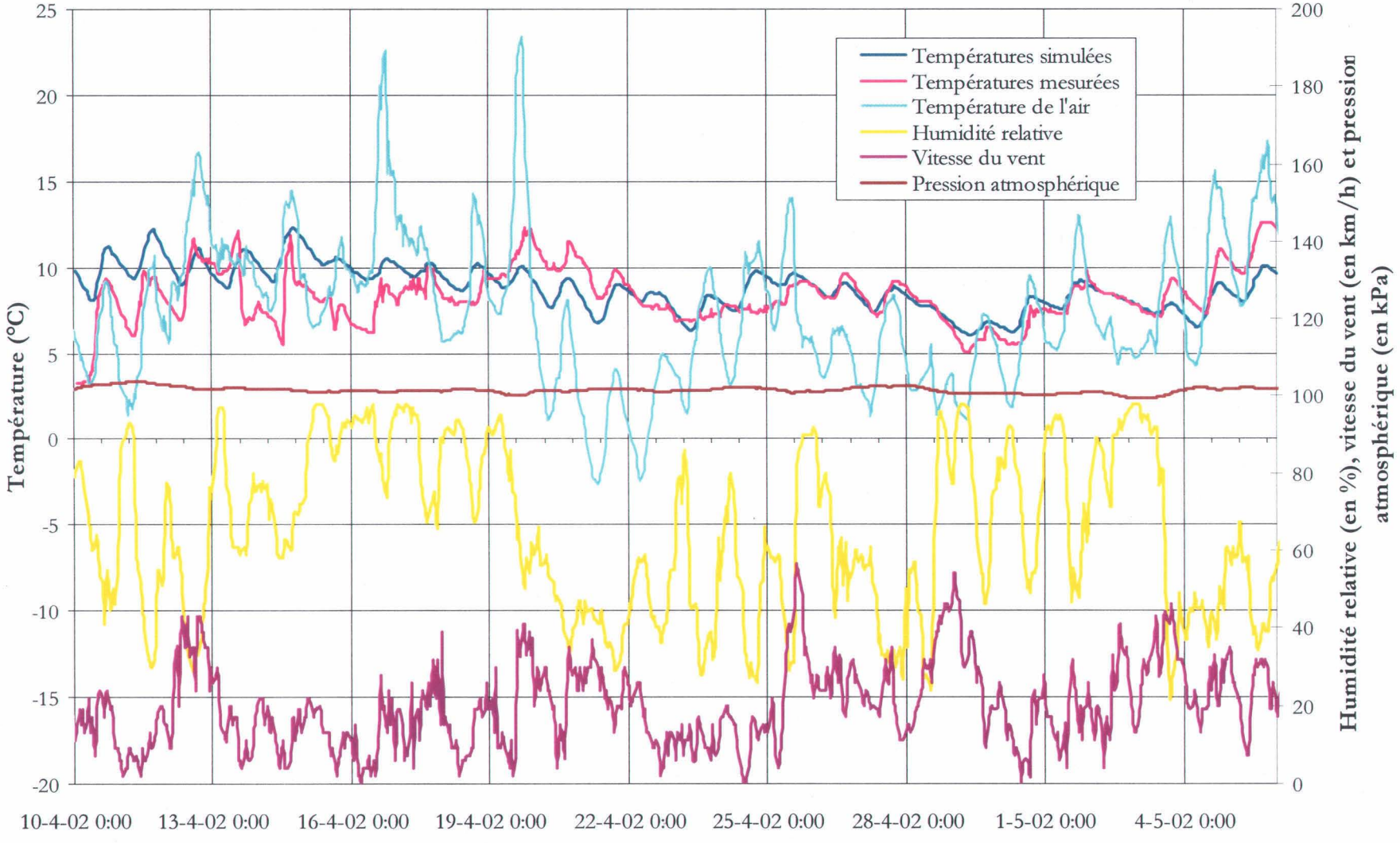


Figure 6.10 - Mise en parallèle de certains paramètres météo en moyenne horaire (T_a, RH, W_2 et P_a) avec les signaux mesuré/simulé au thermistor SUD

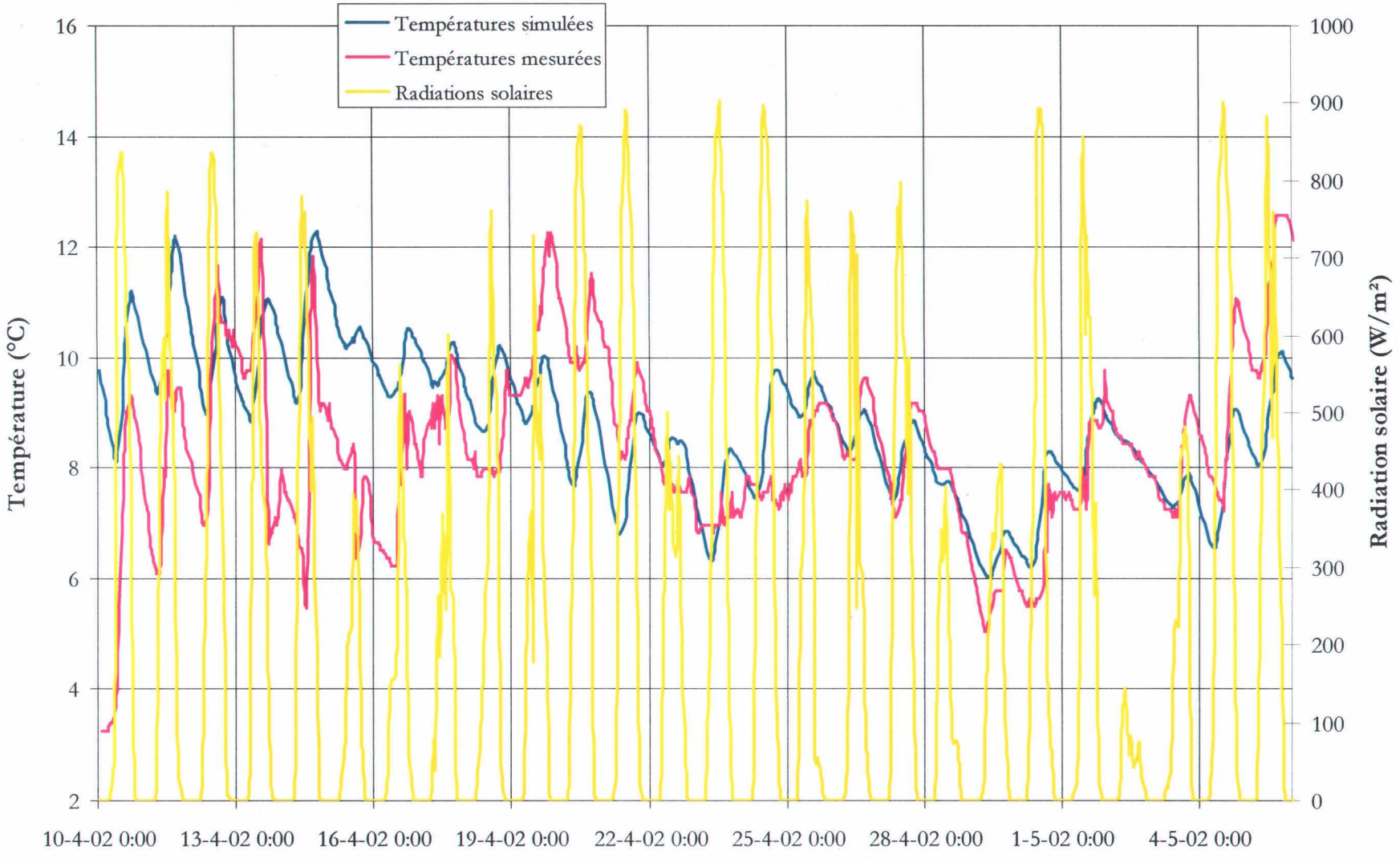


Figure 6.11 – Mise en parallèle de la radiation solaire horaire avec les signaux mesuré/simulé au thermistor SUD

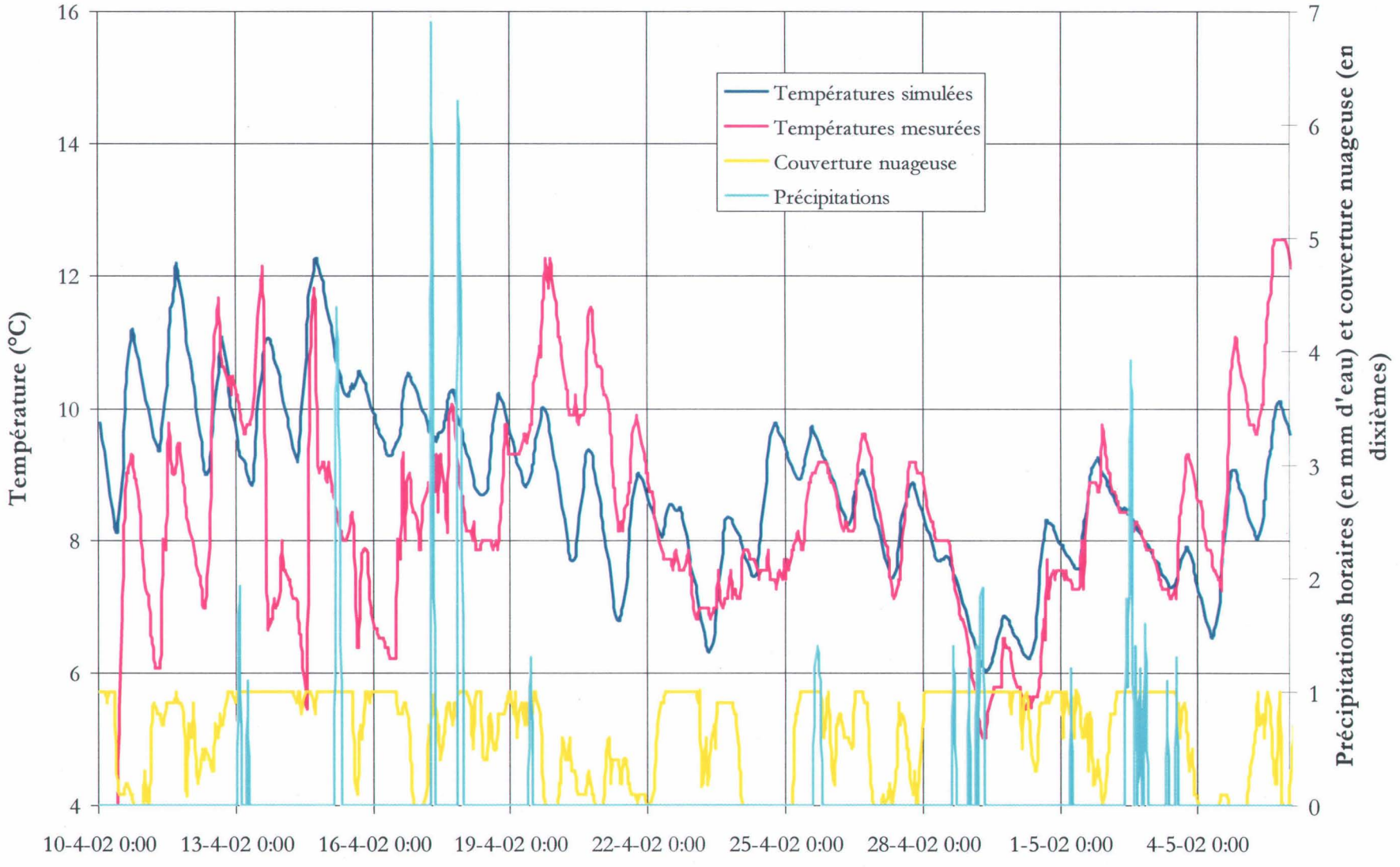


Figure 6.12 – Mise en parallèle des précipitations horaires et de couverture nuageuse horaire avec les signaux mesuré/simulé au thermistor SUD

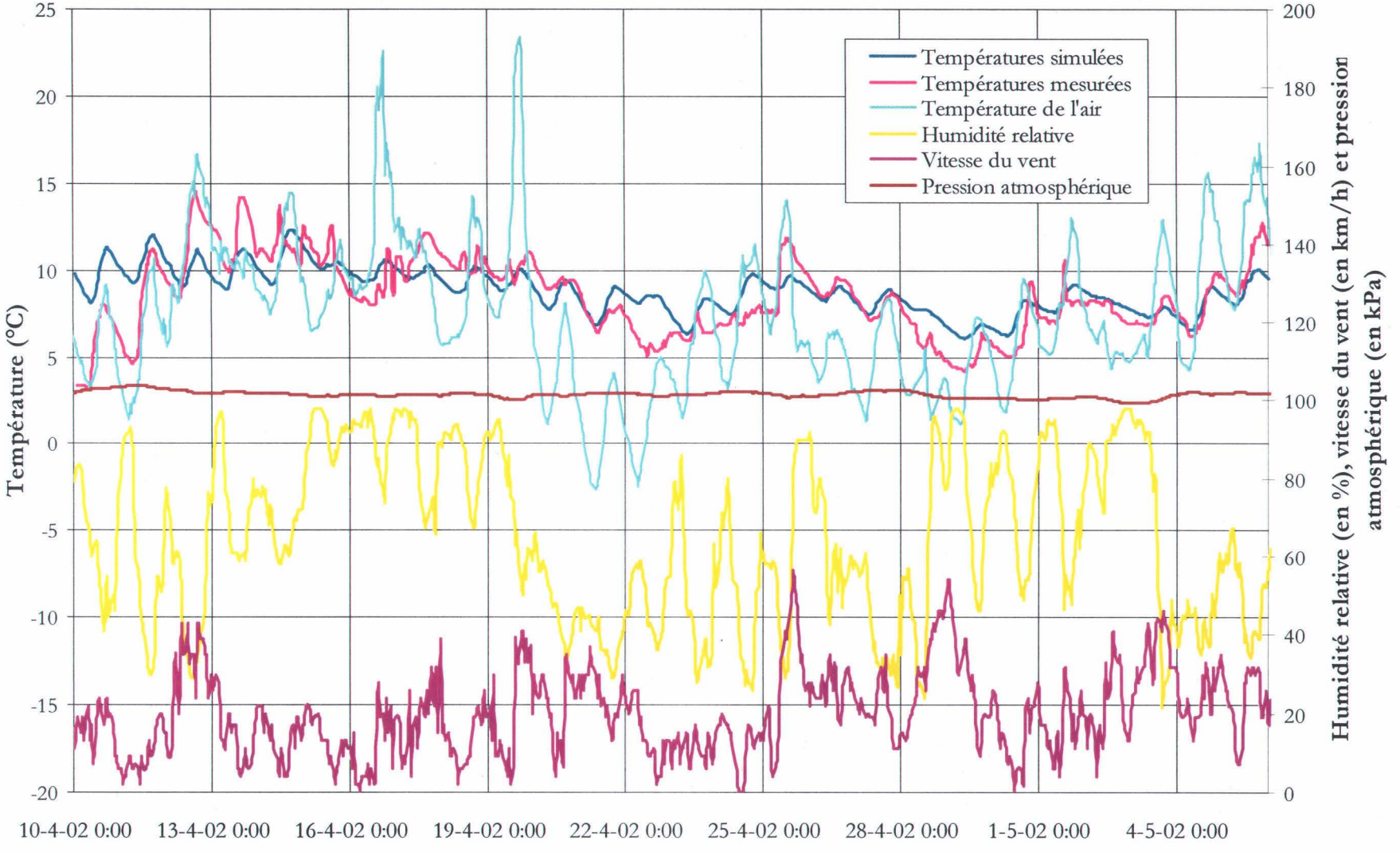


Figure 6.13 - Mise en parallèle de certains paramètres météo en moyenne horaire (T_a, RH, W_2 et P_a) avec le signal mesuré/simulé au thermistor OUEST

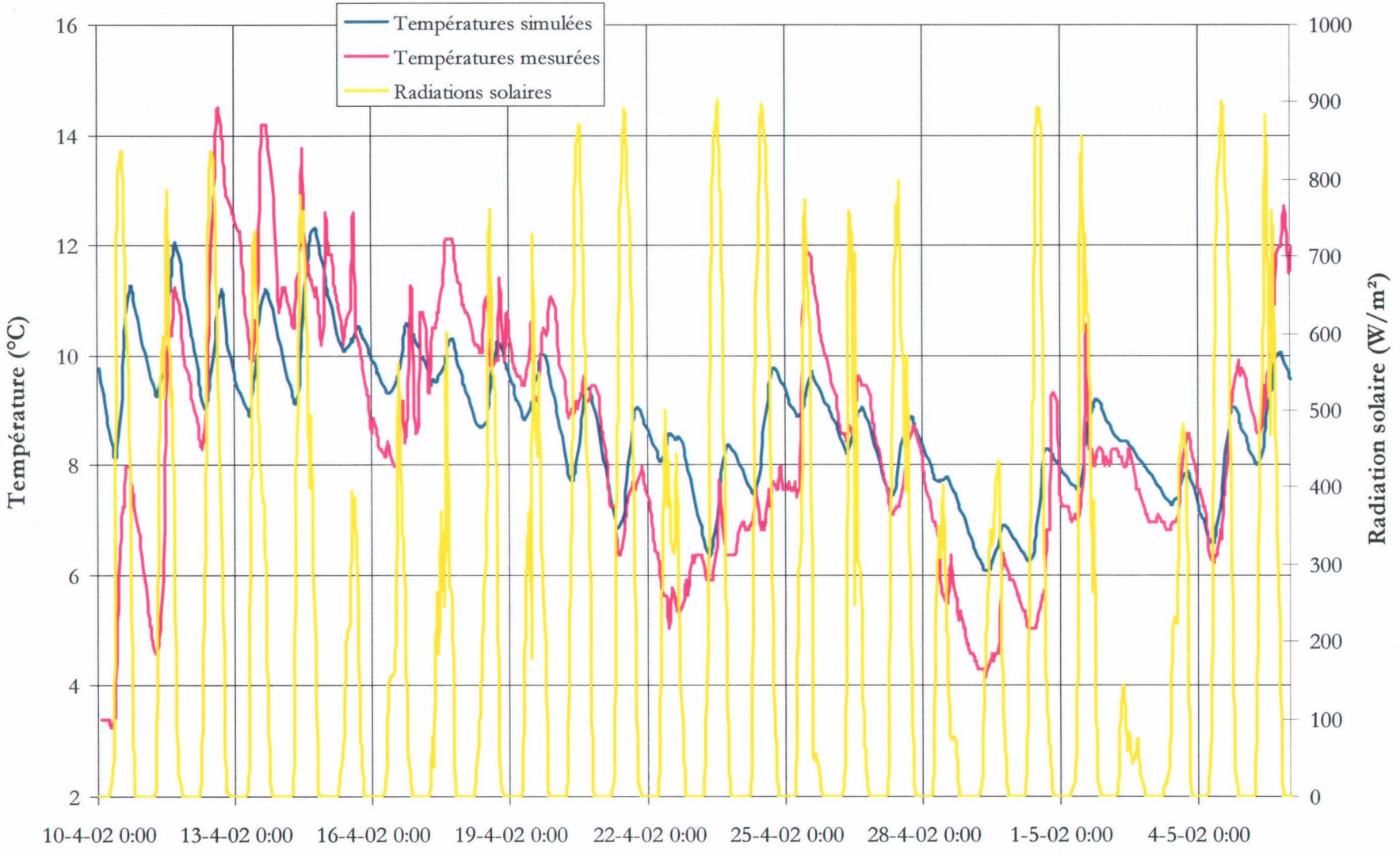


Figure 6.14 - Mise en parallèle de la radiation solaire horaire avec les signaux mesuré/simulé au thermistor OUEST

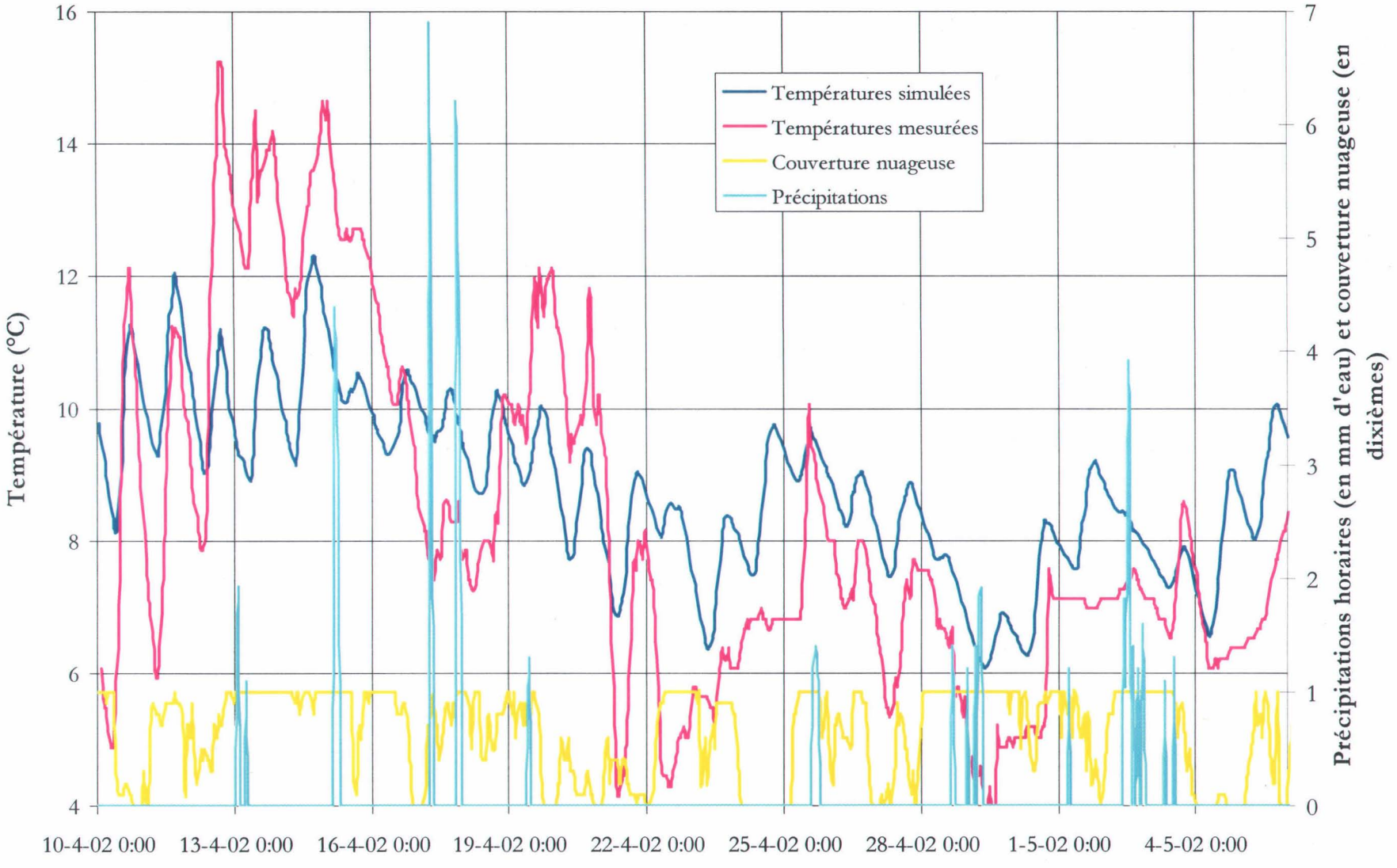


Figure 6.15 - Mise en parallèle des précipitations horaires et de couverture nuageuse horaire avec les signaux mesuré/simulé au thermistor OUEST

6.2.4 *Analyse statistique des extremums journaliers*

Nous analysons ensuite de manière statistique le signal en nous basant sur les maximums et minimums journaliers. On porte en abscisse la mesure du thermistor et en ordonnée la simulation avec un point pour chaque jour de simulation : figures 6.16, 6.18 et 6.20 pour les maximums et figures 6.17, 6.19 et 6.21 pour les minimums.

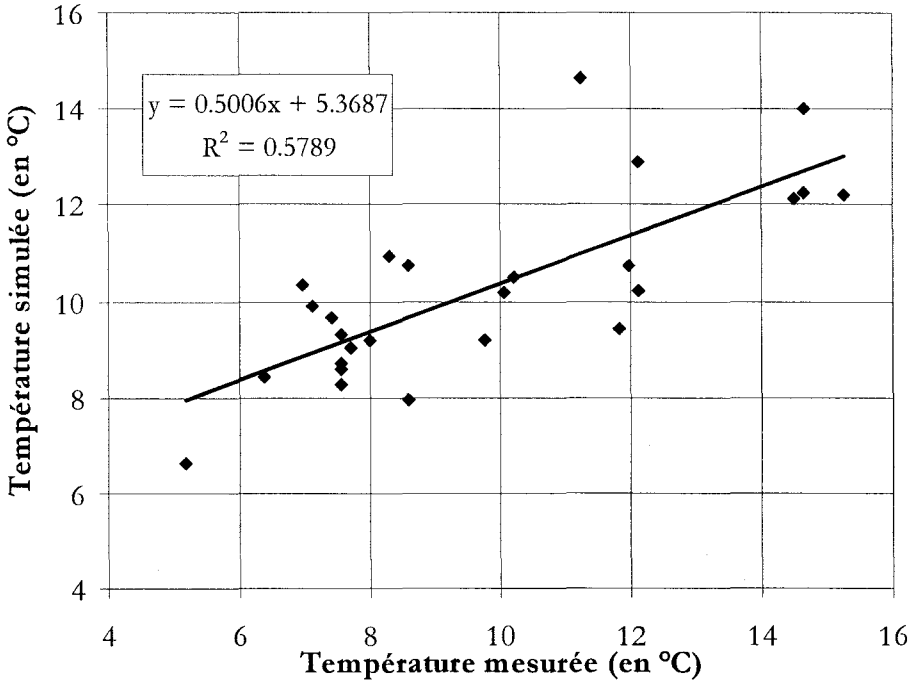


Figure 6.16 – Comparaison mesure/simulation – maximums de températures au thermistor NORD

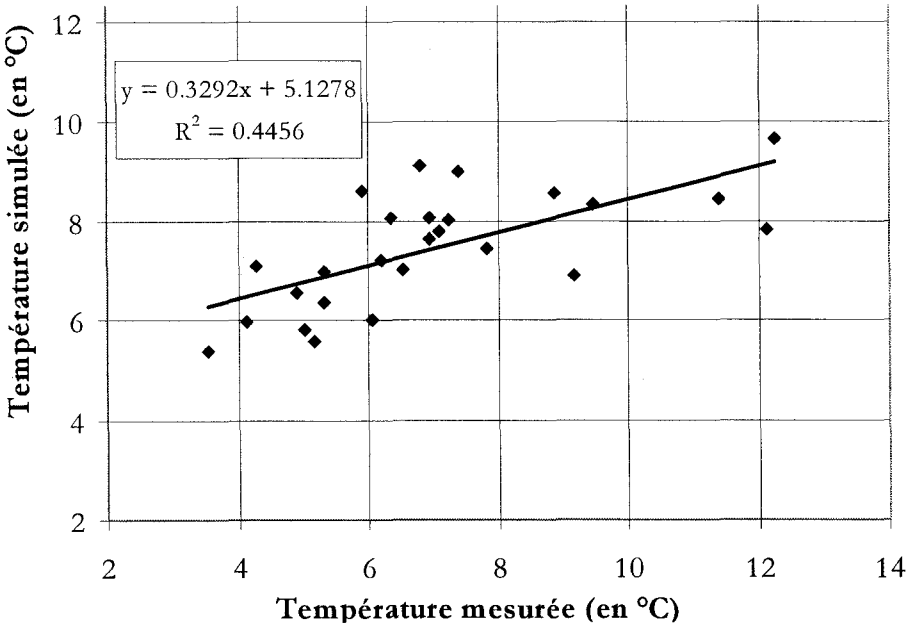


Figure 6.17 – Comparaison mesure/simulation – minimums de températures au thermistor NORD

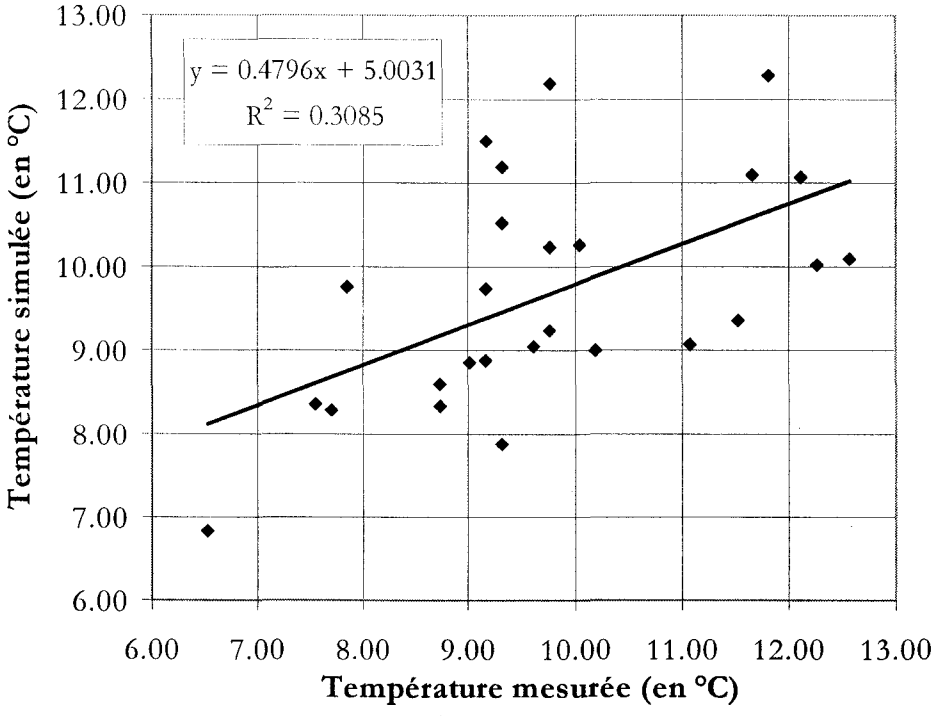


Figure 6.18 – Comparaison mesure/simulation – maximums de températures au thermistor SUD

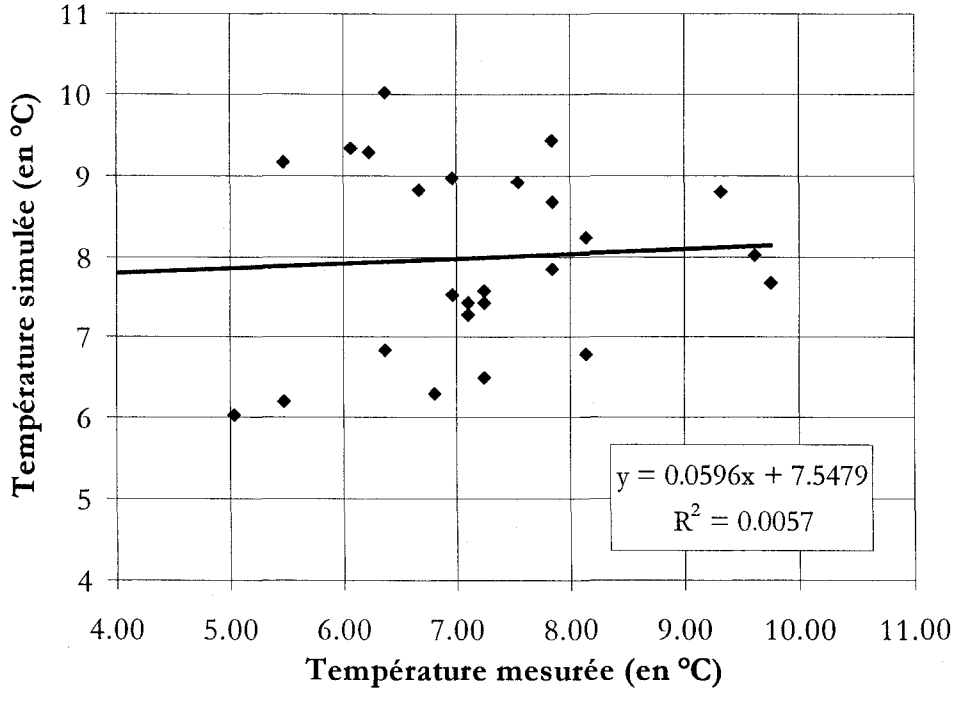


Figure 6.19 – Comparaison mesure/simulation – minimums de températures au thermistor SUD

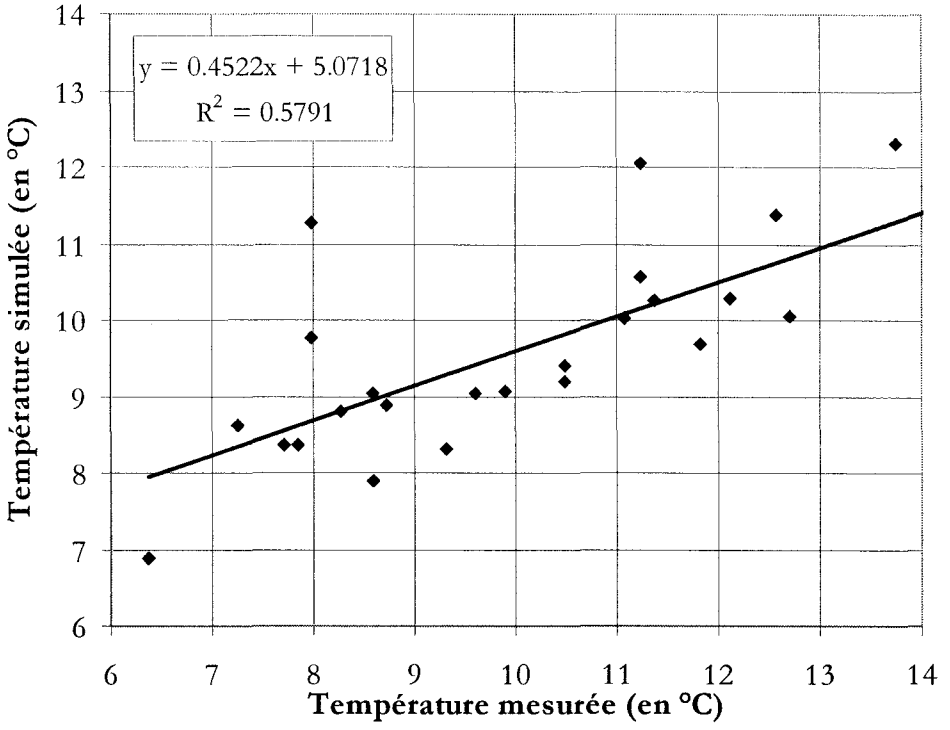


Figure 6.20 - Comparaison mesure/simulation – maximums de températures au thermistor OUEST

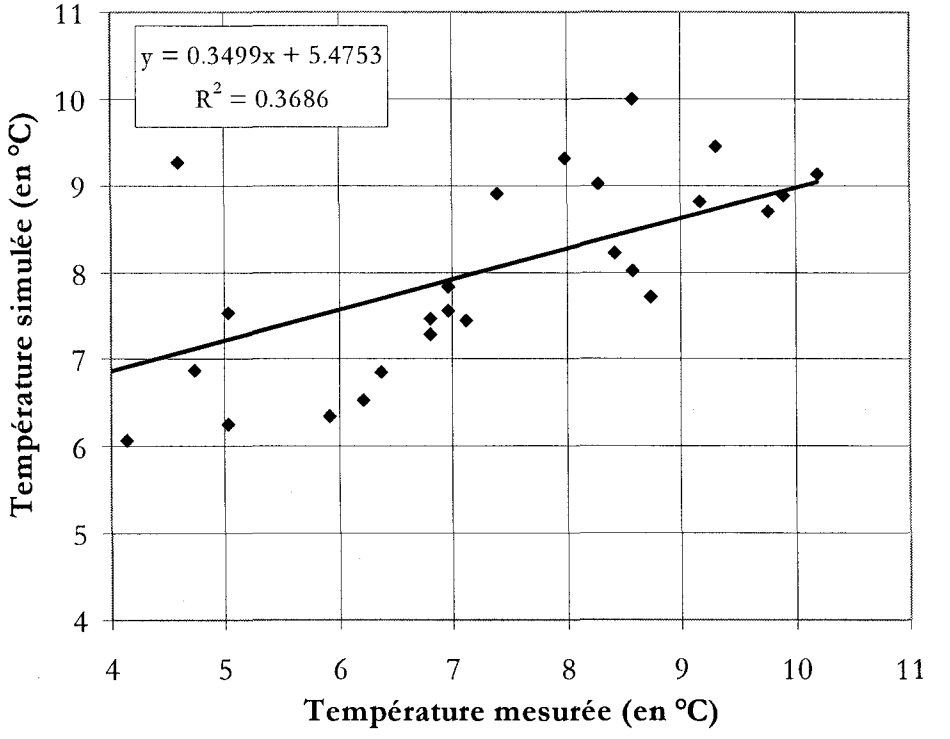


Figure 6.21 – Comparaison mesure/simulation – minimums de températures au thermistor OUEST

La corrélation entre mesures et simulations est généralement faible (tableau 6.4), voire modérée par endroits. Globalement, la corrélation est meilleure pour les maximums que pour les minimums. C'est pour le thermistor NORD qu'elle est la plus forte, alors que la plus faible corrélation observée concerne le thermistor SUD. On note de même que l'ordonnée à l'origine (tableau 6.5) est toujours positive, ce qui montre que la simulation a en moyenne des valeurs plus élevées que la mesure. Ce dernier phénomène est principalement dû au fait que les chutes abruptes de températures mesurées par les thermistors sont plutôt mal simulées dans la présente étude.

Thermistor	minimums	maximums
NORD	0.4456	0.5789
SUD	0.0057	0.3085
OUEST	0.3686	0.5791

Tableau 6.4 – Coefficient de détermination de la droite de régression linéaire

Thermistor	minimums	maximums
NORD	5.1278	5.3687
SUD	7.5479	5.0031
OUEST	5.4753	5.0718

Tableau 6.5 – Ordonnée à l'origine de la droite de régression linéaire

6.2.5 *Analyse statistique du déphasage*

Pour chaque jour de simulation, la différence entre le moment où la température maximale est observée pour les valeurs mesurées et simulées est obtenue. Une moyenne de ces différences est ensuite calculée. Le tableau 6.6 contient donc les différences moyennes entre les moments des maximums pour les températures simulées et obtenues avec H2D2.

	Différence moyenne (en h)	Données aberrantes	Différence moyenne sans les données aberrantes (en h)
Thermistor NORD	4:23	aucune	-
Thermistor SUD	2:00	17:04	1:24
Thermistor OUEST	3:46	aucune	-

Tableau 6.6 – Différence moyenne entre les moments des maximums mesurés/simulés

C'est pour le thermistor SUD que le déphasage est le plus petit entre les signaux simulés/mesurés. En y enlevant la donnée aberrante présente, on obtient une différence moyenne de seulement 1:24. D'autre part, le déphasage est plus important pour les deux autres thermistors, plus particulièrement pour le NORD.

Similairement, pour chaque jour de simulation, la différence entre le moment où la température minimale est observée pour les valeurs mesurées et simulées est obtenue. Une moyenne de ces

différences est ensuite calculée. Le tableau 6.7 contient donc les différences moyennes entre les moments des minimums pour les températures simulées et obtenues avec H2D2.

	Différence moyenne (en h)	Données aberrantes	Différence moyenne sans les données aberrantes (en h)
Thermistor NORD	2:29	15:47 et 11:42	1:33
Thermistor SUD	2:40	aucune	-
Thermistor OUEST	2 :44	aucune	-

Tableau 6.7 – Différence moyenne entre les moments des minimums mesurés/simulés

On constate que le déphasage est, globalement, inférieur pour les moments des minimums que pour ceux des maximums. Une fois lesté de ses deux données aberrantes, la différence moyenne pour le thermistor NORD est plus particulièrement faible.

6.2.6 *Analyse statistique d'amplitude*

Nous pouvons ensuite analyser de manière statistique le signal en nous basant sur les amplitudes journalières (différence entre les maximums et minimums journaliers). On porte donc en abscisse l'amplitude mesurée et en ordonnée la simulation avec un point pour chaque jour de simulation : voir les figures 6.22 à 6.24.

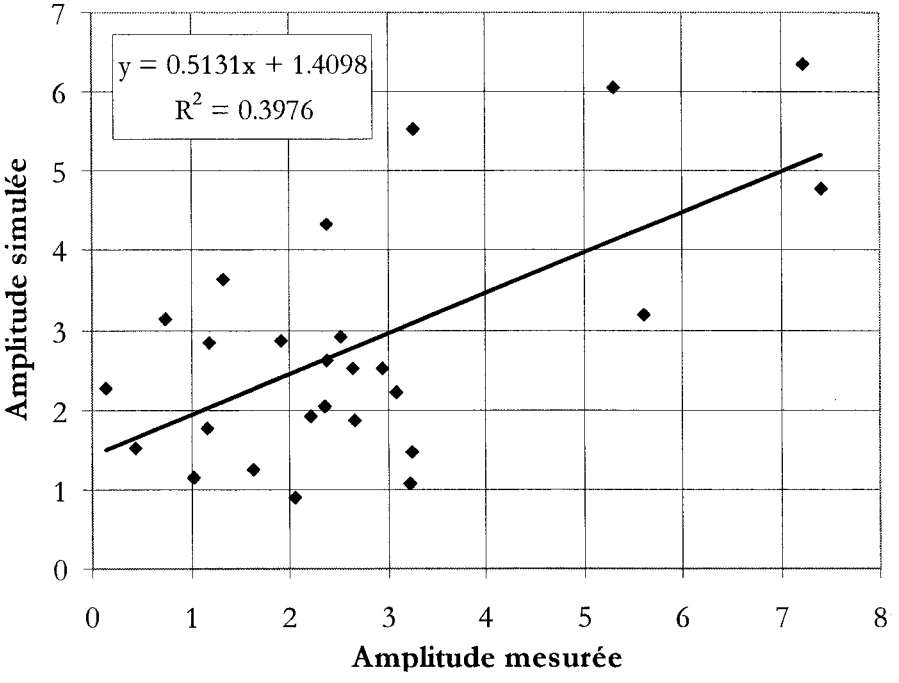


Figure 6.22 – Comparaison mesuré/simulé – amplitudes au thermistor NORD

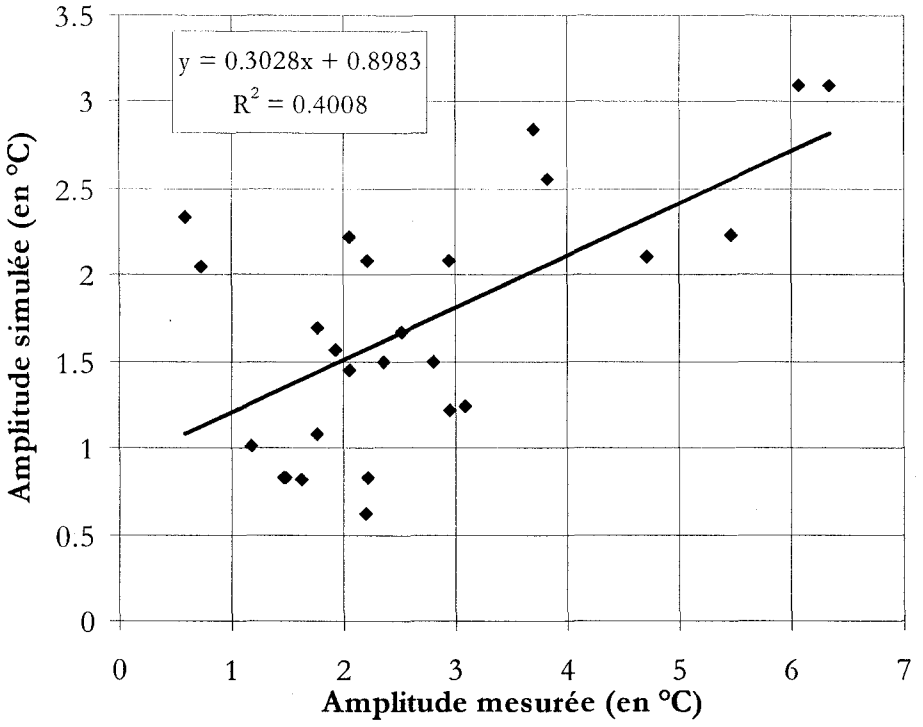


Figure 6.23 - Comparaison mesuré/simulé – amplitudes au thermistor SUD

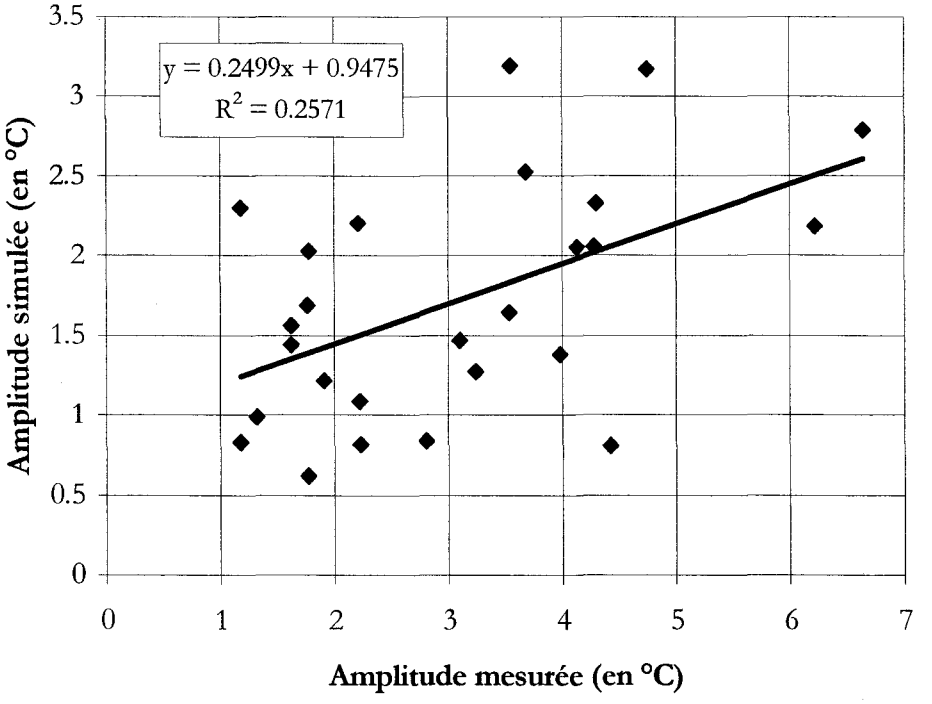


Figure 6.24 – Comparaison mesuré/simulé – amplitudes au thermistor OUEST

La corrélation entre mesures et simulations est modérément faible (tableau 6.4). Elle est sensiblement la même pour les thermistors NORD et SUD, alors que pour le thermistor OUEST

elle est inférieure (tableau 6.5). On note de même que l'ordonnée à l'origine est toujours positive, ce qui montre qu'en moyenne, la simulation a des valeurs plus élevées d'amplitude que la mesure.

Thermistor	R^2
NORD	0.3976
SUD	0.4008
OUEST	0.2571

Tableau 6.8 – Coefficient de détermination de la droite de régression linéaire

Thermistor	Ordonnée à l'origine
NORD	1.4098
SUD	0.8983
OUEST	0.9475

Tableau 6.9 – Ordonnée à l'origine de la droite de régression linéaire

6.2.7 Causes potentielles d'erreur

De multiples phénomènes peuvent expliquer la différence entre les résultats obtenus par le logiciel H2D2 et par les thermistors.

- *L'apport d'eau froide via la structure de contrôle ouverte lors de la période de simulation*, plus particulièrement dans la région du thermistor SUD, étant donné que les autres thermistors sont situés à bonne distance de cette zone d'entrée d'eau.
- *Trop de distance entre les stations météorologiques et le marais de Saint-Barthélemy*. Il est évident que les phénomènes très locaux (tels orages ou rafales de vents) sont souvent non documentés. Aussi, il est fort possible qu'un déphasage existe pour ce qui est des données sur la couverture nuageuse, puisque la station source des données (Dorval) est tout de même située à près d'une centaine de kilomètres de la zone d'intérêt.
- *Présence de glace lors du mois d'avril*. Il est connu qu'il y a de la glace dans le marais au moins jusqu'au 10 avril 2002. Bien entendu, peut-être s'agit-il de glace résiduelle occupant les endroits peu profonds et que les thermistors (situés en eau plus profonde) ne sont pas affectés. Voilà pourquoi il est difficile de quantifier ce phénomène.
- *Fonte de la neige*, donc du même coup apport d'eau froide au modèle, ce qui aurait contribué à diminuer les températures enregistrées par les thermistors.
- *Présence de végétation*, qui aurait pour effet de diminuer la température de l'eau dans les zones touchées, soit par l'ombrage produit (arbres,…) ou encore par l'énergie solaire captée (algues,…).

Figure 6.25 – Présence de végétation au nord du canal principal extrême OUEST (l'horizon pointe vers le sud) – photo prise le 10 juin 2004

- *Stratification de la température de l'eau.* Le marais Saint-Barthélemy, outre sous l'action de brassage exercée par le vent, ne connaît aucune activité hydrodynamique. Par conséquent, l'hypothèse posée dans H2D2 selon laquelle la température est une constante sur toute la colonne d'eau se doit d'être reconsidérée, étant donné qu'il ne s'agit pas d'une rivière (où le brassage des eaux est fortement lié à l'écoulement) mais bien d'un marais. Selon une étude effectuée à l'Île du Moine en 2003[9], la stratification est plutôt négligeable au début du printemps (avril) et au début de l'automne (octobre), alors qu'elle est très présente au cœur de l'été (juillet).

En somme, nous ne croyons pas que la différence entre les résultats simulés/mesurés soit liée à des mauvais paramètres de simulation, par exemple une valeur de α inadéquate ou encore des mauvaises conditions limites. La qualité des températures simulées, qui peut être qualifiée de modérée (voire faible), indique tout même l'existence d'une certaine corrélation avec les données mesurées, tant au déphasage qu'à l'amplitude ainsi qu'aux valeurs prises par les extremums journaliers. Il y a de fortes chances qu'une combinaison des facteurs potentiels d'erreur énumérés ci-haut ait eu un impact sur la réponse de la simulation face aux résultats mesurés en pratique.

[9] Voir documents disponibles sur le CD d'accompagnement.

CONCLUSION

Afin de pouvoir valider le modèle de température utilisé par le logiciel de calcul H2D2, nous avons procédé à plusieurs simulations sur un maillage éléments finis adapté au segment 4 du marais aménagé de Saint-Barthélemy. Ce dernier étant bien documenté au plan des données environnementales, il a par la suite été possible de mettre en parallèle les températures simulées et mesurées par les thermistors présent sur le site lors de la période allant du 10 avril 2002 au 6 mai 2002.

Il a été conclu qu'une diffusivité moléculaire de l'eau fixée à 5.00×10^{-6} m²/s offrait des résultats cohérents en tout point du maillage, contrairement à des valeurs plus élevées. De plus, l'hypothèse de négliger le phénomène de la turbidité élevée de l'eau s'est avérée exacte. Il a été de plus démontré qu'une variation du paramètre *b(t)* entraînait une modification du niveau de la courbe des températures simulées. Aussi, il a été conclu que la méthode de résolution numérique la plus adéquate pour cette étude était la méthode d'Euler implicite ($\alpha = 1$), au détriment de l'algorithme de Crank-Nicholson ($\alpha = 0.5$).

Devant des résultats démontrant une corrélation modérée (voire faible) entre les températures simulées et mesurées, de même que devant une description peu approfondie des analyses statistiques et des graphiques présentés (en raison d'un manque de temps), force nous est de constater qu'il y a certainement davantage de conclusions à tirer quant à l'amélioration de H2D2 que ce qui est présenté dans ce rapport. À moins qu'il faille tout simplement en conclure que les différences entre mesures et simulations sont entièrement explicables par des raisons qui sont situées hors de la zone de contrôle du responsable de la simulation (section 6.2.7).

RECOMMANDATIONS

Malheureusement, pour des raisons de manque de temps, l'analyse des résultats a été effectuée de façon précipitée. Certes, on ne peut cependant reprocher un manque d'informations présentés; nombreux sont les graphiques comparatifs et les analyses statistiques. Cependant, il aurait été préférable de scruter davantage les données d'analyse pour pouvoir en tirer des conclusions qui auraient participé à l'amélioration du modèle de température proposé par H2D2. Il serait souhaitable d'étudier plus particulièrement les figures 6.7 à 6.15, où les données météorologiques sont mises en parallèle avec les signaux mesurés/simulés des thermistors, afin d'étudier l'impact des paramètres *a(t)* et *b(t)* (voir équations 1.33 à 1.38) sur la réponse obtenue. Comme il a été préalablement déterminé, le paramètre *b(t)* joue sur le niveau de la courbe des températures simulées. Or, il faudrait déterminer l'impact du paramètre *a(t)*. En se fiant à sa position dans l'équation 2.6, il y a de fortes chances qu'il agisse sur l'amplitude du signal; cependant, cela reste à confirmer.

Aussi, il aurait été pertinent de réaliser davantage de simulations afin de tester l'impact de différentes combinaisons de paramètres.

BIBLIOGRAPHIE

Dhatt, G. et G. Touzot 1981. *Une présentation de la méthode des éléments finis*. Les Presses de l'Université Laval & Maloine S.A. Éditeur. Québec et Paris. 543 p.

Greenberg, M. D. 1998. *Advanced Engineering Mathematics – Second Edition*. Éditions Prentice Hall. Upper Saddle River. Pages 19-23.

Heniche M., Y. Secretan, J. Morin, J.-F. Cantin et M. Leclerc 2002. *Two-Dimensional Depth Averaged Fluvial Thermal Regime Prediction*. Manuscrit à soumettre au Journal of Hydraulic Engineering.

Lilga M.C. 2003, http://www.agri.idaho.gov/gw/conference/Lilga & Stevens Abstract.pdf.

Morin, J., Y. Secretan, O. Champoux et A. Armellin 2002. *Modélisation 2D de la température de l'eau sur la batture Thaillandier*. Rapport technique RT-117. Service Météorologique du Canada, Environnement Canada, Sainte-Foy, 71 p.

Sergent, C. septembre1998. *Les métamorphoses de la neige*. Revue Neige et Avalanches. N°83. Association Nationale pour l'Étude de la Neige et des Avalanches. France.

ANNEXE A – REQUÊTE DE DONNÉES MÉTÉOROLOGIQUES

Voici les détails de la requête de données météorologiques effectuée au Service de Climatologie du Canada (SMC). Il s'agit dans tous les cas de données horaires. En italique est indiqué le nom de code du paramètre, tel que donné par le SMC.

- Radiation solaire (en kJ/m²) : paramètre 061 – *rayonnement solaire global*
- Vitesse du vent (en km/h) : paramètre 076
- Pression atmosphérique (en millibar) : paramètre 077 – *pression à la station*
- Température de l'air (en °C) : paramètre 078 – *température du thermomètre sec*
- Humidité relative (en %) : paramètre 080
- Couverture nuageuse (en dixièmes) : paramètre 082 – *étendue totale des nuages*
- Précipitations horaires totales (pluie + neige, en mm d'eau) : paramètre 123

ANNEXE B – MAILLAGE ÉLÉMENTS FINIS

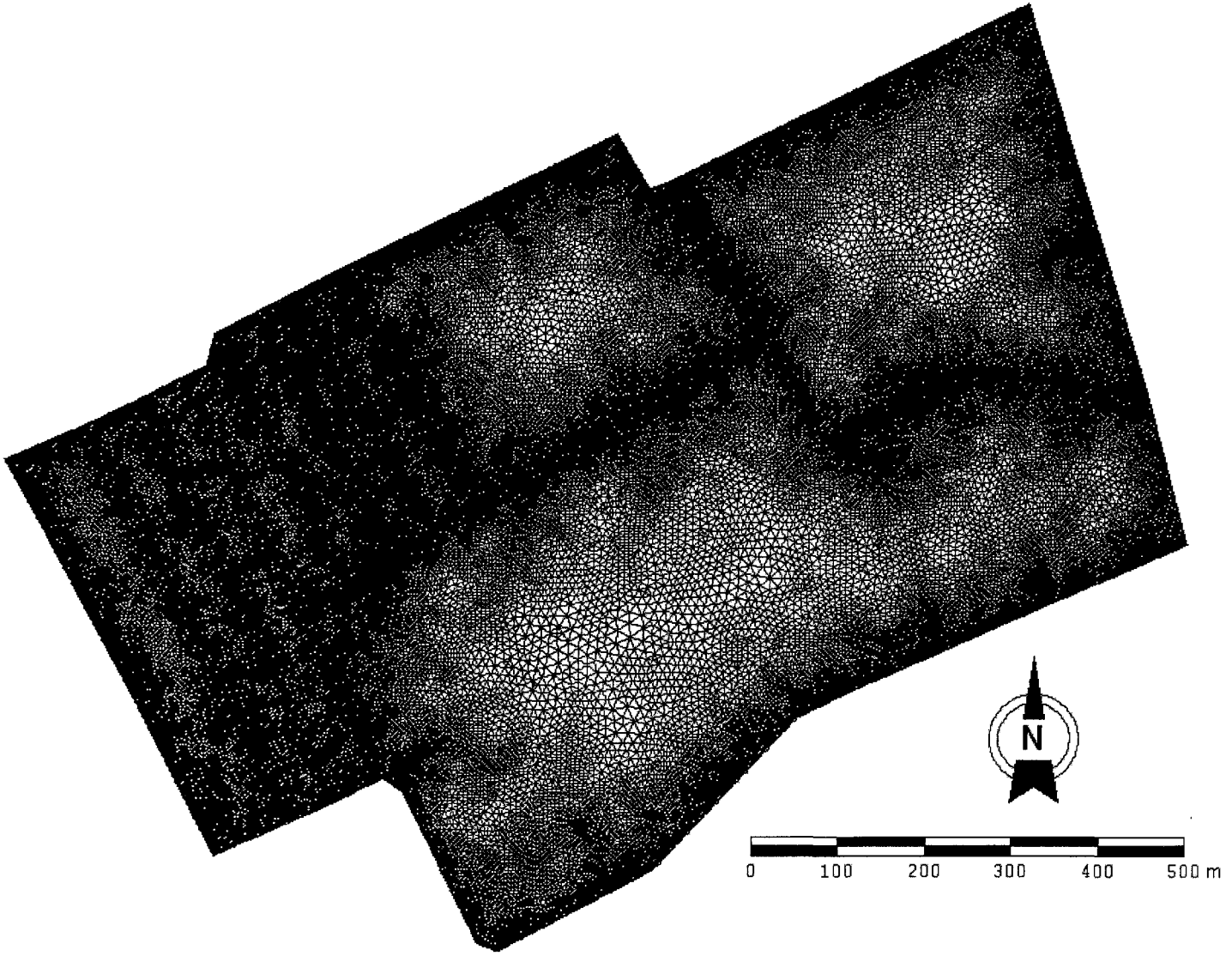


Figure B.0.1 - Maillage éléments finis utilisé pour la présente étude (80 121 nœuds et 157 999 éléments)

ANNEXE C – CHAMP SCALAIRE DE TOPOGRAPHIE PROJETÉ SUR LE MAILLAGE ÉLÉMENTS FINIS

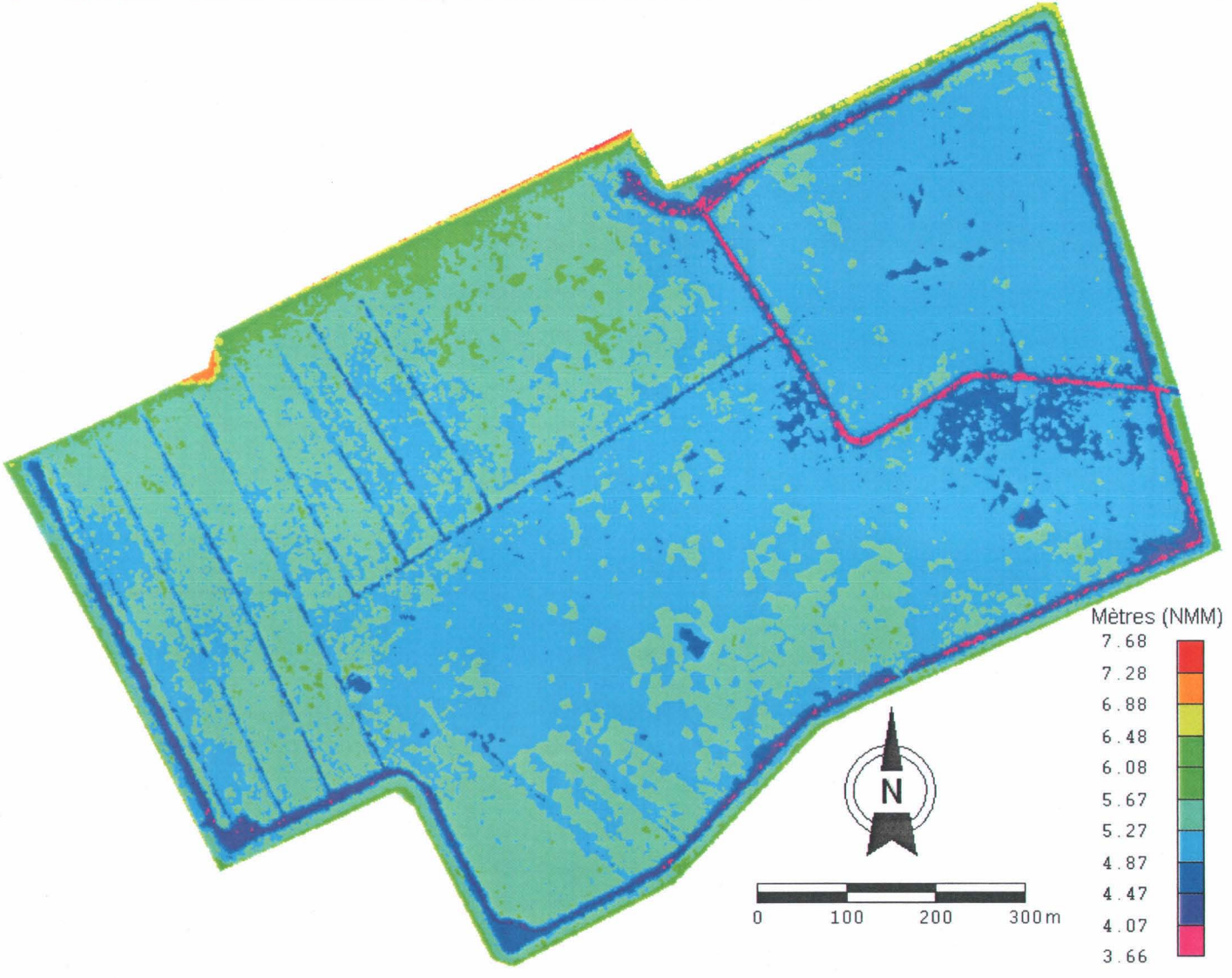


Figure C.0.2 – Champ scalaire des projections des cotes topographiques sur le maillage éléments finis

ANNEXE D – PRÉPARATION DES FICHIERS DE CONDITIONS LIMITES POUR H2D2

Il y a deux types de fichiers qui sont lus par le logiciel, possédant les extensions .cnd (pour condition) et .bnd (pour « boundary », i.e. frontière).

Fichier .bnd

Le fichier .bnd contient les coordonnées des nœuds situés là où la condition limite est appliquée. En outre, il peut s'agir des nœuds sur les arêtes situées en sortie d'un canal test, où appliqué une condition libre de sortie, par exemple.

La syntaxe d'un fichier .bnd va est illustrée à l'exemple suivant :

```
3
condition_1   233   255   624   685   985   988
condition_2   999   1024  1225  1546  1971
condition_3   1989  2014  2238  2589
```

Dans ce cas-ci, il y a trois différentes conditions qui sont appliquées, comme nous l'indique le chiffre en entête. Ensuite, la colonne de gauche représente le nom de la condition et les colonnes situées à droite représentent le numéro des nœuds où cette condition précise est appliquée. Il est à noter que le nombre de colonnes n'a pas besoin d'être le même pour chacune des conditions.

Dans le cas d'une limite longue (où un grand nombre de nœuds est impliqué), il faut la séparer en plusieurs tronçons, étant donné que le nombre de caractères toléré sur une ligne dans un fichier .bnd ne dépasse pas 1000. Attention, il faut répéter le nœud qui est à la limite entre les deux tronçons autrement il y aura un vide dans la détection des éléments de contour. Un exemple de syntaxe s'impose ici :

```
4
tronçon_1   1     2     3     4     [...]  201
tronçon _2  201   258   356   483   [...]  1989
tronçon _3  1989  2014  2238  2579  [...]  2598
tronçon_4   2598  3091  3555
```

Dans le cas précédant, la condition a été subdivisée en quatre tronçons différents, en raison de la quantité importante de nœuds impliqués.

Fichier .cnd

Il y a trois types de conditions qui peuvent être appliquées, et chacune possède un code numérique dans H2D2.

- Code 1 → Dirichlet
- Code 2 → Cauchy
- Code 3 → Sortie libre

Bien entendu, les trois conditions requièrent la présence d'une valeur numérique. Cependant, cette dernière possède est pertinente seulement pour Dirichlet ou Cauchy. En d'autres mots, la valeur numérique liée à une condition de sortie libre n'a pas d'importance : qu'il s'agisse d'un 0.0, de 1000.0 ou de 99.9, cela n'a aucun impact dans le calcul effectué par H2D2.

```
3                1017633600
pourtour_1       3    0.0
pourtour_2       3    0.0
pourtour_3       3    0.0
```

Sur la première ligne, le premier chiffre dénote le nombre de conditions présentes, alors que le second nombre représente la valeur du temps (ici, en secondes, qui correspond au 1er avril 2002 à 00 :00). À partir de la deuxième ligne, la première colonne fait appel au nom des conditions limites (voir fichier .bnd). À droite, la colonne contenant des 3 fait appel à la condition limite de type Neumann. Finalement, la colonne d'extrême droite contient les valeurs numériques associées à la condition de type 3. Ici, on a choisi des zéros, mais nous aurions pu opter pour n'importe quelle autre valeur et cela aurait généré exactement les mêmes résultats.

Dans le fichier .cnd, la valeur des conditions limites peut varier en fonction du temps, comme nous le démontre l'exemple suivant.

```
3                1017979200
Lasalle          1    2.38
MIP              1    2.38
Assomption       1    2.38

3                1017982800
Lasalle          1    2.36
MIP              1    2.36
Assomption       1    2.36

[...]
```

Ici, on remarque que la condition de Dirichlet est différente selon la valeur prise par le temps.

ANNEXE E – EXTRACTION DES DONNÉES MÉTÉOROLOGIQUES

La présente annexe détaille les procédures à suivre afin d'extraire les données météorologiques à partir des fichiers de base de données du Service de Climatologie d'Environnement Canada.

1- Extraction des données brutes par paramètre

- permet d'extraire les paramètres d'intérêt (par exemple, la radiation solaire) à partir d'un fichier possédant les données météorologiques brutes (format du Service de Climatologie)
- utilisation de l'exécutable **xtrmeteo_h.exe**

2- Homogénéisation dans le temps

- utilisation de l'exécutable **hmgtmp.exe**
- permet de ramener toutes les données météorologiques sur le même pas de temps. Par exemple, si nous désirons connaître la valeur horaire d'un paramètre qui a été mesuré à toutes les deux heures par une station météo, l'exécutable **hmgtmp.exe** nous permettra d'y parvenir par interpolation linéaire.
- paramètres d'entrée : 1) fichier source de données météo
  2) temps initial (en repère de secondes)
  3) temps final (en repère de secondes)
  4) pas de temps (en secondes)
- paramètre de sortie : fichier pour le stockage de données homogénéisées

3- Génération de valeurs nodales

- utilisation de l'exécutable **genvno.exe**
- permet d'assigner une valeur pour chacune des propriétés nodales lues par H2D2.

Si une seule station météo possède des données reliées au domaine d'étude, alors chaque nœud se verra assigner la même valeur, peu importe sa position dans le domaine. Or, si nous sommes en présences de plusieurs sources de données (i.e. différentes stations météo), une interpolation linéaire doit être effectuée entre les stations météorologiques.

ANNEXE F – FICHIER DE COMMANDE - SIMULATION SUR TOUTE LA PÉRIODE

Ici est présenté le fichier de commande utilisé lors de la simulation sur toute la période (10 avril 2002 0:00 au 6 mai 2002 0:00).

```
# SIMULATION FINALE avec aucune initialisation stationnaire
# tf=4;turb=0;dm=5E-06;alfa=1;SF=0
# début : 10 avril 2002 à 00:00
# fin : 6 mai 2002 à 00:00

h_frm = form('CD2D_Temperature')
h_limiter = limiter(h_frm, 50.0)

!-----------------------------------
!  Répertoires
!-----------------------------------
rep_mail = 'I:\nadeauda\Etude_Saint-Barthelemy\H2D2_vrai\Maillage'
rep_hydro= 'I:\nadeauda\Etude_Saint-Barthelemy\H2D2_vrai\Hydro'
rep_meteo= 'I:\nadeauda\Etude_Saint-Barthelemy\H2D2_vrai\MNM'
rep_simu = 'I:\nadeauda\Etude_Saint-Barthelemy\H2D2_vrai\Simulations'

!-----------------------------------
!  Maillage
!-----------------------------------
h_cor  = coor(rep_mail+'\segment4_sb.cor')
h_ele  = elem(rep_mail+'\segment4_sb.ele')
h_grid = grid(h_cor, h_ele)

!-----------------------------------
!  Propriétés nodales
!-----------------------------------

! FICHIERS RELIÉS À L'HYDRODYNAMIQUE

h_vno_vitesse_x         = vnod(rep_hydro + '\champ_vitesse_x.prn')
h_vno_vitesse_y         = vnod(rep_hydro + '\champ_vitesse_y.prn')
h_vno_profondeur        = vnod(rep_hydro + '\profondeur.prn')
h_vno_diff_vert         = vnod(rep_hydro + '\diffusivite_vertic.prn')
h_vno_diff_horiz        = vnod(rep_hydro + '\diffusivite_horiz.prn')

! FICHIERS RELIÉS À LA MÉTÉO

h_vno_radiation_solaire= vnod(rep_meteo + '\radiation_solaire\stations\Hrtmp_nnt.vno')
h_vno_couvert_vegetal   = vnod(rep_meteo + '\couvert_vegetal\stations\Hrtmp_nnt.vno')
h_vno_turbidite         = vnod(rep_meteo + '\turbidite\stations\Hrtmp_nnt.vno')
h_vno_temperature_air   = vnod(rep_meteo + '\temperature_air\stations\Hrtmp_nnt.vno')
h_vno_emissivite        = vnod(rep_meteo + '\emissivite\stations\sinokrot.vno')
h_vno_fonction_du_vent  = vnod(rep_meteo + '\fonction_du_vent\stations\brown.vno')
h_vno_humidite_relative = vnod(rep_meteo +
'\humidite_relative\stations\Hrtmp_nnt.vno')
h_vno_precipitation     = vnod(rep_meteo + '\precipitation\stations\Hrtmp_nnt.vno')
h_vno_glace             = vnod(rep_meteo + '\glace\stations\Hrtmp_nnt.vno')
h_vno_extinction        = vnod(rep_meteo + '\extinction\stations\Hrtmp_nnt.vno')
h_vno_fond              = vnod(rep_meteo + '\fond\stations\Hrtmp_nnt.vno')

h_prn  =
prno(h_vno_vitesse_x,h_vno_vitesse_y,h_vno_profondeur,h_vno_diff_vert,h_vno_diff_horiz
,h_vno_radiation_solaire,h_vno_couvert_vegetal,h_vno_turbidite,h_vno_temperature_air,h
_vno_emissivite,h_vno_fonction_du_vent,h_vno_humidite_relative,h_vno_precipitation,h_v
no_glace,h_vno_extinction,h_vno_fond)
```

```
!-----------------------------------
!  Propriétés élémentaires
!-----------------------------------
h_pre  = 0                          ! Pas de prel

!-----------------------------------
!  Propriétés globales
!-----------------------------------
h_pgl  = prgl(5.00000E-06,              ! DIFFUSIVITE MOLECULAIRE
              1.00000E+00,              ! COEF. POND. DIFFUSIVITE VERTICALE
              1.00000E+00,              ! COEF. POND. DIFFUSIVITE HORIZONTALE
              1.00000E+00,              ! COEF. POND. DIFFUSIVITE LONGITUDINALE
              1.00000E+03,              ! MASSE VOLUMIQUE DE L'EAU
              4.18160E+03,              ! CHALEUR SPECIFIQUE DE L'EAU
              9.17000E+02,              ! MASSE VOLUMIQUE DE LA GLACE
              2.07500E+03,              ! CHALEUR SPECIFIQUE DE LA GLACE
              3.34000E+05,              ! CHALEUR LATENTE DE FUSION DE LA GLACE
              101260.0,                 ! PRESSION ATMOSPHERIQUE
              4.00000E+00,              ! TEMPERATURE DU FOND
              1.06297142669025E+08,     ! COEFFICIENT DE LINEARISATION A (entre 0 et
50 C)
              5.33908718317524E+09)     ! COEFFICIENT DE LINEARISATION B (entre 0 et
50 C)


!-----------------------------------
!  Sollicitations
!-----------------------------------
h_slc = 0
h_slr = 0

!-----------------------------------
!  Conditions limites
!-----------------------------------

h_cnd  = condition(rep_simu +'\segment4_sb.cnd')
h_lmt  = boundary(rep_simu + '\segment4_sb.bnd')
h_bc   = boundary_condition(h_lmt,h_cnd)

!-----------------------------------
!  Degrés de liberté
!-----------------------------------
h_ddl  = dof()

!-----------------------------------
!H NUM
!-----------------------------------
h_num = 0

!-----------------------------------
!  Données globales de simulation
!-----------------------------------
h_sol  = simd(h_frm, h_num, h_grid, h_ddl, h_bc, h_slc, h_slr, h_pgl, h_prn, h_pre)

!-----------------------------------
!  Définition des algo de résolution
!-----------------------------------
h_resmat  = ldu_memory()
h_picard  = picard (h_resmat, 1, 1.0e-15)
h_euler   = euler  (h_picard, 1.0)
```

```
!-----------------------------------
!  Résolution
!-----------------------------------
t_ini  = 1018411200       !  Début de la simulation : 10 avril 2002 à 00:00
t_del = 3600              !  Pas de temps horaire
t_npas = 26*24            !  Fin de la simulation : 6 mai 2002 à 00:00

h_post = zzzz (rep_simu+'\Resultats\simu_finale.pst')

init_value(h_sol, 10.0)

solve(h_sol,h_euler,h_limiter,h_post,t_ini,t_del,t_npas)

save (h_sol, rep_simu+'\Resultats\simu_finale.fin')
stop()
```

ANNEXE G – APPEL DU LOGICIEL DE CALCUL H2D2

Afin de pouvoir contrôler la lecture du fichier de commande (**.inp**) et la production d'un fichier **.out** détaillant les résultats numériques de la simulation, une façon pertinente d'appeler l'exécutable **h2d2.exe** est d'employer un fichier **.bat**.

En premier lieu, l'exécutable est appelé, puis sont mentionnés les fichiers d'entrée et de sortie.

Exemple

Supposons que l'exécutable **h2d2.exe** est situé dans le répertoire C:\H2D2. Quant à lui, le fichier de commandes serait localisé dans le répertoire C:\Simulations, soit au même endroit où nous serions intéressés à inscrire le fichier de sortie. Voici donc le contenu du fichier **.bat**.

```
"C:\H2D2\H2D2.exe"                    <"C:\Simulations\fichier_commande.inp">
"C:\Simulations\fichier_sortie.out"
```

ANNEXE H – CHAMP SCALAIRE DE TEMPÉRATURE - 6 MAI 2002 0 :00

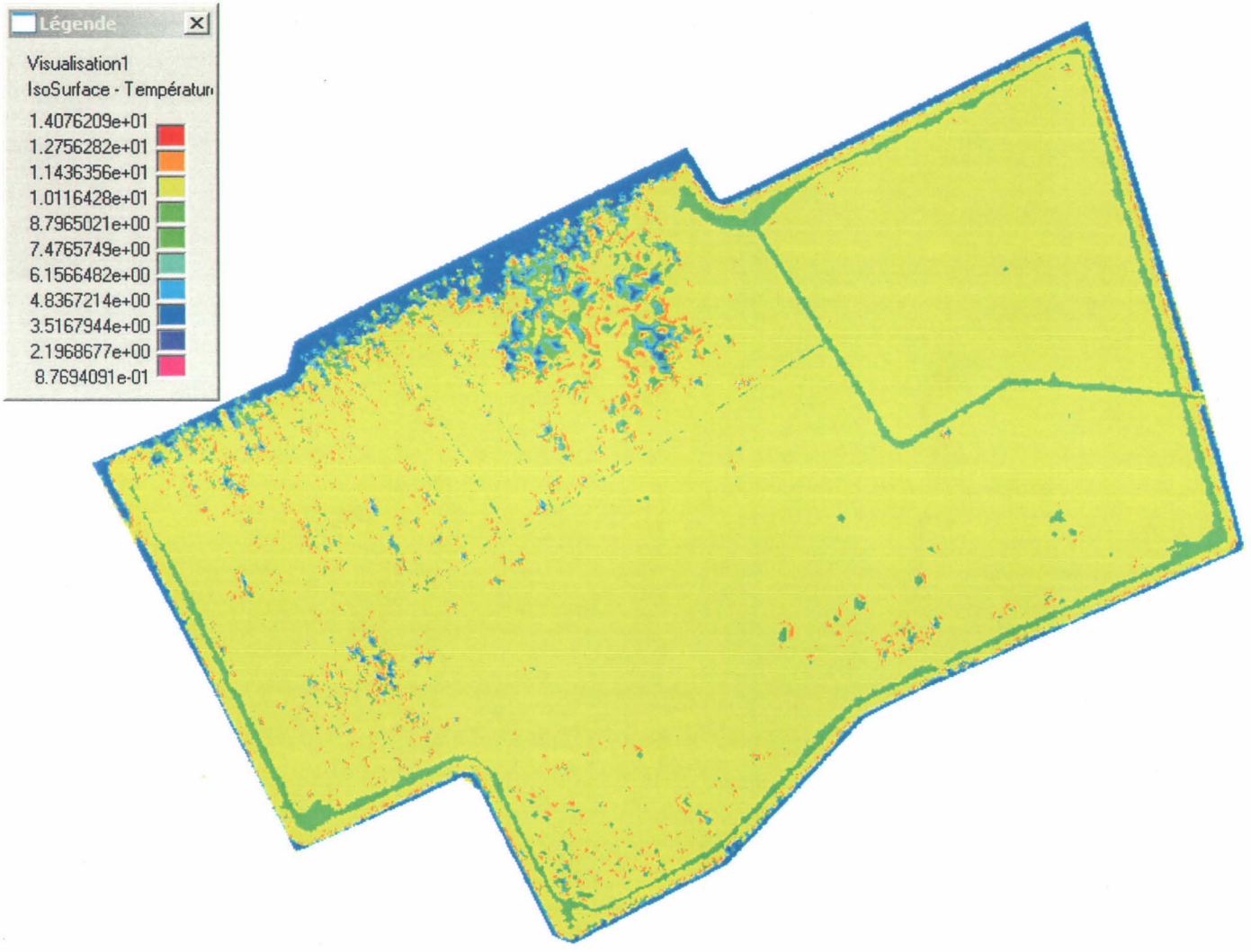


Figure H 0.3 – Champ scalaire de température pour le dernier pas de temps de la simulation sur toute la période