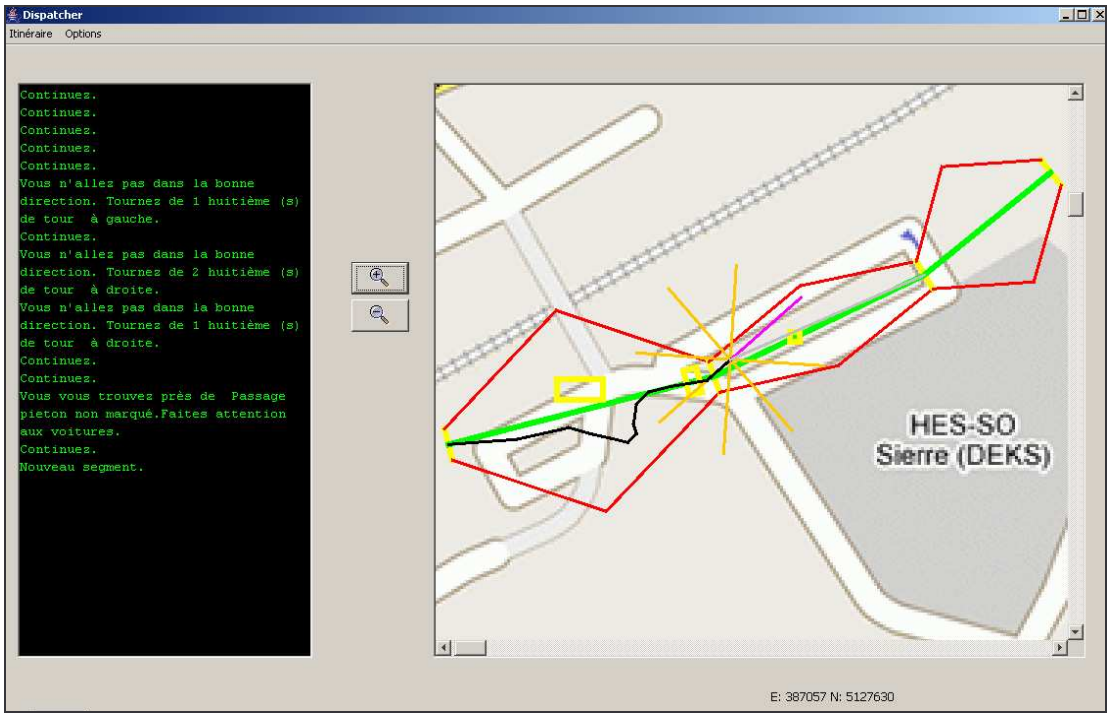
Etudiant : Vladimir Drakic
Professeur : David Russo

Dispatcher – suivi avec GPS



Filière informatique de gestion
Diplôme 2005 / 2006

HEVs
haute école valaisanne
hochschule wallis

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz

Remerciements

Monsieur Nestor von Heydebrand, Product Manager Tracking de l'entreprise *Novasys* pour sa disponibilité et son aide répétée pendant le développement du module de connexion.

Monsieur Jean-Luc Cochard, Sales Director France, Belgium, North and West Africa de l'entreprise *Bucher & Suter* pour m'avoir dédié son temps lors de la phase de définition des besoins.

Professeur David Russo de l'HEVs pour m'avoir guidé et conseillé tout au long de ce travail.

Table des matières

1 Introduction

Les technologies de l'information et de la communication offrent des occasions multiples de créer et d'exploiter des applications qui permettent d'améliorer le quotidien de certaines populations. Dans ce sens, la technologie Global Positioning System (GPS) ouvre la voie à des applications variées et particulièrement intéressantes.

L'utilisation de la technologie GPS et de Java ainsi que la perspective de réaliser un projet réel qui ait un impact concret sur une population donnée ont été des éléments déterminants dans mon choix de travail de diplôme.

Ce travail de diplôme a pour but de développer une application qui utilise la technologie GPS et qui aide les personnes malvoyantes dans leurs déplacements. Cette application constitue un premier prototype, nommé *Dispatcher*, qui permettra à une personne malvoyante de se déplacer sur un itinéraire, défini au préalable, grâce aux indications données par un opérateur distant qui reçoit le positionnement de la personne par GPS.

Le présent rapport est divisé en 5 parties. La partie Cadrage qui décrit le projet et permet de délimiter son périmètre, les parties prenantes et les contraintes qui ont été identifiées.

La partie Démarche qui détaille la manière dont j'ai mené à bien le projet et décrit la planification du projet et les technologies utilisées.

La partie Développement, qui est la plus importante de ce rapport et décrit, dans un ordre logique, tous les composants de l'application, de manière à rendre compréhensible son fonctionnement dans son ensemble. Les raisonnements qui m'ont amenés à la solution livrée et les problèmes rencontrés sont également abordés tout au long de cette partie.

Sous la Synthèse sont décrits les avantages et les désavantages de l'application et enfin la Conclusion qui offre un bilan de ce travail de diplôme.

Le cahier des charges, le manuel d'utilisation de l'application et la documentation Javadoc sont disponibles en annexe.

2 Cadrage du projet

2.1 Parties prenantes

Les parties prenantes du projet sont :

Nom	Institution	Fonction dans le projet
Prof. David Russo	HEVs	Mandant et professeur responsable.
Monsieur Jean-Luc Cochard	Bucher & Suter AG	Client final et expert lors de la défense du travail de diplôme.
Monsieur Nestor von Heydebrand	Novasys AG	Récupération des coordonnées de l'appareil Mambo à l'aide du service web. Configuration de l'appareil.
Vladimir Drakic	HEVs	Réalisation du projet.

2.2 Description du projet

Ce travail de diplôme a pour objectif de créer un logiciel destiné aux personnes malvoyantes. Ce logiciel doit permettre de guider une personne à travers un itinéraire qu'elle ne connaît pas à l'aide d'un appareil portable. Guider signifie que le logiciel analyse le mouvement de la personne et lui envoie des messages concernant sa position. Ces messages aideront l'utilisateur à s'orienter, et à parcourir un itinéraire en dépassant tous les obstacles qui pourraient se présenter à lui. L'itinéraire peut se trouver sur une zone urbaine ou il peut représenter un parcours d'une randonnée dans la nature.

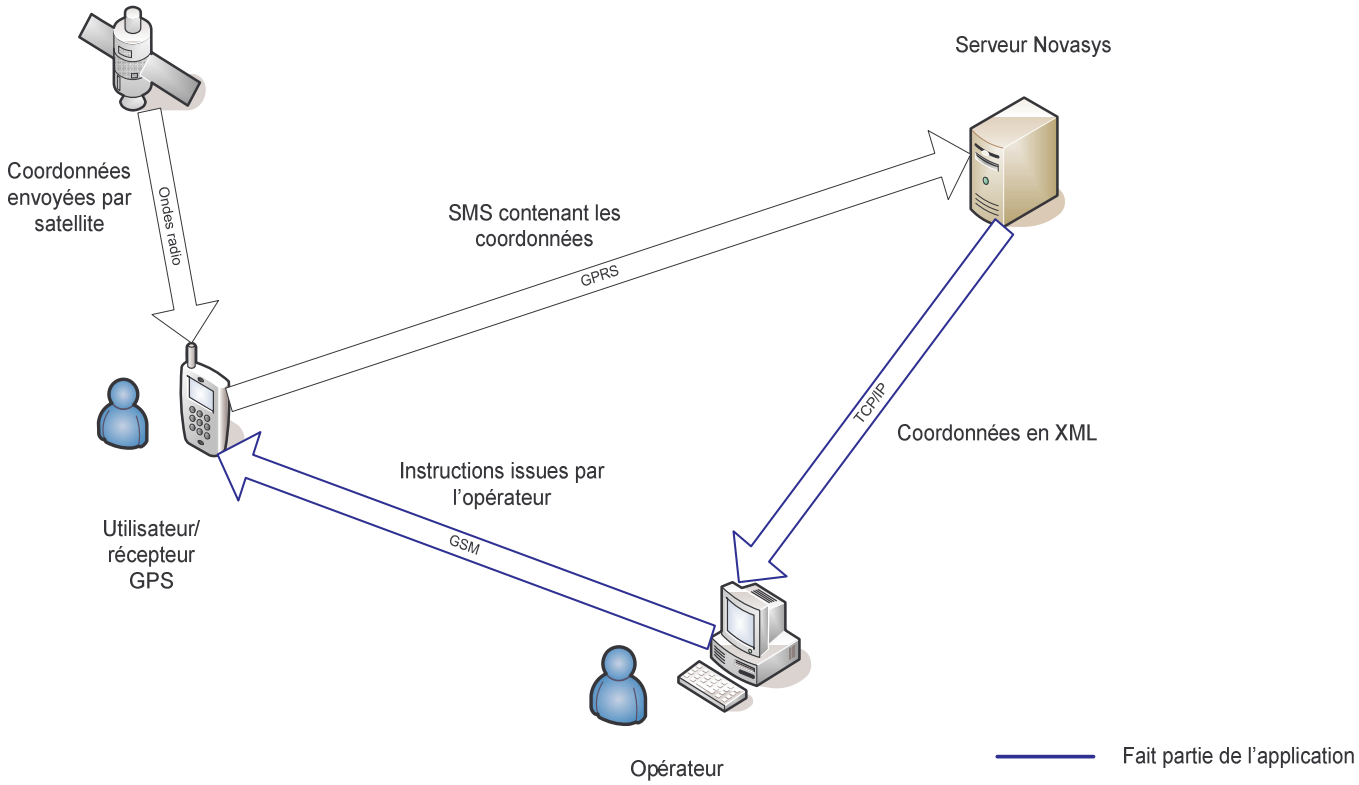
Le mode de fonctionnement est simple. On définit un itinéraire à l'avance. Pendant que l'utilisateur marche, son appareil GPS envoie les coordonnées de sa position vers le serveur intermédiaire de l'entreprise Novasys. Le logiciel récupère les coordonnées depuis ce serveur, les analyse et, si nécessaire, donne une instruction qui redirige l'utilisateur. Le logiciel contrôle la position de l'utilisateur et anticipe quel serait le meilleur prochain mouvement pour qu'il arrive à la destination qui avait été fixée.

Au cours de son déplacement sur l'itinéraire, il peut arriver qu'un obstacle empêche le progrès de l'utilisateur. Dans ce cas, il faut informer l'utilisateur de l'action à entreprendre.

Sur la présente version de l'application, les instructions concernant le mouvement à effectuer s'affichent sur l'écran. Ensuite, c'est une personne, l'opérateur, qui les communique à l'utilisateur par téléphone. On peut aussi considérer l'opérateur devant l'écran comme un utilisateur de l'application, mais pour éviter toute confusion, le terme *utilisateur* dans la suite de ce document, signifie l'utilisateur final, celui qui traverse l'itinéraire avec l'appareil GPS.

L'image suivante illustre le mode de fonctionnement de l'ensemble du dispositif.

Flux de données



2.3 Conditions et contraintes de projet

Selon les données initiales du projet, Java est choisi comme le langage de développement pour que l'application soit exploitable sur plusieurs plateformes.

Un autre élément important est que le concept et l'implémentation de l'itinéraire doit être compatible avec le SIG (Système d'information géographique) afin de laisser ouverte la possibilité d'une intégration future.

Ce logiciel est prévu d'être davantage une librairie réutilisable et ouverte pour le développement futur, que de représenter une application complète et exploitable sur le marché. Pour cette raison, il est crucial que le logiciel soit bien structuré et documenté en détail. Il est également important que l'application soit pleinement configurable.

L'itinéraire sur lequel se déplace l'utilisateur, n'est pas une ligne simple, mais il comprend une marge autour de laquelle le suivi de l'utilisateur est réalisé. Une fois que l'utilisateur dépasse les bornes et sort de la zone de l'itinéraire, le contrôle de son mouvement s'arrête et le suivi n'est pas effectué jusqu'à ce qu'il n'y retourne dedans.

L'itinéraire est créé à l'avance par le développeur et est statique, enregistré sur une application externe. Pendant la création d'un itinéraire il faut s'assurer qu'il n'y ait pas d'éléments dangereux sur le chemin de l'utilisateur. Ces objets comprennent : trous, surfaces d'eau (lacs, rivières, etc.) et tous les autres endroits qui peuvent mettre en jeu la vie de l'utilisateur.

Les éléments dangereux sont écartés mais les obstacles pouvant se trouver sur le chemin doivent être pris en compte par l'application. Les obstacles ne doivent pas présenter des objets dangereux pour la vie. Les obstacles trouvés sur l'itinéraire peuvent être soit des objets empêchant le passage comme un arbre, une roche ou des voitures garées, soit des objets qui exigent une action de la part de l'utilisateur, comme le passage piéton où il faut attendre le feu vert.

Les messages envoyés à l'utilisateur doivent être créés de façon compréhensible à l'humain.

Initialement le logiciel ne prévoyait pas la représentation graphique de l'itinéraire. En accord avec le client, il a été décidé qu'il faut créer un affichage de base de l'itinéraire et des mouvements effectués par l'utilisateur.

3 Démarche

3.1 Calendrier et planification du travail

Le détail de la planification des mes tâches est offerte à l'annexe « Planification ». Le calendrier permettant de suivre la progression détaillée de mon travail se trouve également en annexe.

3.2 Technologies utilisées

Pour le positionnement de l'utilisateur, on utilise l'appareil *Mambo* produit par Falcom, une entreprise allemande. C'est un appareil intégrant le téléphone, le récepteur GPS et GPRS. *Mambo* est adapté aux besoins des personnes aveugles et n'a pas d'écran. On peut le configurer soit en utilisant la connexion Bluetooth et le logiciel de configuration, soit en envoyant un message SMS avec des commandes.

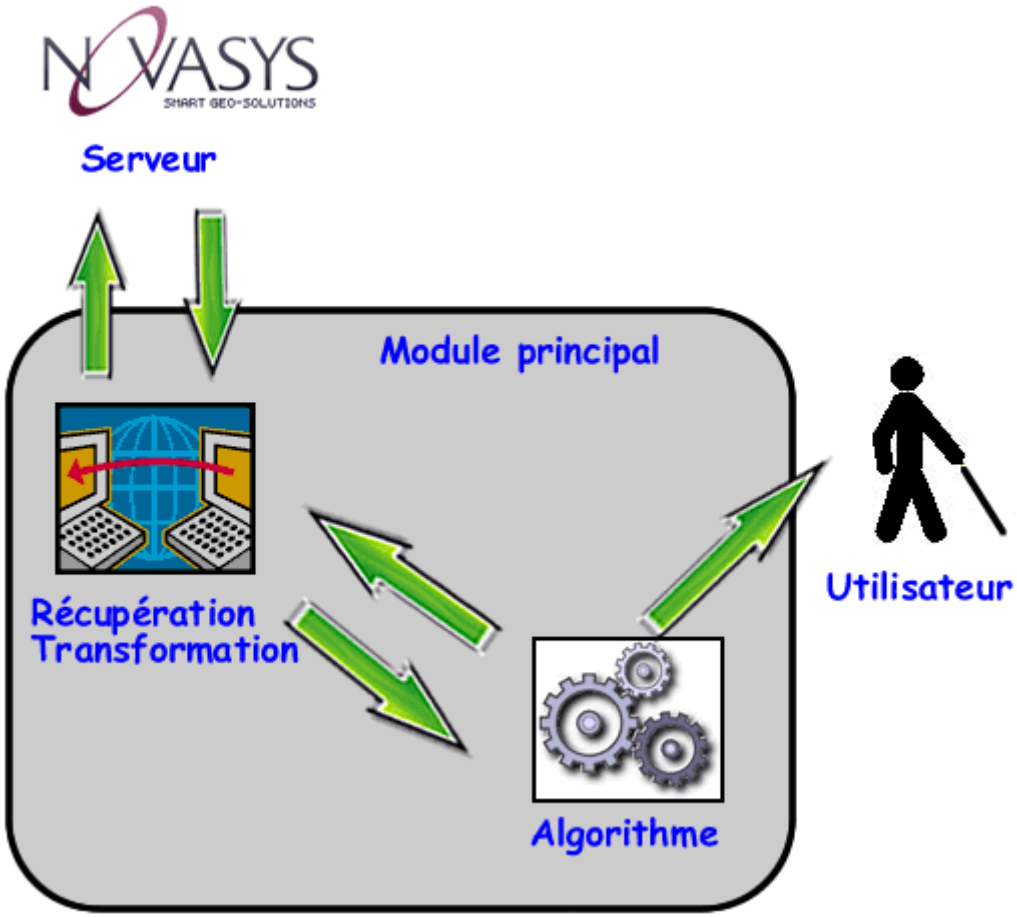
Vu que le logiciel doit être réutilisable, il est important que le code soit bien structuré est standardisé, car sinon il devient illisible pour les futurs développeurs. Pour cette raison, j'ai choisi d'employer *Rational Rose Enterprise Edition* pour la création et la génération des classes Java. Ce logiciel m'a également facilité le développement parce qu'il diminue le nombre d'erreurs commises et la quantité du code à écrire.

Le format du fichier contenant l'itinéraire est *Shapefile*, et pour sa création la suite *ArcGIS* est utilisée. *ArcGIS* est *Shapefile* sont expliqués en détail dans le chapitre « Logiciels externes utilisés».

4 Développement

4.1 Structure de l'application

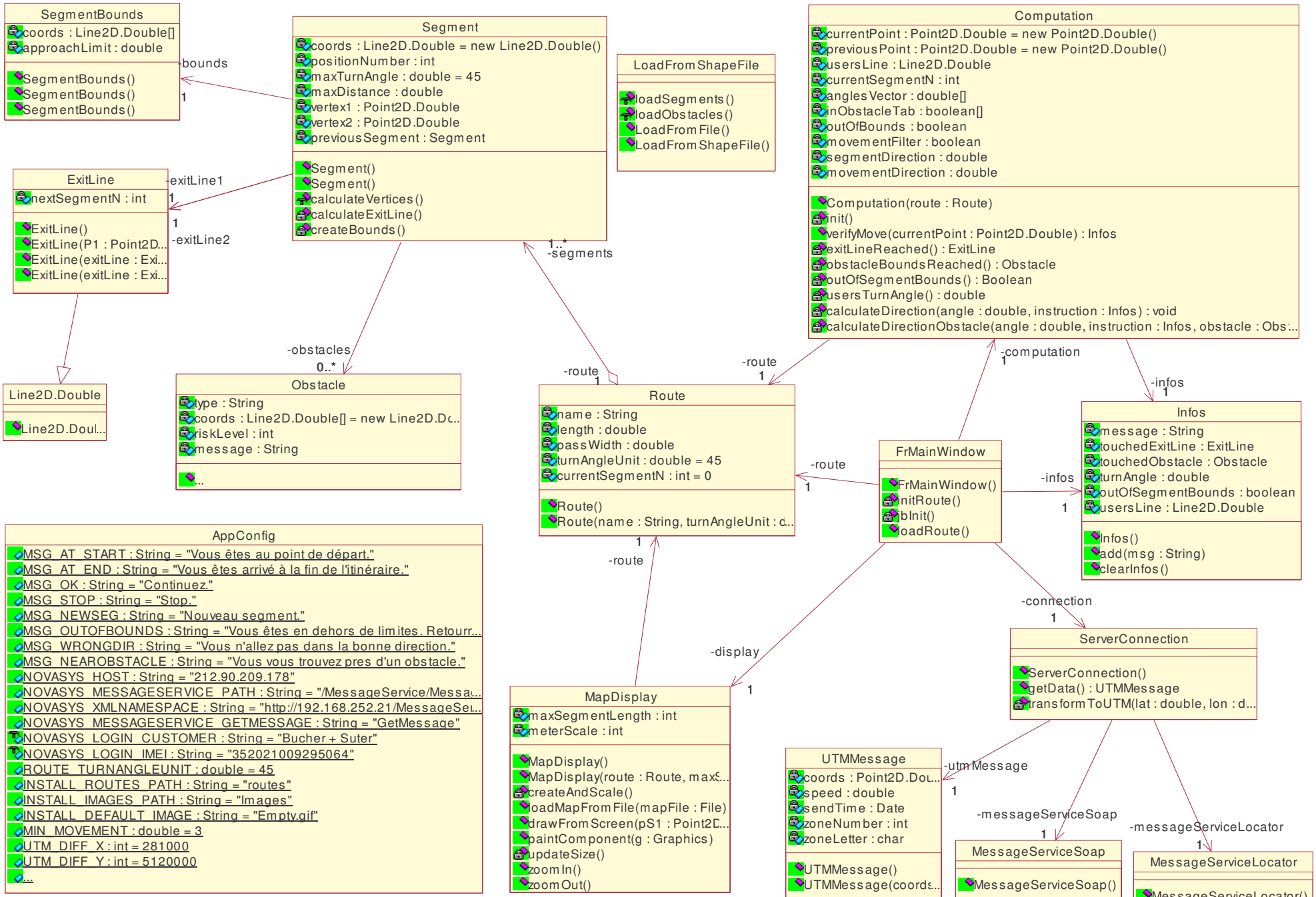
La vision simplifiée du logiciel, vue dans le cahier des charges (voir annexe « Cahier des charges »), comprend quatre entités (modules) qui regroupent des fonctions similaires. Ce sont le module principal, la récupération des données, les calculs – l'algorithme, et l'affichage.



Dès le début du projet j'ai essayé de définir les classes les plus importantes et de les structurer dans un diagramme. Ceci est très important car le diagramme des classes sert de fil conducteur tout au long de travail.

Le diagramme de classes final est représenté à la page suivante.

Diagramme de classes



Concernant le paramétrage du logiciel, toutes les constantes et les valeurs par défaut sont regroupées dans la classe AppConfig sous la forme de variables statiques. Parmi ces valeurs on peut également trouver les fragments des instructions pour l'utilisateur, ce qui facilite l'action de changement éventuel de la langue utilisée.

Globalement, les valeurs ne sont presque jamais écrites « en dur ». Dans la plupart des cas elles se trouvent dans des variables qu'on peut modifier facilement. Ceci veut dire que dans les expressions on ne trouve pas les valeurs concrètes mais des variables définies auparavant. Avec cette stratégie, l'application reste facilement configurable. De plus, il est possible de configurer certains composants de l'application dans les menus de la fenêtre principale.

4.2 Représentation dans le logiciel

Une fois le concept d'itinéraire défini, tel que vu plus haut dans ce document, il a fallu représenter les structures de la « vie réelle » sous une forme adaptée à l'ordinateur.

Pendant la période de l'analyse, j'ai envisagé de nombreuses solutions qui ont été d'abord discutées avec le professeur David Russo puis sélectionnés sur la base d'une comparaison entre ses avantages et ses désavantages.

L'idée est de créer des formes géométriques qui correspondent au mieux aux entités géographiques d'un itinéraire et qui permettent de réaliser les fonctionnalités attendues.

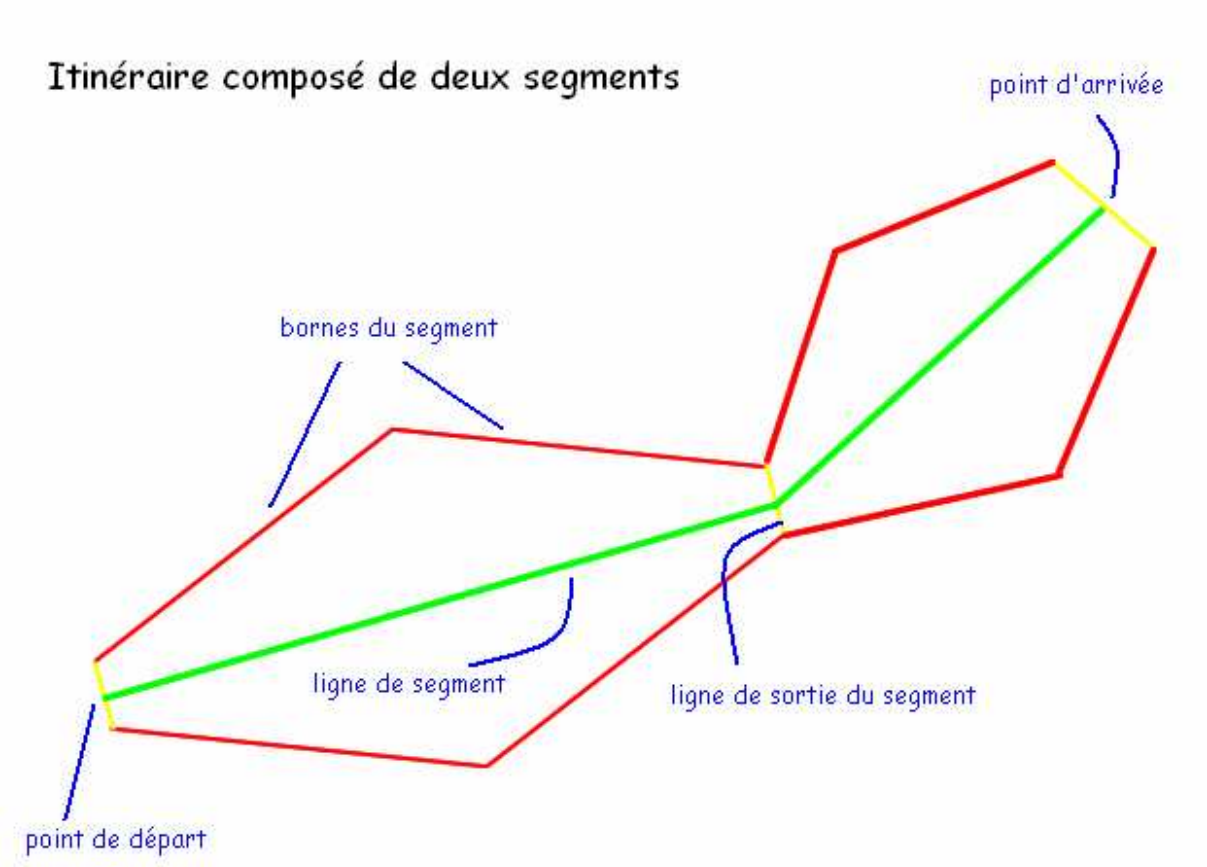
Au même temps, il m'a fallu prendre en compte la contrainte des ressources informatiques, à savoir, concevoir une application qui soit adaptée au matériel à disposition. N'ayant pas d'expérience dans le travail avec le graphisme et ne connaissant pas les techniques permettant de consommer moins de ressources, il était important d'éviter de surcharger le logiciel avec des objets trop complexes.

Pour cette raison, les figures géométriques sont réalisées avec les classes de Java 2D, une API incluse dans JDK (voir Aspect graphique), permettant la création et la manipulation des objets graphiques bidimensionnels.

Les coordonnées des classes comme Segment, Obstacle ou SegmentBounds sont représentées comme plusieurs objets de la classe Line2D de Java 2D. Cette classe comprend plusieurs méthodes utiles comme intersection(), contains() ou ptLineDist(), qui rendent le travail plus facile.

Au début du travail avec les éléments de l'itinéraire, je ne connaissais pas Java 2D, et j'ai dépensé beaucoup de temps, pratiquement 3 jours entiers, pour développer des méthodes mathématiques qui existaient déjà dans les classes géométriques de Java 2D. Les calculs sont basés essentiellement sur la géométrie analytique et sur l'algèbre linéaire de base (arithmétique vectorielle, angle entre deux vecteurs, etc).

Voyons maintenant une représentation graphique de l'itinéraire. Les noms des éléments son ceux des classes correspondantes.



4.3 Eléments d'un itinéraire

4.3.1 Route

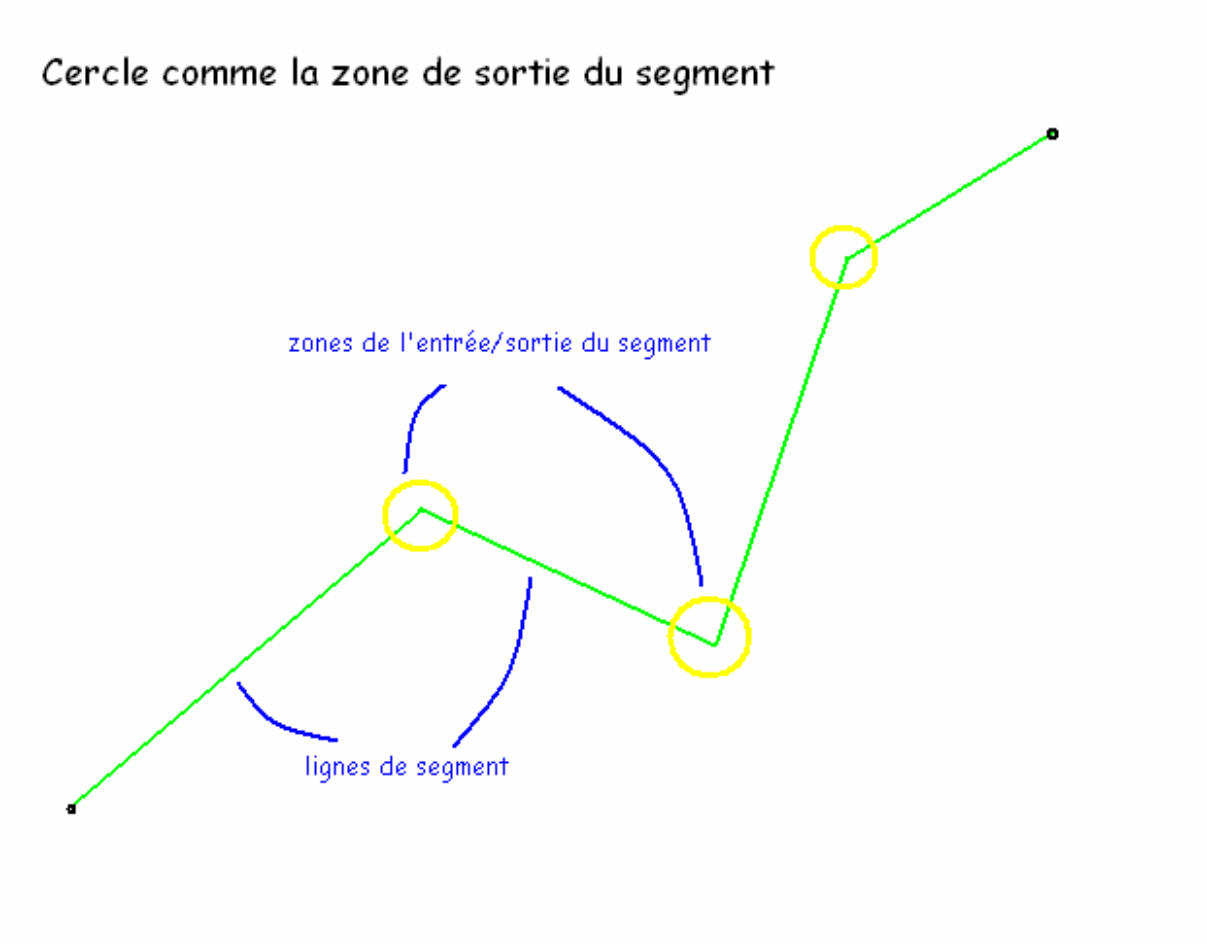
La classe Route décrit l'itinéraire en entier. Si l'on imagine le diagramme des classes comme un arbre, la Route serait la racine de cet arbre. Elle a le rôle d'un conteneur et ne possède pas ses coordonnées, mais contient la référence vers les segments (classe Segment) qui sont liés aux autres éléments ayant les coordonnées.

4.3.2 Segment

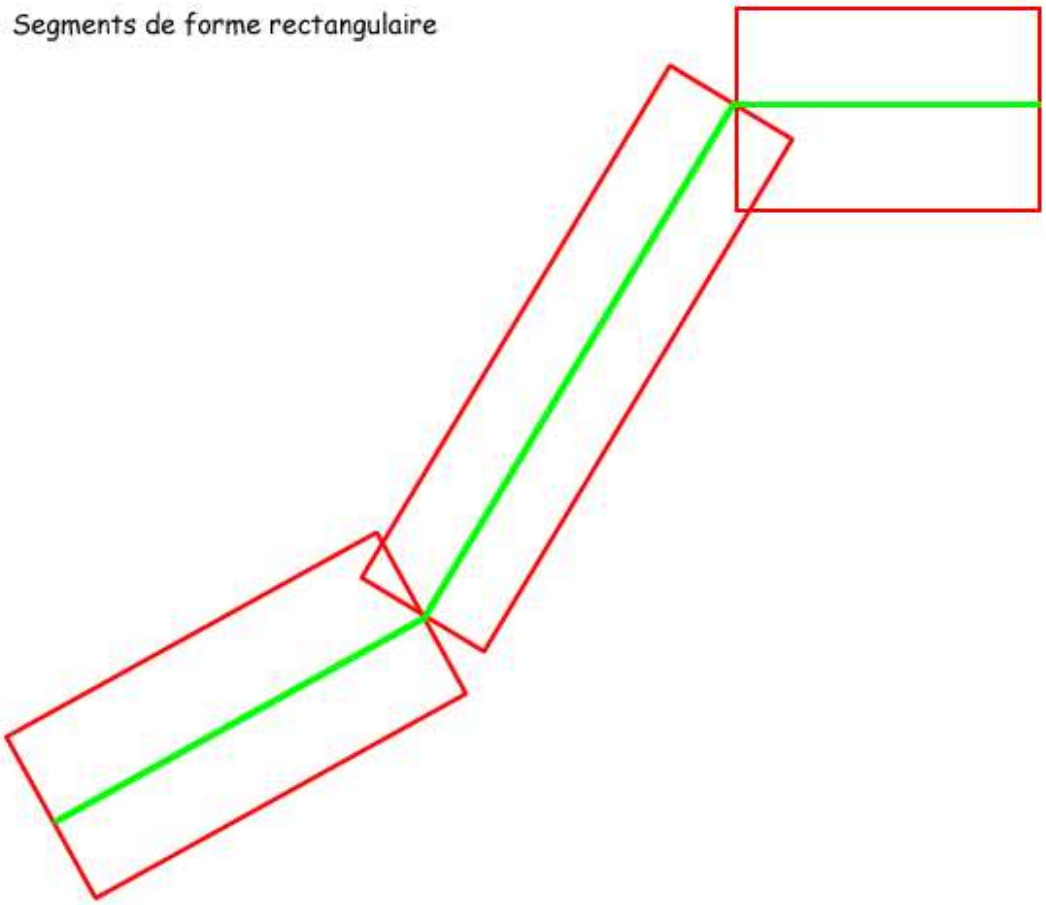
On peut dire que Segment est l'élément principal d'un parcours. L'itinéraire est divisé en plusieurs segments qu'on crée typiquement sur les points de détour ou s'il faut réduire la longueur d'un segment. Un segment est pratiquement une ligne droite ayant le début, la fin et la largeur, c'est-à-dire la limite de sortie du segment. Sa largeur, délimitée par les bornes du segment est en fait la distance de la ligne du segment jusqu'à la barrière (rien à voir avec la classe Obstacle) la plus

proche. Une barrière limitant le mouvement peut être un bâtiment, un arbre, un mur, bornes du trottoir etc. ou la combinaison de ces objets.

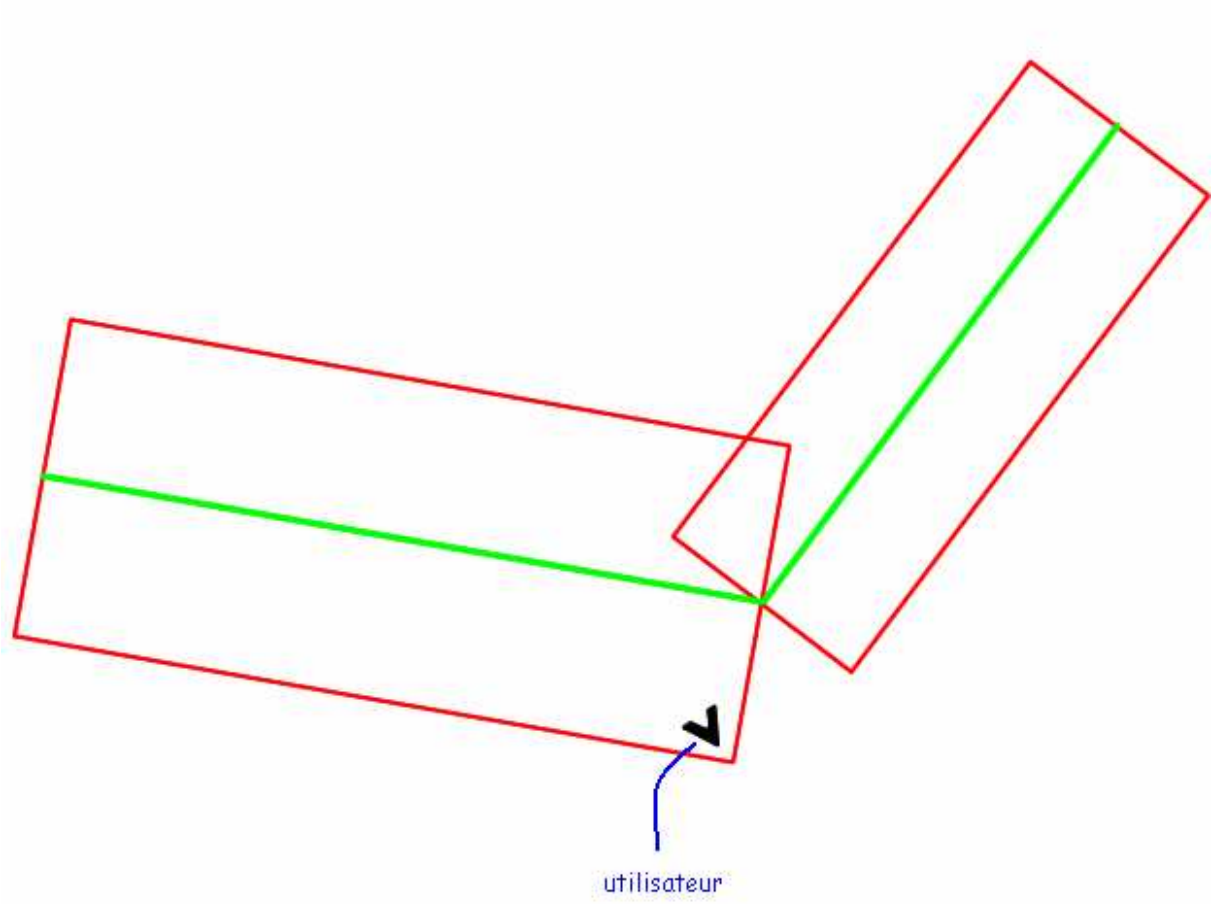
Pendant le design de composants de l'itinéraire, la classe Segment est celle qui m'a exigé le plus de temps. Les questions que je me suis posées étaient : quelle forme géométrique aura mon segment ? (ou comment décrire les bornes vers l'extérieur ?), et comment définir les liaisons vers les segments qui le suivent et le précédent ?
Concernant le passage d'un segment à l'autre, au début il semblait logique d'avoir un cercle (ellipse) comme la zone de sortie. Après avoir réfléchis sur ce sujet, j'ai trouvé que cette approche complique trop le processus car dans ce cas-là, la zone de sortie devient une surface spécifique qu'il faut traiter séparément. En effet, une fois dans le périmètre du cercle, l'utilisateur ne se trouve dans aucun segment, et on doit attendre sa sortie pour continuer le suivi.
L'implémentation d'une telle solution n'est pas impossible, au contraire, théoriquement elle ne pose pas beaucoup de problèmes sauf le temps de travail, mais j'ai voulu éviter d'alourdir l'application en rajoutant des éléments non obligatoires vu qu'il n'y a pas assez de temps imparti pour essayer plusieurs solutions et de trouver la meilleure parmi elles, et vu qu'il s'agit d'une première version et qu'il est donc préférable d'avoir le projet en entier fini que d'avoir des détails parfaits dans une application qui ne marche pas.



En accord avec le professeur Russo, avec lequel j'ai échangé des idées concernant la représentation d'un itinéraire ou de son « unité » principale – segment, nous avons identifié que la forme rectangulaire est, dans notre cas, idéale : l'axe au milieu du rectangle est la ligne du segment (chemin), les deux côtés les plus longs sont les bornes, et les deux côtés restants symbolisent les lignes de sortie du segment. La solution est assez légère en usage des ressources, les calculs sont simplifiés et on évite le problème des zones spéciales dédiées aux liaisons des segments. Cette forme de représentation du segment semblait sur le moment être la solution définitive.



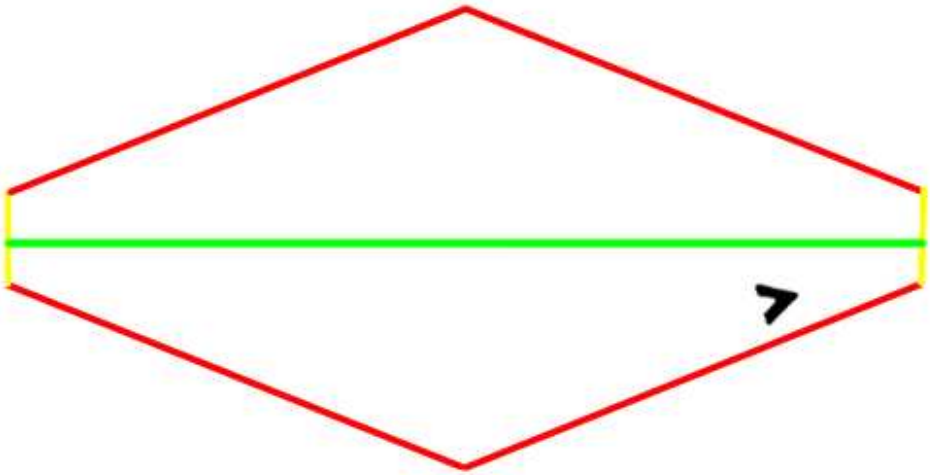
Une fois la forme du segment décidée, je me suis penché sur l'analyse de scénarios possibles des mouvements de l'utilisateur et des « réponses » de côté logiciel. C'est à ce moment que j'ai pu apercevoir que cette solution possède des faiblesses qui sont en contradiction avec le concept de l'application. D'abord, les bornes rectangulaires du segment laissent trop de liberté de mouvement à l'utilisateur dans le sens que cette forme n'aide pas à diriger l'utilisateur vers la sortie, ce qui est le but de l'application. Voyons une illustration de ce cas afin de mieux comprendre le problème.



Avec ce cas, nous voyons que l'utilisateur peut errer dans les coins du rectangle ce qui l'éloigne de la sortie et crée la situation d'une mauvaise navigation.
L'autre défaut de cette approche est que la zone de sortie entre deux rectangles n'est pas clairement définie. Si l'on essaie d'utiliser les côtés dans ce but, le problème persiste à cause des tailles différentes des rectangles.

Il était donc nécessaire de trouver une solution plus optimale. L'objet qui me semblait approprié est une forme composée de deux triangles à chaque côté de la ligne de segment, ce qui donne un rhomboïde. Le triangle isocèle satisfait bien la tâche de navigation vers la sortie du segment car on laisse au randonneur la possibilité de s'éloigner de chemin (ligne de segment) jusqu'au milieu du segment où l'autre côté du triangle commence à le mener vers la sortie.
En utilisant la forme triangulaire à chaque côté on réduit également la surface du segment, qui peut être au maximum une moitié de la surface du rectangle, et au même temps on garde les bons caractéristiques de la forme rectangulaire comme la linéarité et la simplicité des calculs. De plus, ce nouvel objet est un polygone composé des quasi-triangles car un triangle parfait implique un point en tant qu'entrée et sortie, ce qui ne permet pas d'inclure la marge de déplacement de l'utilisateur. Avec des quasi-triangles, il est possible de créer deux lignes de sortie (classe ExitLine) partagées avec le segment précédent et suivant.
L'image suivante illustre et rend plus compréhensible la solution envisagée.

Utilisateur dans le segment de forme triangulaire

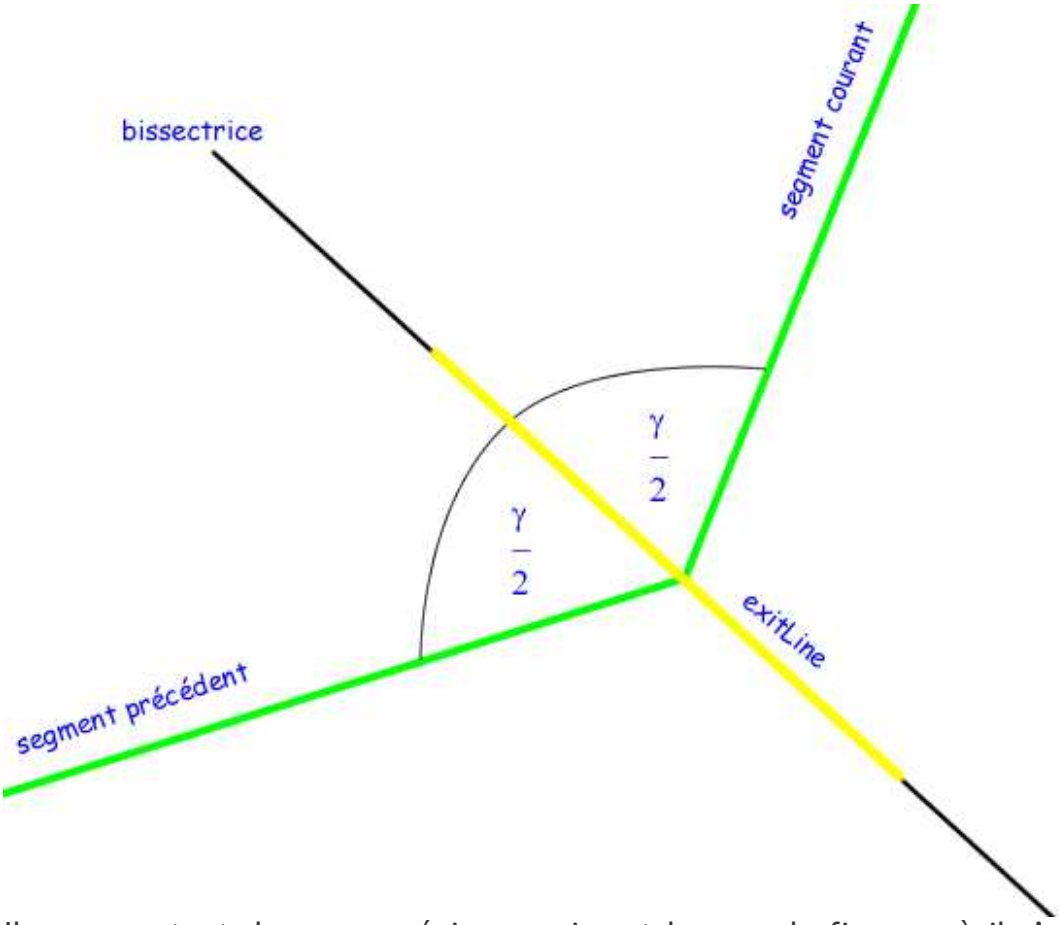


Les détails techniques de la classe Segment sont les suivants. Les bornes d'un segment, représentées par la classe SegmentBounds, sont créées automatiquement une fois la ligne de segment et les lignes de sortie définies. Le sommet du triangle est « orthogonal » par rapport au milieu de la ligne de segment et il en est éloigné de la valeur définie par l'utilisateur (variable maxDistance). Pendant l'instanciation de Segment l'appel est fait aux méthodes calculateExitLine(), calculateVertices et createBounds() qui respectivement calculent et créent les lignes de sortie, les sommets des triangles (le point le plus éloigné) et les bornes du segment. A chaque côté de la ligne de l'itinéraire, les bornes sont créées de la façon suivante : la première ligne va de un bout de la ligne d'entrée (exitLine1) jusqu'au sommet de triangle et de sommet jusqu'au bout de la ligne de sortie (exitLine2), et la même chose de l'autre côté.

ExitLine

Les zones de sortie sont représentées par deux lignes ayant le centre au début et à la fin de la ligne de segment.

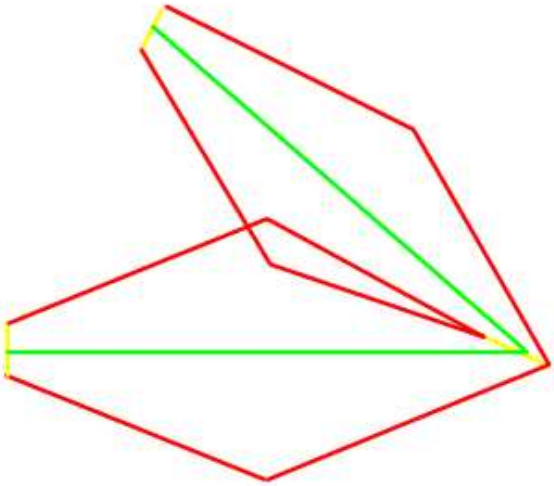
La ligne de sortie entre deux segments est créée de la façon suivante : les deux lignes des segments sont considérées comme deux vecteurs ayant l'origine commune. D'abord l'angle formé par ces deux vecteurs est calculé. La ligne qu'on cherche se trouve sur la droite bissectrice de cet angle. On trouve la droite bissectrice. A la fin, à chaque côté du point d'origine on trouve un point sur la bissectrice éloigné de passWidth depuis l'origine. La ligne déterminée par ces deux point est une ligne de sortie – exitLine.



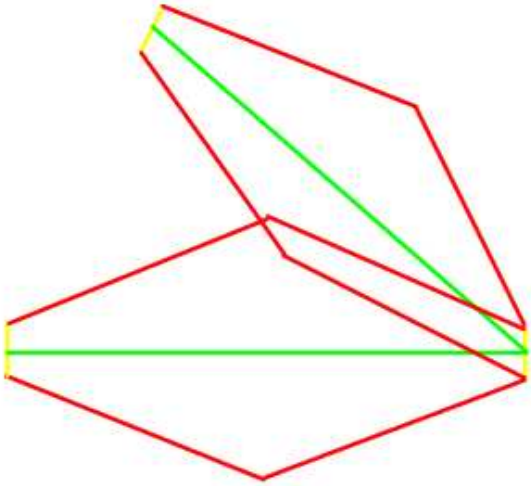
Ils nous restent deux cas spéciaux, qui sont les cas de figures où il s'agit soit du premier, soit du dernier segment. Pour le premier segment, on crée exitLine1 comme la ligne perpendiculaire à la ligne de segment et l'on n'emploie pas la méthode expliquée ci-dessus. Pour le dernier segment, on trouve la première ligne de sortie et on rajoute à la fin exitLine2 perpendiculaire à la ligne de segment.

Pourquoi utilise-t-on cette solution ? Pour la plupart du temps, les lignes de sorties étaient toutes implémentées comme perpendiculaires à la ligne de segment, ce qui est une solution plus simple. Malheureusement dans beaucoup de situations cette approche ne donne pas des bons résultats, ce qui est visible sur les images à la page suivante.

Itinéraire avec la ligne de sortie sur la bissectrice de l'angle.



Itinéraire avec la ligne de sortie perpendiculaire.



Concernant l'implémentation, pour trouver l'angle entre deux vecteurs, on utilise la même procédure que pour la direction de mouvement dans la classe Computation (voir chapitre « Algorithme de suivi »), et on divise l'angle trouvé par deux. Ensuite on cherche la pente k2 de la bissectrice, et on utilise la même méthode que dans `calculateVertices`() pour trouver les points déterminant la ligne de sortie.

ExitLine hérite de la classe Line2D.Double et, à part des coordonnées, contient quelques informations supplémentaires comme le numéro du segment, ses « getters » et « setters », et ses propres constructeurs.

SegmentBounds

Cette classe représente les bornes d'un segment ou, autrement dit, les côtés de notre quasi-rhomboïde, ou ceux des triangles autour de la ligne de segment. Tel que mentionné plus tôt, les bornes s'étendent de la première ligne de sortie (ligne d'entrée) vers le sommet (vertice) au milieu du segment, et jusqu'à la deuxième ligne de sortie. Il en est de même de l'autre cote du chemin. Il s'agit d'un vecteur de quatre lignes (Ligne2D) créé pendant la construction de Segment. Les pentes des lignes décrivant les bornes dépendent de la distance des sommets de la ligne du chemin et qu'on trouve dans la variable maxDistance de la classe Segment. Avec cette valeur on peut alors régler l'angle formé avec la ligne de segment, et par conséquent améliorer la navigation de l'utilisateur de façon qu'on ne le laisse pas trop s'éloigner au début et avant la fin du segment.

Il faut tout de même être prudent avec les valeurs qui sont insérées dans maxDistance car si elles sont trop grandes par rapport à la taille du segment, les sommets sont trop éloignés et on diminue l'efficacité de guidage automatique. Je recommande que la valeur maximale ne dépasse pas trop la moitié de la longueur du segment, qui crée l'angle de 45° entre les lignes des bornes et le segment. Normalement, la distance maximale doit être plus grande ou égale à la moitié de la ligne de sortie, car sinon on aurait une situation contradictoire.
Pour cette raison, j'ai rajouté le contrôle de ces valeurs dans la classe Segment, et si le rapport entre elles n'est pas juste, la valeur de maxDistance est égalée à la moitié de la ligne de sortie.
Le calcul des points vertex1 et vertex2 c'est-à-dire sommets est effectué dans la méthode calculateVertices() de la classe Segment.
Si A, B et M sont respectivement le point de départ, le point final du segment, et le point du milieu de la ligne de segment, et si α est l'angle entre la ligne et l'axe X, on cherche d'abord la pente de la droite qui est perpendiculaire à la ligne et qui la coupe au point M :

$$k = \frac{A_X - B_X}{B_Y - A_Y} = \tan(\alpha)$$

D'où on trouve α :

$$\alpha = \arctan(k)$$

Maintenant on utilise la formule suivante pour calculer les sommets V et W (vertex1 et vertex2 dans le code) où d est la distance maximale :

$$W_x = M_x + d \cdot \cos(\alpha)$$
$$W_y = M_y + d \cdot \sin(\alpha)$$

Le sommet de l'autre côté :

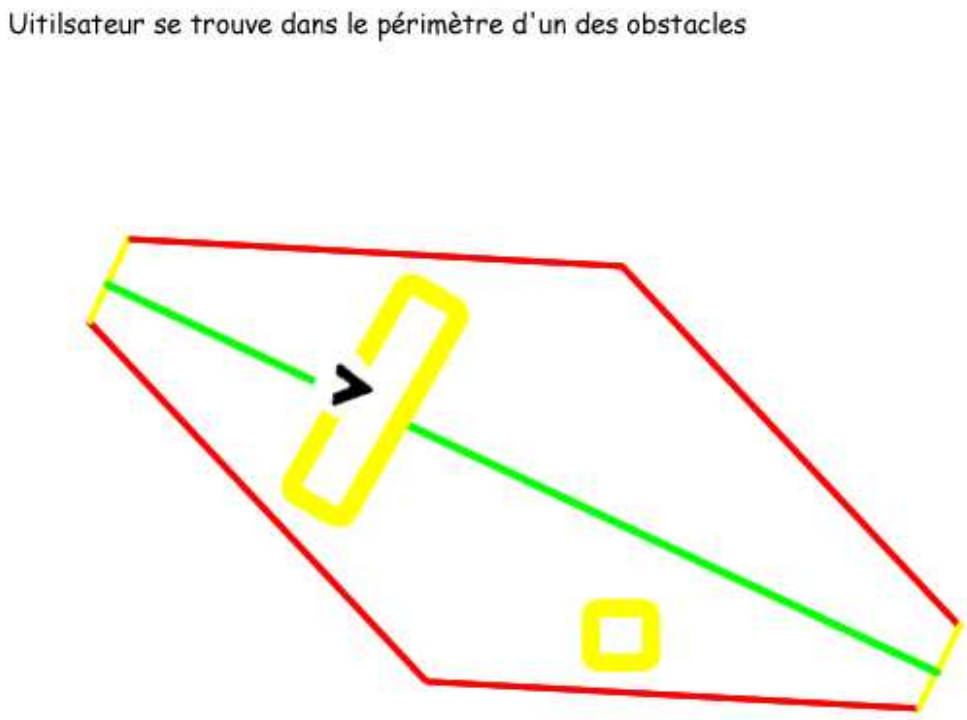
$$V_x = 2 \cdot M_x - W_x$$
$$V_y = 2 \cdot M_y - W_y$$

4.3.3 Obstacles

Pour avoir une implémentation réaliste d'une route sur l'ordinateur, il nous manque encore une couche importante. Nous avons bien défini le chemin et ses bornes, mais il est rare de trouver un itinéraire vide d'éléments qui peuvent ralentir le mouvement. J'ai décidé d'appeler ces objets les obstacles, même s'ils

ne doivent pas toujours avoir forcement le rôle négatif d'obstacles au sens strict du terme.

Les obstacles peuvent être n'importe quels objets se trouvant dans les limites d'un segment. Il peut s'agir d'un objet (une roche, un arbre, les voitures garées etc.) limitant le mouvement d'une personne. Cela exige un changement de la direction. L'autre type d'obstacle est « fonctionnel » ou interactif dans un sens, car il existe des opérations à effectuer afin de dépasser l'obstacle. Il peut s'agir, par exemple, d'un passage piéton où l'utilisateur doit exécuter une série d'instructions fournies par le logiciel afin de traverser la rue sans risques. En fait, un objet Obstacle peut avoir d'autres rôles pas implémentés dans cette version, comme, par exemple, un panneau des renseignements sur l'itinéraire. J'ai implémenté la classe Obstacle de façon très généralisée afin de laisser la liberté aux futurs développeurs de créer les rôles qu'ils souhaitent. Il faut seulement respecter la règle, mentionnée avant, selon laquelle les obstacles ne peuvent pas présenter des endroits très dangereux car pendant la création de l'itinéraire on s'est assuré que l'itinéraire ne contient pas de telles places.



Dans l'application, l'obstacle est représenté avec quatre lignes ayant la forme d'un rectangle qui, en fait, signifient les bornes de l'obstacle. On utilise ces bornes pour savoir si l'utilisateur s'est rapproché de l'obstacle, et pour lui donner les instructions avant qu'il ne le touche. La classe Obstacle contient une variable *riskLevel* qui décrit la dangerosité de l'obstacle. Selon le niveau de la dangerosité on peut changer le message donné à l'utilisateur et, s'il le faut, créer des avertissements. C'est une fonctionnalité qui n'a pas été très exploitée dans cette

version, et qui reste ouverte pour le futur développement. Encore une fois, les obstacles ne peuvent pas êtres des endroits dangereux pour la vie – on ne peut pas laisser un trou profond sur la route, car l'imprécision de la technologie GPS actuelle pourrait coûter la vie à quelqu'un !

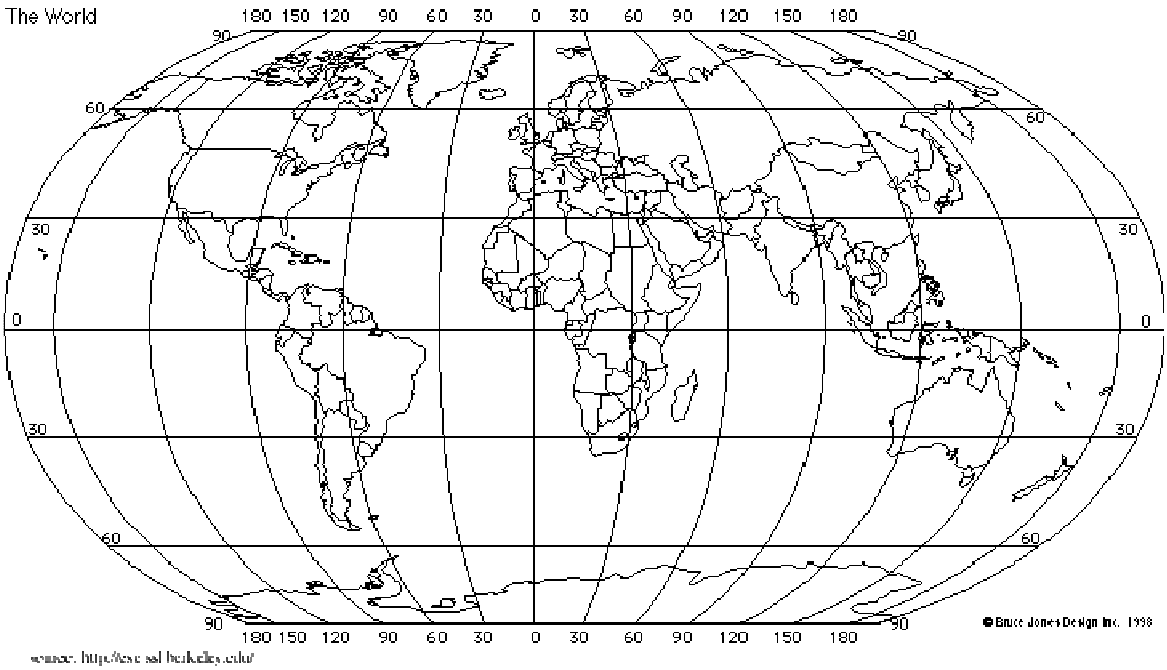
4.4 Coordonnées

Savoir où quelqu'un se trouve c'est savoir ses coordonnées. Dans notre logiciel, tout est calculé par rapport à la position de l'utilisateur, par conséquent les coordonnées ont un rôle crucial pour cette application.
Le système de coordonnées dont on a besoin dépend du système utilisé par l'appareil *Mambo*, des calculs nécessaires pour le suivi, et de la représentation graphique de mouvement en Java.

L'appareil envoie les coordonnées en utilisant le système de longitude et latitude, qui est le système de coordonnées le plus répandu aujourd'hui. Après avoir analysé la future application et après avoir effectué des recherches sur l'Internet, j'ai conclu que le système géographique standard de longitude/latitude ne convient pas à mes besoins. Ce système exige des opérations mathématiques complexes et connaissance détaillée de la cartographie pour le calcul des distances entre des points sur la carte, ce qui est indispensable pour ce type de l'application. Pour les calculs j'ai décidé d'utiliser la projection UTM (Universal Transverse Mercator), qui est le plus souvent utilisé avec les appareils GPS. La conséquence est que les coordonnées longitude/latitude envoyées par *Mambo* doivent être transformées en UTM.

4.4.1 Système longitude/latitude

Ce système, créé par Eratosthène de Cyrène, est le système le plus utilisé pour représenter une position géographique. Les deux composants sont mesurés en degrés, et on utilise plusieurs formats d'écriture. Le plus connu a la forme : degrés, minutes, secondes. Pour être capable d'effectuer des calculs, on transforme les minutes et les secondes en format décimal, par exemple 34,563°. L'autre format utilise par les appareils GPS donne séparément les degrés et les minutes avec la division décimale de secondes (28° 15,345').
La longitude représente l'endroit sur la Terre à l'est ou à l'ouest de méridien, tandis que latitude décrit la position au nord ou au sud de l'équateur.



Voici les définitions de la longitude et de la latitude selon Wikipedia.

Latitude

« La latitude est une mesure angulaire s'étendant de 0° à l'équateur à 90° aux pôles. C'est la distance d'un lieu à l'équateur mesurée en degrés sur le méridien.

La latitude est l'angle que fait la normale à l'ellipsoïde de référence avec le plan équatorial. C'est la latitude de la plupart des cartes. »
[http://fr.wikipedia.org]

Longitude

« Tous les points de même longitude appartiennent à une ligne épousant la courbure terrestre, coupant à angle droit l'équateur et reliant le pôle Nord au pôle Sud, cette ligne est appelée méridien. À la différence de la latitude (position nord-sud) qui bénéficie de l'équateur et des pôles comme références, aucune référence naturelle n'existe pour la longitude. La longitude, généralement notée λ, est donc une mesure angulaire sur 360° par rapport à un méridien de référence, avec une étendue de +180° à -180° ou 180° Est à 180° Ouest. »
[http://fr.wikipedia.org]

4.4.2 UTM

Au début du projet, pendant la phase d'analyse, j'ai consacré une bonne partie de temps à la recherche et à l'étude des systèmes de coordonnées et des fondements de la cartographie. En fait, j'apprenais tout au long du travail car toute la problématique de la navigation fait partie de la géographie appliquée. Même sous ces circonstances, je ne connais que très peu de concepts de ce domaine, et je pense que leur présentation par mes propres mots ne serait pas suffisamment exhaustive et pourrait même créer la confusion chez le lecteur. A cause de cela, je cite surtout les notes de cours de Système d'information géographique trouvées sur le site de l'Université du Québec, et écrites de façon assez compréhensible pour les non-initiés.
Voyons d'abord quelques termes fréquemment utilisés.

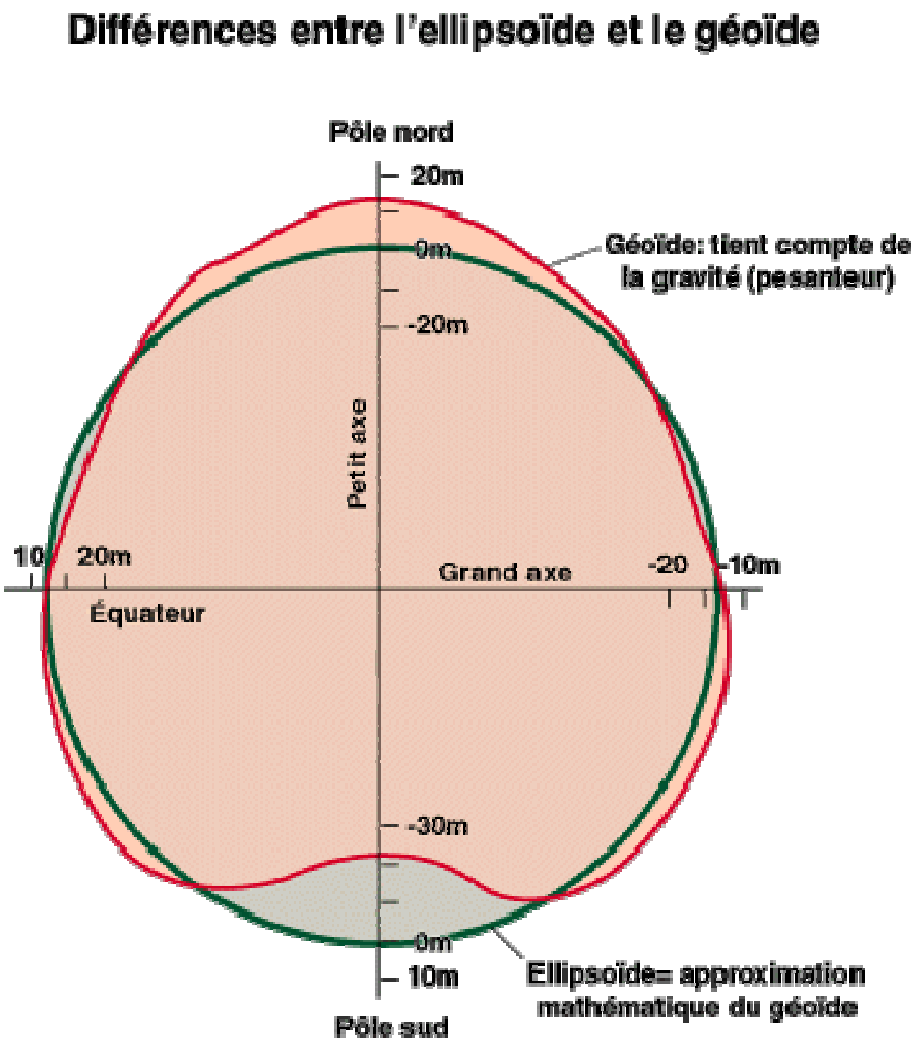
Le géoïde

« Figure mathématique complexe cherchant à reproduire la forme réelle de la Terre, correspondant à la surface moyenne des océans (niveau moyen des mers) et à son prolongement sous les terres émergées. Tenant compte de la force de gravité, le géoïde s'ajuste différemment aux masses d'eau et aux masses continentales. La surface réelle de la Terre s'écarte plus ou moins du géoïde moyen selon les variations locales du relief. » [http://www.unites.uqam.ca]

L'ellipsoïde

« C'est une approximation mathématique du géoïde définie à partir de mesures géodésiques et de la différence entre le rayon équatorial moyen (grand axe de l'ellipsoïde) et le rayon polaire de la terre (petit axe de l'ellipsoïde). » [http://www.unites.uqam.ca]

Dans notre cas on utilise l'ellipsoïde du système géodésique WGS84 (World Geodetic System).



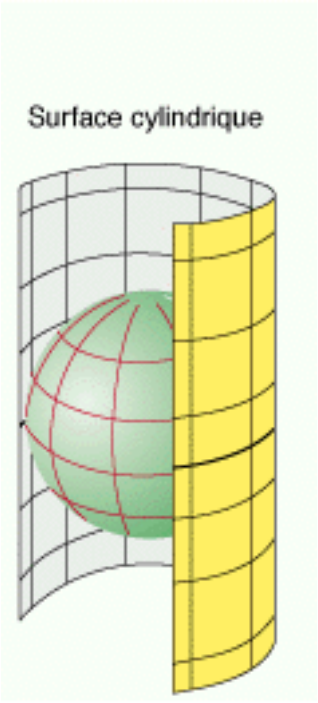
Source : [illegible]

Le système de projection

« L'espace géographique est un espace courbe matérialisé par l'ellipsoïde de référence du géoïde.
Pour passer de l'ellipsoïde à une carte dessinée sur un plan, il est important d'établir une correspondance, la plus fidèle possible, entre les points de l'ellipsoïde et ceux du plan. Ce système de correspondance s'appelle le système de projection. » [http://www.unites.uqam.ca]

Projection cylindrique

« On projette l'ellipsoïde sur un cylindre qui l'englobe. Celui-ci peut être tangent au grand cercle, ou sécant en deux cercles. Puis on déroule le cylindre pour obtenir la carte.



www.unites.uqam.ca

Exemples de projection cylindrique :

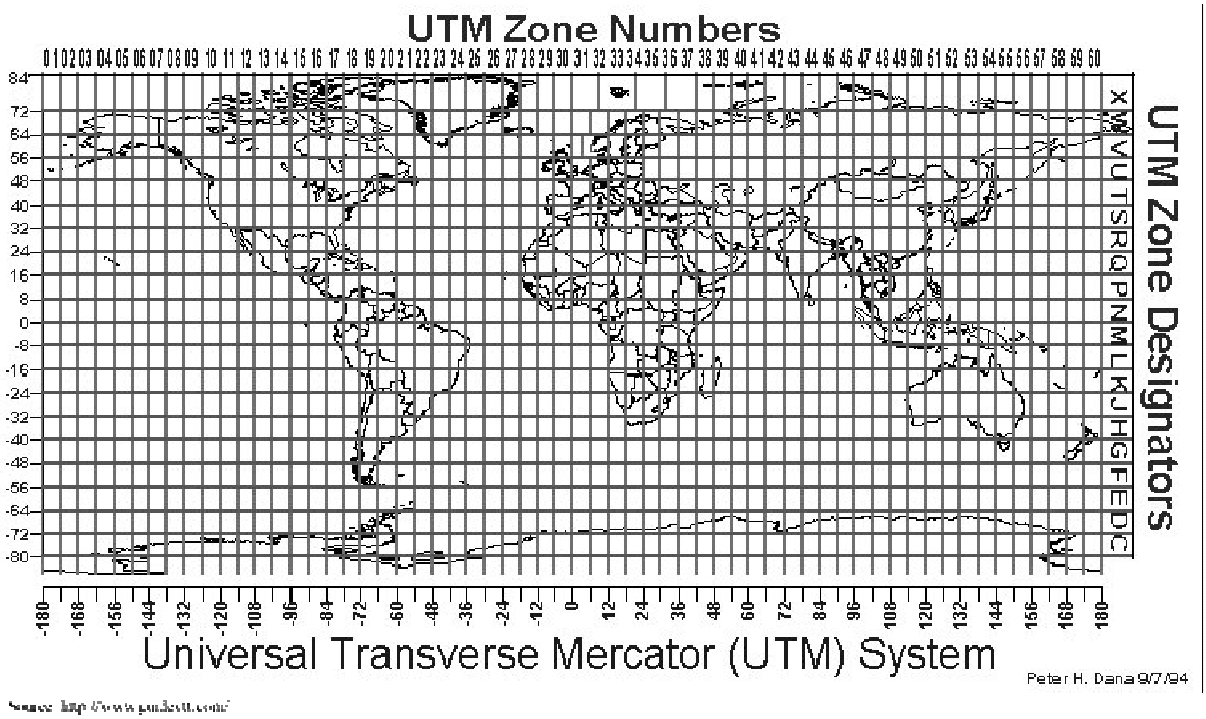
- Projection de Mercator (conforme)
- Projection de Peters (équivalente)
- Projection de Robinson (pseudo-cylindrique, aphylactique)
- Projection UTM aussi appelée Gauss-Kruger (conforme) »

[http://fr.wikipedia.org]

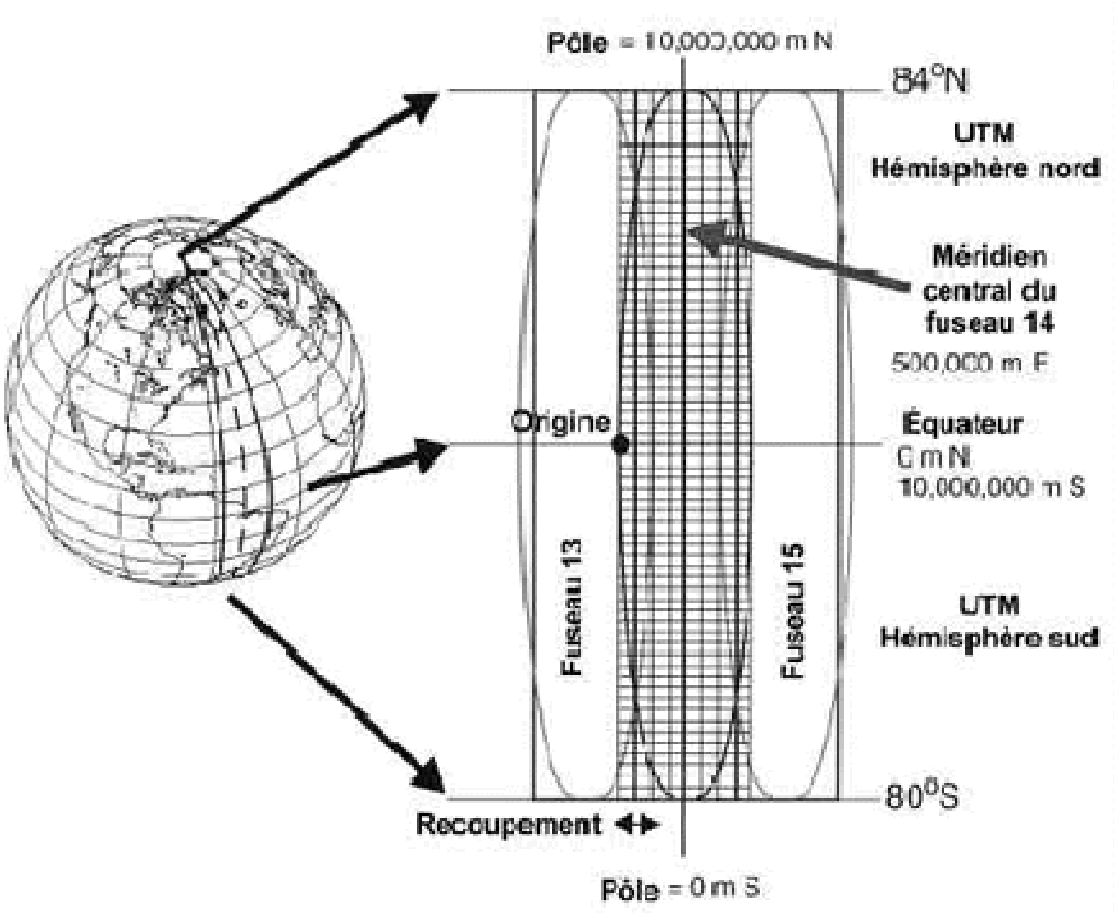
Projection UTM (Universal Transverse Mercator)

« La **Transverse universelle de Mercator** (en anglais Universal Transverse Mercator ou UTM) est un type de projection conforme de la surface de la Terre. Cette projection est une projection cylindrique où l'axe du cylindre croise perpendiculairement l'axe des pôles de l'ellipsoïde terrestre au centre de l'ellipsoïde.
L'UTM est également un système de référence géospatiale permettant d'identifier n'importe quel point sur notre planète.
En pratique, pour couvrir la surface de la Terre, on la découpe en 60 fuseaux de 6 degrés en séparant l'hémisphère Nord et l'hémisphère Sud. Soit au total 120 zones (60 pour le Nord et 60 pour le Sud). On développe alors le cylindre tangent à l'ellipsoïde le long d'un méridien pour obtenir une représentation plane. »
[http://fr.wikipedia.org]

Méridien central d'un fuseau (zone) est le méridien passant au milieu du fuseau et le découpant en deux moitiés.



Comment utilise-t-on les coordonnées du système UTM ? Dans UTM on peut présenter chaque position comme une paire (x, y) mesurée en mètres ce qui paraît beaucoup plus naturel et intuitif que le système de longitude et latitude. Les composants x et y sont appelés respectivement Easting et Northing. Comme l'origine du système de coordonnées, on prend le point d'intersection du méridien central et la ligne de l'équateur. Le méridien central passe au milieu de la zone, et par la suite, on aurait des valeurs x négatives ce qui est la situation qu'on veut éviter. Pour résoudre ce problème, on prend pour l'axe y une ligne imaginaire qui se trouve 500'000 m à l'ouest du méridien central ce qu'on appelle False Easting.



Source: http://www.unites.uqam.ca/dgeo/geo7511-2001/

En utilisant UTM, les coordonnées sont basées sur le système décimal, ce qui facilite les calculs par rapport au longitude/latitude où l'on doit transformer les minutes et les secondes en système décimal. Un autre avantage très important est le fait que les positions sont mesurées en mètres dans un espace bidimensionnel, ce qui réduit les calculs de distance à l'implémentation simple du théorème de Pythagore au lieu des calculs trigonométriques complexes, indispensables dans le système des coordonnées géographiques de longitude et latitude. De plus, UTM et très souvent utilisé par les appareils GPS.

Références : http://fr.wikipedia.org ; Université de Québec. *Projections, systèmes de coordonnées*. http://www.unites.uqam.ca/dgeo/geo7511-2001/htm/section6.htm ; Dr. Sergei ANDRONIKOV. *Introduction to GIS lecture notes*. http://geog.gmu.edu/PEOPLE/fall2004/Sergei/GEOG311.htm ; Don BARTLETT. *A Practical Guide to GPS – UTM*. *http://www.dbartlett.com* ;

4.5 Algorithme de suivi

Ce chapitre décrit la partie centrale du projet – celle du suivi et de la navigation d'un utilisateur du système, implémentée dans la classe Computation. A continuation, chaque cas est analysé séparément et son implémentation est exposée.

A la base, nous avons la situation où une personne marche depuis le point de départ D vers le point final F, tout en respectant les contraintes comme les obstacles ou les limites de l'itinéraire. Comme il s'agit d'une personne malvoyante qui ne connaît pas cet itinéraire, le logiciel doit « savoir » reconnaître le mouvement de l'utilisateur, décider si le mouvement est valable, et préparer les instructions adéquates pour l'utilisateur. En prenant tous les cas possibles de mouvement dans un segment de la route, l'algorithme devrait avoir une forme similaire à celle-ci :

Si la ligne de sortie atteinte
    Passage au nouveau segment
    Message informatif envoyé à l'utilisateur.

Si l'utilisateur se trouve près d'un obstacle
    Envoyer les instructions liées à cet obstacle.

Si les bornes du segment ont été franchies
    Envoyer un avertissement et demander le retour à l'intérieur des bornes.

Si l'utilisateur ne va pas à la bonne direction – il s'éloigne du point final
    Avertir.
    Calculer la bonne direction et communiquer la à l'utilisateur.

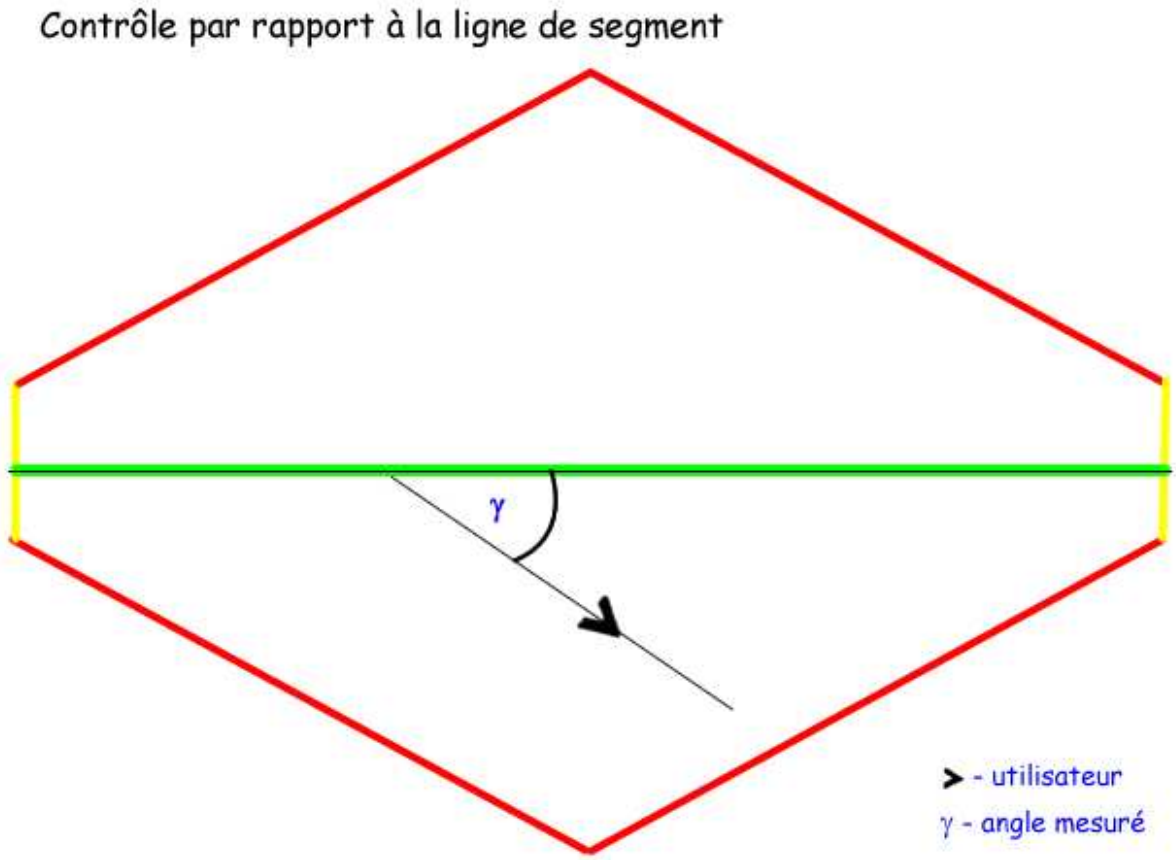
Les instructions pour l'utilisateur et les données liées à sa position courante sont placées dans l'instance de la classe Infos. On y trouve des informations utiles comme sa position par rapport aux bornes et aux obstacles. Les informations d'Info peuvent être lues et traitées par les autres classes de l'application.
Concernant la liaison avec le reste de l'application, la classe Computation contient la méthode publique verifyMove() qui effectue l'algorithme de suivi.

Voyons maintenant les détails de tous les cas trouvés dans l'algorithme.

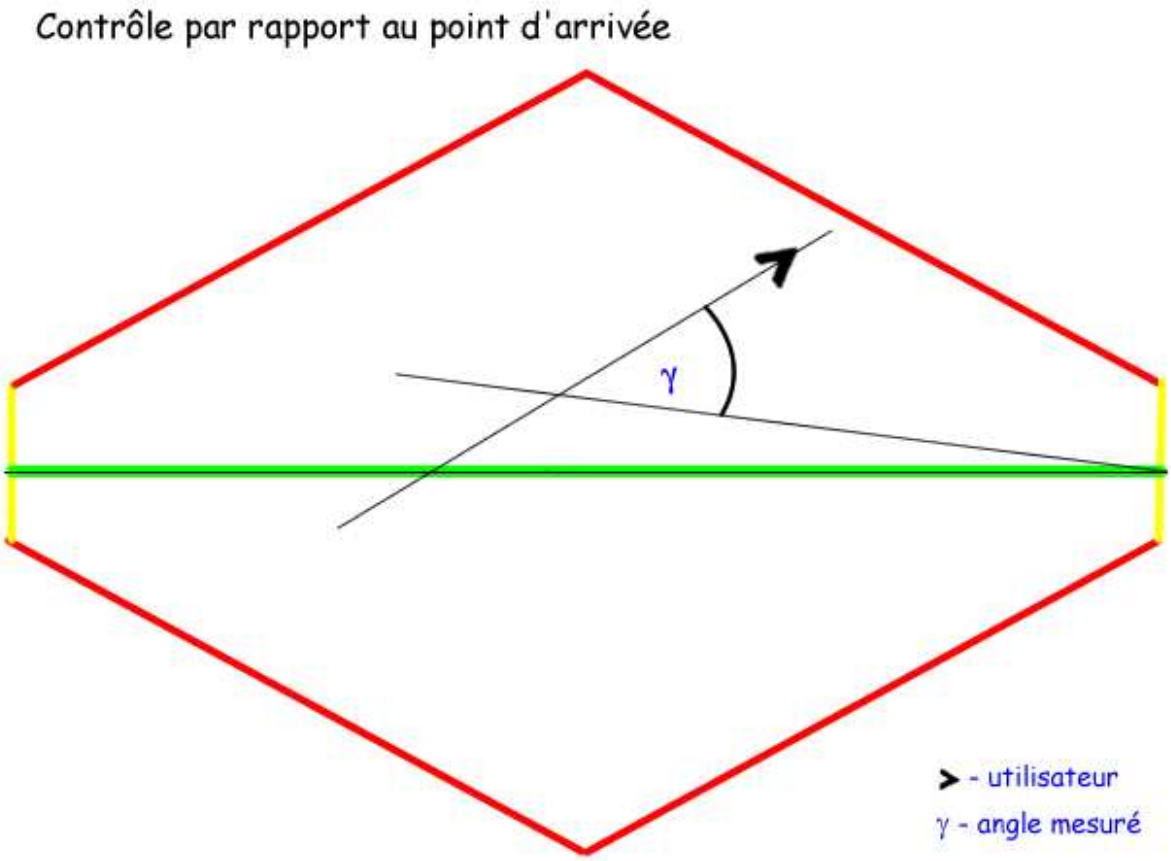
4.5.1 Direction de mouvement

Pendant l'analyse, la première question qui s'est posée était comment savoir si l'utilisateur avance vers le point final ou s'il s'éloigne. A chaque pas d'utilisateur, on garde la position ancienne et la position courante ce qui donne la droite de direction de mouvement, mais comment savoir si elle est valable ? J'ai envisagé plusieurs solutions, certaines ne pouvant pas être considérées à cause de leur

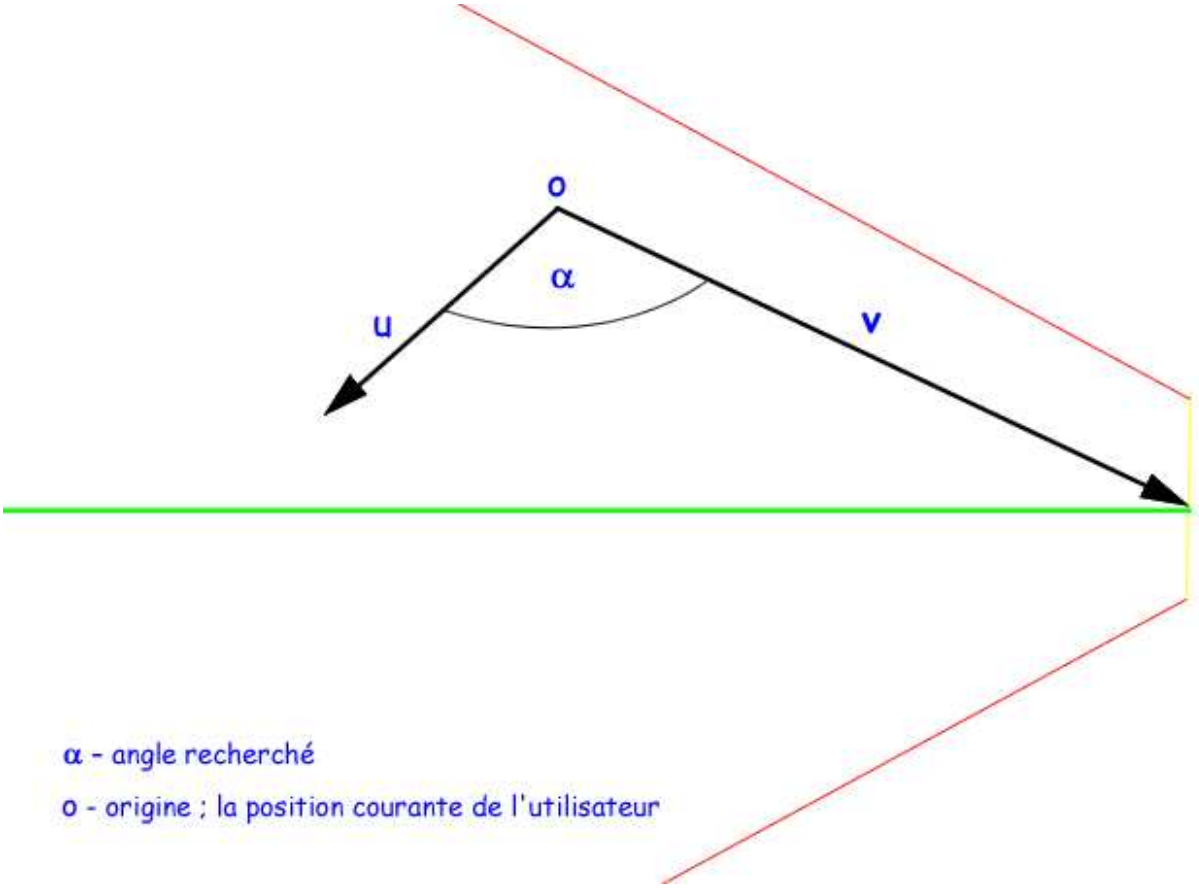
complexité. La solution retenue, qui au même temps garde une certaine simplicité, est de mesurer l'angle formé par la ligne de segment et par la droite de mouvement de l'utilisateur. Si l'angle est plus grand que la valeur prévue, alors ce dernier ne marche pas vers la bonne direction.



La solution semble bonne, mais est aussi rétrograde dans un sens car on cherche la direction par rapport aux points antérieurs, au lieu de se repérer vers le point final, ce qui serait plus logique et plus précis. Heureusement, il est possible de résoudre ce problème tout en gardant l'utilisation de l'angle : on cherche l'angle entre la droite de mouvement, et de la droite déterminée par le point final et le point de position de l'utilisateur. Une fois l'angle trouvé, le principe reste le même que pour la première solution (comparaison avec la valeur prédéfinie). De cette manière, on s'oriente toujours par rapport à l'arrivée, ce qui donne une meilleure précision.



Maintenant on peut voir comment trouver si la direction de mouvement est valide. La meilleure façon de résoudre ce problème serait le calcul vectoriel, donc on va traiter la ligne de mouvement et l'autre ligne significative comme les deux vecteurs u et v, ayant l'origine à la position courante de l'utilisateur. Ces vecteurs forment l'angle recherché α.



On voit que u est déterminé par le point précédent de l'utilisateur, et v par le point final. D'abord il faut trouver le produit scalaire de deux vecteurs :

$$u \cdot v = u_x \cdot v_x + u_y \cdot v_y$$

Ensuite, on cherche l'intensité de chaque des deux vecteurs :

$$\|u\| = \sqrt{u_x{}^2 + u_y{}^2}$$

$$\|v\| = \sqrt{v_x{}^2 + v_y{}^2}$$

Maintenant on utilise l'autre formule de produit scalaire afin de trouver l'angle α :

$$u \cdot v = \|u\| \cdot \|v\| \cdot \cos\alpha \quad \Rightarrow \quad \cos\alpha = \frac{u \cdot v}{\|u\| \cdot \|v\|}$$

Comme on connaît tous le composants de cette équation, le calcul de α devient :

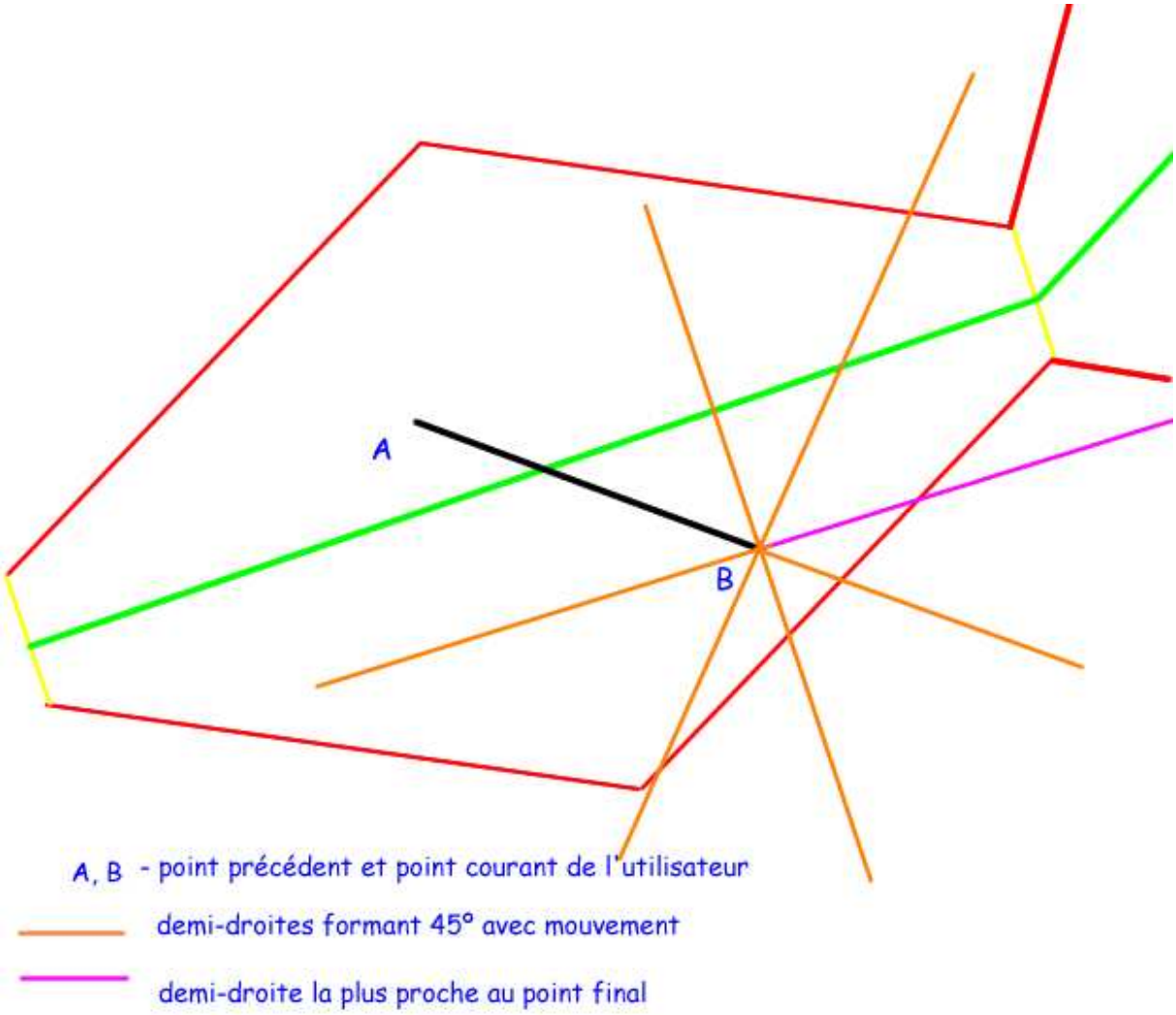
$$\alpha = \arccos\left(\frac{u_x \cdot v_x + u_y \cdot v_y}{\sqrt{\left(u_x{}^2 + u_y{}^2\right) \cdot \left(v_x{}^2 + v_y{}^2\right)}}\right)$$

Quand α est connu, il est possible de le comparer avec l'angle prédéfini et, dans le cas où α est plus grand, conclure que l'utilisateur ne va pas dans la bonne direction.

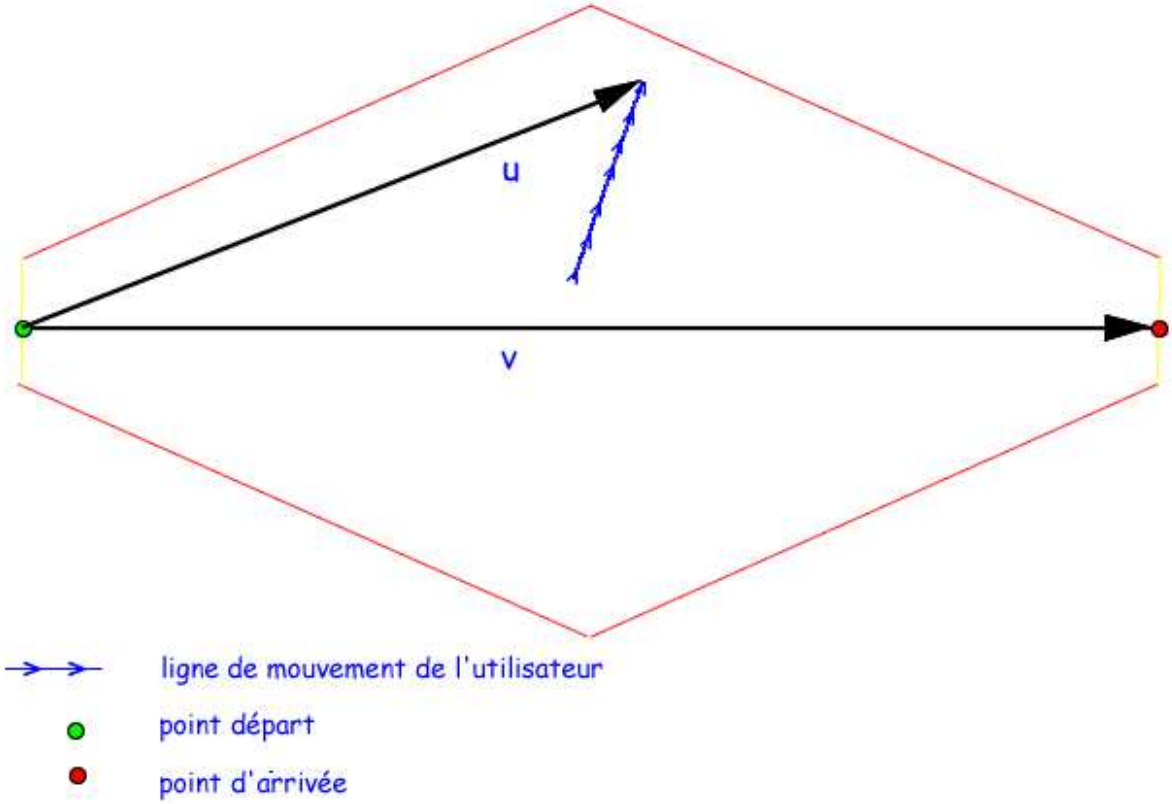
4.5.2 Redirection de l'utilisateur

S'il s'agissait ici d'un programme de guidage pour des machines informatisées, le problème de la redirection serait déjà résolu – on donnerait simplement l'instruction à l'appareil de tourner de α degrés précisément, et de continuer marcher vers le point final du segment.
Malheureusement on ne peut pas appliquer une telle solution quand il s'agit du suivi des humains. On ne peut pas dire à une personne de tourner de 112.49 degrés, mais on doit utiliser les unités de mensuration généralement connues. Normalement, chacun peut comprendre et effectuer l'Instruction qui dit qu'il faut tourner de un quart du tour ou d'un moitié du tour. En utilisant ces unités l'approximation de l'utilisateur sera plus précise que d'essayer de trouver l'angle de 112.49 degrés. A cause de cela on va considérer plusieurs directions possibles et chercher celle qui est la plus proche au point final du segment.

Comment implémenter cette solution ? Prenons 45° (ou $\frac{\pi}{4}$, ou un huitième du tour) comme l'unité de virage. La plupart du temps du développement j'ai travaillé avec cette valeur et je l'ai utilisée pour préparer les messages pour l'utilisateur. J'ai laissé la possibilité d'utiliser d'autres valeurs en définissant la valeur de turnAngleUnit de la classe Route et en ajoutant le mot approprié dans le tableau UNITS_DEG[] d'AppConfig. Ensuite, dans Computation, on rempli anglesVector[] avec les multiples de 45 : 45, 90, 135, 180. La valeur d'un virage peut être 180 degrés au maximum, car l'algorithme prend en compte l'orientation gauche-droite ce qui donne comme le « pire cas » l'instruction de marcher dans le sens opposé. Maintenant les lignes partant de la position courante sous les angles de anglesVector[] sont construites, et on cherche celle qui, prolongée à l'infini de chaque côté, coupe la ligne de sortie le plus près au point final.



L'instruction n'est pas encore prête car il faut dire de quel côté tourner, gauche ou droite. Ce problème s'est montré plus difficile qu'il ne le semblait au début. J'ai trouvé qu'il faut tenir compte de deux facteurs : le sens de mouvement de l'utilisateur par rapport à la ligne de segment (approchement ou éloignement) et au point final, et l'orientation du segment qui peut être plutôt de gauche à droite ou plutôt de droite à gauche. On calcule le sens de mouvement de l'utilisateur en comparant les distances entre l'objet de repérage (point final, ligne de segment) et les points définissant la ligne de direction (courant et précédent). Le sens dépend également de l'orientation du segment. Finalement, pour trouver le coté où il faut tourner, on calcule la déterminante de deux vecteurs ayant l'origine au point de départ. Ces vecteurs sont la ligne de segment et le vecteur allant jusqu'au point courant de l'utilisateur, nommés v et u respectivement.
Dans la situation illustrée sur l'image ci-dessous, l'utilisateur est instruit de tourner à droite.



$$\det(u,v) = (D_y - v_y)\cdot u_x + (v_x - D_x)\cdot u_y + D_x \cdot v_y - v_x \cdot D_y$$

Si $\det(u,v) > 0 \Rightarrow$ point final se trouve à gauche.
Si $\det(u,v) > 0 \Rightarrow$ point final se trouve à droite.

Le cas où la déterminante est égale à zéro a été éliminé avant de commencer les calculs.
Enfin, en peut placer dans Infos l'instruction précise pour l'utilisateur.

Une des lignes de sortie a été atteinte

La méthode exitLineReached() cherche si la ligne de mouvement croise une des lignes de sortie. Si oui, il faut trouver laquelle – celle de l'entrée ou celle d'arrive. Dans le cas le plus commun, où la ligne de l'arrivée a été atteinte, on change simplement la valeur du segment courant dans l'instance de Route utilisée et on passe comme cela le contrôle au segment prochain. Pour la ligne d'entrée dépassée, on passe le contrôle au segment précédent. Si l'utilisateur dépasse la dernière ligne d'arrivée, on lui communique que le parcours de l'itinéraire est fini.

L'utilisateur se trouve près d'un obstacle

Comme cela a été expliqué auparavant, dans l'application, l'utilisateur ne touche pas l'obstacle lui-même, mais il entre dans le périmètre prédéfini de l'obstacle.
Dans la classe Computation, on utilise le vecteur inObstacleTab[] contenant des booléens initialisés à false, qui représente tous les obstacles du segment courant. Chaque fois que l'utilisateur se trouve près d'un obstacle (dans ses bornes), on met le booléen adéquat à true. En utilisant cette approche, on évite quelques situations inhabituelles qui demandent le traitement spécial, comme celle où l'utilisateur en même temps sort d'un obstacle et entre dans un autre, ou quand il entre et sort de l'obstacle d'un coup.
Chaque Obstacle différent contient son propre message qui peut être une instruction (« Attendez le feu vert. »,..) ou un avertissement (« Attention, pas de passage piéton »,…). C'est le message qui sera communiqué à l'utilisateur.

L'utilisateur a franchi les bornes du segment

Pour trouver si l'utilisateur de trouve en dedans des limites du segment, on simplement si la ligne de son mouvement croise une des bornes. Si oui, le message avertissant l'utilisateur que le logiciel ne suit plus son mouvement est placé dans Infos.

4.5.3 Filtre de mouvement

La technologie GPS ne permet pas de créer une application de haute précision. Dans certaines situations, il peut même arriver que le logiciel soit complètement inutile à cause de GPS. Malgré cela, j'ai essayé d'atténuer certains défauts et de donner l'impression d'une meilleure précision de l'appareil GPS. Filtre de mouvement fait partie de ces améliorations.

On peut bien voir le problème de l'imprécision quand l'utilisateur reste sans bouger sur la même position. Comme les coordonnées de sa position ne seront jamais exactes, les données envoyées par GPS vont montrer les petits mouvements autour de sa position ce qui va donner l'impression que la personne continue de marcher de façon incompréhensible. Pour remédier à ce problème, j'ai rajouté un contrôle, effectué avant les autres calculs, de mouvement minimal. Disons que la valeur minimale de mouvement est de 2 mètres. Alors, on va mesurer la longueur de la ligne entre le point précédent et le point courant (ligne de mouvement), et si elle est inférieure a 2 mètres, le mouvement n'est pas considéré comme valable et la méthode verifyMove() fini son exécution sans effectuer les autres calculs de mouvement.
Le même principe est appliqué pour la valeur de pas maximal. Si la distance entre deux points est plus grande que la valeur maximale permise, le mouvement n'est pas pris en compte.
Les valeurs des filtres de mouvement peuvent être configurées dans le menu « Options » de la fenêtre principale.

4.6 Logiciels externes utilisés

4.6.1 ArcGIS

Une des premières fonctionnalités demandée par le client était la compatibilité avec SIG. Cela signifie que la représentation informatique de l'itinéraire doit être conforme aux principes de SIG (éléments, présentation, etc.) et que la possibilité de rajout des nouveaux itinéraires et des composants doit être offerte pour le développement futur de l'application. En fait, la compatibilité avec les formats populaires de SIG satisferait la plupart de ces exigences.

Dès les premiers jours de travail, je me suis mis à la recherche et à l'étude des logiciels existants. J'ai testé les logiciels SIG comme *MapInfo*, *JShape* et *OziExplorer*, qui affichent les cartes et qui permettent leur modification visuelle. Certains parmi eux sont gratuits ou libres, comme *JShape*, et ceux qui sont payants ne représentent pas la plus haute classe et ils ne sont pas très chers.
Certains parmi eux sont de bonne qualité et convenaient bien pour mes besoins, plutôt modestes.

Le seul logiciel que j'ai identifié comme « le haut de gamme » dont le prix est assez élevé, était le ArcGIS de ESRI. Ce paquet offre les applications nécessaires et suffisantes pour la création des SIG, et peut être utilisé pour la réalisation des très grands projets. Normalement, je n'avais pas de besoin d'une application d'une telle complexité car il était prévu que je crée un itinéraire simple avec quelques types d'objets. Ce qui m'a attiré vers cette application est que j'ai trouvé qu'elle représente le logiciel le plus répandu parmi les entreprises dans les domaines de la navigation GPS, de la géographie et de la cartographie informatisées, donc les éléments de SIG. Encore plus important est le fait que les concepts et le format des fichiers de ArcGIS représentent le standard sur le marché des logiciels GIS. Ce type de fichiers standard créé par ESRI s'appelle Shapefile (fichier des formes), et il est souvent utilisé par d'autres logiciels SIG. Alors, l'itinéraire créé avec ArcGIS serait très réutilisable à cause de son format standardisé et le fait que la plupart des entreprises travaillant avec SIG utilisent ce logiciel.

Format Shapefile

« Le Shapefile, ou "fichier de formes" est un format de fichier issu du monde des Systèmes d'Informations Géographiques (ou SIG). Initialement développé par ESRI pour ses logiciels commerciaux, ce format est désormais devenu un standard de facto, et largement utilisé par un grand nombre de logiciels libres (MapServer, Grass, Udig, MapGuide_OpenSource ...) comme propriétaires.

Il contient toute l'information liée à la géométrie des objets décrits, qui peuvent être :

- des points
- des lignes
- des polygones

Son extension est classiquement SHP, et il est toujours accompagné de deux autres fichiers de même nom, et d'extensions :

- un fichier DBF, qui contient les données attributaires relatives aux objets contenus dans le Shapefile
- un fichier SHX, qui stocke l'index de la géométrie »

[http://fr.wikipedia.org]

Pour plus de détails, voir le document « ESRI Shapefile Technical Description » à l'adresse http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf

Travail avec le format Shapefile

On peut voir ici le concept d'utilisation de Shapefile. Les instructions détaillées de travail avec Shapefile se trouvent dans le Guide d'utilisation, en annexe de ce document.

Création avec ArcGIS

Il existe plusieurs pas nécessaires pour la création d'un fichier Shapefile. D'abord il faut définir sa structure dans ArcCatalog. Ensuite, il faut choisir le type de la forme qui peut être : Point, Polyline, Polygon, MultiPoint, MultiPatch. On choisi également le système de coordonnées pour chaque Shapefile. Un Shapefile déterminé par son triplet des fichiers (shp, shx et dbf) ne représente pas un seul objet, mais un ensemble de tous les objets de même type. On aura donc un Shapefile pour tous les polygons, un autre pour tous les lignes, etc. On a la possibilité de rajouter des attributs à chaque Shapefile. Ils peuvent avoir des valeurs numériques ou textuelles. Chaque objet de ce Shapefile aura les attributs auxquels on peut affecter des valeurs.

Pour visualiser le Shapefile et créer ses objets, on utilise ArcMap. S'il y a plusieurs Shapefiles, ils sont représentés comme des couches différentes de une mappe créée avec ArcMap.

4.6.2 GeoTools

Une des conditions pour l'utilisation de Shapefile était la possibilité de lire les valeurs dans les fichiers depuis mon application en Java. Il est possible de créer sa propre solution pour la lecture des fichiers de triplet Shapefile car sa structure et ses spécifications sont ouvertes (voir « ESRI Shapefile Technical Description »), mais cela prendrait beaucoup de temps que je ne pouvait pas consacrer à cette partie du projet. Après avoir effectué des recherches sur l'Internet, j'ai trouvé qu'une telle solution existe déjà. Il s'agit de GeoTools. Ce n'est pas une application, mais un ensemble de librairies Java qui offrent des méthodes

conformes aux standards pour la manipulation des données géomatiques (geospatial data) comme, par exemple, l'implémentation de SIG.

Les librairies GeoTools implémentent les spécifications d'OGC (Open Geospatial Consortium) et collaborent avec les projets GeoAPI et GeoWidgets. GeoTools est développé sous la licence LGPL (Gnu Lesser General Public License). Les librairies GeoTools peuvent être utilisées pour le développement de notre propre logiciel SIG. Dans le cadre de ce travail, elles ne sont nécessaires que pour la lecture des fichiers Shapefile.

Pour la lecture des fichier Shapefile, j'ai créé la classe LoadFromShapeFile qui contient deux méthodes loadSegments() et loadObstacles(). On y utilise les classes ShapefileDataStore, FeatureSource, Feature et FeatureCollection de GeoTools. On récupère les objets en traversant tout les fichiers et on compare les noms des attributs connus avec ceux trouvés dans les fichiers.

4.6.3 JTS

Java Topology Suite de Vivid Solutions est une API des fonctions spatiales 2D, et il est utilisé pour le développement des applications qui travail avec des spatial datasets. JTS implémente les spécifications OpenGIS (http://www.opengeospatial.org) sous la licence LGPL. Certaines méthodes de GeoTools font appel aux classes de JTS comme Coordinate, MultiLineString et Polygon. Les librairies ne viennent pas ensemble avec GeoTools, donc il faut les télécharger séparément depuis le site de Vivid Solutions.

4.6.4 JScience

C'est une librairie compréhensive pour Java prévu pour l'utilisation par la communauté scientifique. La librairie est open-source sous la licence BSD, et chacun peut essayer de l'améliorer. L'idée des créateurs est de créer une synergie entre toutes les sciences, naturelles et sociales, en les intégrant dans la même architecture. Les possibilités de la librairie son assez vastes, mais dans le cadre de ce projet, je ne l'ai utilisée que pour la transformation des coordonnées longitude / latitude reçues de *Mambo*, vers les coordonnées UTM utilisées pour les calculs.

Units

On trouve ici les classes et des constantes liées aux mesures. Elles sont étroitement liées à JScience.

Références :

« ESRI Shapefile Technical Description »
http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf ; Site de GeoTools http://geotools.codehaus.org ; Vivid Solutions Java Topology Suite http://www.vividsolutions.com/jts/jtshome.htm ; JScience http://jscience.org ;

Les logiciels testés : MapInfo Professional http://www.mapinfo.com ; JShape http://www.jshape.com ; OziExplorer http://www.oziexplorer.com ;

4.7 Récupération et transformation des coordonnées

Les données de chaque pas réalisé par l'utilisateur, sont enregistrées dans les fichiers XML, sur le serveur de Novasys.
Nous avons vu précédemment le processus de l'envoi des données de l'appareil Mambo vers Dispatcher - leur destination finale. Dans ce chapitre je vais l'expliquer plus en détail. Il faut mentionner que la connexion et la transformation se passent principalement dans la méthode getData() de la classe ServerConnection.

4.7.1 Fichier de position

L'appareil *Mambo* est configuré de façon à utiliser la norme GPRS pour envoyer les données vers le serveur de Novasys. La configuration comprend, entre autres, l'adresse IP du serveur, son port et le chemin de service web nommé MessageService dans notre cas. Du côté Novasys, les données envoyées par *Mambo* sont extraites et traitées. Par exemple les coordonnées, où la longitude et la latitude sont mises en format plus standard (degrés décimaux). Les informations concernant la position courante sont ensuite sauvegardées dans le fichier XML. Pour chaque envoi de données deux nouveaux fichiers sont créés. A partir des coordonnées de la position, le premier fichier nommé DEVIN contient également des autres informations comme la vitesse et les données géographiques variées, ce qui peut être utile dans le développement futur de l'application. Le deuxième fichier créé, ADMIN, contient les informations concernant *Mambo* ce qui n'a pas d'importance pour Dispatcher.

```
<?xml version="1.0" standalone="yes" ?>
 <DocumentElement>
     <Position>
       <IdPosition>f9007517-e9f9-4304-945e-f3997295b1ca</IdPosition>
       <PositionDateTime>1980-01-06T01:01:19+01:00</PositionDateTime>
             <Valid>false</Valid>
             <Longitude>46.2000</Longitude>
             <EastWest>E</EastWest>
             <Latitude>6.15000</Latitude>
             <NorthSouth>N</NorthSouth>
             <SpeedMSec>0</SpeedMSec>
             <SpeedKmH>0</SpeedKmH>
             <SpeedMpH>0</SpeedMpH>
             <SpeedKnots>0</SpeedKnots>
             <AltitudeAboveSeaLevel>0</AltitudeAboveSeaLevel>
       <AltitudeAboveGroundLevel>0</AltitudeAboveGroundLevel>
           <Heading>0</Heading>
           <Odometer>0</Odometer>
           <MagneticVariation>0</MagneticVariation>
           <MagneticVariationEastWest></MagneticVariationEastWest>
```

```
        <NumberOfSatellitesUsed>0</NumberOfSatellitesUsed>
    </Position>
</DocumentElement>
```

4.7.2 Utilisation du service web MessageService

Pour récupérer le fichier de la position, il faut faire appel au service web de Novasys qui, après l'identification, fourni les données demandées. Il s'agit de la « consommation » d'un service web .NET à partit de l'application Java. Cette connexion avec Novasys m'a posé beaucoup de problèmes, et j'ai consacré une grande partie du temps à sa réalisation. Le plus grand souci est qu'il existe beaucoup d'inconnues dans ce processus et qu'il est difficile de situer où est l'erreur. Par exemple, le problème peut venir à cause des mauvais paramètres de connexion, de fausse configuration de Apache Axis ou des problèmes avec le parseur, mais l'erreur peut exister aussi du côté de serveur (ce qui s'est avéré juste parfois).

Pour utiliser correctement un service web, il faut connaître son interface, qu'on trouve dans le fichier WSDL et qui est MessageService.wsdl dans notre cas. WSDL est le langage qui décrit l'interface d'un service web. J'ai essayé d'abord de réaliser manuellement la communication avec MessageService, mais j'ai échoué après plusieurs essais. Sur les différents sites et forums, j'ai appris qu'il existe un outil Apache Axis, ou seulement Axis, qui automatise le processus de l'exploitation d'un service web. A partir de fichier WSDL, Axis va créer toutes les classes Java de côté client nécessaires à la consommation de service. Apache Axis est en fait un package Java utilisé pour la création de services web, sérialisation/désérialisation des classes Java est d'outils différents des services web.
« Axis est à la fois un environnement d'hébergement de services web, et un toolkit complet de développement pour la création de services et l'accès à des services tiers. » [http://www-igm.univ-mlv.fr/~dr/XPOSE2003/axis_seng]

L'outil que j'ai utilisé pour créer des classes Java s'appelle wsdl2java, tandis que pour l'opération inverse on emploie java2wsdl. Parmi les classes créées, on en utilise que deux directement dans le code, les autres étant appelées intérieurement par ces deux classes. La classe MessageServiceLocator s'occupe de la connexion avec MessageService, et MessageServiceSoap est utilisée comme l'interface vers le service. Maintenant on peut appeler les méthodes de service web comme une méthode de MessageServiceSoap en passant comme des arguments le nom de l'utilisateur et le mot de passe (IMEI de Mambo). Le résultat de l'appel de la méthode getMessage() est une instance de la classe avec long nom GetMessageResponseGetMessageResult.

Récupération des coordonnées

Comme la réponse attendue est en format XML, il faut choisir le parseur adéquat. Deux possibilités habituelles sont le parseur DOM et le parseur SAX. La première décision était d'utiliser DOM à cause de la petite taille de fichier à analyser. J'ai commencé l'implémentation à l'aide des tutoriaux trouvé sur l'Internet. Bientôt, le

travail est devenu trop complexe pour la très petite quantité d'information recherchée. A la fin, j'ai trouvé qu'il est plus adéquat de créer un micro parseur de quelques lignes de code. L'analyse fonctionne de la manière suivante : le résultat récupéré antérieurement est d'abord transformé en String avec casting. On cherche les valeurs des variables <Longitude> et <Latitude> simplement en utilisant les méthodes de String indexOf() et substring(). Si les cordonnées sont trouvées, il faut les transformer.

Transformation des coordonnées

Pour la transformation du système de longitude et latitude vers UTM j'ai utilisé le package JScience. L'utilisation est assez simple car on y trouve les classes LatLong et UTM avec la méthode latLongToUtm() ce qui suffit pour les besoins de ce projet.

UTMMessage

A ce point, les données sont prêtes d'être envoyées à la classe qui fait appel à getData() (normalement la fenêtre principale) . UTMMessage est la classe qui encapsule les coordonnées UTM du dernier mouvement. Il est évident que la méthode ne retourne pas les variables primitives, mais une classe conçue pour ce type de résultat. Encore une fois, j'ai laissé l'espace pour le développement futur de l'application car on peut y placer d'autres informations utiles.

Références : Site de Apache Axis. http://ws.apache.org/axis/ ;
UNIVERSITE DE MARNE-LA-VALLEE. Présentation du projet Axis. http://www-igm.univ-mlv.fr/~dr/XPOSE2003/axis_seng ;

4.8 Aspect visuel de l'application

Selon le cahier des charges, le logiciel de suivi Dispatcher n'était pas censé avoir une représentation graphique à l'exception de l'affichage des instructions pour l'utilisateur. Comme il s'agit plutôt d'une librairie que d'une solution complète, et comme l'utilisateur final reçoit des messages sonores, mon client direct (HEVs) n'a pas eu besoin de visualisation.

Au cours d'une réunion avec Monsieur Jean-Luc Cochard, le client final, nous avons discute des fonctionnalités du futur logiciel. Monsieur Cochard m'a proposé d'essayer de visualiser le parcours de l'utilisateur puisqu'il serait plus facile pour les autres de comprendre l'application, mais aussi pour me faciliter le débogage du code. Cette proposition est correcte, car pendant le développement du logiciel je me suis rendu compte qu'il devient de plus en plus difficile de gérer tout sur la feuille, et qu'il y aurait plus de chances que j'aie des erreurs inaperçues par le fait qu'elles ne sont pas visibles sur l'écran. Etant donné que le travail sur le graphisme prendrait du temps, nous sommes arrivés à l'accord que, en cas de

manque du temps, j'effectuerai moins de tests que prévu au début du projet. Suite à cet accord, j'ai pu commencer à intégrer le graphisme dans mon application.

4.8.1 Le graphisme en Java

Les objets graphiques ont été créés et affichés avec Java 2D. Java 2D est une librairie graphique permettant de créer des figures géométriques en deux dimensions. Elle fait partie de la plateforme Java SE et, par conséquent, elle est fournie avec JDK. Les classes utilisées dans *Dispatcher* comme Line2D ou Point2D, représentent les formes géométriques et implémentent l'interface Shape.
Graphics2D est la classe de base pour tout ce qui est lié à l'affichage des formes 2D. On dessine les formes avec draw(), et on effectue les transformations avec des méthodes comme transform() et translate().

4.8.2 Implémentation de l'affichage

La classe MapDisplay est implémentée comme un composant graphique – une sorte d'écran qu'on peut « coller » aux autres composants. Elle hérite de JPanel et implémente l'interface Scrollable, ce qui permet de défiler l'image (scroll) si elle plus grande que le cadre du composant.

L'idée de base est d'afficher les composants statiques de la route comme les segments avec ses bornes et ses lignes de sortie, et de mettre à jour le chemin parcouru par l'utilisateur après chaque mouvement. L'affichage est effectué dans paintComponent() - la méthode trouvée dans la plupart des composant visuels de Swing qui prend une instance de Graphics comme l'argument, et qui dessine les figures géométriques sur l'écran. Pour dessiner les formes désirées, il faut surcharger cette méthode dans le code du composant Swing, ce qui est la classe MapDisplay dans notre cas.

Les deux problèmes principaux de l'implémentation sont les proportions de l'itinéraire affiché, et le système de coordonnées graphique employé par Java 2D. Le problème des proportions vient du fait que les coordonnées UTM des composants de l'itinéraire ont des très grandes valeurs que l'on n'arrive pas afficher. Il faut les diminuer proportionnellement pour les adapter mieux aux dimensions d'un écran. Autrement dit, il faut changer l'échelle de l'itinéraire.
Pour résoudre ce problème, il faut définir la longueur maximale de l'itinéraire, donnée comme l'argument du constructeur, qui représente la taille maximale en pixels. Ensuite, il faut trouver le reste de la division (modulo) de chaque coordonnée par la longueur maximale (maxRouteLength), ce qui nous donne les valeurs que l'on peut afficher sur l'écran.

Le deuxième problème est que Java 2D emploie le système de coordonnées standard utilisé dans les ordinateurs où l'axe y a le sens inverse par rapport à la représentation mathématique habituelle, autrement dit y grandit « en bas ». La bonne chose est que Graphics2D, la classe de base pour tout ce qui est lié au

graphisme, fournit la possibilité de changer le point d'origine du système de coordonnées à l'aide des méthodes translate() et scale(). Concernant les données du dernier mouvement, les coordonnées de ce point sont modifiées dans FrMainWidnow pour qu'elles soient adaptées au nouveau système de coordonnées.

Pour la meilleure orientation, le plan de la région qui comprend la zone de l'itinéraire est affiché derrière les éléments Java 2D. Dans le cadre de mon projet, l'adaptation d'un plan existant à un itinéraire se fait de manière manuelle. Il faut que les coordonnées de l'itinéraire soient en accord avec les coordonnées du plan, ce qui exige un peu de bricolage de l'image du plan.

Zoom

Au moment de l'écriture de ce chapitre, cette fonctionnalité n'était pas totalement implémentée parce qu'elle n'est pas prioritaire.

Le principe de zoom est d'effectuer une transformation affine des objets graphiques en utilisant la méthode scale() de la classe Graphics2D. La méthode scale() prend comme les arguments l'échelle de l'objet à transformer. Pour agrandir ou réduire l'image la valeur de l'échelle est augmentée ou diminuée. Après le changement de l'échelle, il faut mettre à jour tout les dimensions en appelant updateSize() et createAndScale().
Le défaut toujours présent est qu'après la modification d'échelle l'itinéraire ne se trouve pas à la bonne position, ce qui rend le plan de la région inutile.

La class MapDisplay reste ouverte pour les améliorations. Ce que j'ai créé représente l'affichage de base, mais si l'on veut une application ayant un fort aspect graphique, il faut investir beaucoup de temps.

Références : Java 2D Home Page. http://java.sun.com/products/java-media/2D/ ;

4.8.3 Fenêtre principale

J'ai laissé la description de cet élément pour la partie finale consacrée au développement car tous les autres composants de l'application ont été expliquées jusqu'à maintenant. La fenêtre principale, implémentée dans la classe FrMainWindow, est le point d'entrée dans l'application, et sert de liaison entre l'algorithme de suivi, la connexion, et l'affichage. Elle correspond parfaitement au module principal dans l'organisation modulaire de l'application, vue dans le cahier des charges et dans le chapitre « Structure de l'application ». La fenêtre principale initie la connexion avec Novasys, envoie les coordonnées vers Computation pour l'analyse, et, finalement, elle affiche les messages pour l'utilisateur et fait appel à MapDisplay pour mettre à jour l'affichage graphique. Il est évident que la fenêtre principale tient un rôle central dans le logiciel.

Depuis cette fenêtre on peut également ouvrir ou créer un nouvel itinéraire (route). On y trouve le paramétrage de l'application et de l'itinéraire.

5 Synthèse

5.1 Avantages

Dispatcher offre les avantages suivants :

Possibilité d'utiliser la classe Obstacle pour d'autres rôles utiles.

Tout au long du développement du logiciel j'ai veillé à ce que le code soit réutilisable. Pour cette raison, le code est, dans la mesure du possible, structuré de manière uniforme. Ceci facilitera la compréhension du code à des futurs développeurs de l'application, et permettra une intégration plus aisée des composants.
Pensant au développement futur de logiciel, j'ai également créé une partie de la documentation sous le standard Javadoc. Celle-ci offre une description des éléments Java de l'application destinée spécifiquement aux développeurs.
L'application permet de configurer la plupart de ses éléments. La quasi-totalité des expressions n'utilise pas des valeurs concrètes (« en dur »), mais des variables qui peuvent être modifiées et configurées dans un endroit dans le code, à l'aide des menus, ou dans les fichiers de configuration.

Du coté de travail sur le projet, je trouve que le fait de travailler seul sur un projet de longue durée apport une expérience qui me servira dans ma vie professionnelle.

5.2 Faiblesses de l'application

Le plus grand inconvénient vient de la technologie GPS. Comme on a vu dans les chapitres précédents, l'imprécision pose des problèmes pour le logiciel. Parfois les données envoyées peuvent être tellement incorrectes qu'elles empêchent totalement le suivi de l'utilisateur.

Un autre point faible est que l'application n'a pas été testée en profondeur parce qu'une grande partie de temps prévu pour les tests a été consacrée au développement de l'aspect graphique de l'application, ce qui a été convenu avec le client. Il est possible que, par la suite, il y ait des bogues qui n'ont pas encore été détectés.

5.3 Non-implémenté (développement futur)

L'application Dispatcher développée dans le cadre de ce travail de diplôme est un prototype. Certaines fonctionnalités devront soit être améliorées, soit être développées complètement afin d'arriver à un produit abouti. Les fonctionnalités qui n'ont pas été prises en compte pour ce prototype et qui devront être développées sont décrites ci-dessous :

- L'avertissement avant de la sortie des bornes du segment :
  A présent, l'utilisateur ne peut pas savoir qu'il se trouve devant les bornes. Dans chaque segment il faut créer une sorte de bornes internes. Une fois que l'utilisateur dépasse une de ces lignes, le logiciel l'averti qu'il est presque sorti de la zone du segment.

- Zoom :
  Fonctionnalité de zoom n'est pas encore implémentée, malgré le temps consacré. Pendant l'agrandissement, l'itinéraire ne se trouve pas au bon endroit par rapport à l'image de l'arrière-plan.

- L'énigme d'appels doubles :
  Pour certaines fonctionnalités, le *Action Listener* est appelé deux fois. Un exemple est zoom, où l'échelle incrémente ou décrémente deux fois si on appuie sur le bouton de zoom. Je n'ai toujours pas trouvé la solution.

- Téléchargement du plan de l'itinéraire :
  Au lieu d'adapter l'image du plan manuellement, il peut être possible de télécharger depuis Novasys la partie du plan correspondante aux coordonnées de l'itinéraire. Il s'agit alors d'automatiser le processus de l'intégration du plan.

- Développement d'un outil pour la création de l'itinéraire :
  La réalisation d'un itinéraire serait simplifiée si on pouvait créer un outil dédié à cette tâche. Ce logiciel serait adapté aux caractéristiques de *Dispatcher*, mais il garderait au même temps le format de fichiers Shapefile à cause de la compatibilité avec SIG.

- Améliorations de l'algorithme de suivi :
  Une meilleure reconnaissance des mouvements de l'humain. Reconnaissance des fausses données GPS plus avancée.

- Itinéraires inhabituels :
  Il faut prendre en compte les cas exceptionnels potentiels où la façon habituel de guider n'est pas valable.

6 Conclusion

Ce travail de diplôme permet de démontrer la faisabilité d'un dispositif destiné aux personnes malvoyantes, qui soit basé sur la technologie GPS et Java. Le prototype créé possède des fonctionnalités de base qui permettent à l'utilisateur de trouver son chemin en dépassant les obstacles éventuels grâce aux indications du logiciel et d'un opérateur externe.

Au niveau de la gestion du projet, j'ai pu mettre en pratique les connaissances acquises tout au long de mes études à l'HEVs. La taille du projet a nécessité une planification des étapes et un bon découpage des tâches à réaliser.
Ce travail m'a permis de me confronter à une situation professionnelle réelle qui implique notamment de gérer la relation avec un client et ses exigences.

Ce projet a requis des connaissances au niveau du graphisme 2D Java, de l'utilisation des services web ainsi que du calcul et de l'exploitation de données spatiales. Il m'a également permis de connaître en profondeur le concept de Système d'information géographique (SIG) et les notions de cartographie et de géographie, qui lui sont liées.

Nous avons vu que la technologie GPS, qui reste encore imprécise, limite les performances du dispositif, et ceci malgré les alternatives que j'ai mises en place pour rendre les coordonnées plus précises. Cet élément est à prendre en compte lors des développements futurs de l'application.

Dans cette optique, le logiciel créé dans le cadre de ce travail de diplôme offre, grâce à son code et à sa documentation standard, les conditions nécessaires pour le développement futur de l'application.

7 Références

« ESRI Shapefile Technical Description » :
http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf

Don BARTLETT. *A Practical Guide to GPS – UTM.* *http://www.dbartlett.com* ;

Dr. Sergei ANDRONIKOV. *Introduction to GIS lecture notes.*
http://geog.gmu.edu/PEOPLE/fall2004/Sergei/GEOG311.htm ;

Java 2D Home Page. http://java.sun.com/products/java-media/2D/ ;

JOVANOVIC A. Slobodan. Savremeni francusko-srpski rečnik;

JScience. http://jscience.org ;

JShape. http://www.jshape.com ;

LIPKOVSKI Aleksandar. Linearna algebra i analitička geometrija.

MapInfo Professional. http://www.mapinfo.com ;

Online French Dictionary. http://www.wordreference.com ;

OziExplorer. http://www.oziexplorer.com ;

Site de Apache Axis. http://ws.apache.org/axis/ ;

Site de GeoTools. http://geotools.codehaus.org ;

Spécifications OpenGIS : http://www.opengeospatial.org

Sun Mycrosystems. Java 2D Graphics Tutorial :
http://java.sun.com/docs/books/tutorial/2d/index.html;

UNIVERSITE DE MARNE-LA-VALLEE, présentation du projet Axis. http://www-igm.univ-mlv.fr/~dr/XPOSE2003/axis_seng;

Université de Québec. *Projections, systèmes de coordonnées.*
http://www.unites.uqam.ca/dgeo/geo7511-2001/htm/section6.htm ;

Vivid Solutions. Java Topology Suite.
http://www.vividsolutions.com/jts/jtshome.htm ;

Wikipedia : http://fr.wikipedia.org ;

8 Annexes

8.1 Planification

8.1.1 Planification du travail

ID		Task Name	Duration	Start	Finish	Predecesso
1		**Project GPS Guidance**	**64 days**	**Mon 25.09.06**	**Mon 08.01.07**	
2		**Initialisation**	**1 day**	**Mon 25.09.06**	**Mon 25.09.06**	
3		Séance de démarage	1 day	Mon 25.09.06	Mon 25.09.06	
4	✓	**Cadrage**	**11 days**	**Tue 26.09.06**	**Thu 26.10.06**	**2**
5	✓	Etude du SIG	4 days	Tue 26.09.06	Fri 29.09.06	
6	✓	Recherche sur Java et SIG	2 days	Wed 27.09.06	Thu 28.09.06	
7	✓	Diagrammes de flux de données	1 day	Thu 28.09.06	Thu 28.09.06	
8	✓	Etude de l'appareil Mambo	1 day	Fri 29.09.06	Fri 29.09.06	
9	✓	Etude des logiciels SIG	4 days	Mon 02.10.06	Thu 05.10.06	
10	✓	Tests de graphisme Java2D pour le futur	4 days	Tue 03.10.06	Fri 06.10.06	
11	✓	Etablissement du cahier des charges	2 days	Wed 25.10.06	Thu 26.10.06	
12						
13	✓	**Design**	**11 days**	**Wed 25.10.06**	**Tue 07.11.06**	**11SS**
14	✓	Structuration de l'application	1 day	Wed 25.10.06	Wed 25.10.06	
15	✓	Etude des systèmes des coordonnées	3 days	Wed 25.10.06	Fri 27.10.06	
16						
17	✓	**Module de connexion**	**2 days**	**Fri 27.10.06**	**Mon 30.10.06**	
18	✓	Etude de fonctionnement du système Novasys	2 days	Fri 27.10.06	Mon 30.10.06	
19	✓	Etude de transformation de données lat/long -> UTM	1 day	Mon 30.10.06	Mon 30.10.06	
20	✓	Design de module de connection	1 day	Mon 30.10.06	Mon 30.10.06	
21	✓	**Module de calcul**	**7 days**	**Tue 31.10.06**	**Tue 07.11.06**	
22	✓	Modèle mathématique	5 days	Tue 31.10.06	Sat 04.11.06	
23	✓	Algorithme de suivi	5 days	Thu 02.11.06	Tue 07.11.06	22SS+2 da
24	✓	Diagramme de classes	2 days	Sat 04.11.06	Sun 05.11.06	
25	✓	**Module d'affichage**	**1 day**	**Wed 25.10.06**	**Wed 25.10.06**	
26	✓	Design de module d'affichage	1 day	Wed 25.10.06	Wed 25.10.06	
27	✓	**Module principal**	**1 day**	**Wed 25.10.06**	**Wed 25.10.06**	
28	✓	Design de module principal	1 day	Wed 25.10.06	Wed 25.10.06	
29						
30						
31		**Construction**	**43 days**	**Fri 10.11.06**	**Mon 08.01.07**	
32	✓	**Module de calcul**	**13 days**	**Fri 10.11.06**	**Tue 28.11.06**	**21**
33	✓	Développement d'algorithme	12 days	Fri 10.11.06	Mon 27.11.06	
34	✓	Saisie des données	10 days	Mon 13.11.06	Tue 28.11.06	33SS+3 da
35	✓	Tests	10 days	Mon 13.11.06	Tue 28.11.06	33SS+3 da
36		**Module de connection**	**8 days**	**Mon 04.12.06**	**Wed 13.12.06**	**32**
37	✓	Construction de connection	7 days	Mon 04.12.06	Tue 12.12.06	
38	✓	Transformation lat/long -> UTM avec GeoTools	3 days	Tue 05.12.06	Thu 07.12.06	37SS+1 day
39		Tests	7 days	Tue 05.12.06	Wed 13.12.06	37SS+1 da
40		**Module d'affichage**	**6 days**	**Mon 13.11.06**	**Mon 20.11.06**	**32SS**
41	✓	Développement	4 days	Mon 13.11.06	Thu 16.11.06	
42	✓	Tests	4 days	Mon 13.11.06	Thu 16.11.06	41SS
43		Amélioration de l'apparence	2 days	Fri 17.11.06	Mon 20.11.06	41
44	✓	**Module principal**	**6 days**	**Thu 07.12.06**	**Thu 14.12.06**	
45	✓	Développement	6 days	Thu 07.12.06	Thu 14.12.06	
46	✓	Tests	5 days	Fri 08.12.06	Thu 14.12.06	45SS+1 da
47						
48	✓	Saisie de données réelles avec ArcView et Mambo	3 days	Fri 15.12.06	Tue 19.12.06	32;36;44
49	✓	Tests de l'application en entier	7 days	Tue 19.12.06	Wed 27.12.06	48SS+2 days
50	✓	Documentation (tout au long de travail)	7 days	Thu 28.12.06	Mon 08.01.07	

Project: PlanningV10
Date: Tue 09.01.07

Task
Split
Progress
Milestone
Summary
Project Summary
External Tasks
External Milestone
Deadline

8.1.2 Calendrier de travail

Semaine 1
25 – 29 septembre 2006.
- Recherche sur SIG.
- **Rendez vous** avec David. Discussion général et sur SIG.
- Etude de SIG et lecture des documents concernant SIG.
- Création d'un planning général.

Semaine 2
2 – 6 octobre 2006.
- Etude de SIG.
- Recherche et tests des logiciels et des outils SIG (JShape, GeoTools, ArcView, FGis…).
- Conception de l'interface de l'application.
- Recherche concernant les possibilités de la représentation graphique (mappe, itinéraire et obstacles).
- Etude et tests de Java 2D.
- Développement du module graphique avec la possibilité de « zooming ».
- **Rendez-vous** avec David. Mauvaise compréhension : en fait, il ne faut pas créer le module graphique représentant l'utilisateur sur son itinéraire car l'opérateur n'en a pas besoin. Le travail de 4 jours à 100% est annulé.

Semaine 3
9 – 13 octobre 2006.
Absence pour cause de maladie.

Semaine 4
16 – 20 octobre 2006.
Absence pour cause de maladie.

Semaine 5

23 - 24octobre 2006.
Absence à cause de maladie.
25 octobre 2006.
Cahier des charges écrit et envoyé à David.
Etude des différents systèmes de coordonnées et des projections (UTM et longitude/latitude).
26 octobre 2006.
Rendez-vous avec David.
Lecture des tutoriels et utilisation du logiciel ArcView pour créer des fichiers Shapefile.
Lecture de ces fichiers à partir de java avec GeoTools.
Préparation de rendez-vous avec Me Cochard.

27 octobre 2006.
Rendez-vous avec Me Cochard à Bern. Discussion autour du projet et la représentation d'un itinéraire dans l'application. On s'est mis d'accord que

l'application va comprendre une représentation graphique au prix d'effectuer moins de tests.
Rencontre avec l'ingénieur de Novasys. Explication des détails concernant la connexion avec le serveur de Novasys, le format et le contenu des messages renvoyés par Novasys, la configuration de Mambo…On a découvert que Mambo ne marche plus. L'envoie de nouvel appareil va durer au moins une semaine.

Semaine 6

30 octobre 2006

Rendez-vous avec David. On a échangé des idées concernant le suivi de personne. Certaines idées initiales sont modifiées.
Tests dans ArcView de nouveau concept d'itinéraire.
Début de travail sur l'algorithme.

31 octobre 2006

Toute la journée de travail sur l'algorithme de suivi et sur la forme géométrique de l'itinéraire. Beaucoup des choses modifiées au fur et à mesure.
J'essaie de trouver la solution la plus optimale et la plus agréable pour l'utilisateur final.

1 novembre 2006

Comme le jour précédent, j'ai travaillé la plupart de temps sur l'algorithme.
Travail sur la documentation (planning, rapport).
Normalement, c'est le jour de férié (fête de Toussaint). Comme j'ai travaillé à 100%, je serai absent un autre jour pour mes obligations privées.

2 novembre 2006

Rendez-vous avec David. On a discuté de l'algorithme. Quelque modifications apportées.
Travail sur l'algorithme.

3 novembre 2006

Travail sur l'algorithme.

4 novembre 2006 (samedi)

Travail à 50%.
Toujours le travail sur le design de l'algorithme.

5 novembre 2006 (dimanche)

Travail à 50%.
Algorithme.
Planification de travail en MS Project.
Commencé diagramme de classes de l'application.

Semaine 7

6 novembre 2006

Diagramme de classes. Structuration des classes – variables et méthodes.
Documentation des méthodes et des classes.

7 novembre 2006

Début de développement du module de calcul. J'utilise Rational Rose pour la création des classes, j'y génère du code et je continue développer avec JBuilder.
Je documente les classes au même temps.

8 et 9 novembre 2006

Absence comme annonce dans l'e-mail.

10 novembre 2006

Travail avec Rational Rose et JBuilder – structuration du code, création des classes. Documentation.

11 et 12 novembre 2006 (weekend)

Travail à 50%.
Travail sur exitLineReached() et obstacleBoundsReached() - les méthodes de la classe Computation.
Diagramme de classes.

Semaine 8

13 novembre 2006

J'ai découvert que les méthodes comme celle pour chercher l'intersection de deux lignes ou la distance entre une ligne et un point, qu'elles existent déjà dans JDK - dans la classe Line2D. Comme cela, 3-4 jours de travail sont perdus (analyse, développement).
J'ai modifié les classes et maintenant j'utilise les méthodes fournies par JDK quand c'est possible.
Travail sur les méthodes de Computation.

14 novembre 2006

Développement des méthodes de Computation.
Rendez-vous avec David. On a parlé des questions cruciales comme les obstacles possibles sur l'itinéraire, création des bornes des segments. On a également modifié certaines méthodes de calcul. Les questions mentionnées doivent être posées à Me Cochard par e-mail.

15 novembre 2006

J'ai envoyé les questions à Me Cochard et j'ai reçu la réponse positive, donc les hypothèses posées hier étaient correctes.
Continué de travailler sur la classe Computation.

16 novembre 2006

La méthode de correction de la direction est finie. J'ai également travaillé sur la partie graphique (représentation des différentes lignes) afin de voir mieux le comportement pendant le debugging.

17 novembre 2006

Travail sur les méthodes de Computation.

Semaine 9

20 novembre 2006

Tests et débogage des méthodes de Computation. Tests sur la représentation complète d'un segment.
Amélioration de MapDisplay.
Documentation Javadoc des classes créées.

21 novembre 2006

Recherche sur la transformation de coordonnées. Etude de format UTM. Trouvé la bibliothèque Java utile pour la transformation Lat/Long -> UTM.
L'échelle de l'affichage (MapDisplay) doit être modifiée.

22 novembre 2006

Travail…

23-24 novembre 2006

Absence à cause de voyage au Monténégro. Je vais travailler pendant les weekends afin de suppléer ce temps.

Semaine 10

27 novembre 2006

Déboguage et transformation.

28 novembre 2006

Déboguage et transformation.

29 novembre 2006

Début de Connection.

30 novembre 2006

Débogage de l'algorithme de suivi (classe Computation). Ajout des fonctionnalités comme les obstacles.
Documentation de nouvelles fonctionnalités et la correction de l'ancienne documentation.

1 décembre 2006

Travail…

2 et 3 décembre 2006 (weekend)

Transformation de coordonnées de pixels (utilisés jusqu'à maintenant) vers UTM des environnes de l'HEVs (l'endroit où je teste). L'affichage devrait également être modifié.
Tests et débogage avec les nouvelles coordonnées.
Documentation Javadoc.

Semaine 11

4 décembre 2006

Début du travail sur la connexion avec Novasys. Etude de consommation de web service depuis Java.
Problèmes rencontrés pendant la création du client. Recherche sur Internet.

5 décembre 2006

Trouvé qu'il faut utiliser Apache Axis pour générer les fichier Java. Installation et utilisation de Axis.
Rendez-vous avec David. Il m'a aidé de créer les fichiers client pour un web service quelconque.
Toujours je n'arrive pas à appeler le service de Novasys.

6 décembre 2006

Trouvé un problème lié au fichier wsdl créé par Novasys. J'ai passe la plupart de temps à échanger des emails avec monsieur Heydebrand de Novasys en essayant de corriger le problème. Après avoir parlé avec lui, on a conclu qu'il faut que je vienne à Berne avec Mambo pour que il puisse le configurer.

7 décembre 2006

Toute la journée j'ai passé à Berne. Il y avait des problèmes avec la configuration de Mambo, mais aussi de coté serveur et web service MessageService. Monsieur Heydebrand a reconfiguré Mambo et corrigé les problèmes de serveur.

8 décembre 2006

Maintenant j'ai des problèmes de caractère différent – je n'arrive pas récupérer les fichiers XML contenant les coordonnées, même s'il sont placés sur le serveur.
Comme monsieur Heydebrand n'est pas là aujourd'hui, j'ai passé le reste du temps pour travailler sur le concept des fichiers shapefile décrivant l'itinéraire.

9 et 10 décembre 2006 (weekend)

Après la recherche et le travail de toute la journée, j'ai trouvé pourquoi je ne peux pas récupérer les fichiers XML. En analysant les paquets TCP, j'ai vu que les données XML ont été envoyées, mais elles contenaient des caractères non acceptés par XML, ce qui a provoqué le refus de données de coté parseur. Je ne peux pas continuer de travailler sur cette partie avant que Novasys ne corrige cette erreur. J'ai écrit un email à Monsieur Heydebrand expliquant le problème.
Concernant l'itinéraire, j'ai utilisé Mambo pour la prise des coordonnées de chemin devant l'école et je les ai vérifiées avec Google maps. J'ai trouvé que les coordonnées envoyées par l'appareil ne sont pas toujours fiables – il y a souvent de décalage de quelques mètres, mais également, plus rarement, de plusieurs dizaines de mètres !

Semaine 12

11 décembre 2006

J'ai laissé à coté la connexion avec Novasys et j'ai commencé travailler sur la partie chargement des donnes à partir des fichiers Shapefile.
Cette tâche se montre beaucoup plus complexe que j'ai prévu dans mon planning. Outre de rajout des nouvelles classes, j'ai du modifier certaines classes existantes. J'ai également découvert certains problèmes avec les classes existantes que j'ai corrigés.

12 décembre 2006

Toujours le travail sur le chargement des données. J'ai ajouté les fonctionnalités de création d'une nouvelle route à partir des fichiers Shapefile et des données saisies, et de chargement d'une route existante.
Il me reste toujours du travail pour cette partie.

13 décembre 2006

Fin du travail sur le chargement de données. Débogage et tests de chargement.
Correction des problèmes connus liés à l'affichage et zooming.
Les nouveaux problèmes sont apparus. J'ai changé l'origine du système de coordonnées et maintenant j'ai les problèmes avec l'affichage du point cliqué avec la souris.

14 décembre 2006

Finalement le problème avec la souris est résolu. Je n'ai toujours pas réussi à résoudre le problème avec le zoom.
J'ai changé la fenêtre principale et j'ai amélioré l'interface.

15 décembre 2006

J'ai repris le travail sur la connexion avec Novasys. Cette fois elle marche, et j'ai réussi à lire des coordonnées.
Maintenant le problème est de faire du « parsing » du fichier XML reçu. J'ai fait des recherches sur les parseurs DOM e SAX et j'ai fait des tests avec les exemples que j'ai trouvés sur l'Internet.

17 décembre 2006 (dimache)

La connexion et la lecture des coordonnées depuis le fichier XML sont finies. Comme le fichier est assez petit, au lieu d'utiliser un parseur, je l'ai transformé en string et j'ai cherche les coordonnées avec les méthodes de la classe String.

Semaine 12

18 décembre 2006

Comme il n'y avait pas beaucoup de temps pour effectuer les tests avant, quand j'ai commencer les test aujourd'hui, j'ai trouve quelque bogues graves ! J'ai passé toute la journée en essayant de les corriger.

19 décembre 2006

Toujours la correction des bogues. Ceux que j'ai trouvés j'ai corrigé, mais je ne peux plus tester et corriger car je dois absolument commencer travailler sur la documentation !

20 décembre 2006

Début du travail sur le rapport. Le travail va durer jusqu'à la fin de période de temps prévue pour le TD. S'il reste du temps, je vais améliorer est déboguer le code.

21 décembre 2006 – 9 janvier 2007

Travail sur le rapport. Déboguage.

8.2 Cahier des charges

Travail de diplôme de l'année scolaire 2006/2007

Cahier des charges

Description

Il s'agit d'une application qui servira à aider aux personnes handicapées, malvoyantes à la première place, de traverser un itinéraire critique se trouvant sur leur chemin.
L'utilisateur sera équipé d'un téléphone portable spécial comprenant un récepteur GPS – l'appareil MAMBO produit par Falcom. L'itinéraire que l'utilisateur va prendre, ainsi que les obstacles seront saisis « en dur » dans un fichier du format Shapefile en utilisant un système des coordonnées standard et sous la forme vectorielle.
Quand l'utilisateur commence marcher du point de départ vers son point d'arrivée, son portable envoie en permanence sa position vers l'application qui sera capable de suivre son avancement. S'il s'approche à un obstacle, le logiciel va afficher un avertissement et une instruction pour dépasser l'obstacle. L'opérateur qui se trouve devant l'ordinateur doit communiquer le message à l'utilisateur. Le logiciel va également contrôler si l'utilisateur se trouve toujours sur le bon chemin, et le message approprié sera affiché s'il s'éloigne de l'itinéraire.

Composants de l'application

L'application se composera de plusieurs modules qui interagissent entre eux.

Module principal

Le module principal comprend l'interface graphique de l'application, fait appel aux différents modules et orchestre leurs entrées et sorties afin d'accomplir la tâche de suivi de l'utilisateur.
D'abord, les données de la position courante d'utilisateur sont fournies par le module de récupération des données. La position courante et la position précédente sont envoyées vers le module de calcul afin de contrôler si l'utilisateur se trouve sur le bon chemin. La réponse du module de calcul est envoyée au module d'affichage, et l'opérateur va pouvoir donner la nouvelle instruction à l'utilisateur (arrêter, tourner…).

Module de récupération des données

La position courante de l'utilisateur sera envoyée par SMS vers le serveur intermédiaire appartenant à l'entreprise Novasys. Les données seront transformées est renvoyées par TCP/IP vers l'application.
Le format des données et les paramètres de connexion seront décrits par Novasys prochainement.

Module de calcul

Ce composant va contrôler si l'utilisateur se trouve sur l'itinéraire, s'il se trouve devant un obstacle ou s'il a atteint le point d'arrivée. Le principe est de comparer la position précédente avec la position courante. Comme le résultat du calcul, le message approprié sera envoyé vers le module principal. Le message contient une des instructions : stop, tourner à gauche ou à droite à un certain angle, faire certaines actions devant l'obstacle (dépend du type d'obstacle), avancer (ou pas de message).
Voici les cas possibles :

1. L'utilisateur s'éloigne trop de la ligne décrivant l'itinéraire, la limite étant définie auparavant.
   Réponse : stop, tourner d'un nombre de degrés (90°, 180°...), continuer (vers le chemin).
2. La personne se trouve devant un obstacle. La réponse dépendra du type d'obstacle. Par exemple, devant le feu rouge il faut attendre.
   Réponse : stop, l'instruction spécifique.
3. Arrivée à la fin de l'itinéraire.
   Réponse : fin de suivi.
4. La personne se trouve sur l'itinéraire, il n'y a pas d'obstacles.
   Réponse : continuer (ou pas de message).

Module d'affichage

L'affichage des instructions. Le texte du message envoyé par le module de calcul sera affiche, ainsi qu'un symbole graphique comme la flèche ou le signe de stop.

Professeur responsable	Etudiant
David Russo	**Vladimir Drakic**
_____	_____

8.3 Manuel d'utilisation

Manuel d’utilisation de Dispatcher

Drakic Vladimir

HEVs, décembre 2006

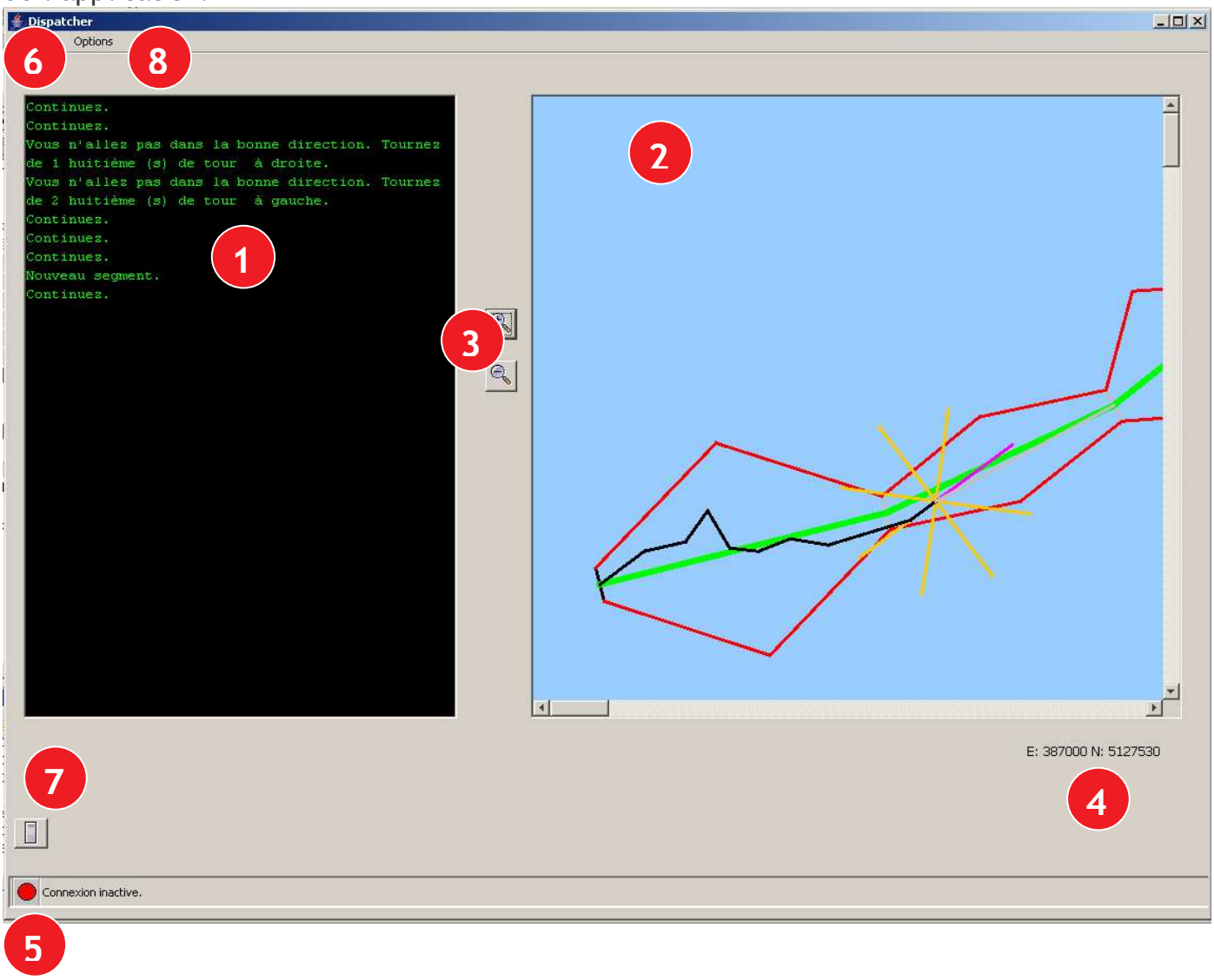
Ce manuel est composé de deux parties : Guide d'utilisateur, et Guide de développeur. La première partie expose l'utilisation et la configuration de l'application du point de vue d'un utilisateur, où tout ce fait à l'aide de l'interface du logiciel. Donc il n'est pas nécessaire de plonger dans le code pour exploiter le logiciel. Le guide de développeur explique des paramétrages avancés et l'utilisation des outils externes comme ArcGIS.

Il est fortement conseillé de lire la documentation de l'application avant d'utiliser ce manuel. Les termes et les concepts de l'application sont considérés connus par le lecteur, et ils ne sont pas expliqués dans le manuel.

Guide de l'utilisateur

Dans cette version de l'application j'ai laissé la possibilité de déplacer l'utilisateur en cliquant sur la représentation graphique de l'itinéraire. J'ai utilisé cette option pendant le débogage et je trouve qu'elle peut faciliter les tests du logiciel.
Je n'ai pas implémenté la version exécutable de l'application à cause de certains bogues connus. Pour le moment, il faut lancer l'application *JBuilder*.

Les numéros sur l'image suivante expliquent en détail chaque élément de l'interface de l'application.



1 – Console d'affichage des messages

Dans cet endroit les instructions et les messages pour l'utilisateur – randonneur sont affichés. Les messages sortent dans la console de façon continue. Si le nombre de messages dépasse l'hauteur de la console, la barre qui permet le défilement (*scrolling*) va apparaître.

2 – Panneau de l'affichage du graphisme

L'aspect graphique de Dispatcher est réalisé dans ce panneau. Nous pouvons voir les segments de l'itinéraire affichés L'image de l'itinéraire et des éléments qui le forment sont affichés de manière qu'ils restent fidèles aux proportions de « l'original ». Pour satisfaire cette contrainte, le changement de l'échelle est appliqué aux coordonnées de l'itinéraire.

Chaque segment est entouré par ses bornes de couleur rouges. La ligne de segment est verte, tandis que les lignes de sortie sont jaunes.
Le suivi du déplacement de l'utilisateur est réalisé en affichant un trait noir pour chaque mouvement.
Les lignes oranges/jaunes démontrent toutes les directions possibles pouvant être proposées à l'utilisateur. Celle de couleur pourpre représente la ligne la plus proche au point d'arrivée et sa direction sera choisie comme la meilleure proposition pour l'utilisateur.

Utilisation

Normalement, après l'établissement de la connexion, vous allez voir le déplacement de l'utilisateur en temps réel, sans influencer l'affichage. Parce que ici il s'agit d'un prototype de l'application demandant encore de tests, j'ai laissé la possibilité de simuler le mouvement de l'utilisateur en cliquant avec la souris sur la région d'affichage.

3 – *Zoom in/out*

Les boutons utilisés pour agrandir ou diminuer l'affichage de l'itinéraire.

Le **défaut** de cette fonctionnalité est que *zoom in* ne positionne pas l'itinéraire au bon endroit par rapport à l'image de derrière plan.

4 – Coordonnées

Les coordonnées qui se trouvent au-dessous de panneau graphique, ont la forme du système UTM représentée avec *northing* et *easting*. Les valeurs affichées montrent la position sur la carte du pointeur de la souris.

5 – Indicateur de la connexion

Cette icône montre l'état de la connexion qui peut être active ou inactive.

6 – Chargement d'un itinéraire

La procédure de chargement d'une route est très simple : il faut choisir l'option *Ouvrir* dans le menu *Itinéraire* et ensuite choisir le fichier correspondant. Les données de l'itinéraire se trouvent dans le type de fichier dont l'extension porte le nom `.rte` (route).
Important : Pour que les objets soient affichés, il faut appuyer une fois sur le bouton de *Zoom in*. Après le chargement de l'image et avoir appuyé sur zoom, utilisez les barres de navigation latérales (avec une direction vers le bas et un peu à droite) pour trouver l'image zoomée.

7 – Bouton de connexion

Le suivi de un utilisateur commence dès le moment où il se trouve dans le périmètre de l'itinéraire et la connexion avec serveur est établie. Pendant le déplacement, l'appareil GPS de l'utilisateur envoi ses coordonnées vers le serveur en permanence. Pour effectuer un suivi en temps réel, Dispatcher doit demander les nouvelles données du serveur aux intervalles de temps courts. En appuyant sur ce bouton, nous démarrons la communication avec le serveur de Novasys.

Important : cette fonctionnalité ne marche pas correctement dans cette version à cause de problème de double appel (mentionné dans le rapport). Il est possible de commencer la connexion seulement une fois. Si vous l'arrêtez et démarrez encore une fois, rien ne va se passer. Pour rétablir la connexion il faut relancer l'application.

8 – Menu Options

Dans ce menu on trouve le paramétrage basique de l'application.
Les paramètres de la connexion configurables sont le nom de l'utilisateur et le mot de passe utilisés pour l'identification sur le serveur. Ces données sont correctes et il ne faut pas les modifier.
L'autre option utile est l'intervalle de temps dans lequel Dispatcher demande les coordonnées de serveur. Il exploite chaque fois le service web MessageService qui est dédié à cette tâche.

Guide du développeur

Dans ce guide j'explique le paramétrage avancé et la création des nouveaux itinéraires. Ces opérations sont désignées pour les développeurs car elles demandent la compréhension et la modification du code.

Logiciels externes

Afin de pouvoir développer et modifier le Dispatcher, il faut installer quelques applications et plusieurs librairies. Les librairies nécessaires viennent avec Dispatcher dans le répertoire *lib*.

Les applications nécessaires sont :

- JBuilder 2006 (avec jdk1.5.0)
  http://www.borland.com
- ESRI ArcView 9.1
  http://www.esri.com
- Apache Axis 1.4
  http://ws.apache.org/axis/

Les autres librairies et logiciels qui vient avec Dispatcher :

- GeoTools 2.2
  http://geotools.codehaus.org
  GeoTools nécessite :
  - Java 3D
    http://java.sun.com/products/java-media/3D/
  - Java Advanced Imaging
    http://java.sun.com/javase/technologies/desktop/media/jai/
  - JTS Topology Suite
    http://www.vividsolutions.com/JTS/JTSHome.htm
- JScience 3.2
  http://jscience.org/

Création d'un nouvel itinéraire

Cette fonctionnalité nécessite un suivi strict des instructions exposées ici parce que certaines erreurs, qui paraissent négligeables, peuvent provoquer le comportement imprévisible de l'application.
La partie de la création concrète des segments s'effectue avec *ArcGIS*. Quand même, il faut configurer le logiciel de façon que le suivi soit précis.

Récupération des coordonnées de l'itinéraire

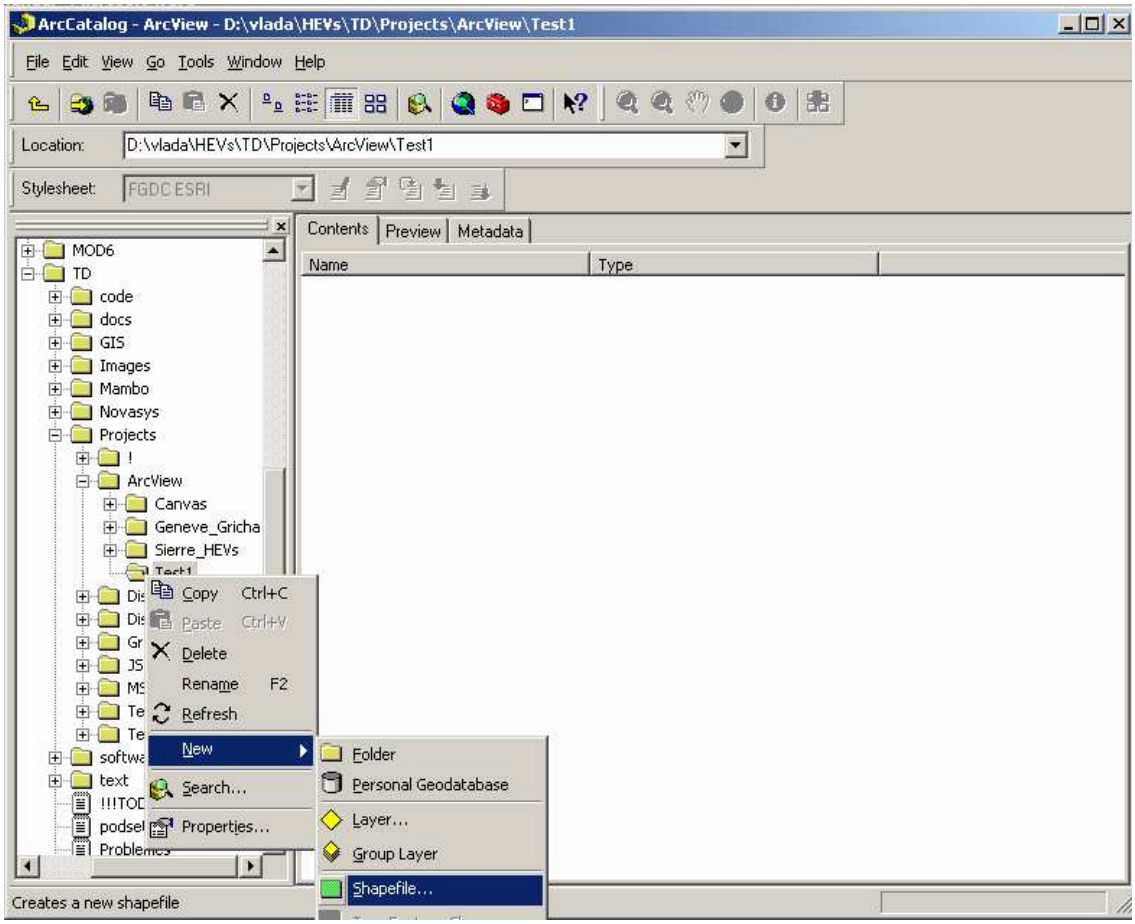
La première phase du travail comprend la collection des coordonnées de l'itinéraire. La façon la plus simple est de se promener sur le parcours avec *Mambo* allumé, ce qui va créer les fichiers des coordonnées sur le serveur. Pour récupérer les donnés l'application s'utilise de manière habituelle. On se connecte à *Novasys* et *Dispatcher* va chercher et afficher les coordonnées trouvées.
L'autre solution est d'utiliser un système comme *Google Earth* et d'y trouver la région de l'itinéraire. J'ai utilisé le site http://mapper.acme.com à ce but où on peut choisir entre différents systèmes de coordonnées et sauvegarder les informations facilement en générant l'*URL* correspondant.

Création d'un itinéraire dans ArcGIS

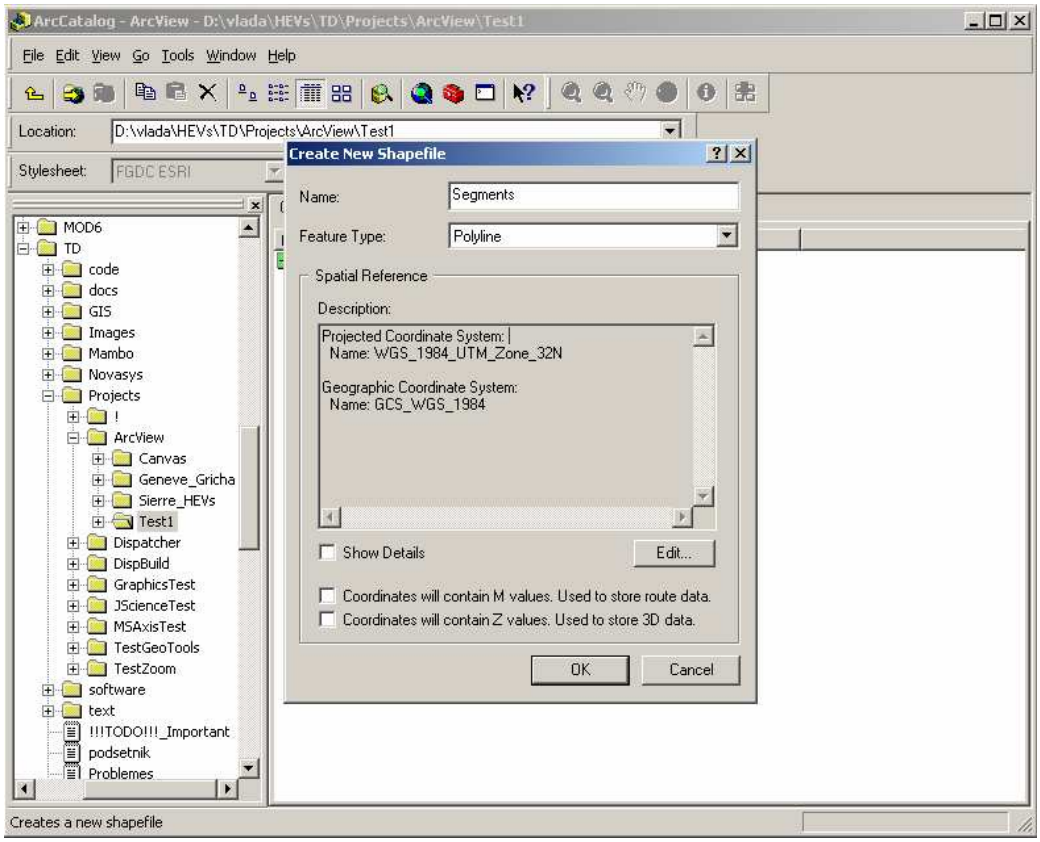
D'abord, les composants d'un itinéraire doivent être créés et catalogues à l'aide de *ArcCatalog*. Avec cette application, appartenant à la suite *ArcGIS*, nous allons définir la structure de chaque composant. Les nouveaux éléments auront le format *Shapefile*, expliqué dans le rapport.

ArcCatalog

Démarrez l'application *ArcCatalog*. Dans le panneau gauche il faut choisir le répertoire pour les fichiers de l'itinéraire. Placez le pointeur sur ce dossier et cliquez sur le bouton droit de la souris. Sélectionnez **New / Shapefile** et donner le nom au *Shapefile*. Le fichier des segments sera créé en premier. Il est recommandé de nommer le fichier *segments* afin de le mémoriser plus facilement.

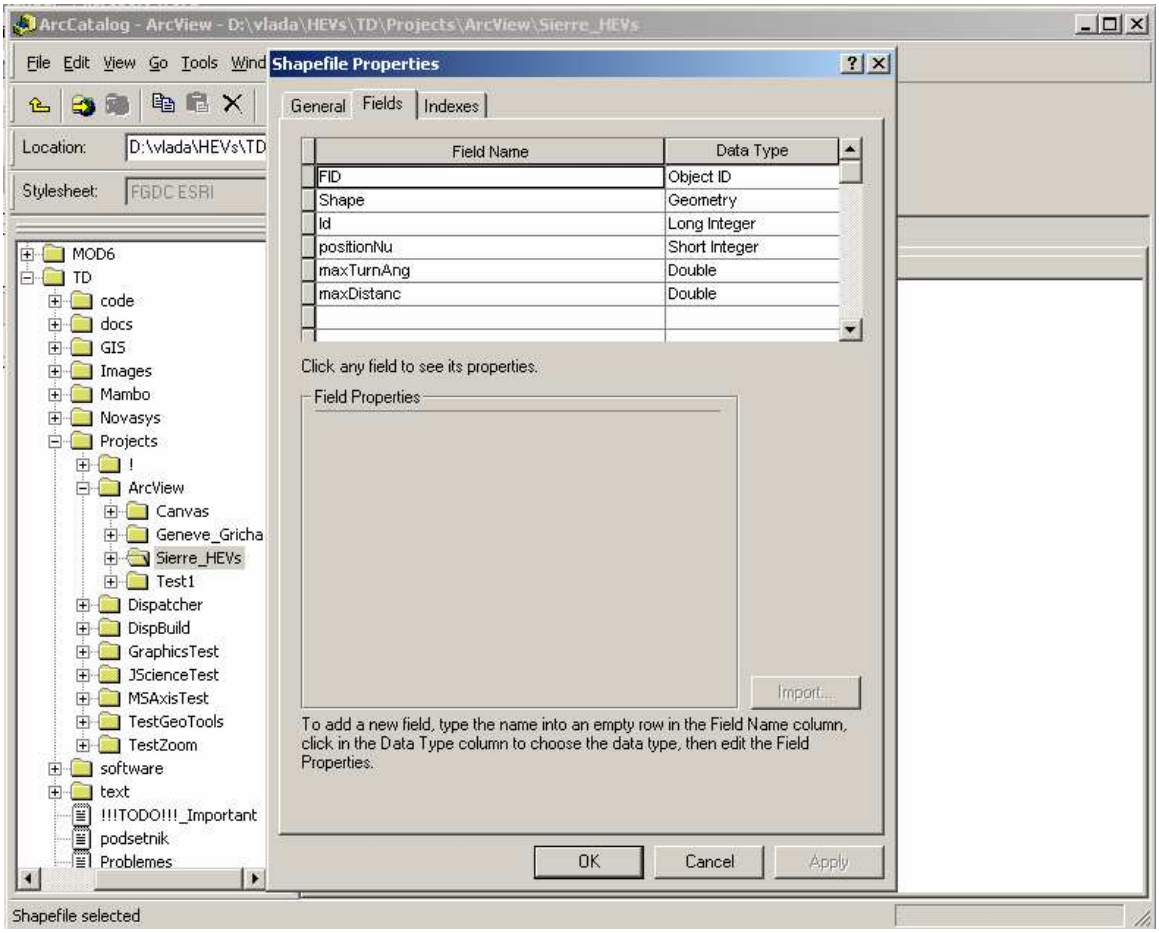


L'itinéraire est stocké avec *ArcGIS* comme une ligne segmentée, donc on va choisir choisir *Polyline* comme la valeur de *Feature Type*.
Dernier options dans cette fenêtre et le système de coordonnées. Appuyez sur **Edit**, e et ensuite sur **Select.** Dans notre cas il faut choisir : **Projected coordinate system / UTM / WGS 1984 / WGS_1984_UTM_Zone_32N.prj**
Avant de confirmer **OK** final, il faut être sur que tout est bien configuré parce que après on ne peut pas changer le nom et le type de *Shapefile*.
L'image montre la configuration correcte :



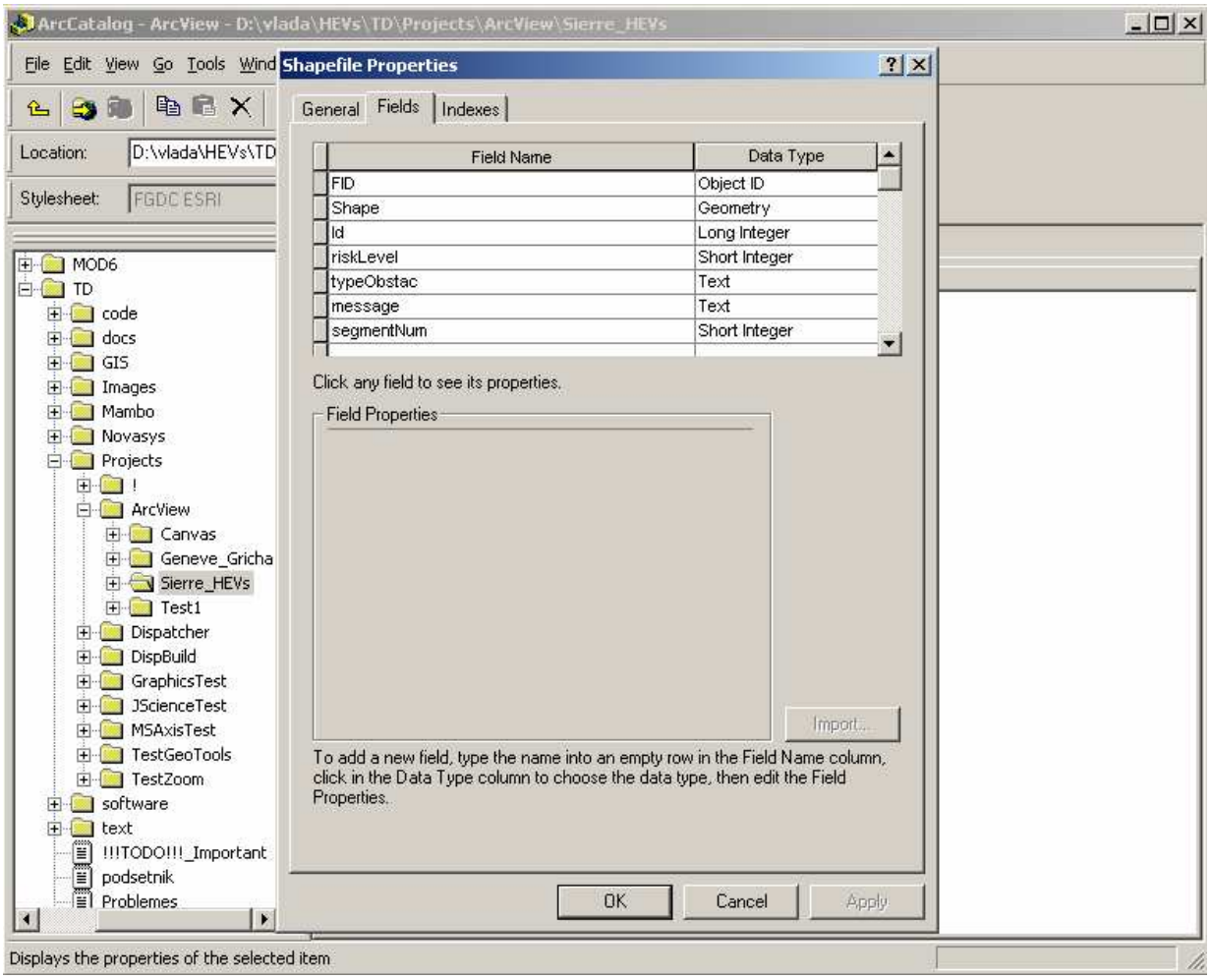
Maintenant il faut créer les attributs de *Shapefile* des segments qui correspondent aux attributs dans la classe `Segment`. On crée d'abord le nom de l'attribut qui peut avoir dix caractères au maximum, et ensuite on définie son type. Encore une fois il faut se rassurer que les données soient correctes car on ne peut pas les modifier une fois qu'on les confirme.
Les attributs à rajouter sont :

- *positionNu* : Short Integer
- *maxTurnAng* : Double
- *maxDistanc* : Double



La création d'un fichier des obstacles est presque identique. Le *feature type* choisi doit être *Polygon*.
Les obstacles ont des attributs suivants :

- *riskLevel* : Short Integer
- *typeObstac* : Text
- *message* : Text
- *SegmentNum* : Short Integer



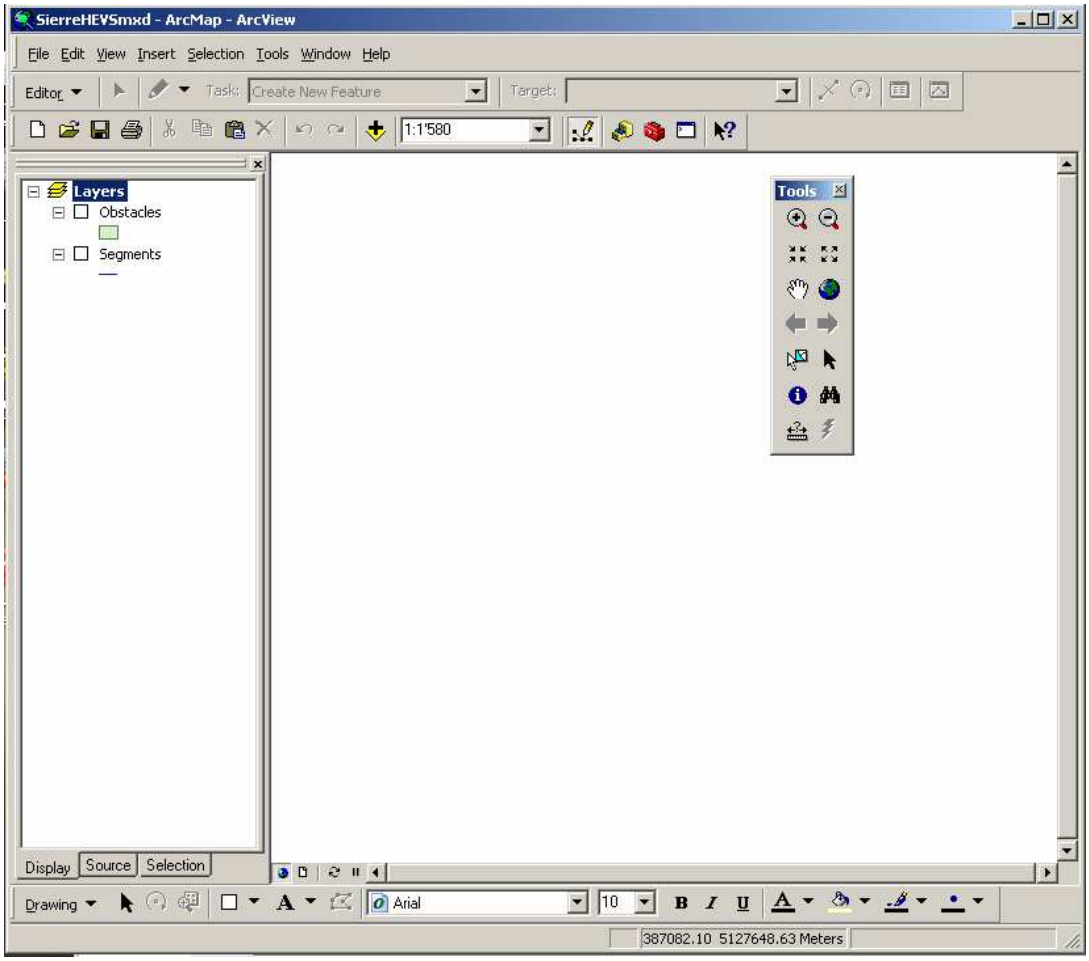
Dernier conseil pour le travail avec *ArcCatalog* : essayez de changer un peu les noms des fichiers pour chaque nouvel itinéraire. Par exemple, *SegmentsSierre*, *SegmentsCrans*, *ObstaclesCrans*, etc. C'est important car la base de données d'*ArcGIS* ne permet pas d'avoir plusieurs fichiers sous la même machine ayant les noms identiques. Dans ce cas seulement un itinéraire sera chargé chaque fois.

ArcMap

Une fois travail avec *ArcCatalog* conclu, il remplir nos fichiers des valeurs. *ArcMap* est l'outil très puissant pour la création des *SIG*. Je n'ai appris qu'un minimum des fonctionnalités nécessaires à la création d'un itinéraire. A cause de ce fait, il est possible que je ne montre pas la meilleure façon de travailler avec *ArcMap*.

Attention : pendant le travail avec *ArcMap* il faut être extrêmement attentif à suivre de près chaque instruction donnée.

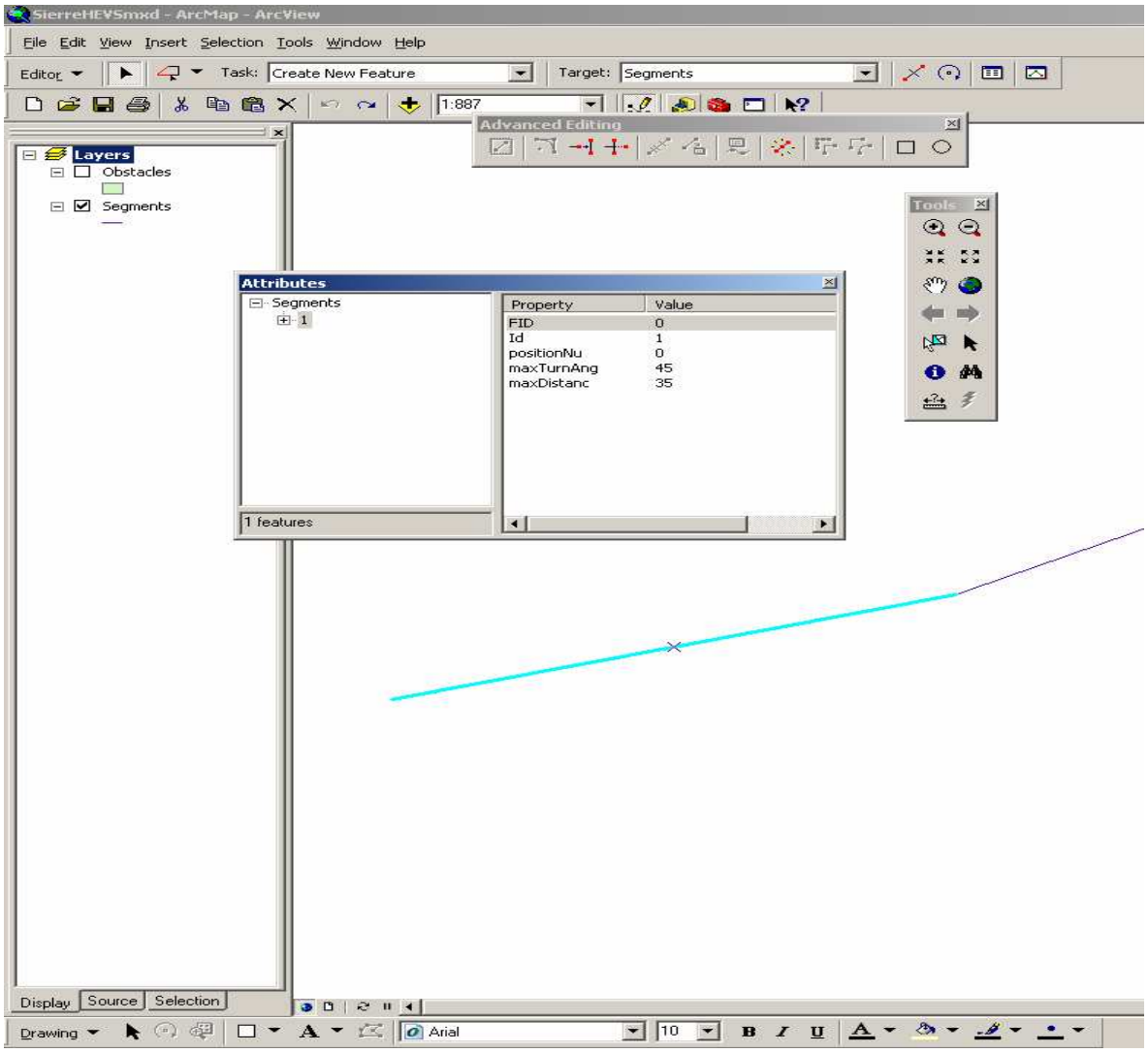
Lancez *ArcMap* et choisissez **Empty new map.** Dans le panneau droit d'édition, cliquez sur le bouton droit de la souris et choisisez **Properties**. Ensuite, dans la partie de inférieure de la fenêtre sélectionnez : **Predefined / Projected Coordinate Systems / UTM / WGS 1984 / WGS_1984_UTM_Zone32N.** Confirmer votre choix. Maintenant vous pouvez voir les coordonnées *UTM* à gauche, dans la partie inférieure du panneau. Importer maintenant les fichiers *Shapefile* comme les couches dans *ArcMap* : depuis *ArcCatalog*, draguez les fichiers un par un dans la partie gauche de *ArcMap* nommé **Layers**.



A l'aide des coordonnées affichées et de zooming trouvez la région de l'itinéraire. Dans la barre d'outils Editor sélectionnez Start Editing. Dans la liste déroulante Target sélectionner le fichier des segments.
Cliquez sur **Sketch Tool** (icône de crayon) pour commencer à réaliser les lignes des segments. Cliquez au endroit qui correspond aux coordonnées du point de départ. Tirer la ligne jusqu'aux coordonnées du point final du segment. Cliquez encore une fois pour finaliser la ligne. La ligne créée représente la ligne du premier segment de l'itinéraire.

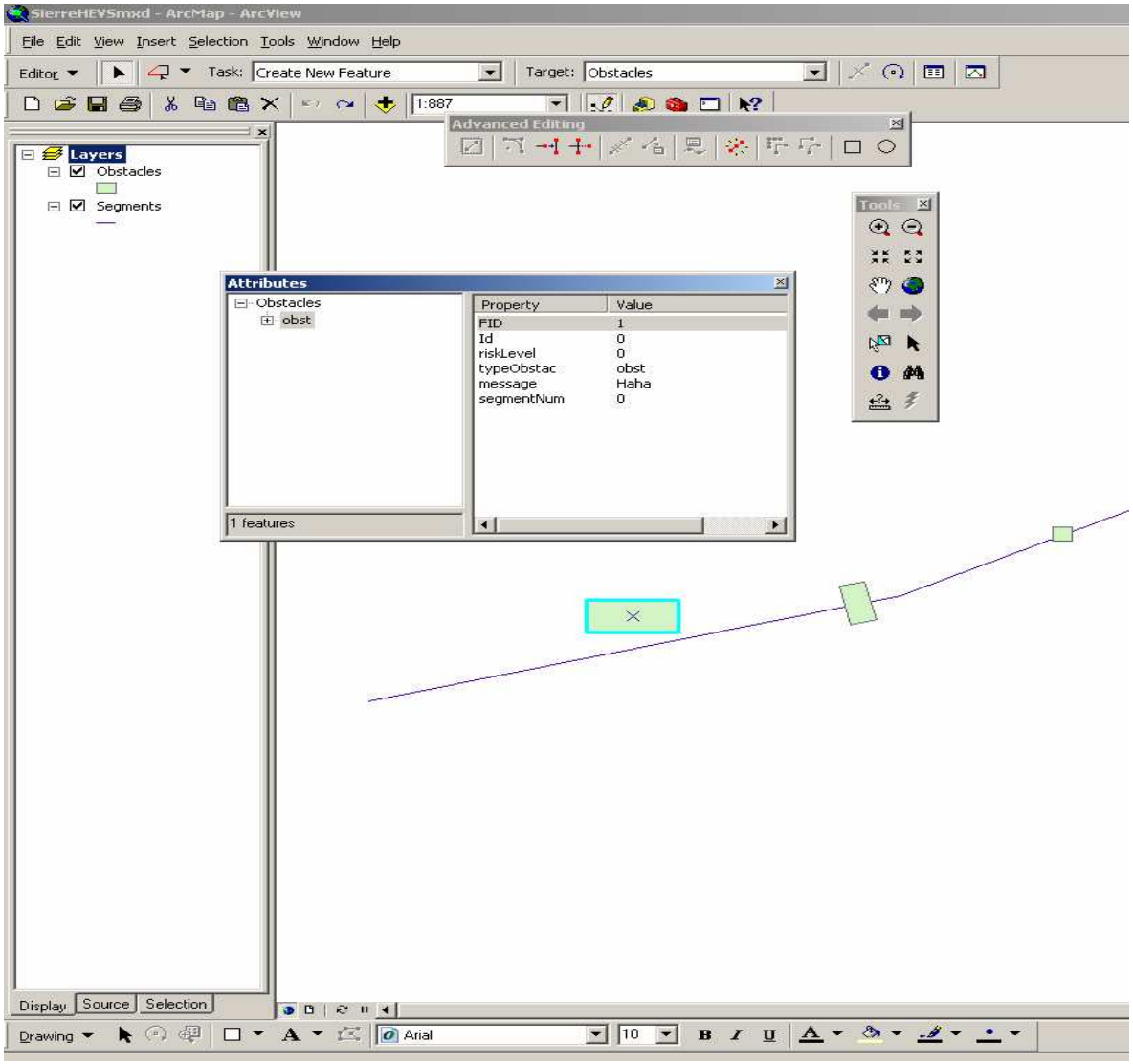
Important : chaque ligne de segment doit être orientée vers la bonne direction ! Cela signifie que la ligne doit être tirée du point d'entrée du segment vers le point de sortie du segment. Donc il faut suivre la direction et le sens de l'itinéraire. Si la ligne est dessinée à la direction inverse, le suivi dans le logiciel ne va pas fonctionner !

Choisissez **Edit Tool** (icône de flèche). Cliquez droit sur la ligne de segment et sélectionnez **Attributs**. Il faut entrer les valeurs des attributs. Le numéro de position du premier segment et 0, et on le trouve dans l'attribut *positionNu*. La numérotation commence par zéro car l'ensemble des segments créés avec *ArcMap* correspond à `segments[]` , un vecteur des segments trouvé dans la classe `Route`. La valeur de l'éloignement maximal de la ligne de segment se trouve dans le champ maxDistanc et elle est exprimée en mètres. Pour approximer cette valeur vous pouvez vous servir de l'outil **Measure** (icône de régleur) trouvé dans la boîte à outils **Toolbox**. Pour vous rappeler, c'est la distance maximale entre la ligne de segment et les bornes, donc elle est mesurée par rapport au milieu de la ligne de segment (« haut » et « bas » de la ligne). Sur la base de cette valeur, définissez la grandeur de l'angle maximale de virage de l'utilisateur dans *maxTurnAng*. Si vous n'êtes pas sur, mettez la valeur 45. On continue de créer les prochains segments de même manière. Le point de l'entrée de nouveau segment est au même temps le point de sortie du segment précédent. Pour créer la ligne du prochain segment, choisissez encore une fois **Sketch Tool**. Positionnez le pointeur sur la ligne précédemment créée, près de son point final. Cliquez sur le bouton droit et sélectionnez **Snap to Feature / Endpoint,** ce qui va lier le début du nouveau segment à la fin du précédent. Tirer la ligne jusqu'au point final du segment. Cliquez pour confirmer la création de la ligne. Entrer les valeurs des attributs correspondantes à ce segment (veillez à ne pas oublier le numéro de la position qui augmente !). Répétez toute l'opération pour chaque segment de l'itinéraire.



Si vous avez ajouté le fichier des obstacles dans *ArcView*, il faut les créer de façon suivante : d'abord dans le menu **View / Toolbars** cochez l'option **Advanced Editing**. Choisissez l'outil **Rectangle** dans la barre d'outils qui apparaît. Trouvez la position pour l'obstacle et créez le rectangle qui représente les bornes de l'obstacle (voir Rapport final). Vous pouvez le déplacer et le tourner à l'aide de l'outil **Rotation** trouvé dans la barre d'outils en haut. Il est important que l'obstacle se trouve en dedans des bornes du segment ! Il faut veillez qu'il ne sort pas même si on ne voit pas les bornes. Pour être sut, il faut essayer de dessiner les bornes avec l'outil **Measure**, ou diminuer la taille de l'obstacle.

Il faut également saisir les valeurs des attributs pour chaque obstacle. Remplissez les valeurs de *riskLevel*, *typeObstac* et *message*. Dans *message* on met le texte qui va être communiqué à l'utilisateur. **Important** : le numéro du segment dans laquelle se trouve l'obstacle (*segmentNum*) doit être correctement saisi !

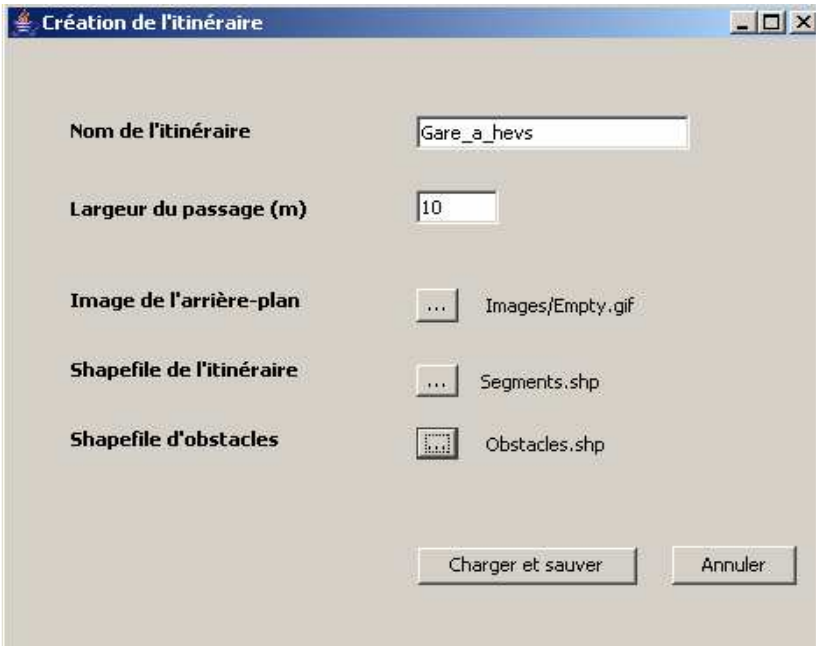


Vous pouvez sauvegarder votre travail et sortir de *ArcMap*.

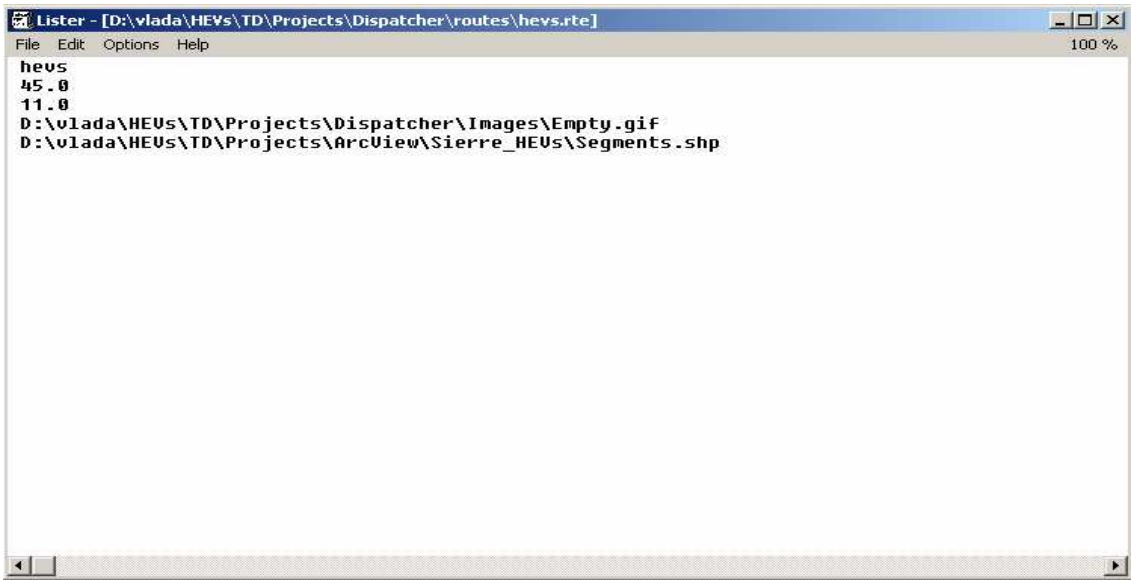
Création d'un itinéraire dans Dispatcher

Après la réalisation de l'itinéraire avec *ArcGIS*, on peut l'intégrer dans la structure de l'application *Dispatcher*.
Démarrez Dispatcher. Choisissez dans le menu **Itinéraire/Créer à partir de Shapefile,** la nouvelle fenêtre va apparaître. Saisissez le nom de l'itinéraire. **Largeur du passage** est la longueur de la ligne de sortie qui, dans cette version, a la même valeur dans chaque segment. Si sa valeur est définie comme plus grande que *maxDistance*, elle va être égalée à cette dernière. **Image de l'arrière-plan** est le fichier qui contient l'image derrière l'itinéraire, comme le plan de la région. Dans **Shapefile de l'itinéraire** il faut choisir le fichier des segments créé avec *ArcGIS*. La même chose vaut pour le fichier des obstacles qui n'est pas obligatoire.

Dans le cas de l'itinéraire inclus avec l'application, vous trouverez les fichiers *Shapefile* dans le répertoire *ArcGIS\HEVS_Sierre*. Il est conseillé d'utiliser les paramètres vus sur l'image ci-dessous. Si vous chargez un autre itinéraire, il ne faut pas oublier de modifier le code (expliqué dans ce manuel).



Confirmez les paramètres. S'il y a des problèmes, la fenêtre de l'avertissement va apparaître expliquant l'exception. Si tout se passe bien, un nouveau fichier texte sera crée, dont le nom est égale au celui de l'itinéraire, et ayant l'extension .rte . Toutes les options choisisses se trouvent dans ce fichier, et on peut le modifier à la main si on a besoin. Voici un exemple de ce fichier :



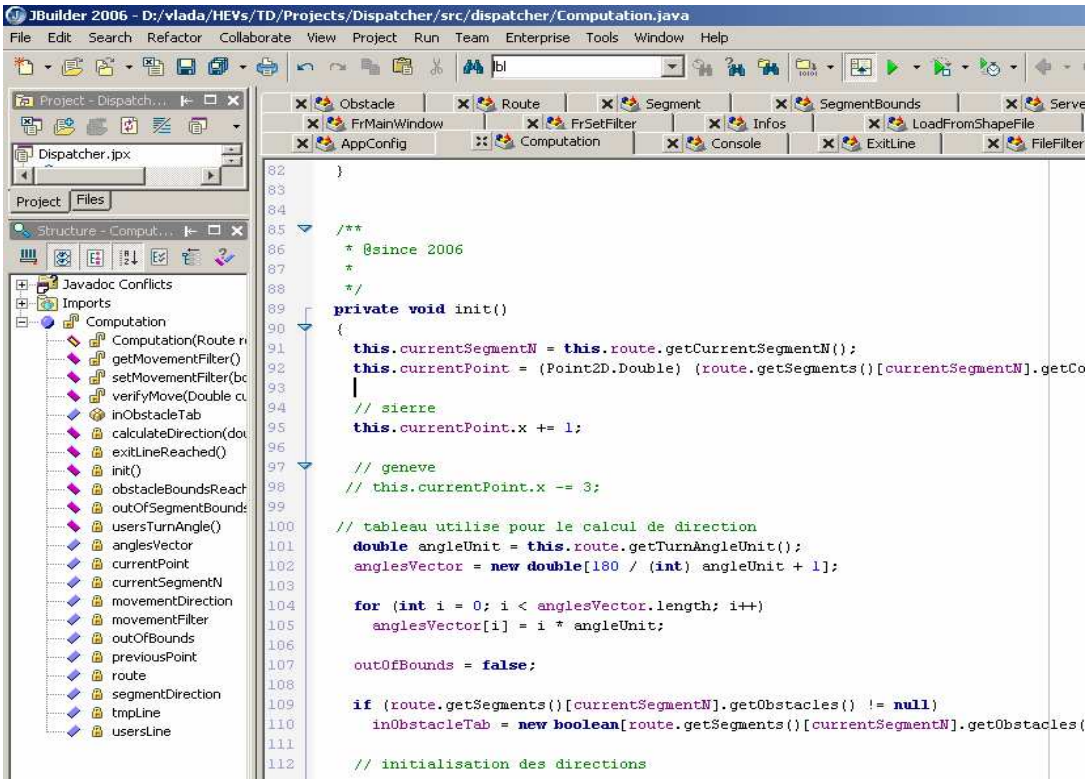
Très important : Il ne faut pas placer *Dispatcher* dans un chemin contenant des caractères spéciaux (!, ?,…), sinon l'application ne va pas fonctionner (Apache n'accepte pas ses caractères).

Adaptation du code

Il reste encore une étape à faire pour qu'on puisse exploiter le nouvel itinéraire. Dans la classe de configuration *AppConfig*, il faut modifier les valeurs des variables *UTM_DIFF_X* et *UTM_DIFF_Y*. Ces variables représentent la différence entre les vraies coordonnées UTM et les coordonnées utilisées pour l'affichage de mouvement de l'utilisateur. Comme le moniteur n'a pas assez de pixels, on va utiliser uniquement les parties inférieures (droites) des deux coordonnées. Cela ressemble le changement de l'échelle car la taille est diminuée, mais les proportions restent correctes. Par exemple, les coordonnées devant l'école sont : 387021E et 5127510N. Nous, on va les traiter comme 21E et 510N. Donc la différence est 387000 et 5127000, ce qu'il faut saisir comme *UTM_DIFF_X* et *UTM_DIFF_Y*. Dans les différents endroits les valeurs changent, donc il ne faut pas oublier de les contrôler.

```
58    public static int NOVASYS_CONNECTION_INTERVAL = 5000;
59
60
61    // geneve
62    // public static int UTM_DIFF_X = 281000;
63    // public static int UTM_DIFF_Y = 5120000;
64
65
66    // sierre
67    public static int UTM_DIFF_X = 387000;
68    public static int UTM_DIFF_Y = 5127000;
69
70    public static String[] UNITS_DEG = new String[]
```

Il y a encore une chose à changer dans le code, c'est le point de départ. Il ne faut pas commencer en dehors de l'itinéraire, et pour cela il faut adapter le premier point à la direction de l'itinéraire. On trouve cette valeur dans la classe `Computation`. Voici les exemples pour Sierre et Genève :



Utilisation d'Apache Axis

Apache Axis est un outil puissant conçu pour le développement et l'accès aux services web. Dans le cas de *Dispatcher* on en a besoin pour l'accès et l'utilisation du service web *MessageService* hébergé sur le serveur de *Novasys*. Pour accomplir cette tâche, il faut correctement installer *Axis* et utiliser l'outil wdl2java pour générer les classes nécessaires.

Installation et configuration d'*Apache Axis*

Téléchargez *axis-bin-1_4.zip* depuis http://ws.apache.org/axis. J'ai utilisé la version 1.4. Dans ce tutoriel le chemin d'installation est *c:\axis-1_4*. Vous pouvez choisir une autre destination, mais veillez que le chemin soit correct dans l'entier de l'installation. Décompressez le fichier.

Ensuite il faut mettre à jour les variables d'environnement allez à: **Panneau de configuration / Système / Avancé / Variables d'environnement**. Dans la partie **Variables système** il faut rajouter des nouvelles variables :

AXIS_HOME = c:\axis-1_4
AXIS_LIB = %AXIS_HOME%\lib
AXISCLASSPATH = %AXIS_LIB%\axis.jar;%AXIS_LIB%\commons-discovery-0.2.jar;
%AXIS_LIB%\commons-logging-1.0.4.jar; %AXIS_LIB%\jaxrpc.jar;
%AXIS_LIB%\saaj.jar; %AXIS_LIB%\log4j-1.2.8.jar;
%AXIS_LIB%\xml-apis.jar; %AXIS_LIB%\xercesImpl.jar; %AXIS_LIB%\wsdl4j-1.5.1.jar

Téléchargez le parseur *Xerces-J-bin.2.9.0.zip* depuis xerces.apache.org/xerces-j/. Il faut copier les fichiers *xercesImpl.jar* et *xml-apis.jar* trouvés dans l'archive vers *c:\axis-1_4\lib*.

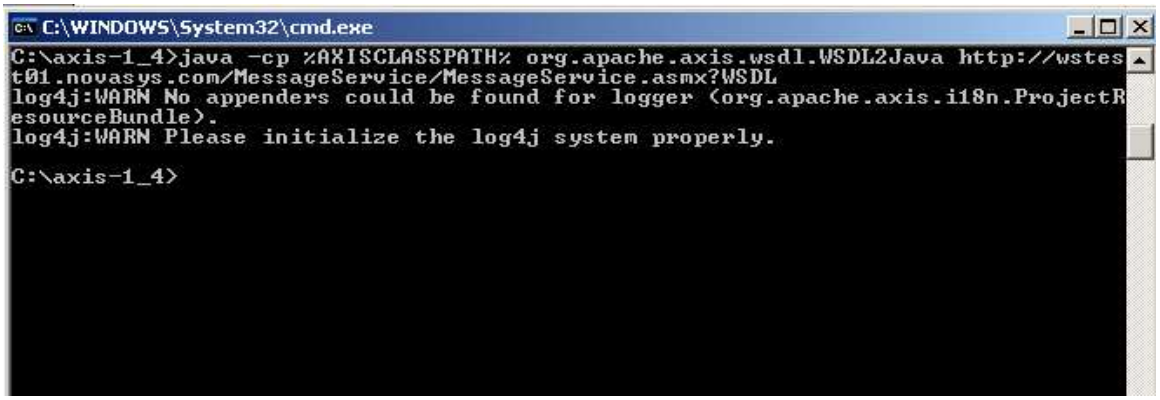
Axis est maintenant prêt pour l'utilisation.

Pour la création des classes Java il faut avoir le fichier WSDL du service en question. Dans notre cas c'est :
http://wstest01.novasys.com/MessageService/MessageService.asmx?WSDL .

Lancez la console MS-DOS et allez dans le répertoire où vous voulez mettre les classes. Tapez la commande (en une ligne !) :

java -cp %AXISCLASSPATH% org.apache.axis.wsdl.WSDL2Java http://wstest01.novasys.com/MessageService/MessageService.asmx?WSDL

Si tout s'est bien passé, l'écran doit ressembler :



Vous allez trouver la structure des répertoires créée : *com\novasys\wstest01* qui contient les fichiers :

GetMessageResponseGetMessageResult.java
MessageService.java
MessageServiceLocator.java
MessageServiceSoap.java
MessageServiceSoap12Stub.java
MessageServiceSoapStub.java
PutMessage.java
PutMessageResponse.java
PutXMLToDB.java
PutXMLToDBResponse.java

Avec *JBuilder* il faut importer cette structure dans notre projet comme un nouveau package.
Dans le code on utilise directement seulement les classes *MessageServiceLocator*, *MessageServiceSoap* et *GetMessageResponseGetMessageResult*. Cet aspect j'ai expliqué dans le rapport, alors ici je expose uniquement les lignes dans le code :

Construction :

```
MessageServiceLocator serviceLocator ;
MessageServiceSoap  serviceSoap ;
GetMessageResponseGetMessageResult result;

serviceLocator = new MessageServiceLocator();
serviceSoap = serviceLocator.getMessageServiceSoap();
```

Utilisation : nom de login est l'IMEI de Mambo doivent être connus :

```
result = serviceSoap.getMessage(LOGIN, IMEI);
```

Pour les détails de la connexion, voir le code source de la classe *ServerConnexion*.

AppConfig

Cette classe contient les constantes et les variables fréquemment utilisées dans Dispatcher. Vous pouvez la considère comme un fichier de configuration, et toutes les options peuvent être modifiées à la main.

8.4 Javadoc

dispatcher

Class Computation

```
java.lang.Object
  └dispatcher.Computation
```

```
public class Computation
extends java.lang.Object
```

Calcule les mouvements de l'utilisateur et génère les instructions.

Field Summary

private double[]	**anglesVector** Tableau contenant des multiples de Route.turnAngleUnit (unités de mouvement).
private java.awt.geom.Point2D.Double	**currentPoint** La position courante de l'utilisateur.
private int	**currentSegmentN**
private Infos	**infos**
private boolean[]	**inObstacleTab** Tableau des booléens où on peut voir si l'utilisateur se trouve dans un obstacle (valeur true).
private double	**movementDirection** Indique si l'utilisateur s'éloigne ou rapproche à la ligne du segment.
private boolean	**movementFilter** Détermine si le filtre des pas de mouvement est actif.
private boolean	**outOfBounds** La variable indiquant si l'utilisateur se trouve en dehors des bornes.
private java.awt.geom.Point2D.Double	**previousPoint** La position précédente de l'utilisateur.
private Route	**route**

	Référence vers la route actuelle.
private double	**segmentDirection** L'orientation du segment: - de gauche vers droite -> + - de droite vers gauche -> -
private java.awt.geom.Line2D.Double	**usersLine** Ligne entre previousPoint et currentPoint.

Constructor Summary

Computation(Route route) Crée une instance de Computation.

Method Summary

private void	**calculateDirection**(double angle, Infos instruction) Calcule la direction du point courant vers le point final et Génère l'instruction à donner à l'utilisateur.
private void	**calculateDirectionObstacle**(double angle, Infos instruction, Obstacle obstacle)
private ExitLine	**exitLineReached**() Vérifie si l'utilisateur a franchi une des deux lignes de sorti du segment courant.
boolean	**getMovementFilter**() Determines if the movementFilter property is true.
private void	**init**() Initialisation des variables de la classe Computation.
private Obstacle	**obstacleBoundsReached**() Contrôle si l'utilisateur se trouve dans le périmètre d'un obstacle.
private java.lang.Boolean	**outOfSegmentBounds**() Contrôle si l'utilisateur se trouve en dehors des bornes du segment.
void	**setMovementFilter**(boolean aMovementFilter) Sets the value of the movementFilter property.
private double	**usersTurnAngle**() Rétourne l'angle que le vecteur de direction courante de l'utilisateur forme avec le vecteur allant jusqu'au point final.
Infos	**verifyMove**(java.awt.geom.Point2D.Double currentPoint) La méthode principale de contrôle de mouvement.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

currentPoint

private java.awt.geom.Point2D.Double **currentPoint**

La position courante de l'utilisateur.

---

previousPoint

private java.awt.geom.Point2D.Double **previousPoint**

La position précédente de l'utilisateur.

---

usersLine

private java.awt.geom.Line2D.Double **usersLine**

Ligne entre previousPoint et currentPoint.

---

currentSegmentN

private int **currentSegmentN**

---

anglesVector

private double[] **anglesVector**

Tableau contenant des multiples de Route.turnAngleUnit (unités de mouvement). On l'utilise pour calculer les instructions pour l'utilisateur. Exemple : Si turnAngleUnit est 45 et l'instruction dit " tournez un demi tour " ,alors l'utilisateur doit tourner de 90 degrés (ou deux fois de 45 degrés).

---

inObstacleTab

private boolean[] **inObstacleTab**

Tableau des booléens où on peut voir si l'utilisateur se trouve dans un obstacle (valeur true). La taille est égale au nombre d'obstacles dans le segment courant.

---

outOfBounds

`private boolean` **`outOfBounds`**
La variable indiquant si l'utilisateur se trouve en dehors des bornes.

---

movementFilter

`private boolean` **`movementFilter`**
Détermine si le filtre des pas de mouvement est actif.

---

segmentDirection

`private double` **`segmentDirection`**
L'orientation du segment: - de gauche vers droite -> + - de droite vers gauche -> -

---

movementDirection

`private double` **`movementDirection`**
Indique si l'utilisateur s'éloigne ou rapproche à la ligne du segment.

---

route

`private Route` **`route`**
Référence vers la route actuelle.

---

infos

`private Infos` **`infos`**

Constructor Detail

Computation

`public` **`Computation`**`(Route route)`
Crée une instance de Computation. Crée la liaison entre route locale et route donnée comme argument. La plupart des opérations d'initialisation est effectuée dans init().
Parameters:
`route` - - Instance de Route. C'est l'itinéraire utilisé.

Method Detail

getMovementFilter

`public boolean` **`getMovementFilter`**`()`

Determines if the movementFilter property is true.

Returns:

`true if the movementFilter property is true`

---

setMovementFilter

`public void` **`setMovementFilter`**`(boolean aMovementFilter)`

Sets the value of the movementFilter property.

Parameters:

`aMovementFilter` - the new value of the movementFilter property

---

init

`private void` **`init`**`()`

Initialisation des variables de la classe Computation. La méthode est appelée pendant la instanciation et après le changement du segment. Par défaut, la valeur de previousPoint est égale au point de départ u segment courant. Le tableau anglesVecteur[] est rempli avec des multiples de turnAngleUnit. La taille de ce tableau dépend de turnAngleUnit. Les valeurs de inObstacleTab[] sont mises à faux, c'est-à-dire l'utilisateur ne se trouve dans aucun obstacle. outOfBounds est également mis à faux.

---

verifyMove

`public` Infos **`verifyMove`**`(java.awt.geom.Point2D.Double currentPoint)`

La méthode principale de contrôle de mouvement. Le point courant est comparé au point précédent afin de trouver la direction. Cette méthode fait appel aux autres méthodes de la classe afin de trouver si l'utilisateur est dépassé la limite, s'il toujours marche vers la fin du segment (de l'itinéraire) ou si il y a un obstacle devant lui. Dépendant des résultats obtenus, l'instruction appropriée est passée au module principal de l'application. Si le filtre de mouvement est actif, les pas (différences entre deux points) plus petits que MIN_MOVEMENT ne sont pas traités. Return value: Infos avec les nouvelles données ou null si le point a été refusé par le filtre de mouvement.

Parameters:

`currentPoint` - - Nouvelle position de l'utilisateur.

Returns:

Infos

---

exitLineReached

`private ExitLine` **`exitLineReached`**`()`

Vérifie si l'utilisateur a franchi une des deux lignes de sorti du segment courant. Si oui, le contrôle passe au segment voisin, précédent au suivant - dépend de sens de mouvement. Si l'utilisateur est sorti de l'itinéraire (point de départ ou point d'arrivée) il est considéré comme dehors des bornes - donc il n'y a pas de suivi. Return value : La ligne franchie ou null.

Returns:
ExitLine

---

obstacleBoundsReached

`private Obstacle` **`obstacleBoundsReached`**`()`

Contrôle si l'utilisateur se trouve dans le périmètre d'un obstacle. Dans ce cas la référence vers cet obstacle est mise dans Infos. On traverse les bornes de chaque obstacle afin de trouver si l'utilisateur se trouve dedans. Si l'utilisateur franchit les bornes d'un obstacle ou il se trouve déjà, cela signifie qu'il est sorti de cet obstacle. Pour cette opération on utilise le vecteur inObstacleTab[]. Les contrôles sont mis en places afin d'éviter les problèmes dans le situations spécifiques. Par exemple : la sortie d'un obstacle et l'entrée dans un autre d'un coup; d'un coup l'utilisateur entre et sort d'un même obstacle etc. Return value : L'obstacle ou null.

Returns:
Obstacle

---

outOfSegmentBounds

`private java.lang.Boolean` **`outOfSegmentBounds`**`()`

Contrôle si l'utilisateur se trouve en dehors des bornes du segment. Si oui, le message correspondant est place dans retInfos et le flag Infos.outOfSegmentBounds est mis à true. La méthode utilise la variable outOfBounds.

Returns:
Boolean

---

usersTurnAngle

`private double` **`usersTurnAngle`**`()`

Rétourne l'angle que le vecteur de direction courante de l'utilisateur forme avec le vecteur allant jusqu'au point final. Return value : l'angle en degrés.

Returns:
double

---

calculateDirection

```
private void calculateDirection(double angle,
                                Infos instruction)
```

Calcule la direction du point courant vers le point final et Génère l'instruction à donner à l'utilisateur. L'angle courant de l'utilisateur par rapport au point final doit être passé comme l'argument. Dans le tableau anglesVector[] on cherche la valeur la plus proche à l'angle donné et on la met dans le paramètre instruction.

Parameters:

`angle` - - l'angle entre le vecteur de l'utilisateur et le vecteur allant jusqu'au point final.

`instruction` - - Résultats de calcul

---

calculateDirectionObstacle

```
private void calculateDirectionObstacle(double angle,
                                        Infos instruction,
                                        Obstacle obstacle)
```

Parameters:

`angle` -

`instruction` -

`obstacle` -

Since:

2006

Package Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS **NEXT CLASS** | **FRAMES** **NO FRAMES** **All Classes** **All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

---

dispatcher

Class ExitLine

```
java.lang.Object
  └ java.awt.geom.Line2D
      └ java.awt.geom.Line2D.Double
          └ dispatcher.ExitLine
```

All Implemented Interfaces:

java.awt.Shape, java.lang.Cloneable

```
public class ExitLine
extends java.awt.geom.Line2D.Double
```

Cette classe est utilisée pour représenter les lignes de sortie d'un segment.

Nested Class Summary

Nested classes/interfaces inherited from class java.awt.geom.Line2D
java.awt.geom.Line2D.Double, java.awt.geom.Line2D.Float

Field Summary

private int	**nextSegmentN** Si exitLine1, c'est le numéro du segment précédent.

Fields inherited from class java.awt.geom.Line2D.Double
x1, x2, y1, y2

Constructor Summary

ExitLine() Création d'une nouvelle instance de ExitLine avec P1(0,0) et P2(0,0) comme les coordonnées. nextSegmentN est initialisé à -1.
ExitLine(ExitLine exitLine) Constructeur par copie.

ExitLine(ExitLine exitLine, int nextSegmentN)
ExitLine(java.awt.geom.Point2D P1, java.awt.geom.Point2D P2, int nextSegmentN) Construction et initialisation de ExitLine à partir des paramètres.

Method Summary

int	**getNextSegmentN**() Access method for the nextSegmentN property.
void	**setNextSegmentN**(int aNextSegmentN) Sets the value of the nextSegmentN property.

Methods inherited from class java.awt.geom.Line2D.Double
getBounds2D, getP1, getP2, getX1, getX2, getY1, getY2, setLine

Methods inherited from class java.awt.geom.Line2D
clone, contains, contains, contains, contains, getBounds, getPathIterator, getPathIterator, intersects, intersects, intersectsLine, intersectsLine, linesIntersect, ptLineDist, ptLineDist, ptLineDist, ptLineDistSq, ptLineDistSq, ptLineDistSq, ptSegDist, ptSegDist, ptSegDist, ptSegDistSq, ptSegDistSq, ptSegDistSq, relativeCCW, relativeCCW, relativeCCW, setLine, setLine

Methods inherited from class java.lang.Object
equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

nextSegmentN

```
private int nextSegmentN
```

Si exitLine1, c'est le numéro du segment précédent. Si exitLine2, c'est le numéro du segment prochain.

Constructor Detail

ExitLine

```
public ExitLine(ExitLine exitLine,
                int nextSegmentN)
```

Parameters:

exitLine -
nextSegmentN -

ExitLine

```
public ExitLine(ExitLine exitLine)
```

Constructeur par copie.

Parameters:

exitLine -

ExitLine

```
public ExitLine(java.awt.geom.Point2D P1,
                java.awt.geom.Point2D P2,
                int nextSegmentN)
```

Construction et initialisation de ExitLine à partir des paramètres.

Parameters:

P1 -
P2 -
nextSegmentN -

ExitLine

```
public ExitLine()
```

Création d'une nouvelle instance de ExitLine avec P1(0,0) et P2(0,0) comme les coordonnées. nextSegmentN est initialisé à -1.

Method Detail

getNextSegmentN

```
public int getNextSegmentN()
```

Access method for the nextSegmentN property.

Returns:

the current value of the nextSegmentN property

---

setNextSegmentN

```
public void setNextSegmentN(int aNextSegmentN)
```

Sets the value of the nextSegmentN property.

Parameters:

aNextSegmentN - the new value of the nextSegmentN property

dispatcher

Class Infos

```
java.lang.Object
  └dispatcher.Infos
```

```
public class Infos
extends java.lang.Object
```

Infos contient les informations sur le dernier mouvement de l'utilisateur et sur sa position courante. Les données sont charges par les méthodes de Computation. On interprète les données dans FrMainWindow.

Field Summary

private java.lang.String	**message** Le message communiqué à l'utilisateur.
private boolean	**outOfSegmentBounds** Si l'utilisateur se trouve en dehors des bornes du segment, la valeur de outOfSegmentBounds est true, si non, c'est false.
private ExitLine	**touchedExitLine** Si l'utilisateur a franchi une ExitLine (ligne de sortie), contient la référence vers cette ligne, si non, la valeur est null.
private Obstacle	**touchedObstacle** Si l'utilisateur se trouve dans le périmètre d'un obstacle (il a traversé les bornes), c'est la référence vers cet obstacle, si non, la valeur est null.
private double	**turnAngle** Si les calculs dans Computation montrent que l'utilisateur ne va pas dans la bonne direction, c'est-à-dire il s'éloigne du point final, un angle de redirection est calculé.
private java.awt.geom.Line2D.Double	**usersLine** La ligne décrite par les points précédent et courant de l'utilisateur.

Constructor Summary

Infos() Création d'une instance de Infos.

Method Summary

void	**add**(java.lang.String msg) Ajout d'un message dans Info.
void	**clearInfos**() Met tous les variables à 0 et null.
java.lang.String	**getMessage**() Access method for the message property.
boolean	**getOutOfSegmentBounds**() Determines if the outOfSegmentBounds property is true.
ExitLine	**getTouchedExitLine**() Access method for the touchedExitLine property.
Obstacle	**getTouchedObstacle**() Access method for the touchedObstacle property.
double	**getTurnAngle**() Access method for the turnAngle property.
java.awt.geom.Line2D.Double	**getUsersLine**() Access method for the usersLine property.
void	**setMessage**(java.lang.String aMessage) Sets the value of the message property.
void	**setOutOfSegmentBounds**(boolean aOutOfSegmentBounds) Sets the value of the outOfSegmentBounds property.
void	**setTouchedExitLine**(ExitLine aTouchedExitLine) Sets the value of the touchedExitLine property.
void	**setTouchedObstacle**(Obstacle aTouchedObstacle) Sets the value of the touchedObstacle property.
void	**setTurnAngle**(double aTurnAngle) Sets the value of the turnAngle property.
void	**setUsersLine**(java.awt.geom.Line2D.Double aUsersLine) Sets the value of the usersLine property.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

message

`private java.lang.String` **`message`**

Le message communiqué à l'utilisateur. Cela peut être une instruction comme " Tournez d'un quart de tour à gauche " ou un avertissement comme " Vous êtes en dehors de limites ". Les messages se trouvent dans la classe AppConfig.

---

touchedExitLine

`private ExitLine` **`touchedExitLine`**

Si l'utilisateur a franchi une ExitLine (ligne de sortie), contient la référence vers cette ligne, si non, la valeur est null.

---

touchedObstacle

`private Obstacle` **`touchedObstacle`**

Si l'utilisateur se trouve dans le périmètre d'un obstacle (il a traversé les bornes), c'est la référence vers cet obstacle, si non, la valeur est null.

---

turnAngle

`private double` **`turnAngle`**

Si les calculs dans Computation montrent que l'utilisateur ne va pas dans la bonne direction, c'est-à-dire il s'éloigne du point final, un angle de redirection est calculé. Si l'utilisateur marche vers le point final, la valeur est 0 - il n'y a pas de correction.

---

outOfSegmentBounds

`private boolean` **`outOfSegmentBounds`**

Si l'utilisateur se trouve en dehors des bornes du segment, la valeur de outOfSegmentBounds est true, si non, c'est false. Une fois l'utilisateur est en dehors du segment, le suivi par l'ordinateur s'arrête car la sécurité ne peut plus être garantie.

---

usersLine

`private java.awt.geom.Line2D.Double` **`usersLine`**

La ligne décrite par les points précédent et courant de l'utilisateur.

Constructor Detail

Infos

`public` **`Infos`**`()`

Création d'une instance de Infos. L'appel à clearInfos() est fait.

Method Detail

getMessage

`public java.lang.String` **`getMessage`**`()`

Access method for the message property.

Returns:

the current value of the message property

---

setMessage

`public void` **`setMessage`**`(java.lang.String aMessage)`

Sets the value of the message property.

Parameters:

`aMessage` - the new value of the message property

---

getTouchedExitLine

`public` `ExitLine` **`getTouchedExitLine`**`()`

Access method for the touchedExitLine property.

Returns:

the current value of the touchedExitLine property

---

setTouchedExitLine

`public void` **`setTouchedExitLine`**`(``ExitLine`` aTouchedExitLine)`

Sets the value of the touchedExitLine property.

Parameters:

`aTouchedExitLine` - the new value of the touchedExitLine property

---

getTouchedObstacle

`public` `Obstacle` **`getTouchedObstacle`**`()`

Access method for the touchedObstacle property.

Returns:
the current value of the touchedObstacle property

---

setTouchedObstacle

```
public void setTouchedObstacle(Obstacle aTouchedObstacle)
```

Sets the value of the touchedObstacle property.

Parameters:
`aTouchedObstacle` - the new value of the touchedObstacle property

---

getTurnAngle

```
public double getTurnAngle()
```

Access method for the turnAngle property.

Returns:
the current value of the turnAngle property

---

setTurnAngle

```
public void setTurnAngle(double aTurnAngle)
```

Sets the value of the turnAngle property.

Parameters:
`aTurnAngle` - the new value of the turnAngle property

---

getOutOfSegmentBounds

```
public boolean getOutOfSegmentBounds()
```

Determines if the outOfSegmentBounds property is true.

Returns:
`true if the outOfSegmentBounds property is true`

---

setOutOfSegmentBounds

```
public void setOutOfSegmentBounds(boolean aOutOfSegmentBounds)
```

Sets the value of the outOfSegmentBounds property.

Parameters:
`aOutOfSegmentBounds` - the new value of the outOfSegmentBounds property

---

getUsersLine

`public java.awt.geom.Line2D.Double` **`getUsersLine`**`()`

Access method for the usersLine property.

Returns:

the current value of the usersLine property

---

setUsersLine

`public void` **`setUsersLine`**`(java.awt.geom.Line2D.Double aUsersLine)`

Sets the value of the usersLine property.

Parameters:

`aUsersLine` - the new value of the usersLine property

---

add

`public void` **`add`**`(java.lang.String msg)`

Ajout d'un message dans Info.

Parameters:

`msg` - - Message à rajouter.

---

clearInfos

`public void` **`clearInfos`**`()`

Met tous les variables à 0 et null.

Package Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES** **All Classes** **All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

---

dispatcher

Class LoadFromShapeFile

```
java.lang.Object
  └ dispatcher.LoadFromShapeFile
```

```
public class LoadFromShapeFile
extends java.lang.Object
```

Avec les méthodes de cette classe on lit et on recherche dans les fichiers Shapefile les objets (Features) créés avec ArcGIS. Ce sont principalement les segments et les obstacles qu'on charge avec les méthodes loadSegments() et loadObstacles(). Pour lire un Shapefile on utilise les classes des librairies GeoTools. Le principe est suivant : tous les objets d'un Shapefile (segments par exemple) doivent être traversés avec un Iterator, et les attributs trouvés (positionNumber, maxDistance,…) sont utilisés pour instancier des classes correspondantes (Segment).

Constructor Summary

LoadFromShapeFile()

Method Summary

void	**LoadFromFile**()
(package private) static void	**loadObstacles**(File obstaclesFile, Route route) Essaie de charger le fichier donné comme argument.
(package private) static void	**loadSegments**(File segmentsFile, Route route) Essaie de charger le fichier donné comme argument.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

LoadFromShapeFile

```
public LoadFromShapeFile()
```

Method Detail

loadSegments

```
static void loadSegments(File segmentsFile,
                         Route route)
                  throws java.lang.Exception
```

Essaie de charger le fichier donné comme argument. Les objets d'un fichier Shapefile ont leurs équivalents dans la classe Features de GeoTools, qui représentent les segments créés dans ArcGIS. Pour chaque Feature du fichier, on sort des attributs attendus (pour le fichier de segments ce sont positionNumber, maxTurnAngle,…) qu'on utilise pour créer une nouvelle instance de Segment. Une fois tous les segments créés, on les associe à Route donné comme l'argument. loadSegments() déclenche l'exception si les attributs attendus ne sont pas trouvés. La même chose vaut dans le cas où le nom de Feature n'est pas " Segment ".

Parameters:

`segmentsFile` - - Fichier Shapefile contenant les segments.

`route` - - Route qui va contenir les segments chargés.

Throws:

`java.lang.Exception`

---

loadObstacles

```
static void loadObstacles(File obstaclesFile,
                          Route route)
                   throws java.lang.Exception
```

Essaie de charger le fichier donné comme argument. Les objets d'un fichier Shapefile ont leurs équivalents dans la classe Features de GeoTools, qui représentent les segments créés dans ArcGIS. Pour chaque Feature du fichier, on sort des attributs attendus (pour le fichier d'obstacles ce sont riskLevel, typeObstacle,…) qu'on utilise pour créer une nouvelle instance de Obstacle. Une fois tous les obstacles créés, on les associe à Route donné comme l'argument. loadObstacles() déclenche l'exception si les attributs attendus ne sont pas trouvés. La même chose vaut dans le cas où le nom de Feature n'est pas " Obstacle ".

Parameters:

`obstaclesFile` - - Fichier Shapefile contenant les obstacles.

`route` - - Route qui va contenir les obstacles chargés.

Throws:

`java.lang.Exception`

LoadFromFile

```
public void LoadFromFile()
```

Since:
2006

dispatcher

Class Route

```
java.lang.Object
  └ dispatcher.Route
```

```
public class Route
extends java.lang.Object
```

Field Summary

private int	**currentSegmentN** Le numéro du segment courant.
private double	**length**
private java.lang.String	**name** Nom de Route.
private double	**passWidth** La taille du passage entre deux segments.
private Segment[]	**segments** Les segments de cet itinéraire.
private double	**turnAngleUnit** Unité de mouvement du message communiqué à l'utilisateur (degrés).

Constructor Summary

Route()
Route(java.lang.String name, double turnAngleUnit, double passWidth) Création d'une instance de Route.

Method Summary

int	**getCurrentSegmentN**() Access method for the currentSegmentN property.
double	**getLength**()

	Access method for the length property.
java.lang.String	**getName**() Access method for the name property.
double	**getPassWidth**() Access method for the passWidth property.
Segment[]	**getSegments**() Access method for the segments property.
double	**getTurnAngleUnit**() Access method for the turnAngleUnit property.
void	**setCurrentSegmentN**(int aCurrentSegmentN) Sets the value of the currentSegmentN property.
void	**setLength**(double aLength) Sets the value of the length property.
void	**setName**(java.lang.String aName) Sets the value of the name property.
void	**setPassWidth**(double aPassWidth) Sets the value of the passWidth property.
void	**setSegments**(Segment[] aSegments) Sets the value of the segments property.
void	**setTurnAngleUnit**(double aTurnAngleUnit) Sets the value of the turnAngleUnit property.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

name

```
private java.lang.String name
```

Nom de Route.

---

length

```
private double length
```

---

passWidth

`private double` **`passWidth`**
La taille du passage entre deux segments. Cette valeur vaut pour tous les segments de cette Route.

---

turnAngleUnit

`private double` **`turnAngleUnit`**
Unité de mouvement du message communiqué à l'utilisateur (degrés).

---

currentSegmentN

`private int` **`currentSegmentN`**
Le numéro du segment courant.

---

segments

`private Segment[]` **`segments`**
Les segments de cet itinéraire.

Constructor Detail

Route

```
public Route(java.lang.String name,
             double turnAngleUnit,
             double passWidth)
```
Création d'une instance de Route. Les segments[] peuvent être rajouter après l'instanciation.
Parameters:
`name` - - Nom de cette Route.
`turnAngleUnit` - - Valeur de turnAngleUnit.
`passWidth` - - Largeur du passage.

Route

`public Route()`
Since:
2006

Method Detail

getName

```
public java.lang.String getName()
```

Access method for the name property.

Returns:

the current value of the name property

---

setName

```
public void setName(java.lang.String aName)
```

Sets the value of the name property.

Parameters:

`aName` - the new value of the name property

---

getLength

```
public double getLength()
```

Access method for the length property.

Returns:

the current value of the length property

---

setLength

```
public void setLength(double aLength)
```

Sets the value of the length property.

Parameters:

`aLength` - the new value of the length property

---

getPassWidth

```
public double getPassWidth()
```

Access method for the passWidth property.

Returns:

the current value of the passWidth property

---

setPassWidth

```
public void setPassWidth(double aPassWidth)
```

Sets the value of the passWidth property.

Parameters:

`aPassWidth` - the new value of the passWidth property

---

getTurnAngleUnit

```
public double getTurnAngleUnit()
```

Access method for the turnAngleUnit property.

Returns:

the current value of the turnAngleUnit property

---

setTurnAngleUnit

```
public void setTurnAngleUnit(double aTurnAngleUnit)
```

Sets the value of the turnAngleUnit property.

Parameters:

`aTurnAngleUnit` - the new value of the turnAngleUnit property

---

getCurrentSegmentN

```
public int getCurrentSegmentN()
```

Access method for the currentSegmentN property.

Returns:

the current value of the currentSegmentN property

---

setCurrentSegmentN

```
public void setCurrentSegmentN(int aCurrentSegmentN)
```

Sets the value of the currentSegmentN property.

Parameters:

`aCurrentSegmentN` - the new value of the currentSegmentN property

---

getSegments

```
public Segment[] getSegments()
```

Access method for the segments property.

Returns:

the current value of the segments property

---

setSegments

```
public void setSegments(Segment[] aSegments)
```

Sets the value of the segments property.

Parameters:

`aSegments` - the new value of the segments property

dispatcher

Class Segment

```
java.lang.Object
  └ dispatcher.Segment
```

```
public class Segment
extends java.lang.Object
```

Représente le segment de l'itinéraire sous la forme d'une ligne droite. Un segment contient deux lignes d'entrée et de sortie, la ligne de l'itinéraire et les bornes. Les bornes forment avec la ligne de segment un quasi-triangle isocèle à chaque coté de la ligne. Les sommets des triangles (hauteurs) sont représentés avec vertex1 et vertex2.

Field Summary

private SegmentBounds	**bounds** Les bornes du segment.
private java.awt.geom.Line2D.Double	**coords** Coordonnées de ce segment (ligne de segment).
private ExitLine	**exitLine1** S'il ne s'agit pas du premier segment, la ligne de sortie exitLine1 se trouve sur la bissectrice de l'angle formé par le ce segment et le segment précédent.
private ExitLine	**exitLine2** S'il ne s'agit pas du premier segment, la ligne de sortie exitLine1 se trouve sur la bissectrice de l'angle formé par le ce segment et le segment précédent.
private double	**maxDistance** La distance maximale de la ligne de segment.
private double	**maxTurnAngle** L'angle maximum permis qui forme le vecteur de mouvement de l'utilisateur avec le vecteur allant jusqu'au point final (voir la documentation).
private Obstacle[]	**obstacles**
private int	**positionNumber**

	Numéro du segment.
private Segment	**previousSegment**
private java.awt.geom.Point2D.Double	**vertex1** Le premier point se trouvant à maxDistance éloigné du milieu de la ligne de segment.
private java.awt.geom.Point2D.Double	**vertex2** Le deuxième point se trouvant à maxDistance éloigné du milieu de la ligne de segment.

Constructor Summary

Segment() Constructeur par défaut.
Segment(java.awt.geom.Line2D.Double coords, int positionNumber, int totalSegments, double maxDistance, double maxTurnAngle, double passWidth, Segment previousSegment) Construction et initialisation de Segment à partir des paramètres.

Method Summary

private void	**calculateExitLine**(double passWidth) Calcul et création de ligne de sortie ayant la longueur de passWidth défini dans Route.
(package private) void	**calculateVertices**() Calcul des points de sommet (haut et bas).
private void	**createBounds**() Création des bornes du segment.
SegmentBounds	**getBounds**() Access method for the bounds property.
java.awt.geom.Line2D.Double	**getCoords**() Access method for the coords property.
ExitLine	**getExitLine1**() Access method for the exitLine1 property.
ExitLine	**getExitLine2**() Access method for the exitLine2 property.
double	**getMaxDistance**() Access method for the maxDistance property.
double	**getMaxTurnAngle**() Access method for the maxTurnAngle property.
Obstacle[]	**getObstacles**() Access method for the obstacles property.

int	**getPositionNumber**() Access method for the positionNumber property.
java.awt.geom.Point2D.Double	**getVertex1**() Access method for the vertex1 property.
java.awt.geom.Point2D.Double	**getVertex2**() Access method for the vertex2 property.
void	**setCoords**(java.awt.geom.Line2D.Double aCoords) Sets the value of the coords property.
void	**setExitLine1**(ExitLine aExitLine1) Sets the value of the exitLine1 property.
void	**setExitLine2**(ExitLine aExitLine2) Sets the value of the exitLine2 property.
void	**setMaxDistance**(double aMaxDistance) Sets the value of the maxDistance property.
void	**setMaxTurnAngle**(double aMaxTurnAngle) Sets the value of the maxTurnAngle property.
void	**setObstacles**(Obstacle[] aObstacles) Sets the value of the obstacles property.
void	**setPositionNumber**(int aPositionNumber) Sets the value of the positionNumber property.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

coords

```
private java.awt.geom.Line2D.Double coords
```

Coordonnées de ce segment (ligne de segment).

---

positionNumber

```
private int positionNumber
```

Numéro du segment. C'est sa position dans la route.

---

maxTurnAngle

```
private double maxTurnAngle
```

L'angle maximum permis qui forme le vecteur de mouvement de l'utilisateur avec le vecteur allant jusqu'au point final (voir la documentation).

---

maxDistance

`private double` **`maxDistance`**

La distance maximale de la ligne de segment. Utilisé pour calculer les bornes et les sommets (vertex) du segment.

---

vertex1

`private java.awt.geom.Point2D.Double` **`vertex1`**

Le premier point se trouvant à maxDistance éloigné du milieu de la ligne de segment.

---

vertex2

`private java.awt.geom.Point2D.Double` **`vertex2`**

Le deuxième point se trouvant à maxDistance éloigné du milieu de la ligne de segment.

---

previousSegment

`private Segment` **`previousSegment`**

---

obstacles

`private Obstacle[]` **`obstacles`**

---

bounds

`private SegmentBounds` **`bounds`**

Les bornes du segment. L'utilisateur n'est pas suivi dehors de la zone limitée par les bornes. Les bornes forment deux quasi-triangles (à cause de lignes de sortie) symétriques par rapport à la ligne de segment. Les quasi-triangles ont les sommets dans les vertex1 et vertex2.

---

exitLine2

`private ExitLine` **`exitLine2`**

S'il ne s'agit pas du premier segment, la ligne de sortie exitLine1 se trouve sur la bissectrice de l'angle formé par le ce segment et le segment précédent. S'il s'agit du premier / dernier segment, la première / deuxième ligne de sortie est perpendiculaire à la ligne de segment. La longueur de chaque exitLine est la valeur de passWidth.

---

exitLine1

```
private ExitLine exitLine1
```

S'il ne s'agit pas du premier segment, la ligne de sortie exitLine1 se trouve sur la bissectrice de l'angle formé par le ce segment et le segment précédent. S'il s'agit du premier / dernier segment, la première / deuxième ligne de sortie est perpendiculaire à la ligne de segment. La longueur de chaque exitLine est la valeur de passWidth.

Constructor Detail

Segment

```
public Segment(java.awt.geom.Line2D.Double coords,
               int positionNumber,
               int totalSegments,
               double maxDistance,
               double maxTurnAngle,
               double passWidth,
               Segment previousSegment)
```

Construction et initialisation de Segment à partir des paramètres. Assure le contrôle de la taille de maxDistance. Les autres éléments sont créés avec calculateExitLine() et calculateVertices().

Parameters:

`coords` - - Coordonnées de la ligne de ce segment.

`positionNumber` - - Le numéro de position du segment.

`totalSegments` - - Nombre total de segments.

`maxDistance` - - Distance maximale de l'éloignement de la ligne du segment. Utilisé pour le calcul de sommets (vertex) et des bornes du segment.

`maxTurnAngle` - - Voir le membre maxTurnAngle.

`passWidth` - - Voir le membre passWidth.

`previousSegment` - - La référence vers le segment précédent.

Segment

```
public Segment()
```

Constructeur par défaut. Il faut plutôt utiliser l'autre constructeur plus explicit.

Method Detail

getCoords

```
public java.awt.geom.Line2D.Double getCoords()
```

Access method for the coords property.

Returns:

the current value of the coords property

---

setCoords

```
public void setCoords(java.awt.geom.Line2D.Double aCoords)
```

Sets the value of the coords property.

Parameters:

`aCoords` - the new value of the coords property

---

getPositionNumber

```
public int getPositionNumber()
```

Access method for the positionNumber property.

Returns:

the current value of the positionNumber property

---

setPositionNumber

```
public void setPositionNumber(int aPositionNumber)
```

Sets the value of the positionNumber property.

Parameters:

`aPositionNumber` - the new value of the positionNumber property

---

getMaxTurnAngle

```
public double getMaxTurnAngle()
```

Access method for the maxTurnAngle property.

Returns:

the current value of the maxTurnAngle property

---

setMaxTurnAngle

```
public void setMaxTurnAngle(double aMaxTurnAngle)
```

Sets the value of the maxTurnAngle property.

Parameters:

`aMaxTurnAngle` - the new value of the maxTurnAngle property

---

getMaxDistance

```
public double getMaxDistance()
```

Access method for the maxDistance property.

Returns:

the current value of the maxDistance property

---

setMaxDistance

```
public void setMaxDistance(double aMaxDistance)
```

Sets the value of the maxDistance property.

Parameters:

`aMaxDistance` - the new value of the maxDistance property

---

getVertex1

```
public java.awt.geom.Point2D.Double getVertex1()
```

Access method for the vertex1 property.

Returns:

the current value of the vertex1 property

---

getVertex2

```
public java.awt.geom.Point2D.Double getVertex2()
```

Access method for the vertex2 property.

Returns:

the current value of the vertex2 property

---

getObstacles

```
public Obstacle[] getObstacles()
```

Access method for the obstacles property.

Returns:

the current value of the obstacles property

---

setObstacles

```
public void setObstacles(Obstacle[] aObstacles)
```

Sets the value of the obstacles property.

Parameters:

`aObstacles` - the new value of the obstacles property

---

getBounds

`public SegmentBounds getBounds()`

Access method for the bounds property.

Returns:

the current value of the bounds property

---

getExitLine2

`public ExitLine getExitLine2()`

Access method for the exitLine2 property.

Returns:

the current value of the exitLine2 property

---

setExitLine2

`public void setExitLine2(ExitLine aExitLine2)`

Sets the value of the exitLine2 property.

Parameters:

`aExitLine2` - the new value of the exitLine2 property

---

getExitLine1

`public ExitLine getExitLine1()`

Access method for the exitLine1 property.

Returns:

the current value of the exitLine1 property

---

setExitLine1

`public void setExitLine1(ExitLine aExitLine1)`

Sets the value of the exitLine1 property.

Parameters:

`aExitLine1` - the new value of the exitLine1 property

---

calculateVertices

`void calculateVertices()`

Calcul des points de sommet (haut et bas). D'abord on cherche la droite (la pente et l'ordonnée) perpendiculaire à la ligne de segment et qui la coupe au milieu. Ensuite on cherche deux points sur cette droite se trouvant à maxDistance loin du point de l'intersection. Ce sont les vertex1 et vertex2.

---

calculateExitLine

```
private void calculateExitLine(double passWidth)
```

Calcul et création de ligne de sortie ayant la longueur de passWidth défini dans Route.

Parameters:

`passWidth` - - La taille du passage entre deux segments. Défini la longueur de la ligne de sortie.

---

createBounds

```
private void createBounds()
```

Création des bornes du segment. Les vertices doivent être calculés d'abord. Cette méthode est appelée par calculateVertices().

---

dispatcher

Class SegmentBounds

```
java.lang.Object
  └ dispatcher.SegmentBounds
```

```
public class SegmentBounds
extends java.lang.Object
```

Field Summary

private double	**approachLimit** Si la limite d'approchement est dépassée, l'utilisateur doit être rédirigé.
private java.awt.geom.Line2D.Double[]	**coords**

Constructor Summary

SegmentBounds()
SegmentBounds(java.awt.geom.Line2D.Double[] coords, double approachLimit)
SegmentBounds(SegmentBounds segmentBounds)

Method Summary

double	**getApproachLimit**() Access method for the approachLimit property.
java.awt.geom.Line2D.Double[]	**getCoords**() Access method for the coords property.
void	**setApproachLimit**(double aApproachLimit) Sets the value of the approachLimit property.
void	**setCoords**(java.awt.geom.Line2D.Double[] aCoords) Sets the value of the coords property.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

coords

```
private java.awt.geom.Line2D.Double[] coords
```

---

approachLimit

```
private double approachLimit
```

Si la limite d'approchement est dépassée, l'utilisateur doit être rédirigé.

Constructor Detail

SegmentBounds

```
public SegmentBounds(SegmentBounds segmentBounds)
```

Parameters:

segmentBounds -

SegmentBounds

```
public SegmentBounds(java.awt.geom.Line2D.Double[] coords,
                     double approachLimit)
```

Parameters:

coords -

approachLimit -

SegmentBounds

```
public SegmentBounds()
```

Since:

2006

Method Detail

getCoords

```
public java.awt.geom.Line2D.Double[] getCoords()
```

Access method for the coords property.
Returns:
the current value of the coords property

---

setCoords

```
public void setCoords(java.awt.geom.Line2D.Double[] aCoords)
```

Sets the value of the coords property.
Parameters:
`aCoords` - the new value of the coords property

---

getApproachLimit

```
public double getApproachLimit()
```

Access method for the approachLimit property.
Returns:
the current value of the approachLimit property

---

setApproachLimit

```
public void setApproachLimit(double aApproachLimit)
```

Sets the value of the approachLimit property.
Parameters:
`aApproachLimit` - the new value of the approachLimit property

Package Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

---

dispatcher

Class ServerConnection

```
java.lang.Object
  └ dispatcher.ServerConnection
```

```
public class ServerConnection
extends java.lang.Object
```

ServerConnection utilise les classes créées par Apache Axis (http://ws.apache.org/axis/) pour se connecter à Novasys et consommer leur service web MessageService.asmx . Le résultat de cette connexion est un fichier XML contenant, parmi les autres données, la position courante de l'utilisateur sous la forme longitude/latitude.

Field Summary

private com.novasys.wstest01.MessageService.MessageServiceLocator	**messageServiceLocator**
private com.novasys.wstest01.MessageService.MessageServiceSoap	**messageServiceSoap**
private UTMMessage	**utmMessage**

Constructor Summary

ServerConnection() Les instances de MessageServiceLocator et MessageServiceSoap sont créées avec les données appropriées.

Method Summary

UTMMessage	**getData**() Effectue l'appel au service et récupère le résultat XML.
private UTMMessage	**transformToUTM**(double lat, double lon) Cette méthode utilise les librairies JScience pour la transformation de coordonnées de la forme longitude/latitude vers UTM.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

utmMessage

```
private UTMMessage utmMessage
```

---

messageServiceLocator

```
private com.novasys.wstest01.MessageService.MessageServiceLocator
messageServiceLocator
```

---

messageServiceSoap

```
private com.novasys.wstest01.MessageService.MessageServiceSoap
messageServiceSoap
```

Constructor Detail

ServerConnection

```
public ServerConnection()
```

Les instances de MessageServiceLocator et MessageServiceSoap sont créées avec les données appropriées. Le client est prêt pour consommer le service.

Method Detail

getData

```
public UTMMessage getData()
                  throws java.lang.Exception
```

Effectue l'appel au service et récupère le résultat XML. Le résultat et ensuite parsé pour la position de l'utilisateur en longitude/latitude. Si les données sont trouvées, elles sont transformées par transformToUTM() et retournées à FrMainWindow. Return value: UTMMessage contenant les coordonnées UTM de l'utilisateur ou null.

Returns:
UTMMessage
Throws:
`java.lang.Exception`

---

transformToUTM

```
private UTMMessage transformToUTM(double lat,
                                  double lon)
                           throws java.lang.Exception
```

Cette méthode utilise les librairies JScience pour la transformation de coordonnées de la forme longitude/latitude vers UTM. Après la transformation le nouveau UTMMessage est retourné. Return value: Nouveau UTMMessage contenant les coordonnées UTM de l'utilsiateur.

Parameters:

`lat` - - Latitude.

`lon` - - Longitude.

Returns:

UTMMessage

Throws:

`java.lang.Exception`

Package Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS **NEXT CLASS** **FRAMES** **NO FRAMES** **All Classes** **All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

---

dispatcher

Class UTMMessage

```
java.lang.Object
  └ dispatcher.UTMMessage
```

```
public class UTMMessage
extends java.lang.Object
```

Représentation UTM du paquet GPS reçu du serveur. Les coordonnées ont été transformées par la classe ServerConnection des coordonnées géographiques (latitude / longitude) vers la Transverse universelle de Mercator ou UTM.

Field Summary

private java.awt.geom.Point2D.Double	**coords** Coordonnées UTM reçues de l'appareil GPS.
private java.util.Date	**sendTime** Le temps où la position a été envoyée de l'appareil GPS vers le serveur Novasys.
private double	**speed** La vitesse de mouvement calculée sur le serveur Novasys.
private char	**zoneLetter** Lettre de la zone UTM
private int	**zoneNumber** Numéro de la zone UTM.

Constructor Summary

UTMMessage() Constructeur par défaut.
UTMMessage(java.awt.geom.Point2D.Double coords, int zoneNumber, char zoneLetter) Crée une nouvelle instance à partir des paramètres.

Method Summary

char	**getZoneLetter**() Access method for the zoneLetter property.
int	**getZoneNumber**() Access method for the zoneNumber property.
void	**setZoneLetter**(char aZoneLetter) Sets the value of the zoneLetter property.
void	**setZoneNumber**(int aZoneNumber) Sets the value of the zoneNumber property.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

coords

private java.awt.geom.Point2D.Double **coords**

Coordonnées UTM reçues de l'appareil GPS.

---

speed

private double **speed**

La vitesse de mouvement calculée sur le serveur Novasys. Pas utilisé dans cette version.

---

sendTime

private java.util.Date **sendTime**

Le temps où la position a été envoyée de l'appareil GPS vers le serveur Novasys. Pas utilisé dans cette version.

---

zoneNumber

private int **zoneNumber**

Numéro de la zone UTM.

---

zoneLetter

```
private char zoneLetter
```
Lettre de la zone UTM

Constructor Detail

UTMMessage

```
public UTMMessage(java.awt.geom.Point2D.Double coords,
                  int zoneNumber,
                  char zoneLetter)
```
Crée une nouvelle instance à partir des paramètres.

Parameters:

`coords` - - Les coordonnées UTM du point.

`zoneNumber` - - Le numéro de la zone UTM.

`zoneLetter` - - La lettre de la zone UTM.

UTMMessage

```
public UTMMessage()
```
Constructeur par défaut. Met les coordonnées à zéro et la zone à numéro inexistante.

Method Detail

getZoneNumber

```
public int getZoneNumber()
```
Access method for the zoneNumber property.

Returns:

the current value of the zoneNumber property

---

setZoneNumber

```
public void setZoneNumber(int aZoneNumber)
```
Sets the value of the zoneNumber property.

Parameters:

`aZoneNumber` - the new value of the zoneNumber property

---

getZoneLetter

```
public char getZoneLetter()
```
Access method for the zoneLetter property.

Returns:
the current value of the zoneLetter property

---

setZoneLetter

```
public void setZoneLetter(char aZoneLetter)
```

Sets the value of the zoneLetter property.

Parameters:

`aZoneLetter` - the new value of the zoneLetter property

Package Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS

FRAMES **NO FRAMES** **All Classes** **All Classes**

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD
