Librairie de modélisation 3D en VRML pour pocketPC

Nicolas Schoeni

Informatique 8e semestre

Assistant : T. Fong
Professeur : R. Clavel

Juillet 2002

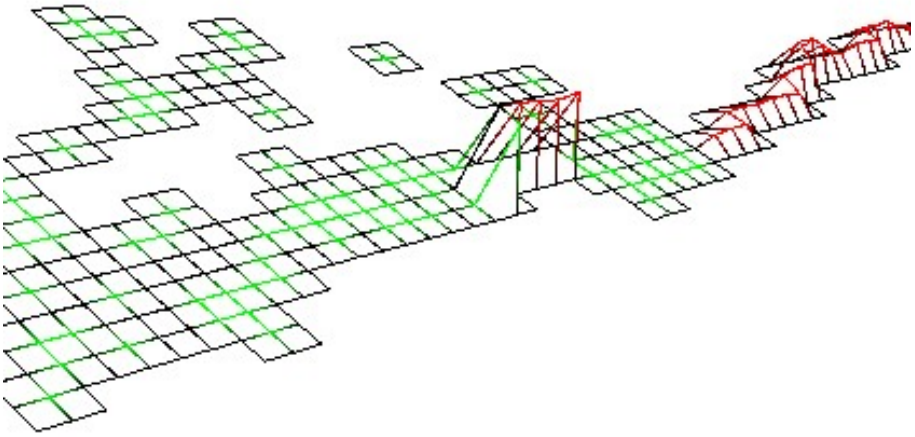

Table des matières

Introduction

0.1 Le projet PerceptOR

Ce projet de semestre s'inscrit dans le cadre d'un projet du groupe *VRAI* intitulé *PerceptOR Software Systems (Perception for Off Road Robots)* [6].

PerceptOR est un projet de recherche du *Tactical Technology Office* du *DARPA*[1] où L'EPFL collabore avec le *CMU*[2], *SAIC*[3], *NASA Jet Propulsion Laboratory*, *Applied Perception Inc.* et *Visteon Corporation*.

Le but est de développer des modules prototypes de perception pour robots mobiles *outdoor*.

FIG. 1 – AATVIL (SAIC)

FIG. 2 – Navlab 11 (CMU)

Plusieurs capteurs doivent pouvoir être pris en compte (capteur de types divers, embarqué ou global). Les capteurs permettent de percevoir l'environnement dans des conditions réelles (urbaines ou «off-road»).

L'enjeu est la détection et l'évitement d'obstacles à bord pour que les robots soient autonomes, mais aussi l'interaction humain-robot (téléopération des véhicules, transmission de l'environnement perçu).

Le but principal de mon projet de semestre est ce dernier point. Développer un système d'interface utilisateur graphique sur périphérique mobile (*PDA*) afin d'avoir connaissance de ce que perçoit le robot pour analyser son environnement,

[1] Defence Advanced Research Projets Administration
[2] Carnegie Mellon University
[3] Science Applications International Corporation

pour surveiller les algorithmes de détection automatique, déterminer ou contrôler son chemin parcouru ou alors pour lui donner des ordres (le chemin à suivre par exemple).

0.2 Le choix du PocketPC

Le choix du dispositif utilisateur s'est porté sur un *iPAQ* de *Compaq*. Cela pour plusieurs raisons :

- La puissance de ces appareils devient vraiment intéressante (processeur *StrongARM* cadencé à 206 MHz, 32 MB de mémoire).
- L'écran *LCD* couleur (64'000 couleurs), tactile, de grande taille et avec une très bonne résolution (240x320 pixels).
- La remarquable luminosité de l'écran *(reflective TFT)* permet l'utilisation même à l'extérieur en plein soleil (ce qui n'est pas le cas d'un laptop standard).
- Le poids et l'encombrement minime de ces appareils.
- La facilité de manipulation (un stylet remplace à la fois la souris et le clavier).
- La possibilité d'ajouter une carte réseau sans fil (802.11b[4]), un *GPS* ou n'importe quel périphérique *PCMCIA*.
- Une interface pour piloter des robots à distance déjà existante [7].
- L'intérêt de tester les capacités de ces *PocketPC* pour afficher des données 3D complexes en temps réel.

L'un des seuls points négatifs est la faible autonomie (2 à 3 heures seulement).

0.3 Le choix du VRML

Le challenge est de pouvoir afficher des données tridimensionnelles sur un ordinateur de poche. Sur les stations de travail on peut simplement se contenter d'utiliser les capacités de calcul monstrueuses des cartes graphiques ou des processeurs *FPU*. Mais sur les dispositifs mobiles, les circuits graphiques ou de calcul embarqués se doivent d'être plus modestes (consommation électrique, refroidissement, taille, ...).

Cependant l'imagerie 3D est en plein essor sur ces machines. Cela devient même une nécessité pour rendre l'interactivité entre l'utilisateur et les données plus aisée.

On peut par exemple imaginer de nombreuses applications professionnelles utilisant de la 3D (*CAO*, géonavigation, médecine, e-learning, ...). Et lorsque l'on envisage le transfert de données graphiques entre des dispositifs portables et des

[4]Récemment rebaptisée *WiFi*, c'est une norme de transfert de données sans fil. Avec un débit théorique de 11 Mbps et une portée de 30 m à quelques centaines de mètres, *WiFi* exploite la fréquence radio de 2,4 GHz. Cette norme pourrait s'imposer à l'avenir car bien plus puissante que *Bluetooth* (débit maximum 1 Mbps).

serveurs[5], une solution performante est d'équiper le terminal portable lui-même des qualités nécessaires au traitement de l'information 3D. Ainsi les serveurs de données n'ont pas besoin de capacités 3D spéciales et les réseaux de communication sont soulagés. L'utilisateur doit effectivement télécharger une grande masse d'informations sur son *PDA* avant de devenir nomade, ce qui posera d'ailleurs de moins de moins de problèmes étant donné la capacité mémoire grandissante des *PDA*.

Les formats de données 3D actuellement supportés sur *PDA* sont peu nombreux. Le langage *VRML*[6] en est un. Développé initialement par *Silicon Graphics* et déjà bien connu sur les stations fixes, il permet de modéliser une scène tridimensionnelle à la manière d'un langage de programmation. Sur PDA, il est depuis peu supporté par *Pocket Cortona* de *Parallel Graphics*[10], première société à avoir sorti un lecteur de *VRML* pour *PDA*.

A l'avenir le format *X3D*, tentative de combiner les fonctionnalités de *VRML* avec la puissance du langage *XML*[7], pourrait s'imposer. On peut aussi citer la société *Ekkla Research*[8] qui développe actuellement des solutions 3D pour *PDA* basées sur *OpenGL* ou *Direct3D*. Ou encore *Superscape*[9] qui travaille sur son produit *Swerve I3d* (Basé sur le langage Java pour *iPAQ*).

Actuellement, le choix du *VRML* semble judicieux. Ce langage repose déjà sur une certaine notoriété et possède une documentation largement fournie[12] [13] [14] [15]. Enfin la solution commerciale proposée par *ParallelGraphics* semble suffisamment aboutie, proposant d'une part une application *standalone* permettant d'afficher les models 3D, d'autre part la possibilité d'utiliser les librairies du moteur 3D en *C++* grâce au kit de développement fourni (SDK).

0.4 Buts du travail

Le but du projet est donc de développer des outils de programmation pour traiter des données 3D (cartes, images stéréoscopiques, ...) et de les visualiser sur un *PocketPC*. Il faut aussi avoir des méthodes pour sélectionner, simplifier et convertir des données 3D en modèles *VRML*.

Deux composants logiciels doivent être créés :

Le *VRMLserver*

- Traiter les données 3D, créer les cartes sous forme de fichiers *VRML*.
- Permettre la sélection de régions d'intérêt.
- Envoyer les fichiers *VRML* au *PDA*.

[5]La norme *4G* par exemple doit permettre à l'avenir de délivrer des films en haute résolution avec des débits de 20 Mbps, et pourrait aussi délivrer du streaming 3D selon les projets de certains constructeurs.

[6]Virtual Reality Modeling Language

[7]Extensible Markup Language

– S'intégrer avec les modules de perception de *CMU* (le scanner *Sick* et l'algorithme *«line-stripe classifier»*).
– Programmé en C sous *Linux* (comme l'algorithme *«line-stripe classifier»*).

L'interface utilisateur sur le *PDA*

– Transférer des données et des fichiers depuis le serveur.
– Sélectionner une région d'intérêt (interaction avec les modules de perception et le *VRMLserver*).
– Lancer des applications externes (moteur *VRML*) ou accéder directement aux routines du moteur *VRML*.
– Programmé en C++ sous *Windows CE*.

0.5 Design de l'architecture logicielle

D'un côté se trouve le serveur Linux embarqué sur le robot mobile. Il reçoit des coordonnées de points, soit par des capteurs embarqués, soit depuis des fichiers pré-enregistrés. Après un traitement de classification des points *(Line-stripe classifier)*, il en génère une carte tridimensionnelle *VRML* qui peut être envoyée à l'application cliente.

Le client, sur le *PocketPC*, se connecte sur le serveur. L'utilisateur peut, au moyen d'une interface graphique (*GUI*), sélectionner les cartes ou régions d'intérêt, qui seront téléchargées depuis le serveur. Le moteur 3D de *Pocket Cortona* permet d'afficher les cartes 3D.

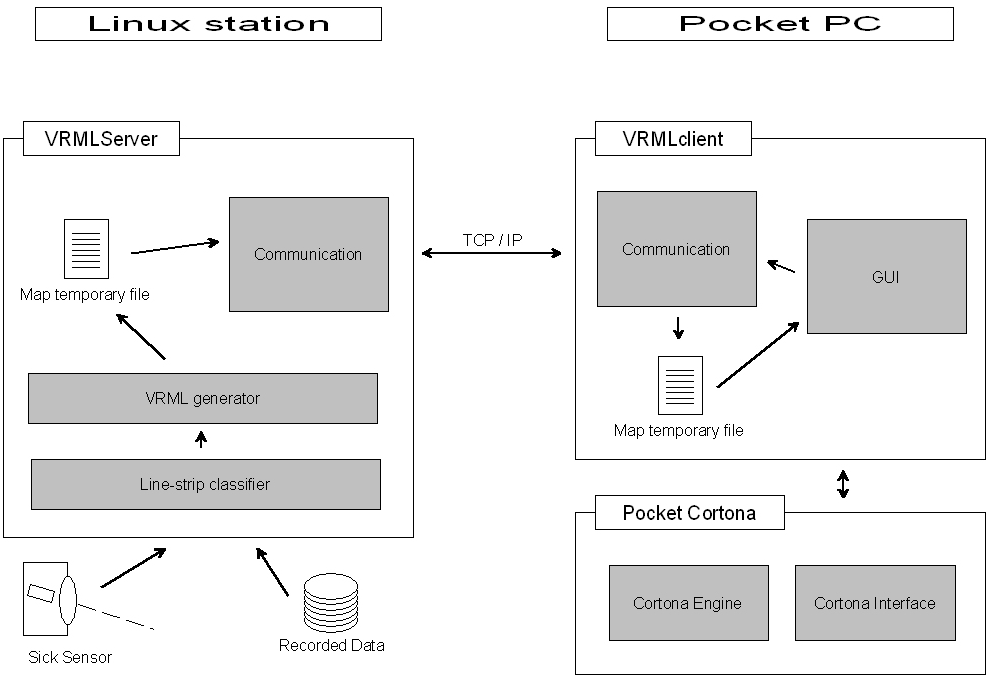

FIG. 3 – L'architecture logicielle

Chapitre 1

Création de cartes 3D

1.1 Génération de code VRML

Il faut générer des fichiers *VRML* qui sont des fichiers textes. Dans un fichier *VRML*, la scène est décrite au moyen de nœuds qui peuvent ou non posséder des propriétés. Les nœuds peuvent aussi être imbriqués.

Pour faire une carte (*mesh*) représentant un terrain, le plus simple est d'utiliser un nœud *IndexedFaceSet*. Cela crée une surface composée de sommets avec des altitudes variables. On donne une liste de sommets (par leurs 3 coordonnées), une liste codant la connexion des points entre eux pour former des polygones et éventuellement une liste des couleurs attribuées à chaque point.

IndexedFaceSet produit une surface solide ; les polygones sont colorés en utilisant des dégradés depuis la couleur des points. Il est aussi possible d'avoir une surface «en fil de fer». Dans ce cas il faut utiliser un nœud *IndexedLineSet*

```
DEF mySurface IndexedFaceSet {
 coord Coordinate { point [0 0 0, 0 1 0, 0 2 0, 1 0 0, 1 1 .5,
                           1 2 0, 2 0 0, 2 1 0, 2 2 0] }
 coordIndex [0 1 3 -1 1 4 3 -1 1 2 4 -1 2 5 4 -1 3 4 6 -1
             4 7 6 -1 4 5 7 -1 5 8 7 -1]
 color Color { color [0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 1 0,
                      0 0 1, 0 0 1, 0 0 1, 0 0 1] }
}
```

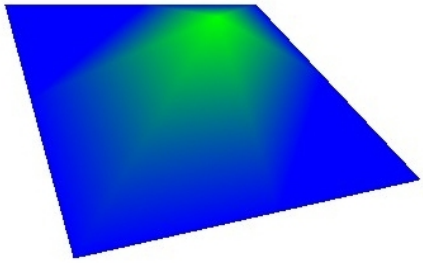

FIG. 1.1 – IndexedFaceSet

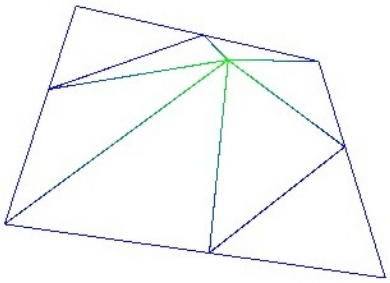

FIG. 1.2 – IndexedLineSet

Au début du fichier on peut aussi spécifier la couleur de fond (`Background`), la position de la caméra (`Viewpoint`) et le mode de navigation (`NavigationInfo`).

```
Background { skyColor [1 1 1] }
Viewpoint { position 0 0 20 }
NavigationInfo { type "EXAMINE" }
```

Un nœud `Transform` permet quant à lui d'effectuer une translation ou une mise à l'échelle, grâce à ses deux propriétés `translation` et `size`.

D'autres nœuds sont bien sur disponibles mais ne sont pas vraiment utiles pour le projet.

1.2 Scripts

Il est possible d'inclure des scripts dans un fichier *VRML*. Ceux-ci doivent être écris en *JavaScript*. Les browsers *VRML* ne sont pas forcément tous en mesure d'exécuter ces scripts, mais *Pocket Cortona* dispose heureusement de cette fonctionnalité fort intéressante.

Il devient alors possible d'associer des fonctions écrites en Java à des événements *VRML*. Ces événements sont par exemple le fait de cliquer avec le stylet sur un objet de la scène (nœud `TouchSensor`). La fonction JavaScript est en mesure de savoir à quelle cordonnée tridimensionnelle correspond le point cliqué.

L'effet de la fonction JavaScript peut être de spécifier le paramètre d'un nœud `Switch`. Ce nœud permet de définir pour un même objet plusieurs versions différentes, un paramètre permettant de choisir laquelle est affichée.

1.3 Simplification

Les cartes générées sont des ensembles de points d'altitudes propres. On peut représenter la carte en mode «fil de fer» ou en mode «solid». En mode «fil de fer», les points sont connectés pour former des quadrilatères ; seules les arrêtes sont alors

affichées. Les points étant alignés, cela forme donc des lignes de coordonnées orthogonales. En mode «solid», on divise arbitrairement les quadrilatères en 2 triangles pour avoir des polygones planaires. Ils sont ensuite remplis en dégradant les couleurs de chacun des 3 sommets.

On remarque tout de suite qu'une grande partie de ces polygones ne sont pas utiles. Ce sont les polygones qui ont tous leurs sommets à la même altitude et de la même couleur.

La première simplification possible consiste donc à les retirer de la carte. Il suffit donc d'enlever tous les points qui ont les mêmes caractéristiques que chacun de leurs 8 voisins.

Cette simplification, très simple mais diaboliquement efficace, suffit déjà à rendre la manipulation des cartes fluides sur le *PocketPC*. Typiquement, elle réduit le nombre de polygones environ d'un facteur 10.

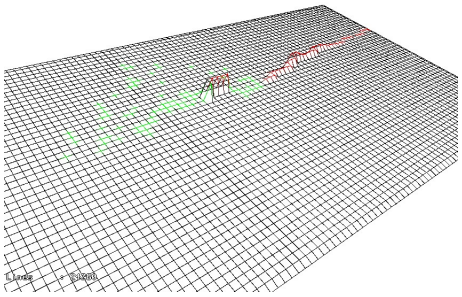

FIG. 1.3 – Carte originale (avec tous les points)

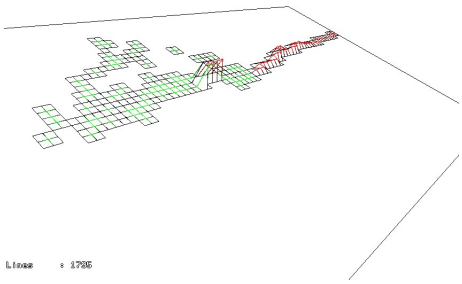

FIG. 1.4 – Carte simplifiée (seulement les points «utiles»)

Il est par ailleurs important de noter que cette «compression» ne perd aucune information. Seules des données redondantes (c'est à dire qui n'influencent pas le résultat graphique) sont enlevées.

Il serait possible d'implémenter des méthodes de simplification plus sophistiquées [3] ou avec perte d'information (approximer ou fusionner certains polygones par exemple) mais pour les cartes que l'on dispose cela ne semblait pas vraiment nécessaire ; je me suis donc arrêté à cette première approche.

1.4 Levels of detail

Un nœud *VRML* (`LOD` pour *Level Of Detail*) permet de déclarer des résolutions différentes pour un même objet, selon sa distance de la caméra virtuelle. Pour ce faire, on définit l'objet dans chaque résolution désirée. Le moteur 3D choisira de manière dynamique quelle version de l'objet afficher.

Il serait tout à fait possible d'appliquer cette technique en spécifiant les cartes avec plusieurs résolutions. Mais comme il n'y a qu'une seule carte affichée à la fois (un

seul objet), cela ne me semblait pas vraiment intéressant. Cela augmente par ailleurs la taille des fichiers *VRML*, étant donné que les cartes doivent être définies plusieurs fois. J'ai préféré développer des techniques de zoom, ce qui me paraît plus utile car l'interface de *Pocket Cortona* n'est pas très pratique pour déplacer la caméra plus près ou plus loin de l'objet.

Il pourrait aussi être intéressant de modifier la résolution de cartes. Mais cela doit plutôt être fait en relation avec l'algorithme de classification, aussi responsable de la lecture des données binaires. Mais pour les données pré-enregistrées, mes quelques essais ne se sont pas avérés très fructueux. La rapidité n'est pas meilleure en diminuant la résolution, ni la précision plus fine lorsque l'on augmente la résolution.

1.5 Zoom

Il est utile de pouvoir sélectionner des régions d'intérêt sur la carte. La meilleure méthode pour cela me semble être la possibilité de zoomer sur une partie de la carte. L'idée de départ était de fabriquer des fichiers *VRML* «autonomes». C'est à dire que toute l'information pour les différentes vues possibles est contenue dans le même fichier

D'autres idées peuvent cependant être envisagées.

Par exemple j'ai pensé à faire un fichier principal qui contient uniquement les mécanismes permettant de choisir une zone à zoomer. Lorsque l'utilisateur décide de visualiser une région spécifique, un fichier contenant cette carte est alors téléchargé depuis le serveur (sur le serveur, les fichiers sont soit créés en temps réel soit pré calculés). J'ai du abandonner cette idée pour des raisons de cache internet. Il est possible dans un fichier *VRML* d'importer d'autres fichiers dont le nom est une adresse internet. Le problème est que le fichier n'est réellement téléchargé qu'une seule fois, ensuite il est simplement pris de la copie locale dans le cache *Windows*, même si l'original a été modifié sur le serveur. De plus le temps de téléchargement semble assez rapide. Il est finalement préférable de tout télécharger en une seule fois.

Une autre solution serait de transférer des fichiers binaires contenant uniquement les coordonnées des points. La carte ou la région de carte désirée serait alors construite par l'application cliente. J'ai aussi écarté cette solution car il semble raisonnable d'effectuer le plus de calculs possibles et la conversion de binaire en texte là où se trouve la puissance de calcul la plus importante, c'est à dire sur le serveur.

La première méthode implémentée a finalement été de diviser la carte en 9 régions de même surface. Chaque région est elle-même une carte autonome et peut être affichée ou non (au moyen d'un nœud `Switch`). Au début toutes les régions sont affichées. Lorsque l'on clique sur la carte, une fonction *JavaScript* analyse le point d'impact, trouve la région à laquelle il appartient et masque toutes les autres régions. La région choisie est finalement agrandie au moyen d'un nœud `Transform`. Ensuite, lors d'un nouveau clic n'importe où sur la région zoomée, on revient à la carte complète initiale.

Il faut préciser que l'on enlève ni ne rajoute aucun point ou précision lorsque l'on zoome. Les points ne sont pas très nombreux et cette méthode permet de mieux analyser ou se rendre compte des altitudes d'une région.

J'ai ensuite amélioré la méthode en divisant la carte en 81 (9x9) cases. Lorsque l'on clique sur une case on affiche une région comprenant cette case et ses 8 voisins.

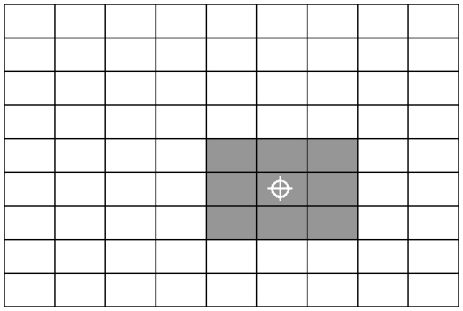

FIG. 1.5 – Zoom précalculé 9x9 cases

Cela permet d'avoir une précision raisonnable. Le zoom est très rapide étant donné que la carte 3D correspondant à chacune des cases est précalculée. Les cases sont simplement affichées côtes à côtes pour former les régions.

Par ailleurs, la granularité (le nombre de cases distinctes) est réglable, ainsi que le niveau maximum de zoom possible. Pour les tests je n'ai cependant implémenté qu'un seul niveau de zoom.

J'ai aussi implémenté une autre méthode de zoom, où une nouvelle carte est entièrement calculée en temps réel en JavaScript. Le script détermine les points devant être pris en compte pour la nouvelle carte et génère un nouvel objet. Cela est possible mais la lenteur de calcul en JavaScript (langage interprété) rends cette solution inutilisable (plus de 20 secondes pour afficher un zoom).

1.6 Autres informations sur la carte

Le chemin parcouru par le robot peut être affiché sur la carte sous forme de segments de droites. De même, dans une autre couleur, le chemin prévu ou qu'il lui reste à parcourir pour arriver à un certain but par exemple.

Il serait possible d'afficher aussi d'autres informations additionnelles sur les cartes :

- Des buts, des cibles ou positions à atteindre.
- Des zones, régions à éviter ou au contraire à explorer.
- Des référentiels globaux ou relatifs au robot.

Chapitre 2

Protocoles de communication

2.1 Etablissement d'un socket

Le serveur et l'application cliente communiquent au moyen d'une connexion *TCP/IP* ouverte grâce aux fonctions standards d'établissement de socket en C.

Du côté du serveur, un port (pouvant être choisi à la compilation) est ouvert, en attente d'un éventuel client.

Du côté du client, lors de l'établissement de la communication, il est nécessaire de spécifier l'adresse et le port d'entrée du serveur. Le port de sortie du client est réglé à la compilation. C'est le port 5000 par défaut.

La connexion n'est pas sécurisée. Cela implique qu'il n'est pas possible de se connecter par Internet si l'un des protagonistes est dans le réseau de l'*EPFL* et l'autre non. Mais si le client et le serveur sont tous deux dans le réseau sécurisé de l'*EPFL*, ou au contraire dans un réseau non sécurisé, cela ne pose aucun problème.

L'adresse du serveur peut être spécifiée soit par l'adresse *ip*, soit par *hostname*, dans ce cas l'adresse est résolue par recherche *DNS*.

2.2 Transfert de données

Les données transitent sous forme de packets. La taille maximum d'un packet pour les API des compilateurs Linux et Windows CE ne doit pas dépasser 256 octets. Le mécanisme de communication n'est pas très rapide, je pense qu'il devrait être possible de l'optimiser afin de profiter au mieux de la vitesse maximale d'une connexion *Wireless Lan* (11 MBit/s théoriquement mais en pratique avec l'iPAQ, plutôt 2 Mbit/s mesurés).

De plus les communications se font en mode synchrone. Pour que le programme ne soit pas bloqué pendant les transferts de données, les communications peuvent se dérouler dans un *thread* séparé.

Dans la version actuelle, le serveur accepte un seul client à la fois. Il est cependant aisé de modifier son comportement afin de pouvoir servir plusieurs clients simultanément.

Le serveur est toujours en attente et c'est le client qui donne des commandes

Le client doit d'abord se connecter au serveur. Il lui est possible ensuite de demander la liste des cartes disponibles, de demander une prévisualisation d'une carte ou d'en télécharger une. Le fonctionnalité que le serveur avertisse le client lorsqu'une carte a été modifiée n'est pas implémenté pour le moment (Les cartes ne peuvent pas bouger...), mais le client peut à tout moment mettre à jour sa copie locale en envoyant une commande de téléchargement. Finalement le client se déconnecte lorsqu'il désire libérer le serveur.

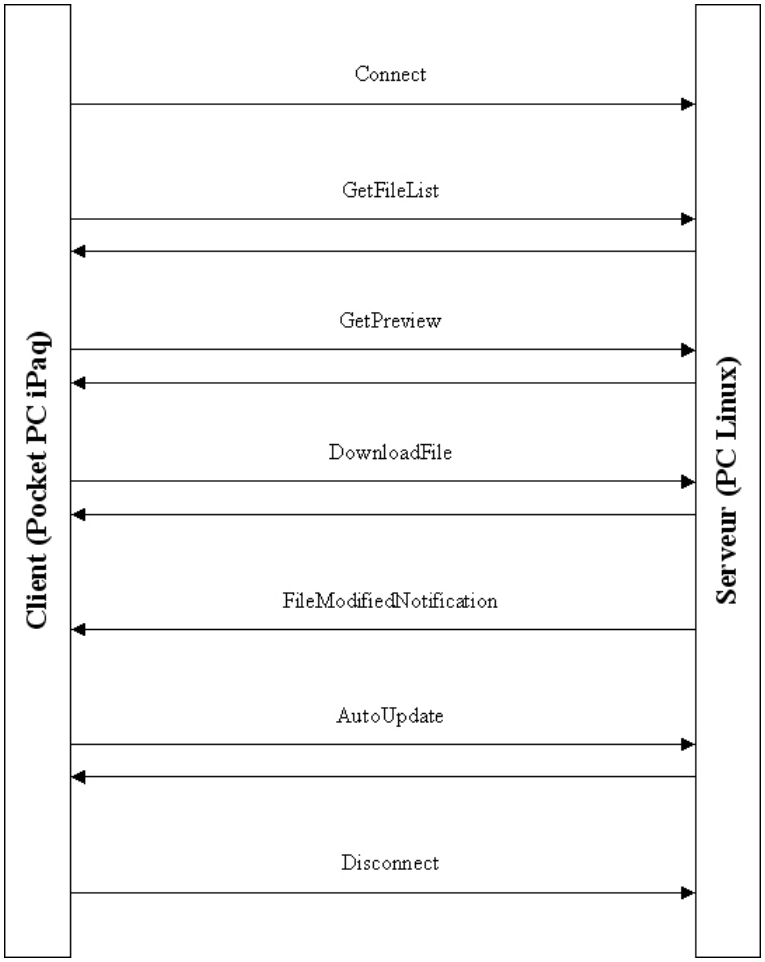

FIG. 2.1 – La communication entre le serveur et le client

2.3 Le Protocole de commande

Le protocole de commande est développé selon la méthode KISS[1]. Le client envoie un packet contenant la commande au serveur, le serveur répond par un ou plusieurs packets de données. Lorsque plusieurs packets successifs doivent être envoyés dans la même direction, le destinataire acquitte chaque packet par un packet spécial (afin de séparer les packets).

Un packet de commande consiste en :

- 1 byte de commande
- 1 byte de paramètre éventuel
- Un nom de fichier éventuel *(ASCII Null Terminated)*

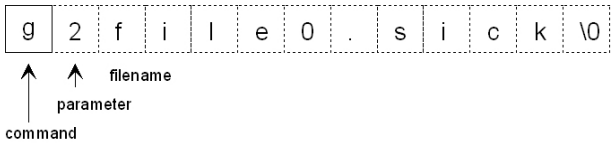

FIG. 2.2 – Les packets de commande

La commande *List available files*

Demande le transfert de la liste des fichiers disponibles sur le serveur. Ce dernier répond en envoyant un packet pour chaque fichier existant, avec le nom, la taille, la date et l'heure de création.

Byte de commande	:	`l`
Paramètre	:	aucun
Nom de fichier	:	inutile

La commande *Get file*

Demande le transfert d'un fichier spécifique. Le serveur répond par les données textes du fichier, splittées en packets de 250 octets. Le mode est un caractère numérique (0 : sans zoom, 1 : mode fil de fer, 2 : zoom statique 3x3, 3 : zoom statique 9x9, 4 : zoom dynamique).

Byte de commande	:	`g`
Paramètre	:	mode
Nom de fichier	:	présent

[1]Keep It Simple & Stupid

La commande *Preview*

Demande le transfert de la prévisualisation d'un fichier. La *preview* est la réduction de la carte à un bitmap noir/blanc de 60x30 pixels. Les points particuliers sur la carte (altitude différente de zéro ou coloré) sont en noirs, les points au sol sont en blanc. Les données constituant la prévisualisation constituent un unique packet de 250 octets.

Byte de commande	:	`p`
Paramètre	:	aucun
Nom de fichier	:	présent

Chapitre 3

Le serveur

L'application serveur se greffe sur l'algorithme de classification des points[2] déjà existant. Tous deux sont écrits en langage C. Il est possible de compiler le serveur à la fois sous Windows avec l'environnement *Borland C++* et à la fois sous Linux avec le compilateur *gcc*.

Le serveur est un programme de type console sans interface graphique. Il peut prendre ou non un argument. Si celui-ci est présent, il indique dans quel répertoire se trouvent les fichiers de données du capteur *Sick*. Sinon les fichiers sont supposés se trouver dans le répertoire courant.

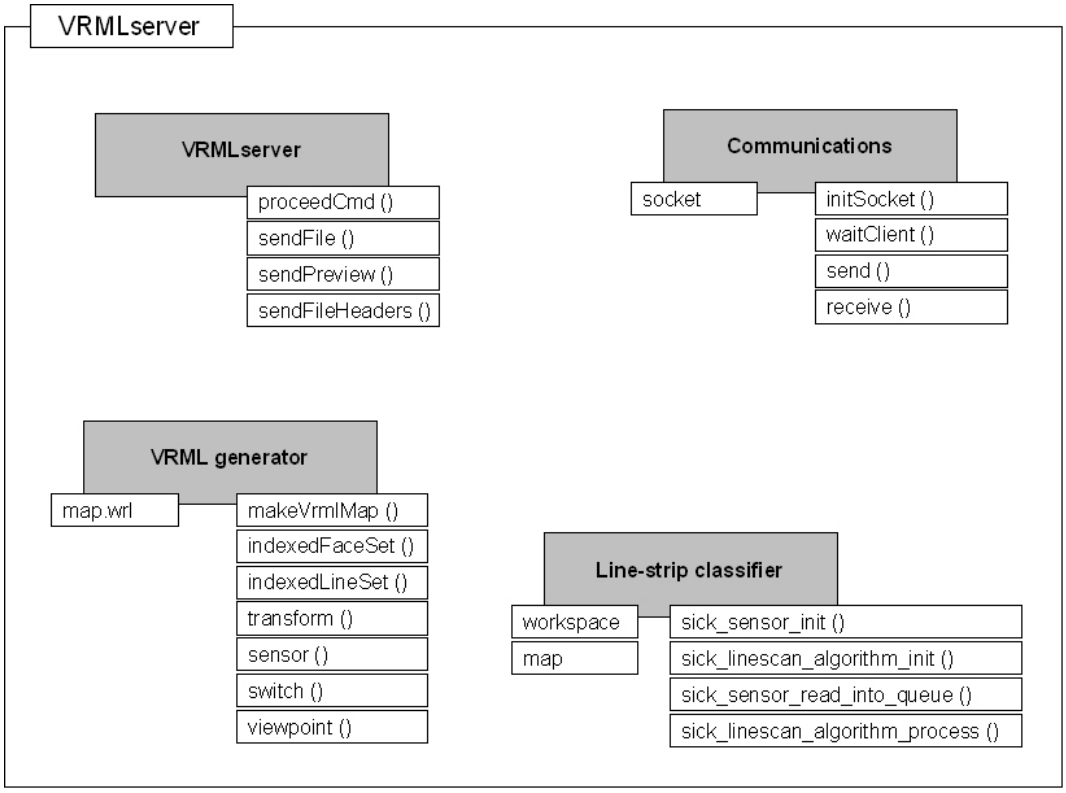

FIG. 3.1 – L'architecture du serveur

Le *Linestrip classifier algorithme* permet, après initialisation et lecture des données, de générer une liste de points tridimensionnels *(workspace)* et une carte des points avec leurs altitudes.

Le module de communication permet d'initialiser une connexion *TCP/IP* puis de prendre en charge un client distant. Il est responsable d'envoyer ou de recevoir des packets de données.

Le générateur *VRML* construit un fichier contenant la scène. Il dispose de fonctions pour créer chaque noeud *VRML* utilisé. Il est aussi capable d'ajouter des fonctions *JavaScript*.

Finalement la tâche récurrente du serveur consiste en une boucle qui, lorsqu'un client est connecté, attend une commande de celui-ci.

3.1 Les données disponibles

Le capteur

On considère pour le projet un seul type de capteur. C'est un scanner à balayage laser de type *Sick LMS-220*. Il mesure le temps de vol de la lumière pour atteindre l'obstacle. Ses caractéristiques techniques sont les suivantes :

Angle de balayage	:	180 deg (demi-plan)
Résolution angulaire	:	0.5 deg (361 mesures)
Portée maximum	:	50 m

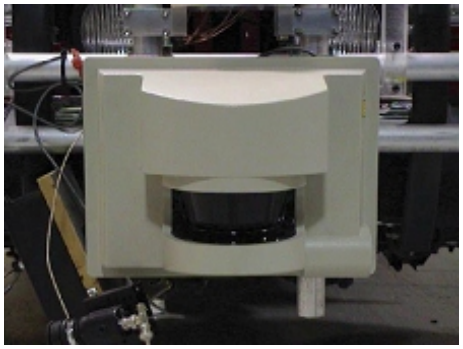

FIG. 3.2 – Sick LMS-220

L'EPFL ne disposant pas de ce capteur, la réalisation de ce projet s'est fait en utilisant des données qui sont des *scans* pré-enregistrées sous forme de fichiers binaires sur le disque dur de la station serveur.

L'algorithme de classification des points

Les points renvoyés par le scanner laser ne sont pas utilisés tels quels. Un algorithme permet de les classifier. Cet algorithme, *Line-stripe classifier* [2], transforme les données du capteur en points cartésiens.

L'algorithme calcule les coordonnées x et y dans le plan (au sol), puis la coordonnée z (distance du point au sol). Il cherche ensuite les surfaces «solides», afin de classifier les points et de déterminer leur éventuelle appartenance à un obstacle. Une couleur est affectée a chaque point suivant le résultat de cette classification.

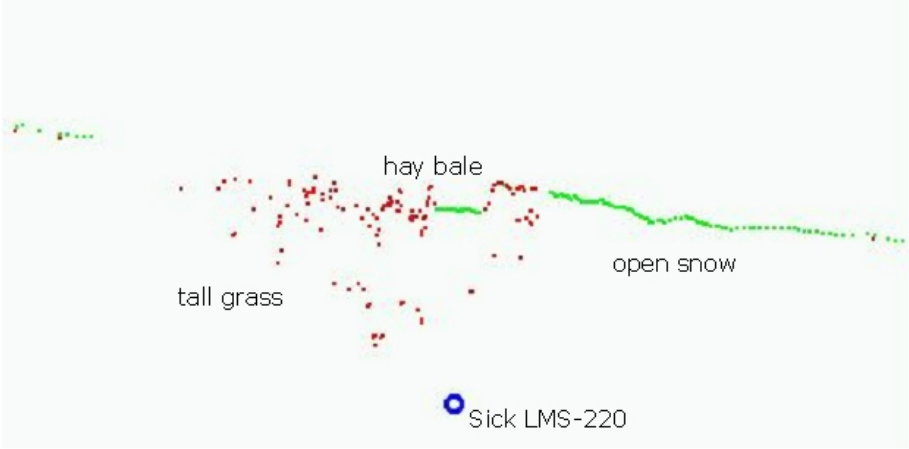

FIG. 3.3 – Line-stripe classifier

On peut intuitivement comprendre que les points qui sont bien alignés seront considérés comme faisant partie d'un même objet (obstacle). Par contre les régions ayant beaucoup de points disséminés avec des altitudes différentes seront plutôt considérés comme des régions libres d'obstacles (hautes herbes par exemple dans le cas où le robot évolue dans un champ).

Chapitre 4

L'application client

C'est une application écrite en *C++* avec l'environnement *eMbedded Visual C++* de *Microsoft*. Elle est compilée pour *Windows CE* sur *iPAQ*.

L'interface graphique se compose d'une fenêtre principale comportant une barre de menu. L'espace de la fenêtre principale est utilisé pour afficher la fenêtre *Cortona* qui permet d'afficher les données 3D.

Deux boîtes de dialogue permettent de se connecter au serveur, puis de télécharger des cartes depuis celui-ci.

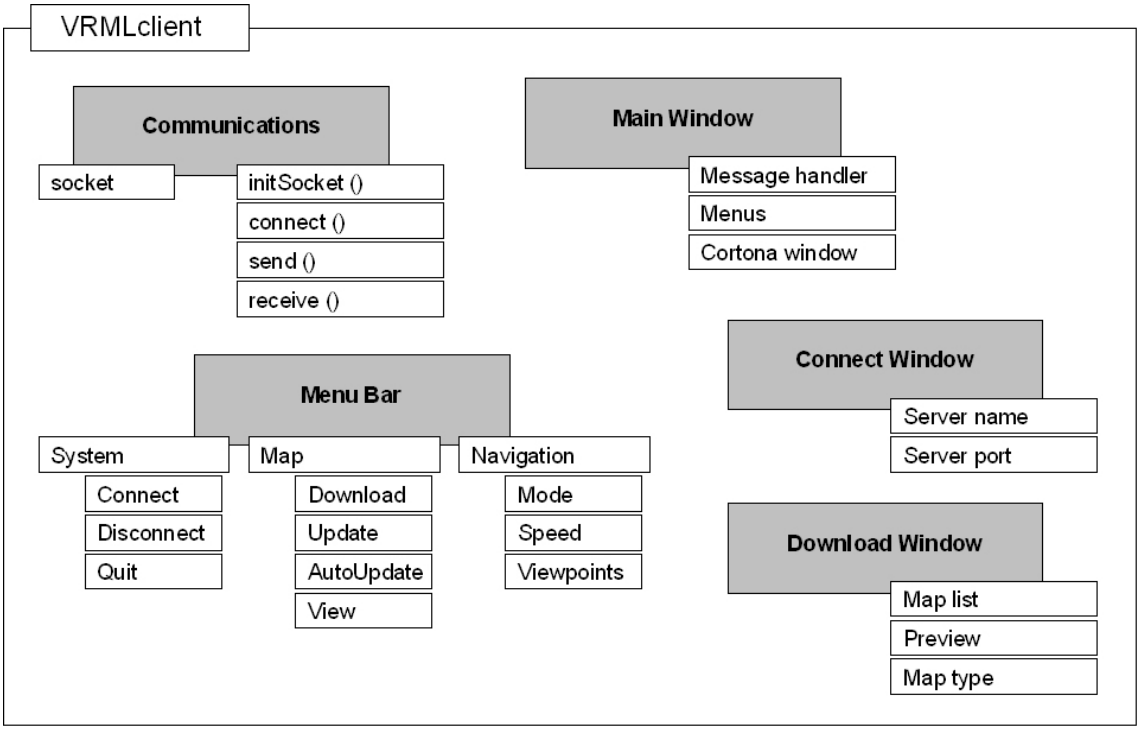

FIG. 4.1 – Design de l'interface utilisateur

4.1 Les menus

Le menu *System*

Permet de se connecter au serveur, de se déconnecter ou de quitter l'application.

Le menu *Map*

Lorsque la connexion au serveur est active, permet de télécharger des fichiers depuis le serveur (*Download*). Une fois qu'un fichier a été téléchargé une première fois, il est possible de demander une mise à jour du fichier depuis le serveur (*Update*). Il est aussi possible de demander une mise à jour automatique (*AutoUpdate*) à une certaine fréquence (par défaut toutes les 5 secondes).

Lorsqu'un fichier est téléchargé, il est placé dans le répertoire «Mes Documents» du *PocketPC*, sous le nom `map.wrl`.

La commande *View* crée la fenêtre Cortona dans la fenêtre principale de l'application. Le fichier `map.wrl` est alors chargé par le moteur 3D. Lorsque la fenêtre Cortona est créée, cette dernière commande est remplacée par *Refresh*. Cela permet simplement de recharger la scène (sans la télécharger à nouveau).

Le menu *Navigation*

Ce menu n'apparaît que lorsque la fenêtre Cortona est créée. Il permet de modifier le mode de navigation dans la scène 3D. Les modes de navigation possibles sont :

- *Walk* pour déplacer la caméra avec gravité.
- *Fly* pour déplacer la caméra sans gravité.
- *Examine* pour tourner la caméra autour de la scène.
- *None* désactive la navigation de la caméra.

Il est aussi possible de modifier la sensibilité de la caméra virtuelle par rapport à la vitesse du stylet sur l'écran (*Speed*).

Finalement certaines actions sont possibles sur la caméra virtuelle *(Viewpoint)* :

- *Fit* éloigne la caméra de la scène jusqu'à ce que tous les objets soient entièrement visibles.
- *Restore* revient au point de vue original spécifié dans le fichier *VRML*.
- *Goto* s'approche d'un point de la scène donné par l'utilisateur au moyen du stylet.

4.2 Utilisation de Cortona pour afficher les cartes

Pour afficher les cartes sur le *PocketPC*, on utilise donc le logiciel *Pocket Cortona*. La première approche consistait à lancer l'application *standalone* de Cortona avec le nom du fichier *VRML* en paramètre. Le problème de cette technique est que lorsque

Cortona à été exécuté une première fois, il n'affiche ensuite plus directement la scène 3D, mais seulement la fenêtre d'exploration des fichiers. Le temps de chargement est assez important et il n'est pas très pratique de passer sans cesse entre Cortona et l'application cliente.

Pour cela il nous a paru préférable d'utiliser le kit de développement (SDK) de Cortona afin d'inclure directement les fonctionnalités du moteur graphique dans l'application cliente. Il devient alors possible d'utiliser les contrôles, les objets et les méthodes de Cortona pour travailler en *VRML*.

4.3 Le SDK de Cortona

Les fichiers proposés par Cortona sont les suivants :

`CortonaControl.h`	`CortonaControl.c`
`CortonaDispatch.h`	`CortonaDispatch_i.c`
`Shelley.h`	`Shelley_i.c`

Pour les générer, il faut utiliser le compilateur `midl` de Microsoft sur les fichiers `.idl` fournis par Cortona. La syntaxe de la commande est par exemple la suivante :

```
midl /Oicf /h "Shelley.h" /iid "Shelley_i.c" Shelley.idl
```

La documentation de Cortona n'est pas très complète, en tout cas en ce qui concerne la prise en main. Mais le support technique de Cortona[11] dispose de techniciens très compétents, patients et sympathiques.

Pour utiliser les fonctionnalités de Cortona il faut donc inclure ces fichiers dans le projet *C++*, puis utiliser les 5 objets ou interfaces mis à disposition.

Cortona Control Object

C'est l'objet maître. Il permet de créer tous les autres. Il sert aussi à créer la fenêtre graphique Cortona où seront affichées les scènes 3D.

Il permet en plus de créer une nouvelle scène ou de charger une scène à l'écran à partir d'un fichier (méthode `SetScene`).

Cortona Engine

C'est un pointeur sur le moteur de Cortona. Il sert entre autre à créer ou à éditer des nœuds *VRML*.

Cortona Control Interface

C'est un pointeur sur l'interface Cortona, Les méthodes de cette dernière permettent de choisir le mode de navigation *(walk, fly, examine)*, la vitesse de manipulation ou d'exécuter des actions relatives au point de vue de la caméra (*viewpoint*).

I3DViewService3 interface

Le pointeur sur cette interface permet à l'application de contrôler à un level bas niveau la position de la caméra, d'accéder au *pixel buffer* de l'image 3D ou de contrôler le processus de rendu 3D.

Cortona Events Handler

Cet objet est une instance de la classe `CCortonaEvents`. Les méthodes de cette classe permettent d'intercepter des événements relatifs au moteur 3D. Par exemple un clic ou déplacement de la souris.

Lorsque l'on charge une scène à partir d'un fichier, elle est créée de manière asynchrone. Le programme se poursuit et l'événement `OnSceneLoaded` est levé lorsque la scène est prête.

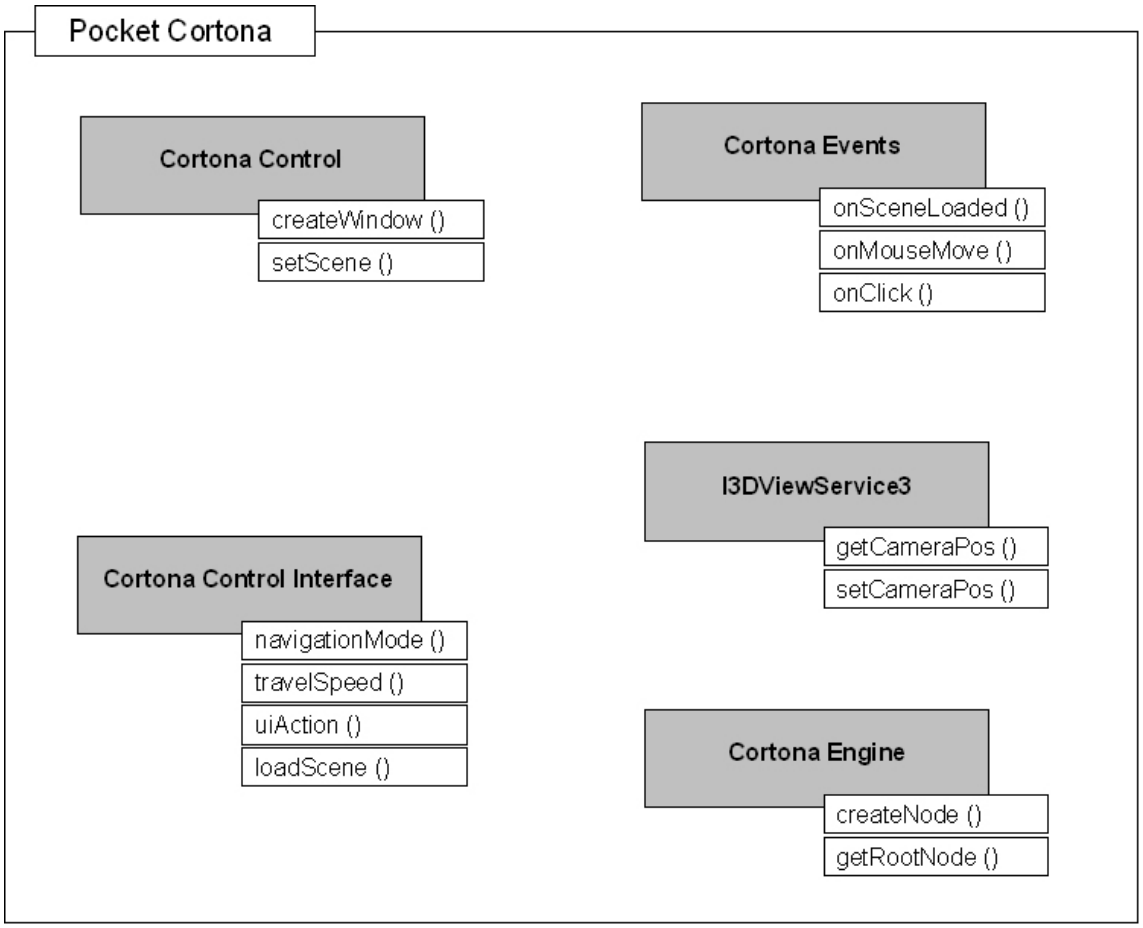

Fig. 4.2 – Les objets de la librairie Cortona

4.4 Quelques captures d'écran

Les fenêtres de connexion et de téléchargement permettent de communiquer avec le serveur.

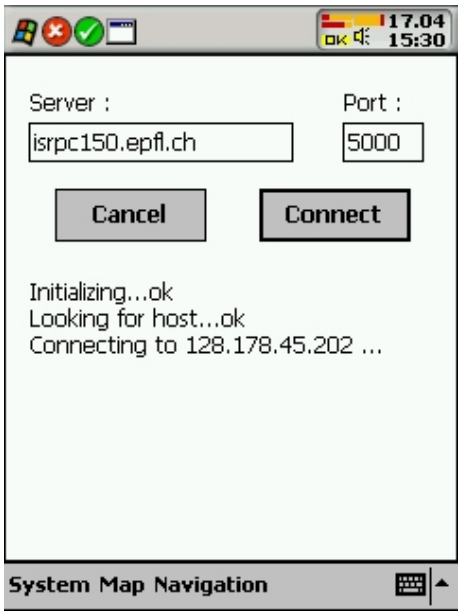

FIG. 4.3 – La fenêtre de connexion

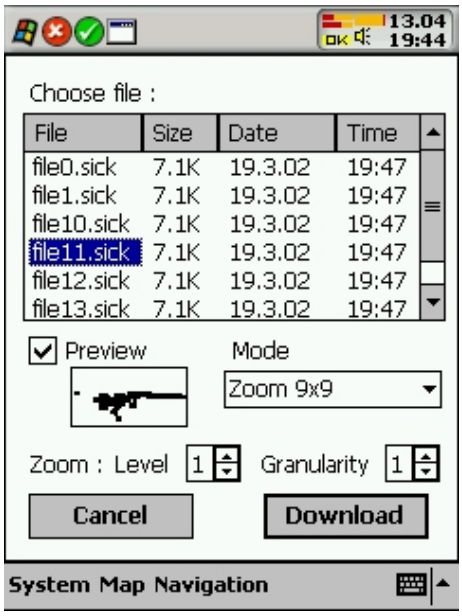

FIG. 4.4 – Choix d'un fichier

La représentation des terrains au moyen des librairies Cortona. On peut distinguer différents obstacles sur la carte, que l'on représente d'une part avec l'altitude et d'autre part avec des couleurs. On peut aussi voir le chemin parcouru ou restant à parcourir par le robot.

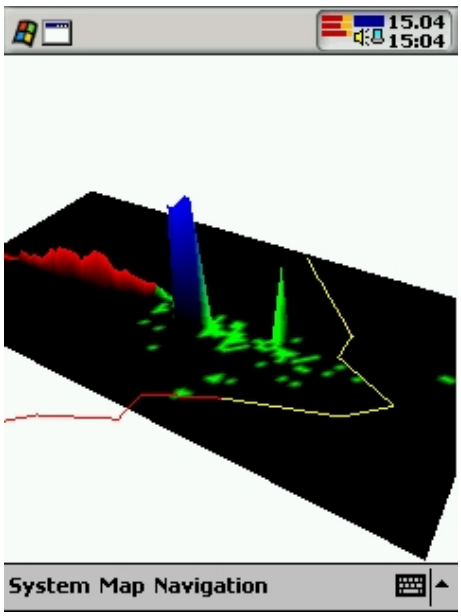

FIG. 4.5 – Une carte *VRML* typique

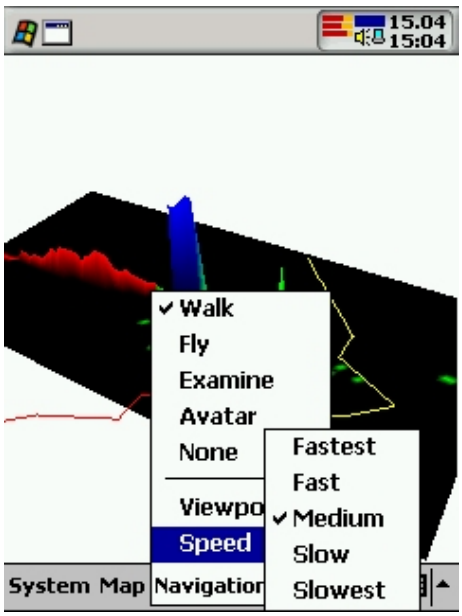

FIG. 4.6 – Les menus de navigation

Chapitre 5

Mise à jour de la carte

5.1 Evolution de la carte

Pour l'instant les données d'entrée proviennent de fichiers pré-enregistrés. Les cartes sont donc statiques et n'évoluent jamais. Cependant lors de développements ultérieurs notamment en connectant directement les capteurs au serveur, il est plausible d'imaginer que la carte puisse évoluer. Il faut donc trouver des solutions pour répercuter ces changements de données sur l'interface graphique 3D de l'utilisateur.

L'application cliente est capable de demander une mise à jour de la carte à intervalle fixe. La notification par le serveur de la modification de perception d'un capteur est prévue mais pas encore implémentée.

Même avec des fichiers de données statiques, il est cependant possible de modéliser des changements de perception en utilisant les différents fichiers disponibles successivement.

5.2 Comment envoyer les changements

Pour l'instant la carte entière, donc le fichier *VRML* complet, est retransmise à chaque mise à jour.

Il parait difficile de transférer seulement certains points de la carte qui auraient bougé. La fusion de plusieurs cartes, afin de représenter l'évolution est aussi un point intéressant mais difficile d'approche.

5.3 Scène dynamique dans Cortona

Grâce aux librairies du moteur Cortona, il est possible d'éditer les objets 3D en temps réel sur le client. Donc d'éditer ou de créer de nouveaux noeuds *VRML*, voire des ensembles de points.

Il serait finalement possible de transférer les points de la carte sous forme binaire. Le temps de transfert serait alors diminué. Les objets 3D devraient alors être créés sur le *PocketPC*. Il semble que ce ne soit pas impossible vu la puissance de calcul finalement assez importante.

Le zoom par exemple pourrait alors être calculé par l'application en *C++* plutôt qu'en JavaScript à l'intérieur du code *VRML*.

Une autre méthode serait de créer les objets *VRML* sur le serveur puis de transférer ceux-ci en binaire par le réseau. Il parait cependant impossible de lier les librairies Cortona sous Linux (plus de documentation et plus d'expérience seraient nécessaires).

Conclusion

L'application développée permet donc de disposer d'une interface utilisateur graphique tournant sur un *PocketPC* relativement puissant. En plus de pouvoir téléguider des robots, il devient maintenant possible de représenter les informations 3D renvoyées par des capteurs embarqués sur le robot. La puissance de l'ordinateur de poche s'avère suffisante pour manipuler aisément des représentations tridimensionnelles, pour autant qu'elles soient suffisamment traitées et simplifiées au préalable.

Il est bien sûr possible d'améliorer encore l'application, ou surtout d'y ajouter de nouvelles fonctionnalités. Mais on dispose déjà d'une base construite sur une architecture client-serveur performante et ouverte.

Les possibilités d'améliorations

Il faudrait pouvoir disposer de capteurs réels afin de pouvoir effectivement connecter ceux-ci au serveur et ainsi adapter le logiciel pour représenter des données effectives ou provenant de plusieurs capteurs.

D'autres travaux dans la simplification et la mise en forme des cartes pourraient aussi être faits. Pour des données plus complexes que celles disponibles actuellement, il faudrait peut-être mettre en œuvre d'autres méthodes de simplification ou des terrains pouvant avoir plusieurs niveaux de détail possibles [4].

Il est aussi possible d'aller plus loin dans la manipulation des données 3D sur le client lui-même. Cela donne la possibilité de pouvoir modifier les terrains affichés sans devoir faire de mise à jour complète.

Remerciements

Mark Moll et Todd Jochem pour l'algorithme «Line-strip classifier» qui a servi de point de départ pour le projet.

Bibliographie

[1] David FLANAGAN, *JavaScript - The Definitive Guide*, O'Reilly, 1997.

[2] Mark MOLL & Todd JOCHEM, *Line scan algorithm*, Applied Perception.

[3] Michael GARLAND, 1999, *Quadric-Based Polygonal Surface Simplification*, Ph.D. thesis, Technical Report CMU-CS-99-105, Computer Science Department, Carnegie Mellon University.
`http://graphics.cs.uiuc.edu/~garland/CMU/quadrics/quadrics.html`

[4] Multiple level-of-detail terrain models, *Tile Set Manager (SRI)*
`http://www.tvgeo.com/tsmApi`

[5] *Cortona Software Developers Kit - User Guide*, ParallelGraphics, 2001.

[6] Le projet PerceptOR Software Systems
`http://vrai-group.epfl.ch/projects/ati/perceptor/`

[7] T. FONG F. CONTI, S. GRANGE & C. BAUR, *Novel interfaces for remote driving : gesture, haptic and PDA*, SPIE 4195-33, SPIE Telemanipulator and telepresence Technologies VII, Boston, 2000.
`http://vrai-group.epfl.ch/projects/ati/pdadriver/`

[8] Le site officiel de Ekkla Research
`http://www.ekkla-research.com`

[9] Le site officiel de Superscape
`http://www.swerve3d.com`

[10] Le site officiel de Cortona
`http://www.parallelgraphics.com`

[11] Le support technique de Cortona
`support@parallelgraphics.com`

[12] The Virtual Reality Modeling Language (VRML), `ISO/IEC 14772-1:1997`, 1997.

[13] The VRML Repository
`http://www.web3d.org/vrml/vrml.htm`

[14] Ressources, tutoriels, cours VRML
`http://www.web3d-fr.com`

[15] Floppy's Web3D Guide
`http://web3d.vapourtech.com`

Annexe A

Extraits du fichier source VRMLclient.cpp

```
// ----------------------------------------------------------------------------
// -- Cortona global objects
// ----------------------------------------------------------------------------

CCortonaCtl                 CortonaCtl;       // cortona control object (main object)
CComPtr<IEngine>            pEngine;          // the engine
CComPtr<ICortona>           pCortona;         // cortona control interface
CComPtr<I3DViewService3>    pI3dvs;           // cortona I3DViewService interface
CCortonaEvents              CortonaEvents;    // cortona events handler
DWORD                       m_dwCoockie;
_POSTAG                     CamPos;           // Camera position


// ----------------------------------------------------------------------------
// -- Prototypes
// ----------------------------------------------------------------------------

ATOM MyRegisterClass(HINSTANCE, LPTSTR);
BOOL InitInstance(HINSTANCE, int);
HWND CreateRpCommandBar(HWND, unsigned int);
HWND CreateListView(HWND);
void CreateUpDownEdit(HWND hWndParent);
bool GetPreview(HWND hWndParent, char *buf);
void CheckPreview(HWND hWndParent, int item);
bool DownloadFile(HWND hWndParent, bool progress);

void ViewFile();
void CreateScene();
void LoadScene();
void InitCortona();
void ViewScene();
void RefreshScene();

void WhichMenu(int connected, int mapset, int cortona, int autoup);
void SetCortonaMode(short mode);
void SetCortonaSpeed(long speed);
void CortonaAction(TCHAR *cmd);

void SetMenuEnableCheck(unsigned int menu, unsigned int item, bool enable, bool check);
void SetNavigationModeMenu(short mode);
void SetNavigationSpeedMenu(long speed);

DWORD WINAPI ThreadProcConnect(LPVOID lpParameter);
```

```
// --------------------------------------------------------------------------
// -- Callback window functions
// --------------------------------------------------------------------------

LRESULT CALLBACK  WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK  AboutProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK  ConnectProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK  DownloadProc(HWND, UINT, WPARAM, LPARAM);


// --------------------------------------------------------------------------
// -- InitCortona()
// --------------------------------------------------------------------------
// -- Initialize Cortona window and objects
// --------------------------------------------------------------------------

void InitCortona()
{
  RECT rt;
  GetClientRect(hWnd, &rt);

  // init Cortona window
  if (!CortonaCtl.Create(L"Cortona", WS_VISIBLE | WS_CHILD, rt,
      CWnd::FromHandle(hWnd), ID_CORTONA_CONTROL))
  {
    eprintf("Unable to create Cortona window !");
    return;
  }

  // init Cortona Engine
  IDispatch* pdisp = CortonaCtl.GetEngine();
  if (!pdisp || FAILED(pdisp->QueryInterface(IID_IEngine, (void**)&pEngine)) || !pEngine)
  {
    eprintf("Can not get Cortona Engine");
    return;
  }

  // init Cortona control interface
  CortonaCtl.GetControlUnknown()->QueryInterface(IID_ICortona, (void**)(&pCortona));

  // init Cortona event class
  AfxConnectionAdvise(CortonaCtl.GetControlUnknown(), DIID__CortonaEvents,
                      CortonaEvents.GetIDispatch(FALSE), FALSE, &m_dwCoockie);

  // init the I3DViewService3 interface
  CortonaCtl.GetControlUnknown()->QueryInterface(IID_I3DViewService3, (void**)(&pI3dvs));

  CortonaLaunched = true;
}


// --------------------------------------------------------------------------
// -- ViewScene()
// --------------------------------------------------------------------------
// -- View a new scene launching Cortona
// --------------------------------------------------------------------------

void ViewScene()
{
  InitCortona();
  LoadScene();

  // Turn on the headlight (some scenes without self illuminating materials
  // can be invisible when headlight is turned off)
  CortonaCtl.SetHeadLight(TRUE);

  // Instructs engine to immidiately apply changes in of VRML node fields values
  pEngine->put_AutoRefresh(TRUE);
}
```

```
// ----------------------------------------------------------------------------
// -- LoadScene()
// ----------------------------------------------------------------------------
// -- Load a new scene from map file and initialize Cortona
// ----------------------------------------------------------------------------

void LoadScene()
{
  CortonaCtl.SetScene(L"\\My Documents\\map.wrl");
}


// ----------------------------------------------------------------------------
// -- SceneLoadedEvent()
// ----------------------------------------------------------------------------
// -- Called when the scene is loaded. Comes from CortonaEvent
// ----------------------------------------------------------------------------

void SceneLoadedEvent()
{
  short mode;
  long  speed;

  if (SetCamWhenLoaded) pI3dvs->SetCameraPos(&CamPos);

  pCortona->get_NavigationMode(&mode);
  SetNavigationModeMenu(mode);

  pCortona->get_TravelSpeed(&speed);
  SetNavigationSpeedMenu(speed);
}


// ----------------------------------------------------------------------------
// -- RefreshScene()
// ----------------------------------------------------------------------------
// -- Refresh the scene from map file
// ----------------------------------------------------------------------------

void RefreshScene()
{
  pI3dvs->GetCameraPos(&CamPos);
  LoadScene();
  SetCamWhenLoaded = true;
}


// ----------------------------------------------------------------------------
// -- CreateScene()
// ----------------------------------------------------------------------------
// -- Create a scene from a string
// ----------------------------------------------------------------------------

void CreateScene()
{
  // It is a must call if you want to edit empty scene
  CortonaCtl.Edit();

  INodeObject* pNewNode = NULL;

  BSTR bstr = SysAllocString(L"Shape { geometry Text {string \"VRMLclient\" fontStyle FontStyle
                              {family \"TYPEWRITER\" justify \"MIDDLE\"} maxExtent 5 length 3}}");
  pEngine->CreateNodeFromString(bstr, &pNewNode);
  SysFreeString(bstr);

  if (!pNewNode) { eprintf("Bad wrml"); return;}

  CComPtr<INodesCollection> RootNodes;
  if (FAILED(pEngine->get_RootNodes(&RootNodes)))
```

```
  {
    eprintf("Get root node failed !");
    return;
  }

  // NOTE: you can pass the next VARIANT if an operand in method is optional
  CComVariant Default(0,VT_ERROR);

  // add the node to the root nodes collection
  SUCCEEDED(RootNodes->Add(pNewNode, &Default));
}


// ----------------------------------------------------------------------------
// -- CortonaAction(TCHAR *)
// ----------------------------------------------------------------------------
// -- Performs an action or put in a specified mode (viewpoint stuff)
// ----------------------------------------------------------------------------
// -- Commands available : "plan", "pan", "turn", "roll", "goto", "align",
// --                      "next_vp", "prev_vp", "restore", "fit".
// ----------------------------------------------------------------------------

void CortonaAction(TCHAR *cmd)
{
  BSTR bstr = SysAllocString(cmd);
  pCortona->uiAction(bstr);
  SysFreeString(bstr);
}


// ----------------------------------------------------------------------------
// -- SetCortonaMode(short)
// ----------------------------------------------------------------------------
// -- Set the navigation mode
// ----------------------------------------------------------------------------
// -- Mode availables : nmNone, nmWalk, nmFly, nmExamine
// ----------------------------------------------------------------------------

void SetCortonaMode(short mode)
{
  pCortona->put_NavigationMode(mode);
}


// ----------------------------------------------------------------------------
// -- SetCortonaSpeed(long)
// ----------------------------------------------------------------------------
// -- Set the navigation speed
// ----------------------------------------------------------------------------
// -- Speeds : tsMuchSlower, tsSlower, tsNormal, tsFaster, tsMuchFaster
// ----------------------------------------------------------------------------

void SetCortonaSpeed(long speed)
{
  pCortona->put_TravelSpeed(speed);
}


// ----------------------------------------------------------------------------
// -- ViewFile()
// ----------------------------------------------------------------------------
// -- View a file with Cortona standalone application (not used anymore !)
// ----------------------------------------------------------------------------

void ViewFile()
{
  SHELLEXECUTEINFO si;

  si.cbSize = sizeof(SHELLEXECUTEINFO);
  si.fMask = 0;
```

```
  si.lpVerb = L"open";
  si.lpFile = L"\\Program Files\\Cortona\\Cortona";
  si.lpParameters = L"\"\\My Documents\\map.wrl\"";
  si.lpDirectory = L"";
  si.nShow = SW_SHOWNORMAL;

  ShellExecuteEx(&si);
}
```

Annexe B

Extraits du fichier source VRMLwriter.cpp

```
// ------------------------------------------------------------------------
// -- PointOk(int, int)
// ------------------------------------------------------------------------
// -- Check if a point is okay to be included in the map (colored or not groud)
// ------------------------------------------------------------------------

BOOLEAN PointOk(int x, int z)
{
  int i, j;
  float e;
  RGB c;
  BOOLEAN r = FALSE;

  e = getElevation(x, z);
  c = getColor(x, z);

  for (j=z-1; j<=z+1; j++)
    for (i=x-1; i<=x+1; i++)
      if (e != getElevation(i, j) || !IsRgbEqual(c, getColor(i, j)))
      {
      r = TRUE;
      break;
      }
  return r;
}


// ------------------------------------------------------------------------
// -- ColorNode()
// ------------------------------------------------------------------------
// -- Creates the "color" property for an Indexed(Face/Line)Set
// ------------------------------------------------------------------------

void ColorNode(int x0, int z0, int x1, int z1)
{
  int i, j;

  fprintf(f, "[ ");
  fprintf(f, "0 0 0,0 0 0,0 0 0,0 0 0,");

  for(j=z0; j<=z1; j++)
    for(i=x0; i<=x1; i++)
      if (PointOk(i, j))
        fprintf(f, "%.2f %.2f %.2f,", getColor(i, j).r, getColor(i, j).g, getColor(i, j).b);
  fprintf(f, "] ");
}
```

```
// ------------------------------------------------------------------------
// -- CoordNode()
// ------------------------------------------------------------------------
// -- Creates the "coord" property for an Indexed(Face/Line)Set
// ------------------------------------------------------------------------

void CoordNode(int x0, int z0, int x1, int z1, float s)
{
  int i, j, c = 4;

  fprintf(f, "[ ");
  fprintf(f, "%.4f %.4f %.4f,", s*(-(x1-x0)/2.0), s*(-(z1-z0)/2.0), -0.05);
  fprintf(f, "%.4f %.4f %.4f,", s*((x1-x0)/2.0), s*(-(z1-z0)/2.0), -0.05);
  fprintf(f, "%.4f %.4f %.4f,", s*(-(x1-x0)/2.0), s*((z1-z0)/2.0), -0.05);
  fprintf(f, "%.4f %.4f %.4f,", s*((x1-x0)/2.0), s*((z1-z0)/2.0), -0.05);

  for (j=0; j<=z1-z0; j++)
    for (i=0; i<=x1-x0; i++)
      if (PointOk(i+x0, j+z0))
      {
        fprintf(f, "%.4f %.4f %.4f,", s*((float)i-(x1-x0)/2.0), s*((float)j-(z1-z0)/2.0),
                getElevation(i+x0, j+z0));
        itab[i+j*(x1-x0+1)] = c++;
      }
  fprintf(f, " ] ");
}


// ------------------------------------------------------------------------
// -- CoordIndexNode()
// ------------------------------------------------------------------------
// -- Creates the "coordIndex" property for an Indexed(Face/Line)Set
// ------------------------------------------------------------------------

void CoordIndexNode(int x0, int z0, int x1, int z1)
{
  int i, j;

  fprintf(f, " [ ");
  fprintf(f, "0 1 2 -1 1 3 2 -1 ");

  for (j=0; j<=z1-z0-1; j++)
    for (i=0; i<=x1-x0-1; i++)
      if (PointOk(i+x0, j+z0) && PointOk(i+1+x0, j+z0) && PointOk(i+1+x0, j+1+z0)
          && PointOk(i+x0, j+1+z0))
      {
        fprintf(f, "%d %d %d -1 ", itab[i+j*(x1-x0+1)], itab[i+1+j*(x1-x0+1)],
                itab[i+(j+1)*(x1-x0+1)]);
        fprintf(f, "%d %d %d -1 ", itab[i+1+j*(x1-x0+1)], itab[i+1+(j+1)*(x1-x0+1)],
                itab[i+(j+1)*(x1-x0+1)]);
      }
  fprintf(f, " ]");
}


// ------------------------------------------------------------------------
// -- IndexedFaceSet()
// ------------------------------------------------------------------------
// -- Creates an IndexedFaceSet object
// ------------------------------------------------------------------------

void IndexedFaceSet(int x0, int z0, int x1, int z1, float s, VOID_F_2INT *color)
{
  fprintf(f, "IndexedFaceSet { ");
  fprintf(f, "\n solid FALSE");
  Coord(x0, z0, x1, z1, s);
  CoordIndex(x0, z0, x1, z1);
  if (color) color(x0, z0, x1, z1);
  fprintf(f, " }\n");
}
```

```
// ------------------------------------------------------------------------
// -- TransformT()
// ------------------------------------------------------------------------
// -- Creates a Translation node
// ------------------------------------------------------------------------

void TransformT(float x, float y, float z)
{
  fprintf(f, "Transform { translation %f %f %f children [ ", x, y, z);
}


// ------------------------------------------------------------------------
// -- Sensor()
// ------------------------------------------------------------------------
// -- Creates a mouse Sensor node
// ------------------------------------------------------------------------

void Sensor(int id)
{
  fprintf(f, "DEF sens%d TouchSensor { } ", id);
}

// ------------------------------------------------------------------------
// -- Switch()
// ------------------------------------------------------------------------
// -- Creates a Switch node
// ------------------------------------------------------------------------

void Switch(int id, int def)
{
  fprintf(f, "DEF sw%d Switch { whichChoice %d choice [ ", id, def);
}


// ------------------------------------------------------------------------
// -- Wrml_z()
// ------------------------------------------------------------------------
// -- Creates a VRML map with simple zoom (3x3)
// ------------------------------------------------------------------------

void Wrml_z(char *filename, int x0, int z0, int x1, int z1, float s)
{
  int i, j, n=3;

  f = fopen(filename, "w");
  // to store the coord indexes
  itab = (int *) malloc((x1-x0+1)*(z1-z0+1)*sizeof(int));

  fprintf(f, "#VRML V2.0 utf8\n");
  Background(RGB_WHITE);
  Viewpoint(x0, z0, x1, z1, s);
  Navigation("EXAMINE");

  Switch(1, 0);

    Group();
    for(j=0; j<n; j++)
      for(i=0; i<n; i++)
      {
        Group();
          TransformT(i*3-3, j*6-6, 0);
            Define(j*n+i);
            IndexedFaceSet(x0+(x1-x0)/3*i, z0+(z1-z0)/3*j, x0+(x1-x0)/3*(i+1),
                           z0+(z1-z0)/3*(j+1), s, Color);
          EndTransform();
          Sensor(j*n+i);
        EndGroup();
      }
    EndGroup();
```

```
    Group();
      Switch(2, 0);
        for(j=0; j<n; j++)
          for(i=0; i<n; i++)
          {
            TransformS(3, 3, 1);
            Use(j*n+i);
            EndTransform();
          }
      EndSwitch();

      Sensor(100);
    EndGroup();

  EndSwitch();


  fprintf(f, "DEF scr Script { \n");
  for (i=0; i<9; i++)
    fprintf(f, "eventIn SFBool s%d\n", i);
  fprintf(f, "eventIn SFBool s100\n");
  fprintf(f, "eventOut SFInt32 aff1\n");
  fprintf(f, "eventOut SFInt32 aff2\n");
  fprintf(f, "url \"javascript:\n");
  for (i=0; i<9; i++)
    fprintf(f, "function s%d(value) { if (value) {aff1=1; aff2=%d;} }\n", i, i);
  fprintf(f, "function s100(value) { if (value) {aff1=0; aff2=0;} }\n");
  fprintf(f, "\" }\n");


  fprintf(f, "ROUTE scr.aff1 TO sw1.whichChoice\n");
  fprintf(f, "ROUTE scr.aff2 TO sw2.whichChoice\n");
  fprintf(f, "ROUTE sens100.isActive TO scr.s100\n");
  for (i=0; i<9; i++)
    fprintf(f, "ROUTE sens%d.isActive TO scr.s%d\n", i, i);


  fclose(f);
  printf("Wrote file %s\n", filename);
}
```