|  |  |
| --- | --- |
| École de technologie supérieure | Eric Thé, Francis Bourdeau et Samuel Deslauriers-Gauthier |
| INF-155 Travail pratique #3 |
| Travail en équipe |  |
| *La voiture intelligente autoguidée* | Remise à la fin de session |

# Objectifs

* Se familiariser avec les enregistrements (*struct*) et le mode graphique.
* Utiliser les notions déjà acquises sur les fichiers textes et l’allocation dynamique.
* Apprendre comment faire une simulation avec un peu d’intelligence artificielle.

# Description du problème : Une voiture intelligente qui suit un chemin

# Le rêve de voir des voitures intelligentes autoguidées nous transporter de ville en ville n’est pas si lointain qu’on le pense. On n’est pas encore prêt à avoir des autoroutes remplies de voitures intelligentes mais on aura bientôt des navettes autoguidées pour transporter des gens à travers un terminal d’aéroport, un manège de maison hantée ou bien dans un tour touristique guidé à travers un musée ou un site historique.

# Récemment, des élèves de l’université MIT ont réussi à programmer une voiturette de golf à suivre un chemin tracé par une liste de points GPS. De plus, leur véhicule était muni d’un senseur qui pouvait détecter des piétons ou autres obstacles devant lui et le forcer ainsi à freiner pour éviter une collision. Votre tâche consiste à simuler le même comportement que ce véhicule.

# Développement et structure du programme

# Comme pour le TP2 votre travail sera divisé en plusieurs modules. Premièrement vous aller concevoir le module « donnees\_route », qui vous permettra de lire les données de départ à partir d’un fichier texte de configuration. Ensuite, vous allez compléter le module d’affichage « graphiques\_lib » avec lequel pourrez afficher la route et le chemin à suivre par votre navette intelligente. L’étape suivante verra l’introduction de la navette intelligente avec le module « t\_auto ». Votre navette pourra alors naviguer en suivant le chemin tracé par la liste de points à suivre et faire le parcours du début à la fin de façon fluide. Finalement, l’introduction des piétons sera rajouté au projet avec son module « t\_obstacle ». Les piétons et la navette devront donc apprendre à se détecter l’un et l’autre pour éviter des collisions.

Voici ces quatre modules expliqués en détails :

## Module « donnees\_route »

On vous offre le fichier « **donnees\_route.h** » qui contient entre autres la définition du type « **t\_pt2d**» qui permet de stoquer soit une position dans l’espace R2 ou un vecteur de vélocité ou d’accélération (pour des raisons de simplicité on va considérer ce type comme étant public à tous et ses deux champs « .X » et « .Y » sont accessibles en tout temps sans utiliser une fonction informatrice). On y défini aussi le type structuré « **t\_route** » avec lequel on pourra retenir les données de la route (les lignes ou bordures) ainsi que le type structuré « **t\_chemin** » avec lequel on pourra retenir les données du chemin à suivre (les points GPS). Notez que ces deux structures contiennent un tableau dynamique qui devra être alloué dès que l’on obtient le nombre d’éléments à partir du fichier texte de configuration.

Ce module offre les 8 fonctions suivantes :

* int lire\_fichier(char \*nomF, t\_route \*route, t\_pt2d \*depart, double \*dir\_dep,

t\_chemin \*chemin, int \*nb\_obs);

Cette fonction reçoit un nom de fichier texte à ouvrir. Les 6 autres paramètres sont des références vers des variables à remplir à partir des données du fichier. Voici le format unique de tous les fichiers textes de configuration (eg. fichier « Aeroport\_Parking.txt » :

1000 600 //dimensions de la route : dimx, dimy

7 //le nombre de lignes à afficher (nb\_lig)

0 0 0 600 //les (nb\_lig) lignes : x1, y1, x2, y2

1000 0 1000 600

0 0 1000 0

0 600 1000 600

0 200 800 200

800 200 800 400

200 400 800 400

100 100 90 //position et angle de départ (degrés) de l’auto : posx, posy, angle

28 //le nombre de points à suivre (nb\_pts)

200 100 // les (nb\_pts) points du chemin : x, y

300 100

… …

700 300

5 //le nombre d’obstacles à générer

Notez que le tableau dynamique « .liste\_lignes » devra être alloué dynamiquement dès que l’on aura lu le nombre de lignes et que le tableau dynamique « .liste\_pts » devra être alloué dynamiquement dès que l’on aura lu le nombre de points à suivre. Notez aussi que l’angle de départ de l’auto devra être convertie en radians avant d’être retourné via le paramètre « \*dir\_dep » avec la formule *rad* = *angle/180* \* *π*. La fonction retourne **1** si le fichier texte a pu être ouvrir correctement, et **0** sinon.

* void detruire\_route(t\_route \*route, t\_chemin \*chemin);

Cette fonction va libérer les allocations dynamiques des listes dans la route et le chemin reçu. Si ces deux pointeurs valaient déjà NULL, on ne fera rien.

* double dist(t\_pt2d p1, t\_pt2d p2);

Simplement calculer et retourner la distance scalaire entre les points *p1* et *p2*.

Les 5 fonctions suivantes permettent d’interroger les champs de « t\_route » et « t\_chemin » :

* void obt\_dim\_route(const t\_route \*route, int \*dimx, int \*dimy);

Permet d’obtenir les dimensions de la route (champs « .dimx » et « .dimy »).

* t\_ligne obt\_ligne(const t\_route \*route, int pos);

Permet d’obtenir une copie de la ligne #*pos* dans la route (indice à partir de 0).

* int obt\_nb\_lignes(const t\_route \*route);

Permet d’obtenir le nombre total de lignes dans la structure de route.

* t\_pt2d obt\_pt(const t\_chemin \*chemin, int pos);

Permet d’obtenir une copie du point #*pos* dans la liste des points (indice à partir de 0).

* int obt\_nb\_pts(const t\_chemin \*chemin);

Permet d’obtenir le nombre de points à suivre dans la structure du chemin.

## Module « graphiques\_lib »

Ceci est le module d’affichage, il est quasi-complet. Vous devez écrire les deux fonctions d’affichage « **dessiner\_route** » et « **dessiner\_chemin** ». Pour la fonction « dessiner\_route » vous faites des lignes blanches pour illustrer chacune des lignes (ou bordures) de la route. Dans le cas de la fonction « dessiner\_chemin », faites afficher des petits points rouges (avec la constante POINT) pour illustrer chaque point du chemin à suivre. Notez qu’un chiffre (à partir de 1) devra être affiché à côté de chaque point pour illustrer l’ordre des points à suivre. N’oubliez pas d’inclure et d’utiliser les fonctions informatrices du module « **donnees\_route.h** ».

Votre première version de programme principal pour tester les modules « donnees\_route » et « graphiques\_lib » aura la forme suivante :

Initialiser le mode graphique

Demander et saisir le nom du fichier de données avec la fonction *saisie\_nomF()*

Si (lecture du fichier de données a réussi) {

* Effacer l’écran
* Dessiner la Route
* Dessiner le Chemin
* Faire une pause-écran

}

Fermer le mode graphique

## Module « t\_auto »

Le prochain module à implémenter sera le module de gestion de la navette intelligente. Vous êtes responsable de la conception complète de ce module. Au début du fichier d’interface (\*.h) vous allez inclure le module « **graphiques\_lib.h** » et mettez les déclarations suivantes :

/\* les dimensions de l'auto \*/

#define LONG 70

#define LARG 40

typedef struct{ //Enregistrement des infos sur l'auto

t\_pt2d acc; //vecteur accélération

t\_pt2d vel; //vecteur vitesse

t\_pt2d position; //position de l'oeil (point de référence)

int longueur, largeur; //dimensions de l'auto

t\_pt2d supG, supD, infG, infD; //coordonnées des 4 coins de l'auto (pour afficher)

double dir; //angle de direction (en radians)

int attente; //booléen pour indiquer si l'auto attends

//le passage d'un piéton

} t\_auto;

Le type de base « **t\_auto** » contient donc toutes les informations pertinentes sur votre navette.

Ce module offrira les 4 fonctions publiques suivantes à développer :

* t\_auto init\_auto(t\_pt2d pos\_depart, double dir\_depart);

Cette fonction va créer et initialiser la navette avec les données de départ. La longueur et la largeur de la navette seront initialisées avec les constantes (LONG, LARG). La position de départ ainsi que l’angle de départ de la navette sont copiés dans les champs « .position » et « .dir » respectivement. Les vecteurs d’accélération et de vitesse sont initialement mis à zéro de même que le booléen « .attente ».

Pour ce qui est des 4 coins du rectangle représentant la navette à l’écran (supG, supD, infG et infD), ces positions doivent être calculées en fonction de la direction actuelle de la navette (.dir). Sans entrer dans trop de détails trigonométriques, voici les formules à utiliser pour le calcul de ces quatre coordonnées :

/\* calculer les 2 coins du devant \*/

supG.X = position.X - (cos(direction) \* (moitié de la largeur))

supG.Y = position.Y + (sin(direction) \* (moitié de la largeur))

supD.X = position.X + (cos(direction) \* (moitié de la largeur))

supD.Y = position.Y - (sin(direction) \* (moitié de la largeur))

/\* calculer les 2 coins du derrière \*/

infG.X = supG.X - (sin(direction) \* longueur)

infG.Y = supG.Y - (cos(direction) \* longueur)

infD.X = supD.X - (sin(direction) \* longueur)

infD.Y = supD.Y - (cos(direction) \* longueur)

* void obt\_pos\_auto(const t\_auto \*navette, t\_pt2d \*pos\_ref,

t\_pt2d \*supG, t\_pt2d \*supD, t\_pt2d \*infG, t\_pt2d \*infD);

Cette fonction informatrice sert à récupérer la position de référence et les 4 coins de la navette. Cette fonction sera utile à chaque fois que l’on voudra faire afficher la navette.

* void changer\_acc\_auto(t\_auto \*navette, t\_pt2d dest);

Cette fonction va calculer la nouvelle accélération de la navette pour la diriger vers le point-cible « *dest* » reçu en paramètre. On pourra ainsi la réorienter, l’accélérer ou la ralentir.

Voici les étapes de ce calcul :

Déclarez 2 variables, **dX** = dest.X - (position-auto).X et **dY** = dest.Y - (position-auto).Y

et calculez la **distance** entre *dest* et la position de l’auto.

Vecteur de base d’accélération (**aX**, **aY**) : aX = dX/distance et aY = dY/distance

Si la **distance** est inférieure à la moitié de la largeur de l’auto, réduire le vecteur d’accélération de 90%.

Si **dX** est inférieur à 1.0 on fixe **aX** à 0, si **dY** est inférieur à 1.0 on fixe **aY** à 0 (cette mise-à-jour est pour éviter les divisions par zéro plus tard lorsque dX ou dY sont près de zéro).

* void deplacer\_auto(t\_auto \*navette);

Cette fonction va faire la mise-à-jour du vecteur de vitesse ainsi que de la position actuelle de la navette. Normalement, on ajoute le vecteur d’accélération fois le temps écoulé à la vitesse pour obtenir le nouveau vecteur de vélocité et on ajoute de vecteur de vélocité fois le temps écoulé à la position actuelle pour obtenir la nouvelle position. Par contre, nous allons grandement simplifier nos calculs en utilisant un facteur de temps écoulé constant (=1), donc on n’aura pas besoin de multiplier explicitement par le temps à chaque étape.

Voici les 4 étapes principales de cette mise-à-jour :

Mise-a-jour de la vitesse : nouveau vecteur = ancien vecteur + vecteur d’accélération

NORMALISER : si la longueur du nouveau vecteur de vitesse est supérieure à la vitesse maximale MAXVEL, on divise le vecteur par sa longueur et on multiplie par MAXVEL.

Mise-a-jour de la position : nouvelle position = ancienne position + vecteur de vélocité

Mise-a-jour de la direction : nouvelle direction = atan( vélocité en X / vélocité en Y )

Les fonctions trigonométriques *sin()* et *cos()* fonctionnent avec des angles compris entre [-*π*.. *π*] mais la fonction *atan()* peut retourner une valeur qui n’est PAS dans cet intervalle.

Donc vous devez faire les ajustements suivants suite au calcul de la nouvelle direction :

Si la différence absolue entre l’ancienne direction et la nouvelle direction est supérieure à 3*π* / 2 (= 3 quarts de tour), on soustrait (-*π*)à la nouvelle direction (inversion de signe).

Si la direction est négative ET inférieure à (-*π*), on la convertie en valeur positive < *π.*

Donc, direction = *π –* ( *|* direction *| - π )* //avec *fabs()* de *<*math.h*>*

Mise-a-jour des 4 coins : procédez de la même manière que dans la fonction « *init\_auto* ».

AIDE : déclarez une constante **DEMI\_PI** égale à asin(1.0) et vous avez votre valeur *π/2.* Ensuite les autres valeurs *π* et *3π / 2* peuvent être des constantes qui utilisent DEMI\_PI.

Le programme principal pour tester le module « t\_auto» aura la forme suivante :

Initialiser le mode graphique

Demander et saisir le nom du fichier de données avec la fonction *saisie\_nomF()*

Si (lecture du fichier de données a réussi) {

Initialiser l’auto avec « init\_auto() »

Dessiner la route et le chemin (optionnel)

cible = Récupérer le premier point du chemin

Afficher ce point à la droite de la route avec « afficher\_pos() »

Afficher l'auto

dist\_precedente = distance entre l’auto et la cible

Tant que (on n’a pas terminé le chemin) :

Changer l’accélération de l’auto pour aller vers la cible

Effacer et réafficher la route (pour effacer l’auto en même temps)

Déplacer l'auto

Afficher l'auto

Faire une pause de 50 msec.

Si on a pesé sur ESC on quitte la boucle immédiatement

distance = distance entre l’auto et la cible

Si (on est à moins d'une demi-largeur de la cible ET

qu'on s'éloigne de la cible),

cible = Récupérer le prochain point s’il en reste, sinon terminé = VRAI

distance = calculer la nouvelle distance entre l’auto et la cible

Afficher ce point à la droite de la route avec « afficher\_pos() »

Fin Si

Transférer la distance dans dist\_precedente

Fin boucle

Fin Si

Fermer le mode graphique

Avec ce programme de test votre navette devrait faire le parcours du début à la fin de façon fluide. Si ce n’est pas le cas, la meilleur façon de débugger votre programme sera de faire afficher toutes les valeurs importantes (acc, vélocité, position, cible, direction, etc.) dans la console texte avec des *printf()*. Vous allez ainsi pouvoir vérifier vos calculs à chaque itération. La détection d’une touche se fera avec la fonction « saisie\_touche() » du module graphique.

## Module « t\_obstacle »

Vous êtes maintenant prêts à ajouter les obstacles (ou piétons) dans la zone d’affichage de la route. Chaque obstacle aura une position, un vecteur de vélocité et une couleur d’affichage (un entier). Une liste d’obstacles est simplement un enregistrement contenant un tableau dynamique d’obstacles avec un entier qui indique le nombre d’obstacles dans cette liste. À vous de définir les types structurés « **t\_obstacle** » et « **t\_liste\_obs** » dans votre fichier d’interface « **t\_obstacle.h** ». Ce fichier devra inclure le module « **graphiques.lib.h** ».

Ce module va offrir 4 fonctions publiques et une fonction privée. Voici les fonctions publiques :

* t\_liste\_obs creer\_liste\_obs(int nb\_obs, int dimx, int dimy);

Cette fonction est le constructeur de la liste des obstacles. La quantité d’obstacles désirée « *nb\_obs* » et les dimensions de la route sont reçus en paramètre et proviennent du fichier texte de configuration. Le tableau dynamique des obstacles devra premièrement être alloué et le nombre d’obstacles copié dans le champ approprié de la structure.

Ensuite, pour chaque obstacle à générer, on doit attribuer une position aléatoire située à l’intérieur de la route. Pour des raisons techniques, cette position ne doit pas être trop collée à une des bordures de la route, donc assurez-vous qu’elle soit à une distance minimale de DIST\_MIN = 5 pixels d’une bordure. Le vecteur de vélocité sera initialisé avec des valeurs non-zéro entre [-3..-1] ou [1..3]. Pour ce qui est de la couleur d’affichage, veuillez exclure les valeurs 0 (noir), 1 (bleu) et 15 (blanc) car ce sont des couleurs « réservées ». La fonction retourne la nouvelle structure ainsi créée, ou bien une structure vide si l’allocation dynamique a échouée.

* void detruire\_obs(t\_liste\_obs \*obstacles);

Simplement libérer l’allocation dynamique de la liste d’obstacles.

* void afficher\_obs(const t\_liste\_obs \*obstacles);

Simplement afficher chaque obstacle à l’écran avec la fonction graphique « dessiner\_rond ».

* void deplacer\_obs(t\_liste\_obs \*obstacles, int dimx, int dimy);

Cette fonction va effectuer la mise-à-jour de la vélocité et de la position de tous les obstacles de la liste. Tout comme avec les calculs vectoriels de la navette, on considère le temps écoulé comme constant (=1) donc inutile de multiplier chaque calcul par le temps.

La nouvelle position d’un obstacle = position actuelle + vélocité.

Suite à ce calcul, si la nouvelle position de l’obstacle est située à moins de DIST\_MIN pixels d’une des bordures de la route, vous devez inverser la composante du vecteur de vélocité qui a causée la collision avec ce « mur ». Pour des raisons de simplicité, les obstacles peuvent traverser les lignes blanches internes de la route sans problèmes.

La fonction privée du module fera la simulation de la « vision » de l’obstacle. Les obstacles pourront ainsi voir directement devant eux et éviter d’avoir des collisions avec la navette.

* static int detecter\_auto(const t\_obstacle \*obs);

Il s’agit ici de tester 5 positions situées devant l’obstacle (selon la direction de sa vélocité). Voici une illustration des 5 positions à tester pour un obstacle en mouvement :

p1

p4

p2

p1 = (p3.X + dy, p3.Y – dx) //*vraiment à gauche*

p2 = (p3.X + dy/2, p3.Y – dx/2) //*un peu à gauche*

p3

p3 = (pos.X + dx, pos.Y + dy) //*30 pix.* *droit devant*

(dx, dy)

p4 = (p3.X - dy/2, p3.Y + dx/2) //*un peu à droite*

p5

p5 = (p3.X - dy, p3.Y + dx) //*vraiment à droite*

Votre vecteur de déplacement (dx, dy) devra avant tout être calculé de la façon suivante :

vecteur déplacement = (vecteur vélocité / (longueur du vecteur)) \* DIST\_MAX (= 30 pixels)

Ensuite sondez le contenu des 5 positions directement à l’écran en utilisant la fonction graphique « detecter\_pixel ». Si on détecte la couleur AUTO (bleu) on retourne immédiatement **1**, sinon la fonction retourne **0**. Les autres couleurs seront ignorées.

Ajout de la détection de la navette :

Dans votre fonction « créer\_liste\_obs », à chaque fois que vous allez attribuer une position initiale à un obstacle, si vous détectez la navette à cette position vous devez choisir une autre position de départ. On veut que les obstacles soient loin de la navette.

Dans votre fonction « deplacer\_obs », commencez par faire appel à « detecter\_auto ». Si on a détecté la navette devant l’obstacle à déplacer, faites dévier sa trajectoire en faisant :

* Inverser la composante de vélocité la plus forte en multipliant par -1.1.
* Réduire la composante de vélocité la plus faible de moitié.

Ensuite, appliquez immédiatement un déplacement équivalent à deux fois la nouvelle vélocité ainsi obtenue. Votre « piéton » agira comme si il était surpris par l’auto devant lui.

Modification du programme principal pour ajouter les obstacles (inclure « **t\_obstacle.h** ») :

* Déclarez une nouvelle variable « t\_liste\_obs obstacles ».
* Avant de démarrer la boucle principale, créer les obstacles et affichez-les à l’écran.
* Dans la boucle, tout de suite après avoir dessiné l’auto, déplacez et affichez les obstacles.
* N’oubliez pas de détruire la liste d’obstacles à la sortie de la boucle de simulation.

# Ajout de la détection des obstacles

Puisque les obstacles ont une vision périphérique limitée (les piétons sont trop occupés à regarder leur téléphone), ils peuvent se faire frapper de côté ou par derrière par la navette. On doit donc équiper la navette avec un senseur doit permettra de détecter les obstacles se trouvant devant lui. Nous allons donc imiter la présence d’un tel senseur avec une nouvelle fonction privée dans le fichier « t\_auto.cpp » :

* static int detecter\_obs(const t\_auto \*navette);

La navette aura une plus grande vision périphérique que les piétons. En fait nous allons appliquer un rayon de détection de 180 degrés (de .dir - *π*/2 a .dir + *π*/2 par pas de 3 degrés) et ce à trois distances différentes (r1, r2, r3) à partir de la proche à la plus éloignée.

Voici à quoi ressembleraient tous les points de détection à tester devant la navette :

.dir + *π*/2 r1 = (largeur de l’auto) / 2

r2 = r1 +15

r3 = r2 + 15

.dir - *π*/2 **α** ***p***

Donc, si l’angle de détection vaut **α** radians, les coordonnées du point de détection ***p*** seront calculées à partir du point de référence de l’auto (.position) : ***p*** = position + (**dx**, **dy**)

Où **dx** = (rayon de détection) \* sin(**α**) et **dy** = (rayon de détection) \* cos(**α**). On fait la détection d’obstacle avec la fonction « detecter\_pixel ». Si on détecte n’importe quelle couleur autre que AUTO (bleu) la fonction retourne immédiatement **1**, sinon on retourne **0**.

Cette fonction sera appliquée dans le code de la fonction « changer\_acc\_auto ». Au début de cette fonction faite la détection d’obstacle. Si elle retourne vrai, la navette doit freiner donc le nouveau vecteur d’accélération sera le vecteur actuel de vélocité fois -70%. Vous allez aussi allumer le champ booléen « .attente » en le mettant égal à **1**. Sinon, on fera le calcul normal d’accélération. Ensuite dans le calcul normal d’accélération, si le champ « .attente » est allumé, on fera accélérer graduellement la navette en réduisant son vecteur d’accélération de 30% et on éteint le champ « .attente ». Votre programme principal demeurera inchangé puisque l’ajout de la détection d’obstacles se fait uniquement dans « changer\_acc\_auto ». Par contre n’affichez pas les points-cibles car ils seront détectés comme des obstacles à l’écran !

# Programme principal et menu

Il ne reste qu’à faire le menu principal et la gestion de ses 3 options. Au début de votre « *main* » initialisez une variable « char choix\_menu = ’**N**’; » pour forcer la lecture d’un fichier texte dès le départ de l’exécution et initialisez le mode graphique.

Ensuite, voici les tâches de la boucle principale de votre programme :

Répétez :

Selon le choix-menu,

choix-menu = ’**N**’:

Faire le programme qui testait les modules « donnees\_route » et « graphiques\_lib » (section 3.2). Sauf qu’on enlève la pause-écran et on ajoute l’affichage du menu. N’oubliez pas de détruire la route avant de lire un nouveau fichier de données. On fermera le mode graphique seulement si l’ouverture du fichier texte a échoué.

choix-menu = ’**D**’:

Faire le programme de simulation qui testait les modules « t\_auto » et « t\_obstacle » (sections 3.3 & 3.4). Sauf qu’on ajoute l’affichage du menu et on élimine l’initialisation du mode graphique et la lecture du fichier texte. On ne fermera pas le mode graphique ici. Tout le code de cette étape (sauf l’appel de « init\_auto ») devrait idéalement être une sous-fonction définie dans le module principal.

choix-menu = ’**Q**’:

Détruire la route et fermer le mode graphique.

Fin des choix-menu

Si une touche-clavier a été pesée saisir le choix-menu, sinon AUCUN\_CHOIX

Tant que (le choix n’est pas ’**Q**’).

# Contraintes de l'enseignant

Votre programme devra respecter les contraintes suivantes :

* Remettez une copie imprimée de votre programme avec une page-titre de présentation sur laquelle vous inscrivez vos noms, codes permanents, le numéro de votre groupe ainsi que le nom de votre enseignant.
* Il devra respecter les exigences de remise des travaux pratiques. N’oubliez pas de bien commenter votre programme. L’orthographe du français sera aussi évaluée.
* Il ne devra contenir aucune variable globale publique.

BON TRAVAIL!