Projet
conception d'un robot mobile

Professeur : Valentin Gies

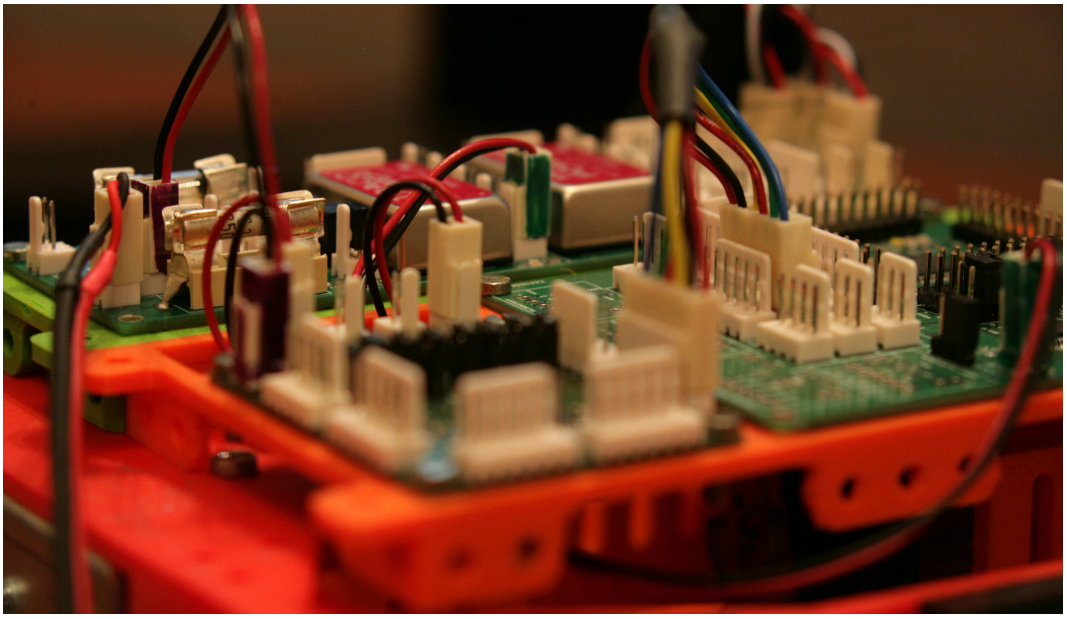

Table des matières

1 A la découverte de la programmation orientée objet en C#

Dans cette partie, vous allez apprendre à programmer en C# avec des interfaces graphiques en *WPF* (Windows Presentation Foundation). Le but est de réaliser un terminal permettant l'envoi et la réception de messages sur le port série du PC. Ce terminal servira dans un premier temps de messagerie instantanée entre deux PC reliés par un câble série, puis il servira ensuite à piloter un robot mobile tout en observant son comportement interne.
Pour commencer, vous apprendrez à travailler avec les objets de base des interfaces graphiques (*Button*, *RichTextBox*, ...). En particulier vous apprendrez à gérer les propriétés de ces objets et les évènements qui leurs sont associés.

1.1 Simulateur de messagerie instantanée

$\Longrightarrow$ Créez un projet C# dans Visual Studio. Pour cela lancer Visual Studio puis *Fichier* $\rightarrow$ *Nouveau* $\rightarrow$ *Projet*. Choisir *Application WPF (.NET Framework)*.
Avant de créer le projet, donnez lui un nom explicite tel que *RobotInterface* et spécifiez un chemin d'accès sur le disque dur tel que *C :/Projets/RobotWPF/*. Créez le projet, vous devriez avoir un écran similaire à celui de la figure 1.

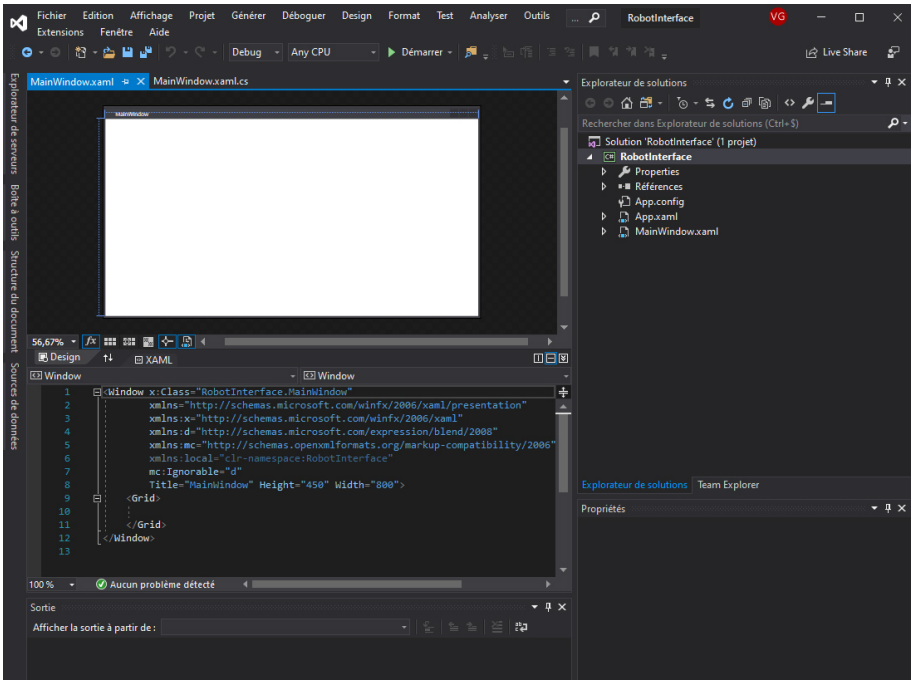

FIGURE 1 – Projet Visual Studio nouvellement créé

La partie droite de l'écran correspond à l'explorateur de solution, qui vous permet de voir les classes du projet, mais également les interfaces graphiques (fichiers *XAML*) en *C# WPF*. A la création du projet, un écran graphique dénommé *MainWindow.xaml* est créé par défaut. Il apparaît à gauche de l'écran en version graphique (en haut) et code XAML (en bas). Dans son état de base, il continet juste une grille (*Grid*) vide.

$\Longrightarrow$ Ajoutez à présent à *Form1* deux objets de type *GroupBox* en les faisant glisser depuis la *Boite à outils* $\rightarrow$ *Conteneurs*. Une *GroupBox* est une boite destinée à contenir d'autre éléments graphiques. Par défaut sa bordure est transparente. En regardant le code XAML, vous pouvez vous apercevoir que les GroupBox ont été ajoutées au code comme indiqué à la figure Figure 2 dans la partie centrale à gauche. Il est important de noter que le XAML décrit totalement ce qui apparait graphiquement dans la partie du dessus.

$\Longrightarrow$ En cliquant dans l'une des *GroupBox* dans la partie graphique, vous pouvez à présent modifier leurs

propriétés. Celles-ci apparaissent en bas à droite de l'écran comme indiqué à la Figure 2. Vous pouvez par exemple mettre une couleur de fond (dans la catégorie Pinceau, définir une couleur uniforme de *Background* valent #FFDDDDDD par exemple) et une bordure noire (dans la catégorie Pinceau, mettre *BorderBrush* en couleur uniforme à #FF000000 par exemple). Faire de même pour la seconde *GroupBox*.

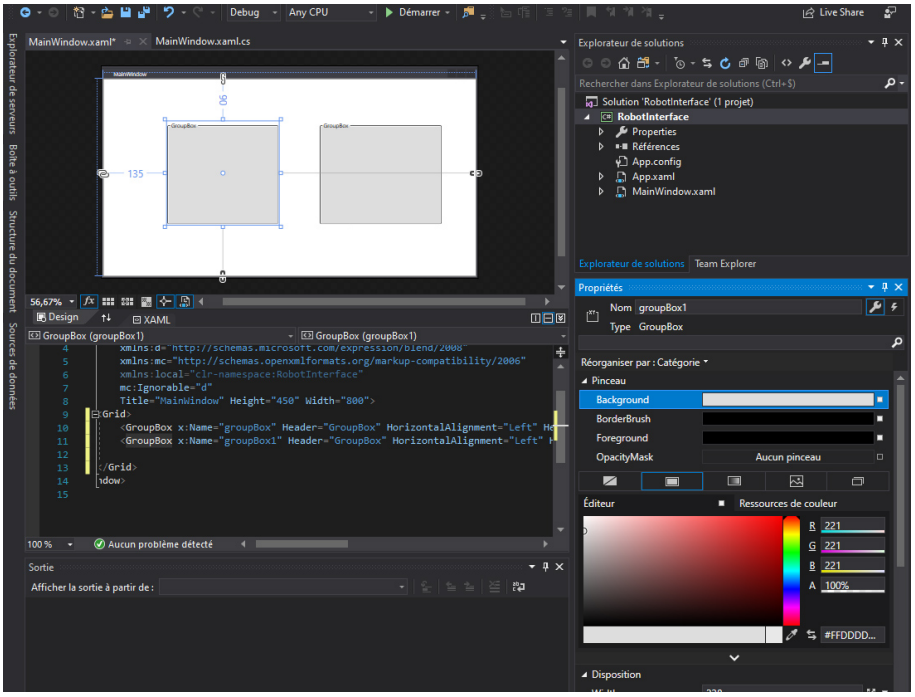

FIGURE 2 – Modification des propriétés de la GroupBox

⟹ A présent, vous devez avoir compris comment modifier les propriétés d'objets graphiques en WPF. Par vous même, renommez la *GroupBox* de gauche en *Emission* et celle de droite en *Réception*.

⟹ Arrivés à ce stade, vous pouvez voir à quoi ressemble votre application en la lançant. Pour cela appuyez sur *F5* ou *Démarrer* (flèche verte).

Si vous redimensionner votre fenêtre lors de l'exécution de l'application, vous devez constater que les GroupBox ne se redimensionnent pas et restent en position initiale. C'est regrettable et peu conforme aux standards des applications modernes. Le *WPF* est fait pour gérer simplement ces situations, ce qui lui confère un avantage indéniable par rapport au *WinForms* classiques. Pour ce faire, nous allons pousser plus loin le concept de grille introduit tout au début de cette partie.

⟹ Après avoir sélectionné la grille de fond d'application, cliquez sur le configurateur de Collones (*ColumnDefinitions*) dans les propriétés de la grille, onglet *Disposition* comme indiqué à la figure 3. Un éditeur de collections s'ouvre. Ajouter 5 ColumnDefinition (5 colonnes). Dans la première, la 3e et la 5e, spécifier une largeur de 30 pixels. Dans la 2e et la 4e, spécifiez une largeur de 1 Star. Vous devriez avoir une mise en forme des colonnes ressemblant à la figure figure 3.

Quelques explications sont nécessaires : une largeur en *pixels* sera fixe quelque soit la taille de la fenêtre de l'application. Une largeur en *Star* sera dépendante de la taille de la fenetre de l'application : dans notre cas, si la fenêtre a une largeur de 590 pixels, une fois retiré les 3 colonnes de 30 pixels fixes, il reste 500pixels à répartir entre deux collonnes de même taille (1 *Star* chacune).

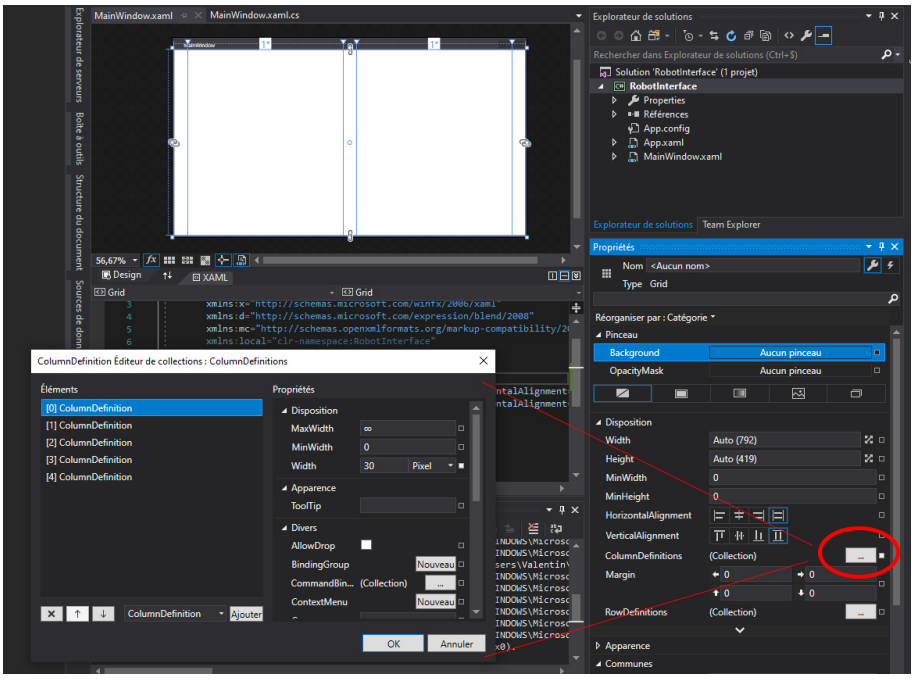

FIGURE 3 – Modification des propriétés des colonnes d'une *Grid WPF*

⟹ Maintenant que vous maitrisez les colonnes, faire de même avec les lignes (*Rows*), et définissant une première et une 3e ligne de 30 *pixels*, et une 2e ligne de 1 *Star*.

⟹ Il est temps à présent d'ancrer les GroupBox initialement créées dans les cases de la Grid. Pour cela sélectionnez l'une des GridBox (soit en passant avec la souris au dessus de l'endroit ou elle doit être, même si elle est cachée, soit en cliquant sur la ligne de la GroupBox dans le XAML). Glissez la dans la case de la grille ou vous voulez la mettre (ici une des grandes cases à redimensionnement automatique). Dans l'onglet *Disposition* des *Propriétés*, définissez la largeur et la hauteur en automatique, puis mettez les marges à 0 dans toutes les directions et les alignements horizontaux et verticaux à *Stretch*. Vous devriez avoir des propriétés proches de celles de la figure 4.

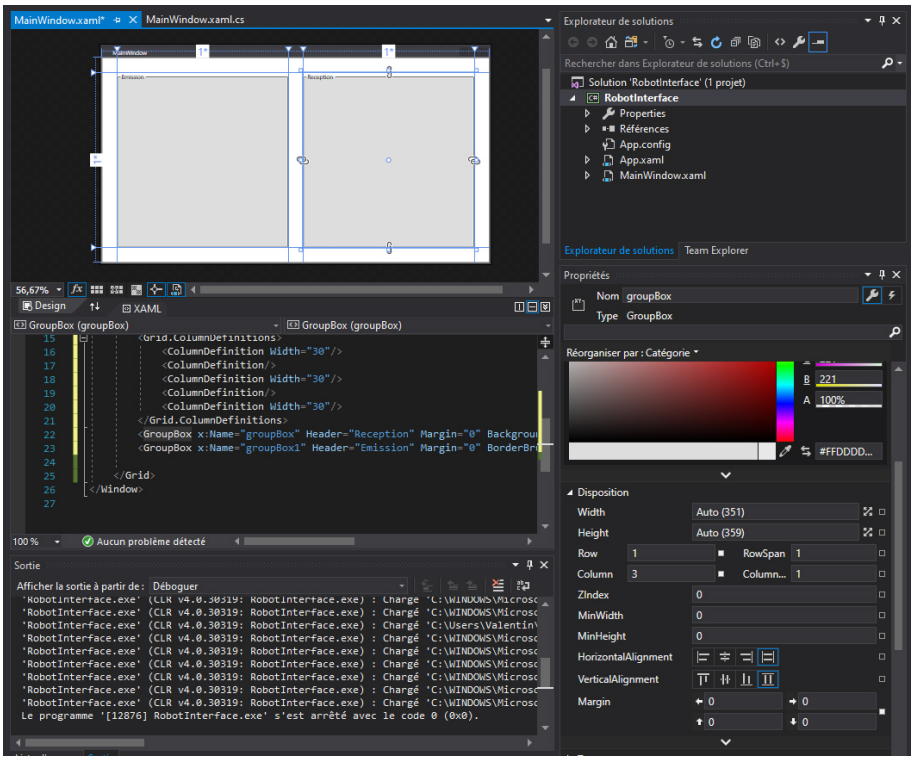

FIGURE 4 – GroupBox WPF en version redimensionnement automatique en fonction de la taille de l'application

Si vous exécutez le programme, vous pouvez vérifier que les *GroupBox* sont bel et bien redimensionnées dynamiquement si on change la taille de la fenêtre.

⟹ A présent, vous maitrisez le placement de composants dans une fenêtre de taille dynamique. Ajoutez à la *GroupBox Emission* une *TextBox* depuis la *Boite à Outils*. Faites en sorte que cette TextBox prenne toute la place possible dans la GroupBox (*Width* et *Height* en *Auto*, *Margins* à 0, et *Alignement* à *Stretch*), et supprimez sa couleur de fond et de bordure. Changez le nom de cette *TextBox* en *textBoxEmission*. Enfin, supprimez le texte par défaut *TextBox* dans les propriétés communes, et mettez la propriété *AcceptsReturn* à true (pour permettre des texte de plusieurs lignes).

⟹ Faites de même avec une *TextBox* de réception, en mettant en plus la propriété *IsReadOnly* à true afin d'empêcher l'utilisateur d'écrire dans cette *RichTextBox*. Exécutez le code, vous pouvez écrire dans la fenêtre d'envoi mais pas dans celle de réception.

⟹ Vous allez à présent rajouter un bouton à la grille pour envoyer les messages écrits dans la *TextBox* d'émission. Pour cela modifier la grille en rajoutant deux lignes de hauteur fixe égale à 30 pixels. Insérer en suite depuis la ToolBox un Bouton à l'avant dernière ligne sous la *RichTexBox* d'émission comme indiqué à la figure5. Redimensionner ce bouton de manière à ce qu'il fasse la hauteur de la case dans laquelle il est inséré et qu'il soit centré horizontalement avec une largeur fixe égale à 100 pixels. Renommer le bouton en *buttonEnvoyer* et changer son texte (*Content*) en *Envoyer*.

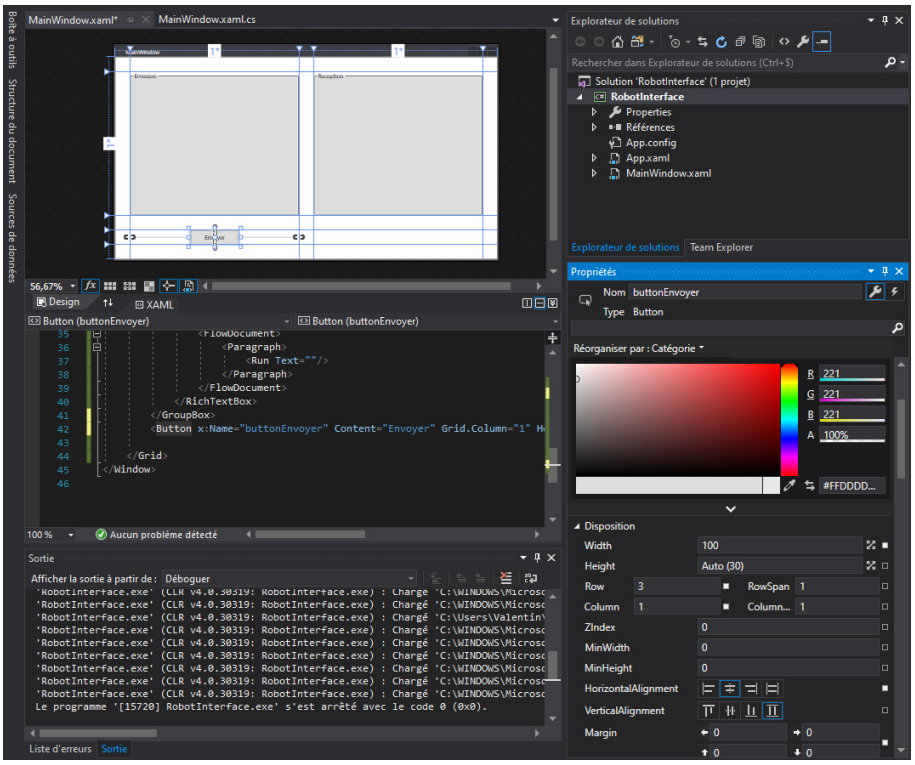

FIGURE 5 – Insertion du bouton d'envoi

⟹ Une fois les propriétés du bouton d'envoi définies, vous pouvez lui faire déclencher des actions. Pour cela, dans les propriétés, cliquez sur l'icône en forme d'éclair. Vous ouvrez un autre onglet qui présente les évènements associés à l'objet bouton. Parmi ces évènements figure l'évènement *Click*. Double-cliquez dans la case vide située à droite de l'évènement *Click*. Une fenêtre dénommée *MainWindow.xaml.cs* a du s'ouvrir dans la fenêtre principale de *Visual Studio* comme indiqué à la figure 6. Cette fenêtre montre le code associé à l'écran graphique XAML *MainWindow* sur lequel nous avons travaillé jusqu'ici : ce code est dénommé *Code Behind* de la fenêtre.

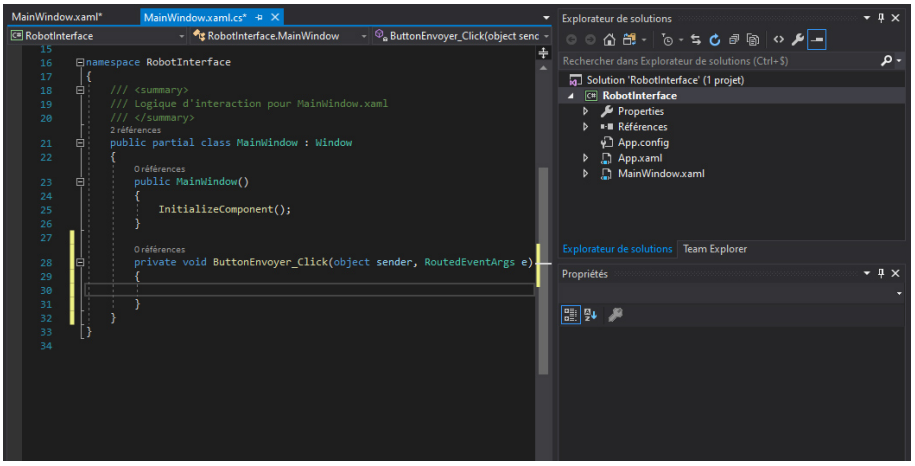

FIGURE 6 – Code Behind du bouton d'envoi

Dans le *Code Behind*, une fonction *ButtonEnvoyer_Click* a été automatiquement créée. Cette fonction est exécutée chaque fois que l'utilisateur clique sur le bouton envoyer. Pour illustrer son fonctionnement, ajouter dans cette fonction le code suivant :

```
private void ButtonEnvoyer_Click(object sender, RoutedEventArgs e)
```

```
        {
            buttonEnvoyer.Background = Brushes.RoyalBlue;
        }
```

⟹ Exécutez le programme et cliquez sur le bouton *Envoyer*. Que constatez-vous ? Que se passe-t-il si l'on appuie plusieurs fois sur le bouton *Envoyer* ? Commentez.

⟹ Modifier le code à votre convenance de manière à ce que la couleur de fond du bouton *Envoyer* évolue alternativement de la couleur *RoyalBlue* à *Beige* à chaque click. Valider avec le professeur.

Vous avez à présent fait connaissance avec les objets, les propriétés des objets et les évènements qui leurs sont associés.

⟹ A présent, nous souhaitons simuler l'envoi d'un message de la *TextBox* d'émission vers la *TextBox* de réception. Pour cela dans la fonction *buttonEnvoyer_Click*, rajouter du code permettant de récupérer le texte de la TextBox d'émission pour le placer dans la TextBox de réception, précédé d'un retour à la ligne et de la mention : "Recu : ". La *TextBox* d'emission doit également être vidée.

Le comportement doit être proche de celui de la figure 7 où 4 messages ont été envoyés successivement. Valider avec le professeur.

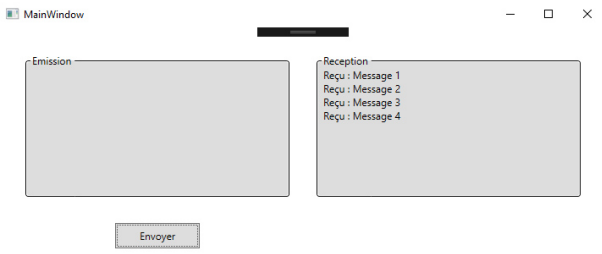

FIGURE 7 – Exemple d'exécution du simulateur de messagerie

⟹ Le comportement de l'ensemble simule presque un service de messagerie instantanée, à ceci près que dans ce type de service les envois sont réalisés par appui sur la touche *Entrée*. Il faut pour cela gérer les évènements clavier dans la *TextBox* d'émission, en vérifiant que la source de l'appui soit le bouton *Entrée*. Rajoutez à la TextBox d'émission un *évènement (éclair) KeyUp*.

La fonction (ou méthode) associée à cet évènement et dénommée *TextBoxEmission_KeyUp*, possède un argument de type *KeyEventArgs*, qui permet de savoir si la touche appuyée est la touche *Entrée* en utilisant par exemple le code suivant :

```
if (e.Key == Key.Enter)
{
    SendMessage();
}
```

⟹ Implanter le code permettant l'envoi des messages sur appui sur la touche Entrée, ou sur le bouton d'envoi, en évitant les duplications de code. Valider le fonctionnement de votre simulateur de messagerie instantanée avec le professeur.

2 Et si on passait aux choses sérieuses ?

Vous avez appris comment coder en *C#* avec une interface graphique sur un exemple simple en *WPF*.

A présent vous allez directement vous attaquer à la conception d'un robot complet couplant *C#* sur PC pour la partir pilotage et capteurs de haut niveau, et *C* sur micro-contrôleur pour la partie embarquée temps réel.

Ce couplage PC - micro-contrôleur est indispensable en robotique car les atouts et inconvénients des PC et des micro-contrôleurs sont très différents. Voici un récapitulatif à ce sujet, si vous avez des questions complémentaires, demandez au professeur :

- Les microcontrôleurs :
  - Atouts : les micro-contrôleurs donnent un accès total à des périphériques électroniques (timers, UART, PWM) implanté en silicium, donc très rapides, sans latence (retard) et complètement paramétrables. Un micro-contrôleur peut donc servir à implanter des opérations dont l'exécution doit se faire un intervalle parfaitement régulier et en temps réel.
  - Inconvénients : les micro-contrôleurs ont une mémoire vive (RAM) en général très limitée (quelques kilo-octets), ce qui rend très complexe leur usage pour des applications gourmandes en RAM telles que du traitement d'image issues de caméras. De plus, compte-tenu du très bas niveau de programmation, il est difficile d'implanter des fonctionnalités très complexes sur un micro-contrôleur.

- Les PC :
  - Atouts : Les ordinateurs permettent d'effectuer efficacement un grand nombre de tâches de haut niveau et utilisant beaucoup de mémoire (traitement d'image, lidar...), sans que nous ayons à nous soucier de comment les faire cohabiter. De plus, l'implantation d'interfaces graphiques est très simple sur un PC, par exemple en *C#*
  - Inconvénients : les ordinateurs sont gérés par un Operating System (OS tel que Linux ou Windows). Pour faire simple, c'est une couche qui est placée entre les ressources électroniques et le code utilisateur de haut niveau. Elle permet une gestion des tâches sur l'ordinateur, qui vous permet d'utiliser plusieurs logiciels à la fois (plusieurs tâches) sans vous préoccuper de l'accès aux ressources hardware. Les tâches effectuées sur l'ordinateur ne se font pas en parallèle, mais l'une après l'autre de manière séquentielle, l'ordonnancement étant géré par un séquenceur de tâches. La conséquence de cela est que l'on ne peut garantir le moment exact d'exécution d'une tâche, ce qui signifie que la précision temporelle de l'instant de lancement des tâches est assez mauvaise. Par exemple, si l'on utilise un timer logiciel à 50Hz en *C#*, le temps mesuré entre deux ticks pourra varier entre $15ms$ et $30ms$. C'est extrêmement gênant pour des opérations nécessitant une grande précision temporelle, telles que calcul d'une vitesse par dérivation de la position par exemple.

Le but de votre travail est de construire un robot complet dont on pourra faire évoluer les fonctionnalités au fur et à mesure. Le but final est d'arriver à avoir un robot capable de se déplacer en mode autonome ou piloté, de détecter des objets dans son environnement et d'effectuer des actions avec ces objets. Le schéma synoptique décrivant le robot est présenté à la figure 8.

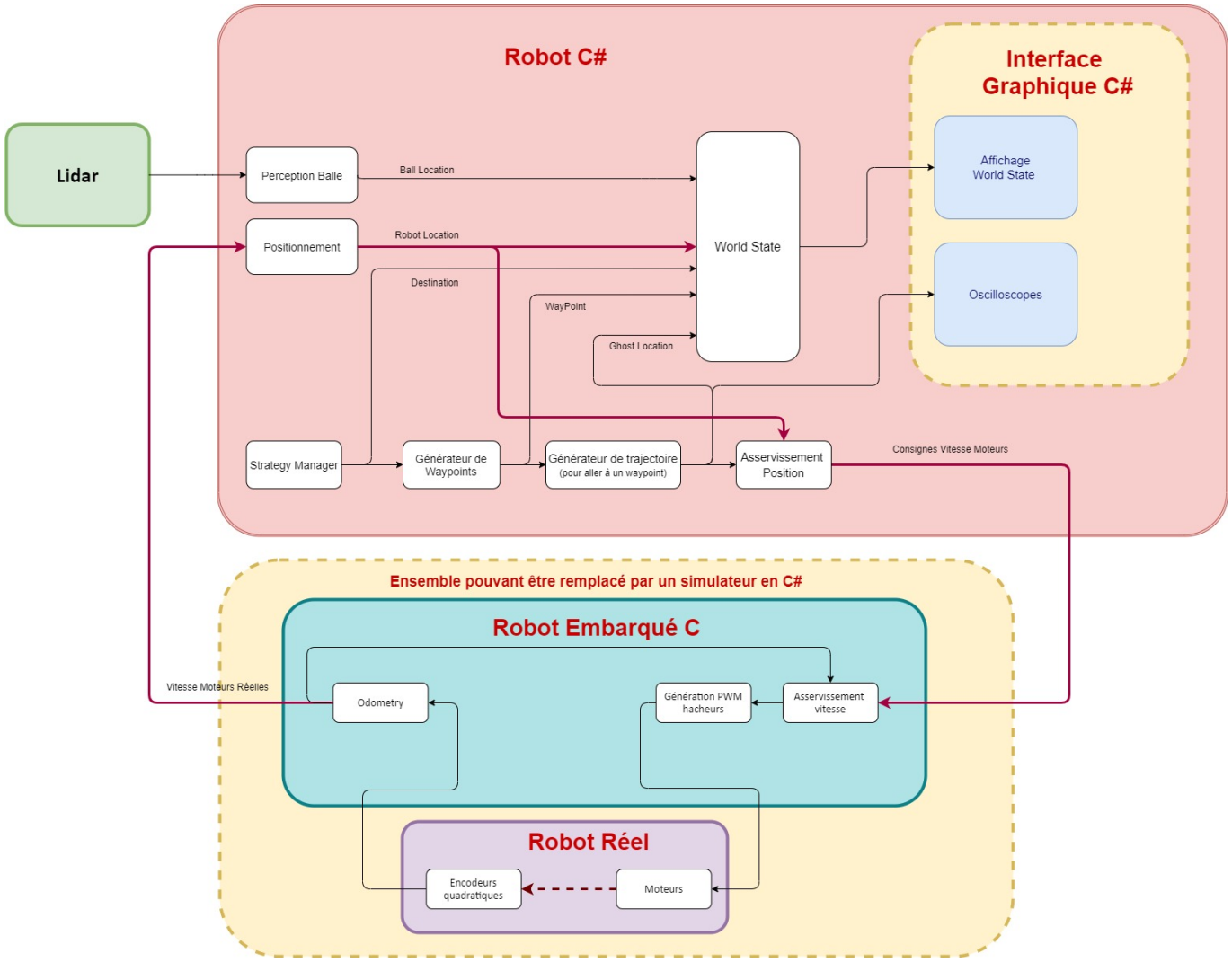

FIGURE 8 – Schéma synoptique du robot

Les différentes parties seront abordées séparément. Nous commencerons par l'implantation du code du robot en *C#*, en utilisant à la place du robot réel et son électronique embarquée un simulateur (fourni) de manière à ce que vous puissiez faire le projet juste avec un PC équipé de *Visual Studio*.

2.1 Création du projet Robot

Vous allez donc développer depuis la première ligne de code une nouvelle application robot correspondant au schéma représenté à la figure 8.

Elle sera basée sur une *application console* qui gère le robot. Cette application ne fait rien, à part charger des modules indépendants qui sont liés entre eux ensuite en communiquant par des évènements (nous y reviendrons en détails). Les évènement correspondent aux flèches liant les modules sur le schéma synoptique.

⟹ Commencez par créer une application *Console (.NET Framework)* comme indiqué à la figure 9. **Attention à ne pas vous tromper dans le type de projet** sans quoi il faudra tout refaire !.

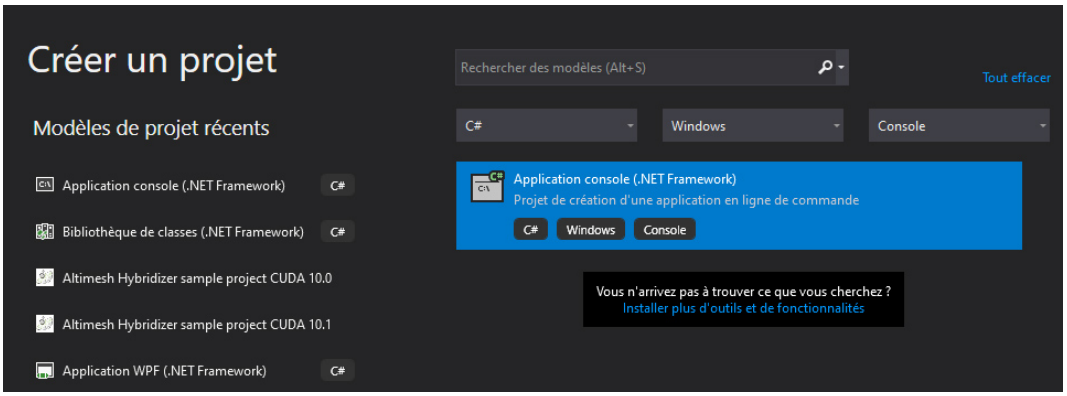

FIGURE 9 – Choix du projet Console

Dans l'écran de configuration de projet (cf. Fig. 10), donnez lui comme nom : *Robot* et placez là dans un répertoire nommé *CodeRobot* ou vous placerez tous les projets utiles au robot. Dans *Framework*, spécifiez *.NET Framework 4.8*. Le cas échéant, téléchargez le et installez-le si ce n'est pas fait sur votre ordinateur.

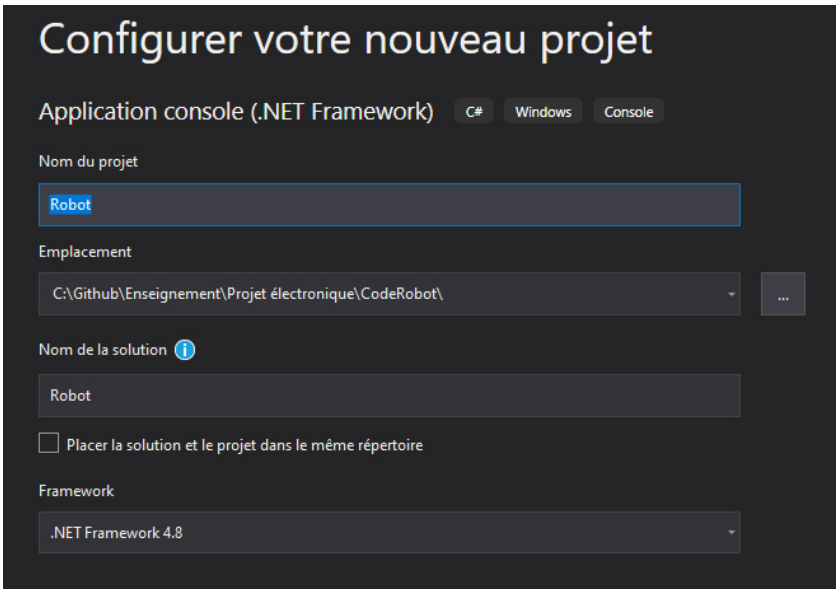

FIGURE 10 – Configuration du projet Console

Le projet créé est vide pour l'instant, et il n'a pas d'interface graphique. Si vous le compilez et que vous l'exécutez (en appuyant sur *F5*), vous devriez voir s'ouvrir une console (terminal texte noir) avant qu'elle ne se ferme toute seule immédiatement après.

2.2 Interface Graphique de supervision et de pilotage du robot

2.2.1 Création de l'interface graphique et intégration à l'application *Robot*

A présent vous allez créer le module d'interface graphique du robot. Elle est indispensable pour nos applications en robotique car :

— Elle permet d'afficher en temps réel les variables internes du robot telles que les distances mesurées par des capteurs, l'étape en cours dans une machine à état, la perception de l'environnement par le robot, la vitesse des moteurs...

— Elle permet d'envoyer des ordres au robot à distance depuis une interface homme machine (IHM) telle qu'une manette de jeu ou un clavier.

Toutefois en mode autonome, typiquement durant les matchs des coupes de robotique, l'usage de l'interface graphique est inutile, voir nuisible car :

— Il consomme des ressources et peut limiter la vitesse d'exécution du code.
— Il peut tout simplement planter et interrompre le fonctionnement du robot pour un simple bug graphique.

En conséquence, l'usage d'une interface graphique est absolument nécessaire pour mettre au point un robot, mais il serait bien de pouvoir ne pas l'utiliser si on le souhaite.

Afin de pouvoir s'en servir uniquement quand on le souhaite, nous allons créer cette interface comme un projet à part dans le robot qui ne servira qu'à l'affichage graphique. Si ce projet est appelé par l'application Console Robot, l'interface sera démarrée, sinon elle ne le sera pas mais le robot fonctionnera de la même manière.

⟹ Dans solution Robot, ajoutez un nouveau projet de type *Console (.NET WPF)* comme indiqué à la figure 11. **Attention à ne pas vous tromper dans le type de projet** sans quoi il faudra tout refaire !.

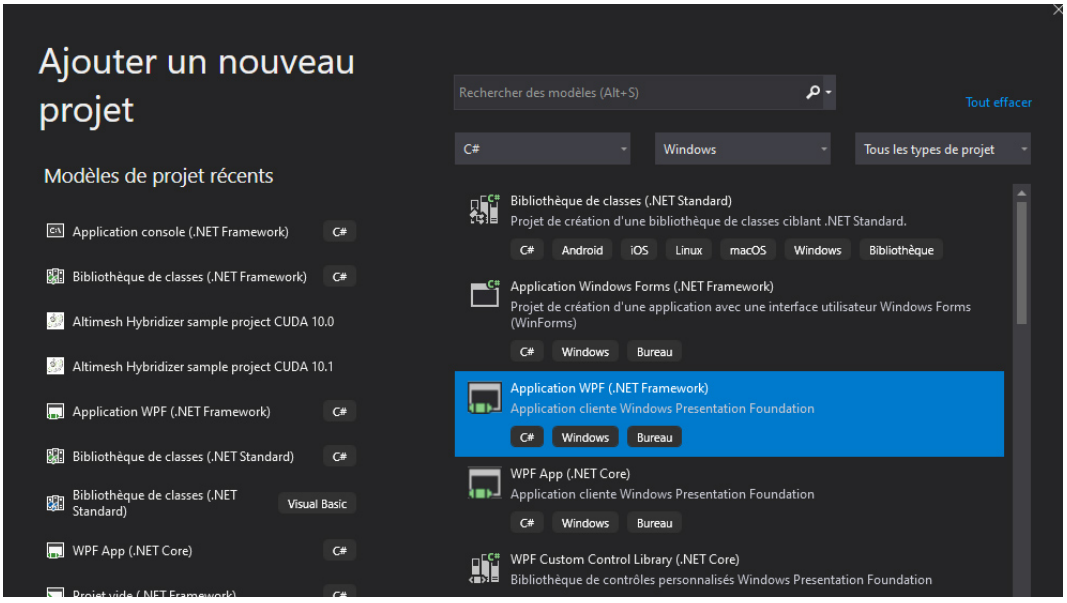

FIGURE 11 – Choix du projet WPF .NET Framework

Dans l'écran de configuration de projet (cf. Fig. 12), donnez lui comme nom : *WpfRobotInterface* et placez le dans un répertoire nommé *CodeRobot* où sont placés tous les projets. Dans *Framework*, spécifiez *.NET Framework 4.8*.

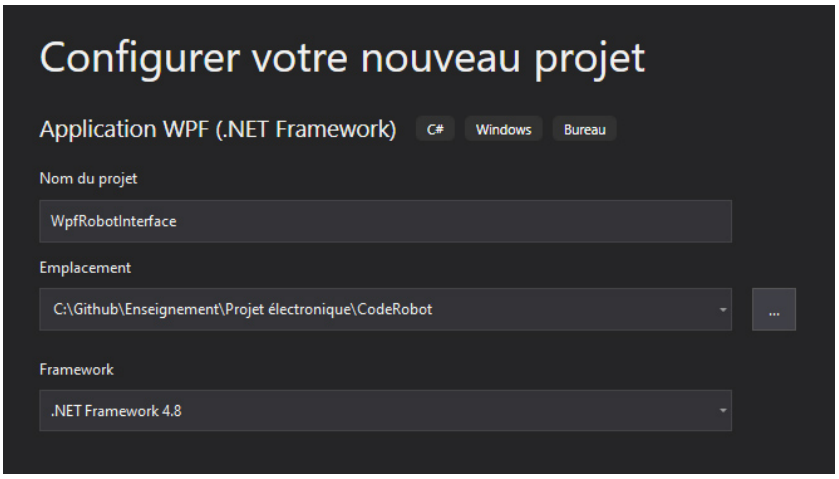

FIGURE 12 – Configuration du projet WPF .NET Framework

Le projet est une interface graphique vide pour l'instant. Il n'est pas pour l'instant appelé par le projet Console.

⟹ Nous allons à présent configurer le lancement de l'interface depuis le projet *Robot*. Commencez par ajouter une référence au projet WpfRobotInterface depuis l'onglet références du projet Robot comme indiqué à la figure 13. La référence à ajouter se trouve dans l'item *Projets/Solution*.

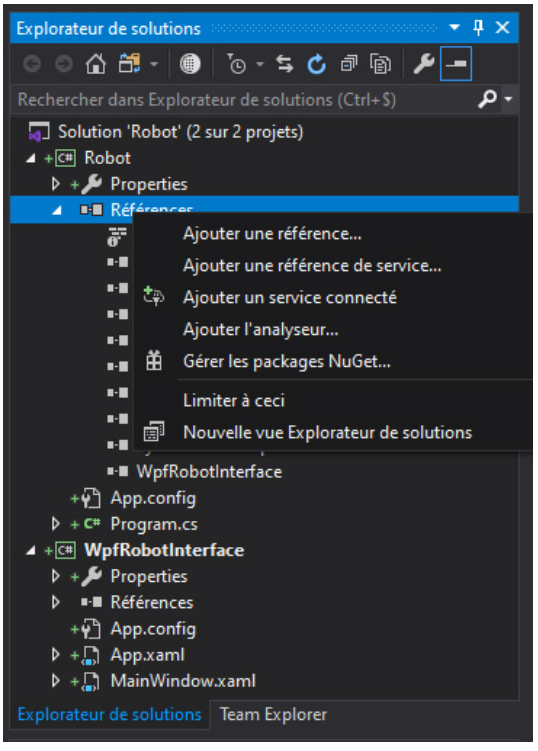

FIGURE 13 – Ajout d'une référence au projet Robot

⟹ Le nom du fichier xaml généré est *MainWindow*. Ce nom n'est pas très pertinent, vous allez le renommer. Attention à bien suivre la procédure, il est facile d'oublier des choses. Commencez par renommer *MainWindow.xaml* en *RobotInterface.xaml*. Dans le fichier xaml (interface graphique), renommez *MainWindow* en *RobotInterface* à la première ligne dans le champ x :Class (il faut voir la partie XAML du designer graphique). Enfin

dans le fichier *RobotInterface.xaml.cs*, Renommer toutes les occurences de *MainWindow* en *RobotInterface*. Valider les modifications en tentant de compiler WpfRobotInterface en cliquant droit sur le projet et en appuyant sur *Générer*. Si la génération réussi ça doit être bon.

⟹ A présent, vous allez appeler l'interface graphique depuis l'application Console *Robot*. Pour cela remplacez le code présent dans la classe Program par le code suivant :

```
class Program
{
        static bool usingRobotInterface = true;
        static RobotInterface interfaceRobot;
        static void Main(string[] args)
        {
                if (usingRobotInterface)
                StartRobotInterface();
        }

        static Thread t1;
        static void StartRobotInterface()
        {
                t1 = new Thread(() =>
                {
                        interfaceRobot = new RobotInterface();
                        interfaceRobot.ShowDialog();
                });
                t1.SetApartmentState(ApartmentState.STA);
                t1.Start();
        }
}
```

Normalement le type Thread ne doit pas être reconnu. Il est souligné en rouge dans le code comme indiqué à la figure 14, pour y remédier utilisez l'outil de correction interractive de Visual Studio : en plaçant votre souris sur le mot souligné en rouge, un menu contextuel apparait, vous devez choisir *Afficher les corrections éventuelles*, et sélectionner ajouter *using System.Threading ;* comme indiqué à la figure 14. Le *using System.Threading* nécessaire est alors ajoutée aux références automatiquement.

Le type *RobotInterface* ne doit pas être reconnu non plus, faire de même que pour *Thread*. Vous aurez ensuite le même problème avec *ShowDialog*, faire de même trois fois de suite (trois bibliothèques différentes seront ajoutées).

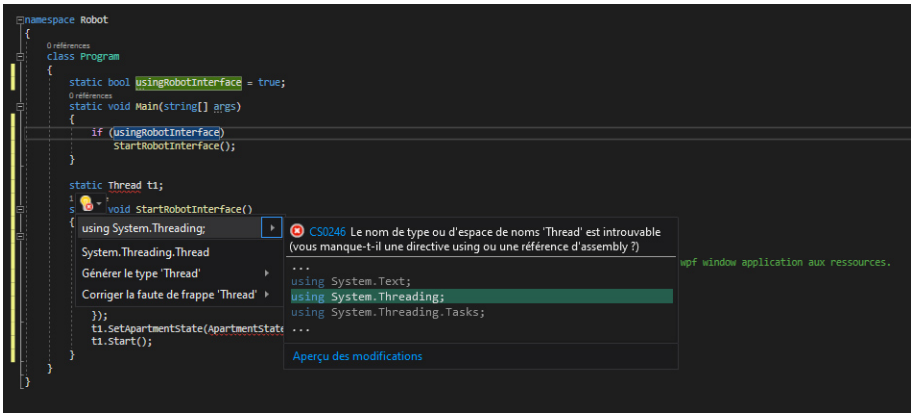

FIGURE 14 – Ajout interactif de using System.Threading ; au projet

⟹ Normalement, si vous exécutez votre code (*F5*), la console devrait lancer la fenêtre graphique (pour l'instant vide). Lorsque vous fermez cette fenêtre graphique, le console doit se fermer également.

Quelques explications sur le code étrange que vous avez ajouté : l'interface graphique doit impérativement être instanciée dans une tâche indépendante, on appelle cela un *Thread*. La règle est de les déclarer en *static* dans l'application console (comme tous les objets instanciés dans cette *Robot*). A l'appel de *StartRobotInterface* ; un *Thread* (une tâche) dénommé *t1* est donc créé. Ce thread, quand il sera démarré par *t1.Start()*, lancera le code :

```
{
        interfaceRobot = new RobotInterface();
        interfaceRobot.ShowDialog();
}
```

Ce code a été passé en paramètre du Thread en utilisant une *lambda expression*. Il n'est pas nécessaire que vous soyez capable de comprendre l'intérêt de ces fonctions pour l'instant (c'est de la programmation avancée), mais les plus curieux pourront chercher sur Internet.

$\Longrightarrow$ Arrivé à ce point, vous noterez que si vous passez le booléen *usingRobotInterface* à *false* dans l'initialisation du *Program*, alors l'interface graphique n'est pas lancée, c'est ce qui nous permettra de choisir entre l'usage de l'interface ou pas.

2.2.2 Et si on faisait quelque chose avec l'application graphique ?

Vous l'avez noté, notre application graphique reste désespérément vide. Il est à présent temps de s'en servir pour y afficher des informations utiles à la mise au point et à la supervision de notre robot. En particulier, il est nécessaire de pouvoir visualiser la position du robot, sa destination, la carte des points vus par un Lidar, le tout sur un terrain qui sera le terrain de la Coupe de Robotique des IUT. Il est également nécessaire de visualiser des variables internes au robot telles que la vitesse réelles des encodeurs, les consignes de vitesse des moteurs, et toute autre variable temporelle qui servira à régler les asservissements ultérieurement.

Après avoir fait une première interface graphique dans un projet précédent, vous êtes capables d'ajouter des grilles, des labels, des boutons à votre interface. Nous ne reviendrons pas sur ces points dans cette partie les considérant comme acquis.

En revanche, il ressort de l'analyse précédente qu'il faut pouvoir ajouter un composant graphique permettant l'affichage de l'état du monde environnant (*World State*) et des composants graphiques faisant office d'oscilloscopes. La conception des ces briques est un peu complexe pour débuter, elles vous sont donc fournies. Vous pouvez les télécharger depuis la page du projet.