PROJET IMA4 P38 :

INTERFACE GRAPHIQUE ROBOTINO 2 UPGRADÉ

François BRASSART & Jérôme HAON

Informatique Microélectronique Automatique 4

Année universitaire 2018/2019

Remerciements

Nous remercions notre encadrant principal Vincent Coelen pour son soutien et sa disponibilité tout au long du projet.

Son aide et ses conseils ont été des éléments clés qui nous ont permis de mener à bien ce projet.

Nous remercions également Xavier Redon, pour les réponses aux diverses questions et son aide durant ce projet.

Sommaire

Introduction 1
1. Analyse du projet 2
1.1. Positionnement par rapport à l'existant 2
1.1.1. Robotino 2 2
1.1.2. Robotino 3 2
1.2. Présentation du nouveau robotino 3
1.3. Scénario d'usage 4
2. Réalisation du projet 5
2.1. Présentation de Web Toolkit 5
2.2. Menu principal 5
2.3. Utilisation du CSS 6
2.4. Navigation entre les menus 7
2.5. Affichage de l'état des capteurs 8
2.6. Connexion réseau 16
2.7. Lancement programmes de démonstration 18
Réflexions, conclusion 21

Introduction

Durant notre deuxième année du parcours des écoles d'ingénieur Polytech (PEIP), nous avons eu l'occasion d'utiliser et de programmer des robotinos lors d'un bureau d'étude afin de découvrir la spécialité IMA.

Les robotinos sont des robots mobiles, produits par l'entreprise Festo, à entrainement omnidirectionnel ce qui leur permet de se déplacer dans toutes les directions et de tourner sur eux-mêmes. Ils sont dotés de différents capteurs (infrarouges, optiques, inductifs...) ainsi que d'une Webcam et d'un gyroscope. Ils peuvent être contrôlés et programmés sur un PC via une connexion Wifi ou directement grâce à l'interface Homme-Machine (IHM) intégrée.

Les robotinos sont utilisés depuis une dizaine d'années en salle de TP et commençaient à vieillir. L'objectif de notre projet de quatrième année est de moderniser ces robots afin de les remettre au gout du jour. Les robotinos bénéficieront d'un nouvel ordinateur embarqué, d'un nouveau logiciel, de nouvelles batteries et d'un écran tactile 7 pouces.

La modernisation des robotinos a déjà été commencée avant le début de notre projet. Les nouveaux composants ont déjà été choisis et l'installation est en cours. Le but de notre projet est de concevoir la nouvelle interface Homme-Machine. Il s'agira d'une interface graphique directement accessible via l'écran tactile, programmée en C++ grâce à Web Toolkit qui permettra entre autres de :

- Afficher l'état et les données des différents capteurs (camera, laser, sharps, bumper) afin de faciliter la programmation et le débogage des programmes.
- Lancer les programmes de démonstration que nous aurons préalablement programmés.
- Configurer le robotino au réseau : pouvoir modifier facilement la configuration réseau du robot (se connecter à un réseau wifi)

L'interface graphique étant réalisé avec Web Toolkit, qui permet de créer des interfaces graphiques pour le web, elle sera également accessible sur des ordinateurs ou smartphones connectés au même réseau que le robotino. Ainsi, nous pourrons visualiser l'état des capteurs, lancer des programmes de démonstration et configurer le réseau depuis un smartphone ou un ordinateur de la salle AIP.

Cette interface graphique devra évidemment être fonctionnelle et agréable à utiliser.

1. Analyse du projet

1.1. Positionnement par rapport à l'existant

Les robotinos utilisés depuis 10 ans en salle de TP sont des robotinos de deuxième génération. Il existe actuellement des robotinos de troisième génération. Nos 2 concurrents sont donc le robotino 2 et le Robotino 3.

1.1.1. Robotino 2

Le robotino 2 est équipé d'un ordinateur embarqué AMD Geode, avec l'OS Ubuntu 9.04.

Il dispose également d'une IHM sur son châssis composé d'un petit écran et de plusieurs boutons. Elle permet de :

- Sélectionner la langue,
- Afficher l'état des batteries
- Sélectionner les adresses du réseau
- Sélectionner des programmes de démonstrations préenregistrés.

Le robotino 2 est devenu vétuste, Festo cherche à s'en débarrasser et vend maintenant uniquement des robotino de 3eme génération.

1.1.2. Robotino 3

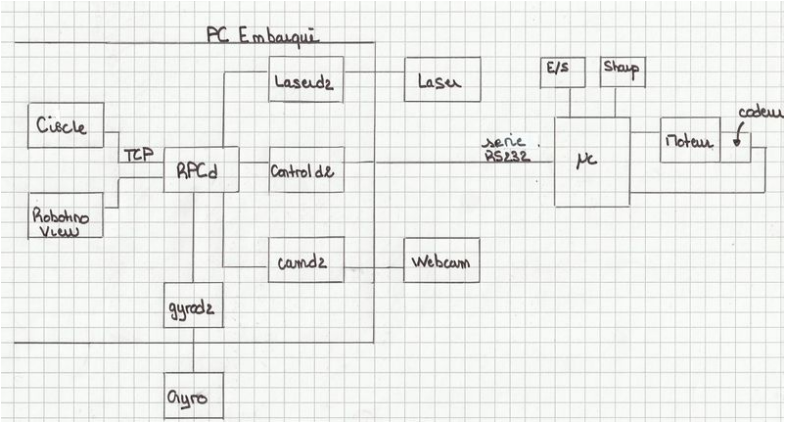
Le robotino 3, plus récent, est équipé d'un ordinateur embarqué Intel Core i5 ou Intel Atom, avec l'OS Ubuntu 12.04.

Il ne dispose pas d'IHM intégrée, il doit être contrôlé à distance par un PC.

1.2. Présentation du nouveau robotino

Notre robotino "amélioré" sera équipé d'un ordinateur embarqué Intel NUC, avec l'OS Ubuntu 18.04. Il disposera d'une IHM intégrée notamment grâce à un écran tactile 7 pouces, dont nous allons concevoir l'interface graphique avec la librairie Web Toolkit.

Le nouvel ordinateur embarqué est relié au microcontrôleur du Robotino via une liaison série RS232. Lorsqu'un programme de démonstration est lancé (directement depuis l'ordinateur embarqué ou via le logiciel RobotinoView depuis un ordinateur), les commandes sont envoyées en TCP au RPCd., qui est chargé d'envoyer les commandes au bon destinataire. La Webcam ou le laser sont reliés à l'ordinateur embarqué. Les autres capteurs et actionneurs (moteurs, roues...) sont reliés au microcontrôleur du Robotino.



Le robotino offre une plateforme de programmation très évoluée. En effet, il peut-être programmé avec de nombreux langages : RobotinoView (logiciel graphique), C, C++, JAVA, Matlab, Labview, Visual Basic…

1.3. Scénario d’usage

Samedi 2 février 2019, Journée Portes Ouvertes à Polytech Lille.

Pierre, étudiant en IMA est chargé de faire la démonstration de l'utilisation des robotinos sur le terrain de Hockey/Football de la salle C305. Dès que tous les futurs étudiants et leurs parents sont rentrés dans la salle, il n'a qu'à choisir en cliquant sur l'écran tactile d'un robotino le programme qu'il désire dans l'onglet "programme de démonstration". Il peut ainsi commencer sa présentation pendant que le robot suivra une ligne, contournera un obstacle ou fera demi-tour quand il rencontrera un mur.

Vendredi 25 Janvier 2019, Bureau d'études IMA.

Léna et Clément, étudiants en PEIP2, ont choisi le bureau d'études IMA afin de découvrir cette spécialité et finaliser leur orientation. Il consiste en la commande d'un robotino. La première mission est de le faire suivre une ligne. Léna et Clément ont beaucoup de mal à comprendre le fonctionnement de ce Robot. Clément a mis des sur-chaussures pour pouvoir marcher sur le terrain pendant que Lina commande manuellement le robot depuis le PC. En cliquant sur l'onglet "Visualisation état des capteurs", Clément peut directement voir sur l'écran du robot l'état des capteurs, et ainsi comprendre leur fonctionnement en fonction de la position du robot et de son environnement.

2. Réalisation du projet

2.1. Présentation de Web Toolkit

Web Toolkit (WT) est un Framework, une bibliothèque logicielle orientée objet et développés en C++. Un Framework (ou infrastructure logicielle en français) désigne un ensemble d'outils et de composants logiciels à la base d'un logiciel ou d'une application. C'est le Framework qui établit les fondations d'un logiciel ou son squelette applicatif. Tous les développeurs qui l'utilisent peuvent l'enrichir pour en améliorer l'utilisation. L'objectif du Framework est de simplifier et d'uniformiser le travail des développeurs. Il fonctionne comme un cadre ou un patron, mais son maniement suppose d'avoir déjà un profil expérimenté.

WT permet de programmer des interfaces graphiques pour le web : il offre des composants d'interface graphique appelés widgets, d'accès aux données, de connexions réseaux, de gestion des fils d'exécution… WT utilise également un système de signal piloté par événement. WT est semblable à Qt, Framework plus connu permettant de programmer des interfaces graphiques. Cependant, Qt ne permet pas d'implémenter des interfaces pour le web.

Web Toolkit dispose d'une documentation qui explique le fonctionnement de certaines classes. Cependant, peu de tutoriels clairs existent pour expliquer son utilisation.

L'intérêt pour notre projet d'utiliser Web Toolkit (par rapport à Qt) est d'avoir une interface graphique pour le web. Ainsi, nous pourrons visualiser notre interface graphique sur l'écran tactile du robotino, mais également sur un ordinateur ou smartphone connecté au même réseau que celui du robotino.

2.2. Menu principal

Lorsqu'on lance l'interface graphique, apparait le menu principal. Celui-ci est composé d'un bandeau et de trois boutons permettant d'accéder aux différentes fonctionnalités de l'interface : connexion réseau, état des capteurs, lancement des programmes de démonstration.
Le bandeau, toujours visible, affiche en continu les informations importantes : adresse IP du robotino, nom du programme en cours (s'il y a), le nom du réseau wifi auquel le robotino est connecté ainsi que la qualité de la connexion wifi.

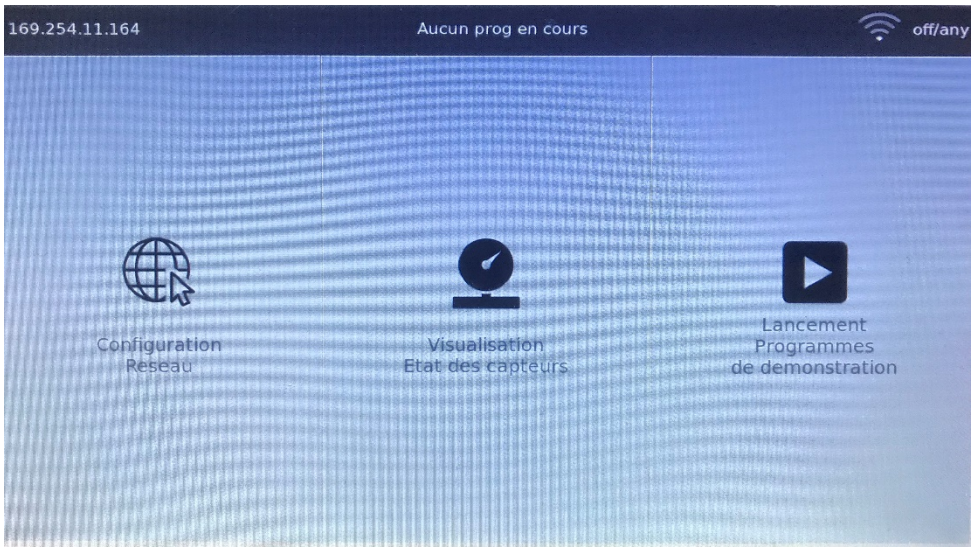


Figure 1 Menu Principal de l'interface

Notre projet est organisé de la sorte :

- bandeau.C : code qui définit la classe « bandeau ».
- interface.C : code qui définit la classe « interface » qui correspond au « corps » de l'interface : la partie qui affichera les différents menus.
- main.C : code qui crée l'interface en créant une instance de la classe bandeau et une instance de la classe interface et se connecte au robotino.

Les classes bandeau et interface sont des sous-classes de *WContainerWidget*. Un *WContainerWidget* permet de séparer les widgets dans des parties différentes de l'écran. Nous pouvons donc facilement ajouter des widgets dans ces classes.

Pour le menu principal, nous avons choisi d'ajouter dans le *WContainerWidget* trois boutons. Ces boutons permettent de lancer les trois fonctionnalités principales de notre interface. Un bouton est créé en ajoutant une instance de *WPushButton*. Pour l'ajouter dans un *WContainer* particulier, nous utilisons la méthode *addWidget*.

```
WPushButton *btn = container->addWidget(std::make_unique<Wt::WPushButton>("Etat des
capteurs"));
```

Pour le bandeau, nous avons opté pour un *WHBoxLayout*. Ce type de widget permet d'organiser horizontalement des widgets, dans notre cas du texte et des images. Cela nous permet de régler correctement l'espacement et le centrage des informations contenues dans le bandeau.

Pour ajouter un texte, nous utilisons le widget *WText*. Pour ajouter une image, nous utilisons le widget *WImage*.

2.3. Utilisation du CSS

Pour le design de l'interface, nous utilisons une feuille de style CSS. Ainsi, nous pouvons mettre en forme les widgets : modifier la taille, la couleur, centrer…

Pour ce faire, nous utilisons la fonction useStyleSheet qui permet d'indiquer le fichier CSS à utiliser.

```
useStyleSheet("stylesheet.css")
```

Nous avons créé un répertoire « ressources » qui contient la feuille de style et les images nécessaires à l'interface. Lors de la compilation, nous indiquons où se trouvent les ressources grâce à l'option *docroot*. Nous indiquons également l'adresse que nous souhaitons utiliser ainsi que le port.

```
./main --docroot ./ressources --http-address 0.0.0.0 --http-port 8080
```

L'objectif étant de pouvoir utiliser l'interface sur l'écran tactile du robotino, mais également sur un ordinateur ou un smartphone, il est important que les différents widgets s'adaptent en fonction de la taille de l'écran. Pour cela, nous positionnons la hauteur et la largeur des différents widgets en fonction de la taille de l'écran utilisé.

Par exemple,

```
.btnCapteurs {
    height: 11vh;
    width:49vw;
}
```

Les boutons ayant pour classe CSS « btnCapteurs » auront une hauteur de 11% de la hauteur de l'écran et une largeur de 49% de la largeur de l'écran. Cela assure un confort visuel peu importe l'appareil utilisé.

2.4. Navigation entre les menus

Pour naviguer entre les différents menus, voici l'explication que nous avons trouvée sur le site de WT :

"Wt allows the application to define internal paths and handle changes of internal paths. Because a Wt application is not composed of pages, a URL does not define to particular page, but rather a particular application state"

Chaque fois que l'on veut changer ce qui est affiché à l'écran (lors de l'appui sur un bouton principalement), le "path" (l'URL) est modifié et un signal InternalPathChanged est envoyé. Nous définissons donc une fonction handlePathChanged() qui est appelée lorsque ce signal est reçu. Cette fonction supprime tous les widgets du WContainerWidget principal corps grâce à la méthode clear(), puis appelle une fonction qui va remplir ce container avec de nouveaux widgets.

```
void interface::handlePathChange()
{
   WApplication *app = WApplication::instance();
   corps->clear();
   if (app->internalPath() == <chemin1>)
     <fonction à appeler quand URL est chemin1>;
   else if (app->internalPath()== <chemin2>)
     <fonction à appeler quand URL est chemin1>;
}
```

Pour lier cette fonction au signal de changement d'url, nous récupérons une instance de l'application *WApplication* actuelle et connectons le signal à la fonction. *WApplication* représente une instance d'application pour une session donnée (une connexion à l'interface). La méthode *internalPath()* renvoie l'URL de l'application.

```
WApplication *app = WApplication::instance();
app->internalPathChanged().connect(this, &interface::handlePathChange);
```

Nous avons donc créé une fonction pour chaque fonctionnalité de notre interface, et associé une URL à chaque fonction.

Les boutons présents sur chaque menu permettent de naviguer entre les menus. Nous configurons les boutons pour changer l'URL vers celle de la fonctionnalité désirée grâce à la méthode *setLink*.

```
Btn->setLink(WLink(Wt::LinkType::InternalPath,<URL apres clic sur le bouton>));
```

Ainsi, lors d'un clic sur le bouton "Etat des Capteurs", l'URL devient localhost:8080/capteurs et notre fonction *handlePathChanged* supprime les widgets présents dans la partie du corps et appelle la fonction capteurs qui va afficher le menu d'état des capteurs. Cette fonction prend en paramètre le *WContainerWidget* corps, qui est un attribut de la classe corps.

2.5. Affichage de l'état des capteurs

Une fonction *capteurs* affiche des boutons pour chaque capteur disponible. Les boutons sont ancrés dans un *WContainerWidget* associé à un *WGridLayout.* Le *WGridLayout* permet de ranger les widgets dans une "grille", un tableau. Pour cela, il faut ajouter les widgets au layout en précisant la colonne et la ligne désirée dans le tableau. Ainsi, pour optimiser l'espace, nous avons créé 2 colonnes de 4 boutons qui permettent d'accéder aux états de 8 types de capteurs différents.

Nous avons également ajouté un bouton retour permettant de revenir au menu principal.

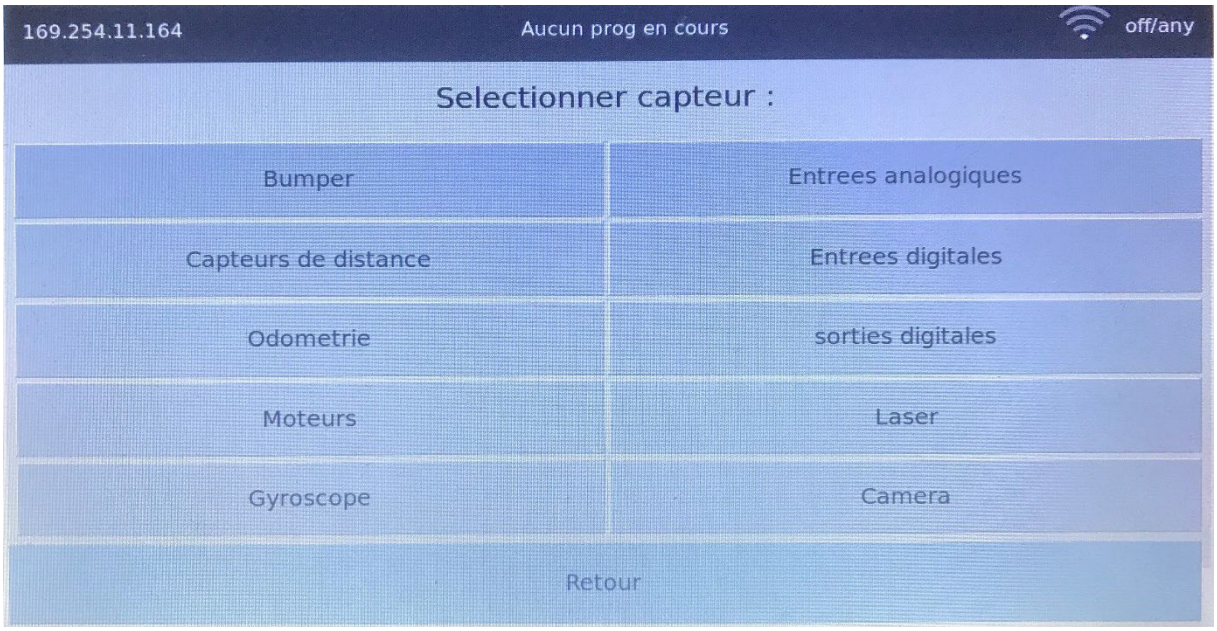


Figure 2 Sélection des capteurs

Chaque bouton change l'URL de l'application et la fonction *handlePathRequest* appelle la fonction correspondant au capteur. Nous avons donc créé une fonction pour chaque type de capteur. Chaque instance de capteur est créée en variable globale. Pour pouvoir conserver son état.

Nous souhaitons que les valeurs des capteurs s'actualisent en direct sans avoir à rafraichir la page. Pour cela, nous utilisons un timer. WT fournit un widget *WTimer* qui propage un signal à intervalle de temps régulier. Nous avons créé une fonction permettant d'initialiser et de démarrer un *WTimer* dans un container donné.

```
WTimer* addStartTimer(WContainerWidget *container, int time_ms){
  WTimer *timer=container->addChild(std::make_unique<Wt::WTimer>());
  timer->setInterval(std::chrono::milliseconds(time_ms));
  timer->start();
  return timer;
}
```

Pour récupérer le signal et exécuter des fonctions à intervalle de temps régulier, nous utilisons l'attribut timeout du timer.

```
mytimer->timeout().connect([=] {
        /*code à exécuter à intervalle de temps du timer*/ });
```

Nous avons choisi d'actualiser les données toutes les 10ms. Pour ne pas récupérer les données inutilement lorsque nous ne les affichons pas, le timer est arrêté quand on quitte le menu du capteur.

Etat du bumper

Le bumper est un capteur de collision situé sur tout le tour de la coque du Robotino. Quand il subit une pression, il renvoie un signal, cela permet de configurer le Robotino pour qu'il s'arrête au moindre choc. Pour créer une instance du bumper afin d'accéder à son état, nous mettons en variable globale :

```
rec::robotino::api2::Bumper bumper;
```

Nous ferons de même pour tous les autres capteurs.

Pour récupérer son état, la classe bumper dispose d'une méthode

```
bool rec::robotino::api2::Bumper::value()
```

Cette méthode renvoie

- *True* quand le Robotino heurte quelque chose (un mur par exemple)
- *False* sinon

Nous mettons cette fonction à l'intérieur d'un timer afin d'actualiser l'état du bumper toutes les 10ms.

Pour une meilleure visualisation, nous avons configuré le CSS afin de mettre le fond de l'interface en vert quand le bumper n'est pas touché, ou rouge quand le bumper subit un choc.

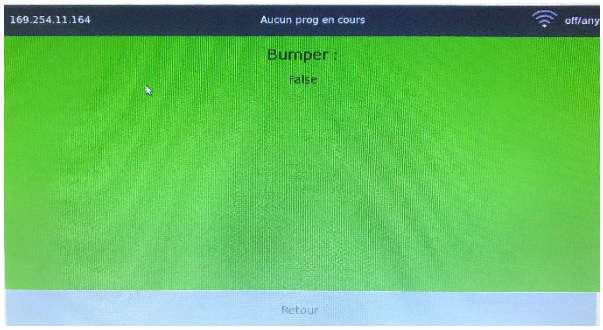


Figure 3 Etat du bumper : false

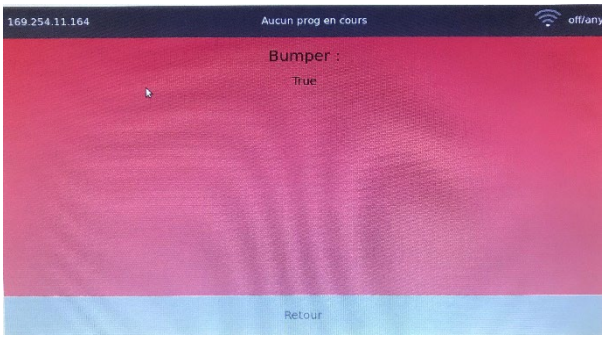


Figure 4 Etat du bumper : true

Capteur d'odométrie

L'odométrie fournit la position momentanée par rapport à la position de départ. Les codeurs des moteurs permettent de savoir combien de tours le moteur a effectué et donc calculer, à l'aide du diamètre de la roue, la distance parcourue. Cette méthode fournit localement des résultats relativement précis mais l'erreur croît fortement sur de longues distances.

La méthode suivante nous permet de récupérer la position en x, en y et l'orientation phi.

```
void rec::robotino::api2::Odometry::readings(double *x, double *y, double *phi,
unsigned int *sequence = 0)const
```

De la même manière que le Bumper nous actualisons l'affichage de l'odométrie à chaque fois que le Timer émet un signal.

Cependant nous pouvons constater que la fin du prototype de cette méthode de récupération de l'état de l'odométrie, il y a un const() à la fin. Le mot clé const signale que la fonction membre s'engage à ne pas modifier l'objet au départ duquel elle est appelée. Pour réussir à modifier l'état des variables, nous avons alors déclaré à nouveau les variables à l'intérieur du timer.

```
 mytimer->timeout().connect([=] {
  unsigned int sequence =0;
  double x,y,phi;
  odometry.readings(&x,&y,&phi,&sequence);
  stateX->setText("x = "+std::to_string(x)+" m");
  stateY->setText("y = "+std::to_string(y)+" m");
  statePhi->setText("phi = "+std::to_string(phi)+" °");
});
```

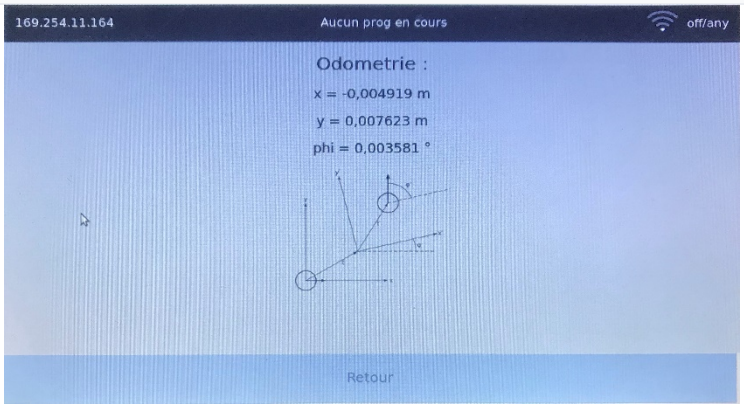


Figure 5 Etat de l'odométrie

Gyroscope

Le gyroscope est utilisé pour augmenter la précision de détermination des positions. Le gyroscope détermine la variation de l'orientation du Robotino. Dès que le système d'exploitation de Robotino détecte le gyroscope, il utilise son signal pour corriger les positions sur la base du système du système odométrique. L'odométrie est considérablement améliorée par le capteur gyroscopique.

La méthode suivante retourne la valeur de l'angle perçu par le gyroscope.

```
float rec::robotino::api2::Gyroscope::angle ()const
```

De la même manière que l'odométrie nous actualisons l'affichage du Gyroscope à chaque fois qu'un timer émet un signal.

Capteur de distance

Les capteurs de distance infrarouge déterminent la distance d'objets situés dans l'entourage du Robotino (entre 4cm et 30cm). Le Robotino possède au total neuf capteurs infrarouge, répartis par pas de 40° autour de la base. Chaque capteur de distance délivre une tension en volts, dont l'amplitude dépend de la distance par rapport à un objet réfléchissant.

Nous créons toujours dans le main une variable globale, qui cette fois correspondant aux capteurs de distance, que nous utilisons dans le programme du corps de l'interface. Pour afficher l'état des capteurs, la méthode suivante retourne un tableau contenant les valeurs de chaque capteur.

```
void rec::robotino::api2::DistanceSensorArray::headings (float *readings)const
```

Pour optimiser l'affichage, nous utilisons un *WGridLayout*, afin d'afficher les capteurs dans 2 colonnes différentes.

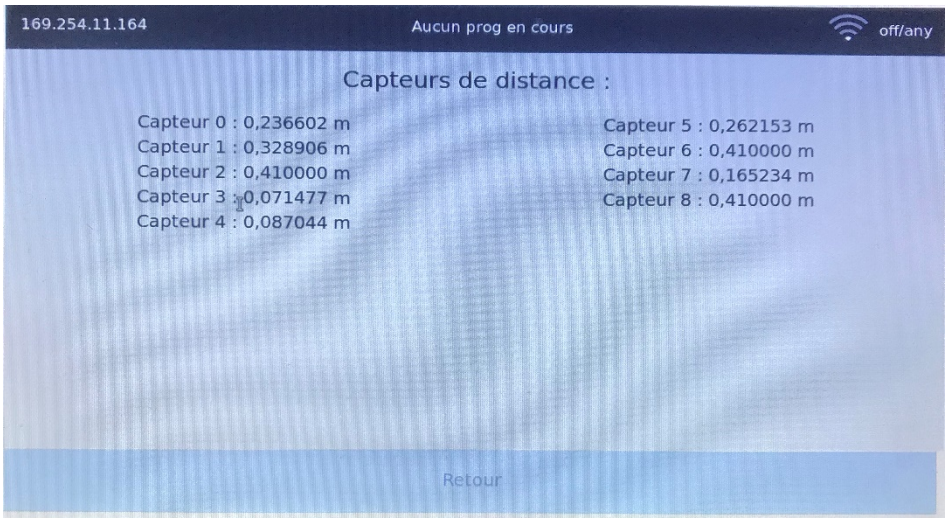


Figure 6 Etat des capteurs de distance

Vitesse des roues

Le Robotino dispose de trois moteurs indépendants entraînant trois roues omnidirectionnelles. Sur chaque moteur est monté un codeur incrémental mesurant la rotation du moteur. Les trois roues permettent de faire évoluer le système dans toutes les directions et de faire tourner le Robotino sur place dans une position donnée. Nous créons donc dans le main une variable globale, correspondant à la vitesse des roues, que nous utilisons dans le programme du corps de l'interface. Nous utilisons la méthode suivante qui renvoie la vitesse de chaque roue dans un tableau.

```
void rec::robotino::api2::MotorArray::actualVelocities (float* readings)const()
```

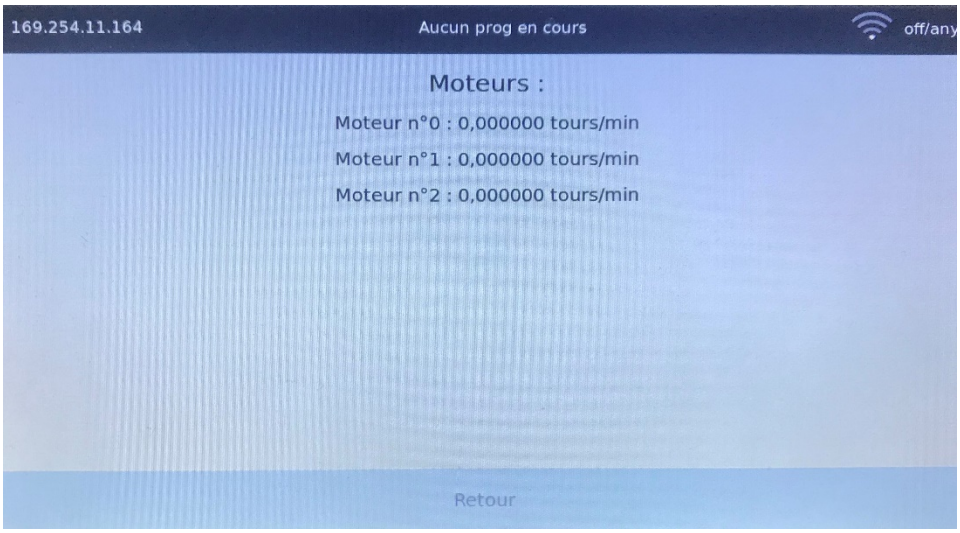


Figure 7 Etat des moteurs

Capteur laser

Le capteur Laser, celui-ci n'est pas sur le Robotino, il est optionnel. Il permet de maîtriser les défis tels que la cartographie, la localisation et la navigation.

Nous ajoutons donc dans le main une variable globale, correspondant au capteur laser, que nous utiliserons dans le programme du corps de l'interface. Pour afficher la valeur du capteur laser dans le menu, nous utilisons la fonction suivante :

```
LaserRangeFinderReadings rec::robotino::api2::LaserRangeFinder::readings ()const
```

Comme pour les fonctions précédentes, pour afficher les valeurs du capteur laser en instantanée avec un timer.

Entrée/Sortie

L'interface d'Entrées/Sorties du robotino permet de raccorder des extensions, telles que des capteurs et actionneurs. L'interface met à disposition des entrées/sorties numériques (TOR), des entrées analogiques. L'interface d'E/S est directement montée sur la platine principale de l'unité de commande.

Figure 8 Entrées/Sorties du robotino

Entrées analogiques et digitales

Les entrées analogiques ou digitales permettent de connecter des capteurs ou des actionneurs, par exemple : Les capteurs optiques gauche et droit permettent de détecter diverses surfaces et couleurs sur la base des différences de coefficient de réflexion. Ils s'exploitent par l'intermédiaire des entrées digitales. Le capteur inductif quant à lui permet de détecter des objets métalliques, il s'exploite par l'intermédiaire des entrées analogiques.

Nous ajoutons donc dans le main une variable globale, correspondant aux entrées digitales et analogiques, que nous utiliserons dans le programme du corps de l'interface. Nous utilisons les méthodes suivantes pour récupérer l'états des entrées analogiques et digitales.

```
void rec::robotino::api2::DigitalInputArray::values(int *readings)const
void rec::robotino::api2::AnalogInputArray::values(float *readings)const
```

Pour optimiser l'affichage, nous utilisons un *WGridLayout*, afin d'afficher les capteurs dans 2 colonnes différentes.

Sorties digitales

Les sorties digitales envoient la valeur True si la sortie est activée ou alors False si la sortie est désactivée au Robotino. Pour pourvoir envoyer les valeurs au Robotino, nous avons fait le choix de faire des boutons dans notre interfaces graphique (car ils émettent un signal), pour avoir la possibilité de voir quelles sorties nous activons ou désactivons. La méthode suivante modifie toutes les sorties digitales en même temps.

```
void rec::robotino::api2::DigitalOutputArray::setValues(const int *values, unsigned
int size)
```

Afin de conserver en mémoire les sorties digitales déjà activées auparavant, nous utilisions un tableau en variable globale qui mémorise leur état. L'appui sur un de ces boutons appelle la méthode pour activer les sorties. Si la sortie est désactivée, le bouton est rouge, si elle est activée, il est vert.

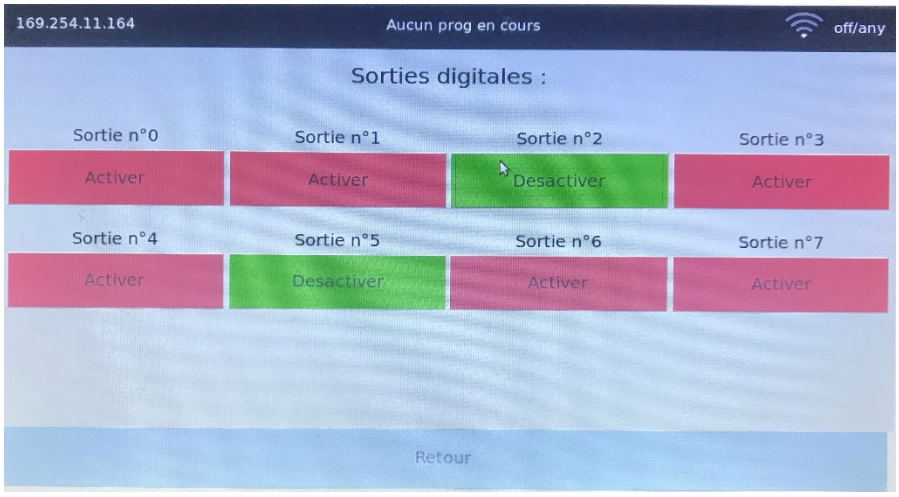


Figure 9 Etat des sorties digitales

Camera

A l'heure actuelle, Web Toolkit ne permet pas de diffuser un flux vidéo en direct. Nous nous résolvons donc à afficher des images, qui défileront à un intervalle de temps régulier, de manière à ce que le changement soit invisible. Nous avons un programme de démonstration fourni avec le Robotino qui permet de prendre une photo avec caméra toutes les 10ms et l'enregistre dans un fichier image. Nous allons donc lancer ce programme en parallèle de notre interface et afficher les images.

Nous avons essayé d'ajouter une image provenant de la caméra, et grâce à un timer, d'actualiser son contenu. Cependant, lors du lancement de l'application web, l'image est chargée et reste la même jusqu'à ce que l'on quitte. Après de nombreuses recherches, nous avons découvert l'existence du Widget *WFileResource*. Il s'agit d'une ressource qui crée des données provenant d'un fichier local en continu. Nous créons donc un *WFileRessource* à partir du fichier "camera.jpg" généré par le programme de caméra toutes les 10ms. Nous lions cette ressource à une WImage.

```
auto imageFile = std::make_shared<Wt::WFileResource>("image/jpg",
"./ressources/camera.jpg");
auto image = std::make_unique<Wt::WImage>(Wt::WLink(imageFile));
```

Ainsi, à l'aide d'un timer, nous sommes capables d'afficher le flux vidéo en direct. Avec un intervalle de temps de 10ms, nous ne voyons pas qu'il ne s'agit en fait que d'images successives.

Pour lancer le programme de caméra, nous devons être en super utilisateur. Nous ne voulons pas lancer l'interface graphique en super utilisateur car il y a des risques que les utilisateurs puissent lancer d'autres commandes en super utilisateur. Nous avons donc pensé à utiliser super, que nous avions utilisé en 3e année en systèmes et réseau. Super permet à un utilisateur non root de lancer des commandes prédéfinies comme s'il était root.

Nous avons donc créé un script camera.sh qui lance le programme caméra. Nous avons ajouté dans le fichier de configuration de super, */etc/super.tab*

```
camera /home/robotino/work/robotino2_gui/scripts/camera.sh robotino uit=root
```

Ainsi, pour lancer la caméra, nous utilisons

```
system("super camera") ;
```

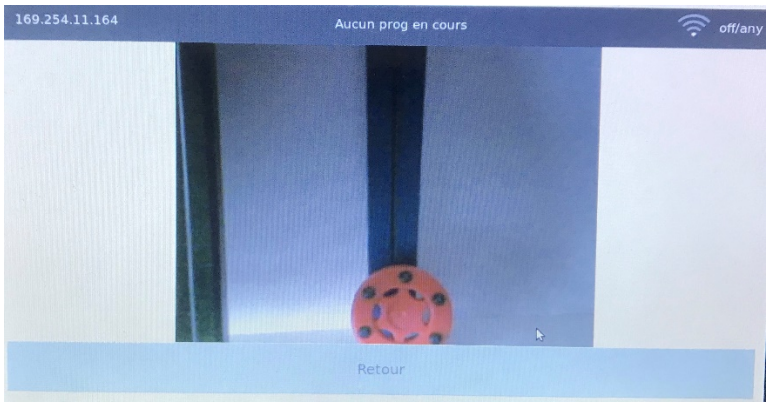


Figure 10 Affichage de la caméra

2.6. Connexion réseau

<u>Lister les réseaux Wifi disponible</u>

Grâce à la commande *iwconfig*, nous savons que la carte réseau sans fil du robotino est nommée wlp1s0.

Dans un premier temps nous devons déconnecter le network-manager afin de configurer manuellement le réseau en ligne de commande. Network-Manager est l'outil de gestion des connexions réseau d'Ubuntu. Pour cela, nous utilisons la commande :

```
sudo service network-manager stop
```

Si l'on souhaite réactiver network-manager, il faut utiliser la commande :

```
sudo service network-manager start
```

Dans un second temps, nous utiliserons la commande suivante pour lister tous les réseaux Wifi.

```
iwlist wlp1s0 scan
```

Cette commande renvoie tous les réseaux wifi détectés ainsi que des informations les concernant. Nous utilisons ensuite des commandes sed et grep afin de retourner uniquement les noms de réseaux wifi.

```
iwlist wlp1s0 scan | grep ESSID | sed -e "s/ESSID:\"//g"| sed -e "s/\"//g" |
sed "/x00/d"
```

<u>Connexion a un réseau Wifi</u>

Pour se connecter à un réseau wifi, nous utilisons les commandes :

```
wpa_passphrase nom_reseau_wifi mot_de_passe > ~/wpa.conf
wpa_supplicant -B -c ~/wpa.conf -i wlp1s0
```

La première ligne de commande sert à enregistrer le nom du réseau sur lequel on souhaite se connecter ainsi que son mot de passe, et la deuxième ligne de commande sert à s'enregistrer.

Cependant, si les commandes précédentes ne fonctionnent pas, nous utilisons la commande suivante qui permet de tuer les processus wpa_supplicant déjà existants et de relancer la connexion.

```
sudo killall wpa_supplicant
```

Enfin une fois connecté et reconnu par la borne Wifi, on va demander une nouvelle adresse IP avec le protocole DHCP grâce à la commande

```
dhclient wlp1s0
```

Lancement des commandes

Pour lancer les commandes, nous devons utiliser la fonction *popen* pour lister les réseaux Wifi et les récupérer, et la fonction *system* pour lancer la connexion et la récupération d'une adresse IP.

Cependant, les commandes de connexions doivent être exécutées en super utilisateur. Nous ne voulons pas lancer l'interface graphique en super utilisateur car il y a des risques que les utilisateurs puissent lancer d'autres commandes en super utilisateur. Nous avons donc pensé à utiliser *super*, qui permet à un utilisateur non root de lancer des commandes prédéfinies comme s'il était root.

Nous avons donc créé un script pour chaque fonctionnalité que l'on ajoute dans le fichier de configuration de super.

```
listeWifi /home/robotino/work/robotino2_gui/scripts/listeWifi.sh robotino
uit=root
```

Ainsi, pour lister les réseaux wifi, il faut simplement lancer

```
System("super listeWifi");
```

Interface connexion réseau

Nous avons choisi d'utiliser un *WSelectionBox*, qui affiche une liste de textes et permet d'en sélectionner un. Nous ajoutons un bouton permettant de valider le choix du réseau Wifi.

Une fois le réseau Wifi sélectionné, nous affichons un *WLineEdit* permettant de rentrer du texte. Nous le configurons en mode *Password* afin de ne pas afficher le mot de passe en clair sur l'interface.

Nous ajoutons un bouton permettant de valider le mot de passe et de lancer la connexion en exécutant les commandes citées précédemment.

Récupération des données, pour l'affichage dans le bandeau

Dans le bandeau, nous désirons afficher l'adresse IP et le réseau wifi auquel le Robotino est actuellement connecté. Pour récupérer l'adresse IP, nous lançons la commande :

```
ip a | grep "inet.*wlp1s0" | cut -f6 -d" " | sed "s|\([0-9.]*\).*|\1|g
```

ip a liste toutes les interfaces réseau et leurs caractéristiques. Nous récupérons donc les informations concernant l'interface réseau sans fil wlp1s0 pour en extraire l'adresse ip. Nous utilisons *popen* pour lancer cette commande, récupérer le résultat et l'afficher dans le bandeau.

De même pour récupérer le Wifi auquel le robotino est connecté, nous lançons la commande

```
iwconfig wlp1s0 | grep ESSID | sed -e "s/.*ESSID:\(.*\)/\1/g"| sed
"s/\"\(.*\)\"/\1/"
```

iwconfig permet de connaitre la configuration de l'interface réseau sans fil wlp1s0. Nous extrayons ensuite le réseau wifi. Nous utilisons *popen* pour lancer cette commande, récupérer le résultat et l'afficher dans le bandeau.

Enfin, pour récupérer la qualité du wifi, nous procédons de même grâce à *iwconfig* qui affiche la qualité

```
iwconfig wlp1s0 | grep Quality | sed -e "s/.*Quality=\([0-9]*\).*/\1/"
```

Nous utilisons *popen* pour lancer cette commande, récupérer le résultat. En fonction de la qualité du wifi, nous affichons une icône wifi, avec une barre si la qualité est médiocre, 2 ou 3 barres si elle est moyenne et 4 barres si la qualité est bonne.

2.7. Lancement programmes de démonstration

Pour lancer les programmes de démonstrations, nous ajoutons un *WSelectionBox* ainsi qu'un bouton permettant de valider le choix. Les programmes de démonstrations sont placés dans le dossier /usr/local/robotino/examples/bin/. Pour ajouter un nouveau programme de démonstration, il faut l'ajouter à la boite de sélection dans le fichier corps/interface.C, dans la méthode ChoixDémo :

```
model->addString(<nom du prog>);
model->setData(<index suivant>, 0, std::string("<chemin vers le programme>
127.0.0.1"), Wt::ItemDataRole::User);
```

Pour lancer le programme, nous utilisons par exemple pour la démonstration circle :

```
system("/usr/local/robotino/examples/bin/example_circle 127.0.0.1 &");
```

Le nom du programme à lancer est récupéré en fonction de la sélection. Le programme est lancé en arrière-plan pour ne pas interférer sur notre interface graphique.

Lorsqu'un programme de démonstration est lancé, son nom est récupéré et affiché dans le bandeau. Nous avons ajouté dans le bandeau un timer afin d'actualiser le nom du programme en cours.

Pour arrêter un programme de démonstration, nous ajoutons dans le bandeau un bouton arrêt qui s'affiche uniquement quand le programme est lancé. Lors d'un appui sur ce bouton, nous récupérons le pid du processus correspondant au programme de démonstration pour l'arrêter.

```
system("kill -9 $(pidof -x \"<nom du processus>\"")
```

Réalisation de notre programme de démonstration

En plus de réaliser l'interface graphique, nous avons réalisé un programme de démonstration en C++, Nous voulions que ce programme soit différent de ceux déjà existant (Circle, Suivie de mur, Exploration, Évitement d'obstacle). Nous nous sommes rendu compte qu'aucun programme de démonstration n'utilisait la caméra, alors nous avons pensé à un Robotino joueur de hockey.

Le principe de fonctionnement du programme est le suivant :

Le Robotino tourne sur lui-même jusqu'à ce que la caméra détecte un palet rouge. Une fois détecté, le Robotino avance vers lui pour ensuite tirer dedans.

Pour commencer, nous avons effectué du traitement d'image, pour la détection du palet, nous voulions une fonction qui renvoie :

- *True* si le Robotino détecte le palet rouge
- *False* sinon

Nous avons commencé par implémenter une fonction qui compte le nombre de pixels rouges dans l'image que capte la caméra. Nous avons pour cela repris le principe de l'exemple caméra. Au lieu d'écrire chaque pixel dans un fichier image pour obtenir la photo, nous regardons si le pixel est de couleur rouge. Un compteur permet de connaitre le nombre de pixels rouges.

Pour détecter un pixel rouge nous avons utilisé la condition suivante :

```
if(R>150 && G<100 && B<100)
   nombre_pixel_rouge++;
```

Maintenant que le Robotino détecte la couleur rouge, il faut que celui-ci s'arrête en face du palet pour aller le chercher (que le palet se trouve au centre de l'image). Pour cela nous voulions que le robotino ne compte les pixels rouges que sur une bande horizontale au centre de l'image (320*240 pixels) alors nous avons modifié la condition suivante :

```
if(i>longueur/2-20 && i<longueur/2+20 && R>150 && G<100 && B<100)
   nombre_pixel_rouge++;
```

Ainsi, le compteur ne détecte que les pixels rouges se trouvant au centre de la caméra.

En fonction du nombre de pixels détecté, le robotino arrête le traitement d'image et avance vers le palet. Après plusieurs tests, nous avons fixé le seuil de pixels rouge à 200. Nous faisons ensuite avancer le Robotino jusqu'à ce que le capteur de distance avant le détecte dans la pince. Puis on effectue des mouvements afin de tirer dans le palet.

Réflexions, conclusion

Nous sommes globalement satisfaits de notre projet et de ce que nous avons réussi à produire. Notre interface graphique est totalement fonctionnelle. Elle pourra être utilisée à l'avenir sur les autres robotinos 2 qui seront upgradés. Elle offre un confort d'utilisation qui n'était pas présent avec l'interface Homme-Machine des robotinos 2.

Nous notons cependant deux petits soucis. Les batteries du robotino que nous avons utilisé ne tiennent pas longtemps. Un bouton d'arrêt complet du robotino a été installé afin que le robot ne consomme pas de batterie quand il est éteint. Ensuite, nous avons constaté que le tactile n'est pas optimisé avec Ubuntu. En effet, il faut parfois appuyer plusieurs fois sur un bouton avant qu'il soit activé. Une solution pourrait être d'utiliser un autre système d'exploitation mieux adapté au tactile.

Pour conclure, nous avons apprécié travailler sur ce projet. Nous étions en réelle situation d'ingénieur avec un projet à réaliser dans un temps limité et des contraintes et un cahier des charges à respecter. Nous avons réalisé ce projet en organisation des réunions bimensuelles avec nos encadrants et en leur faisant part de notre avancée tout au long du projet. Ce projet fut très enrichissant et nous a apporté de nombreuses compétences, aussi bien techniques (utilisation de c++, d'un Framework, d'une API) que d'organisation, d'adaptation et d'autonomie.

Annexe : Tuto Web Toolkit

Etant donné que nous avons eu du mal à trouver des tutoriels sur Web Toolkit (nous avons uniquement la documentation de la bibliothèque en anglais qui n'est pas évidente à comprendre), nous exposons ci-après un "mini-tutoriel" qui reprend tous les widgets que nous avons utilisés pour notre projet ainsi que le fonctionnement général de WT. Cela pourra être utile si des personnes veulent mieux comprendre notre code ou mettre à jour notre interface graphique.

Main : lancement de l'application

Dans notre fichier main.C, la fonction main retourne la fonction WRun qui prend en paramètre une fonction createApplication.

```
int main(int argc, char **argv)
{
  return WRun(argc, argv, &createApplication);
}
```

La fonction WRun lance le serveur de l'application. Le paramètre createApplication est une fonction qui crée une instance de l'application pour un nouveau visiteur de l'interface. C'est dans cette fonction que nous créons une instance de la classe bandeau et une instance de la classe interface, qui est le corps de l'interface.

```
std::unique_ptr<WApplication> createApplication(const WEnvironment& env)
{
 appl->useStyleSheet("main.css");
 std::unique_ptr<WApplication> appl= cpp14::make_unique<WApplication>(env);
 auto band = appl->root()->addWidget(cpp14::make_unique<bandeau>());
 band->setStyleClass("bandeau");
 auto corps= appl->root()->addWidget(cpp14::make_unique<interface>(env));
 corps->setStyleClass("corps");
 appl->setTitle("Robotino GUI");
 return appl;
}
```

Classes créees

Nous avons créé des classes de bandeau et d'interfaces. Ces classes sont des sous-classes de WContainerWidget. Elles créent donc un WContainerWidget et le remplissent.

```
class bandeau : public WContainerWidget {
public:
        bandeau();
};
```

Widgets utilisés

Voici la liste des Widgets principaux de WT que nous avons utilisés. Cette liste est évidemment non exhaustive.

WContainerWidget

Permet de séparer les widgets dans des parties différentes de l'écran. Nous avons créé une fonction nous permettant d'ajouter et de paramétrer facilement un WContainerWidget.

```
WContainerWidget* ajoutContainer(Wt::WContainerWidget *parent,const char*
classeCSS)
{
  WContainerWidget* container=parent-
>addWidget(std::make_unique<Wt::WContainerWidget>());
  if(classeCSS!=NULL)
    container->setStyleClass(classeCSS);
  return container;
}
```

WText

Pour afficher du texte, il faut utiliser la classe WText(). Etant donné que nous utilisons souvent ce widget, nous avons créé une fonction permettant de créer et paramétrer ce widget.

```
WText* ajoutText(WContainerWidget *container,const char* text,const char*
classeCSS, int centered)
{
  auto texte = container->addWidget(std::make_unique<Wt::WText>(text));
  if (centered==1)
    texte->setTextAlignment(Wt::AlignmentFlag::Center);
  texte->setInline(0);
  if(classeCSS !=NULL)
    texte->setStyleClass(classeCSS);
  return texte;
}
```

WPushButton

Permet d'ajouter un bouton. Nous avons créé une fonction nous permettant d'ajouter et de paramétrer facilement un WPushButton.

```
WPushButton* ajoutBouton(WContainerWidget *container,const char* text, const
char* id, const char* classeCSS, const char* path){
  WPushButton *btn = container-
>addWidget(std::make_unique<Wt::WPushButton>(text));
  btn->setTextFormat(TextFormat::XHTML);
  if (id!=NULL)
    btn->setId(id);
  if(classeCSS!=NULL)
    btn->setStyleClass(classeCSS);
  btn->setLink(WLink(Wt::LinkType::InternalPath,path));
  return btn;
```

```
}
```

Le path entré en paramètre permet de changer l'URL lorsque le bouton est cliqué et d'envoyer un signal InternalPathChanged

Pour exécuter d'autres fonctions lorsqu'un bouton est cliqué, nous utilisons

```
btn->clicked().connect([=] { /*code à executer lorsque le bouton est cliqué*/
});
```

La méthode setIcon() permet d'ajouter une image dans le bouton.

WImage

Permet d'ajouter une image. Nous avons créé une fonction nous permettant d'ajouter et de paramétrer facilement une WImage.

```
void ajoutImage(WContainerWidget *container,const char* name,const char*
classeCSS){
  auto image =Wt::cpp14::make_unique<Wt::WImage>(Wt::WLink(name));
  if(classeCSS!=NULL)
    image->setStyleClass(classeCSS);
  container->addWidget(std::move(image));
}
```

WTimer

Permet de créer un timer qui crée un signal à intervalle de temps régulier. Nous avons créé une fonction permettant d'ajouter et de paramétrer facilement un WTimer. Cette fonction configure le timer et le lance.

```
WTimer* addStartTimer(WContainerWidget *container, int time_ms){
  WTimer *timer=container->addChild(std::make_unique<Wt::WTimer>());
  timer->setInterval(std::chrono::milliseconds(time_ms));
  timer->start();
  return timer;
}
```

Pour récupérer le signal et exécuter des fonctions à intervalle de temps régulier

```
mytimer->timeout().connect([=] { /*code à executer à intervalle de temps du
timer*/ });
```

WBreak

Permet d'ajouter un espace dans un WContainerWidget.

```
void ajoutBreak(WContainerWidget *container)
{
  container->addWidget(Wt::cpp14::make_unique<Wt::WBreak>());
}
```

WSelectionBox et WListSringModel

Permet d'afficher une liste de texte et d'en sélectionner un. Utilisation de WStringListModel pour lister les textes à afficher dans le sélectionneur ainsi qu'une valeur correspondant à chaque texte.

```
auto sélection =<container>-
>addWidget(Wt::cpp14::make_unique<Wt::WSelectionBox>());
auto model = std::make_shared<Wt::WStringListModel>();
model->addString( <1er texte>);                                    //Le texte
affiché à l'indice 0
model->setData(1, 0, <valeur> , Wt::ItemDataRole::User);           //la
valeur correspondant à l'indice 0
model->addString(<2e texte>);
model->setData(1, 0, <valeur> , Wt::ItemDataRole::User);
selection->setModel(model);
selection->setCurrentIndex(0);                                        //Le
texte d'indice 0 sera par défaut selectionné
```

Navigation entre les menus

Chaque fois que l'on veut changer ce qui est affiché à l'écran (lors de l'appui sur un bouton principalement), le "path" ou l'URL est modifiée et un signal InternalPathChanged est envoyé. Pour cela, nous récupérons une instance de l'application WApplication actuelle et définissons la fonction à appeler lorsque l'URL est modifiée.

```
WApplication *app = WApplication::instance();
app->internalPathChanged().connect(this, &interface::handlePathChange);
void interface::handlePathChange()
{
   WApplication *app = WApplication::instance();
   corps->clear();
   if (app->internalPath() == <chemin1>)
     <fonction à appeler quand URL est chemin1>;
   else if (app->internalPath()== <chemin2>)
     <fonction à appeler quand URL est chemin1>;
}
```

Cette fonction supprime tous les widgets du WContainerWidget principal corps grâce à la méthode clear(), puis appelle une fonction qui va remplir ce container avec de nouveaux widgets.

Layouts : Agencement des Widgets dans un WContainerWidget

Les widgets sont ajoutés au WContainerWidget dans leur ordre d'ajout et les uns à la suite des autres. Ils seront côte à côte ou les uns en dessous des autres en fonction de leur disposition. Il existe des layouts permettant de réaliser d'autres configurations.

WGridLayout

Permet de ranger les widgets dans une "grille", un tableau. Pour cela, il faut ajouter les widgets au layout en précisant la colonne et la ligne désirées dans le tableau.

```
auto layout = std::make_unique<Wt::WGridLayout>();
layout->addWidget(<Widget>,<ligne>,<colonne>);
```

WHboxLayout

Permet de ranger les widgets dans des cases à l'horizontal sur une ligne. Existe aussi WVBoxLayout qui permet de ranger les widgets sur une colonne à la verticale.

```
auto hbox = std::make_unique<Wt::WHBoxLayout>();
hbox->addWidget(<Widget>);
```

Définir le layout d'un WContainerWidget

Une fois le layout défini, il suffit d'indiquer au container que l'on veut utiliser ce layout.

```
<container> -> setLayout(std::move(<layout>));
```