INTERFACE GRAPHIQUE DE CONTRÔLE POUR LE ROBOT LEENBY

Aujourd'hui, la seule application permettant de contrôler le robot Leenby n'est utilisable que sous Windows. Le but de cette interface, est de fournir un moyen de contrôler le robot Leenby à distance avec un ordinateur sous Linux, en utilisant ROS.

Pour cela, un package hybride ROS-Qt est utilisé. L'interface est codée en C++ avec l'utilisation des widgets Qt, ainsi que certains widgets personnalisés comme le joystick en lui même. En parallèle de cela, des informations de commande sont publiées sur des topics ROS.

Dans cette documentation, nous expliquerons le fonctionnement général du point de vue de l'utilisateur, puis nous inspecterons morceau par morceau le code Qt permettant l'affichage, puis le code permettant le traitement et la communication des informations.

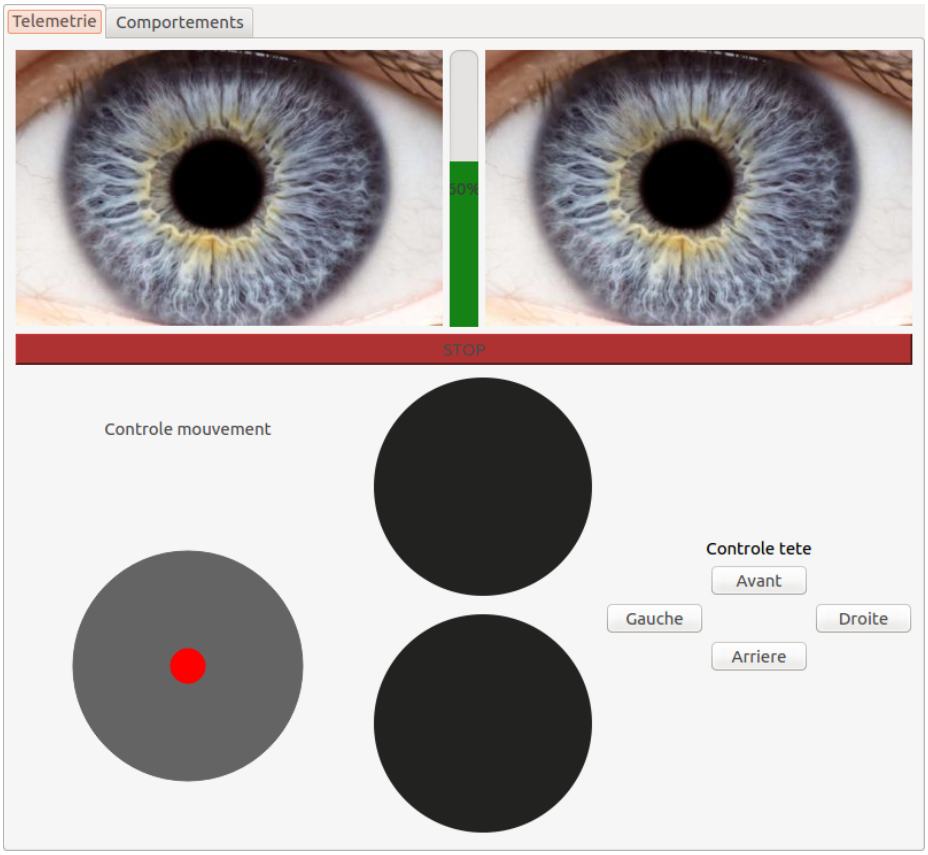

Illustration 1: L'interface

<u>Note :</u> Au moment de la rédaction de cette documentation, l'intégralité des fonctionnalités ne sont pas implémentées. Nous décrirons ici seulement les parties fonctionnelles.

Présentation générale

Commençons d'abord par lister les fonctionnalités prévues pour cette interface :

- Affichage des retours des caméras
- Affichage du niveau de batterie restant
- Bouton d'arrêt d'urgence (x)
- Joystick (x)
- Affichage des retours des lidars
- Contrôle des mouvements de la tête

Note : Les fonctionnalités suivies d'un « (x) » sont celles qui sont actuellement implantées et fonctionnelles.

Description des fonctionnalités :

Les deux images d'œil en haut à gauche et en haut à droite de l'illustration 1 sont les emplacements destinés à accueillir les retours des caméras. Ces retours ne sont actuellement pas mis en place. Ce qui est prévu est de récupérer les images des caméras en temps réel via un topic ROS, puis de les afficher, toujours en temps réel, dans ces emplacements.

L'affichage du niveau de batterie restant se fait via la barre verte en haut au centre de l'interface. Tout comme le retours des caméras, cette fonctionnalité n'est pas encore mise en place. Ce qui est prévu, est simplement de récupérer la valeur de batterie restante qui serait publiée sur un topic ROS, puis de mettre à jour cette valeur dans l'interface.

Le bouton d'arrêt d'urgence : ce bouton rouge a pour but de stopper la Leenby en cas de danger. Pour cela, un message de commande est envoyé pour stopper les moteurs, puis tous les widgets permettant de contrôler la Leenby sont désactivés.

Le joystick est au moment de la rédaction de cette documentation, la fonctionnalité la plus travaillée. Ce joystick est un widget, et peut donc être dupliqué. Nous pourrions par exemple utiliser un joystick à la place des boutons permettant le contrôle des mouvements de la tête. Ce joystick permet d'obtenir deux valeurs : une pour la vitesse linéaire, et une pour la vitesse de rotation. Ces deux informations subissent ensuite un léger traitement, avant d'être publiées.

Les deux ronds noirs présent sur l'illustration 1 sont les emplacements prévus pour recevoir et afficher les retours des deux lidars de la Leenby. Actuellement non implémentés, il s'agit là d'une des fonctionnalités pour laquelle la futur implémentation est encore en réflexion. Les choix principaux sont : l'affichage d'un flux d'images ou d'un flux vidéo en temps réel des retour des lidars, ou d'utiliser la bibliothèque librviz pour incruster un widget de visualisation 3D.

Finalement, les quatre boutons présent sur la droite de l'interface permettrons dans le futur de contrôler les mouvements de la tête de la Leenby, toujours en publiant des informations de contrôle sur un topic ROS.

Pour accéder à certaines fonctionnalités importante, des raccourcis clavier ont été mis en place :

Quitter	Ctrl + q
Arrêt d'urgence	Space
Réactiver les contrôles	Ctrl + Alt + Space
Tête avant	z
Tête arrière	s
Tête gauche	q
Tête droite	d

Tableau 1: Raccourcis clavier

Comportements du joystick :

L'objectif a été d'avoir un joystick sous forme de widget, avec un comportement aussi réaliste que possible. Les principales règles pour ce mimétisme ont été les suivantes :

1) Devoir attraper la partie mobile du joystick pour pouvoir la déplacer.

2) La partie mobile du joystick ne doit pas pouvoir sortir d'un cercle défini

3) Si le curseur de l'utilisateur sort du cercle défini, la partie mobile doit être positionnée à sa distance maximale, et toujours suivre le curseur.

Ensuite, l'information envoyée par le joystick est déterminée en fonction de la position de la partie mobile.

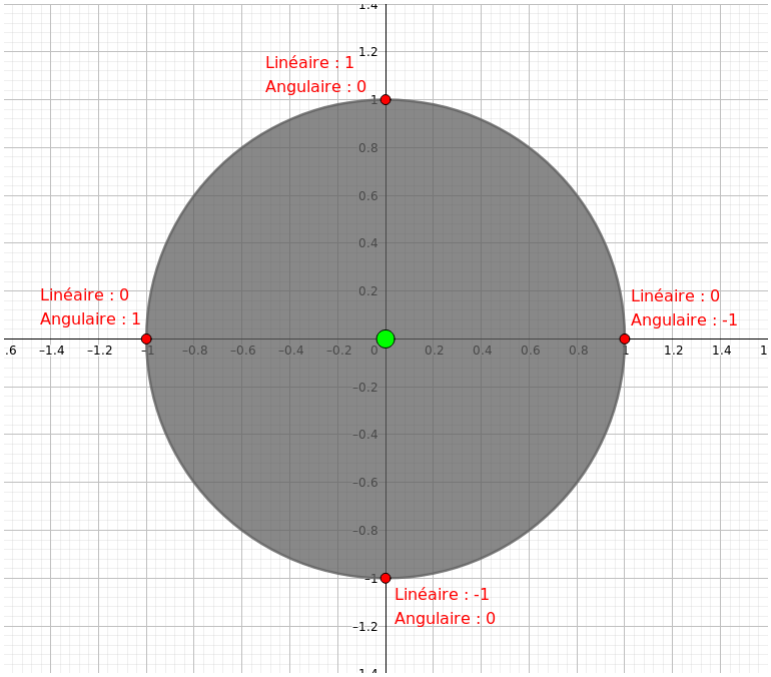

Illustration 2: Graphique de comportement du joystick

Comme nous pouvons le voir sur l'illustration 2 ci-dessus, la commande linéaire est déterminée en fonction de la position sur l'axe Y de la partie mobile (sur cette illustration en vert), et la commande angulaire est déterminée en fonction de la position sur l'axe X de la partie mobile.

L’affichage (C++ et Qt)

La répartition des éléments graphiques dans les containers est assez simple :

Un QBoxLayout englobe la totalité de la fenêtre. Ce layout est ensuite divisé en trois QHBoxLayout, qui accueillent respectivement, les retours caméras ainsi que l’afficheur du niveau de batterie, le bouton d’arrêt d’urgence, le joystick ainsi que les retours des lidars et les boutons de commande de la tête.

Une représentation de ces containers pourrait être la suivante :

<table>
<tr><td>Caméra gauche</td><td>Batterie</td><td>Caméra droite</td></tr>
<tr><td colspan="3">Bouton d’arrêt d’urgence</td></tr>
<tr><td>Texte
Joystick</td><td>Retour lidar tête
Retour lidar base</td><td>Texte
Bouton avance
Bouton gauche
Bouton droite
Bouton recul</td></tr>
</table>

Tableau 2: Répartition des widgets dans les containers

Cette disposition est celle qui, au moment de la rédaction de la documentation, fonctionne correctement. En fonction de l’évolution de l’implantation des nouvelles fonctionnalités, cette disposition pourra être modifiée.

Voici une deuxième disposition qui pourrait être utilisée :

<table>
<tr><td>Joystick
mouvement</td><td>Choix de la partie
a bouger</td><td>Retour des caméras
en une image</td></tr>
<tr><td colspan="3">Bouton d’arrêt d’urgence</td></tr>
<tr><td>États des capteurs</td><td>Niveau de
batterie
restant</td><td>Map SLAM</td></tr>
</table>

Les éléments :

<u>Note :</u> Dans les captures d'écran qui vont suivre, certaines lignes de code ne seront pour le moment pas expliquées. La plupart de ces lignes sont des connexions de signaux à des slots ou des raccourcis clavier. Pour ce qui est des raccourcis, le tableau 2 ci-dessus les listes tous. Quant aux signaux et aux slots, ils seront abordés dans la partie suivante.

Les retours des caméras sont au moment de la rédaction de cette documentation représentés par des QLabel hébergeant des images fixes. Dans le futur, il faudrait remplacer ces QLabel par un widget capable d'accueillir un flux vidéo en temps réel.

Pour ce qui est de l'affichage du niveau de batterie restant, nous avons utilisé une QProgressBar positionnée verticalement, de couleur verte, et pouvant afficher des valeurs comprises entre 0 et 100. Ces valeurs sont exprimée en pourcentage de batterie restante.

```
QHBoxLayout *camContainer = new QHBoxLayout;

    m_renduCamGauche = new QLabel();
    m_renduCamGauche->setPixmap(QPixmap(":/images/oeil.jpeg"));

    QProgressBar *batteryLevel = new QProgressBar();
    batteryLevel->setOrientation(Qt::Vertical);
    batteryLevel->setTextVisible(true);
    batteryLevel->setStyleSheet("QProgressBar::chunk { background-color:rgb(20,130,20); }");
    batteryLevel->setAlignment(Qt::AlignCenter);
    batteryLevel->setValue(60);

    m_renduCamDroite = new QLabel();
    m_renduCamDroite->setPixmap(QPixmap(":/images/oeil.jpeg"));

    m_renduCamGauche->setAlignment(Qt::AlignCenter);
    batteryLevel->setAlignment(Qt::AlignCenter);
    m_renduCamDroite->setAlignment(Qt::AlignCenter);

camContainer->addWidget(m_renduCamGauche);
camContainer->addWidget(batteryLevel);
camContainer->addWidget(m_renduCamDroite);
```

Illustration 3: Code du container d'affichage des retours caméras et du niveau de batterie

Le bouton d'arrêt d'urgence est simplement un QPushButton de couleur rouge. Du fait que ce bouton est placé seul dans un QHBoxLayout, il s'étendra jusqu'à le remplir. Cela permet d'avoir un bouton large et visible, facilement accessible en cas de besoin.

Malgré la simplicité du code de ce widget, il ne faut pas le sous-estimer. Il s'agit sûrement du widget le plus important de cette interface. Il est le seul moyen d'arrêter la Leenby à distance.

```
QHBoxLayout *stopContainer = new QHBoxLayout;
    m_boutonStop = new QPushButton("STOP");
    m_boutonStop->setStyleSheet("QPushButton { background-color : rgb(175,50,50);}");
    m_boutonStop->setShortcut(QKeySequence("Space"));
    QObject::connect(m_boutonStop, SIGNAL(clicked()), this, SLOT(emergencyStop()));
stopContainer->addWidget(m_boutonStop);
```

Illustration 4: Code du container d'affichage du bouton d'arrêt d'urgence

Le texte au dessus du joystick est simplement contenu par un QLabel. Cependant, le joystick est un widget personnalisé plus complexe du type JoystickWidget qui hérite de la classe QWidget. Il s'agit de deux cercles, le gris correspond à la surface d'utilisation du joystick, et le rouge est l'élément mobile. Sa taille idéale est de 300x300. Ces cercle sont redessiné à chaque nouvel événement lancé par la souris.

Les affichages des retours des lidars sont pour le moment des images dans des QLabel. Il faudra par la suite intégrer un nouveau widget afin d'afficher le vrai retour.

Finalement, un QGridLayout contient le texte et les boutons qui contrôlent la tête de la Leenby. De façon assez peu académique, le texte est un bouton désactivé et personnalisé. Cela est simplement pour éviter le décalage que créerai un QLabel. Pour finir, les quatre boutons sont du type QPushButton.

```
QHBoxLayout *bottomContainer = new QHBoxLayout;

    QVBoxLayout *boutonDeplacementContainer = new QVBoxLayout;
        QLabel *txtDeplacement = new QLabel("Controle mouvement");
        txtDeplacement->setAlignment(Qt::AlignCenter);
        joystick = new JoystickWidget();
        QObject::connect(joystick, SIGNAL(hasMoved()), this, SLOT(joystickCallback()));

    boutonDeplacementContainer->addWidget(txtDeplacement);
    boutonDeplacementContainer->addWidget(joystick);

    QVBoxLayout *lidarsOutput = new QVBoxLayout;

        QLabel *teteLidar = new QLabel();
        teteLidar->setPixmap(QPixmap(":/images/rond.png"));
        QLabel *baseLidar = new QLabel();
        baseLidar->setPixmap(QPixmap(":/images/rond.png"));

        lidarsOutput->addWidget(teteLidar);
        teteLidar->setAlignment(Qt::AlignCenter);
        lidarsOutput->addWidget(baseLidar);
        baseLidar->setAlignment(Qt::AlignCenter);

    QVBoxLayout *boutonTeteContainer = new QVBoxLayout;

        QGridLayout *gridBoutonTete = new QGridLayout;
            QPushButton *txtTete = new QPushButton("Controle tete");
            txtTete->setStyleSheet("QPushButton { color : black; border:0px}");
            txtTete->setDisabled(true);
            m_boutonTeteAvant = new QPushButton("Avant");
            m_boutonTeteAvant->setShortcut(QKeySequence("z"));
            qnode.connect(m_boutonTeteAvant, SIGNAL(clicked()),this, SLOT(bougerTeteAvant()));
            m_boutonTeteArriere = new QPushButton("Arriere");
            m_boutonTeteArriere->setShortcut(QKeySequence("s"));
            QObject::connect(m_boutonTeteArriere, SIGNAL(clicked()),this, SLOT(bougerTeteArriere()));
            m_boutonTeteGauche = new QPushButton("Gauche");
            m_boutonTeteGauche->setShortcut(QKeySequence("q"));
            QObject::connect(m_boutonTeteGauche, SIGNAL(clicked()),this, SLOT(bougerTeteGauche()));
            m_boutonTeteDroite = new QPushButton("Droite");
            m_boutonTeteDroite->setShortcut(QKeySequence("d"));
            QObject::connect(m_boutonTeteDroite, SIGNAL(clicked()),this, SLOT(bougerTeteDroite()));

        gridBoutonTete->addWidget(txtTete,0,0,1,3);
        gridBoutonTete->addWidget(m_boutonTeteAvant,1,1);
        gridBoutonTete->addWidget(m_boutonTeteArriere,3,1);
        gridBoutonTete->addWidget(m_boutonTeteGauche,2,0);
        gridBoutonTete->addWidget(m_boutonTeteDroite,2,2);

    boutonTeteContainer->addLayout(gridBoutonTete);
```

Illustration 5: Code d'affichage du container du joystick, des retours des lidars et des boutons de contrôle de la tête

Le traitement des informations (C++ et ROS)

Les fichiers sont hiérarchisés de la façon suivante :

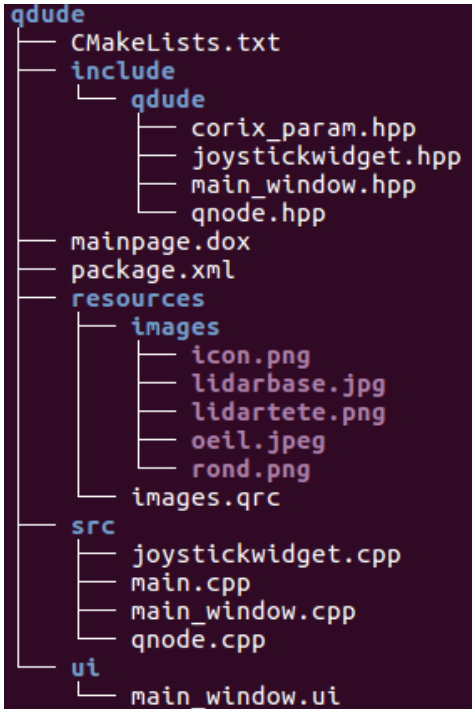

Illustration 6: Arborescence des fichiers du package

Le CMakeLists est le fichier régissant la compilation du package.

Dans le dossier include se trouvent les fichiers d'en-tête. Le corix_param.hpp défini quelques constantes interne au robot, et les trois autres fichiers sont simplement des fichiers d'en-tête de déclaration de classes ou de prototypes.

Le fichier mainpage.dox n'est pas utilisé et le fichier package.xml donne des informations simples sur le package.

Dans le dossier resources se situent les images qui sont actuellement en lieu et place des futurs retours caméras et lidars.

Le dossier src contient tout les fichiers sources. Dans ces derniers, le joystickwidget.cpp détermine le comportement du joystick, le main est peu utile et ne sert que de point d'entrée au programme. Ensuite le fichier main_window.cpp est le fichier d'affichage de la fenêtre, mais c'est aussi le fichier dans lequel les slots sont implémentés et les commandes du joystick envoyées au fichier suivant, le qnode.cpp. Ce fichier est le cœur du programme ROS. C'est dans ce fichier que la publication des informations sur les topics ROS se fait.

<u>Note :</u> Dans cette partie, les fichiers mainpage.dox, package.xlm, les fichiers de ressources et les fichiers en .ui ne seront pas abordés, ils ne sont pas spécialement intéressant.

Le joystick :

Commençons par énoncer les différents attributs, méthodes et signaux du joystick :

Premièrement, nous avons un constructeur et un destructeur. Ensuite, la méthode sizeHint() renvoie une taille idéale pour le widget.

La méthode reset() replace le joystick dans son état de départ.

```
public:
  JoystickWidget( QWidget *parent=0 );
  ~JoystickWidget(){}
  QSize sizeHint() const;

  void reset();
  double getYBase();
  double getY();
  double getXBase();
  double getX();

Q_SIGNALS : void hasMoved();


protected:
  void paintEvent( QPaintEvent* );

  void mouseMoveEvent( QMouseEvent* );
  void mouseReleaseEvent( QMouseEvent* );

private:
  double x, y, r, xBase, yBase, rBase;
  QColor color, colorBase;
  bool isMoving;
```

Illustration 7: Header du joystick

S'en suivent une série de getters traditionnels pour les attributs.

Les attributs x et y définissent la position de la partie mobile du joystick (rond rouge sur l'illustration 1). Ensuite, les attributs xBase et yBase déterminent le point 0, le centre du joystick. Ensuite, rBase défini le rayon du cercle dans lequel la partie mobile peut bouger (rond gris sur l'illustration 1).

Les attributs color et colorBase, sont respectivement la couleur de la partie mobile et la couleur du cercle dans lequel la partie mobile peut bouger.

Enfin le booléen isMoving indique simplement si la partie mobile est en train de bouger. Ce booléen est lié au signal hasMoved(). Ce signal est envoyé chaque fois que la partie mobile du joystick bouge.

Trois événements sont ensuite utiles pour faire fonctionner le joystick en temps réel. Ce widget étant contrôlé par la souris, il est logique de voir apparaître un mouseMoveEvent, qui est lancé chaque fois que la souris est cliquée et bougée, puis un mouseReleaseEvent, qui est lancé chaque fois que le clique de la souris est relâché. Finalement, un paintEvent permet de redessiner le widget à chaque nouveau changement.

Voyons maintenant le code de chacune de ces fonctions :

Le constructeur est très simple, il initialise simplement les attributs, pour que le joystick soit au centre du widget. Il initialise également les couleurs. Le booléen isMoving est initialisé à false, pour indiquer que le joystick ne bouge pas.

```
JoystickWidget::JoystickWidget( QWidget *parent ) : QWidget( parent )
{
    rBase = 100;
    xBase = rBase+50;
    yBase = rBase+50;
    colorBase = QColor(100,100,100);

    x=xBase;
    y=yBase;
    r = 15;
    color = QColor( 255, 0, 0 );

    isMoving = false;
}
```

Illustration 8: Constructeur du joystick

La fonction reset() sert donc à replacer le joystick dans son état de départ. Pour cela, elle replace la partie mobile au centre du widget, passe le booléen isMoving à false pour indiquer que le joystick ne bouge pas, puis elle met à jour le dessin.

```
void JoystickWidget::reset()
{
    x=xBase;
    y=yBase;
    update();
    isMoving = false;
}
```

Illustration 9: Fonction reset du joystick

Le premier événement est le paintEvent. Son rôle est simplement de dessiner le joystick. Il dessine des ellipses avec deux rayons similaires pour faire des cercles.

Le mouseMoveEvent est lui plus complexe. En premier deux distances sont calculées, elle sont exprimées à la puissance deux pour éviter un traitement via une racine carrée. La première est la distance entre le curseur de la souris et le centre du widget et la seconde est la distance entre le centre de la partie mobile est le curseur de la souris. Ensuite la partie « intelligente » du joystick est mise en place. Rappelons que cet événement n'aura lieu que si la souris est cliquée et déplacée. Premier test, si la distance entre le curseur et le centre de la partie mobile est inférieure au rayon au carré de la partie mobile, alors cela signifie que l'utilisateur vient de bouger la partie mobile. On passe donc le booléen isMoving à true pour indiquer un mouvement du joystick. Deuxième test, si le joystick est en mouvement, alors un nouveau test se fait. Si la distance entre le curseur et le centre du widget est supérieur au rayon au carré du cercle, alors cela signifie que le curseur est au-delà de la limite. Dans ce cas, on ramène via un simple théorème de Thalès, la partie mobile du joystick à la limite. Sinon, si ce test n'est pas vrai, alors cela signifie que la partie mobile du joystick est à l'intérieur du cercle, donc on la place au même endroit que le curseur. Finalement, on envoie le signal que le joystick a bougé, puis on met à jour le dessin.

```
void JoystickWidget::mouseMoveEvent( QMouseEvent *e )
{
    int dist =(xBase-e->x())*(xBase-e->x()) + (yBase-e->y())*(yBase-e->y());
    int distRedDot = (x-e->x())*(x-e->x()) + (y-e->y())*(y-e->y());

    if( distRedDot < (r*r) )
    {
        isMoving = true;
    }

    if( isMoving)
    {
        if( dist > (rBase*rBase) )
        {
            float ratio = rBase / sqrt(dist);    // ------
            x = xBase + ( e->x() - xBase)*ratio; // Thalès
            y = yBase + ( e->y() - yBase)*ratio; // ------
        }
        else
        {
            x = e->x();
            y = e->y();
        }
        Q_EMIT JoystickWidget::hasMoved();
    }

    update();
}
```

Illustration 10: Événement du mouvement de la souris

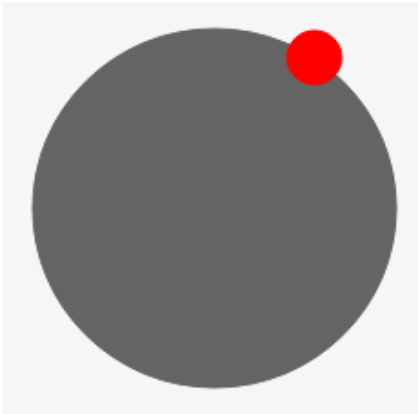

Illustration 11: Partie mobile du joystick à la limite

S’en suit logiquement l’événement mouseReleaseEvent, qui intervient après le mouseMoveEvent, quand le clique de la souris est relâché. Cet événement est très simple, il appelle simplement la fonctions reset() vue plus haut (Illustration 9), puis envoie le signal que le joystick a bougé.

```
void JoystickWidget::mouseReleaseEvent( QMouseEvent *e )
{
    reset();
    Q_EMIT JoystickWidget::hasMoved();

}
```

Illustration 12: Événement du clique de la souris relâché

La transmission des informations :

Décryptons maintenant la façon avec laquelle les informations envoyées par le joystick sont traitées et publiées sur un topic ROS.

Note : Au moment de la rédaction de cette documentation, seulement le joystick publie des commandes sur un topic ROS.

Maintenant nous nous situons dans le fichier main_window.cpp. Ce fichier est le lien entre le joystick et le fichier qnode.cpp qui publie les informations sur ROS. Les slots les plus important, sont les trois affichés dans l’illustration 13 ci-dessous.

```
public Q_SLOTS:
        void joystickCallback();
        void emergencyStop();
        void restoreControls();
```

Illustration 13: Slots important dans la communication

Le premier slot est connecté au joystick et est déclenché par le signal hasMoved. Ce slot utilise la méthode changerCommande() du qnode. Cette méthode permet de changer la commande envoyée et publiée sur le topic pour contrôler le robot.

```
void qdude::MainWindow::joystickCallback()
{
    // Les coefficents 100 peuvent et doivent être modifiés en fonction de l'utilisation
    qnode.changerCommande( (( joystick->getYBase()-joystick->getY() )/100.0) * VIT_LIN_MAX ,0,0,0,0, ((joystick->getXBase()-joystick->getX())/100.0) * VIT_ROT_MAX );
}
```

Illustration 14: Fonction callback du joystick

```
void QNode::changerCommande(double vx, double vy, double vz, double rx, double ry, double rz){
        commande.linear.x=vx;
        commande.linear.y=vy;
        commande.linear.z=vz;
        commande.angular.x=rx;
        commande.angular.y=ry;
        commande.angular.z=rz;
}
```

Illustration 15: Fonction changerCommande() du qnode

Les deux slots suivants fonctionnent ensemble. Le premier emergencyStop() est un slot appelé par le bouton d'arrêt d'urgence. Ce slot permet d'arrêter le robot, puis de désactiver les outils de commande. Et le second, restoreControls(), permet de réactiver les méthodes de contrôle. Pour cela, il envoie par sécurité un message qui immobilise le robot, puis il réinitialise le joystick via la fonction reset et finalement il réactive les méthodes de contrôle.

```
void qdude::MainWindow::emergencyStop()
{
    //Procédure d'arrêt d'urgence
    qnode.changerCommande(0,0,0,0,0,0);
    joystick->setEnabled(false);
    m_boutonStop->setEnabled(false);
    m_boutonTeteArriere->setEnabled(false);
    m_boutonTeteAvant->setEnabled(false);
    m_boutonTeteDroite->setEnabled(false);
    m_boutonTeteGauche->setEnabled(false);
    /*tocomplete*/
}
```

Illustration 16: Slot d'arrêt d'urgence

```
void qdude::MainWindow::restoreControls()
{
    //Réactivation des commandes
    qnode.changerCommande(0,0,0,0,0,0);
    joystick->reset();
    joystick->setEnabled(true);
    m_boutonStop->setEnabled(true);
    m_boutonTeteArriere->setEnabled(true);
    m_boutonTeteAvant->setEnabled(true);
    m_boutonTeteDroite->setEnabled(true);
    m_boutonTeteGauche->setEnabled(true);
    /*tocomplete*/
}
```

Illustration 17: Slot de réactivation des moyens de contrôle

Finalement, une fois la commande changée, soit par un mouvement du joystick, soit par un arrêt d'urgence, elle est publiée par le qnode qui n'est rien de plus qu'un thread lancé en parallèle de l'affichage.

```
bool QNode::init() {
    ros::init(init_argc,init_argv,"qdude");
    if ( ! ros::master::check() ) {
        return false;
    }
    ros::start(); // explicitly needed since our nodehandle is going out of scope.
    ros::NodeHandle n;
    // Add your ros communications here.
    chatter_publisher = n.advertise<geometry_msgs::Twist>("/cmd_vel", 1);
    start();
    return true;
}
```

Illustration 18: Fonction d'initialisation du nœud ROS

```
void QNode::run() {
    ros::Rate loop_rate(5);
    while ( ros::ok() ) {
        chatter_publisher.publish(commande); // Publie la commande
        ros::spinOnce();
        loop_rate.sleep();
    }
    std::cout << "Ros shutdown, proceeding to close the gui." << std::endl;
    Q_EMIT rosShutdown(); // used to signal the gui for a shutdown (useful to roslaunch)
}
```

Illustration 19: Thread du nœud ROS