# Capítulo 4

# Interfaz Romeo HMI

## 4.1. Introducción

En primer lugar hemos de determinar el uso específico de la aplicación. La interfaz gráfica Romeo HMI se utiliza para controlar y monitorizar el vehículo autónomo Romeo-4R. Está orientada a facilitar el uso del robot durante la realización de experimentos de navegación y guiado, por lo cual lo primero que debemos destacar es que no se trata de una interfaz hombre máquina orientada para un usuario final, sino que viene a satisfacer algunos problemas que la experiencia ha demostrado que pueden resultar muy útiles en cuanto tiempo y facilidad de uso para los desarrolladores de Romeo-4R.

Es por ello, que desde la aplicación se tiene acceso a los datos provenientes de los múltiples sensores que dispone el robot, lo cual pueden servir de orientación para saber que está viendo en cada momento el robot, o por ejemplo para comprobar si se está produciendo algún error excesivo en la localización del mismo.

También se creó un mecanismo de introducción de las trayectorias que debe seguir el robot, algo que facilitó mucho el anterior mecanismo basado en la generación de ficheros de texto plano, así como un mecanismo para lanzar y detener los distintos módulos de funcionamiento que han sido explicados en el capítulo 2 de este texto.

Comentar por último que los aspectos gráficos de la interfaz, como el tamaño de los botones y widgets por ejemplo, han sido diseñados teniendo en cuenta el modelo de pantalla táctil que utiliza Romeo-4R, que se corresponde con una pantalla táctil de 15 pulgadas de la serie ET1515L del fabricante ELO, así como también la resolución utilizada por defecto de *1024 x 768 píxeles*.

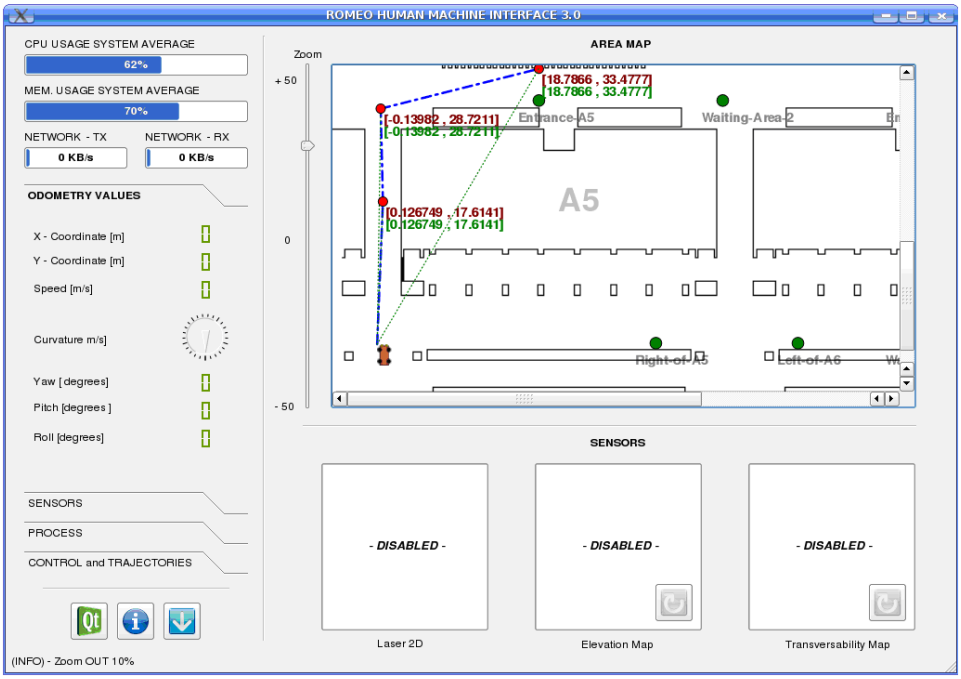


**Figura 4.1: Interfaz gráfica Romeo HMI.**

## 4.2. Definición de clases principales

Veremos a continuación una descripción de las clases principales implementadas en la aplicación así como una descripción funcional de las mimas. Los diagramas que aparecen en esta sección han sido creados con la herramienta de generación automática de documentación Doxygen [3].

### 4.2.1. RomeoMainWindow

Sin duda RomeoMainWindow es la clase principal de la aplicación. Partimos de la clase base QMainWindow, que nos proporciona el marco para la creación de la interfaz de usuario de la aplicación.

En Qt disponemos de QMainWindow y sus clases relacionadas para la gestión de la ventana principal. QMainWindow tiene su propio diseño a la que puede agregar QToolBars, QDockWidgets, un QMenuBar e incluso una QStatusBar.

De hecho, el main.cpp de la aplicación se reduce a algo tan sencillo como lo recogido en la tabla 4.1.

Donde como vemos simplemente se genera un objeto de la clase QApplication,

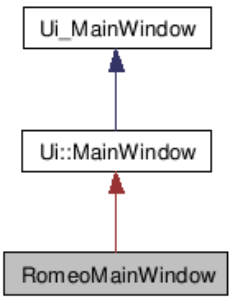


**Figura 4.2: Diagrama de Herencia de clase RomeoMainWindow.**

que gestiona el flujo de la aplicación con interfaz gráfica, así como los ajustes principales y se genera la pantalla principal de la aplicación.

La clase QApplication contiene el bucle de eventos principal, donde todos los eventos del sistema de ventanas y otras fuentes son procesados y enviados. También maneja la inicialización de la aplicación y finalización de la misma, proporciona la administración de sesiones y gestiona todos los ajustes de la aplicación.

Una vez generada la pantalla principal, mediante la función exec() entramos en el bucle de eventos principal una vez ha sido generada como decimos la pantalla principal. Es necesario llamar a esta función para iniciar la gestión de eventos.

```
1  #include <QApplication>
2  #include "romeomainwindow.h"
3
4  int main(int argc, char *argv[])
5  {
6          Q_INIT_RESOURCE(application);
7          QApplication app(argc, argv);
8          new RomeoMainWindow();
9          return app.exec();
10 }
```

**Tabla 4.1:** ***Proyecto URUS – main.cpp.***

El bucle de eventos principal recibe los eventos del sistema de ventanas y los redirige a los distintos widgets de la aplicación.

Parte de la definición de la clase RomeoMainWindow se recoge en la tabla 4.2, simplemente para mostrar los elementos más significativos. Recogemos aquí las variables y funciones principales de la clase, incluyendo la definición de los puer-

tos YARP a utilizar, la definición de un objeto de la clase MapScene (ver apartado 4.2.2) así como algunas de las variables de control utilizadas (ver apartado 4.3).

La definición de los widgets que componen la interfaz, así como su disposición en los distintos frameworks de la misma han sido generados mediante el uso de la aplicación visual QtDesigner, basada en el método arrastrar y soltar, que guarda el resultado un fichero con extensión .ui (User Interface), el cual es simplemente un fichero de texto plano en formato XML, del que recogemos un ejemplo (extracto) en la tabla 4.3.

Utilizando el llamado enfoque de herencia múltiple, todos los componentes de interfaz de usuario definido en la ventana son directamente accesibles en el ámbito de la subclase, y permite conexiones de forma habitual mediante señales y slots. Únicamente tenemos necesidad de incluir el encabezado del archivo que genera el UIC (user interface compiler) desde el archivo con extensión .ui, que en nuestro caso se recoge en la tabla 4.4.

Finalmente el constructor de la subclase es el que realiza las tareas de generación de la interfaz y colocación de los widgets y la configuración de los atributos de los mismos.

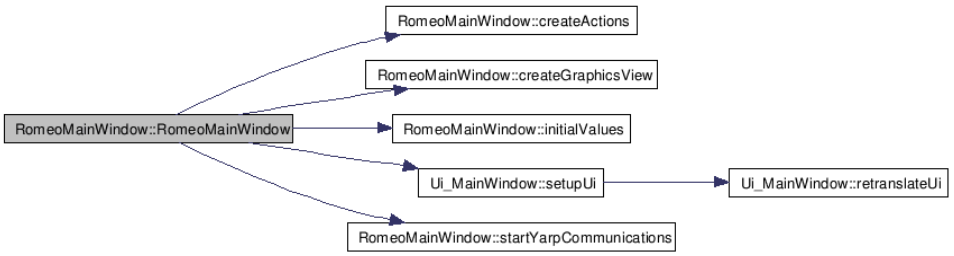


**Figura 4.3: Llamadas desde el constructor RomeoMainWindow.**

Este procedimiento es el que se utiliza también para la generación del resto de formularios que se utilizan en la aplicación, tal y como se recoge en la figura 4.4.

### 4.2.2. MapScene

La clase MapScene es una clase derivada de QGraphicsScene, implementada para mejorar la gestión de mensajes al realizar ciertos eventos sobre el área gráfica interactiva que muestra el mapa y la localización del robot (ver sección 4.4). La definición de la clase se recoge en la tabla 4.6.

La figura 4.5 nos muestra el gráfico de dependencias en el fichero mapscene.cpp.

```
class RomeoMainWindow:public QMainWindow,private Ui::
   MainWindow
{
        Q_OBJECT

        public:
                RomeoMainWindow(QWidget *parent=0,Qt::
                   WFlags fl = 0);
                ~RomeoMainWindow();

        signals:
                void  message(QString); // Info Text
                   Message Signal

        protected:
                void closeEvent(QCloseEvent *event);

        private slots:
                void exitApplication();
                void openMap();
                void loadMap(QString mapName);
            ...

        private:
                void createActions();
                void initialValues();
                void createGraphicsView();
                void startYARPCommunications();
                QProcess *LAUNCHER_Process;
                QString LAUNCHER_Program;
                QString LAUNCHER_Directory;
                MapScene scene;
                QPixmap map;
                ScaleDialog MapScale;
                trajectoryDialog SelectTrajectory;
                bool trajectory_edit_mode;
                bool trajectory_executing_mode;
                bool setting_initial_point;
                bool theta_edit_mode;
                bool trajectoriesBYinterface;
                bool yarpRunning;
                CDcxData *pyarpDcx;     // Fron DCX
                CLocData *pyarpEkf;     // From EkFloc
                yarp::os::BufferedPort<CLocData> locInput;
                yarp::os::BufferedPort<CDcxData> odomInput;
            ...
}
```

**Tabla 4.2:** ***Proyecto URUS – romeomainwindow.h.***

```
class Ui_MainWindow
{
public:
    QAction *actionAboutQT;
    QAction *actionAboutRomeoHMI;
    QAction *actionOpenMap;
    QAction *actionOpenTrajectory;
    QWidget *centralwidget;
    QToolBox *toolBox;
    QWidget *page;
    QLabel *XcoordinateLabel;
    QLCDNumber *YcoordinatelcdNumber;
    QLabel *PITCHlabel;
    QLCDNumber *XcoordinatelcdNumber;
    ...

    void setupUi(QMainWindow *MainWindow)
    {
            if (MainWindow->objectName().isEmpty())
            MainWindow->setObjectName(QString::fromUtf8("
               MainWindow"));
            MainWindow->setWindowModality(Qt::WindowModal);
        MainWindow->resize(1024, 695);
            actionAboutQT = new QAction(MainWindow);
            ...
            retranslateUi(MainWindow);
            toolBox->setCurrentIndex(3);

        QMetaObject::connectSlotsByName(MainWindow);
        } // setupUi
        ...
};
namespace Ui {
    class MainWindow: public Ui_MainWindow {};
} // namespace Ui
```

**Tabla 4.3:** ***Proyecto URUS – ui-MainWindow-HMI2.h.***

```
#include "ui_MainWindow_HMI2.h"
```

**Tabla 4.4:** ***Proyecto URUS – ui-MainWindow-HMI2.h.***

```
RomeoMainWindow::RomeoMainWindow(QWidget* parent, Qt::
    WFlags fl):QMainWindow(parent, fl), Ui::MainWindow()
{
        setupUi(this);
        initialValues();
        createActions();
        createGraphicsView();
        check_YARP_network();
        startYARPCommunications();
        statusbar->showMessage(tr("(INFO) - Ready"));
}
```

**Tabla 4.5:** ***Proyecto URUS – romeomainwindow.cpp.***

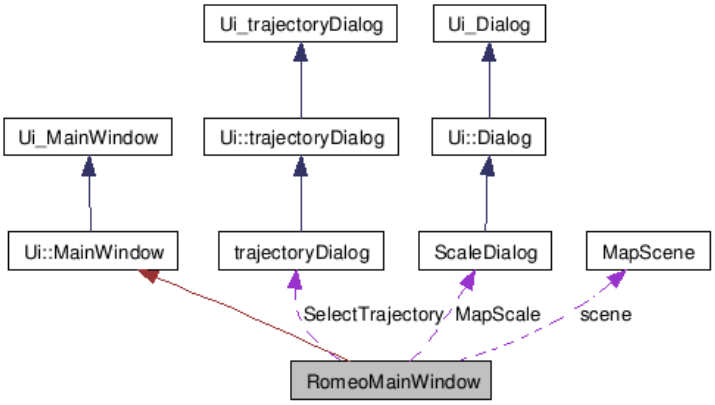


**Figura 4.4: Diagrama de colaboración de clase RomeoMainWindow.**

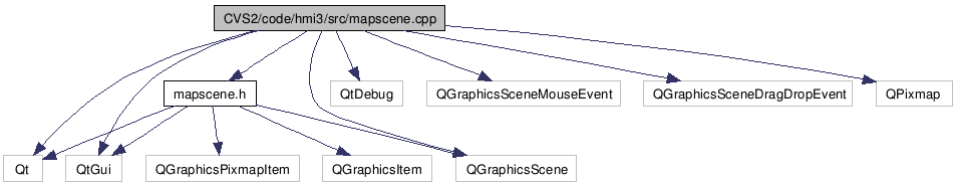


**Figura 4.5: Gráfico de dependencias fichero mapscene.cpp.**

```
1  class MapScene : public QGraphicsScene
2  {
3          Q_OBJECT
4
5  public:
6          MapScene();
7          ~MapScene();
8
9  signals:
10         void messagemap(QString);
11         void mapScenePressEvent(QPointF clickedPoint);
12         void changeRobot(QPoint);
13         void ungrabRobot();
14
15 protected:
16
17         void mousePressEvent (QGraphicsSceneMouseEvent *
               event);
18         void dragMoveEvent (QGraphicsSceneDragDropEvent *
               event);
19         void dropEvent (QGraphicsSceneDragDropEvent * event
               );
20 };
```

**Tabla 4.6:** ***Proyecto URUS – mapscene.h.***

No habría sido estrictamente necesario la realización de esta clase, pues podría haberse implementado su funcionalidad con la clase genérica QGraphicsScene, sin embargo se mantuvo en el desarollo para implementar el sistema de paso de mensajes a la barra de estado de la aplicación.

## 4.3. Variables

Se han utilizado mútiples variables a lo largo de la aplicación. Vamos a realizar una clasificación de las mismas según su utilidad. Distinguimos varios grupos, entre ellos:

- **Variables para sensores robot Romeo-4R**

  Las variables de la tabla 4.7 recogen la información de localización del robot en en el mundo real, relativas siempre al mapa de la zona, al denominado en el capítulo 2 como World Coordinate System (WCS).

```
double theta;
double previous_theta;
double theta_offset;
double locale_romeo_x_ref;
double locale_romeo_y_ref;
double romeo_speed_reference;
```

**Tabla 4.7:** ***Romeo HMI – Variables locales de localización del robot.***

Mientras que las variables de la tabla 4.8 recogen la información de localización que el módulo EKFLOC exporta y se corresponden con la información del sistema de referencia local del robot Romeo Coordinate System (RCS).

```
double x_ref;
double y_ref;
double orient_ref;
double curv_ref;
double previous_orient_ref;
```

**Tabla 4.8:** ***Romeo HMI – Variables de localización del robot - módulo EKFLOC.***

Otros valores de odometría procedentes de las lecturas de los encoders en las ruedas que nos dan la velocidad (exportado por el módulo DCX) así como las estimaciones de los ángulos de Tait-Bryan, Yaw, Pitch y Roll (exportados

```
double speed_ref;        // From DCX
double yaw_ref;          // From IMU
double pitch_ref;        // From IMU
double roll_ref;         // From IMU
```

**Tabla 4.9:** ***Romeo HMI – Variables odométricas del robot - módulos IMU y DCX.***

por el módulo IMU).

- **Variables de la vista gráfica correspondiente al mapa**

  Estas variables contienen la información necesaria para localizar al robot en el mapa así como para poder escalar el mismo con sus dimensiones reales, pudiendo mostrar en cada instante la estimación de la posición del robot en el mapa.

```
double map_width;        // in Pixels
double map_height;       // in Pixels
double map_x_axis_length;        // in meters
double map_y_axis_length;        // in meters
double map_X_scale;      // Pixels/meter
double map_Y_scale;      // Pixels/meter
double scene_x_axis_length;
double scene_y_axis_length;
double x_ini_global;     // Romeo Initial Position
double y_ini_global;
```

**Tabla 4.10:** ***Romeo HMI – Variables vista gráfica del mapa.***

- **Variables Estado de Sistema**

  Con los valores de estas variables se va actualizando los datos de la información del sistema, carga de CPU, Memoria y estadísticas de red, que aparecen en la parte superior del frame izquierdo de la aplicación (ver sección 4.4).

- **Variables Estado Aplicación**

  La aplicación sigue un diagrama de estados de funcionamiento en función de los valores de estas variable binarias, que inicialmente tienen todos sus valores a FALSE.

```
double m_stats[4];      // Values from /proc/stat
double m_stats_new[4];  // For CPU Usage evaluation
double tx_net_stats;    // Values from /proc/dev/net
double tx_net_stats_new;
double rx_net_stats;    // Values from /proc/dev/net
double rx_net_stats_new;
```

**Tabla 4.11:** ***Romeo HMI – Variables Monitor Estado del Sistema.***

```
bool trajectory_edit_mode;
bool trajectory_executing_mode;
```

**Tabla 4.12:** ***Romeo HMI – Variables estado aplicación.***

El diagrama de estados de funcionamiento de la aplicación, recogido en la figura 4.6 es en realidad muy sencillo. Tras un cuasi estado de inicialización de todo el sistema (estado 0), se entra en el modo general de espera (estado 1).

Utilizando los controles de trayectoria entramos en el estado de introducción de trayectoria (estado 2). Si se cancela la trayectoria se vuelve al estado de espera (estado 1).

Una vez preparada la trayectoria se envía al módulo PURE PURSUIT entrando en modo de ejecución de trayectoria (estado 3).

Una vez la trayectoria ha sido ejecutada o si bien se interrumpe la ejecución de la misma se vuelve al modo general de espera (estado 1).

| ESTADO | NOMBRE | VARIABLES ACTIVADAS |
|---|---|---|
| 0 | Inicialización del Sistema | Ninguna |
| 1 | Modo de Espera | Ninguna |
| 2 | Modo Introducción de Trayectoria | trajectory_edit_mode |
| 3 | Modo Ejecución de Trayectoria | trajectory_executing_mode |

**Tabla 4.13: Romeo HMI – Tabla de estados aplicación.**

Por tanto vemos que el modo de uso de la aplicación tiene un comportamiento sencillo, totalmente cíclico y secuencial.

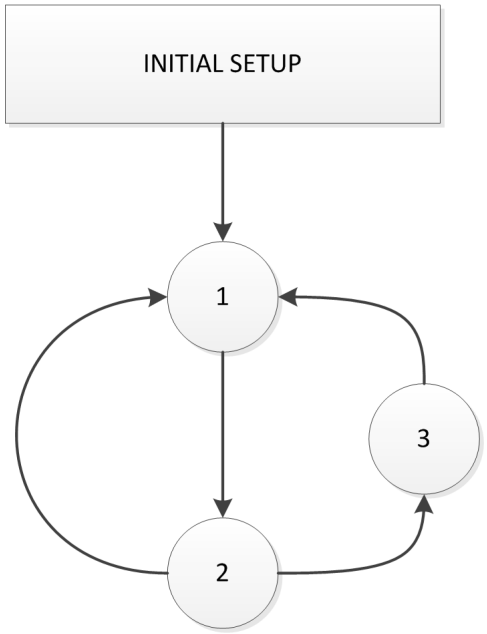


**Figura 4.6: Diagrama estados funcionamiento aplicación Romeo HMI.**

## 4.4. Descripción de la interfaz gráfica

La interfaz gráfica se estructura en tres frames principales, tal y como vemos en la 4.7.

1. **Frame Izquierdo.** Dividido a su vez en dos segmentos. El primero de ellos se dedica a mostrar información del sistema que está ejecutando la propia aplicación, que por lo general ejecutará también la mayor parte de los módulos que componen el sistema. Es por tanto un monitor del estado del sistema, mostrando promedios en tiempo real de datos de consumo de CPU, uso de memoria del sistema y utilización de red (transmisión y recepción).

   El segundo segmento está formado por un widget QToolBox, que nos permite ordenar diferentes widgets en cada una de las pestañas que lo componen, mostrando siempre los widgets bajo la pestaña seleccionada en cada instante. Se han asignado cuatro ventanas, con los siguientes identificadores:

   - *ODOMETRY VALUES*
     Bajo esta pestaña se muestran los valores más significativos de la odometría y de algunos sensores del vehículo.

     Se muestra la posición en el plano estimada del robot, la velocidad instantánea y el ángulo de giro del volante, así como los ángulos Yaw,

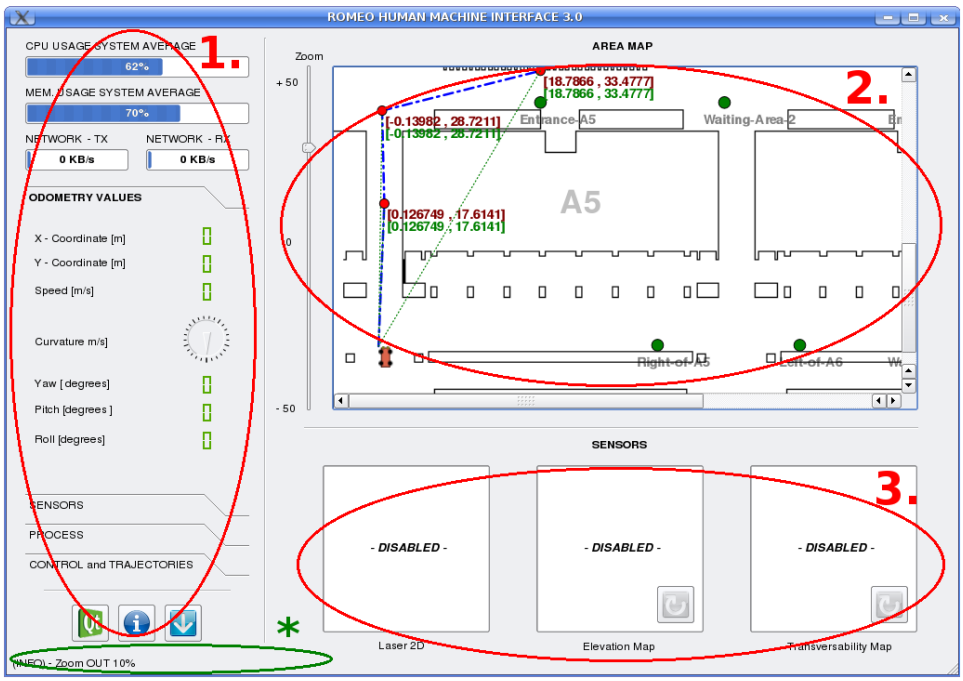


**Figura 4.7: Frames aplicación Romeo HMI.**

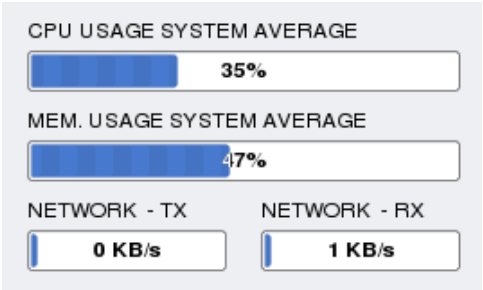


**Figura 4.8: Monitor del sistema.**

Pitch y Roll.


**Figura 4.9: QToolBox Frame izquierdo.**

Toda esta información procede de los puertos YARP correspondientes a los módulos DCX, EKFLOC e IMU. La información se actualiza cada segundo, aunque el intervalo de actualización es modificable.

- *SENSORS*
  Pestaña que contiene los controles que permiten la visualización de los sensores indicados en el área de visualización correspondiente (Frame Inferior Derecho).

  La visualización de los sensores en tiempo real es comprometida para el rendimiento general del sistema, al aumentar considerablemente el flujo de datos a transmitir por la red YARP, ya que se deben generar y transmitir las imágenes que capta el módulo Laser 2D o bien las que generan los mapas de elevación y de transversabilidad.

- *PROCESS*
  En esta pestaña aparece el controlador de arranque y parada de un script generado para inicializar secuencialmente todos los módulos de control, sensorización y navegación que componen el cerebro de Romeo-4R que fueron explicados en el capítulo 2.


**Figura 4.10: Controles visualización de los sensores.**


**Figura 4.11: Controles del Launcher.**

Estos controles sirven pues para iniciar y detener el funcionamiento del robot como tal.

- *CONTROL and TRAJECTORIES*
  Esta pestaña contiene los controles para la generación y el paso de trayectorias al robot.

  Se ha creado un mecanismo de introducción de waypoints secuenciales donde una vez escrita la trayectoria se pasa al módulo correspondiente por un puerto YARP.

  Se han creado mecanismos para indicarle al robot su posición y orientación inicial en el mapa, así como métodos para eliminación y carga tanto de trayectorias como de mapas.

  Al cargar un nuevo mapa (imagen bmp) aparece una ventana emergente que nos permite calibrar el mismo, es decir, hacer la correspondencia entre píxeles – metros.

  Esta pestaña incluye un botón de parada de emergencia por software que detiene todos los procesos de control del robot provocando su parada cuasi inmediata.


**Figura 4.12: Controles de inserción de trayectoria.**

El frame izquierdo finaliza con unos iconos donde se muestran accesos directos a los créditos de Qt, créditos de la aplicación así como una opción de salida de la aplicación, respectivamente.

**Figura 4.13: Accesos directos frame izquierdo.**

2. **Frame Derecho Superior.** En esta zona de la aplicación se ha incluido el mapa de la zona y los controles de zoom de la misma. El mapa cargado se muestra en una clase MapScene que es totalmente interactiva.

   En tiempo de ejecución, se va mostrando la posición actual del robot y puede superponer los mapas de elevación y de transversabilidad que se van generando (con el coste computacional que ello supone).

   En el modo de introducción de trayectoria la zona del mapa responde de forma interactiva según la opción que le hayamos indicado, mostrando los próximos waypoints o mostrando la localización del robot.

3. **Frame Derecho Inferior.** En la zona inferior derecha de la pantalla se muestra la información proveniente del laser frontal (Laser 2D) lo cual sirve de

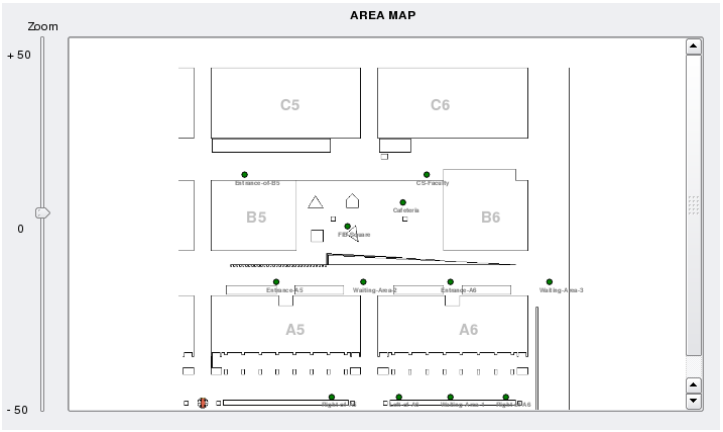


**Figura 4.14: Zona de visualización mapa (I).**

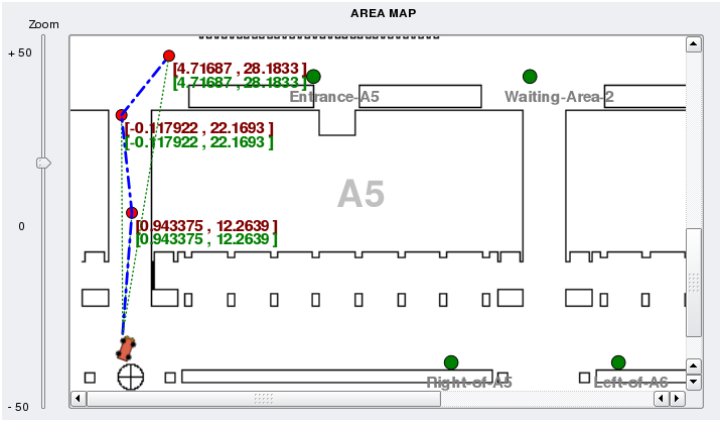


**Figura 4.15: Zona de visualización mapa (II).**

utilidad para *comprobar qué está viendo el robot* así como para mostrar los mapas de elevación y de transversabilidad del entorno que Romeo-4R va generando en tiempo real.


**Figura 4.16: Sensores Romeo HMI.**

La información a mostrar es seleccionable y puede ser integrada con la zona de visualización del mapa.

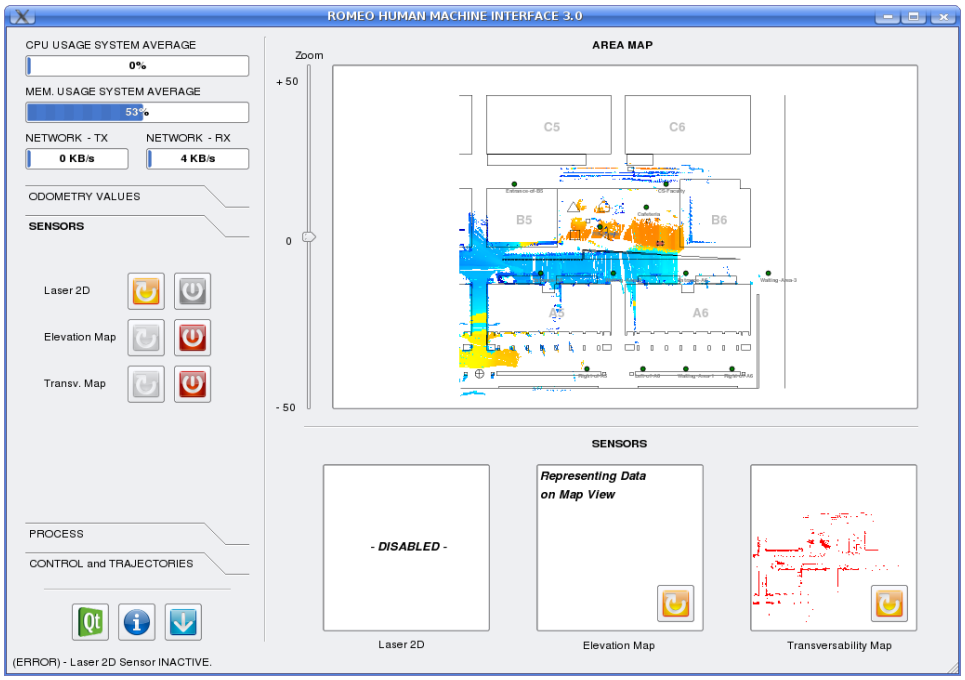


**Figura 4.17: Mapa de elevación integrado en la visualización del mapa.**

Destacar también que la aplicación cuenta con una barra de estado donde van apareciendo los acontecimientos más significativos y distintos mensajes de información, warning o error, que se van produciendo en tiempo de ejecución.