Universidade de Coimbra

Faculdade de Ciências e Tecnologia

Departamento de Engenharia Informática

Interface amigável para colocação em serviço e supervisão de um robô de desminagem

Mestrado em Engenharia Informática

Estágio 2014/2015 - Relatório Final

João Miguel Ventura Sobral
jsobral@student.dei.uc.pt
Estagiário
Data: julho de 2015

Lino Marques
Orientador – ISR

José Prado
Orientador – ISR

DEI
DEPARTAMENTO ENGENHARIA INFORMÁTICA

INSTITUTO DE SISTEMAS E ROBÓTICA
UNIVERSIDADE DE COIMBRA

João Miguel Ventura Sobral
jsobral@studnet.dei.uc.pt
Estagiário

Lino Forte Marques
Orientador – ISR

José Augusto Soares Prado
Orientador – ISR

António Dourado Pereira Correia
Orientador - DEI

César Alexandre Domingues Teixeira
Júri Arguente

Raul Barbosa
Júri Vogal

Resumo

As guerras da era moderna não se limitam a confrontos bélicos entre países ou grupos militares organizados. Muito depois de estes conflitos terminarem ficam ainda resquícios belígeros no terreno que afetam populações e civis. As minas terrestres são um flagelo atual, continuam a ser usadas apesar das restrições internacionais e a sua remoção eficaz, segura e definitiva é um desafio contemporâneo.

Neste sentido, estão em funcionamento e desenvolvimento máquinas, robôs e outros veículos não tripulados com o propósito de localizar, desarmar ou até mesmo acionar e explodir minas terrestres ou outros engenhos. Um desses veículos é o FSR Husky, robô autónomo de procura de minas terrestres desenvolvido no Instituto de Sistemas e Robótica da Universidade de Coimbra.

Este estágio teve por objetivo desenvolver uma interface gráfica para a colocação em serviço e supervisão do FSR Husky, facilitando a interação entre homem e máquina de forma a proporcionar uma desminagem segura e simples para o utilizador do robô. Estes objetivos foram cumpridos no decurso deste trabalho, sendo a operação do robô através da nova interface gráfica fácil e eficaz, com franca melhoria na interação com o robô.

Palavras-chave

"*Autonomous Demining Robots" "Friendly Robot Interfaces" "FSR Husky Interface" "Humanitarian Demining" "Human Robot Interface" "Interfaces for Unmanned Search Vehicles" "Teleoperation of Search Vehicles"*

Abstract

Modern era has brought wars that no longer limit themselves to violent confrontation between countries or military groups. Long after the ending of these conflicts, warlike materials dangerous to populations and civilians remain. Landmines are a current calamity in many countries, their continued production despite international regulation, and its efficient and safe removal is a present-day challenge.

In this regard, machines, robots and other unmanned vehicles are being used and developed with the purpose of locating, disarming or even actively removing landmines and other unexploded ordnance by controlled explosion. One of these vehicles is the FSR Husky, an autonomous landmine search robot developed in the Institute of Systems and Robotics from the University of Coimbra.

This internship intended to develop a graphical user interface for the commissioning and supervision of the FSR Husky tasks, easing the interaction between man and machine in order to provide a safe and simple solution to the robot usability. Robot operability through the developed graphical interface was achieved with ease and efficacy, having greatly improved the interaction with the robot.

Agradecimentos

Quero agradecer aos meus pais e restante família que me proporcionaram as oportunidades necessárias para chegar a esta fase da minha vida académica e pelo apoio incessante em toda a minha vida. Em especial à minha mãe, Alexandra Ventura, que auxiliou e ajudou a resolver problemas burocráticos com a Universidade de Coimbra bem como na revisão deste e de outros documentos deste estágio.

Aos meus orientadores, Professor Lino Marques e José Prado, pela sua valência e conhecimento na área em questão, tal como a sua disponibilidade para o acompanhamento permanente do estágio e do trabalho desenvolvido, tal como a sua inestimável contribuição para o mesmo, o qual não teria sido possível sem a sua presença.

Ao professor António Dourado, pela sua orientação e ajuda na conceção deste documento.

Aos colegas no Instituto de Sistemas e Robótica, Baptiste Gil e Professor Sedat Dogru, os quais promoveram a integração no projeto do Husky e não pouparam esforços para que este estágio chegasse a bom porto. Também ao colega Sérgio Filipe que, participando num estágio semelhante, foi um parceiro capaz e colaborador nas partes coincidentes dos nossos estágios.

A todas as outras pessoas que direta ou indiretamente contribuíram para a produção e execução deste projeto.

Índice

Figuras

Glossário

Acrónimo	Significado
LED	Díodo emissor de luz (*Light Emiting Dyode*)
GPR	Radar de penetração (*Ground Penetrating Radar*)
MT	Minas Terrestres
EEND	Engenhos Explosivos Não Detonados
MA	Domínio da ajuda humanitária ligada ao desenvolvimento de atividades que reduzam os problemas socias, económicos e ambientais ligados às MT e EEND. (*Mine Action*)
GPS	Sistema de posicionamento global (*Global Postitioning System*)
RTK	Técnica para melhoramento da precisão do GPS (*Real Time Kinematic*)
GUI	Interface gráfica do utilizador (*Graphical User Interface*)
XML	*Extensible Markup Language*
HTML	*HyperText Markup Language*
JS	*Java Script*. Linguagem de programação dinâmica para tecnologias da internet.
Qt	Plataforma de desenvolvimento Qt.
API	*Application Programming Interface*. Interface para aceder a serviços.
ISR	Instituto de Sistemas e Robótica
DEEC	Departamento de Engenharia Eletrotécnica e de Computadores
FCTUC	Faculdade de Ciências e Tecnologia da Universidade de Coimbra
FIFO	*First In First Out*
ROS	Conjunto de bases de *software* para o desenvolvimento de robôs e seus controladores (*Robot Operating System*)
RVIZ	(e Gazebo) Ferramenta de Virtualização 3D para o ROS
IMU	Unidade de cálculo de inércia (*Inertial Measurement Unit*)
IDE	Ambiente de desenvolvimento de *software* (*Integrated Development Environment*)
CE	Comissão Europeia
MIT	*Massachusetts Institute of Technology*
UED	*User Environment Design*

Capítulo 1 - Introdução

As minas terrestres são ainda um flagelo em muitas partes do mundo, onde continuam por detonar cerca de sessenta[1] a cem milhões colocadas no terreno, e, pelo menos, outras cento e dez milhões de unidades em armazém[2]. Estes números no entanto não são exatos, pois muitas vezes os países não divulgam, ou não divulgam com verdade, os seus números. Existem estudos que revelam números muito superiores aos oficiais, com relatos de mais de cem milhões de minas anti pessoais[3] em posse da Republica Popular da China e dos Estados Unidos da América. Apesar das mortes associadas a minas terrestres estar a diminuir nos últimos anos (cerca de um terço de 1999 a 2013), os feridos ultrapassam os vinte e cinco mil por ano[4]. O progresso de remoção ou explosão controlada dos engenhos nas áreas afetadas é extremamente lento, sendo que apenas alguns milhares de minas são desativadas por ano[3]. A disseminação destes engenhos por grandes áreas, alem de colocar em perigo a população civil, impede também o acesso a terra arável e aos seus recursos naturais, este facto atrasa o desenvolvimento destas zonas destruídas pela guerra e impossibilita a sua recuperação social e económica[3].

A desminagem é um trabalho altamente perigoso: para cada cinco mil minas desativadas ou destruídas, uma pessoa morre e outras duas ficam feridas[5]. É aqui que os robôs e máquinas autónomas de deteção e remoção de engenhos explosivos não detonados e minas terrestres podem mostrar a sua importância no panorama geral deste flagelo. São métodos de procura mais seguros, com menores custos para a vida humana e que podem cobrir mais terreno autonomamente.

O Instituto de Sistemas e Robótica da Universidade de Coimbra, em parceria com vinte e duas outras entidades faz parte do projeto "TIRAMISU - *Toolbox Implementation for Removal of Anti-personnel Mines, Submunitions and UXO"*[6]. Com financiamento da Comissão Europeia, este projeto tem como missão prover ferramentas que construam uma fundação para todas as ações de desminagem supervisionadas pela CE. Estas ferramentas serão desenvolvidas pelos parceiros, em ligação com utilizadores finais, e testadas em ambientes reais para depois serem postas à disposição das atividades de *Mine Action* (MA).

O FSR Husky é um robô autónomo de procura, inserido no projeto TIRAMISU, montado e caracterizado no ISR, baseado no Clearpath Husky A200[7], uma base de desenvolvimento de veículos autónomos desenvolvido pela "Clearpath Robotics" no Canadá que permite a equipas de desenvolvimento adaptarem e montarem as suas funcionalidades em cima deste suporte. Partindo do A200, o grupo de trabalho do ISR construiu o FSR Husky[8], com uma série de sensores (recebem informação do ambiente) e atuadores (efetuam ações sobre o ambiente) que permitem a navegação do robô no terreno através de GPS, camaras de visão por computador, laser que permite construir uma imagem 3D do ambiente em volta do veículo, e um detetor de metais montado no braço mecânico que se estende do corpo principal do Husky que permite a deteção segura de qualquer metal que possa pertencer a minas anti pessoais ou explosivos não detonados (como também falsos positivos)[8].

No entanto, o Husky não possui nenhum tipo de interface gráfico para o seu uso final, dependendo de algumas ferramentas presentes no ROS como o rviz e gazebo (simulação virtual), para o seu desenvolvimento, o que não substitui a necessidade definitiva de ter uma interface simples, usável, e de acordo com as características do utilizador final. É neste contexto que este estágio se insere e a partir do qual se constituiu o seu objetivo.

O propósito deste estágio foi construir uma plataforma de interface gráfica que facilite a supervisão e o controlo do Husky para que a interação entre o operador e o robô possa ser feita sem conhecimentos técnicos de programação, LINUX ou outras tecnologias usadas no projeto FSR Husky que eram ainda necessárias para colocar o veículo em funcionamento à data do início do estágio.

Capítulo 2 - Estado da Arte

2.1. Interface Homem-Máquina

A supervisão, ação de observação sobre as tarefas de uma máquina ou veículo, e controlo, atuação direta no robô em forma de ordens ou tarefas, é um tema que vem desde os primórdios de máquinas e robôs autónomos criados na industrialização da nossa sociedade [9]. O crescimento destas tecnologias leva-nos a pensar e a dissertar sobre o que deve ser e como deve ser a interação com um robô, um veículo autónomo, uma sonda espacial, submarina ou até mesmo com carros que se guiam a si próprios [10].

A utilização deste tipo de máquinas apresenta uma clara dicotomia entre a teleoperação e a autonomia. A teleoperação de robôs ou veículos não tripulados, com ligação direta a um operador, local, no caso de um carro telecomandado ou do nosso FSR Husky através de um comando, ou longínqua, no caso de *drones* de propósito militar consiste no controlo direto dos veículos através de ordens para os atuadores, onde todas as ações tomadas pela máquina advêm de controlo humano. Por outro lado, veículos autónomos, pré-configurados ou com inteligência própria (pré-programada) utilizam sensores, atuadores e outros meios de interação com o seu ambiente de modo a completar as suas missões sem intervenção humana, como sondas espaciais, máquinas industriais automáticas, ou o modo de procura do FSR Husky. Levando o exemplo ao extremo do estado da arte atual em termos de teleoperação com autonomia variável (robô com características autónomas mas com possibilidade de teleoperação de certas e predeterminadas tarefas), temos as sondas espaciais, que tanto podem ser controladas diretamente a partir da Terra, como executar as suas missões de reconhecimento, experiências científicas e análises ao terreno automaticamente [11], sem intervenção dos operadores que se encontram na Terra.

A latência, tempo que uma ordem leva a chegar do operador à máquina, atesta bem o problema de todos os robôs e veículos teleoperados à distância. Este é o maior entrave, e aquele que faz com que muitas soluções hoje em dia sejam mais autónomas, e menos dependentes de um operador, como demonstrado pelo estado do mercado dos veículos de desminagem e pelo domínio das máquinas blindadas autónomas. Dotado de inteligência artificial, sensores e capacidade de interpretação do ambiente à sua volta, um robô consegue atingir um nível de autonomia bastante elevado, mantendo sempre no entanto um certo controlo por parte dos operadores através da supervisão, e, nalguns casos, deter o poder de resgatar o comando, como por exemplo na aviação civil com o seu piloto automático de aeronaves [10]. O problema da latência pode ser resolvido através de autonomia variável, como também através da definição de missões mais breves, simples e diretas em vez de uma missão ou tarefa geral do veículo. Assim sendo, podemos enviar uma tarefa de movimentação do ponto A ao ponto B, deixando que o robô decida o melhor caminho a percorrer, e também, se tal for um propósito mais prioritário, parar para efetuar outras tarefas. Este balanceamento entre teleoperação e autonomia é essencial para maximizar a produtividade e aproveitar o tempo eficazmente, diminuindo os tempos mortos e mitigando os efeitos da latência e da distância entre operador e robô.

Sheridan [10] defende no seu livro que não se deve olhar para o problema em termos de Homem versus Máquina, classificando-os como simplistas, não produtivos e derrotistas, optando por argumentar que a cooperação entre Máquina e Homem é fundamental para o futuro desta indústria, chamando-lhe *Telerobotics* ou simplesmente Supervisão e Controlo. Dentro desta visão, o FSR Husky é um *telerobot*, tendo sensores, atuadores, decisão e controlo por inteligência artificial mas que recebe supervisão de um utilizador ou operador. Esta visão de Sheridan, ex-Professor de Engenharia e Aeronáutica no MIT, apesar da data de publicação, continua a ser pertinente, sendo que trabalhos recentes nesta área continuam a sustentar-se nesta e em outras bases do mesmo autor [12].

As funções de supervisão passam por: planear tarefas ou missões, passar essa informação ao *telerobot*, monitorizar as ações autónomas por forma a detetar problemas ou simplesmente assegurar o normal funcionamento do sistema, intervir no sistema, reduzindo a autonomia, e, acionando a teleoperação (caso seja necessário), aumentar a experiência de uso, e, por fim, melhorar as performances para o futuro.

A planificação das tarefas a desempenhar, a sua engenharia e execução prática, é a função mais difícil de modelar, pois requer que os desenvolvedores do sistema compreendam totalmente o ambiente onde o veículo vai operar, e o processo físico em que o robô está inserido. Ter o cuidado de definir objetivos atingíveis e que sejam mensuráveis para o sistema. Esta fase foi assegurada pelos membros do projeto FSR Husky no Instituto de Sistemas e Robótica anteriormente a este estágio. Sendo apenas necessário uma interface gráfica para ligar aos sistemas de planificação já desenvolvidos.

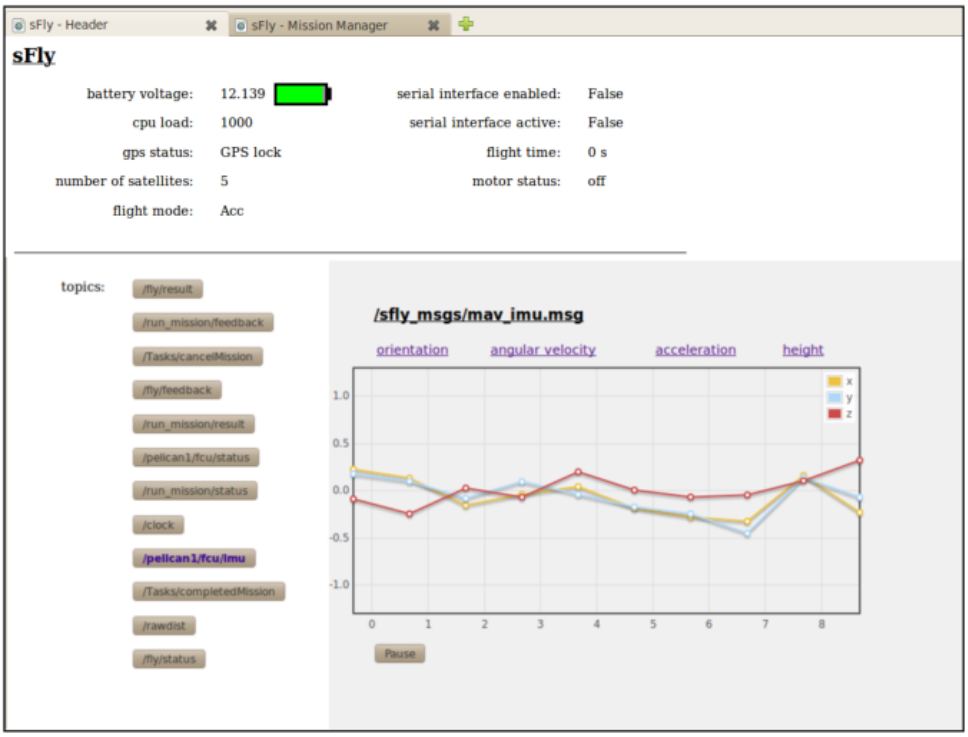


Figura 1 - Interface gráfico do estado do IMU do Sfly [13]

A monitorização, que é outra parte em que este estágio se focou e da qual temos um exemplo na figura 1, requer que o operador ou o utilizador do sistema robótico observe a performance, assegure o seu normal funcionamento, utilizando a sua visão, presença, ou, caso esteja a teleoperar o veículo, dados provenientes dos sensores do mesmo para observar e no caso de ser necessário, tomar decisões, atuar e controlar o robô. Isto é exatamente o que se verifica no FSR Husky, onde existe a possibilidade de fazer uma paragem de emergência durante uma tarefa de procura autónoma.

2.2. Veículos Autónomos de Reconhecimento de MT e EEND

Estes veículos autónomos ou de autonomia variável centram-se na descoberta, etiquetagem e marcação dos objetos encontrados. Sejam eles MT, EEND ou outro tipo de explosivos. Podem ser usados em espaços mais pequenos devido ao seu reduzido tamanho, em zonas de difícil acesso para máquinas de desbaste e também, podendo ser teleoperados, dispõem de uma maior diversidade de tarefas e missões [14].

Figura 2 - Alguns exemplos de robôs autónomos desminadores

Como se depreende da Figura 2, existem vários tipos de paradigmas para este tipo de robô de procura: robôs terrestres, híbridos ou exclusivamente aéreos. Estes aqui representados foram robôs que participaram na última edição do *Minesweepers: Towards a Landmine-free World*, competição coorganizada pela Universidade de Coimbra, através do ISR, Instituto de Sistemas e Robótica em Outubro de 2014 [15]. Patrocinada pela *IEEE Robotics & Automation Society*, teve mais de cem equipas subscritas, com vinte participantes finais [16].

Estes pequenos robôs autónomos possuem vários sensores e atuadores sofisticados, desde navegação por GPS, virtualização do ambiente a três dimensões, sistema de locomoção por patas, rodas e lagartas. No entanto, compreensivelmente, o foco do desenvolvimento está na descoberta e deteção de MT e EEND [17]. Se a descoberta e deteção forem executadas com grande grau de certeza, a posterior neutralização será mais segura e mais rápida. Apesar destes sistemas terem tido um bom impacto e um bom progresso em processos de desminagem e no avanço da tecnologia que a serve, não têm sido usados com a generalidade que se esperava. Isto deve-se principalmente a serem detetores de superfície, com problemas ao nível da descoberta no terreno devido a vegetação e profundidade das MT e dos EEND [17], fatores limitantes para o uso destes robôs sofisticados mas com algumas fragilidades.

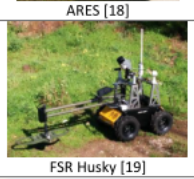
	Type	Autonomy	Cost	Dimensions (L×W×H)	Speed (m/s)	Mine Handling	Additional Capabilities	Operation Time	Payload
 Gryphon-IV [14]	Four-wheeler all-terrain vehicle (Yamaha GRIZZLY 660)	Remotely Controlled or Manually driven	≥ 30,000€	2.08×1.15× 1.20 (m) without the long-reach 3 m manipulator	Typical: 1.5 Max: 5.0	Non-metallic pantograph arm with 3DOF with GPR and MD	Terrain modeling, Stereo vision, RTK GPS, Long Range WiFi	10 h Gasoline (25 L tank)	170 kg
 SILO-6 [16]	Six-legged walking robot (hexapod)	Semi-Autonomous	≥ 15,000€	1.38×0.95× 0.76 (m) without the ≈1 m manipulator, 71.34 kg	Typical: 0.1 Max: 0.5	5 DOF manipulator with IR range sensors placed around the MD	EKF fusing DGPS and odometry, Radio Ethernet, Electronic compass	1 h DC Batteries	5.7 kg
 RMA tEODor [17]	Heavy Tracked Outdoor Platform	Semi-Autonomous	≥ 50,000€	1.3×0.7× 1.24 (m) without the MD system, 350 kg	Max: 0.8	Multi-Channel system with 5 independent MDs	Adjustable soil compensation, Bumpers, 4 cameras, Traversability analysis, Long Range WiFi	2-3 h Lead-gel recharge-able battery; 4 x 12 V, 85 Ah	27 kg max: 200 kg
 ARES [18]	Four independ-ently steered wheels vehicle	Semi-Autonomous	≥ 7,500€	1.36×1.51× 0.66 (m), 80 kg	Typical: 0.5 Max: 1.0	TNT detection using odor sensor	Different locomotion modes, DGPS, magnetometer, Long Range WiFi, Sonars, Stereo Vision	4 h Lead-Acid Batteries	N/D
 FSR Husky [19]	Four-wheeler all-terrain vehicle (Clearpath Husky A200)	Semi-Autonomous	≈ 23,000€	0.99×0.67× 1.35 (m) without the mine-clearance arm, 70 kg	Max: 1.0	2 DOF mine-clearance arm with MD and artificial nose	2 stereo cameras, Pan/Tilt LRF, RTK GPS with IMU, Long Range WiFi	3 h Sealed lead acid 24V, 20Ah	20 kg max: 75 kg
 LOCOSTRA [20]	4 wheeled agricultural mini tractor with articulated skid steering	Remotely Controlled	≈ 50,000€	3.5×1.6× 2.15 (m) without the excavation system, 1850 kg	Typical: 3.0 Max: 8.3	Landmine removal by excavation	AP blast resistant wheels, Vegetation cutters, 2 IP cameras, Long Range WiFi	Diesel (50 L tank)	max: 1800 kg

Figura 3 - Grupo de robôs desminadores onde se insere o FSR Husky

2.3. Análise de Interfaces Homem-Robô

Pegando num dos veículos da imagem anterior, o SILO-6 possui uma interface gráfica desenvolvida em JAVA, suportando a teleoperação do veículo, a visualização de dados em tempo real e a definição de missões [18]. A interface com o operador permite que se defina a tarefa de procura descrevendo o tamanho da área de procura, *waypoints* (pontos de passagem obrigatória) e estratégias de navegação. Esta interface está representada na figura seguinte:

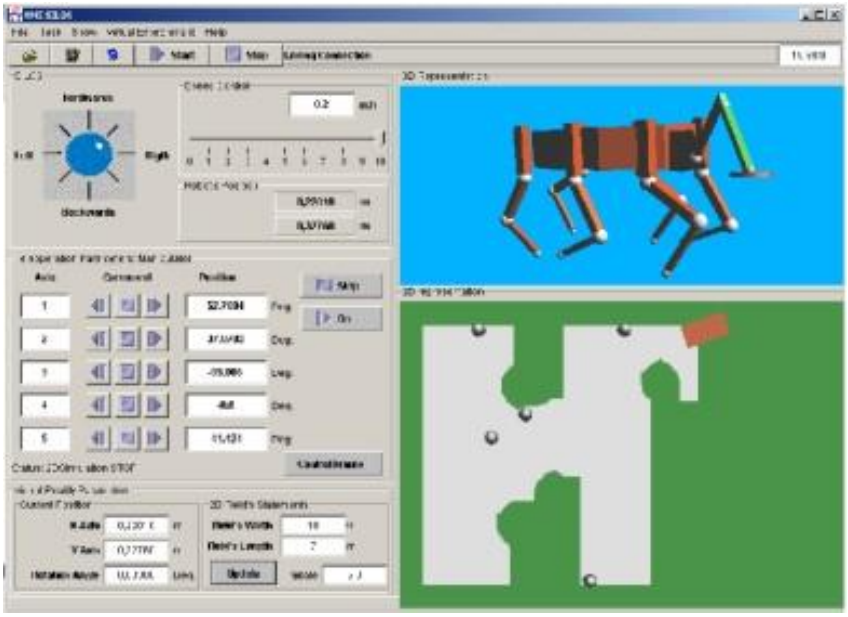


Figura 4 - Interface do SILO-6

Esta plataforma, apesar de um pouco crua, é bastante completa, além do estado do robô em tempo real apresentada em cima à direita, possui também uma vista aérea da área de procura, bem como os pontos de passagem. É possível discernir botões de Stop, ligar e desligar o veículo bem como o controlo dos seis eixos (patas) presentes na locomoção do robô.

Em seguida, temos um robô teleoperado movido a energia solar que tem como missão procurar fugas de gás. Criado por Jens Altenburg, este robô possui um *software* de controlo que o autor denomina por "Robot Control Center" [19] detém uma panóplia de botões e opções para permitir a teleoperação do veiculo. Podemos observar as imagens da câmara, gravar os vídeos por ela capturados, mudar de câmara e ajustar a luminosidade e contraste da imagem. Sobre o lado direito temos gráficos com cores para representar o estado do veículo, a qualidade do ar e a energia disponível.

Em baixo, à direita, os controlos de velocidade e as teclas para moverem o robô. Apesar da simplicidade do modelo, a interface está bem conseguida, tendo toda a informação importante disponível num só ecrã, no entanto torna-se bastante difícil controlar o robô

com eficácia, visto que o controlo de velocidade está dissociado dos comandos de movimento. Uma figura ilustrativa desta interface está em seguimento.

Figura 5 - Interface de um robô teleoperado

A Procerus Technologies apresenta o seu Virtual Cockpit como uma solução unificada para os seus produtos, veículos aéreos não tripulados. A interface é muito robusta, com dezenas de opções e configurações importantes para o utilizador, desde mapas automaticamente georreferenciados, dados de elevação do terreno, planeamento de missões, suporte para múltiplos veículos entre muitas outras características presentes no manual de referência [20]. As características mais importantes tendo em conta o contexto deste estágio é a georreferenciação das imagens, e a definição de missões, no caso de um veículo aérea, se limitam a pontos de passagem. É também útil observar nesta interface que as imagens ou vídeo captadas pelo robô compõem maior parte da janela, o que lhe confere bastante importância e se torna o foco central da interface.

De notar que é possível ligar e desligar sobreposições no mapa, dotando a interface e o utilizador com ferramentas de representação em camadas dos dados. Uma valência que deve ser tida em conta num projeto deste tipo.

A imagem da interface encontra-se na página seguinte.

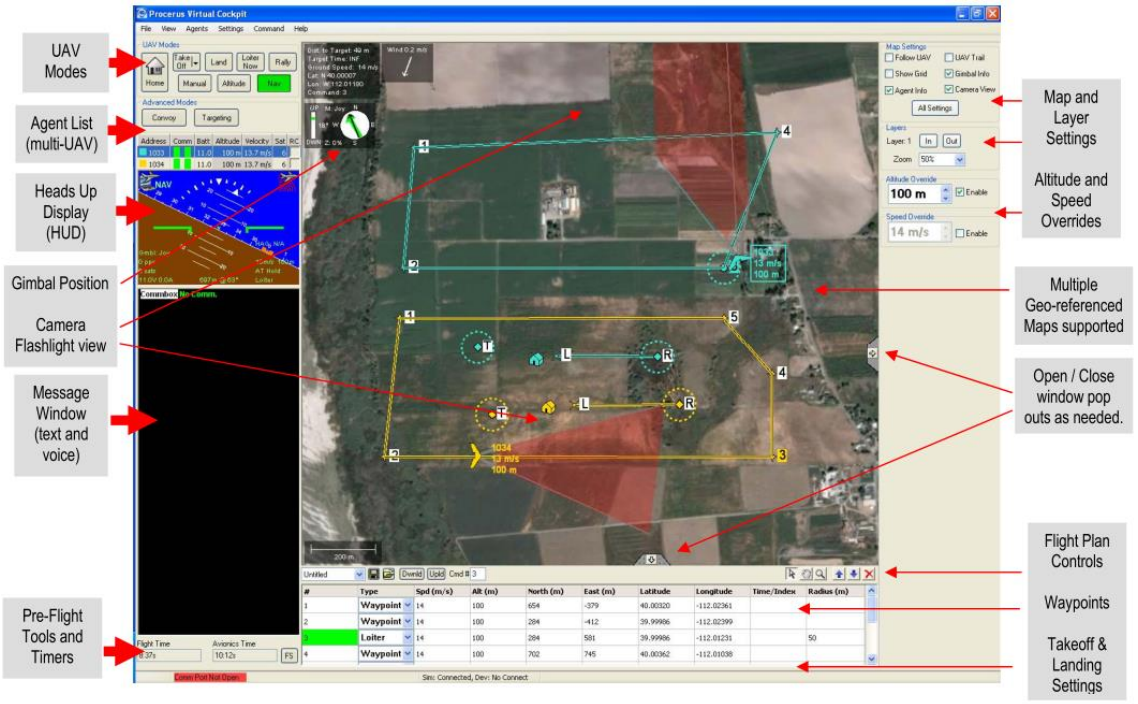


Figura 6 - Interface da Procerus Techonologies para os seus veículos aéreos

2.4. Estado da Arte do Interface do FSR Husky

O FSR Husky não está dotado de nenhuma interface direcionada para o utilizador final, todas as funções do robô requerem um conhecimento profundo do seu desenvolvimento e do ambiente onde se encontra. Não só é necessário conhecer todos os comandos e serviços para colocar o veículo em funcionamento, como também para fazer a sua supervisão. É preciso conhecer o sistema ROS para obter informação sobre o estado do robô, o que está a fazer e o que planeia executar em seguida. Para além disto, as únicas ferramentas de visualização de informação e de testes são simuladores, o Gazebo [21] é um instrumento de teste que cria simulações e mundos de demonstração para testar o *software* dos robôs. Aliado à biblioteca *rviz*, é possível ver as simulações criadas pelo Gazebo num ambiente 3D. Para ter uma ideia dos passos necessários para instalar e executar estas ferramentas, fica aqui um resumo:

```
$ sudo apt-get install python-wstool
$ source /opt/ros/hydro/setup.bash
$ mkdir ~/hratc2015_workspace
$ cd ~/hratc2015_workspace
$ wstool init src https://raw.github.com/ras-
sight/hratc2015_framework/master/hratc2015_framework.rosinstall -j8
$ rosdep install --from-paths src -i -y
$ catkin_make
$ source ~/hratc2015_workspace/devel/setup.bash
$ roslaunch gazebo_ros empty_world.launch
$ roslaunch hratc2015_framework run_simulation_world.launch
```

Apenas depois de concluir estes passos, poderemos finalmente testar e ver o comportamento do *software* do veículo num mundo simulado.

Não incluí nesta demonstração os passos necessários para arrancar todo o projeto em ROS do Husky, a sua configuração ou parametrização. O que por si só é um problema para um utilizador não versado em tecnologias da informação e desligado do desenvolvimento do robô.

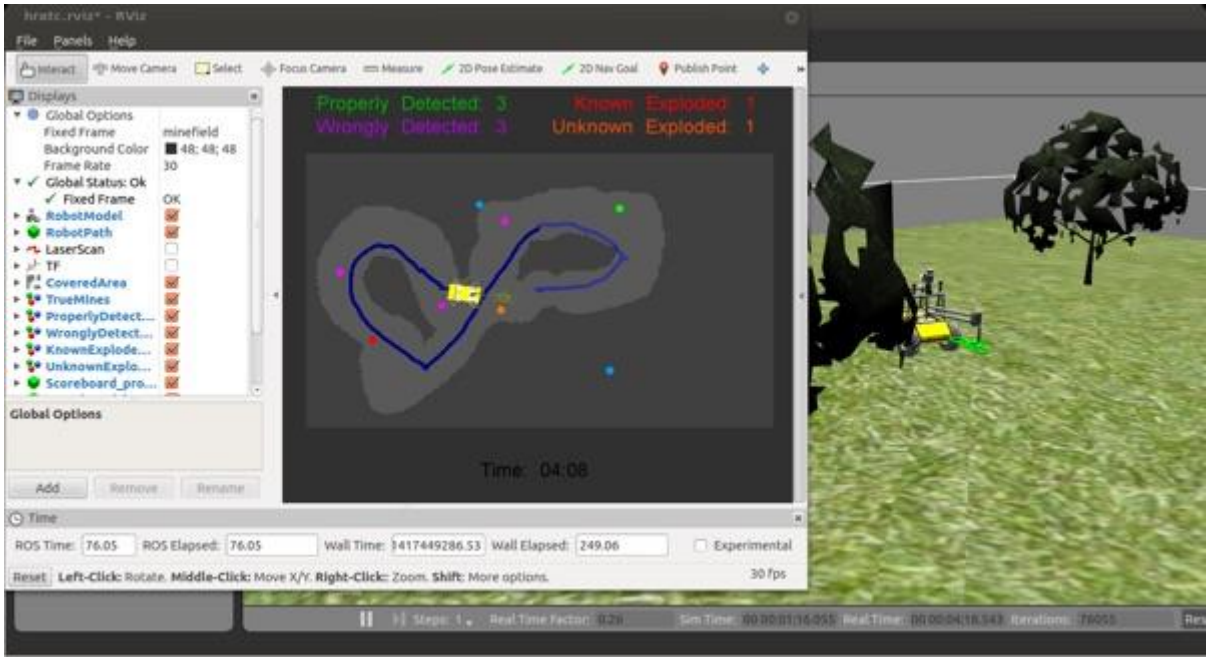


Figura 7 - Simulação através do Gazebo e rviz

Na figura anterior podemos ver esta interface de simulação, relembrando que apenas serve para esse propósito e tem pouca utilidade em testes no terreno, sendo ainda mais impraticável para o uso corrente de um utilizador final. O objetivo desta análise foi contextualizar o estado atual do Husky e da sua interação com os desenvolvedores e utilizadores, pretendendo assim introduzir o problema de interação que existe entre o Husky e o utilizador final

Capítulo 3 - Método de Abordagem

Este capítulo descreve a abordagem por mim seguida para a implementação da plataforma de interface gráfica para o robô FSR Husky de modo a poder supervisionar e controlar as suas tarefas. Esta plataforma, ou GUI, integrou-se com todo o projeto já desenvolvido no ISR, quer as tecnologias robóticas, quer todo o *software* desenvolvido em ROS.

Neste capítulo está incluída uma secção sobre este sistema, como funciona e em que medida as suas características influenciaram e ajudaram a moldar a interface criada.

De seguida um estudo sobre o contexto da aplicação e uma análise aos modelos e diagramas que auxiliaram na criação da plataforma de interface gráfica.

3.1. FSR Husky

Figura 8 - O robô FSR Husky

O FSR Husky, robô desenvolvido no Instituto de Sistemas e Robótica, é um veículo de procura autónomo, montado e caracterizado com a missão de deteção de minas e outros explosivos não detonados, usando para esse efeito uma série de sensores a atuadores controlados por um *software* também desenvolvido no ISR que acompanha o robô como uma só entidade.

Este veículo é composto por vários módulos eletrónicos, peças e circuitos assentados em cima da base da Clearpath Robotics[7] o Clearpath Husky A200. Este suporte apresenta bastante espaço interior, facilitando a montagem de *hardware* no seu interior, bem como uma capacidade máxima de carga de 75Kg, o que permite a utilização do braço mecânico e do detetor na sua extremidade sem problemas. Possui quatro rodas motrizes, ativáveis duas a duas (para permitir a condução em qualquer direção) e uma velocidade máxima de 1m/s, cerca de 3,7 quilómetros por hora[8].

É alimentado por uma bateria recarregável e substituível, o que facilita o uso do veículo no terreno. Todos os sistemas do robô são alimentados por esta bateria, apesar disso, alguns dos módulos são sustentados por diferentes voltagens, o que obriga a uma gestão da energia elétrica, tarefa assegurada pelos sistemas desenvolvidos no ISR que podem ser consultados em pormenor no manual do Husky em referência.

Em termos de *hardware* instalado no Husky pelo grupo de trabalho do Instituto de Sistemas e Robótica previamente ao início deste estágio, temos:

- Duas câmaras para a reconstrução stereo do ambiente que envolve o Husky através de algoritmos de visão por computador.
- Um laser montado numa plataforma giratória (verticalmente) que permite uma perceção 3D em volta do veículo com 270º de campo de visão.
- Um giroscópio para referências de direção no terreno.
- Uma unidade de GPS compatível com o sistema RTK.
- Uma antena e ponto de acesso *Wi-Fi* próprio para o uso ao ar livre.
- Uma unidade de computação Gigabyte BRIX[22].

No caso do *software* que controla e gere todo o sistema do FSR Husky, é usado o ROS, *Robot Operating System*[23], sendo que todo o projeto do Husky se encontra integrado neste quadro. O ROS é uma plataforma flexível e robusta para a concretização de projetos de robótica, é uma coleção (*framework)* de ferramentas, bibliotecas (subprogramas usados no desenvolvimento de *software*) e convenções que auxiliam e tornam possível a conceção de máquinas e robôs com um sistema complexo e sólido para uma grande variedade de aplicações.

O ROS pretende ter uma abordagem simples, de colaboração e partilha das suas bibliotecas, com suporte para várias linguagens de programação (Python, C++, Lisp) e a sua independência, podendo nós de diferentes linguagens comunicar entre si sem problemas através de ROS-*messages*. Isto promove a reutilização de *software*, a encapsulação das tarefas e a comunicação limpa entre nós. O ROS existe apenas para plataformas baseadas em Unix, como o Ubuntu, Mac OS X. Assim sendo, a unidade central de processamento do FSR Husky corre Ubuntu, e vem preparado com todas os pacotes e dependências necessárias.

Explicando com um pouco mais de pormenor como funciona este *framework*, vou abordar algumas das noções mais importantes, e aquelas que foram mais relevantes para compreender o funcionamento do robô, como também aquelas que foram usadas na implementação da plataforma de interface gráfica desenvolvida.

Ao nível do sistema de ficheiros, os projetos de ROS estão contidos dentro de *packages*, são pacotes que juntam todos os ficheiros necessários, bibliotecas, configurações e os nós. É nestes últimos que se encontra o código fonte, e a partir dos quais se podem executar os programas. Outro ponto importante do sistema de ficheiros é a definição e descrição dos tipos de mensagens, em aquivos do tipo *msg*. Estes definem o conteúdo de uma mensagem, a sua estrutura e que campos de informação se incluem na mesma. Existem variados tipos de mensagens predefinidos pelo ROS, e no caso da implementação do FSR Husky, a sua maioria são desse padrão, com algumas criadas de raiz, como por exemplo, a mensagem de informação da deteção de metais.

O ROS identifica a sua rede de computação, ou seja os processos que correm e executam tarefas de *software*, como um *Computation Graph*. Esta definição de grafo computacional prende-se com o facto de todos os nós, tópicos, mensagens e serviços estarem ligados entre si, como se de um grafo se tratasse.

Desenhado para ser modular, no ROS, um projeto normalmente contém vários nós, usualmente, um para cada um dos módulos de *hardware*. O código fonte de um nó é escrito com recurso a uma biblioteca base da linguagem escolhida, neste caso o C++. Esta linguagem de programação é bastante antiga, existindo desde 1979, quando Bjarne Stroustrup[24] fazia o seu doutoramento. Primeiramente chamada "C with Classes", em referência à tentativa de usar a linguagem C com o paradigma de programação orientada a objetos, tendo o seu nome mudado para C++ (*C plus plus*) em 1993, quando uma grande parte das funções distintivas da linguagem foi implementada. O nome desta biblioteca é *roscpp* e funciona como uma ponte entre o código fonte do nó e as funções do implícitas do ROS, e o que permite usar a *framwork*. Resumindo, um nó é um processo de computação que é inicializado através do comando *rosrun* no terminal do Ubuntu. Com o comando *rosnode* podemos pedir informações sobre o nó (*info*), todos os nós que correm na máquina onde estamos (*machine*), ou mesmo todos os nós que correm no sistema, independentemente da sua localização física (*list*).

Antes de passar à introdução sobre as mensagens e os tópicos, é importante referir como este sistema funciona em termos de gestão de nomes e procura dos mesmos no grafo, este suporte chama-se ROS Master, e é responsável por manter o servidor de nomes e o *lookup* (ação de procura no grafo pelo nome pedido) para encontrar as mensagens, os tópicos e outros nós. Sem ele não seria possível aos nós comunicarem entre si, enviarem mensagens, lerem mensagens ou invocarem serviços.

As mensagens, como descrito em cima, são uma estrutura de dados, compostas por campos de informação, não muito diferentes de uma *struct* em linguagem C. Suportam tipos de dados primitivos e também listas, sendo ainda importante reafirmar que estas estruturas têm de ser definidas antes da execução de qualquer programa, fazendo parte do sistema de ficheiros, só assim será possível utiliza-las para enviar e receber informação através dos tópicos. Com recurso ao terminal, podemos observar os campos de uma mensagem com o comando *rosmsg show*, e também uma lista das mensagens com *rosmsg list*. É também possível mostrar quais as mensagens que estão a ser usadas por uma *package*, ou, quais as *packages* que estão a usar a mensagem que procurámos.

Por fim, os tópicos são estruturas com um nome associado, a partir dos quais é possível escrever e ler (*publish* e *subscribe*) mensagens. O nome do tópico é muito importante, pois é com base neste que o ROS Master procura quais os tópicos que cada nó pretende encontrar. Qualquer número de nós pode publicar e subscrever um tópico, e, além disso os publicadores e os subscritores não têm conhecimento mútuo, o que desliga a produção de mensagens do seu consumo, tornando o sistema assíncrono. O comando *rostopic* no terminal dá acesso a uma multiplicidade de opções, entre as mais importantes estão o *echo*, que reproduz o conteúdo de um tópico, ou seja, mostra as mensagens recebidas em tempo real, e na prática, funciona como se o terminal fosse um subscritor do tópico, e, o *rostopic list*, que retorna uma lista de todos os tópicos presentes e os seus identificadores. Estes comandos foram extensamente usados para identificar o funcionamento do projeto

do FSR Husky e ajudar a definir objetivos e requisitos para a plataforma de interface gráfica.

Na figura seguinte podemos ver um exemplo simples de como funciona o sistema de nós (*Node*), publicadores (*publish*), tópicos (*Topic*) e nós subscritores (*subscribe*).

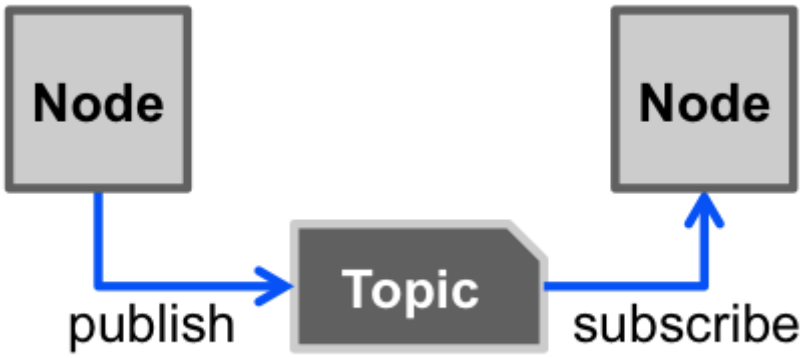


Figura 9 - Conceito básico de ROS

O projeto de ROS do FSR Husky abrange um grande número de nós, tópicos e mensagens, sendo que vou apenas descrever o funcionamento global de algum do *software* do robô para melhor relatar a realidade do veículo. Uma análise detalhada pode ser encontrada no manual de referência do FSR Husky[8].

Os oito módulos principais, e em equivalência com os módulos de *hardware*, são:

- Husky (Base): controla as rodas, a bateria e a base de todo o robô. É possível aceder aos consumos de energia, a velocidade, a temperatura dos motores e também controlar os atuadores, isto é, as rodas.
- Localização: a unidade de cálculo de inércia (IMU), que funciona como bússola e acelerómetro, e o sistema de GPS, que retorna coordenadas da antena, funcionam em conjunto para localizar o robô no espaço.
- Câmaras: duas câmaras de vídeo para analisar o ambiente do robô. São usadas em conjunto com o módulo do laser para construir imagens 3D da vizinhança do robô.
- Laser: com 270 graus de campo de visão, este módulo permite analisar o que existe à frente do robô e construir com isso uma nuvem de pontos 3D que reconstrói o terreno e os obstáculos encontrados. É essencial para a autopreservação do robô.
- Braço: o braço mecânico do aparelho consegue movimentos laterais *(Sweep)* e verticais *(Lift)*. É o módulo que faz a deteção de MT através de um detetor de metais.
- Navegação: este módulo é o que permite a autonomia do robô. É possível fazer o mapeamento de uma zona de terreno, designar uma área de procura e simplesmente supervisionar a ação do Husky. É um dos módulos que teve mais

influência na interface gráfica, visto que apenas posso atribuir missões que estejam programadas para o veículo. Neste caso, um polígono de procura.

- *Joystick*: é a interface de teleoperação do Husky, feita através de um comando XBox360.

3.2. Análise de Requisitos

Objetivo

O objetivo da análise de requisitos é apresentar uma descrição detalhada das funcionalidades da interface gráfica para o FSR Husky, o seu funcionamento esperado, interfaces e sistemas, o que cada um deles faz e o que foi necessário para implementar a plataforma. Serve também para explicar como reage, o que é expectável do sistema e os limites do mesmo.

Esta análise foi elaborada com a ajuda da equipa do ISR que desenvolve o robô desminador FSR Husky, bem como com os orientadores do projeto.

Âmbito

A plataforma desenvolvida é uma interface gráfica para supervisão e controlo de um robô de desminagem. Esta permite a um utilizador ou operador definir missões de desminagem através da marcação de uma área de procura, num mapa previamente georreferenciado. O nome do *software* é "FSR Husky", homónimo do robô. Uma segunda valência da plataforma é habilitar um operador avançado a controlar e supervisionar cada um dos módulos do FSR Husky. Esta plataforma não possibilita a teleoperação direta do mesmo, o que é feito através de um comando XBox360, desenvolvido para o efeito, no entanto deve ser possível enviar comandos simples de teste e otimização do veículo.

Esta aplicação funciona num computador portátil, que acompanha o operador enquanto o veículo autónomo conduz as suas missões. Este computador já existia no sistema do Husky, sendo denominado "*Base Station"*. Esta estação usa uma antena GPS de alto rendimento, assistindo assim o robô na definição da sua localização.

Realçando, o interface gráfico tem dois modos de funcionamento:

- Modo Normal: Georreferenciação de uma imagem, planeamento de missão e supervisão dos trabalhos.

Neste modo o objetivo é acompanhar o robô nas suas tarefas autónomas e que fazem parte do comportamento normal do veículo, de acordo com as especificações desenvolvidas no ISR pela equipa responsável pelo mesmo. Este modo é escolhido por defeito quando a plataforma inicia e deve ser usado pelos utilizadores finais.

- Modo Avançado: Supervisão detalhada de todo o funcionamento do robô e controlo individual e coletivo dos módulos.

Este modo tem de ser ativado manualmente na interface e deve ser apenas utilizado por pessoal com conhecimentos técnicos do robô, da sua programação ou por eventuais desenvolvedores futuros desta mesma plataforma. As informações necessárias ao uso operacional do robô estão todas disponíveis no Modo Normal.

Descrição Geral

Perspetiva do Produto

Existem muitos tipos de interfaces para robôs e para controlo de veículos autónomos, mas a especificidade de cada unidade requer uma solução à medida de cada um. A solução que existe neste momento é o contacto direto com o *software* através de linhas de comando que enviam ordens para o FSR Husky, e recebem o *feedback* em texto, muitas vezes indecifrável. Ora é este paradigma que se pretendia mudar, e tornar a interação com o robô fácil e intuitiva. Não só para o utilizador final, o objetivo primário, mas também para nós, os investigadores que desenvolvem este sistema de desminagem.

Sendo parte integrante de todo o projeto do FSR Husky, algumas restrições e requisitos são herdados diretamente do projeto mãe, como por exemplo os tipos de solo que o robô pode desminar e sistemas de GPS já implementados. Também ao nível de interfaces do *software*, a utilização do ROS, do seu sistema de subscritores e mensagens e, finalmente, da rede sem fios instalada no veículo, a qual vamos utilizar para recolher a informação das atividades do Husky.

Figura 10 - Interação com o FSR Husky

Interfaces do Sistema

ROS:

Nós*:* Os nós de ROS são módulos de código em C++ ou python (neste projeto, sempre o primeiro) onde é feita a parte lógica de programação do robô.

Publicadores: São nós que publicam informação nos tópicos.

Subscritores: Estes são os módulos que subscrevem os tópicos e recebem a sua informação por mensagens.

Tópicos: Onde a informação enviada pelos *publishers* fica armazenada até ser consumida pelos subscritores do tópico, existindo várias configurações ao nível de *durable subscription* e entrega a múltiplos subscritores.

Plataforma da Interface Gráfica:

Named Pipes: É uma estrutura do tipo *FIFO* (*first in first out*) da linguagem de programação C e C++, através da qual o nó subscritor que detém a informação sobre o robô comunica com a plataforma e vice-versa.

Disco Rígido: O disco do computador portátil onde a plataforma é utilizada contém não só as imagens a usar na aplicação, como também variáveis globais e de sistema associadas ao *software* FSR Husky.

Interfaces Gráficas

Janela Principal:

- Abrir imagem do disco.
- Mostrar/Esconder janela de Georreferenciar imagem.
- Mostrar/Esconder janela de procura (abrir imagem já georreferenciada).
- Parar todas as operações (botão de emergência).

Menus de Contexto:

- Acionar modo avançado.
- Configurações.
- Sobre FSR Husky.

Informação Geral:

- Dados iniciais e relevantes de feedback do robô.

Modo avançado:

- Abas para cada um dos módulos:
  - Husky: motor, baterias.
  - Localização: GPS.
  - Câmaras: Estado e *feed* das câmaras.
  - Laser: Estado e imagem da nuvem de pontos 2D.

- Navegação: Posição do alvo e velocidades lineares.

- Braço: posição, controlos simples.

- Comando: botões, *log*.

- Terminal, linha de comandos.

Seletor de Ficheiros:

- Escolher imagem a partir do sistema de ficheiros.

Janela de Georreferenciação:

- Marcar dois pontos na imagem e estabelecer a sua localização GPS.

- Confirmar dados (gravar em disco estes dados)

Janela de Procura:

- Marcar pontos na imagem, construir o polígono de procura

- Iniciar trabalhos

Maquetes para todas estas janelas e interfaces estão disponíveis em anexo a este documento.

Interfaces de Software

O sistema não comunica com *software* externo ao projeto.

Apesar disso, é necessário para que a interface funcione que existam imagens de satélite, de balão, ou outros meios não previstos nestes requisitos.

O *Google Maps* (https://www.google.com/maps), *website* que possibilita captar imagens de satélite de quase todo o mundo, é a fonte de dados recomendada para a obtenção das imagens para a plataforma.

Também é possível usar o *Open Street Maps* (https://www.openstreetmap.org), com o mesmo efeito, embora não permita imagens do terreno.

Interfaces de Comunicação

A plataforma da interface gráfica utiliza os meios já disponíveis para comunicar com o FSR Husky. Esta comunicação pode ser feita por *ssh* (*Secure Shell «Protocol»*) através de *sockets* TCP (*Transmission Control Protocol*) e por VNC (*Virtual Network*

Computing) em que qualquer cliente desta família se consegue ligar ao veículo. Informações adicionais sobre estas ligações podem ser consultadas no anexo respeitante ao manual do FSR Husky.

No caso específico de envio de ordens para o Husky, é usado o protocolo próprio do ROS, em que é enviada uma mensagem do tipo correspondente à ordem pretendida pelo nó transmissor (transparente para o utilizador). Esta mensagem é interpretada pelo ROS e colocada no tópico correspondente através de TCP (também tudo transparente para o utilizador).

Para receber a informação do veículo, no sentido inverso, o subscritor ligado à plataforma recebe os dados dos tópicos através do ROS, e consequentemente por TCP.

Limitações de Memória ou Recursos

Nenhumas limitações a apresentar.

Funções do Produto

O principal objetivo de todo o sistema é procurar e detetar minas terrestres, facilitar essa tarefa e torná-la mais eficaz e rápida. A interface gráfica ajuda a passar os dados do terreno para mapas digitais, o que permite realizar a tarefa de neutralização de minas mais facilmente e com menos risco. Pretende-se ainda com a implementação do modo avançado ajudar no desenvolvimento do sistema, dando aos investigadores ferramentas mais capazes, imediatas e inteligíveis de modo a acelerar a investigação e alcançar metas e objetivos mais concretos com esta tecnologia.

O Modo Normal é o mais importante para o projeto, sendo este que foi desenvolvido primeiro e com maior pormenor.

Modo Normal:

- Georreferenciar imagem. O utilizador marca dois pontos num mapa ou imagem, digita (ou usa o GPS do Husky) as coordenadas GPS das duas marcas, verifica os dados e aprova a georreferenciação.

- Desenhar área de procura. Colocando os vértices do polígono que forma a zona a desminar, o utilizador representa no mapa a área de procura para o Husky. Pode também modificar a localização de qualquer um dos vértices arrastando o mesmo para outro local.

- Iniciar tarefa de desminagem. Um botão de iniciar dá ordem para o começo das operações de desminagem.

- Supervisionar a tarefa através do mapa e dados recebidos do Husky. Através do ecrã principal e do ecrã do mapa, podemos ver a área a explorar, as minas encontradas, e a localização corrente do Husky.

- Paragem de emergência. Um botão de paragem situa-se na janela principal de modo a que seja possível, em caso de emergência, parar o robô. Isto também é possível recorrendo a uma combinação de teclas no comando XBox360.

- Ver e exportar os mapas para utilização futura.

Modo Avançado:

- Selecionar qualquer um dos módulos e ver a informação detalhada sobre eles. Selecionando uma das abas do modo avançado, podemos inferir sobre a informação e dados recebidos do veículo.

- Ligar, desligar, testar e observar módulos. Dependendo do módulo selecionado, é possível interagir com o robô e alguns dos seus módulos de forma a poder obter informação sobre o estado do robô.

Características do Utilizador

Os utilizadores alvo do sistema diferem consoante o modo de utilização.

Para o modo normal, o sistema é desenhado para que uma pessoa com formação básica em informática possa exercer funções de supervisão, necessitando ter também algum conhecimento de técnicas de desminagem e conhecimento sobre o terreno a desminar, tal como normas de segurança para o operador e outros intervenientes na ação de desminagem.

O modo avançado é projetado para os investigadores do ISR que trabalham e desenvolvem o FSR Husky, sendo mais uma ferramenta de ajuda ao seu desenvolvimento futuro. É também desenhado para que nos trabalhos no terreno existam ferramentas disponíveis para despistar problemas e fazer *troubleshooting*, isto é, clarificar questões ou dificuldades que surjam no veículo durante uma missão no terreno.

Limitações

Algumas limitações do projeto prendem-se com problemas normais neste tipo de implementação, os quais já existiam no uso do FSR Husky à data do início deste estágio.

A comunicação feita entre o computador portátil e o robô é realizada por rede sem fios, que tem distâncias máximas e é suscetível a ruídos e interferências. Deparamo-nos também com a questão do tempo de vida da bateria, quer do computador portátil, como do próprio robô. É igualmente necessária uma ligação GPS estável no robô e nas antenas que compõem a *Base Station*, para que todos os sistemas baseados na localização funcionem com a devida eficácia.

É imperativo que se tracem planos de segurança para possíveis falhas. Perder o controlo sobre um veículo num campo minado pode ser extremamente perigoso para o sistema e claro, também para o operador ou outras pessoas presentes. É assim essencial, a existência de um sistema de segurança de avaria ou outros problemas, para terminar as operações e tarefas imediatamente. O botão de paragem de emergência está disponível na interface gráfica e o operador deve estar preparado para o utilizar, caso seja necessário.

Pressupostos e Dependências

É assumido neste projeto que o sistema pode mudar a qualquer momento, sendo um projeto de investigação, a equipa do ISR continua a conceber melhoramentos e mudanças no robô e nos seus sistemas. É portanto de salientar que alguns destes requisitos possam mudar, e que outras partes da plataforma possam sofrer alterações, nomeadamente ao nível da subscrição de tópicos e envio de mensagens. No entanto, o sistema ROS e a base da interface devem manter-se iguais.

A plataforma da interface gráfica é dependente do funcionamento correto do Huksy, quer ao nível da rede sem fios que dele origina, dos seus atuadores (rodas e braço mecânico) ou dos seus sensores (detetor de metais, laser, câmaras e GPS). No caso do sistema GPS, que localiza o robô no seu ambiente, a dependência recai sobre alguns fatores, como a condição atmosférica, a quantidade e qualidade dos satélites disponíveis naquela zona, e também a hora do dia. Simultaneamente, estas dependências também se aplicam à *Base Station* e à antena GPS que a complementa.

Requisitos de Implementação Futura

Os requisitos aqui explanados que se referem à utilização dos dados de pesquisa do robô para carregamento para uma base de dados GIS (*Geographic Information System*) desenvolvida por um dos parceiros do projeto TIRAMISU ficam para uma futura versão da plataforma da interface gráfica.

De igual modo, a integração com um sistema mobile, desenvolvido no ISR, no qual se pode fazer a supervisão de uma missão num *smartphone* ou *tablet*, e a sua descrição ficam programadas para próxima versão da plataforma.

Requisitos Específicos

Requisitos da Interface Gráfica

ID: RIG1
Nome: Janela de Missão
Descrição: Deverá existir uma janela onde o utilizador vê a imagem georreferenciada e pode marcar a missão.
Razão: Supervisão do Husky.

ID: RIG2
Nome: Marcar Terreno a procurar.
Descrição: Na Janela de Missão exposta no requisito anterior, o utilizador pode marcar pontos no terreno que correspondem a vértices de um polígono de procura.
Razão: Marcação do terreno a desminar pelo Husky.

ID: RIG3
Nome: Modificar Polígono de procura.
Descrição: Depois de criado o polígono, deve ser possível manipular a posição de cada um dos vértices, atualizando o polígono de procura.
Razão: Aumentar a sensibilidade do sistema e permitir melhor parametrização do polígono de procura.

ID: RIG4
Nome: Supervisão da Missão - Husky.
Descrição: A plataforma deve permitir ao utilizador supervisionar a missão através da janela de missão mostrando a posição do Husky.
Razão: Representar o Husky no mapa de modo a facilitar a supervisão.

ID: RIG5
Nome: Supervisão da Missão - Minas.
Descrição: A plataforma deve permitir ao utilizador supervisionar a missão através da janela de missão mostrando a posição das minas ou engenhos explosivos não detonados encontradas no terreno.
Razão: Representar no mapa os sinais positivos de deteção do Husky dando *feedback* rápido ao supervisor.

ID: RIG6
Nome: Botão de Emergência - STOP.
Descrição: Deve ser possível ao utilizador parar o robô em situação de emergência.
Razão: Paragem de emergência é um requisito em todas as máquinas deste tipo pelo perigo que pode surgir num terreno minado.

ID: RIG7
Nome: Botão do Modo Avançado.
Descrição: A plataforma terá um modo avançado de supervisão onde informação mais detalhada é reproduzida. Este modo é acionado através do botão existente para o efeito.
Razão: Permitir testes e supervisão detalhada aos desenvolvedores do Husky.

ID: RIG8
Nome: Aba do Modo Avançado - Husky Base.
Descrição: Informação sobre:
- Estado da bateria.
- Consumos Energéticos.
- Temperatura dos motores.
- Velocidade dos motores.

Razão: Permitir testes e supervisão detalhada acerca da base do Husky.

ID: RIG9
Nome: Aba do Modo Avançado - Localização.
Descrição: Informação sobre:
- Coordenadas GPS.
- Acelerómetro x, y, z.
- Orientação 3D.

Razão: Permitir testes e supervisão detalhada acerca da localização do Husky.

ID: RIG10
Nome: Aba do Modo Avançado - Câmaras.
Descrição: Imagens em tempo real:
- Câmara esquerda.
- Câmara direita.

Razão: Dar possibilidade de uma telepresença mais forte na supervisão.

ID: RIG11
Nome: Aba do Modo Avançado - Laser.
Descrição: Imagens em tempo real:
- Plano de pontos 2D.
- Construção de nuvem de pontos 3D.
- Construção de nuvem de pontos projetada em plano 2D.

Razão: Dar possibilidade de uma telepresença mais forte na supervisão e uma perceção mais próxima da visão eletrónica e tomada de decisão do Husky.

ID: RIG12
Nome: Aba do Modo Avançado - Navegação.
Descrição: Informação sobre:
- Ponto objetivo.

Razão: Ver para onde o robô se está a dirigir neste momento.

ID: RIG13
Nome: Aba do Modo Avançado - Braço.
Descrição: Informação sobre:
- Posição em Varrimento (*sweep*).
- Posição em Altura (*lift*).

Razão: É importante saber a posição do braço de modo a saber que zona está a ser alvo de procura.

ID: RIG14
Nome: Aba do Modo Avançado - Comando.
Descrição: Informação sobre:
- Teclas do comando premidas.
- Usar comando virtual.

Razão: Possibilidade de *debugging* do comando e operar o robô mesmo quando este fica sem bateria ou sem alcance para o Husky.

ID: RIG15
Nome: Aba do Modo Avançado - Terminal.
Descrição: Um terminal *sh* (*Shell*) na aba correspondente.
Razão: A linha de comandos do LINUX continua a ser útil para algumas tarefas de programação.

ID: RIG16
Nome: *Zoom* nas imagens.
Descrição: Ambas as janelas de georreferenciação e a de missão deverão permitir *zoom*, aproximação de detalhes.
Razão: Aumentar a precisão da georreferenciação e permitir ver detalhes na imagem.

ID: RIG17
Nome: Arrastar as imagens.
Descrição: Ambas as janelas de georreferenciação e a de missão deverão permitir arrastar a imagem de modo a não perder informação quando é feito *zoom*.
Razão: Permitir ver detalhes na imagem.

ID: RIG18
Nome: Movimentação dinâmica.
Descrição: Depois de iniciada a missão, a janela de missão deve acompanhar os movimentos do robô dinamicamente, mantendo sempre o foco neste.
Razão: Manter o foco no Husky.

Requisitos das Interfaces de Hardware

ID: RIH1
Nome: Rede sem fios.
Descrição: O computador onde a plataforma se encontra em funcionamento deve ser capaz de se ligar à rede sem fios que origina do Husky. É usado o padrão 802.11g.
Razão: Sem a ligação, o sistema não funciona.

Requisitos das Interfaces de Comunicação

ID: RIC1
Nome: Subscrição das mensagens dos tópicos do ROS.
Descrição: É necessário que a plataforma seja capaz de subscrever aos tópicos do Husky, ler, tratar e guardar a informação sobre o robô. Nesse sentido, é usado um nó de ROS que subscreve todos os tópicos necessários, e obtém todos os dados.
Razão: A interface de *software* com o ROS tem de ser desenvolvida para a plataforma.

ID: RIC2
Nome: Publicar mensagens nos tópicos do ROS.
Descrição: Para enviar qualquer tipo de ordem para o robô, é necessário enviar mensagens para os tópicos respetivos da ordem a enviar. Para esse efeito é usado, de forma transparente, um nó de ROS criado para o efeito, enviando as mensagens diretamente para os tópicos. Para isso o computador onde está a correr a plataforma, precisa de estar na rede sem fios do Husky, e ter o ROS Master configurado com a variável de sistema $URI_MASTER apontando para o robô.

Requisitos Funcionais

ID: RF1
Nome: Carregar Imagem.
Descrição: Depois de inicializada a aplicação, o utilizador pode carregar uma imagem de satélite correspondente ao terreno a procurar.
Se a imagem já tiver sido georreferenciada numa sessão anterior, deve carregar esses dados.
Razão: Possibilita a supervisão das operações do robô.

ID: RF2
Nome: Georreferenciar a Imagem
Descrição: A plataforma deve permitir a georreferenciação de uma imagem através da escolha de dois pontos na mesma, de forma a marcar as coordenadas GPS desses mesmos pontos.
Razão: É necessário georreferenciar a imagem de modo a relacionar as coordenadas GPS com as coordenadas em pixéis da imagem.

ID: RF3
Nome: Corrigir ou anular uma Georreferenciação.
Descrição: Deverá ser possível corrigir ou cancelar uma georreferenciação previamente feita, tanto através da manipulação dos pontos escolhidos, como também da sua remoção.
Razão: Se uma georreferenciação não tiver sido feita corretamente, podemos corrigir o erro.

ID: RF4
Nome: Guardar a Georreferenciação para uso futuro.
Descrição: A plataforma deve guardar os dados de uma Georreferenciação em disco. Em consonância com os *standards* do ROS, deve ser usado o tipo de ficheiro "YAML".
Razão: Evitar repetição de tarefas e facilitar integração com outras soluções informáticas no ISR.

ID: RF5
Nome: Guardar Mapa de Final de Missão.
Descrição: Depois da missão cumprida, a plataforma deverá possibilitar a opção de exportar o mapa de procura para um ficheiro de imagem.
Razão: Mostrar imagem final do resultado da missão.

ID: RF6
Nome: Guardar Georreferenciação num ficheiro YAML.
Descrição: Um ficheiro YAML é um ficheiro de texto normalmente usado como suporte à computação nos sistemas ROS. Devido à familiaridade da equipa do ISR com estes ficheiros e a facilidade de integração destes dados noutros projetos, o ficheiro YAML terá o mesmo nome das imagens em execução na plataforma.
Razão: Evitar repetição de tarefas e facilitar integração com outras soluções informáticas no ISR.

ID: RF7
Nome: Conversão de Sistemas Espaciais.
Descrição: De modo a conseguir converter coordenadas GPS em coordenadas locais e vice-versa, o sistema tem de implementar as fórmulas matemáticas para o efeito. De salientar que as coordenadas GPS estão em consonância com a Projeção de Mercator, sendo por isso necessário acautelar a direção norte das imagens e forma especial de converter uma Projeção de Mercator numa projeção linear.

Requisitos de Performance

ID: RP1
Nome: Taxa de atualização mínima.
Descrição: A taxa de atualização dos dados recebidos do Husky não deve ser inferior a 1Hz, ou seja, uma atualização por segundo. É recomendada uma taxa entre os 5 e os 10 Hertz.
Razão: Os dados devem ter uma taxa de atualização que permita acompanhar todos os momentos do veículo.

ID: RP2
Nome: Latência máxima.
Descrição: O tempo que o robô leva a receber uma ordem do operador não pode superar os 500 milissegundos. A latência de uma ligação sem fios do padrão 802.11g é por norma inferior a 20 milissegundos.
Razão: É importante que as ordens dadas a partir da plataforma de interface gráfica cheguem ao robô o mais rápido possível.

ID: RP3
Nome: Fidelidade da apresentação dos dados.
Descrição: Os dados recebidos do Husky devem ser representados exatamente como lidos.
Razão: Os valores apresentados na interface têm que ser iguais aos presentes nos tópicos do ROS do robô, de modo a não haver discrepâncias.

3.3. Modelo de Utilizadores

O perfil dos utilizadores de um *software* influencia a maneira como este deve ser construído, a sua experiência e atitude têm um peso na interação com o sistema. A própria

atividade que o utilizador vai realizar tem também importância para a conceção de uma interface gráfica, neste caso, para o FSR Husky e as suas tarefas de desminagem.

As necessidades do utilizador final devem ser parte integrante da interface, o que o operador quer ver, como quer ver e a importância que cada um dos componentes da interface deve ter.

Fatores de Experiência

Os militares e técnicos ligados ao TIRAMISU que irão utilizar este sistema estão preparados e treinados para enfrentar campos minados, marcação do terreno, coordenação com as autoridades locais, entre muitas outras competências técnicas de desminagem. Assume-se que todas as preocupações de segurança e de uso do robô no terreno estão asseguradas, focando-se este estudo dos utilizadores no *software* propriamente dito.

O nível de experiência necessário para usar a plataforma é baixo. Conhecimentos básicos de computadores, interfaces digitais e uso do *hardware* do computador (rato, teclado). Considero que estas competências estão adquiridas pelos utilizadores finais da plataforma. Estes utilizadores têm também experiência com outros tipos de desminagem, como por exemplo: detetores manuais. Isso confere-lhes outra vantagem de conhecimento em operações de desminagem.

A aprendizagem do sistema será mista. Através do modelo de referência, do manual e pela operação do robô no terreno. A plataforma é de simples aprendizagem, os módulos e as valências da aplicação estão bem assinaladas, e em pouco tempo é possível utilizar todos os processos.

Matriz Perfil

Características	Perfil A	Perfil B	Perfil C	Perfil D
Estilo de aprendizagem	Heurístico[1]	Epistémico[2]	Epistémico	Heurístico
Domínio sobre desminagem	4/4	3/4	4/4	2/4
Domínio sobre informática	4/4	1/4	3/4	4/4
Profissional	Sim	Não	Sim	Sim
Tarefas				
Georreferenciar imagem	3/4 4/4	2/4 4/4	3/4 3/4	4/4 4/4
Marcar área de procura	4/4 4/4	4/4 4/4	4/4 4/4	4/4 4/4
Testar módulos	2/4 1/4	1/4 1/4	1/4 1/4	3/4 4/4
Usar terminal	1/4 1/4	1/4 1/4	1/4 1/4	2/4 3/4

[1] Descoberta e aprendizagem por exploração e experimentação
[2] Conhecimento adquirido por estudo prévio

Perfil A - Um profissional da desminagem com experiência em robôs e interfaces para sua supervisão. Exemplo: Militar profissional ligado à desminagem há vários anos.

Perfil B - Um membro da administração local que pretende por em funcionamento o Husky na sua zona. Exemplo: Funcionário municipal de uma zona afetada em Angola.

Perfil C - Profissional que domina todos os aspetos da desminagem mas que necessita do estudo do manual para compreender a plataforma.

Perfil D - Um investigador informático que tem perfeita noção do uso da plataforma, mas apenas tem experiência de desminagem em testes e simulações.

As tarefas descritas na tabela referem-se à frequência e importância (respetivamente) das mesmas para esse perfil de utilizador.

Observando a tabela, podemos concluir que as tarefas mais específicas, como usar o modo avançado, testar os módulos ou usar o terminal são pouco importantes para a maioria dos potenciais utilizadores, tendo apenas expressão no perfil D, o perfil onde se incluem os investigadores do ISR e eu próprio.

Por outro lado, as tarefas base do sistema são transversalmente importantes, sendo essas que têm mais valor para todos os perfis.

3.4. *Design* de Contexto

De modo a expandir os requisitos e abranger também outras partes do projeto onde a interface desenvolvida se insere, e também, tendo em conta que se trata de um *software* que lida diretamente com o utilizador, decidi, tendo em base as competências adquiridas na licenciatura e mestrado que agora concluo, usar uma abordagem de *design* de contexto para descrever todo o projeto.

Esta fase de pesquisa foi também, tal como a anterior, feita em colaboração com os investigadores do ISR e os orientadores do DEEC. Com a sua ajuda foi possível esmiuçar alguns componentes da organização do sistema que não seria possível de outra forma.

Modelo de *Workflow*

Este modelo, descrito na imagem da próxima página, permite verificar como o sistema flui, de onde vêm e vão os fluxos de informação ou interação. Também descortina onde existem problemas de fluxo ou problemas de integração.

Ao ler o modelo podemos verificar, como seria de esperar, que o robô é a peça central de todo o sistema, que se divide em três partes principais. Os sensores, que recolhem a informação do ambiente do veículo, os tópicos de ROS que guardam essa informação, e

o módulo de navegação que é responsável por dirigir o Husky na sua busca autónoma por minas e outros explosivos não detonados.

Os dois maiores problemas surgem precisamente na interação do robô com o supervisor, o que corrobora toda a importância deste estágio. O operador tem muitas dificuldades, ou mesmo impossibilidade, de cumprir a sua tarefa de supervisão com sucesso. É este paradigma que a plataforma desenvolvida pretendeu mudar.

Nos restantes fluxos do modelo, apesar de um ou outro problema de comunicação entre os intervenientes, o sistema funciona e é eficiente. O operador tem competências claras, tal como a equipa do ISR, que desenvolve o robô e se integra do projeto TIRAMISU através do coordenador e líder do projeto. Eu integrei também esta equipa de desenvolvimento durante este estágio.

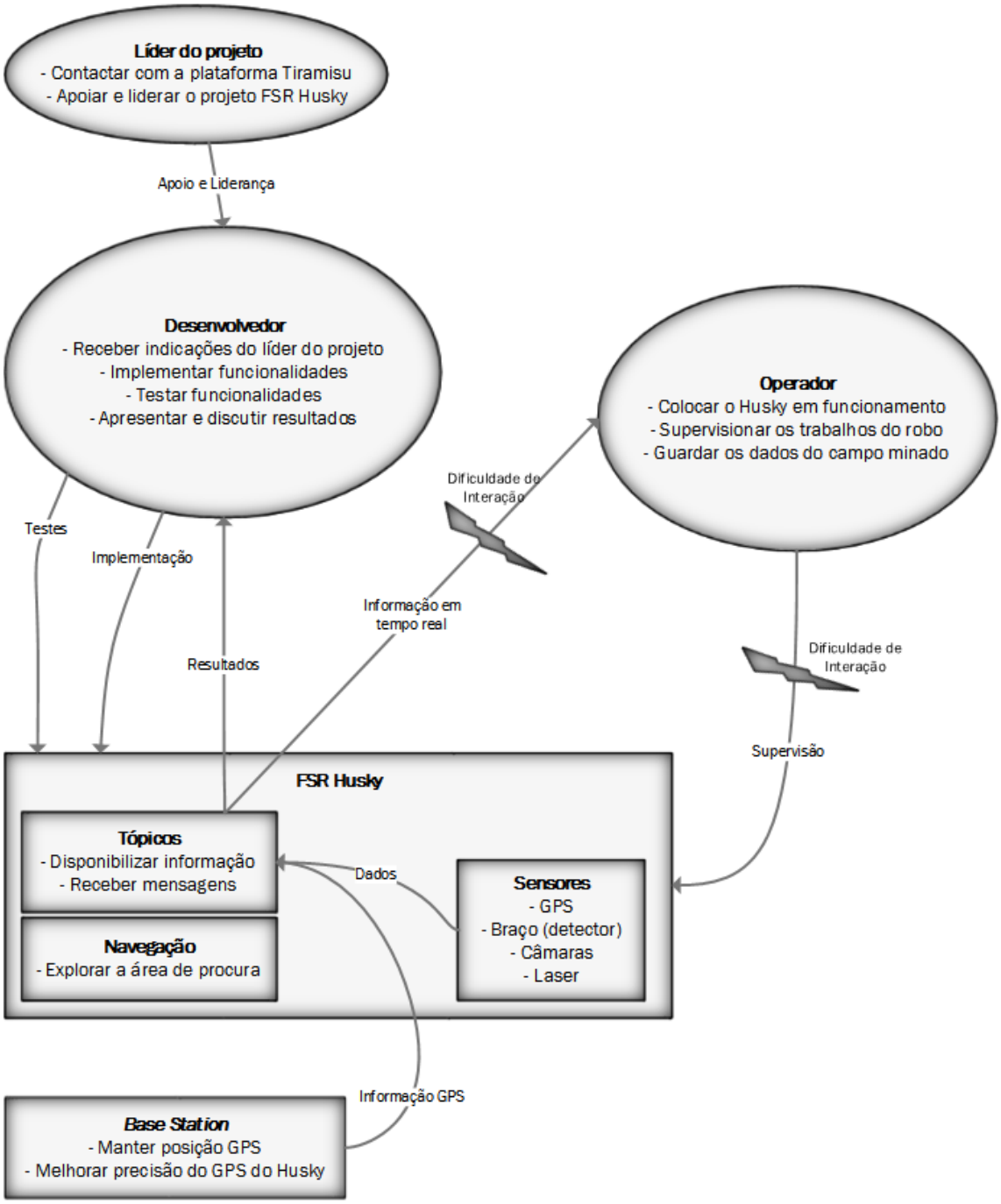


Figura 11 - Modelo de Workflow

Modelos de Sequência

Nestes modelos aqui descritos, não tentei simular ou desenhar atividades que exponham todas as sequências possíveis, ou todas as sequências de um determinado módulo. Foi feito sim, um levantamento de sequências de eventos reais, observados, que levantaram problemas de fluidez, ou problemas no próprio processo.

Intento: Iniciar o robô

1. Ligar energia do robô
2. Ligar PC
3. Ligar à rede "Husky"
4. Iniciar terminal no PC
5. Escrever todos os comandos necessários
6. *Start up* completo

Intento: Testar a resposta do braço

1. Iniciar o robô
2. Iniciar terminal no PC
3. Escrever a mensagem ROS no terminal
4. Observar resposta do braço
5. Ler os tópicos no terminal
6. Tirar conclusões do teste

Intento: Iniciar trabalhos de desminagem

1. Iniciar o robô
2. Iniciar terminal no PC
3. Escrever a mensagem ROS para o primeiro vértice do polígono de procura
4. Escrever a mensagem ROS para o segundo vértice do polígono de procura
5. (…)
6. Enviar mensagem ROS de início dos trabalhos

Com estas três sequências, em que a segunda pode ser facilmente extrapolada para os vários outros módulos do Husky, podemos perceber que nas tarefas descritas, as dificuldades (assinaladas a sombreado) centram-se sempre na interação com o terminal e na forma como se recebe e envia informação de e para o robô. Ora desse modo, podemos também concluir que estes problemas podem ser resolvidos ou minorados pela plataforma de interface gráfica desenvolvida, permitindo fazer estes passos de uma forma transparente e muito mais simples para o utilizador ou supervisor.

User Environment Design

Este passo é dos mais importantes na conceção de uma interface pois descreve todos os seus componentes, o objetivo de cada um e como se relacionam em termos de fluxo da interface. Isto permite que se observe como chegar a cada uma das opções, a cada um dos propósitos que a plataforma deve oferecer. É um esquema bastante completo que foi sendo desenvolvido ao longo do tempo e que aqui apresento a versão final.

Na figura que se apresenta em seguida, estão representadas cada uma das áreas de foco que o operador tem acesso ao usar a interface gráfica. Mais uma vez, o papel central da janela principal da plataforma, a partir da qual podemos navegar para cada uma das funcionalidades. Existem seis áreas de foco, cada uma com as suas características e funções, as quais são enunciadas no bloco correspondente a cada uma delas.

Este diagrama é também vantajoso para descrever como é que o sistema, e a interface gráfica em particular, vão ajudar o operador nas suas tarefas. Se as funcionalidades mais importantes estão facilmente identificáveis, e se a navegação entre áreas de foco está estabelecida de forma a fluir naturalmente e intuitivamente para o utilizador.

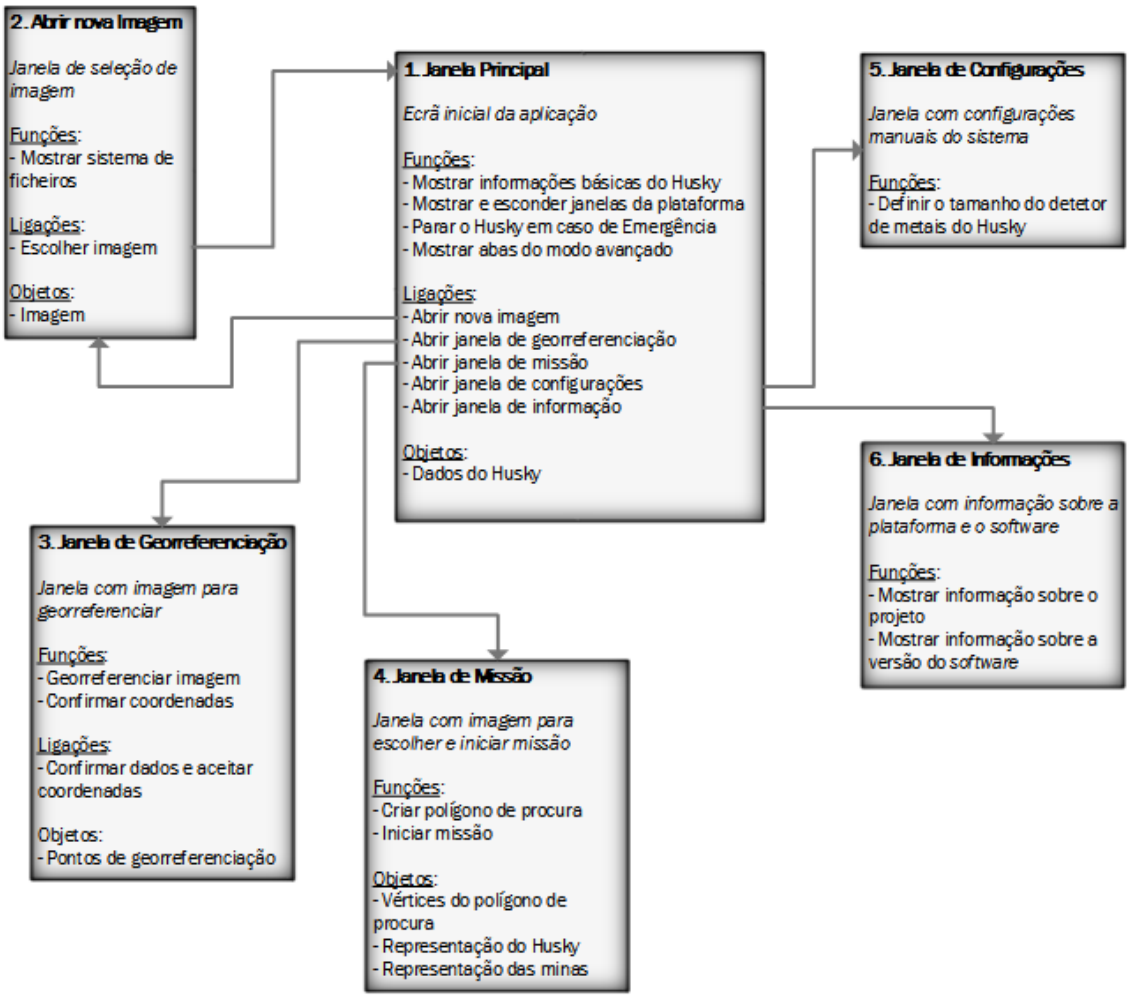


Figura 12 - Diagrama UED

Modelo Físico

Com este modelo pretendemos explicar como o sistema se comporta no terreno, quais os seus componentes gerais e como se relacionam fisicamente no mundo real. É especialmente útil para reiterar as limitações ao nível de distâncias, segurança e posições de GPS.

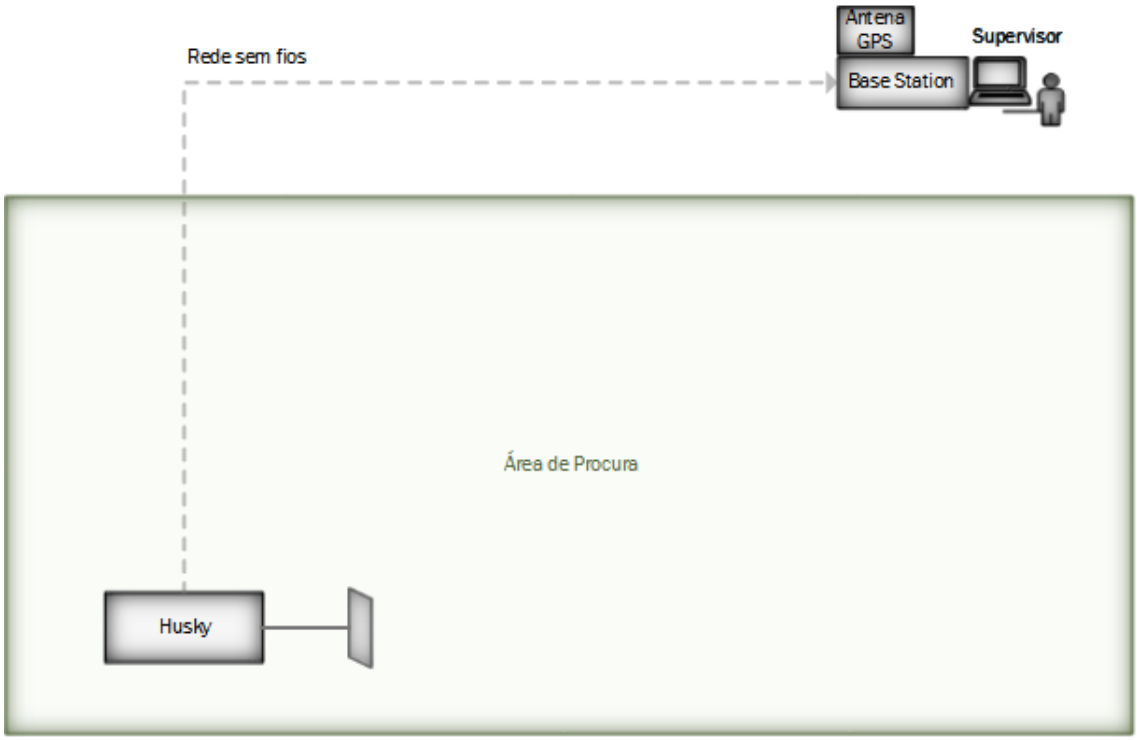


Figura 13 - Modelo Físico

Podemos observar no modelo representado na figura anterior que a disposição física do sistema é bastante simples, fugindo ao âmbito do estágio tentar resolver problemas a este nível. No entanto, tal como escrito em cima, torna-se importante para perceber que a *Base Station* se encontra junto à antena GPS, a qual melhora a precisão do sinal e da posição do robô. É através deste sistema duplo que a posição GPS do Husky se torna usável para um sistema de desminagem, com um erro de apenas poucos centímetros. Isto possibilita a marcação numa imagem georreferenciada e manter o robô e a as potencias deteções referenciadas nesse sistema.

Prototipagem e *Storyboarding*

Esta é a primeira fase de execução do *design* e a passagem da análise e estudo para a criação de uma solução para a interface gráfica. Este passo é muito importante, e apesar da natureza crua dos desenhos, a primeira reação dos utilizadores a um protótipo em papel

é extremamente útil para aferir a colocação de botões e funções, verificar se o operador consegue identificar as opções mais importantes e se estas têm a visibilidade necessária para completar as suas tarefas.

É um passo que envolve alguma criatividade, experiência e julgamento acerca do objetivo da interface, mas através de todos os estudos e análises concretizadas previamente, a intenção é fazer uma decisão informada, baseada nos factos e alicerçada em conhecimento.

Na figura seguinte é apresentado este primeiro protótipo em papel, que representa o primeiro contacto com o *design* da interface.

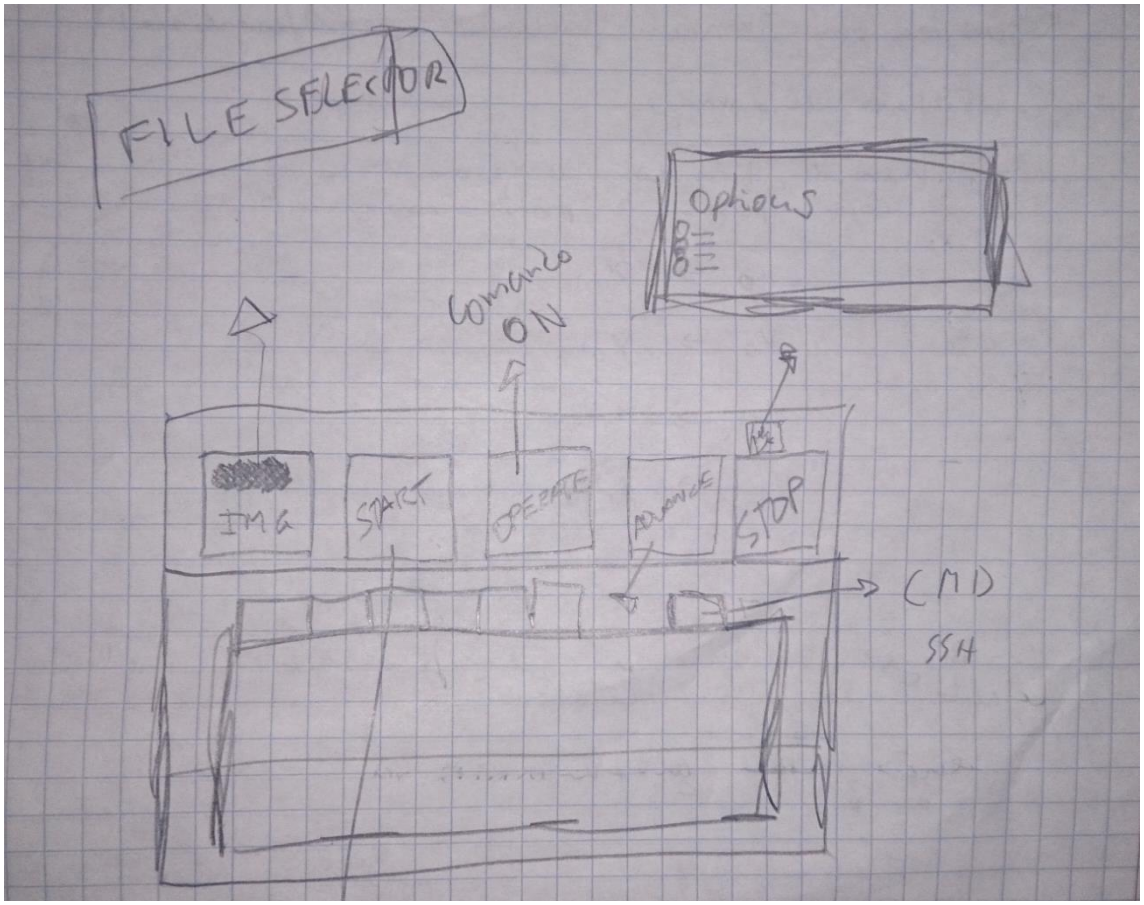


Figura 14 - Primeiro protótipo em papel

Nesta primeira e essencial abordagem ao *design*, decidi colocar as cinco ações mais importantes da plataforma como botões grandes e intuitivos. Abrir imagem, iniciar trabalhos, ligar o comando, acionar o modo avançado e o botão de paragem de emergência (STOP). Um pequeno botão de contexto, para abrir a janela de configurações, e um sistema de abas para o modo avançado. Esta decisão baseia-se no agrupamento de conjuntos semelhantes de informação, dividindo em várias partes organizadas de acordo com os módulos correspondentes ao robô. Assim, a informação encontra-se agregada e

organizada, sem ser necessário discernir de um grande número de dados a informação que pretendemos obter.

Também de realçar que nesta primeira fase ficou assente que o ecrã de procura, ou janela de procura, não iria fazer parte da janela principal, de modo a ser possível acompanhar e supervisionar a missão no mapa, e, também dos dados do veículo, em simultâneo.

Em versões subsequentes, foi adicionado um botão de contexto, que se encontra no menu, para a georreferenciação de uma imagem, visto que esta funcionalidade deve ser usada apenas uma vez por imagem, não faz parte das ações mais importantes.

Alguns *storyboards* foram também desenvolvidos nesta fase do projeto, estes representam casos de uso do interface gráfico, do ponto de vista do programador, ou seja, ajudaram a complementar os estudos prévios e a criar a arquitetura do *software*.

3.5. Arquitetura do *Software*

Neste capítulo pretendo descrever a metodologia usada para a construção do *software*, os passos que foram dados para suportar a estrutura e o código da plataforma de interface gráfica. Vamos começar pelo diagrama de camadas, passando pelo *User Environment Design*, um modelo de sequência e finalmente uma descrição do modelo da rede e de mensagens.

Este passo é extremamente importante para descrever como foi construído, em que bases assentou a estrutura do projeto de *software* e como se relacionaram as diferentes partes da plataforma.

Diagrama de Camadas

Um diagrama de camadas permite descrever a arquitetura de alto nível da plataforma desenvolvida. Define grupos lógicos abstratos chamados camadas, onde podemos observar as tarefas de cada um dos grupos de forma fácil, identificando possíveis conflitos entre a arquitetura e os objetivos do projeto. É também útil para a eventual mudança de algum artefacto, ou até, a adição de novas camadas, influir sobre o seu impacto geral no *software* já desenvolvido, como por exemplo no caso deste estágio, a possível inclusão futura de um *tablet* no sistema.

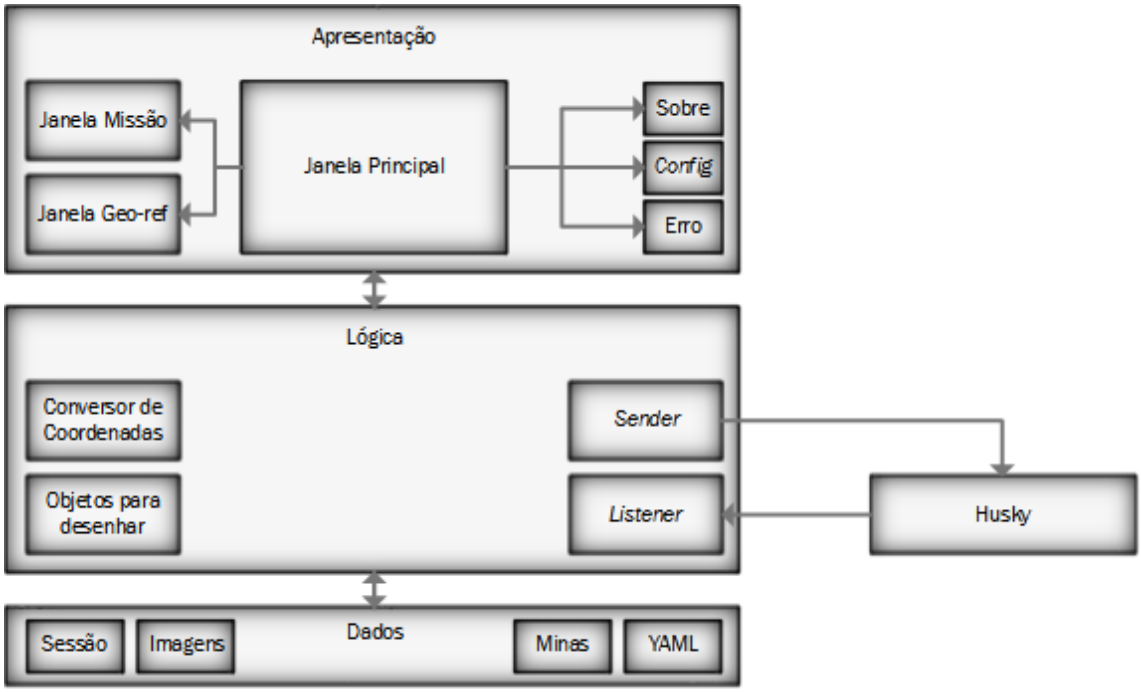


Figura 15 - Diagrama de Camadas

Analisando a figura anterior, podemos observar que existem duas camadas principais na plataforma de interface gráfica, a camada de apresentação e a camada de lógica. A primeira contém toda a parte de *design* da interface gráfica, o que o utilizador final vê, e com a qual interage no exercício das suas funções. A Janela principal é o ponto central desta camada, e a partir da qual os outros componentes da interface são inicializados e controlados.

Na camada de lógica temos as funções principais do programa, como a conversão de coordenadas, a gestão dos objetos desenhados, dos ficheiros e imagens, a ligação com a camada de dados e as *Threads* (excertos de código que correm em paralelo, coincidentemente com todas as *threads* do processo principal) que fazem a ligação entre a plataforma da interface gráfica com os tópicos do ROS que correm no Husky.

Diagrama de Sequência

Este diagrama pretende modelar a interação entre os objetos e também o ator numa sequência temporal para um cenário específico. Neste caso, o uso completo de todo o modo normal da aplicação, desde o arranque da plataforma de interface gráfica até ao final de uma missão de desminagem, passando pela seleção e georreferenciação de uma imagem, a marcação da área de procura e a ordem de início de missão.

Com base nesta representação podemos ver os estímulos que o utilizador envia para o *software* e as respostas que este devolve ao operador. Quase que como uma conversa entre os dois, o que facilita a perceção do que é necessário para que ambos entendam as intenções e haja uma resposta condizente do sistema.

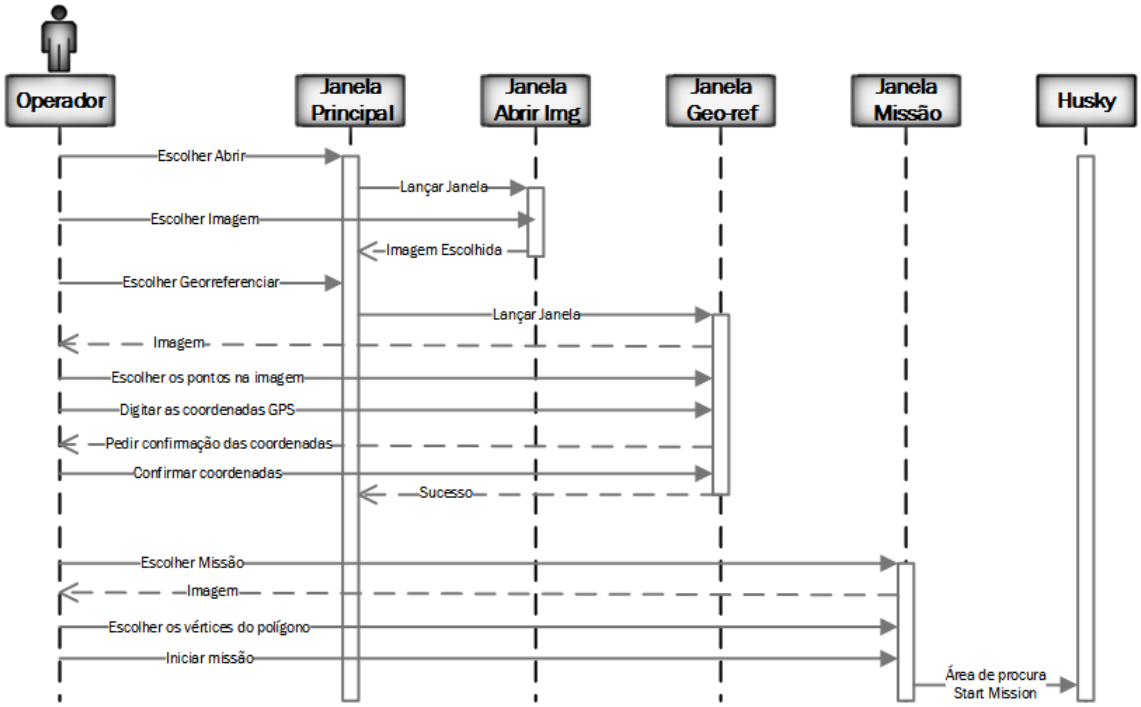


Figura 16 - Diagrama de Sequência

Descrição da Rede

A ligação que é feita entre o Husky e o computador que corre a plataforma de interface gráfica é sustentada pelo ROS, uma implementação de alto nível que usa *sockets* de TCP para comunicar entre os tópicos, os subscritores e os publicadores. Sendo que essa camada de dados de transmissão não é aberta e sim transparente para os desenvolvedores, a sua especificação e manipulação é redundante, optando assim por utilizar todos mecanismos oferecidos pelo *rostopic*.

As mensagens enviadas de e para o Husky através de ROS-*messages* usam a rede sem fios que é mantida pelo robô, sendo também transparente quer para o utilizador, quer para os investigadores que desenvolvem o Husky. Este sistema está descrito com mais pormenor no capítulo relativo à análise de requisitos.

Casos de Uso

Um diagrama de casos de uso descreve uma sequência de ações que mostram algum tipo de valor mensurável para o ator que as desencadeiam. Estas sequências são representadas por elipses. O ator, neste caso, representa o operador do Husky nas suas tarefas de desminagem e é representado por uma figura de forma humana.

No caso do diagrama apresentado na imagem seguinte, o operador consegue medir a sua ação de georreferenciação como um sucesso ou insucesso, dependendo dos dados digitados na interface gráfica, sendo que não é possível completar com sucesso essa sequencia se não executar todos os passos incluídos nessa mesma sequência, assinalados na imagem como *includes* a ela ligados.

Da mesma forma, não é possível completar a sequência de início de missão se não for completada com sucesso a criação do polígono de procura.

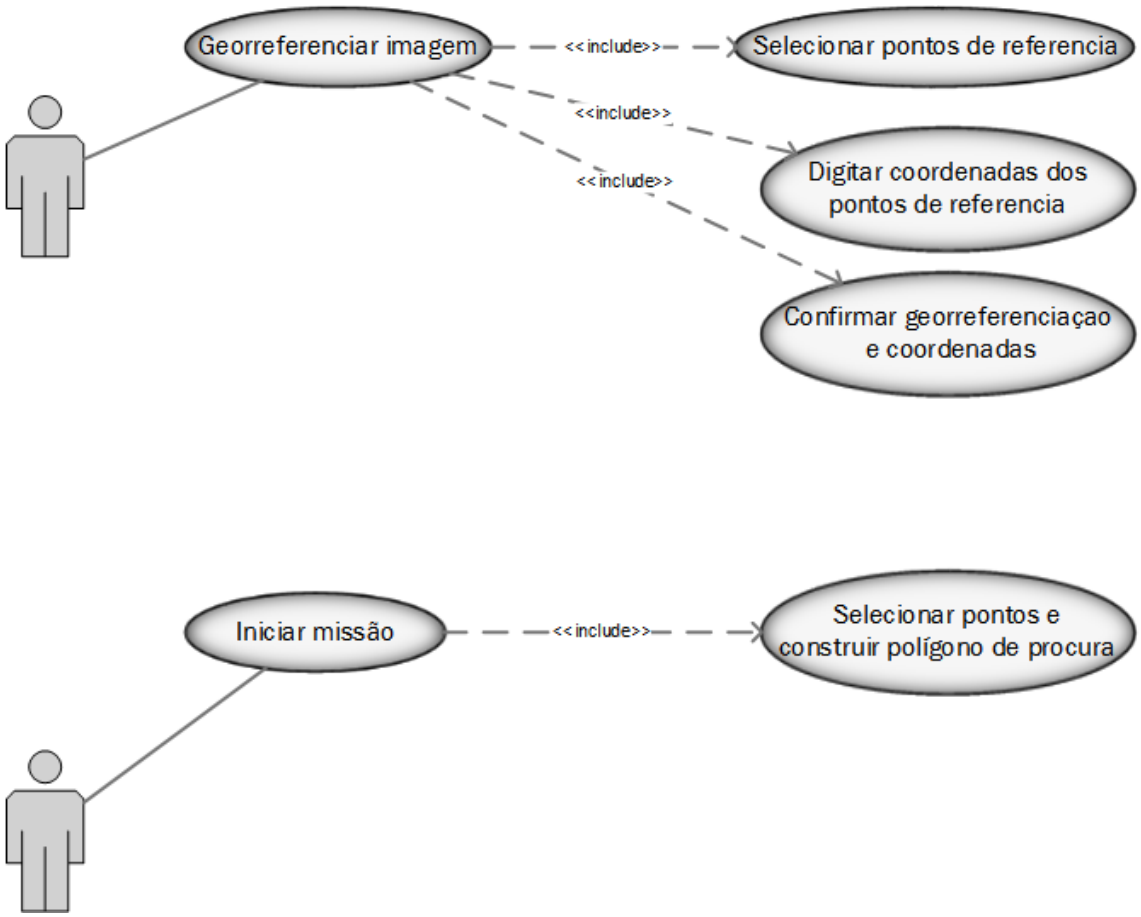


Figura 17 - Diagrama de Casos de Uso

3.6. Estudo de Ferramentas de Trabalho

Aquando da abordagem inicial ao estágio, foram consideradas e discutidas algumas alternativas para a implementação da plataforma de interface gráfica para o FSR Husky. A primeira hipótese estudada foi baseada na linguagem Java, com a tecnologia "Java Desktop Applications" (http://docs.oracle.com/javase/tutorial/uiswing/), mais conhecida como *javaswing*, onde já tinha alguma experiência adquirida no passado aquando do desenvolvimento de aplicações e interfaces gráficos para vários projetos académicos. No entanto, esta opção dificultava a integração com o projeto Husky, que como escrito no capítulo anterior, é baseado em ROS e C++, sendo que o ROS não tem suporte oficial para Java, e as bibliotecas disponíveis estão ainda em fase de teste e experimentação.

De seguida, e em concordância com o colega do Instituto de Sistemas e Robótica que desenvolveu uma aplicação para um detetor de metais manual, foi tentado o uso do Unity3D (http://unity3d.com/), uma plataforma de desenvolvimento de gráficos a duas e três dimensões, o que seria uma boa opção para fazer o *render* (aplicação de técnica que permite melhorar a visibilidade de imagens a duas e três dimensões) das imagens e acompanhar as missões do robô. Esta opção não foi seguida pela razão de que algumas bibliotecas não estavam disponíveis para o Ubuntu, e também outras não tinham a integração esperada com a linguagem C++.

As plataformas baseadas em tecnologias da internet foram também abordadas de forma breve, tenho concluído que o processo de aprendizagem das mesmas dentro do tempo definido para este estágio não era suficiente.

Por fim, a última opção e a que acabou por ser usada para o trabalho proposto por este estágio foi a solução Qt Creator (http://qt-project.org/), uma plataforma de desenvolvimento (IDE) integrada que contém o Qt Designer, para o *design* gráfico de aplicações, o ambiente de programação com suporte para C++, *debugging* (análise de erros de código) e lançamento de versões (para teste com diferentes contextos). Esta plataforma foi primeiramente sugerida pelo Professor José Prado, que sustentado no seu conhecimento e experiência, propôs o Qt Designer para a parte da interface gráfica da plataforma. Ao explorar esta possibilidade, encontrei a possibilidade de não só usar o Qt Designer, como também o Qt Creator, integrando todo o desenvolvimento num só IDE. A grande vantagem desta abordagem foi a integração simplificada com o projeto de ROS do FSR Husky, visto que a linguagem é a mesma (C++), facilitando a comunicação entre processos, e, além disso, o Qt Creator proporciona uma base de programação e desenvolvimento nativa para Linux (Ubuntu) com possibilidade para explorações futuras noutras plataformas para a interface (Android).

O Qt Designer permite a construção de uma interface gráfica, uma camada de apresentação para um programa, através de um sistema de *drag and drop* dos componentes (*widgets*) desejados para uma janela que representa o resultado final da interface. É depois necessária a programação dos eventos que ocorrem aquando da utilização da interface, sendo esta parte desenvolvida no gestor de projeto Qt Creator, em C++, de forma totalmente integrada. O ficheiro de interface tem extensão *ui* e é na

verdade um XML (*Extensible Markup Language*) que é interpretado pelo compilador do Qt.

Cada janela do programa tem o seu próprio ficheiro *ui*, tal como o seu código no ficheiro *cpp* e o ficheiro de declarações (*header file*). A comunicação entre janelas é feita com *Slots*, que permite a ligação entre quaisquer duas funções do projeto, funcionando como uma execução de serviços remotos.

3.7. Metodologia de Desenvolvimento

O projeto seguiu a metodologia de desenvolvimento *Scrum*[25], uma metodologia de trabalho iterativa e incremental, sugerida pelo Professor José Prado, que consiste num conjunto de práticas e papéis definidos para apoiar a decisão no desenvolvimento de *software*. O ponto principal desta ideia é que reconhecendo que os requisitos podem mudar a qualquer altura do ciclo de desenvolvimento, e essas mudanças não são endereçadas com facilidade pelas metodologias tradicionais. Usando uma abordagem empírica, esta metodologia assegura que o trabalho desenvolvido tem em foco a resolução dos problemas e desafios que são colocados ao desenvolvedor. Os papéis do *Scrum* são, em geral, o *Product Owner*, neste caso o Professor Lino Marques, o *Scrum Master*, papel desempenhado pelo Professor José Prado, e a equipa de desenvolvimento, eu. O *Product Owner* representa o promotor do *software*, e o ator que comunica os requisitos e as alterações. O Professor José Prado teve o papel de líder de equipa, com a tarefa de apoiar a decisão e sugerir resoluções para os problemas gerais. Cada fase de iteração de funcionalidades é chamada de *Sprint*, em que no final de cada um os resultados são apresentados e discutidos com os supracitados atores dos papéis *Scrum*.

As soluções para as funcionalidades do *software*, incluindo a sua implementação foram sendo apresentadas ao *Scrum Master* e ao *Product Owner*, que nas reuniões aferiam sobre a valência e eficácia das soluções apresentadas. Sendo o produto destas reuniões, em documentos ou notas usadas para uso posterior neste relatório de estágio.

Capítulo 4 - Trabalho Efetuado

4.1. *Design*

Adotando a metodologia descrita no final do capítulo anterior, os primeiros ciclos de desenvolvimento focaram-se na criação da interface gráfica, a camada de visualização do Diagrama de Camadas apresentado na subsecção respetiva do capítulo três.

Começando pela Janela Principal (*mainwindow.ui*), e, usando o módulo de desenvolvimento do Qt Creator dedicado ao *design* da interface gráfica do programa (Qt Designer), procedi à passagem do protótipo em papel exposto na figura 14 para um protótipo não funcional, ou seja, foi apenas executado o *design* da aplicação, sem funcionalidades da camada de lógica.

Nesta passagem tomei a decisão de colocar os botões principais da aplicação na vertical, do lado esquerdo, isto deve-se a estas opções serem transversais ao sistema, e deverem estar sempre disponíveis, mesmo quando outras janelas são inicializadas, nesse sentido, é mais acessível a sua colocação junto à barra de tarefas (à esquerda, por defeito, no Ubuntu) e deixando o espaço de segmentação vertical para o sistema de abas do modo avançado, maximizando o espaço e a usabilidade. Assim temos uma clara distinção entre o agrupamento das abas, e os botões principais, que se mantêm sempre presentes.

Para executar esta passagem, criei uma barra de ferramentas onde inseri os ícones (*area.o icons*) e o texto de cada botão, tal como as respetivas opções, como por exemplo, o botão de *Start* e *Stop* serem *checkable*, ou seja, terem dois estados, ligado e desligado, e não apenas um simples botão de ação.

Esta versão do *design* foi bem recebida pela equipa de investigadores e manteve-se até à versão final, e da qual temos um exemplo na figura 18.

Além destas opções, foram também usados menus de contexto, que se acedem a partir da barra do programa e permitem escolher variadas opções, como "Abrir imagem", "Guardar imagem", "Georreferenciar imagem", "Configurações" e menu de ajuda. Estas listas foram criadas através do Qt Designer no espaço próprio para a sua formulação.

De seguida, num *container* (caixa que contém outros *widgets*) criei um componente de abas, descrevendo cada uma de acordo com o especificado na análise de requisitos para o modo avançado, concebendo depois para cada uma o *design* respetivo, em conformidade com as maquetes apresentadas em anexo. Os *widgets* usados foram: *Push Button* (botões), *Tab Widget* (abas), *Line Edit* (linha de texto editável), *Text Edit* (corpo de texto editável), *Spin Box* (campo incremental), *Horizontal* e *Vertical Slider* (controle deslizante horizontal e vertical) e *Label* (para mostrar texto e imagens estáticas).

Com o *design Sprint* finalizado, revisto, e aceite pelos orientadores e promotores deste estágio, passei à exploração e implementação de funcionalidades.

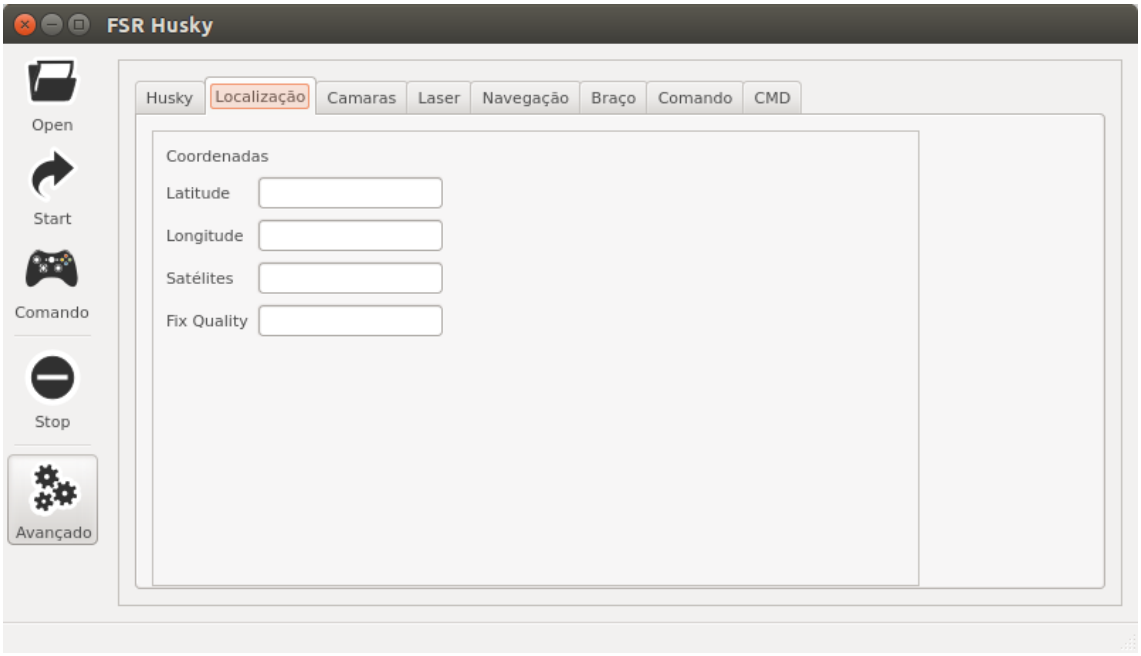


Figura 18 - Design da Janela Principal

4.2. Uso de Recursos

Para guardar os ícones, e outros recursos, utilizados na aplicação usei uma valência do Qt Creator que permite definir ficheiros de *Resources*. Estes fazem parte da fonte do programa, e quando este é compilado são automaticamente convertidos em binário e guardados num ficheiro C++ de modo a serem portáveis sem necessitar de exportar os sistemas de ficheiros. O ficheiro usado é o "icons.qrc".

Outros *Resource Collection Files* (*qrc*) usados no projeto foram:

- "config.qrc" - Ficheiro que guarda as configurações da plataforma.
- "html.qrc" - Ficheiro que contém o código fonte, em HTML e *JavaScript* (JS) usados nos testes e explorações feitas com a API do Google Maps.
- "images.qrc" - Arquivo das imagens usadas para a interface, excluindo as imagens de missões do Husky. As imagens guardadas são: Logotipo do ISR, logotipo do TIRAMISU, imagem a duas dimensões representativa do Husky.

Noutro âmbito, usei também uma biblioteca do Qt, denominada *QSettings*, em que, dependendo do sistema operativo, são guardadas as variáveis como variáveis de sistema, o que facilita a transição entre sessões e mantém os dados organizados e seguros. Uma instância do *QSettings* é criada com um nome de sistema, neste caso "Husky", e, usando esse nome podemos adicionar e ler valores de qualquer parte do programa. Os valores guardados nesta estrutura são o raio do sensor, e a última imagem usada para uma missão.

4.3. Funcionalidades

Para dotar a interface criada de funcionalidades, passei à implementação do código de gestão de eventos dos diferentes componentes da janela principal.

Iniciei a implementação dos eventos e das janelas mais simples, o menu de contexto e as opções: janela de configurações e janela de informação da aplicação.

Configurações e Informação

Na primeira, foi apenas necessário criar uma nova janela (*configdialog.ui*) no qual foi colocada uma *Label* com o nome da opção, e uma *Spin Box* para a seleção do valor para o tamanho do sensor. Quando é alterado o valor da *Spin Box* e confirmada a ação (carregar no "OK" da janela de configurações), um sinal é enviado da classe *ConfigDialog* por um *Slot* para o código da janela principal, em *mainwindow.cpp* que é reenviado para uma função que guarda o valor no *QSettings* supracitado.

A ligação entre o *Signal* e o *Slot* declarada no inicio da classe *mainwindow.cpp*, onde "cd" é um apontador para a instancia da classe *configDialog*:

```
QObject::connect(cd, SIGNAL(radiusChange(double)), this,
SLOT(setSensorRadius(double)));
```

E a emissão do sinal na classe *configDialog*, que executa quando as mudanças na *Spin Box* são aceites pelo utilizador:

```
void ConfigDialog::on_buttonBox_accepted() {
emit radiusChange(ui->sensorRadius->value()); }
```

No caso da janela de informação, foram apenas utilizados os recursos contidos no ficheiro *qrc* correspondente aos arquivos de imagens em *Labels*, para apresentar um resumo do projeto FSR Husky dentro do contexto TIRAMISU.

Em seguida conclui uma janela de diálogo simples, para comunicar erros ou notificações da plataforma para o operador. Como por exemplo tentar iniciar uma missão com uma imagem que ainda não tenha sido georreferenciada, ou no término da missão do Husky. Esta janela (*alertdialog.ui*) contém apenas uma *Label*, sendo que a classe A*lertDialog* implementa uma função para definir o texto da *Label*, o que permite criar uma instância desta classe e definir a mensagem da janela de qualquer ponto do projeto:

```
void AlertDialog::setLabel(QString qstr) {
ui->alertLabel->setText(qstr); }
```

Georreferenciação

Na exploração de possibilidades para a representação de uma imagem ou mapa na aplicação, considerei integrar o Google Maps na plataforma, o que daria possibilidade de obter todos os mapas diretamente da API dos mapas do Google e acesso a funções para

desenhar e marcar zonas diretamente no mapa. Nesse sentido, criei uma nova classe, denominada *MapDialog* que na sua janela apenas continha um *Widget QWebView*, que permite visualizar conteúdo *Web*, com um interpretador similar a um *browser* (navegador de internet). Usando o ficheiro HTML contido no ficheiro *html.qrc*, foi possível correr o Google Maps na plataforma, mas rapidamente se mostrou altamente falível e com problemas ao nível da performance. O facto de esta solução depender de uma ligação à internet foi motivo suficiente para abandonar o seu desenvolvimento, mas, além disso, a performance do mapa quando desenhado com objetos para marcar o robô, as minas e outros explosivos era insatisfatória. O código fonte desta exploração encontra-se em anexo.

Com os problemas encontrados na proposição anterior, decidi criar uma solução com base nas ferramentas oferecidas pelo Qt e as suas *Widgets*. A *QGraphicsView* é um módulo que permite mostrar imagens, ilustrar, desenhar e manipular todos os objetos contidos na *QGraphicsScene* que é instanciada dentro desse *Widget*. Criada a cena, podemos carregar uma imagem e desenhar sobre a mesma, o que reuniu todas as condições para implementar os requisitos funcionais da plataforma de interface gráfica. Esta tornou-se a base para a janela de georreferenciação e para a janela de missão.

```
scene = new QGraphicsScene(this);
QPixmap pic = QPixmap(mw->getMapPic());
scene->addPixmap(pic);
ui->graphicsView->setScene(scene);
```

O excerto de código acima representado mostra como se coloca uma imagem na janela de georreferenciação, criando uma nova *QGraphicsScene*, declarando uma imagem, indo buscar a mesma à *MainWindow* (*mw*), adicionando a imagem à cena, e finalmente enviando a cena para a janela, na última linha apresentada. Tendo esta base, foi possível desenvolver a parte mais importante de todo o sistema, a georreferenciação de uma imagem e a marcação de uma missão para o Husky de uma forma eficiente e sem problemas de performance. Esta abordagem, permitiu, em testes, desenhar mais de dois mil objetos na imagem, sem perder a fluidez de arrastamento ou de *zoom*, uma vantagem esmagadora em relação à abordagem anterior.

Para a georreferenciação da imagem, optei por criar um filtro de eventos, ou seja, uma função que capta todos os eventos que são desencadeados na *graphicsView*. Isto é feito é possibilitado através da seguinte declaração:

```
ui->graphicsView->viewport()->installEventFilter(this);
```

Sendo que a filtragem propriamente dita é feita na função criada para o efeito. Nessa filtragem, o objetivo é captar apenas os cliques com o botão direito do rato e os movimentos da roda do rato ou equivalentes (*scrolling*), visto que os cliques com o botão esquerdo estão reservados para arrastar a imagem, sendo que essa propriedade é definida na declaração da *QGraphicsView*. Apanhando os eventos de *scrolling* e definindo um fator de multiplicação, neste caso 1,15, podemos aumentar e diminuir o *zoom* da imagem, aumentando ou diminuindo o tamanho da imagem em 15% a cada evento, ou seja, a cada passo do *scrolling*.

O evento de captura do botão direito do rato está dividido em duas partes, o aperto do botão e a sua libertação. No Qt é sempre necessário tratar estes dois eventos, pois eles são

sempre considerados dois eventos separados, e é possível captar um sem o outro. No aperto do botão direito, é colocada uma marca na imagem no local onde se encontra o rato, esta marca é na realidade uma elipse sobreposta na imagem, que pode ser movida dentro da cena e a sua posição guardada. Podem no máximo ser colocadas duas elipses, o que corresponde aos dois pontos necessários para fazer a georreferenciação de uma imagem.

Na libertação do botão esquerdo do mapa, é feita uma atualização das posições das elipses, pois sendo possível arrastar e mudar uma elipse de local, ação que é realizada com o botão esquerdo, quando o arrastamento é finalizado e é levantado o botão, é feita essa atualização.

Para cada uma das elipses existe uma classe chamada *CoordDialog* que não é mais do que uma janela com dois campos de introdução de texto, limitada a valores *floating point*, para a introdução das coordenadas GPS dos pontos representados pelas marcas. A gestão destas janelas é feita por contadores e outros recursos da linguagem para a proteção da integridade dos dados e do programa. Se a imagem já estiver georreferenciada, os pontos que foram originalmente mapeados são automaticamente colocados na imagem, sendo ambos passiveis de alteração.

Quando ambas as marcas forem colocadas, e tiverem sido digitadas e aceites (evento de *ButtonAccepted*) as coordenadas de ambas nas janelas *CoordDialog* respetivas, é lançada uma confirmação final dos dados inseridos, que quando aceite envia os dados das coordenadas em pixéis e as coordenadas em GPS de volta para a classe *MainWindow* através do sinal respetivo.

Para a georreferenciação em si, abordei o problema primeiramente com uma solução inoperante, que quando usada tinha um erro de várias dezenas de pixéis na transformação das coordenadas de GPS em coordenadas da imagem, em pixéis. Esta solução, baseada em transformações lineares, não era a mais apropriada para o problema, pois o sistema de coordenadas GPS usa a latitude e a longitude como medidas de posição, as quais não são lineares, e sim baseadas na Projeção de Mercator[26] para a representação da Terra. A distância em metros entre uma casa decimal de latitude depende da sua posição no globo, logo, é necessário transformar esta diferença numa distância real. Para isso usamos a medida da circunferência da Terra na longitude em questão, dessa forma podemos fazer uma relação entre a distância em longitude e a distância linear para essa latitude. Usando os dois pontos de referência marcados pelo operador como vértices de um retângulo, podemos, com um erro extremamente diminuto (menor a um pixel), transformar as coordenadas, usando os pontos inicialmente marcados pelo utilizador como georreferenciação da imagem. Da mesma forma, a conversão inversa, é apenas a transposição das operações matemáticas, que apresento em seguida, onde as coordenadas a converter são *lat* e *long*:

```
double rectWidth = fabs(getCoordAuxX() - getCoordOriginX());
double rectHeight = fabs(getCoordAuxY() - getCoordOriginY());
double deltaLon = fabs(getGPSOriginLon() - getGPSAuxLon());
double LatDegree = getGPSAuxLat() * M_PI / 180;
xPix = (lon - getGPSOriginLon()) * (rectWidth / deltaLon);
double latid = lat * M_PI / 180;
double worldWidth = rectWidth / deltaLon * 360 / (2 * M_PI);
double offsetY = (worldWidth / 2 * log((1 + sin(LatDegree)) / (1 -
sin(LatDegree))));
yPix = rectHeight - ((worldWidth / 2 * log((1 + sin(latid)) / (1 -
sin(latid)))) - offsetY);
```

As coordenadas auxiliares e originais correspondem às duas marcas feitas pelo operador aquando da georreferenciação da imagem.

É guardado num ficheiro YAML, com o mesmo nome da imagem, mas com extensão "yml", as coordenadas GPS e em pixéis das duas marcações. Dessa forma, quando uma imagem é selecionada, se existir um ficheiro é porque a mesma já se encontra georreferenciada, e essa informação é usada para saltar este passo, não existindo necessidade de voltar a referenciar uma imagem, a menos que se queira corrigir as marcas.

Missão

Usando a mesma base descrita na subsecção anterior, os eventos captados para o *scrolling*, botões direito e esquerdo do rato são os mesmos, mas agora temos um máximo de cinquenta objetos, elipses, ao contrário do máximo de dois para a georreferenciação. A cada objeto colocado com o botão direito, é adicionado num vetor o ponto (coordenadas *x* e *y* em pixéis) correspondente à elipse. A cada inserção de nova elipse, ou modificada a posição de algum dos objetos já colocados, o vetor é usado para desenhar um polígono com os vértices que condizem com os pontos colocados nesse mesmo vetor, com a condicionante que é necessário que o primeiro e último elemento sejam iguais, de modo a construir um polígono fechado.

```
polyone = scene->addPolygon(poly, pen, brush);
```

Onde *poly* é o vetor, *pen* é a cor do contorno do polígono e *brush* a cor do interior do polígono.

Quando é pressionado o botão de *Start Mission*, na janela *MapMission*, as elipses passam a inamovíveis, a cor do polígono de procura muda para verde e as coordenadas dos vértices da área de procura são enviadas para a *MainWindow*, que os converte em coordenadas GPS e os guarda num ficheiro YAML, na diretoria do sistema de ficheiros correspondente a uma pasta que é monitorizada pelo sistema ROS do husky. Este procedimento de envio das coordenadas para o robô precede a realização deste estágio, sendo que foi uma nova limitação para o projeto.

Coincidentemente com essa ação, é enviada para a classe *Sender* uma mensagem com a ordem de início de missão, que é encaminhada para o robô.

Esta classe tem também funções para o desenho do robô na imagem, tal como das deteções (positivos e falsos positivos), recebendo como parâmetro a localização respetiva.

Para esse efeito, criei uma representação do Husky em *pixel art* (forma de arte digital que cria imagens com edição pixel a pixel) para a sua colocação na janela de missão. As minas são representadas, de acordo com a configuração do tamanho do sensor, através de um circulo sobreposto na imagem.

Sender e *Listener*

Para a comunicação com o robô, foi necessário desenvolver dois nós em ROS, que integrados no projeto geral do Husky, facilitaram a interação entre as plataformas. A tecnologia que usei para fazer esta ligação foi o *named pipe*, em detrimento de outras opções como *shared memory*, *memory mapped files*, ou *sockets*. A escolha da solução que adotei prende-se com vários fatores, o principal dos quais, a escrita atómica das mensagens a enviar (a garantia que uma mensagem é enviada e lida como um todo, nunca parcialmente), o que é proporcionado pelo *named pipe*, visto que é possível configurar o número de *bytes* a escrever e ler do mesmo. Um *named pipe* é uma estrutura do estilo FIFO (*first in first* out) que permite a comunicação entre processos, com escrita e leitura. Usualmente, um escritor e um leitor comunicam unidireccionalmente através do *pipe*. Apesar da existência de um ficheiro com o nome do *pipe*, as transações acontecem na memória, o que proporciona uma comunicação rápida, ao contrário dos *memory mapped files* que muitas vezes pecam por lentidão. No caso da *shared memory*, memória partilhada, tem o problema de sincronismo, em que é imperativo ter um sistema de semáforos e outras proteções para manter a integridade dos dados. As *sockets* seriam apenas um último recurso, visto que a comunicação inter-processual que era necessária, se desencadeia na mesma máquina.

Criei então uma *package* do ROS, denominada "info_node_merger", que foi integrada na *metapackage* (conjunto de packages de um projeto) do Husky e corre sempre que é inicializado o robô. Esta *package* contém dois nós:

- "info_node_merger_node"
- "joy_publisher"

O primeiro tem como missão principal subscrever todos os tópicos relevantes e organizar a informação numa só mensagem. Deste modo é necessário apenas um *pipe* para receber a informação vinda dos tópicos do Husky. O *pipe* tem o nome de "receiverPipe" e se não existir, pode ser criado tanto na classe *Listener* do projeto da plataforma de interface gráfica, como no próprio nó. Os valores de cada campo estão formatados para a uniformização da mensagem (uma lista de valores), deste modo, a mensagem tem sempre o mesmo tamanho, e assim poder ser assegurada a atomicidade da mesma. A subscrição dos tópicos pelo nó:

```
ros::Subscriber joy_sub_ = n.subscribe<sensor_msgs::Joy>("joy", 1,
&joyCallback);
ros::Subscriber joint_states_sub =
n.subscribe<sensor_msgs::JointState>("/joint_states", 1,
&jointStateCallback);
ros::Subscriber gps_fix_sub = n.subscribe("/gps/fix", 1,
gpsFixCallback);
```

```
ros::Subscriber gps_status_sub = n.subscribe("/gps/status", 1,
gpsStatusCallback);
ros::Subscriber laser_state_sub_ =
n.subscribe<std_msgs::String>("laser_state", 1, &laserStateCallback);
ros::Subscriber coils_sub_ = n.subscribe("coils", 1, &coilCallback);
ros::Subscriber move_goal_sub_ = n.subscribe("/move_base_simple/goal",
1, &moveGoalCallback);
ros::Subscriber imu_sub_ =
n.subscribe<sensor_msgs::Imu>("/imu/data_corrected",1,&ImuCallback);
ros::Subscriber ekf_sub_ =
n.subscribe("/robot_pose_ekf_cp/odom",1,&ekfCallback);
ros::Subscriber husky_power_sub_ =
n.subscribe("/husky/data/system_status",1,&huskyPowerCallback);
ros::Subscriber husky_encoder_sub_ =
n.subscribe("/husky/data/encoders",1,&huskyEncodersCallback);
ros::Subscriber husky_speed_sub_ =
n.subscribe("/husky/cmd_vel",1,&huskyCmdvelCallback);
```

Por outro lado, o *joy_publisher* tem o objetivo de enviar as ordens para o robô. A razão pela qual o nó tem essa denominação é porque as ordens são enviadas simulando mensagens enviadas pelo modulo do comando do Husky, isto é, como escrito previamente na introdução do ROS, os publicadores e os subscritores dos tópicos não têm conhecimento mútuo, podendo a escrita no tópico que contém as mensagens com as teclas do comando, vir do próprio comando ou de outro qualquer nó que escreva nesse tópico, que é o casso do *joy_publisher*.

A mensagem do tópico do *joystick* é a seguinte:

```
header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 0
  frame_id: ''
axes: [0.0, 0.0, 1.0, 0.0, 0.0, 1.0]
buttons: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Onde os campos que foram relevantes para o desenvolvimento da plataforma são os últimos dois, vetores que contêm os eixos e os botões do comando. Ao publicar uma mensagem deste tipo, podemos definir qualquer combinação de teclas, o que me permitiu, de acordo com as especificações criadas pelo grupo de desenvolvimento do Husky no ISR, completar todas as funcionalidades da interface. Podemos então assim mapear as ações com as teclas:

- Parar (STOP) - Tecla “B”, que corresponde ao segundo elemento do vetor de botões. Esta tecla é enviada quando o evento na *MainWindow* relativo à ação “STOP” é acionado. O envio da mensagem de libertação do botão “B” só é enviado quando o operador volta a pressionar o botão “STOP” e o desliga.
- *Start mission* - Tecla “Start”, que corresponde ao oitavo elemento do vetor. O sistema do Husky recebe a ordem de início de missão e procura no seu sistema o ficheiro YAML que contém as coordenadas dos vértices da área de procura gerado pela plataforma de interface gráfica.
- Braço - Teclas “A”, “X” e “Y”, correspondendo aos elementos um, três e quatro do vetor de botões. Estas teclas controlam o braço do robô, ordenando a subida e descida do mesmo com as teclas “A” e “X” respetivamente. A tecla “Y” coloca o

braço na sua altura máxima, funcionalidade usada para ultrapassar obstáculos no terreno.

O *pipe* usado para esta comunicação chama-se "senderPipe" e se não estiver criado no sistema, é inicializado na classe *MainWindow*, no início da execução da plataforma de interface gráfica.

```
int fifo = mkfifo("/tmp/senderPipe", S_IWUSR | S_IRUSR | S_IRGRP |
S_IROTH);
```

A classe *Sender* faz uso do *pipe* escrevendo as mensagens no mesmo:

```
void Sender::write2Pipe(QString arg1) {
if ((fifo = open("/tmp/senderPipe", O_WRONLY)) < 0) {
return; }
if ((num = write(fifo, arg1.toLocal8Bit().constData(),
arg1.toLocal8Bit().size()+1)) < 0) {
return; }
close(fifo); }
```

O nó recebe as mensagens do outro lado, interpretando-a e publicando a mensagem ROS para o Husky:

```
if ((fifo = open("/tmp/senderPipe", O_RDONLY)) < 0) {
return -1; }
if ((num = read(fifo, temp, sizeof(temp))) < 0) {
return;}
.
.
.
joy_pub.publish(new_joy_command);
ros::spinOnce();
close(fifo);
```

A penúltima linha deste código simplificado é a que faz publicar todas as mensagens que o nó tem em espera, neste caso, a mensagem a publicar é a "new_joy_command", que contém a combinação de teclas de acordo com a mensagem recebida pelo *pipe*, enviada pela plataforma de interface gráfica.

4.4. Resultados e Testes

Os resultados estéticos da aplicação desenvolvida encontram-se em anexo, na secção denominada Imagens do Funcionamento da Plataforma.

O *software* desenvolvido foi amplamente testado em ambiente de laboratório, e, em algumas ocasiões, no terreno. Os testes focados no *design* da plataforma, nas funcionalidades do programa e na comunicação entre o Husky e a interface gráfica foram na sua totalidade conduzidos na estação de trabalho do ISR. De realçar que o próprio Husky continua em desenvolvimento, tendo em conta essa limitação, os testes no terreno não foram exclusivos para a avaliação do *software* desenvolvido em propósito deste estágio.

Figura 19 - Testes no terreno com o Professor Lino Marques, Professor Sedat Dogru e Baptiste Gil

Os testes efetuados são apresentados na seguinte tabela:

Teste	Resultado	Observações
Carregar Imagem	Sucesso	A aplicação carrega todos os tipos de imagem relevantes. Imagens muito grandes (milhares de pixéis de largura ou altura) podem gerar comportamentos estranhos.
Georreferenciar a Imagem	Sucesso	É possível georreferenciar qualquer imagem carregada a partir de dois pontos de GPS capturados com a antena. Os pontos não podem ser coincidentes. Quanto menor a distância entre os pontos, menor a precisão, recomenda-se que a área de procura esteja contida dentro do quadrilátero formado pelas marcas.
Corrigir ou anular uma Georreferenciação	Sucesso	Mudar os pontos, a coordenada GPS ou simplesmente cancelar a georreferenciação é suportado pela plataforma. O ficheiro YAML é reescrito em caso de a imagem já ter sido referenciada.
Guardar a Georreferenciação para uso futuro	Sucesso	Um ficheiro YAML é gerado no sistema com os dados da georreferenciação. Quando uma imagem já referenciada pelo sistema é escolhida, o ficheiro YAML é lido e a referenciação é feita automaticamente. Esta valência é suscetível à remoção manual do ficheiro YAML.
Marcar Terreno a procurar	Sucesso	A plataforma permite colocar 50 pontos na imagem de modo a construir um polígono de procura. É possível também reposicionar qualquer um dos vértices.

Iniciar Missão	Sucesso	Depois de selecionado o polígono, a aplicação recebe a ordem de início de missão de procura com varrimento, enviando a ordem para o Husky. O ficheiro YAML com os vértices da área de procura é colocado na pasta correta.
Supervisão da Missão	Parcial	É possível seguir a trajetória do Husky na imagem e também das deteções de engenhos. No entanto a precisão das localizações é baixa. Este facto prende-se com o funcionamento do robô e da sua posição GPS que é recebida pela plataforma.
Terminar Missão	Parcial	Alguns dados não são passiveis de serem guardados, enquanto outros não estão a ser guardados corretamente. A área que o Husky efetivamente varreu na sua procura não está a ser guardada, são demasiados dados para tratamento.
Guardar Mapa de Final de Missão	Sucesso	No final da missão é gerado uma imagem com a área de procura e as deteções de engenhos sobrepostas.
Botão de Emergência	Sucesso	Quando é acionado o botão de paragem, é imediatamente enviada uma mensagem ao Husky. A missão pode ser retomada desligando o botão de STOP na interface gráfica.
Definir tamanho do sensor	Sucesso	Opção encontra-se na janela de configurações. A configuração mantem-se entre sessões.
Modo Avançado	Sucesso	Opção encontra-se nos menus de contexto da janela principal. Tal como o botão STOP, pode ser ligado e desligado.
Módulos do Modo Avançado	Parcial	A ligação entre a plataforma e o Husky faz-se por mensagens do ROS e posteriormente, via *named pipe*, As mensagens ROS são extensas e complexas, a interpretação dos dados corretos pode conter erros não identificados nos testes. Mensagens de imagens não funcionais. Mensagens do laser não funcionais. Taxa de transmissão de dados fixada em 10 *Hertz*.
Modo Avançado - Comando	Falhou	Não é possível enviar ordens através do comando simulado. No entanto, o suporte para todo o tipo de combinação de teclas está disponível, faltando uma interface para as acionar.
Modo Avançado - Terminal	Parcial	O terminal encontra-se em funcionamento para o envio de mensagens para o terminal. O terminal da plataforma é do tipo *Shell* o que impossibilita o uso de certos comandos mais complexos. Pode ser resolvido se o terminal usado for do tipo *bash*.
Falha de ligação sem fios	Falhou	Não basta que se verifique a ligação, é necessário também que o robô esteja ligado e a base do ROS esteja em execução. A plataforma não deteta a rede sem fios.
Zoom nas imagens	Sucesso	Funcionamento total.
Arrastar as imagens	Sucesso	Funcionamento total.
Movimentação dinâmica	Sucesso	Funcionamento total.

4.5. Desenvolvimentos Futuros

A possibilidade de novas funcionalidades não está posta de parte, sendo que o desenvolvimento programado para este estágio foi atingido no seu todo, no entanto é possível que novos requisitos sejam criados, o que pode levar a novos módulos e classes.

Poderá existir a intenção de abordar outras plataformas e suportar diferentes sistemas operativos, nomeadamente WINDOWS, ANDROID ou iOS. Nestes últimos dois casos, seria necessário também desenvolver um aplicação móvel para telemóveis e/ou *tablets*. Existe já em desenvolvimento uma aplicação semelhante no ISR para um detetor de metais manual. Seria interessante se fosse possível integrar ambos os projetos.

A teleoperação do veículo é feita neste momento com o comando XBox360. Outros métodos de teleoperação podem vir a ser incluídos na plataforma que desenvolvi, como uma interface que permita a teleoperação do sistema através da aba do comando no modo avançado, algo que foi proposto numa fase já muito adiantada do trabalho e que não foi possível implementar.

A ligação entre a plataforma e o FSR Husky é feita por rede sem fios local que funciona a partir do veículo. Seria um desenvolvimento interessante poder supervisionar as missões do robô a partir de fora desta rede. Essa possibilidade existe usando a plataforma como retransmissor dos dados, acedendo por um *browser* em qualquer parte do mundo.

Capítulo 5 - Conclusões

O trabalho desenvolvido ao longo deste estágio complementou um projeto já existente mas que se encontrava bastante limitado ao nível da interação do robô com o utilizador final. Creio que os progressos que fiz neste campo aumentaram o valor do FSR Husky como produto e fornecem ao mesmo tempo a possibilidade de demonstrar as capacidades do robô de forma mais eficaz dentro do ecossistema do projeto TIRAMISU.

O propósito principal deste estágio era a conceção de uma interface gráfica para o FSR Husky que permitisse a um operador, sem formação em tecnologias da informação e sem conhecimentos de robótica, colocar em serviço e supervisionar as tarefas de procura por varrimento do robô. Este objetivo foi concretizado na sua totalidade.

Referências

[1] Y. Baudoin e M. K. Habib, *Using Robots in Hazardous Environments: Landmine Detection, De-Mining and Other Applications*. Elsevier, 2010.

[2] M. K. Habib, «Humanitarian demining: Reality and the challenge of technology-the state of the arts», *Int. J. Adv. Robot. Syst.*, vol. 4, n. 2, pp. 151–172, 2007.

[3] I. C. to B. Landmines, *Landmine Monitor Report*. Human Rights Watch, 2014.

[4] F. Naota e H. Shiegeo, «Development of Mine Hands: Extended Prodder for Protected Demining Operation.pdf», 2005. [Em linha]. Disponível em: http://www.expo21xx.com/automation21xx/18224_st3_university/842_Mine_Hands_Extended_Prodder_Protected_Demining.pdf. [Acedido: 26-Jan-2015].

[5] R. Cross, «Landmines must be stopped: Chapter VI Mine Clearance», *Spec. Broch. Int. Comm. Red Cross Geneva Switz.*, 1995.

[6] «FP7-Tiramisu.» [Em linha]. Disponível em: http://www.fp7-tiramisu.eu/. [Acedido: 27-Jan-2015].

[7] «Husky UGV - A Rugged All-Terrain Robot. A Mobile Robot for Any Terrain», *Clearpath Robotics Inc.* [Em linha]. Disponível em: http://www.clearpathrobotics.com/husky/. [Acedido: 14-Jul-2015].

[8] G. Cabrita, «The FSR Husky.» ISR, 2014.

[9] M. A. Goodrich e A. C. Schultz, «Human-Robot Interaction: A Survey», *Found. Trends® Hum.-Comput. Interact.*, vol. 1, n. 3, pp. 203–275, 2007.

[10] T. B. Sheridan, *Telerobotics, Automation, and Human Supervisory Control*. MIT Press, 1992.

[11] «Rover - Mars Science Laboratory.» [Em linha]. Disponível em: http://mars.nasa.gov/msl/mission/rover/. [Acedido: 02-Fev-2015].

[12] T. B. Sheridan, *Humans and automation: system design and research issues*. John Wiley, 2002.

[13] E. Stumm, «ROS/Web Based Graphical User Interface for the sFly Project.pdf», 2010. [Em linha]. Disponível em: http://students.asl.ethz.ch/upl_pdf/289-report.pdf. [Acedido: 27-Jan-2015].

[14] K. Furuta e J. Ishikawa, *Anti-personnel Landmine Detection for Humanitarian Demining: The Current Situation and Future Direction for Japanese Research and Development*. Springer Science & Business Media, 2009.

[15] «Minesweepers: Towards a Landmine-free World.» [Em linha]. Disponível em: http://www.landminefree.org/index.php/en/. [Acedido: 26-Jan-2015].

[16] «Minesweepers: Towards a Landmine-free World | Robohub.» [Em linha]. Disponível em: http://robohub.org/minesweepers-towards-a-landmine-free-world/. [Acedido: 26-Jan-2015].

[17] Š. Havlík, «Unmanned Robotic Vehicles for Demining», *Slovak Acad. Sci.*, 2011.

[18] «DeptOfAutomaticControl-SILO6-IAI-CSIC.» [Em linha]. Disponível em: http://www.car.upm-csic.es/fsr/provisional/silo6/SILO6_Controller.htm. [Acedido: 20-Jul-2015].

[19] C. J. Abate, «DIY Solar-Powered, Gas-Detecting Mobile Robot», *Circuit Cellar*. .

[20] Procerus Technologies, «Ground Control Station Virtual Cockpit v2.6.pdf», 2010. [Em linha]. Disponível em: http://www.lockheedmartin.com/content/dam/lockheed/data/ms2/documents/procerus/Virtual_Cockpit_2.6_11_18_08.pdf. [Acedido: 02-Fev-2015].

[21] «Gazebo.» [Em linha]. Disponível em: http://gazebosim.org/. [Acedido: 20-Jul-2015].

[22] «GIGABYTE - Desktop PC - Mini-PC Barebone - GB-XM1-3537 (rev. 1.0).» [Em linha]. Disponível em: http://www.gigabyte.com/products/product-page.aspx?pid=4581#ov. [Acedido: 17-Jul-2015].

[23] «ROS.org | Powering the world’s robots.» .

[24] «Bjarne Stroustrup’s Homepage.» [Em linha]. Disponível em: http://www.stroustrup.com/. [Acedido: 19-Jul-2015].

[25] «Home | Scrum Guides.» [Em linha]. Disponível em: http://www.scrumguides.org/. [Acedido: 19-Jul-2015].

[26] «Mercator’s Projection.» [Em linha]. Disponível em: http://www.math.ubc.ca/~israel/m103/mercator/mercator.html. [Acedido: 20-Jul-2015].

Anexos

Maquetes da Interface Gráfica

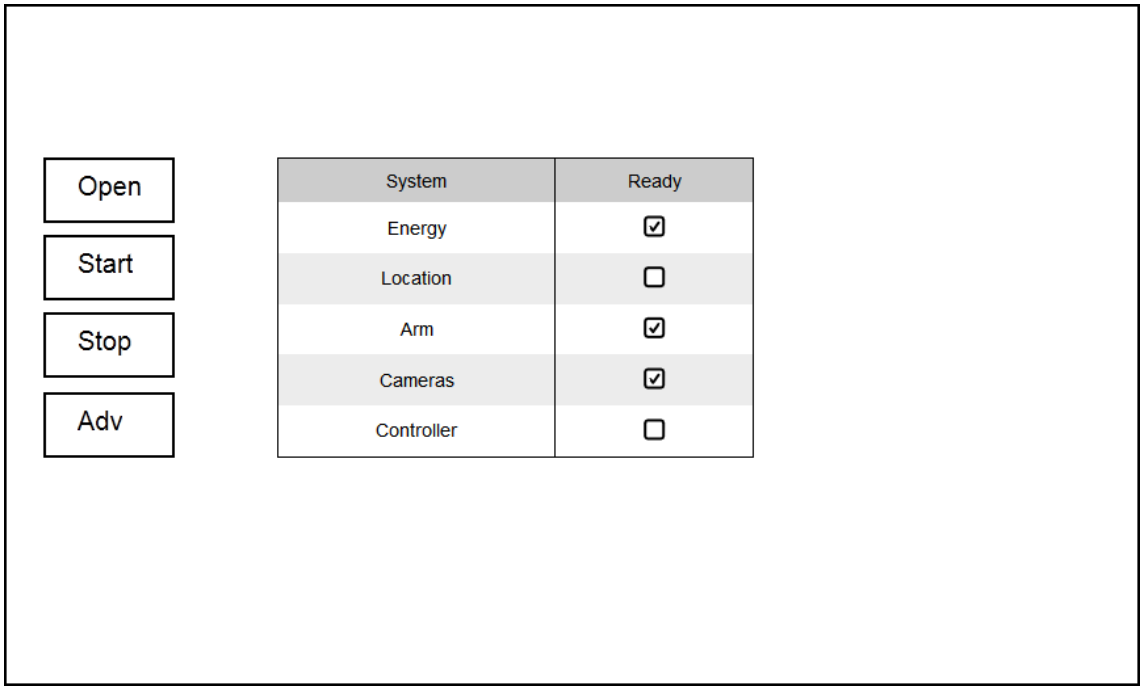


Figura 20 - Maquete da Janela Principal

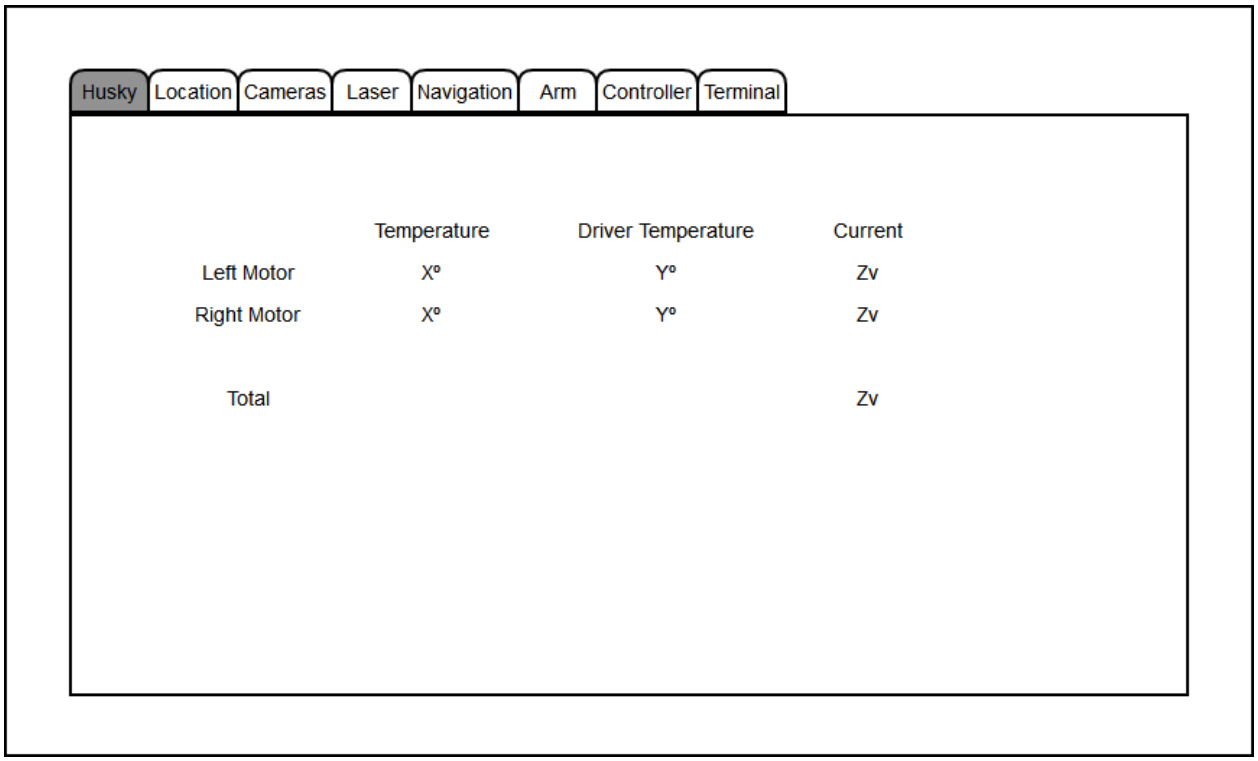


Figura 21 - Maquete do Modo Avançado - Husky

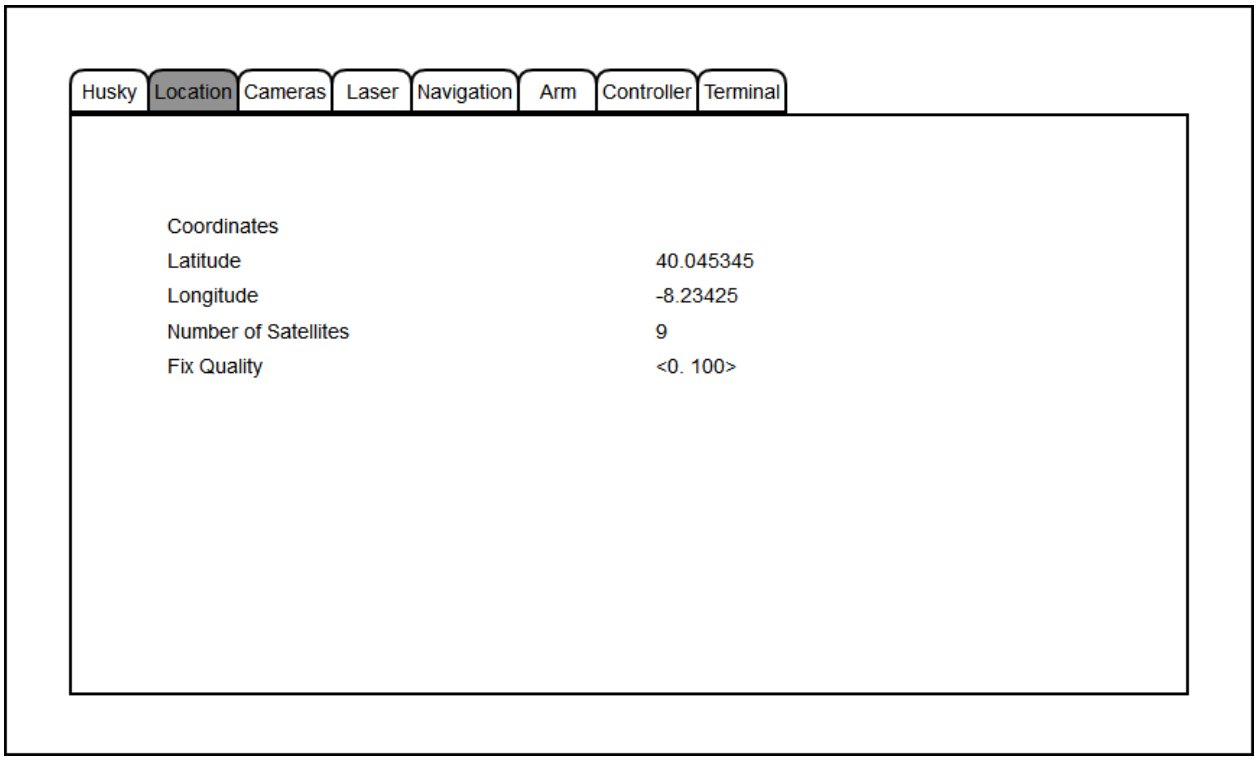


Figura 22 - Maquete do Modo Avançado - Localização

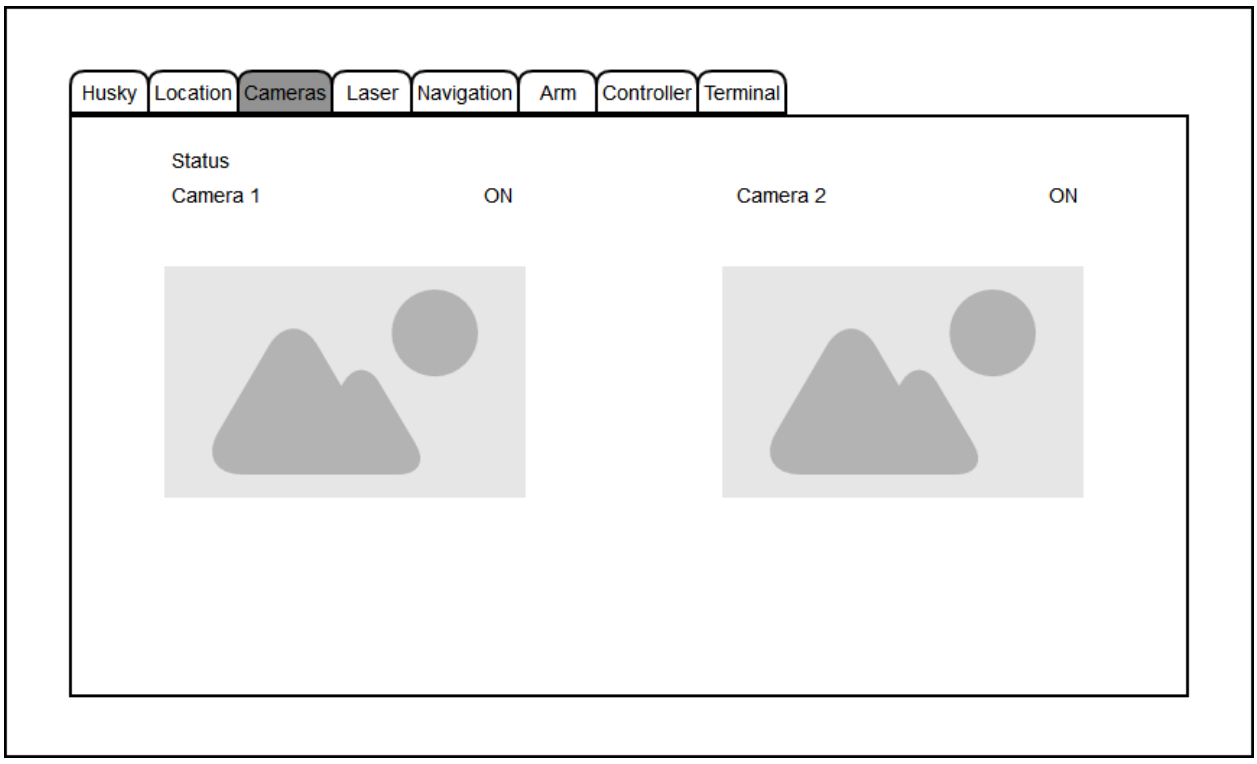


Figura 23 - Maquete do Modo Avançado - Câmaras

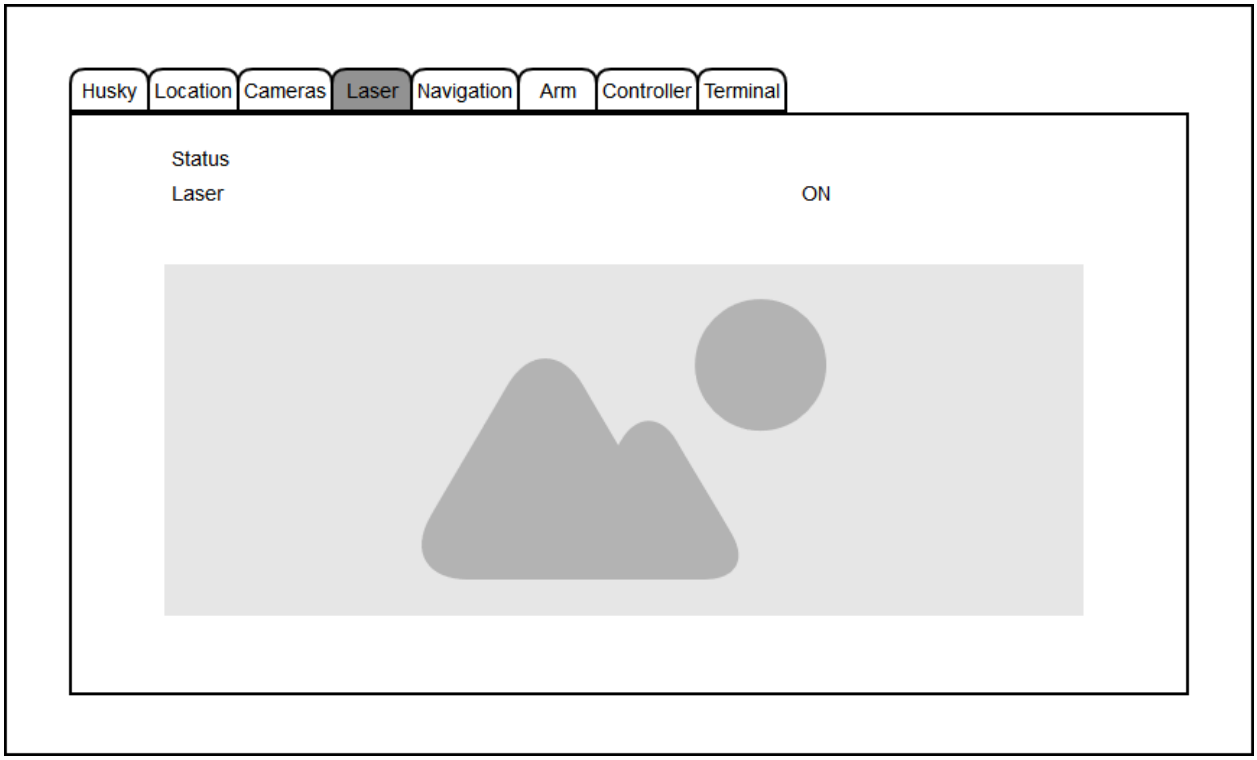


Figura 24 - Maquete do Modo Avançado - Laser

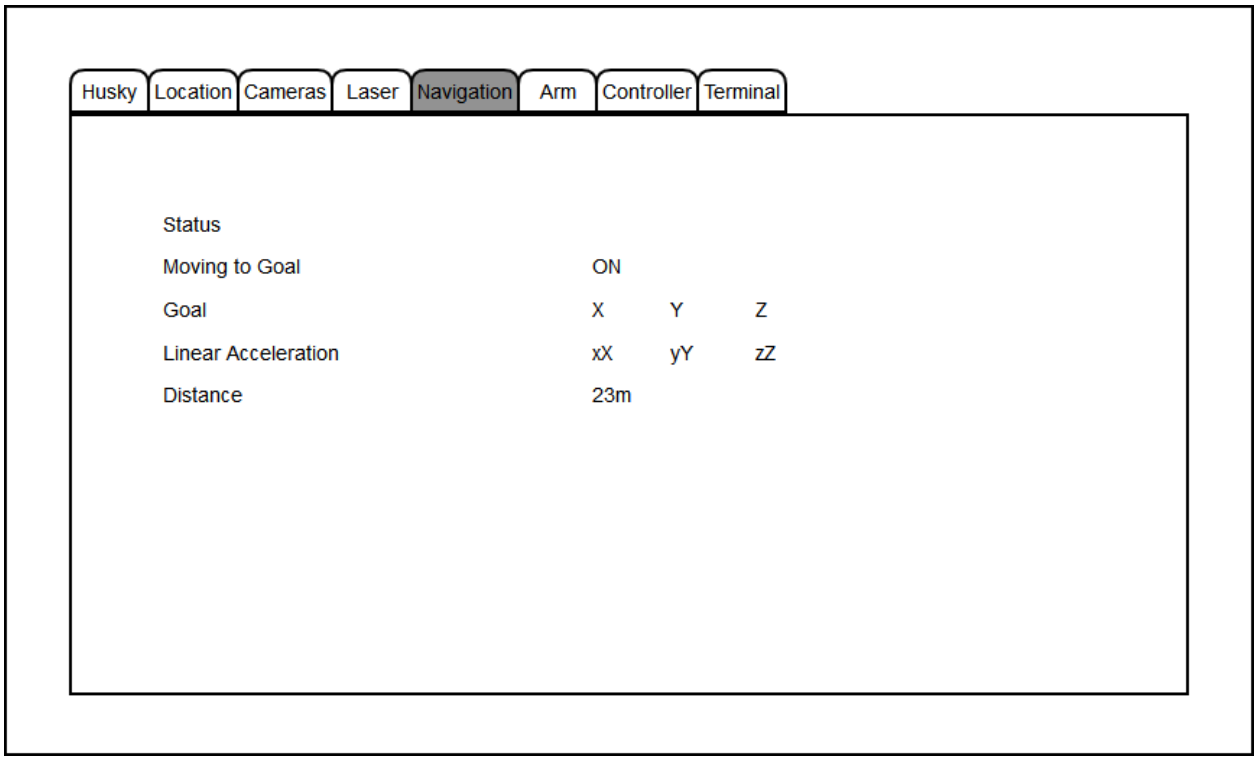


Figura 25 - Maquete do Modo Avançado - Navegação

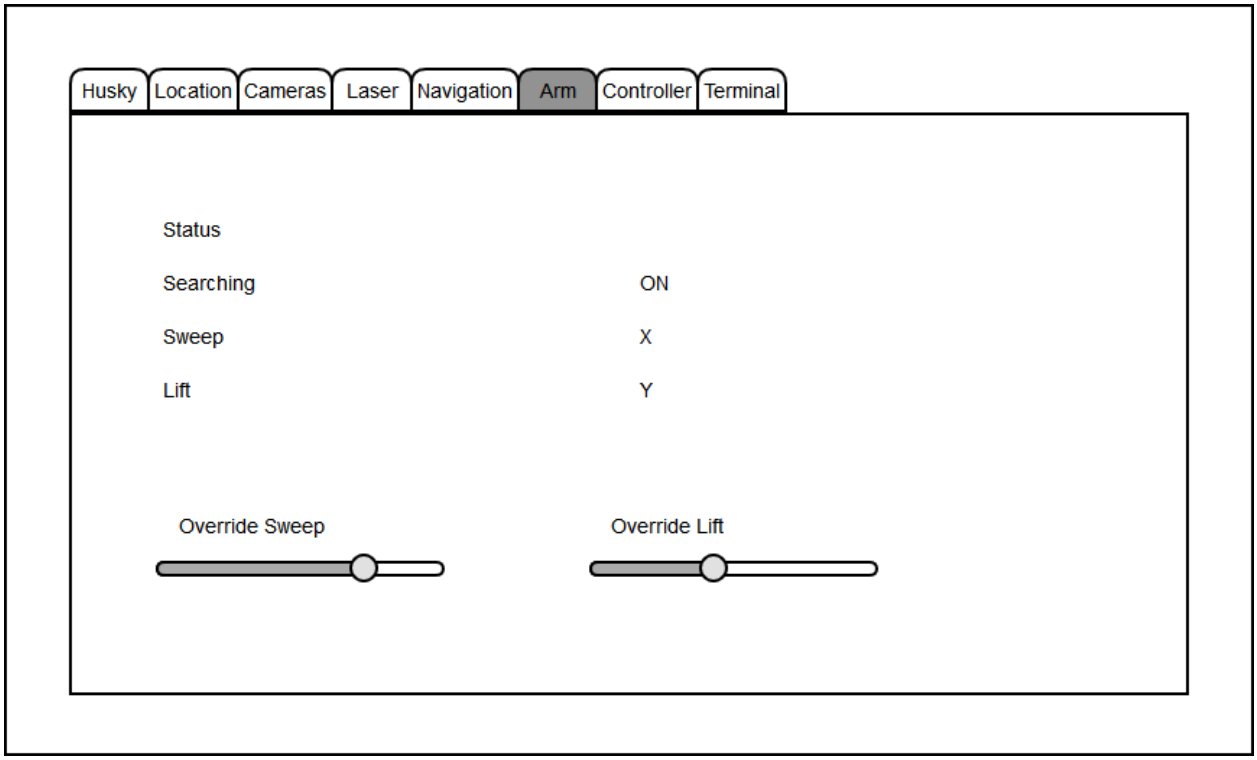


Figura 26 - Maquete do Modo Avançado - Braço

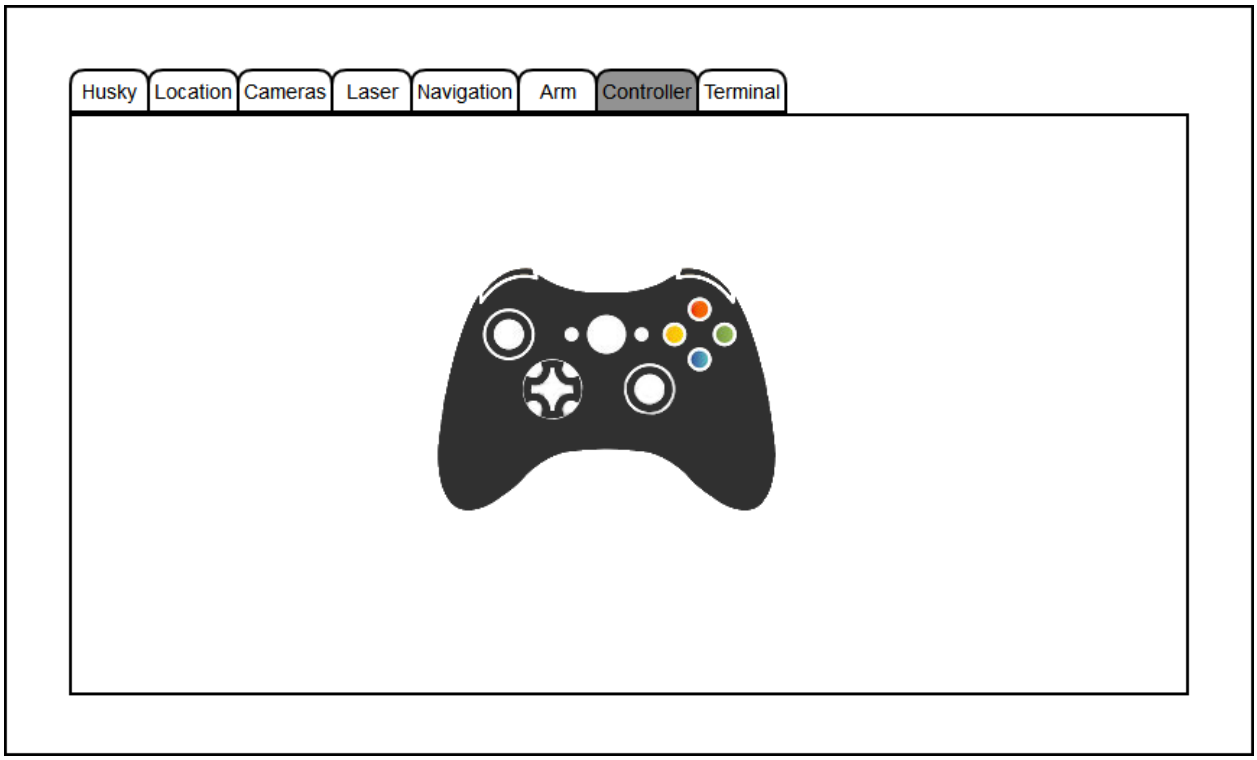


Figura 27 - Maquete do Modo Avançado - Comando

Figura 28 - Maquete do Modo Avançado - Terminal

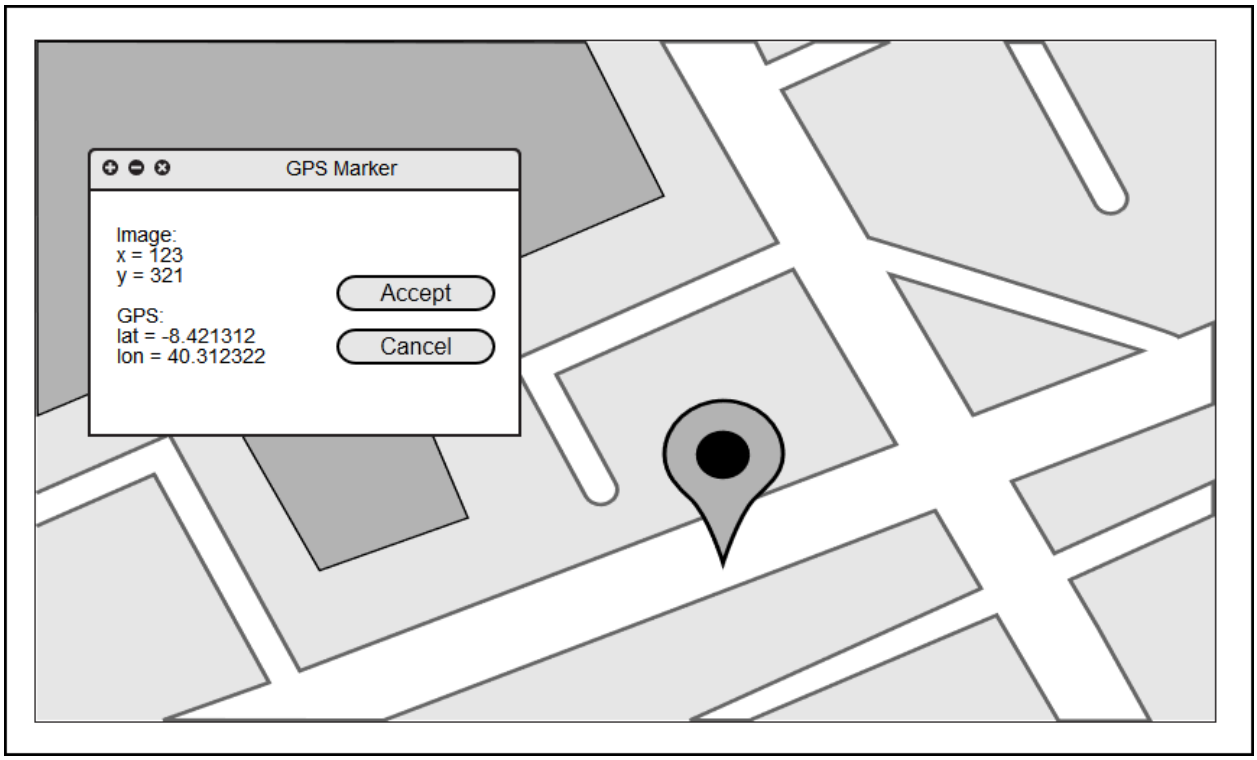


Figura 29 - Maquete da Janela de Georreferenciação

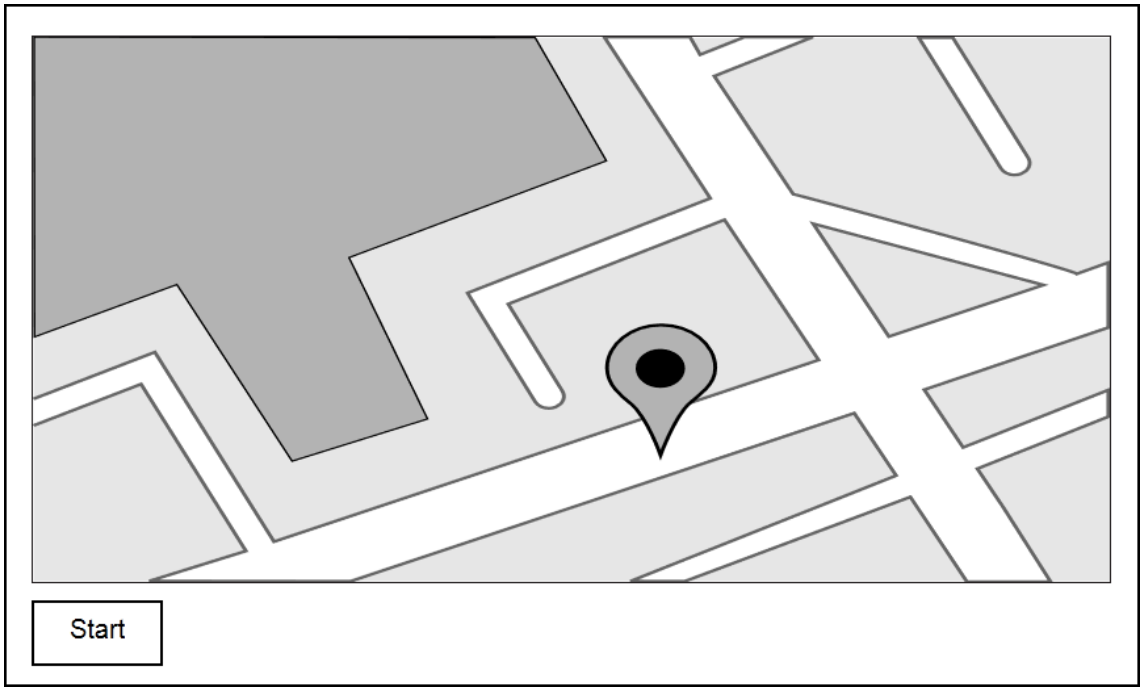


Figura 30 - Maquete da Janela de Procura

Google Maps HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="initial-scale=1.0, user-
scalable=no">
    <meta charset="utf-8">
    <title>Circles</title>
    <style>
      html, body, #map-canvas {
        height: 100%;
        margin: 0px;
        padding: 0px
      }
    </style>
    <script
src="https://maps.googleapis.com/maps/api/js?v=3.exp&signed_in=true"><
/script>
    <script>

var cityCircle;
var map;

function initialize() {
  // Create the map.
  var mapOptions = {
    zoom: 4,
    center: new google.maps.LatLng(40.185128, -8.415847),
    mapTypeId: google.maps.MapTypeId.SATELLITE
  };

  map = new google.maps.Map(document.getElementById('map-canvas'),
      mapOptions);

}

google.maps.event.addDomListener(window, 'load', initialize);

function draw(lat, lng, rad, colour)
{
    var Options = {
      strokeColor: '#FF0000',
      strokeOpacity: 0.8,
      strokeWeight: 2,
      fillColor: '#FF0000',
      fillOpacity: 0.35,
      map: map,
      center: new google.maps.LatLng(lat,lng),
      radius: rad
      };

    cityCircle = new google.maps.Circle(Options);
}

    </script>
  </head>
  <body>
    <div id="map-canvas"></div>
  </body>
</html>
```

Imagens do Funcionamento da Plataforma

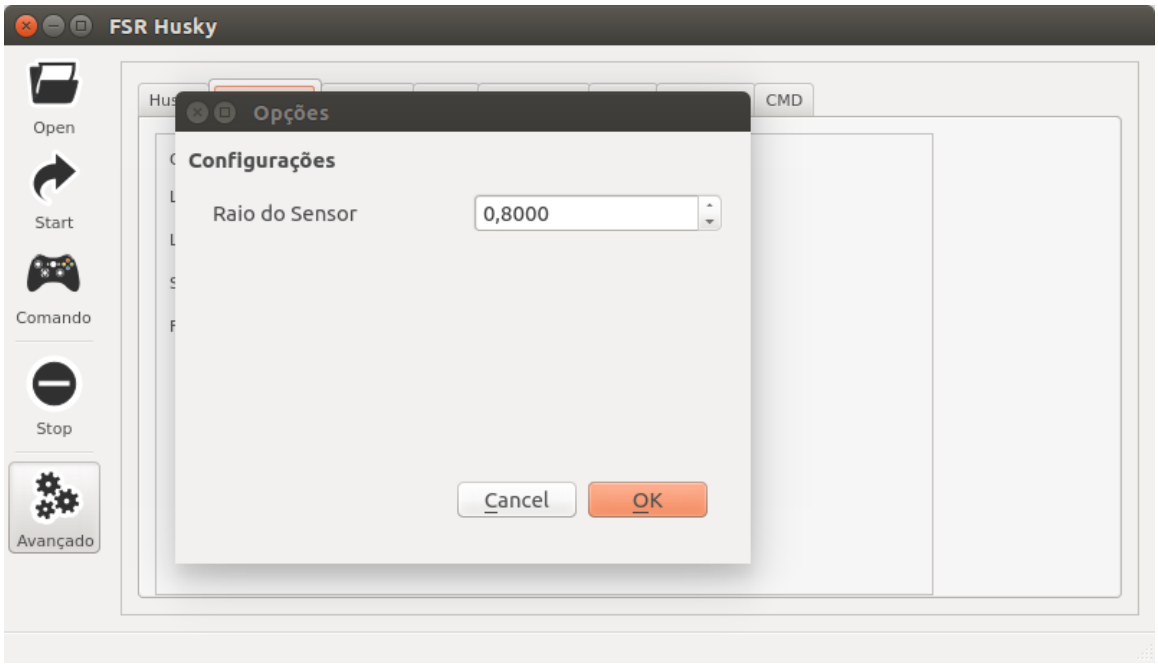


Figura 31 - Janela de Configuração

Figura 32 - Janela de informação do programa

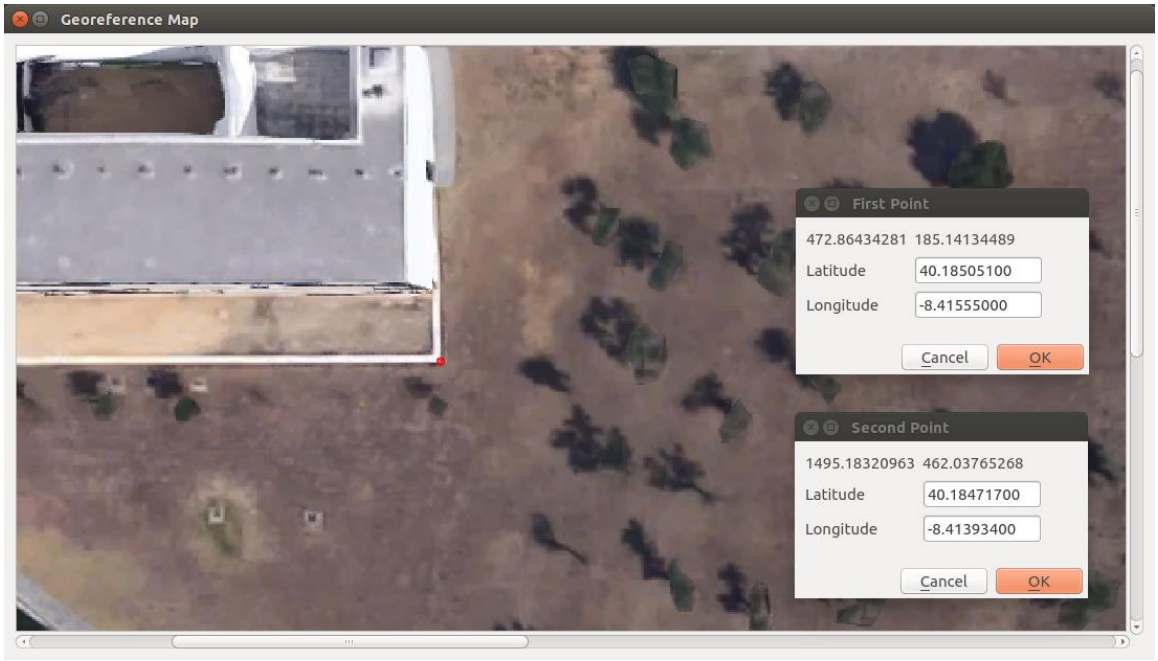


Figura 33 - Georreferenciação - Configuração dos pontos

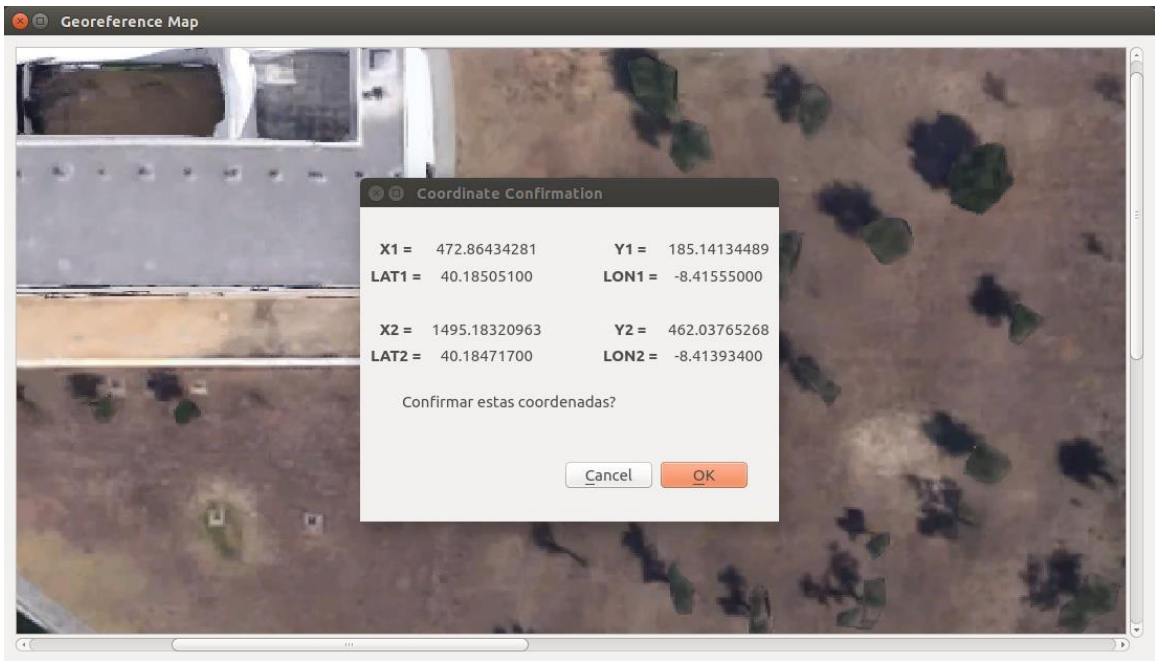


Figura 34 - Georreferenciação - Confirmação

Figura 35 - Janela de Missão - Polígono de procura

Figura 36 - Janela de Missão - Área de procura e Husky

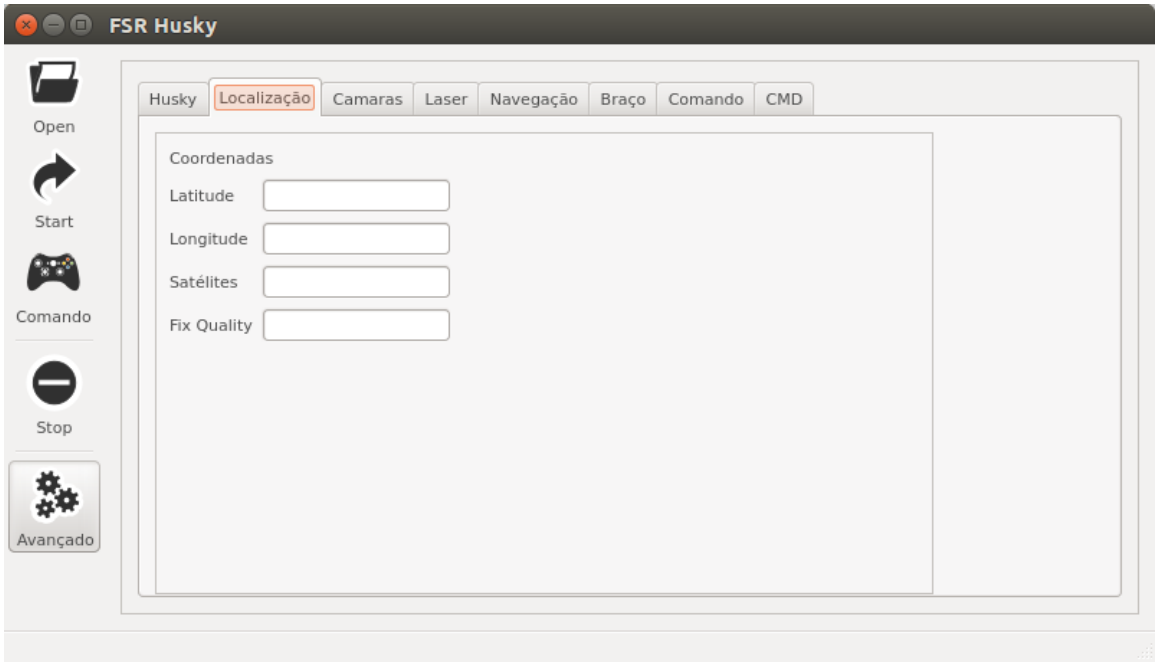


Figura 37 - Modo Avançado - Localização

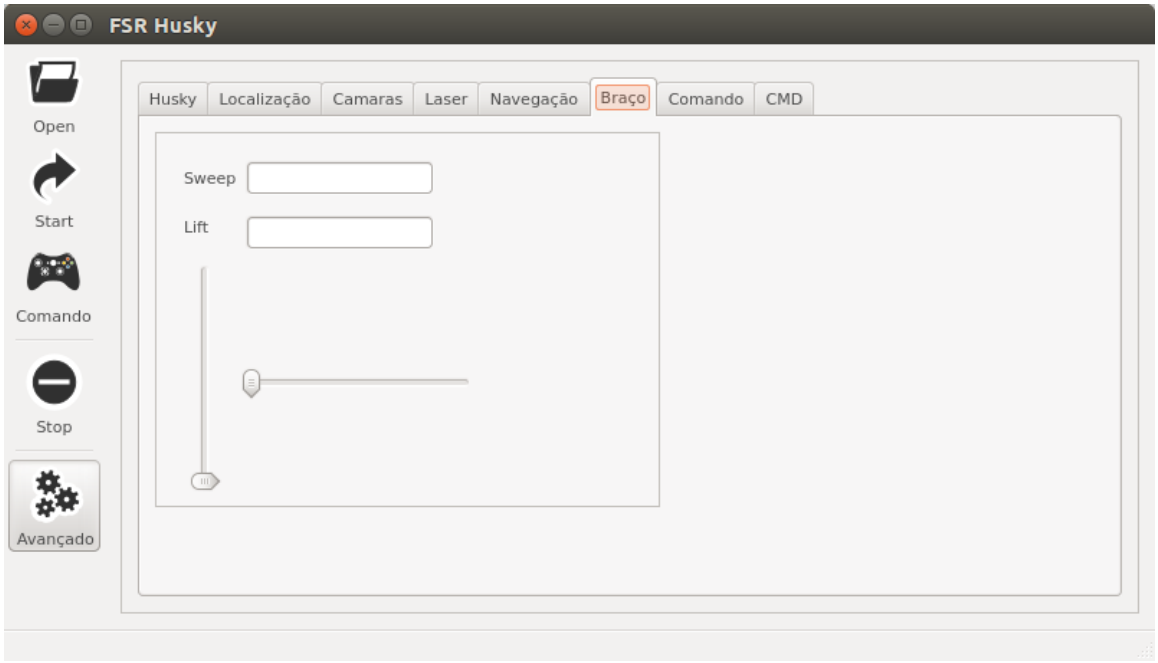


Figura 38 - Modo Avançado - Braço

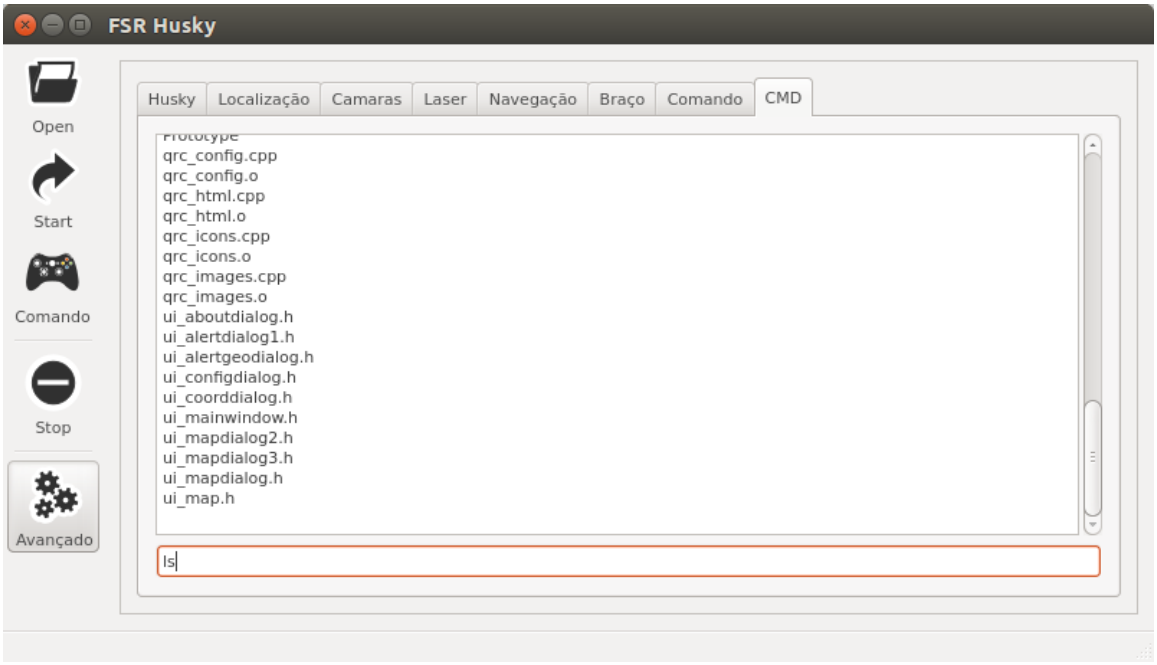


Figura 39 - Modo Avançado - Terminal

Programa Final

https://meocloud.pt/link/721bbdfc-75f1-4be1-8715-e186dc045252/Prototipo%20final.zip/