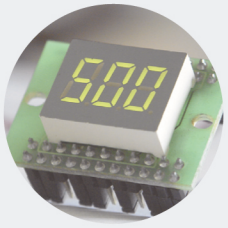
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES






Controlador para missões de exploração robóticas autónomas á base de Behaviour Trees

MIGUEL FONSECA GONÇALVES
Novembro de 2023

POLITÉCNICO
DO PORTO

1.5em

Esta página foi intencionalmente deixada em branco.

Controlador para missões de exploração robóticas autónomas á base de Behaviour Trees

Miguel Fonseca Gonçalves
Nº 1150449

Mestrado em Engenharia Eletrotécnica e de Computadores
Área de Especialização de Sistemas Autónomos
Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

2023

Dissertação, para satisfação parcial dos requisitos do Mestrado em Engenharia Eletrotécnica e de Computadores

Candidato: Miguel Fonseca Gonçalves

Nº 1150449

Orientador: Alfredo Martins

Mestrado em Engenharia Eletrotécnica e de Computadores

Área de Especialização de Sistemas Autónomos

Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

25 de outubro de 2023

Abstract

Technological evolution has increased the demand for raw materials that have never been used until now. There are more than 30,000 abandoned and flooded mines all over Europe where they are believed to contain all these materials that previously had no economic value. Unfortunately, exploiting these mines is expensive in terms of excavation and the removal of existing millions liters of water, leading to uncertainty about the economic viability of exploring and extracting this materials. This problem leads companies to look for new solutions.

Project Underwater Explorer for Flooded Mines (UNEXMIN) intends to develope an Autonomous underwater vechicle (AUV), UX-1 for the autonomous exploration of these mines, in order to enhance its economic value through the sensory information obtained by the vehicle. This dissertation as an objective to develop autonomous exploration maneuvers capable of exploring individual sections of mines such as tunnels, shafts and galleries.

A mission control system was developed using Behaviour trees (BT) allowing the implementation of a more complex level of missions. All the work will be developed with Robot Operating System (ROS) through the Gazebo simulation platform, using realistic models of the UX-1 and the Kaatiala mine, in Finland.

Keywords: AUV, Autonomous exploration, Behaviour tree, Gazebo, Obstacle avoidance, ROS, Sensor fusion, Simulation

Esta página foi intencionalmente deixada em branco.

Resumo

A evolução tecnológica, fez aumentar a procura de matérias primas até agora nunca utilizadas. Existem por toda a Europa, mais de 30.000 minas abandonadas e inundadas onde se pensa conter todos esses materiais que antigamente não tinham valor económico. Infelizmente, explorar essas minas têm um grande custo em termos de escavação e remoção da água existente, levando a uma incerteza sobre a viabilidade económica da exploração e extração das matérias primas.

Esta dissertação pretende contribuir com utilização do UX-1, veículo desenvolvido pelo projeto *Underwater Explorer for Flooded Mines* (UNEXMIN), para exploração autónoma dessas minas, de forma a potenciar o seu valor económico através das informações sensoriais obtidas pelo veículo. O objetivo é desenvolver manobras de exploração autónomas capazes de explorar secções individuais das minas como túneis, poços e galerias.

Foi desenvolvido um sistema de controlo de missões utilizando *Behaviour trees* (BT) permitindo a implementação de missões complexas.Todo o trabalho desenvolvido foi feito em *Robot Operating System* (ROS) através da plataforma de simulação Gazebo, utilizando modelos realísticos do UX-1 e da mina Kaatiala, na Finlândia.

Palavras-Chave: AUV, Behaviour tree, Exploração autónoma, Fusão sensorial, Gazebo, Obstacle Avoidance, ROS, Simulação

Esta página foi intencionalmente deixada em branco.

Conteúdo

Esta página foi intencionalmente deixada em branco.

Lista de Figuras

Lista de Tabelas

Esta página foi intencionalmente deixada em branco.

Lista de Siglas e Acrónimos

AI Artificial Intelligence

AUV Autonomous Underwater Vehicle

BT Behaviour Tree

CPU Central Processing Unit

DEPTHX Deep Phreatic Thermal Explorer

DVL Doppler Velocity Logs

FSM Finite State Machine

GE Game Engine

GUI Graphical User Interface

IMU Inertial Measurement Unit

INESC TEC Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência

LiDAR Light Detection and Ranging

LSA Laboratório de Sistemas Autónomos

MSIS Mechanical Scanning Imaging Sonar

NPC Non Player Character

OSRF Open Source Robotics Foundation

PID Proportional Integral Derivative controller

SLS Structured Light Sensor

UE5 Unreal Engine 5

UNEXMIN Underwater Explorer for Flooded Mines

ROS Robot Operating System

SLAM Simultaneous localization and mapping

Agradecimentos

Na realização da presente dissertação, contei com o apoio direto ou indireto de múltiplas pessoas e instituições às quais estou profundamente grato. Correndo o risco de injustamente não mencionar algum dos contributos quero deixar expresso os meus agradecimentos:

Gostava de agradecer ao Engenheiro Alfredo Martins, pela oportunidade de participar neste projeto e por toda a orientação para projeto que foi um grande desafio, mas que me deu em troca novos conhecimentos, capacidades de trabalho e de envolvência no mundo da robótica.

A todos os amigos e colegas que de uma forma direta ou indireta, contribuíram e me auxiliaram e me motivaram a terminar a dissertação perguntando constantemente "É este mês que vais entregar"ao longo destes dois anos. Em especial, um agradecimento aos meus amigos de mestrado Ana Cláudia, Jónatas Gomes e Tiago Xastre, que ao longo destes 3 anos formamos uma boa equipa onde passamos juntos por dificuldades e situações engraçadas.

Aos restantes investigadores e professores do LSA, queria agradecer pela orientação recebida e pela possibilidade de trabalhar e desenvolver um projeto nas melhores condições possíveis.

Não poderia deixar de agradecer à minha família por todo o apoio económico, pela força e motivação que sempre me prestaram ao longo de toda a minha vida académica, a qual sem o seu apoio teria sido impossível.

"I'm an Engineer; because I invent, transform, create, and destroy for a living, and when I don't like something about the world, I change it."

Esta página foi intencionalmente deixada em branco.

Capítulo 1

Introdução

Nos dias de hoje, a procura por matérias-primas tem vindo aumentar. A grande evolução da tecnologia do séc. XXI tem impulsionado a indústria á utilização de uma grande variedade de materiais. Aparelhos de uso diário como o *smartphone* utiliza mais de 60 tipos de metais, conduzindo as grandes empresas apostar na procura de novas fontes desses materiais preciosos. A exploração mineira, por parte dos humanos já acontece há milhares de anos, mas no passado apenas metais como o ouro, prata e cobre é que eram valorizados. Atualmente, só na Europa, existem cerca de 30.000 minas fechadas, algumas delas encontrando-se fechadas há mais de 100 anos [1]. Ao longo do tempo, essas minas fecharam portas por diversas razões, escassez de matéria prima, altas concentrações de gases, radiação, derrocadas e inundações, etc. A necessidade de recursos por parte das empresas, têm proporcionado o estudo e a reabertura de muitas dessas minas abandonadas, pois muitas das matérias primas, hoje em uso na indústria, antigamente não tinham qualquer valor económico. Uma questão surge, "será que compensa a exploração dessas minas?". A inatividade das minas, faz com não seja feita a devida manutenção, levando a que muitas delas estejam inundadas, o que torna um processo lento e dispendioso. Todo o processo de reabrir a mina, retirar milhões de litros de água para no final não existir lucro na exploração, faz com que as empresas apostem em métodos alternativos de determinar o valor económico da exploração. O projeto *Underwater Explorer for Flooded Mines* (UNEXMIN), é um projeto Europeu que tem como exploração minas abandonadas inundadas. A missão tem como objetivo a utilização do veículo (UX-1), desenvolvido pelo INESC TEC no LSA-ISEP, figura 1.1, capaz de ser teleoperado, para fazer um mapeamento 3D dos túneis, e galerias dessas minas, visão em tempo real, através de câmeras embutidas no veículo e de forma a ser possível identificar possíveis matérias-primas que tornem viável a exploração da mina.

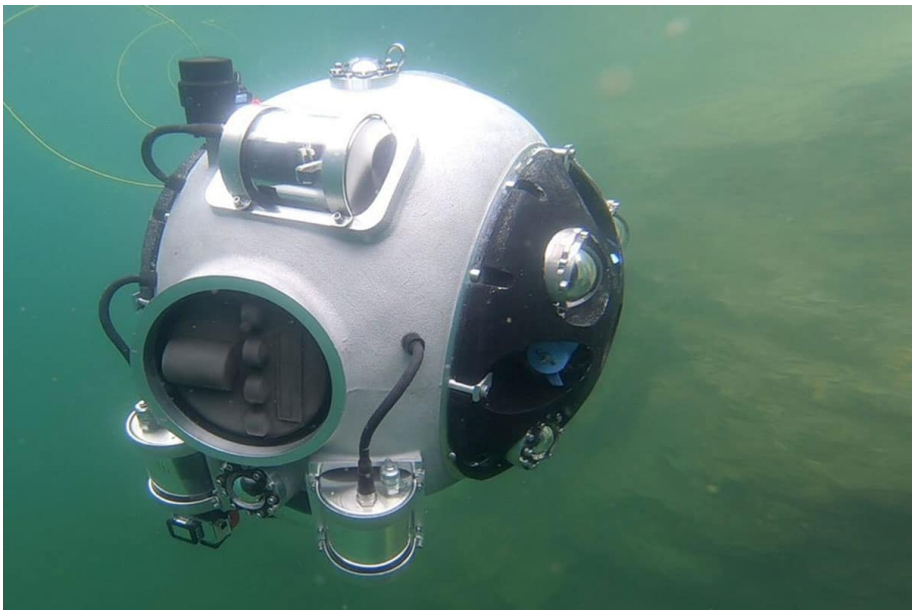


Figura 1.1: Veículo de exploração UX-1.

Atualmente o controlo da missão de exploração acontece por uma equipa completa de investigadores, onde cada membro tem uma tarefa ativa no seu desenvolvimento. Isso torna a missão, muito complicado a nível de logística, pois á equipa da empresa composta por profissionais na área da mineração, junta-se o acréscimo de levar uma equipa experiente no controlo do robô de exploração. O ideal para estes estudos será a companhia poder alugar ou comprar o veículo autónomo, e este ser capaz de explorar as minas autonomamente, com programas e algoritmos pré-definidos [2].

logisticamente é muito dispendioso para quem contrata o robô de exploração, pois normalmente as empresas que necessitam deste tipo de missões são compostas por profissionais na área da mineração. Tendo este problema em mente, pretende-se dividir o trabalho em três partes. A primeira etapa envolve a identificação das possíveis situações ou cenários que o veículo autônomo pode encontrar durante a sua missão. Esta análise ajuda a compreender exatamente o que o veículo pode enfrentar no mundo real. Tornando o processo de desenvolvimento mais prático e focado. Apesar do estado das minas ser desconhecido, sabemos que deveremos encontrar secções como túneis, poços e galerias, pois em tempos, essas mesmas eram frequentadas por mineiros que utilizavam constantemente os mesmo padrões. Após a caracterização do problema, a segunda parte do trabalho passa por desenvolver manobras individuais capazes de resolver o problema em questão, como por exemplo mapear a mina, percorrer túneis, descer poços, entrar

em cavidades, etc. Todas estas manobras serão desenvolvidas e testadas no ambiente de simulação Gazebo. A última fase terá como objetivo estudar um método capaz de implementar todas as manobras desenvolvidas. O foco do estudo será nas *Behaviour Trees* (BT) como organizador de missões para o UX-1.

1.1 Motivação

A exploração do nosso planeta, tem levado a humanidade a quebrar os seus limites em busca de novas ideias e conhecimentos. Conseguimos estender-nos por todos os todos continentes e ilhas do globo, conseguimos conquistar o céu e mais recentemente o espaço. Toda essa expansão exige uma grande quantidade de recursos, que no caso de algumas matérias primas já se encontram em escassez. Isto tudo leva a que se explore os nossos oceanos e tudo que se encontra neles, visto que grande parte do nosso planeta é composto por mares e oceanos [3].

O ser humano por si próprio não consegue explorar estes cenários devido aos seus limites físicos, mas com ao auxilio de veículos de exploração autónoma, somos capazes de atingir os nossos objetivos. Esta dissertação, surge com a necessidade de criar algoritmos simples, capazes de serem adaptados a qualquer tipo de missão, para que os veículos sejam capazes de explorar autonomamente, os ambientes subaquáticos, independentemente do tipo de experiência que se tem na área da robótica.

1.2 Objetivos

Esta dissertação pretende alcançar dois objetivos principais. Primeiro, a construção de manobras de locomoção de forma a dar resposta a qualquer tipo de situação que se pode encontrar nas minas sub-aquáticas. Segundo, o estudo da viabilidade de utilizar *behaviour trees* como forma de controlo de missão.

Desta forma pretende-se cumprir os seguintes objetivos, descritos na lista abaixo:

- Estudo dos diferentes tipos de cavidades e cenários numa mina.
- Criação de manobras autónomos para o UX-1, para cada tipo de cenário.
- Estudo sobre *behaviour trees*.
- Criação de uma missão usando as *behaviour trees* como controlo de missão.
- Validação dos métodos desenvolvidos e avaliação de desempenho em comparação com outros métodos.

1.3 Estrutura

Esta dissertação encontra-se organizada em oito capítulos. No segundo capítulo encontra-se a Formulação do Problema, com descrição do dos objetivos para este projeto de exploração autónoma, bem como os desafios e dificuldades a ser abordados.

O estado da Arte exposto no capítulo 3, apresenta um estudo sobre as soluções existentes de outros robôs para exploração autónoma, algoritmos de *path planning* e claro uma descrição FSM e *Behaviour trees*. Terminando com uma conclusão sobre o caminho a seguir

Os Fundamentos teóricos, no capítulo 4, dão suporte a todo o trabalho a ser desenvolvido e implementado, dando bases e conhecimentos.

O Ambiente de desenvolvimento, no capítulo 5. Dá a conhecer as ferramentas a serem utilizadas no projeto. Desde a simulação, aos programas necessários para implementar.

O Projeto é descrito no Capítulo 6, neste consta toda a estrutura e arquitetura do sistema desenvolvido, aplicações utilizadas e metodologia adotada.

O capítulo 7 apresenta os desenvolvimentos realizados, dividido entre as manobras de exploração criadas, seguindo depois para a utilização das BT como controladores.

Os resultados estão presentes no capítulo 8, o comportamento veículo em resposta aos cenários apresentados, bem como a viablidade e os resultados da utilização das *behaviours trees*.

Por fim a conclusão no capítulo 9, tece comentários a todo o trabalho realizado e resume os desenvolvimentos futuros.

Capítulo 2

Formulação do problema

Nesta secção será abordado as condições do ambiente de simulação em conjunto com os problemas a serem resolvidos.

2.1 Ambiente de Simulação

Este trabalho tem como base o ambiente de simulação Gazebo criado pelos investigadores do laboratório. Onde se encontra disponível um modelo do UX-1, e para simular as condições encontradas pelo robô, foram utilizadas réplicas do tanque do Laboratório, e da mina Kaatiala, Finlândia de onde já foi um teste real e através do mapa efetuado foi feito um modelo em Gazebo para testes.

2.2 Problema 1

A exploração autónoma de uma mina sub-aquática, apresenta dois desafios importantes para o sucesso da missão. O primeiro é o facto de normalmente serem ambientes desconhecidos. A falta de informação sobre as dimensões, quantidade de cavidades e o estado dos acessos da mina, faz com que seja difícil saber e preparar o veículo para a missão. Em relação a mina podemos não saber as suas dimensões e condições, mas sabendo que foi utilizada por humanos leva a que seja mais fácil identificar os cenários de exploração. Uma mina, quando explorada pode ser reduzida a três tipos de cavidades, túneis com diferentes inclinações, poços para subir ou descer em altura e galerias, grandes espaços de onde foram extraídos anteriormente os recursos. Como tal, será necessário desenvolver algoritmos independentes para cada situação.

2.3 Problema 2

O segundo desafio aparece de como dar a conhecer ao robô em que tipo de estrutura ele se encontra, e qual o melhor algoritmo para explorar aquela secção. Pois, algoritmos apresentam limitações, devido a falta de existência de um mapa completo da estrutura ou elementos de orientação capazes de guiar o robô. O estudo de algoritmos relacionados com a inteligência do robô será fundamental para resolução do problema.

Capítulo 3

Estado da Arte

Nesta secção são apresentados os diversos trabalhos realizados no âmbito da exploração em ambientes desconhecidos. Os robôs autónomos permitem a realização de tarefas em ambientes condicionados sem a assistência humana. A exploração de ambientes desconhecidos conduz a inúmeras questões e cuidados a ter em consideração, dado que o robô não tem qualquer tipo de informação acerca do ambiente a explorar. Atualmente existem inúmeros veículos subaquáticos dedicados à inspeção de docas, petrolíferas e gasodutos ou mesmo estruturas naturais como cavernas e fissuras. Algo que todos têm em comum é a necessidade de serem operados manualmente por um operador, restando apenas uma pequena percentagem que opera autonomamente. Visto o projeto ter como objetivo a exploração autónoma, os veículos a serem apresentados na secção abaixo permitem ter uma noção do tipo de problemas existentes na exploração subaquática, uma análise de como resolvem os problemas manualmente ou automaticamente e possíveis melhoramentos a serem feitos.

3.1 Veículos Submarinos Autónomos (AUV - Autonomous Underwater Vehicle

3.1.1 DEPTHX

O *Deep Phreatic Thermal Explorer* (DEPTHX) [4], representado na Figura 3.1, é um veículo subaquático autónomo projetado e construído pela *Stone Aerospace*. O DEPTHX tem dois objetivos principais, o primeiro é ser um veículo totalmente autónomo sendo

capaz de navegar num território desconhecido, recorrendo às suas capacidades de mapeamento 3D. O segundo objetivo deste robô é identificar zonas com potencial existência de via microbiana.

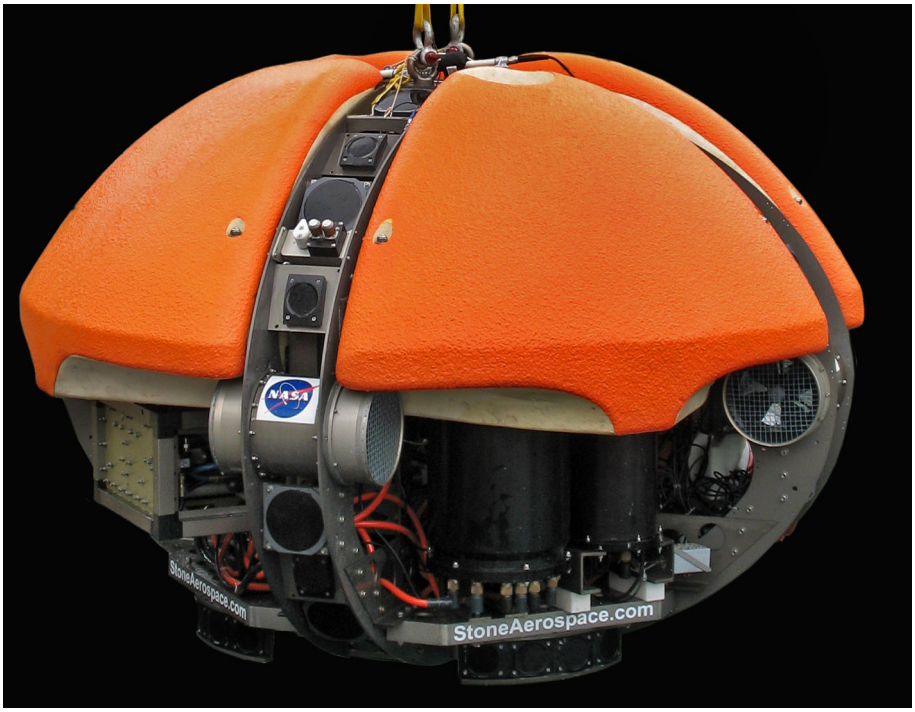


Figura 3.1: Deep Phreatic Thermal Explorer (DEPTHX). [4]

O DEPTHX, figura 3.2, apresenta 54 sonares espaçados em torno da sua estrutura. O sonar utiliza pulsos de ondas sonoras de alta energia permitindo localizar objetos a um raio de 250 a 300 metros do veículo. As informações adquiridas são retransmitidas para os computadores de bordo para o controlo de navegação. Para além disso, este robô também possui acelerómetros, medidores de profundidade e uma unidade de orientação inercial, assim como um *Doppler Velocity Logs* (DVL) que determina a velocidade com que o veículo se está a mover em relação ao fundo. De seguida, envia essas informações para o computador principal[5]. À medida que o DEPTHX se movimenta, os computadores usam as informações provenientes dos diversos sensores, anteriormente referidos, criando imagens em 3D que são sobrepostas na memória do computador para criarem um mapa geométrico progressivo. O módulo de *Pose Estimation*, calcula a posição do veículo dependendo do estado do robô, condições do ambiente e dos dados obtidos. A posição pode ser obtida por três métodos diferentes. Utilizando *dead reckoning* quando apenas se obtém leituras do Inertial Measurement Unit (IMU) e do sensor de profundidade.

O segundo método, utiliza as leituras dos sonares, em conjunto com modelos 3D do ambiente previamente construídos. Por último, fazendo uso de uma técnica chamada de *Simultaneous Localization And Mapping* (SLAM), além da posição, produz um mapa [6].

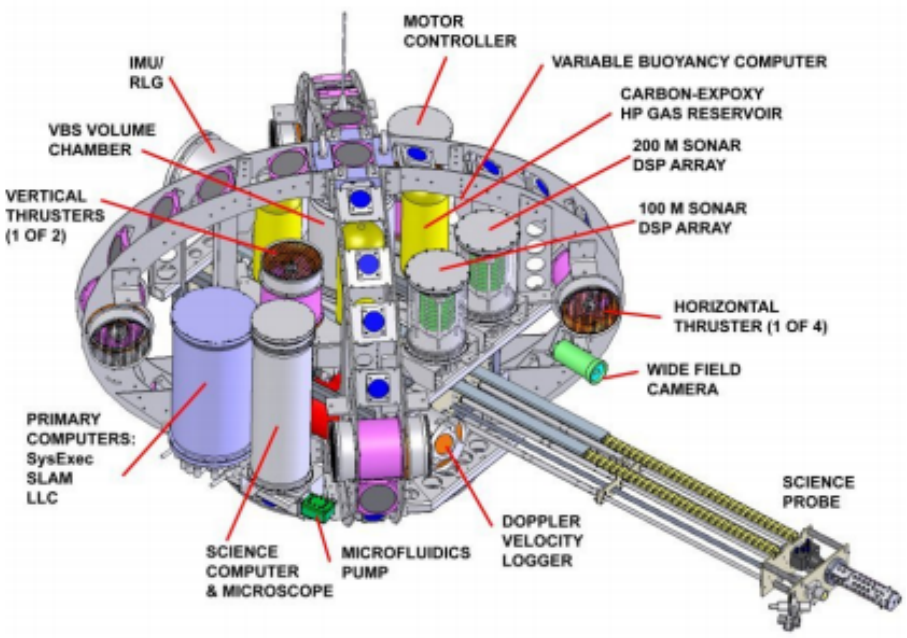


Figura 3.2: Instrumentos científicos e de navegação do DEPTHX.[4]

Este veículo foi feito para determinar a existência de vida microbiana, na Lua Europa. Dado as cavidades desta Lua terem grandes dimensões, este veículo apenas faz uso de uma manobra de exploração que permite ao veículo acompanhar uma parede, este método é o *Near-Wall*, figura 3.3. O robô aproxima-se da parede mais próxima a uma distância de afastamento de d_0 e uma direção relativa de *theta*, mantendo a profundidade constante.

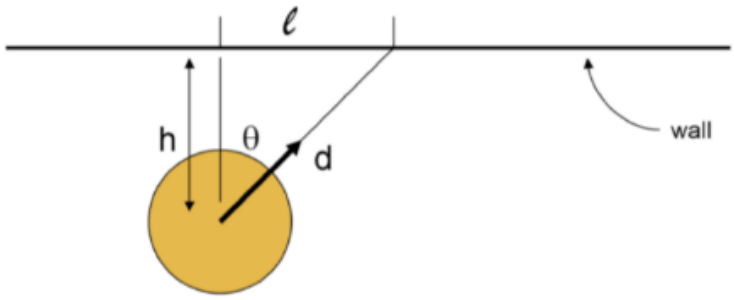


Figura 3.3: Método de navegação recorrendo a uma parede.[4]

Em síntese, o método de exploração adotado pelo DEPHTX é uma das técnicas que se pode ter em consideração a executar no robô UX-1. O uso de uma parede como ponto de referência em ambos os robôs será útil como forma de deslocação, mas também como fonte de informação sobre os elementos presentes no meio ambiente.

3.1.2 SPARUS

A universidade de Girona, desenvolveu um Autonomous Underwater Vehicle (AUV) denominado SPARUS [7], Figura 3.4, com vista à exploração subaquática. Este veículo foi planeado para aplicações industriais, científicas e académicas [8]. E, tem a vantagem de ser leve, fácil de operar e manobrar, desempenhar várias tarefas e ser programado em ROS, o que permite a incorporação de nós já desenvolvidos e ao mesmo tempo a possibilidade a reutilização e partilha de informação.

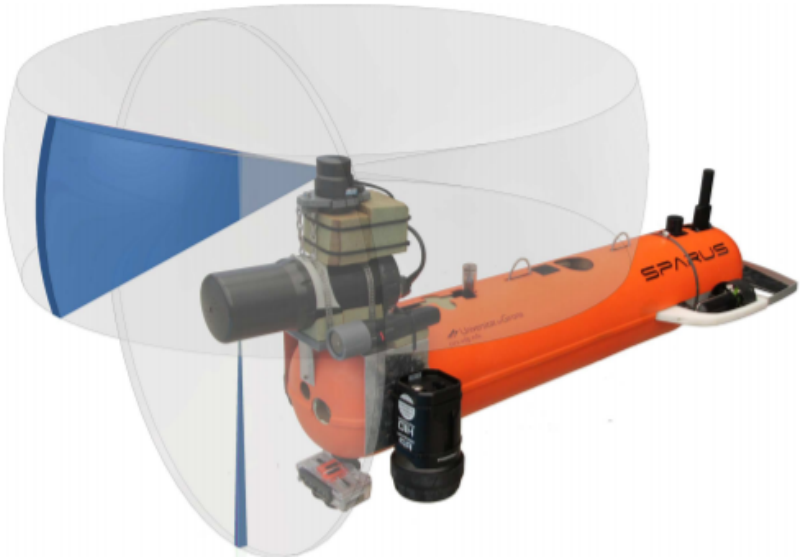


Figura 3.4: SPARUS.[7]

O SPARUS a faz uso de SLAM como forma de orientação e deslocamento. Apesar de os cenários de atuação serem diferentes devido ao SPARUS não conseguir operar em espaços confinados, os problemas apresentados são semelhantes ao do UX-1. Ambos os veículos, desconhecem o o cenário de atuação, tem o objetivo de mapear o ambiente envolvente e partilham a necessidade de se locomover apenas através do sensores existentes abordo. Alguns dos sensores em comum entre os dois, é por exemplo o *Mechanical Scanning Imaging Sonar* (MSIS). No caso do SPARUS, foram adicionados dois, um na

horizontal e outro na vertical. Estes são utilizados como forma de deteção de obstáculos. Os dados obtidos pelos sensores providenciam um plano 2D do meio ambiente onde se pode detetar os obstáculos através das intensidades dos pontos recolhidos. Com esta implementação obtemos um exemplo importante para analisar, tirar proveito e implementar utilizando o MSIS existente no UX-1, no desenvolvimento de manobras de exploração autónomas.

3.1.3 SUNFISH

O SUNFISH (AUV), foi desenvolvido pela *Stone Aerospace*, uma empresa que tem como objetivo de criar uma plataforma robótica avançada, usando todos os recursos possíveis para o efeito. É um AUV compacto, portátil, com capacidade para pairar, controlável em todos os seis graus de liberdade. Ele contém um conjunto básico de sensores para realizar uma variedade de missões: operações de levantamento, inspeção de precisão, intervenção submarina e exploração e mapeamento de geometrias 3D complexas [9]. Ele tem a capacidade de integrar cargas adicionais para tarefas mais específicas, como amostragem de água ou perfil biogeoquímico. O SUNFISH, figura 3.5 pode utilizar uma fibra de comunicação para exibição de dados em tempo real e intervenção em nível de tarefa nas operações do veículo. O objetivo do desenvolvimento do SUNFISH é estender os comportamentos de exploração e a tomada de decisões além das missões com planeadas, até o ponto em que o veículo aceita metas de alto nível e toma suas próprias decisões locais sobre como atingir esses objetivos.

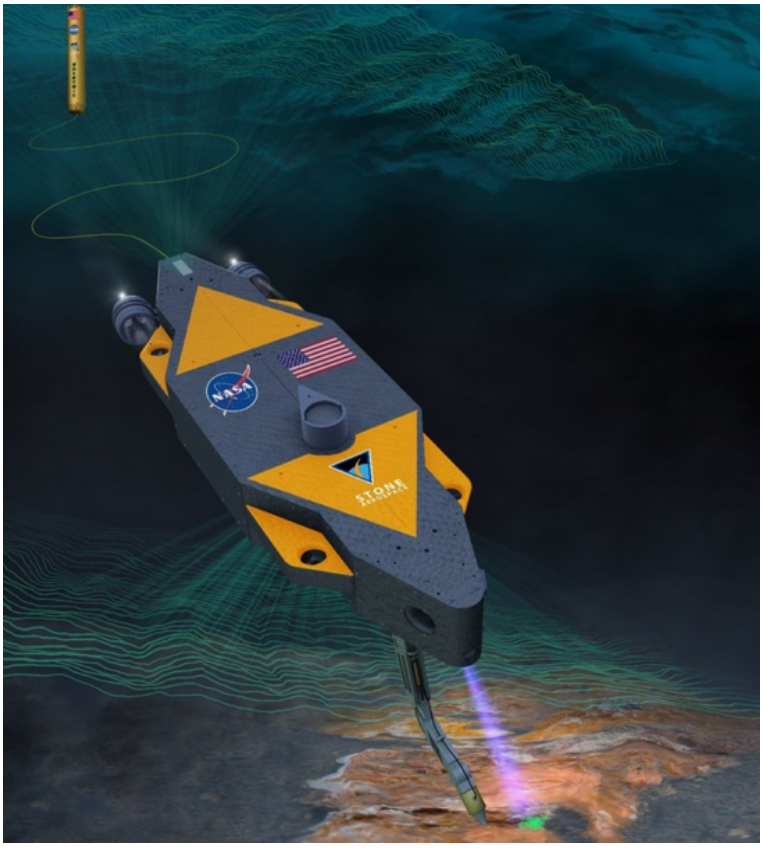


Figura 3.5: SUNFISH.[9]

Fazendo uso das suas capacidades, o SUNFISH tornou-se no primeiro veículo autónomo a mergulhar de verdade em cavernas. Em novembro de 2017, o SUNFISH navegou de forma autónoma pelo ambiente labiríntico do sistema de cavernas *Peacock Springs*, na Flórida, figura 3.6. O SUNFISH demonstrou a capacidade de ser capaz de traçar o seu próprio caminho, sem qualquer intervenção humana, para produzir um mapa 3D detalhado do que explorou. O resultado foi o primeiro mapa de alta resolução deste conhecido sistema hidrológico, figura 3.7.

Figura 3.6: Mina subaquática de Peacock Springs, Flórida. [10]

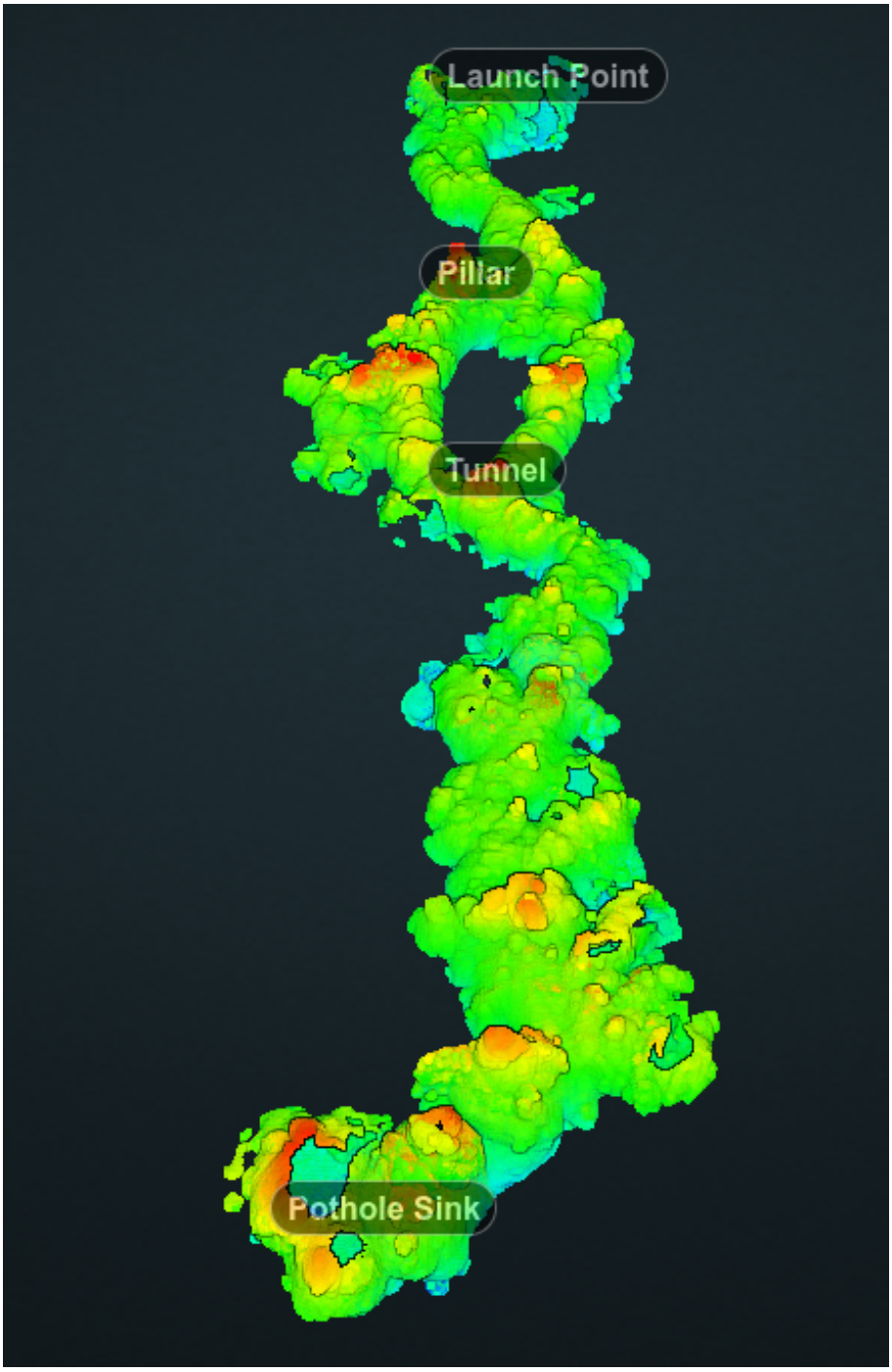


Figura 3.7: Mapa 3D de alta resolução obtido pelo SUNFISH. [10]

O SUNFISH faz uso de um sistema de navegação e mapeamento baseado em SLAM com recurso a um sonar subaquático que usa a localização detetada e geometria local como referência de navegação. É excecionalmente poderoso como uma ferramenta de navegação onde existe topografia discernível (por exemplo, fissuras, vales, recifes, poços, etc.).

Semelhante ao UX-1, o SUNFISH enfrenta o problema de ser difícil mapear e orientar em simultâneo, sendo que a discussão para o método que mais eficiente de SLAM ainda hoje é estudado e debatido por todo o mundo. Por um lado temos o "*Frontend*" focado na filtragem de dados dos sensores, extração de *features* e determinando a localização através das correspondências, e o do outro lado "*Backend*" que tendo em conta todos os dados, determinar a melhor estimativa do mapa completo e da trajetória do veículo através o mapa [9].

3.2 Path Planning

A navegação autónoma é um ativo valioso para robôs. Ajuda a diminuir a sua dependência da intervenção humana. No entanto, também envolve muitas tarefas ou problemas para resolver, por exemplo, *path planning* [2]. Esta tarefa consiste por encontrar o melhor caminho de forma a fazer com que um robô alcance o estado desejado a partir do atual. Por exemplo, ambos os estados podem ser, respetivamente, a posição inicial e a meta. Este curso de ação vem na forma de um caminho, também chamado de rota. O caminho serve para guiar o robô ao estado desejado em questão. No entanto, pode haver inúmeros caminhos possíveis, dado o espaço livre em que o robô pode se mover. Os algoritmos de planeamento de caminho geralmente tentam obter o "melhor caminho"ou pelo menos uma aproximação eficiente dele. O melhor caminho a que se refere, resulta da utilização de um ou mais algoritmos de otimização para atingir o caminho mais curto tendo em conta os objetivos. Por exemplo, esse caminho pode ser aquele que se faz em menor quantidade de tempo ou a que gasta menos combustível. Isso é fundamental em missões como as do campo de busca e salvamento [11]. Outra função de otimização a ser considerada pode ser a gestão de energia do robô. No caso da exploração planetária, esta é uma situação crítica onde os recursos energéticos disponíveis são limitados [12]. Ao mesmo tempo, o caminho gerado pelo controlador deve seguir as restrições impostas. Estes podem vir das limitações na adaptabilidade do robô a determinados terrenos. A locomoção do robô e as características do terreno existente limitam o tipo de manobras que podem ser realizadas. Na literatura há um grande número de abordagens de planeamento de caminho e este número tem continuado a aumentar ao longo dos anos. Por esse

motivo, selecionar a abordagem mais adequada atendendo a determinados requisitos (por exemplo, as restrições de locomoção, condições do ambiente, etc) pode ser uma tarefa desafiadora.

A Figura 3.8 retrata quatro categorias de controladores de planeamento, dividindo cada uma delas em duas subcategorias. Essa classificação se baseia nos princípios e mecanismos fundamentais usados para construir e executar um caminho. Uma visão mais detalhada dessas categorias é fornecida na próxima subsecção.

Figura 3.8: Esquema que exemplifica as diferentes arquiteturas de planeamento, bem como como se relacionam entre elas. [13]

Como observado na figura 3.8, existem múltiplas abordagens para fazer *path planning*, dependendo da área de atuação alguns algoritmos tornam-se mais eficientes para atuar. As quatro categorias *C-Space Search*, *Soft Computing*, *Reactive Computing* e *Optimal Control* apresentam respetivamente duas subcategorias cada. Este esquema permito-nos comparar estas diferentes categorias usando as diferentes escalas. Por exemplo é possível identificar que duas das categorias tendem para a área do planeamento local, *Reactive Computing* e *Soft Computing* bem como *C-Space Search* e *Optimal Control* estão mais

focadas no planeamento global. No entanto também podemos agrupar dois a dois as subcategorias se consideramos o *core* dos algoritmos, agrupando assim por *Parameter Tuning*, *Stochastic Iterations*, *Preliminary Graph* e *Numerical Optimization*. O cenário de atuação irá ditar os melhores algoritmos a utilizar.

3.2.1 Global Planning

Os algoritmos baseados em *Global Planning*, têm por base o conhecimento da totalidade da área de atuação. Como tal eles são normalmente utilizados para idealizar o melhor caminho no tempo mais curto possível, ou o caminho com o menor consumo, pois é conhecido todo o mapa. Um exemplo é utilização de um drone para mapear um edifício pelo exterior. O objetivo para o *path planning* do drone passa por ser capaz de mapear toda a área do edifico, no menor tempo possível. O algoritmo apenas tem de se preocupar de calcular a rota mais eficiente para o efeito. Neste cenário, não existe necessidade de ter algoritmos de deteção de obstáculos, ou de reação a obstáculos, pois o espaço disponível é o ar onde não existe nada disso ou é pouco provável uma situação dessas.

3.2.2 Local Planning

Ao contrário do tipo *path planning* descrito em cima para se utilizar um algoritmo de *Local Planning*, é pressuposto que não se conhece o ambiente envolvente mas conhecem-se a grande maioria dos possíveis acontecimentos. Nesta área algoritmos reativos ou com inteligência artificial prevalecem. Quando utilizados estes tipos de algoritmos, são dotados de respostas pré-programadas para o cenário em questão, mesmo não se conhecendo o meio envolvente eles são capazes de atuar em qualquer cenário pois os acontecimentos prováveis são os mesmos de acontecer.

Por exemplo, num cenário onde um veículo autónomo é posto a conduzir numa cidade. Do ponto de visto conceptual é um cenário simples, pois o veículo apenas tem de seguir as regras de trânsito. No entanto, certos elementos causam imprevistos, como por exemplo as pessoas que circulam pela cidade, podem atravessar a frente do veículo, ou um condutor humano se distrair e ir em direção ao robô. É impossível prever estes acontecimentos, no entanto dotando o veículo autónomo com um algoritmo reativo será mais fácil ele se prever em relação a esses acontecimentos.

3.3 Arquiteturas de controlo

Como explicado no capítulo 2 desta dissertação, um dos problemas encontrados para executar uma missão de mapeamento autónoma, era o controlo da mesma. Por outras palavras, tendo o robô capacidades de resposta aos tipos de cenários encontrados numa mina submersa (túneis, poços e galerias), como é que ele identifica e disse quando e qual das funcionalidades usar. Um dos desafios, em construir um estrutura de controlo para um sistema autónomo, passa por dar capacidades, numa ordem e num momento específico do veículo de reagir aos cenários. No entanto, ao contrário de uma fábrica, onde existe um mapa do ambiente, sinais de orientação como cores e linhas e o robô é feito a medida para o local de trabalho, tornando-se fácil de construir uma estrutura de controlo. Numa mina subaquática, não existe nenhum dos elementos anteriormente descritos. O que leva a que arquitetura de controlo numa missão destas passe por ser um algoritmo reativo ou que beneficie de alguma inteligência de decisão, pois ao não sabermos o que vamos encontrar, A medida que avançamos no ambiente desconhecido, e são efetuados mais missões podemos recolher novas informações sobre o meio envolvente e possíveis acontecimentos. Como tal este algoritmo tem de ser fácil de afinar e de aumentar o número algoritmos de reação do robô. Sendo um objetivo deste projeto utilizar manobras independentes para cada cenário, é necessário um algoritmo de controlo capaz de utilizar estes blocos de código. Tendo em conta estes requisitos, duas metodologias *Finite State Machine* (FSM) e *Behavior Trees*(BT), abaixo descritos, enquadram-se no tipo controlo que procuramos para este projeto.

3.3.1 Finite State Machine (FSM)

Finite State Machine (FSM) são um dos muitos tipos de autómatos. Em termos de hierarquia, eles são considerados um algoritmo de nível baixo, ou seja, em relação a autómatos como *push-down*[14] ou máquinas de Turing o seu poder relativo não é grande [15]. No entanto, eles são suficientes para muitas tarefas em várias aplicações e áreas (incluindo a robótica), onde podem ajudar significativamente a implementar de forma eficiente tarefas. Além disso, devido a sua natureza podem ser consideradas fiáveis, pois apresentam metodologias precisas e de fácil compreensão de antecipar acontecimentos e resultados, antes da sua implementação. Tudo isto permite a construção um sistema poderoso e complexo, partindo peças simples.

A figura 3.9 e 3.10 representa uma FSM de uma forma visual, e todos os elementos necessários para a sua construção. Uma FSM é sempre composta pelos estados ou "*sta-*

tes"representados por círculos, e pelas transições ou "*transitions* representadas por setas. Uma FSM muda de estado sempre que um evento ocorre, provocando assim a mudança do estado atual para o estado seguinte através da transição correspondente.

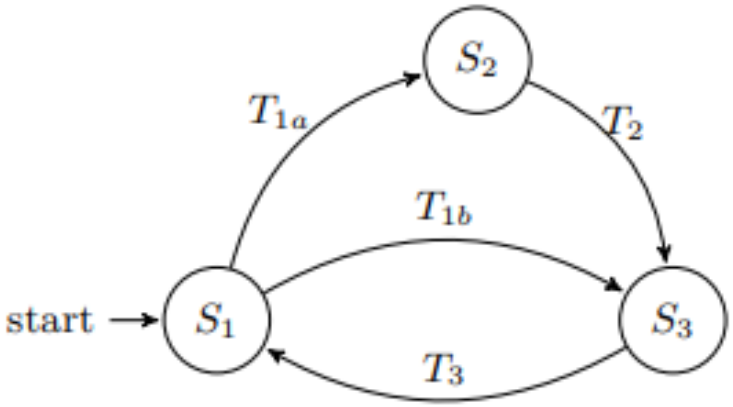


Figura 3.9: Exemplo geral de um diagrama de uma FSM. [15]

Formal definition: A FSM is a five-tuple $\mathcal{A} = (S, \Sigma, \delta, s_0, F)$, where:

- S is a finite, non-empty set of states,
- Σ is the input alphabet (a finite, non-empty set of symbols),
- δ is the transition function: $\delta : S \times \Sigma \to S$,
- s_0 is the initial state, $s_0 \in S$ and
- F is the final state set, $F \subseteq S$.

Figura 3.10: Nomenclatura e condições dos elementos presentes numa FSM.[15]

Para ser mais fácil de compreender e como implementar uma FSM, podemos utilizar um caso simples e muito usual da robótica, um robô que segue uma linha. Para este exemplo iremos considerar que o veículo apenas tem 2 sensores, e que o *output* dos sensores é de 1 quando se encontra em cima da linha preta e 0 quando se encontra em cima do branco. Tendo estas informações podemos construir uma FSM muito simples com apenas 4 estados possíveis(Na linha, Muito á direita, Muito á esquerda, Fora da pista). As figuras 3.11 e 3.12 representam os cenários possíveis do robô á esquerda e o diagrama da FSM correspondente á direita.

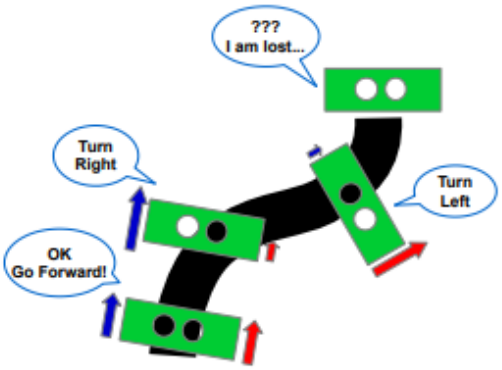


Figura 3.11: Possíveis situações num robô que segue uma linha.[15]

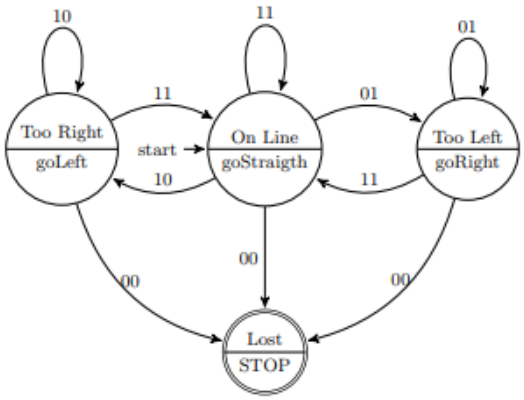


Figura 3.12: Diagrama de estados de um robô que segue uma linha.[15]

No entanto, um processo de desenvolvimento muito típico nesses casos é a criação de um sistema simples que ao longo do tempo é aprimorado para cobrir mais funcionalidades de forma a resolver casos especiais e detalhes específicos. Isso, muitas vezes leva a um sistema que cresce muito além da compreensão e manutenção devido a necessidade de haver sempre mais estados e transições que resolvam esses novos problemas. Esta necessidade de aumentar o algoritmo faz com que seja confuso para o utilizador e difícil de implementar.

3.3.2 Behavior Trees (BT)

As *Behavior Tree* (BT) são algoritmos hierárquicos [16] [17], em forma de árvore uti-

lizados em agentes autónomos de forma a alterar o comportamento dos mesmos face ao aparecimento de uma nova tarefa conhecida. Estes algoritmos foram criados a indústria dos jogos para definir comportamentos para os *non-player character* (NPC). Tanto o Unreal Engine (UE) quanto o Unity (dois dos maiores *softwares* desta área) têm ferramentas dedicadas para criar BTs. Isso não é surpresa, uma grande vantagem dos BTs é que eles são fáceis de compor e modificar, mesmo em tempo de execução. Estas personagens desenvolvidas nos jogos normalmente têm diversas linhas de pensamento muito básicas, mas quando os jogos precisam de muitos NPC´s torna-se um processo demorado e trabalhoso. A solução adotada pelos programadores passa pela utilização das BT's como organizador em cada NPC. Os programadores desenvolvem cada tarefa individualmente, ou seja, criam algoritmos que desempenham as tarefas requeridas independentemente de qual NPC as irá utilizar. Onde, após a conclusão de todas as tarefas básicas definem na árvore de cada NPC quais as tarefas a serem adicionadas e as condições necessárias para entrarem em execução. Um exemplo de como funcionam pode ser observado nas figuras 3.13 e 3.14.

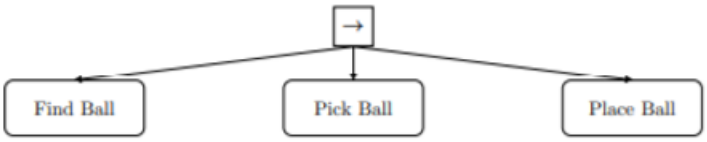


Figura 3.13: Comportamento de um NPC perante uma bola, usando uma BT. [18]

Como se pode observar, o programador apenas teve que definir uma sequência de como um NPC se deveria comportar em relação a uma bola sem ter de se preocupar com o algoritmo por detrás de cada tarefa. Uma forma de nas BT's aumentar o nível de detalhe, é apenas desenvolver novos algoritmos mais minuciosos e adicioná-los na ordem correta a árvore, como se pode observar na figura 3.14.

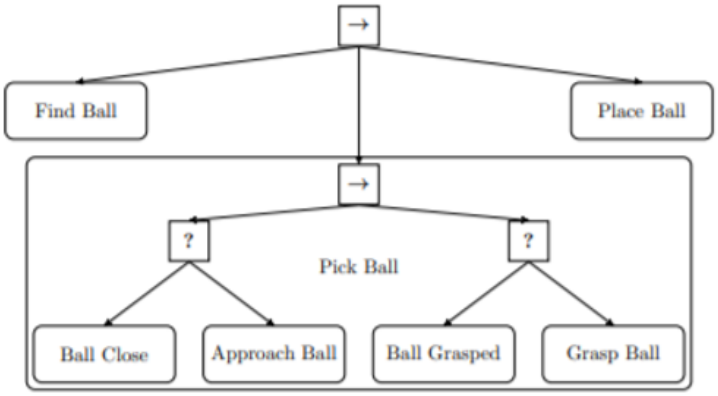


Figura 3.14: Adição de novos algoritmos à BT, para melhorar as capacidades do NPC.[18]

Desde então, os BTs também entraram no domínio da robótica, pois os robôs se tornaram cada vez mais capazes de fazer mais do que simples tarefas repetitivas. Facilmente podemos obter mais conhecimentos e recursos através do livro "Behavior Trees in Robotics and AI: An Introduction" de Michele Colledanchise e Petter Ögren [17].

A variedade de Nós a utilizar e respetivas condições faz com que seja um processo fácil para construir uma BT após o desenvolvimento de algoritmos independentes. A escolha deste algoritmo também teve em conta a sua aplicação em ROS, e o tipo de projetos que fazem uso destes algoritmos. Atualmente já existe uma grande variedade de bibliotecas públicas, tema abordado capítulo 5 que implementam as estruturas das BT em ROS, inclusivamente é possível encontrar um exemplos funcionais utilizando robôs frequentemente usados como o NAO um robô humanoide.

3.4 Discussão do Estado da Arte

Na análise efetuada do estado da arte, foram identificados alguns dos melhores AUV, apresentando as suas vantagens e focos de aplicação, assim como características diversas que apoiam à decisão na escolha para o trabalho em estudo. As diferentes abordagens, efetuadas pelos diferentes veículos ao ambiente de exploração dá uma boa base para a construção das manobras que o UX-1 irá desempenhar.

A arquitetura de controlo escolhida para o projeto serão as *Behavior Tree*, pois foi um dos objetivos a incluir nesta dissertação. Não tirando mérito, as FSM poderiam desempenhar um ótimo trabalho, devido a sua estrutura e á facilidade de verificar erros e problemas antes da sua execução. No entanto, devido a sua expansão ser problemática

devido a sua complexidade, esta arquitetura será mais adequada para cenários mais previsíveis.

A existência de simuladores/motores de jogos como *Unrel Engine* 5 (UE5) e *Unity*, que ja fazem uso das BT's permite ser mais fácil estudar e perceber como são abordados os diversos cenários, e fazer uso das mesmas nos problemas encontrados para o UX-1. Devido ao UX-1 ser embutido com uma grande variedade de sensores, será interessante comparar o tipo de abordagem entre ele e um simples NPC ao mesmo caso.

A combinação de sensores a utilizar é algo condicionada pelos requisitos do ambiente de exploração, o robô não pode depender de elementos externos, pelo que ficam excluídas metodologias com recurso a *beacons* ou marcadores. O robô deve ser capaz de executar a sua missão apenas usando As manobras construídas, usando a BT de missão como controlo.

Esta página foi intencionalmente deixada em branco.

Capítulo 4

Fundamentos Teóricos

4.1 Introdução

Como referido no inicio desta dissertação, todo o trabalho é feito num ambiente de desenvolvimento. Estes ambientes são muito práticos pois permitem o acesso a inúmeras ferramentas de desenvolvimento como:

- Editores de programação: Programas que permitem aos utilizadores, programar e fazer *debbug* dos respetivos códigos, nos diferentes tipos de linguagens como C/C++, Python, Java, etc.

- *Softwares* de cálculo nominal: Aplicações de *software* como por exemplo o Matlab e o Octave.

- Simuladores: Ferramentas de trabalho onde se é possível desenvolver objetos de diferentes dimensões e materiais e simular os mesmos em cenários com diferentes condições físicas e atmosféricas de forma podermos saber como se vai comportar na realidade. Dependendo da aplicação podermos criar usando o SolidWorks, AutoCad, Fusion360, e simular usando o UE5, Unity, Gazebo.

Na secção seguinte vai-se abordar o sistema ROS, pois todo o projeto UNEXMIN foi desenvolvido e implementado no ROS, como tal será necessário ter alguns conhecimentos do mesmo e noção do seu funcionamento.

4.2 ROS

O ROS (Robot Operating System)[19] é um *middleware open-source*, que proporciona um ambiente de desenvolvimento de para a aplicações de robótica. O ROS oferece uma plataforma de software padrão para utilizadores de todos os setores, proporcionando uma experiência boa de pesquisa, desenvolvimento, implementação e finalmente produção.

Atualmente o ROS é utilizado em inúmeras aplicações, por todo o mundo ele é utilizado no desenvolvimento de aplicações no campo da robótica por parte de estudantes, projetos, empresas, etc. O ROS não é um sistema operativo, mas sim um *middleware*. Isso permite a capacidade de gerir a complexidade e heterogeneidade do *hardware* e respetivos aplicativos, de forma a promover a integração de novas tecnologias, simplificando o projeto de *software*, ocultando a comunicação entre os algoritmos de baixo nível[20]. Atualmente, ele apresenta um crescimento no número de aplicações e utilizadores, pois uma das vantagens dele é a os seus projetos poderem ser partilhados e utilizados por todos os membros da comunidade.Uma das barreiras que ele enfrenta é o facto de grande parte das suas funcionalidades, só poderem ser utilizadas em Linux, pois a grande parte do mundo utiliza o Windows como sistema operativo nos seus computadores e a conversão de um para outro nem sempre é a mais simpática e prática.

O ROS apresenta muitas funcionalidades e vantagens, entre elas:

- Funcionamento descentralizado: Uma das vantagens do ROS é capacidade de criar ***Nodes*** ou Nós. Estes são pequeno blocos de código que podem executar tarefas simples. Em termos de hierarquia, eles estão no nível. Os Nós são agrupados em ***package*** ou pacotes que são programas mais complexos. O ***Master Core*** ou Nó principal é o responsável ao mais alto nível para gerir os restantes pacotes e nós.

- Arquitetura e regras: Da forma como ele foi construído, o ROS permite ao utilizador usar o Nós próprios ou externos de forma muito simples. Pois devido a regras impostas na sua construção, a comunicação entre Nós funciona a base de um serviço de *Publisher-Subscriver*. Este serviço permite a um Nó publicar dados de um certo tópico, e um outro Nó importar esse tópico. Isto é vantajoso, pois o *Master* efetua a ligação entre os dois, não interessando quantos mais nós estão a funcionar, nem a linguagem de programação em que estão a trabalhar.

- Tutoriais e Comunidade: O ROS por ser grátis e utilizado por grande parte da comunidade da robótica, disponibiliza imensos tutorias de aprendizagem com exemplos práticos. Além disso existe um grande reportório de Nós, Projetos e ideias feitos por utilizadores disponíveis para todos em arquivos online como o GitHub.

4.2.1 Gazebo

O Gazebo é um ambiente de simulação que faz uso de um conjunto de bibliotecas dedicadas á simulação, renderização, comunicação e interface gráfico.A simulação é constituída por dois programas executáveis :

- *gzserver*: Este é responsável pelo o processamento de todos os dados e a construção da simulação com todos os objetos e cenários.

- *gzclient*: Como o nome indica, este segundo programa é responsável pela parte do utilizador, permitindo ao utilizador visualizar através de um GUI, a simulação e interagir com a mesma.

A gestão de tópicos é realizados pelo Nó *Master* do Gazebo. Sendo este, capaz de controlar todos os restantes nós que lidam com a simulação, serviços de comunicação e processamento de dados.

O Gazebo é constituído por três conjuntos de bibliotecas. Um conjunto de bibliotecas de renderização que utiliza OGRE, que fornece uma interface simples para renderizar cenas 3D para as bibliotecas de GUI. Em segundo a biblioteca relacionadas com a física da simulação, esta fornece componentes fundamentais para os objetos a serem simulados, como a rigidez, colisões, interações com o meio envolvente. ODE, Bullet, Simbody, DART são algumas das bibliotecas *open-source*. Por últimos temos as bibliotecas responsáveis pelo GUI. Esta biblioteca utiliza Qt para criar *widgets* gráficos para simulação.

A figura 4.1 representa a arquitetura da integração Gazebo-ROS.

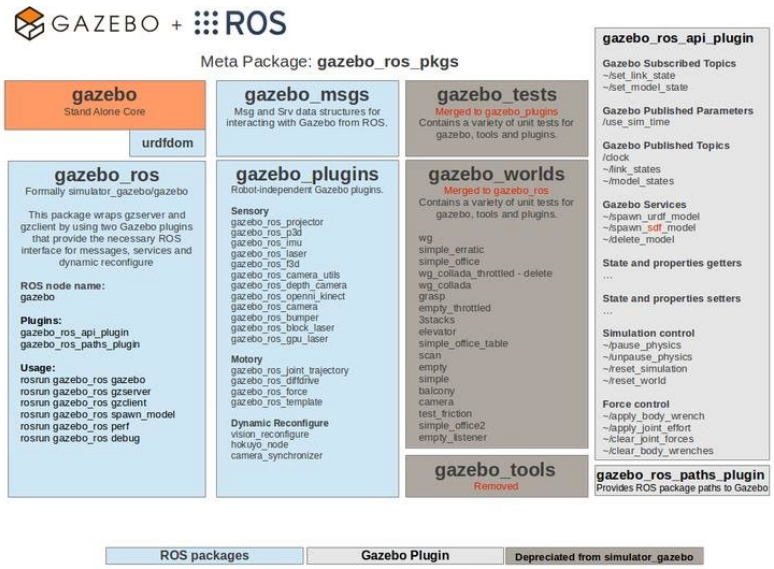


Figura 4.1: gazebo-ros. [21]

4.3 Arquitetura do UX-1

4.3.1 Corpo externo do UX-1

O robô possui um casco de alumínio, resistente a pressão e a oxidação. Conta também com um total de 8 propulsores, estes estão localizados nas duas partes laterais do casco. Devido a configuração do robô é possível atingir 5 graus de movimento, 2 lineares e 3 rotativos, faltando assim poder executar movimentos laterais.

4.3.2 Arquitetura do Hardware

O robô UX-1, 1.1 possui um sistema de computacional distribuído, figura 4.2, para processamento de dados a bordo, interface dos sensor e controlo dos atuadores. O computador principal é responsável pelo controle da missão principal. Este computador tem incorporado o *software* de navegação, orientação e controlo de alto nível. O CPU principal também é responsável realizar armazenar e processar a integração de dados dos vários sensores.

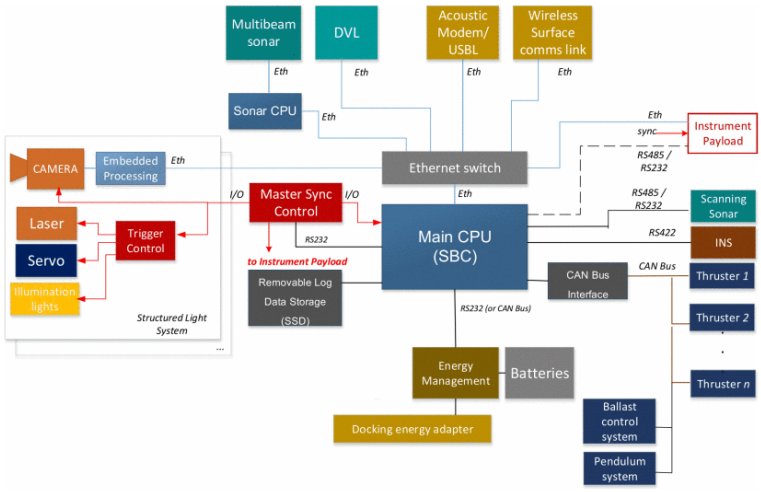


Figura 4.2: Arquitetura de software do UX-1. [1]

Como seria de esperar, o CPU principal tem funções mais importantes do que processar os dados de todos os sensores. Como tal existem CPUs dedicados para tarefas intensivas de processamento de sensores específicos. Um destes é responsável pelo pré-processamento bruto de dados de sonar M3 *Multibeam Sonar* fornecendo uma nuvem de pontos 3D de saída que é depois enviada para o computador principal. Além disso existem um conjunto de 2 CPUs de processamento de imagem de baixo consumo de energia para processar todas as câmeras. Estes são responsáveis pelo processamento de imagem de baixo nível e, em particular, são usados para o processamento em tempo real de dados. Todas as aquisições de dados dos sensores são controladas em conjunto, permitindo assim um sincronização temporal dos dados.

Os sensores comunicam através de Ethernet, RS485/422 ou barramento CAN. Sensores de grande largura de banda, como o sonar *Multibeam Sonar* ou as câmeras, usam *Ethernet*, e os de baixo rendimento usam linha de CAN. Este último é o caso do subsistema que gere a de energia e controlo dos atuadores (propulsores, pêndulo e sistema de lastro variável).

4.3.3 Arquitetura do Software

O software do computador principal do robô é organizado em uma estrutura modular

com vários processos executados num sistema operacional á base de Linux. O *framework* de *middleware* ROS (Robotic Operating System) [8] é usado na implementação para fornecer mecanismos *standard* de intercomunicação entre os múltiplos processos e para facilitar o processo de desenvolvimento de software. As ferramentas e a estrutura do software ROS permitiram a fácil integração dos módulos desenvolvidos por diferentes equipas englobadas no projeto.

A estrutura principal do software está representada na figura 4.3. São quatro componentes principais, responsáveis pelas funcionalidades de *Sensor Fusion*, SLAM, GNC e *Low Level Control*. Cada módulo foi desenvolvido por uma equipe diferente de investigadores, por isso foi necessário um cuidado especial no desenvolvimento para que a integração fosse o mais simples.

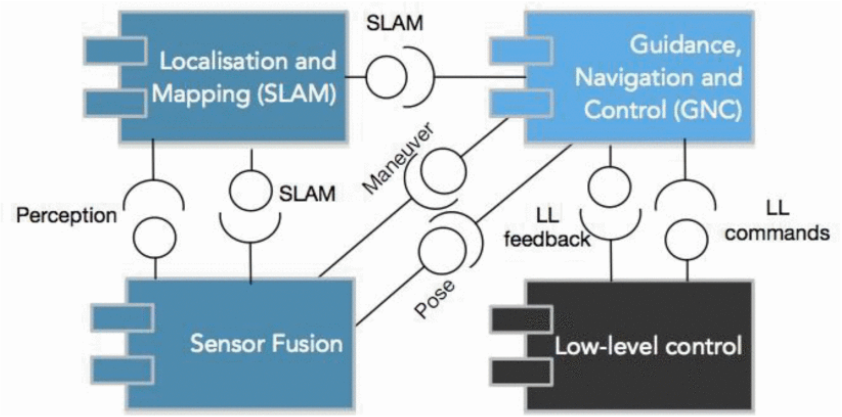


Figura 4.3: Módulos principais de software do UX-1. [1]

O módulo do SLAM é responsável por receber as informações de Posição e Perceção (pointClouds, informações de localização) fornecidas pelo módulo *Sensor Fusion*. Internamente a localização e o mapeamento são dois módulos que compartilham as informações. O módulo do SLAM, é responsável por fornecer dados de mapas e localização global e o Mapeamento é responsável pelo a gerência das estruturas dos mapas (locais, topológicos).

O módulo *Sensor Fusion* é responsável por fornecer dados de posição e perceção, para os módulos de localização e mapeamento. Internamente a fusão de sensores é composta

por três módulos: Estimativa de Posição, Perceção de Sensores, Aquisição de Dados e Registo.

O módulo GNC é formado por três componentes principais: Planner, Guidance e Navigation. O primeiro é o módulo de alto nível encarregado de decidir a estratégia geral de movimento e enviar ações (como por exemplo, virar, parar, avançar) para o módulo Orientação. Este calcula as trajetórias na forma de *waypoints*. Os *waypoints* são alimentados no módulo de navegação que gera perfis de velocidade que são finalmente transformados em comandos de controlo, que por sua vez são alimentados ao módulo de controlo de baixo nível. Este consiste em dois programas monolíticos que executam os dois micro-controladores diretamente conectados aos atuadores.

4.3.4 Instrumentação e navegação

O UX-1 está equipado com um conjunto de sensores utilizados para navegação e perceção do ambiente. A localização do robô usa um IMU baseado em fibra ótica (KVH 1750), figura 4.4 para medir a aceleração linear e velocidade angular. O Doppler Velocity Log (Nortek IMHz DVL), figura 4.5, é utilizado para obter uma estimativa da velocidade relativa do veículo à parede da mina. Os marcos naturais são extraídos no ambiente pelo uso de sensores de imagem e alcance e são integrados ao *framework* SLAM para corrigir os dados de *dead reckoning*.

Figura 4.4: IMU KVH 1750. [22]

Figura 4.5: Nortek IMHz DVL. [23]

O AUV tem incorporado quatro Structured Light Sensor(SLS) desenvolvidos para fornecem dados detalhados em forma de nuvem de pontos da morfologia do ambiente. Estes sistemas SLS são compostos por uma câmera digital, uma CPU de processador de imagem dedicado e um sistema de projetor de luz. O projetor de luz possui uma fonte de luz visível, um projetor LED de luz UV e um projetor de laser rotativo. Tudo isso é sincronizado com o sinal de disparo da captura da câmera e com um pulso de sincronização de tempo global do robô.

Um sonar *Multibeam Sonar* (Kongsberg M3), figura 4.6, também é usado para mapear e fornecer uma imagem 2D em forma de *pointcloud*. O robô possui sonar de varredura mecânico usado principalmente para deteção de obstáculos, embora possa ser integrado na solução de navegação [24]. O sistema SLS também pode fornecer imagens digitais *standart* que podem ser usadas em tempo real para navegação baseada em vídeo e pós-processadas posteriormente para obter imagens da mina contínua.

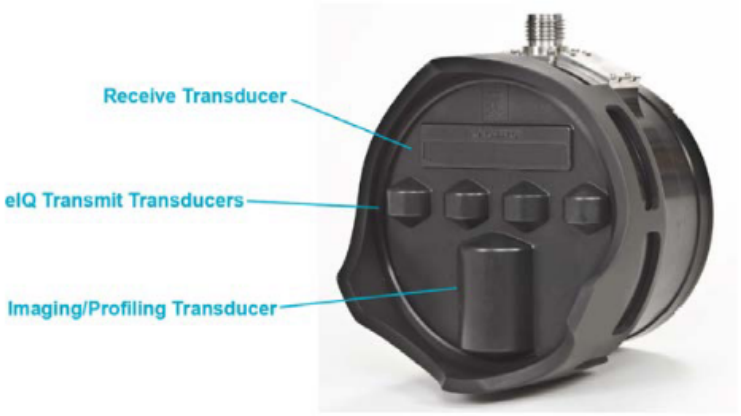


Figura 4.6: M3 Kongsberg Multibeam Sonar.

4.4 Behaviour Tree

Uma BT é composta por três tipos de nós de execução:

- **Raiz ou (Root Node):** Uma BT inicia o seu processo no root node, que em seguida executa os nós filhos por ordem, numa determinada frequência.

- **Nó de Controlo ou Execução (Control Node or Executation Node):** Este nó pode ter 4 variantes distintas(sequence, fallback, parallel ou decorator) e ainda 2 variantes de execução(action ou condition). As condições necessárias à utilização e respetivas condições de sucesso e falhanço, podem ser analisadas através da Figura 4.12.

- **Folhas ou (Leaf Node):** Na base da árvore este nó executa os pequenos algoritmos programados retornando o seu estado e o resultado da tarefa.

Na figura 4.7, podemos observar as quatro variantes de nós de controlo. Estes são como referido: **sequence**, **fallback**, **parallel** e **decorator**, respetivamente.

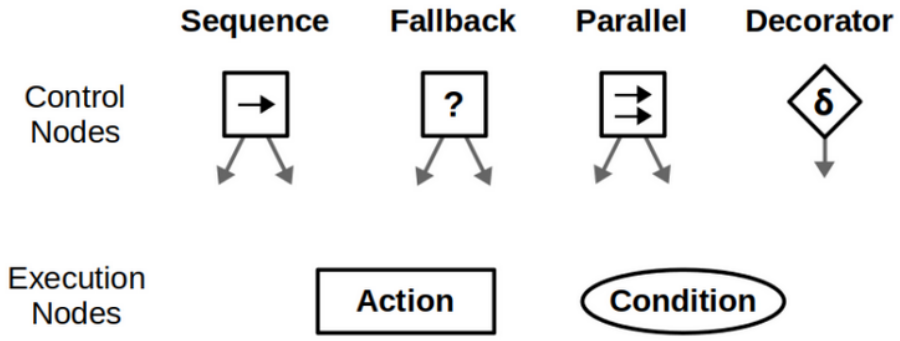


Figura 4.7: Nós de controlo.

Os nós **Sequence**, figura 4.8 como nome indica, promove uma sequência de tarefas. Estas tarefas são executadas por ordem, o nó termina quando a tarefa em funcionamento retorna falha ou então quando todas as tarefas retornaram sucesso.

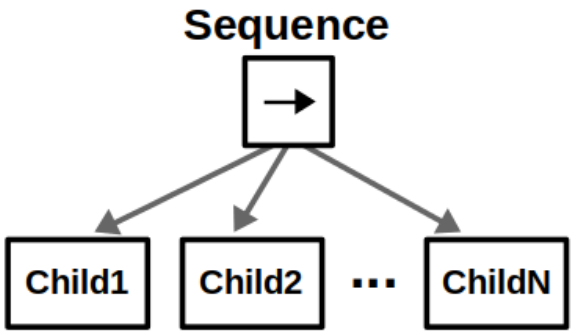


Figura 4.8: Sequence Node.

Os Nós de **Fallback**, são utilizados em decisões. Pois são programados para terminar caso alguma das folhas retorne sucesso, ou quando todas as folhas retornaram falhanço. Um exemplo de atuação é a simples decisão de virar num labirinto. O algoritmo irá decidir primeiro se pode virar a direita, caso seja verdade retorna sucesso e termina, caso de falha repete o processo para as restantes direções até alguma se possível ou então como todas falharam termina significando tem de voltar para trás.

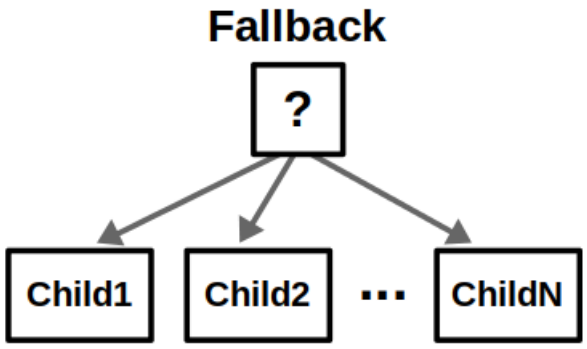


Figura 4.9: Fallback Node.

Os Nós de **Parallel**, permitem executar múltiplas tarefas em simultâneo, claro que sabemos que não existe verdadeiro paralelismo, mas pelo menos a cada *tick* do computador o nó seguinte executado. Este tipo de nós é bom por exemplo para procurar múltiplos objetos, pois não é necessário procurar um objeto de cada vez pois a medida que nos movimentamos podemos encontrar os outros. Estes nós retornam sucesso quando pelo menos uma das folhas retornou sucesso, ou então falha quando todas as folhas falharam.

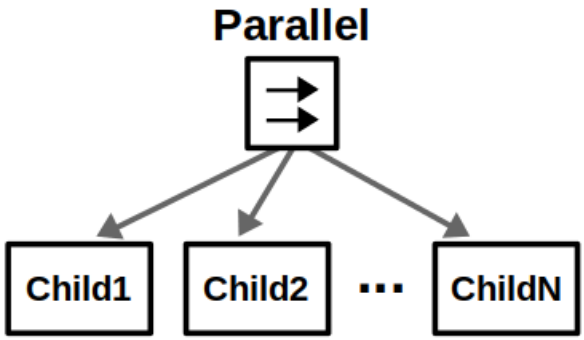


Figura 4.10: Parallel Node.

Por último temos os **Decorator**, estes nós permitem ao utilizador mudar as condições de uma única folha, por exemplo podendo inverter as condições de sucesso e falha, ou até mesmo o número de tentativas que a tarefa vai ser executada. Este nó apenas é utilizado em casos muito específicos, tanto que certas bibliotecas não os têm desenvolvidos, e recomendam utilizar os restantes três.

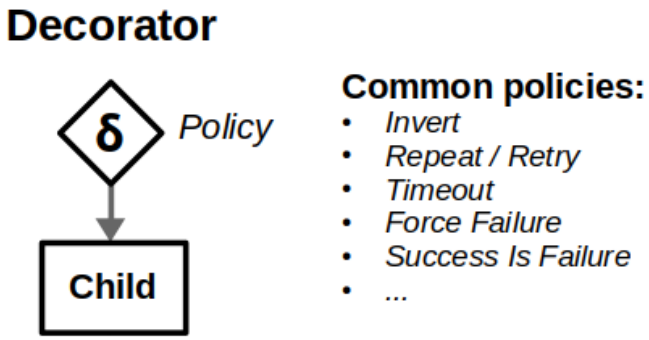


Figura 4.11: Decorator Node.

Na figura 4.12 abaixo podemos ver resumido, todos os nós e as respetivas condições de sucesso e falha.

Node type	Symbol	Succeeds	Fails	Running
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	→	If all children succeed	If one child fails	If one child returns Running
Parallel	⇉	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	◊	Custom	Custom	Custom

Figura 4.12: Tipos de nós de uma BT, e respetivas condições.[18]

Um dos problemas das *behaviours trees* é a repetição, como exemplificado neste artigo [25], temos o cenário de um robô que é responsável por apanhar laranjas e maças em diferentes localizações. Quando apenas temos uma localização podemos utilizar a BT exemplificada na figura 4.13.

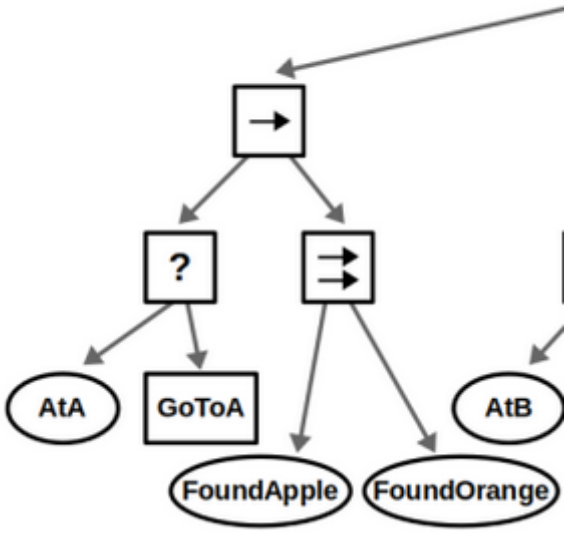


Figura 4.13: BT que permite a um robô ir apanhar laranjas e maças numa determinada localização.

Mesmo um utilizador não entendendo nada de BTs, consegue entender que se quiser adicionar outra localização basta escalar a BT e adicionar cópias do algoritmo descrito em cima, de forma ao robô poder apanhar as laranajs e maças em diferentes localizações. Essa expansão pode ser observada na figura 4.14.

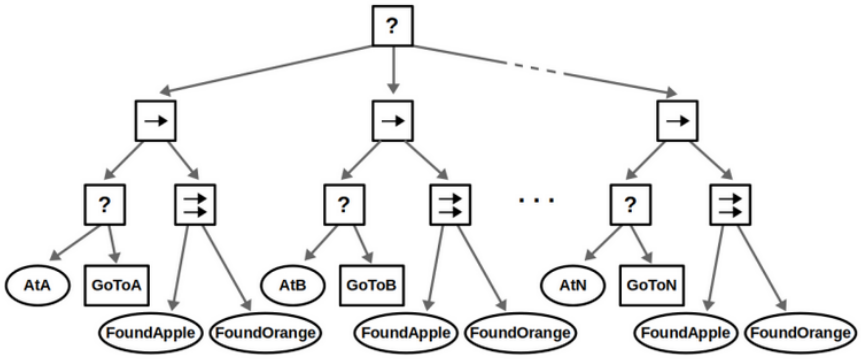


Figura 4.14: Expansão da BT, para o robô poder executar a mesma tarefa em diferentes tarefas.

Este processo apesar de ser simples e repetitivo, se for utilizado num daqueles robôs

que trabalham nos armazéns de empresas como Amazon, FedEx, etc... pode ser tornar caótico pois esses robôs por vezes 20 a 30, pontos de recolha como tal teríamos uma árvore gigante. Este problema pode ser resolvido, como exemplificado na figura 4.15, utilizando um nó *Decorator* com a condição de repetir a BT o número de vezes necessárias em conjunto com uma *BlackBoard* (Bloco de *software* disponível em grande parte das bibliotecas de BT, capaz de armazenar dados), para guardar as coordenadas das diferentes localizações.

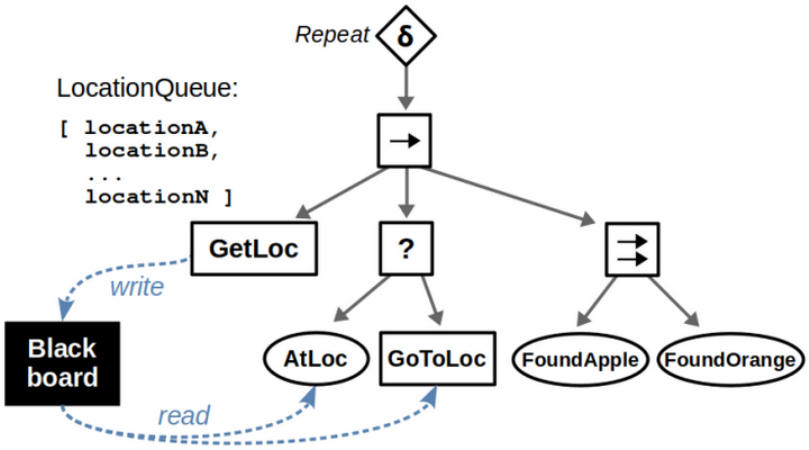


Figura 4.15: Utilização de uma *BlackBoard* e um *Decorator* para simplificar a BT.

Atualmente existem inúmeras bibliotecas que permitem aos utilizadores criarem e testarem as suas BT. Algumas das bibliotecas disponíveis são:

Nome	Linguagem	GUI	ROS	Open source	última atualização
`py_trees`	Python		√	√	atual
Owylb	Python			√	Nov. 29, 2014
gdxAI	Java			√	Jul. 26, 2019
Behavior3	JavaScript	√		√	Oct. 21, 2018
BehaviorTree.js	JavaScript			√	May. 3, 2019
NPBehave	Unity3D/`C#`	√		√	Oct. 24, 2019
BehaviorTree.CPP	C++	√	√	√	atual
ROS-Behavior-Tree	C++	√		√	Oct. 17, 2018

Tabela 4.1: Bibliotecas de BT disponíveis.

- **Owyl**: é uma das bibliotecas de BT inicialmente desenvolvidas e uma das primei-

ras a ser implementada em Python. Com o aparecimento de novas bibliotecas, rapidamente deixou de ser utilizada.

- **gdxAI**: é um *framework* escrito em Java para desenvolvimento de jogos com o libGDX. Ele suporta recursos de movimento para *Artificial intelligence* (AI), *path-finding*, tomada de decisões, tanto via BTs como FSMs.

- **Behavior3**: é uma biblioteca desenvolvida pelos autores de [26]. Possui um editor visual, o Behavior3 Editor (correspondente aos dados da tabela 4.1), que pode exportar as árvores modeladas para arquivos JSON ou ser utilizada por bibliotecas á base de Python e Javascript. Embora um editor visual simplifique o design e a visualização, existe falta de documentação, manutenção regular. Além disso não apresenta suporte para ROS.

- **BehaviorTree.js**: é outra biblioteca orientada ao *design* de AI em jogos implementada em Javascript. Um dos problemas desta biblioteca é ser limitada no processo de execução dos nós, pois os nós apenas retornam Sucesso ou Falha, faltando a execução. Criando o problema de não ser possível identificar que nós se encontram a funcionar.

- **NPBehave**: é uma biblioteca BT que visa a Artificial Intelligence (AI) de jogos para o Unreal Engine, assim como outras bibliotecas: **BT-Framework**, **fluid-behavior-tree** e **hivemind**, que vem com um editor visual com acesso a um *debugger* de recursos em tempo real. Em particular, NPBehave é uma biblioteca orientada a eventos, ou seja, possui árvores de comportamento orientadas a eventos que não precisam ser executadas a partir do nó principal a cada *tick*. Este design é fortemente apoiado pela comunidade por ser mais eficiente e mais simples de usar.

- **ROS-Behavior-Tree**: é uma biblioteca implementada pelos autores em [27]. De acordo com a documentação, a versão mais recente do ROS não é suportada de momento e a última atualização foi em outubro de 2018. Uma provável razão para este estado é porque os autores colaboraram na criação do **BehaviorTree.CPP** que apresenta o GUI editor Groot e uma implementação em ROS.

Para este projeto decidiu-se utilizar as **BehaviorTree.CPP**, criada por Davide Faconti. Esta escolha deveu-se por existir um grande reportório de documentação e exemplos de como executar certas tarefas. Além disso esta biblioteca, já foi num grande projeto robótico RobMoSys, incluí compatibilidade com o editor gráfico Groot e continua ativamente a ser desenvolvida para funcionar com o ROS e futuramente o ROS2.

A **BehaviorTree.CPP** encontra-se disponível para todos no GitHub, e quase compatível com o ROS, sendo apenas modificar alguns *Plugins*.

Como interface gráfico, existe como indicado em cima o Groot, figura 4.16. Este GUI permite a construção de uma BT, utilizando blocos já pré-definidos com acesso a todo o tipo de nós e condições.

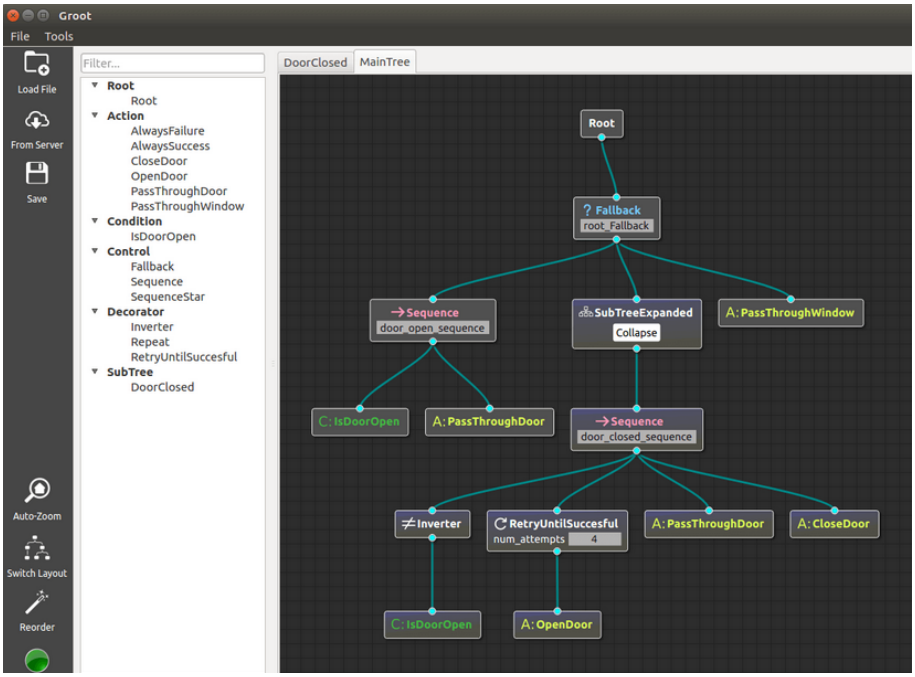


Figura 4.16: GUI do Groot.

Após a construção de uma BT, é possível exportar um ficheiro com a BT com o formato (.XML), exemplificado a abaixo na figura 4.21, onde se encontra a sequência definida pelo utilizador que depois pode ser importada no nosso programa, utilizando uma das funções da bilioteca da **BehaviorTree.CPP**, figura 4.17.

Caso seja necessário receber e enviar *Inputs* e *Outputs* entre a Bt e o programa. A biblioteca disponibiliza um protocolo de comunicação por Portas, figura 4.18 onde um nó envia uma informação e outra fica a espera.

```cpp
int main()
{
    // We use the BehaviorTreeFactory to register our custom nodes
  BehaviorTreeFactory factory;

  // The recommended way to create a Node is through inheritance.
  factory.registerNodeType<ApproachObject>("ApproachObject");

  // Registering a SimpleActionNode using a function pointer.
  // You can use C++11 lambdas or std::bind
  factory.registerSimpleCondition("CheckBattery", [&](TreeNode&) { return CheckBattery(); });

  //You can also create SimpleActionNodes using methods of a class
  GripperInterface gripper;
  factory.registerSimpleAction("OpenGripper", [&](TreeNode&){ return gripper.open(); } );
  factory.registerSimpleAction("CloseGripper", [&](TreeNode&){ return gripper.close(); } );

  // Trees are created at deployment-time (i.e. at run-time, but only
  // once at the beginning).

  // IMPORTANT: when the object "tree" goes out of scope, all the
  // TreeNodes are destroyed
  auto tree = factory.createTreeFromFile("./my_tree.xml");

  // To "execute" a Tree you need to "tick" it.
  // The tick is propagated to the children based on the logic of the tree.
  // In this case, the entire sequence is executed, because all the children
  // of the Sequence return SUCCESS.
  tree.tickWhileRunning();
```

Figura 4.17: Utilização de uma BT em formato (.XML).

Na figura 4.18 abaixo, podemos ver uma simples comunicação pré definida entre dois nós. Onde após o envio da palavra *"Hello"*, a Bt tem guardado que deve responder *"the answer is 42"*.

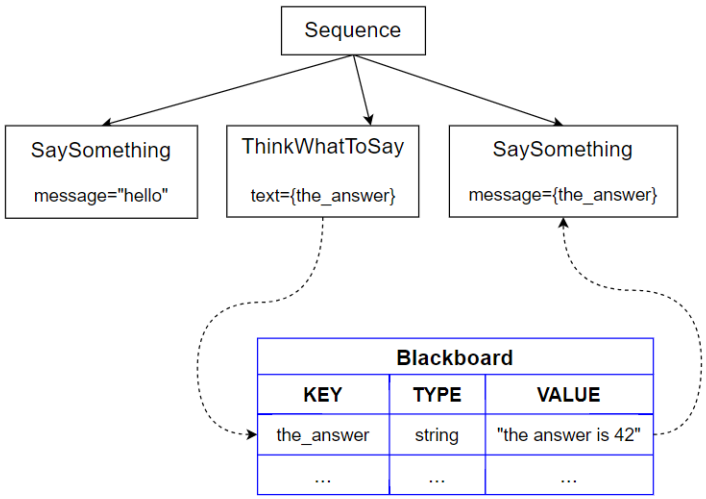


Figura 4.18: Protocolo de comunicação por Portas.

Na figura 4.19, podemos observar simples árvore, o objetivo deste programa é um manípulo ser capaz de agarrar um objeto. Como podemos ver esta árvore usa um nó sequencial, pois testa em primeiro a bateria do robô, seguindo para execução de abrir, aproximar e finalmente agarrar.

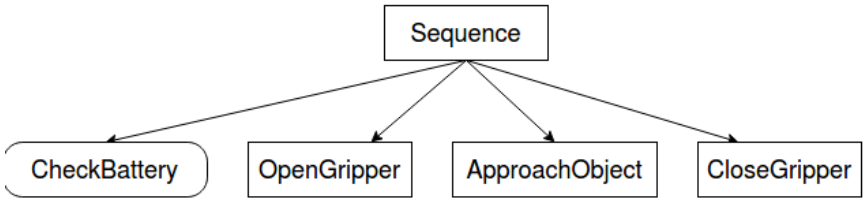


Figura 4.19: BT sequencial, para uma simples tarefa.

Nas figuras 4.20 e 4.21, podemos ver as vantagens de utilizar o Groot. Na Primeira figura podemos observar como seria programar a BT acima descrita usando a biblioteca **BehaviorTree.CPP** e em segundo importando o ficheiro (.XML) criado no Groot.

```
using namespace BT;

// Simple function that return a NodeStatus
BT::NodeStatus CheckBattery()
{
    std::cout << "[ Battery: OK ]" << std::endl;
    return BT::NodeStatus::SUCCESS;
}

// We want to wrap into an ActionNode the methods open() and close()
class GripperInterface
{
public:
    GripperInterface(): _open(true) {}

    NodeStatus open() {
        _open = true;
        std::cout << "GripperInterface::open" << std::endl;
        return NodeStatus::SUCCESS;
    }

    NodeStatus close() {
        std::cout << "GripperInterface::close" << std::endl;
        _open = false;
        return NodeStatus::SUCCESS;
    }

private:
    bool _open; // shared information
};
```

Figura 4.20: BT programando manualmente usando a biblioteca.

```
<root main_tree_to_execute = "MainTree" >
    <BehaviorTree ID="MainTree">
       <Sequence name="root_sequence">
           <CheckBattery   name="check_battery"/>
           <OpenGripper    name="open_gripper"/>
           <ApproachObject name="approach_object"/>
           <CloseGripper   name="close_gripper"/>
       </Sequence>
    </BehaviorTree>
</root>
```

Figura 4.21: Ficheiro (.XML) usando o Groot.

Como podemos ver a estrutura do código utilizando Bt em (.XML), além de ser mais simples compreensão, permite ao utilizador modificar facilmente a sua estrutura no Groot sem editar no seu programa.

Esta página foi intencionalmente deixada em branco.

Capítulo 5

Ambiente de Desenvolvimento

5.1 Ambiente de simulação

Como referido anteriormente o ambiente de simulação em gazebo já possui um modelo realístico do UX-1. O robô UX-1 é um pequeno robô esférico [28], figura 5.1, onde será possível interagir com atuadores do veículo como os propulsores, o pêndulo que controla o centro de massa, e a bomba de flutuabilidade. Será possível obter informação dos principais sensores existentes a bordo, como o MSIS-Micron DST, o *Doppler Velocity Log* (DVL), o M3 *Multibeam Sonar*, os 4 lasers, as 5 câmeras, etc. Todas estas ações e informações serão obtidas através dos tópicos em ROS criados para cada atuador e sensor.

Figura 5.1: Modelo do UX-1 no Gazebo

Para simular as condições encontradas pelo robô, serão utilizadas réplicas em Gazebo do tanque do Laboratório, Figura 5.2 e da mina Kaatiala, Finlândia, Figura 5.3, que foi um dos cinco locais de testes no UX-1 no projecto UNEXMIN.

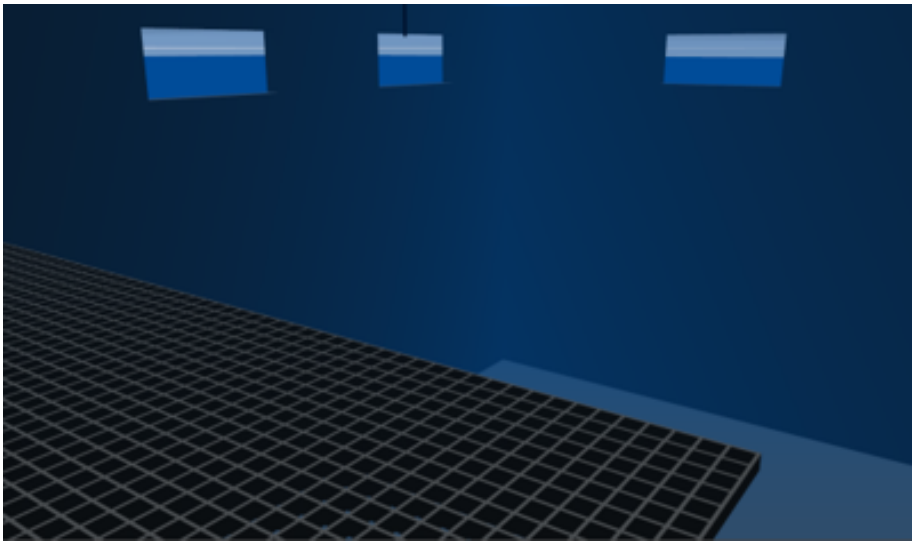


Figura 5.2: Modelo do tanque do laboratório.

Figura 5.3: Modelo da mina Kaatiala, Finlândia.

Além do UX-1 e dos cenários de simulação foi também necessário implementar dois *plugin*, responsáveis pelas físicas do ambiente aquático e pelas texturas. Na figura 5.4, podemos ver o *underwater current plugin*. Adicionando este *plugin* ao modelo 3D da mina Kaatiala, podemos simular condições semelhantes ás encontradas num ambiente subaquático, como gravidade, e resistência da água.

```
    <plugin name="underwater_current_plugin" filename="libuuv_underwater_current_ros_plugin.so">
      <namespace>hydrodynamics</namespace>
      <constant_current>
        <topic>current_velocity</topic>
        <velocity>
          <mean>0</mean>
          <min>0</min>
          <max>0.000001</max>
          <mu>0.0</mu>
          <noiseAmp>0.0</noiseAmp>
        </velocity>

        <horizontal_angle>
          <mean>0</mean>
          <min>0</min>
          <max>1.141592653589793238</max>
          <mu>0.0</mu>
          <noiseAmp>0.0</noiseAmp>
        </horizontal_angle>

        <vertical_angle>
          <mean>0</mean>
          <min>0</min>
          <max>1.141592653589793238</max>
          <mu>0.0</mu>
          <noiseAmp>0.0</noiseAmp>
        </vertical_angle>
      </constant_current>
    </plugin>

    <plugin name="sc_interface" filename="libuuv_sc_ros_interface_plugin.so"/>
```

Figura 5.4: Plugin que simula as físicas do ambiente subaquático

O Segundo *Plugin*, figura 5.5 apenas é responsável pelos gráficos visualizados no si-

mulador, sendo que elementos, como texturas da água, visibilidade subaquática, luz incidente, nuvens, céu são adicionadas de forma tornar a simulação mais realista.

```
<physics name="default_physics" default="true" type="ode">
  <max_step_size>0.002</max_step_size>
  <real_time_factor>1</real_time_factor>
  <real_time_update_rate>500</real_time_update_rate>
  <ode>
    <solver>
      <type>quick</type>
      <iters>50</iters>
      <sor>0.5</sor>
    </solver>
  </ode>
</physics>
<scene>
  <ambient>0.01 0.01 0.01 1.0</ambient>
  <sky>
    <clouds>
      <speed>12</speed>
    </clouds>
  </sky>
  <shadows>1</shadows>
  <fog>
    <color>0.0 0.3 0.6 0.4</color>
   <!-- <color>0.1 0.5 0.1 0.6</color> -->
    <type>linear</type>
    <density>0.001</density>
    <start>0</start>
    <end>80</end>
  </fog>
</scene>
```

Figura 5.5: Plugin que adiciona as texturas ao simulador.

5.2 BehaviorTree.CPP

Como referido no capítulo anterior iremos utilizar a biblioteca **BehaviorTree.CPP** para utilizarmos as BT's.

Este pacote que contém a biblioteca é muito simples de instalar e compilar, basta apenas seguir os passos descritos no GitHub: utilizando o seguinte comandos:

```
sudo apt-get install ros-$ROS_DISTRO-behaviortree-cpp-v3
```

onde `$ROS_DISTRO` é substituído pela distribuição de ROS utilizada, por exemplo kinetic, Indigo etc. Para utilizarmos em conjunto com o ROS, basta instalar e compilar dentro *catkin workspace*.

Em seguida para utilizarmos o GUI Groot com o ROS precisamos de um *plugin* chamado **ZeroMQ** para comunicar entre os dois sendo então necessário seguir os seguintes passos:

Instalação dos pacotes necessários:

```
sudo apt-get install libtool pkg-config build-essential autoconf automake
```

```
sudo apt-get install libzmq-dev
```

Instalação do libsodium:

```
git clone git://github.com/jedisct1/libsodium.git
```

```
cd libsodium
```

```
./autogen.sh
```

```
./configure && make check
```

```
sudo make install
```

```
sudo ldconfig
```

instalação do plugin ZeroMQ:

```
wget http://download.zeromq.org/zeromq-4.1.2.tar.gz
```

```
tar -xvf zeromq-4.1.2.tar.gz
```

```
cd zeromq-4.1.2
```

```
./autogen.sh
```

```
./configure && make check
```

```
sudo make install
```

```
sudo ldconfig
```

Após a instalação do *plugin* podemos instalar então o **Groot**:

Dependências:

```
sudo apt install qtbase5-dev libqt5svg5-dev libzmq3-dev libdw-dev
```

Obter uma cópia do repositório:

```
git clone https://github.com/BehaviorTree/Groot.git
```

Instalação do Groot:

```
cd Groot

git submodule update --init --recursive

mkdir build

cd build

cmake ..

make
```

Capítulo 6

Projeto

Esta secção apresenta a arquitetura de sistema, apresentando os diferentes módulos necessários para capacitação do robô para tarefas em modo autónomo.

6.1 Arquitetura do sistema

A arquitetura de sistema está decomposto em vários módulos, cada um responsável pela gestão de um subsistema do robô. Encontra-se organizado por módulos, de modo a promover uma menor complexidade e ao mesmo tempo permitir construir uma solução de uma forma incremental. Esta é uma estratégia de arquitetura *standard* [29] na robótica denominada por *bahavior-based* introduzida por Ronald C Arkin em [30].

Os diferentes módulos operam concorrentemente sendo necessário haver uma boa relação entre o controlo de missão proporcionado pela BT, as manobras desenvolvidas para os diferentes cenários e as funcionalidades de mobilidade e sensoriais do robô. Os diferentes componentes partilham informação segundo o paradigma de comunicação *publisher – subscriber*, as mensagens são comunicadas assincronamente via *broadcast*, os módulos "interessados" subscrevem a mensagem e recebem uma cópia.

O robô opera segundo a arquitetura descrito pela figura 6.1, uma malha fechada entre o controlo de missão, Controlo de movimento e os atuadores e sensores do UX-1, sendo que o controlo de missão recebe instruções e parâmetros, previamente fornecidos pelo utilizador, de como executar a missão.

- **Missão:** Ao mais alto nível temos o utilizador, este define os parâmetros e os objetivos da missão, esses mesmos podem por exemplo ser: Tempo de exploração, Profundidade, Mapeamento completo, Deslocação até uma posição, etc.

- **Controlo de Missão:** O controlo de missão é responsável por se certificar que os objetivos definidos na missão são cumpridos. Para tal, será utilizada uma BT para decidir o melhor comportamento do UX-1. Esta BT terá acesso a um conjunto de manobras, que em conjunto com o *feedback* dos sensores do robô, poderá escolher a mais adequada para dar resposta aos possíveis cenários de uma mina.

- **Controlo de Movimento:** Este bloco é onde se encontram as manobras disponíveis para a exploração da mina. Estas manobras, descritas na secção seguinte, permitem ao robô movimentar-se pelos vários cenários de atuação(túneis, poços e galerias). Ambas as manobras, fazem uso dos sensores do UX-1 para a deteção e desvio de obstáculos.

- **UX-1:** Ao mais baixo nível da arquitetura temos, o UX-1. Ele é composto pelos vários atuadores e sensores, que executam os comandos necessários para o bom funcionamento da missão. Os sensores do UX-1, são importantes não só nas manobras, mas também para o controlo de missão decidir a melhor manobra a utilizar.

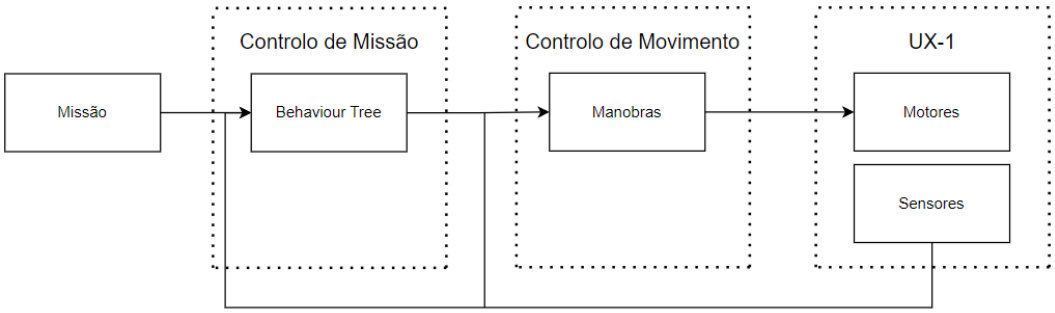


Figura 6.1: Arquitetura do sistema

O simulador comunica através da *framework* ROS com a solução de software desenvolvido. A linguagem de programação utilizada foi o C++, tendo sempre em consideração uma estrutura de código reutilizável e boas práticas de programação.

Tendo por base arquitetura do sistema desenvolvida neste projeto. Após definida uma missão de exploração. Será ativado o controlo de missão onde BT construída para o efeito será responsável pelas decisões do AUV.

Após a certeza do cenário onde se encontra o robô será ativado o controlo de movimento. Este, até mudança de cenário, irá utilizar a manobra apropriada ou cenário em questão. No nível mais baixo, mas não menos importante, os sensores e atuadores do

UX-1 utilizados, os 4SLS, o MSIS-Micron DST, o *Doppler Velocity Log* (DVL), o M3 *Multibeam Sonar*, alimentam os diferentes blocos da arquitetura de controlo do robô. Pois os sensores a abordo, permitiram a identificação do cenário na parte de controlo da missão, seguido de serem necessários para deteção de obstáculos a medida que o UX-1 executa a manobra escolhida pelo controlo.

Além disso as funcionalidade de mapeamento e visão utilizando as múltiplas câmeras existentes abordo vão se encontrar ativas, mas como esse trabalho está ser desenvolvido por outro projeto não será relevante para esta dissertação.

6.2 Manobras de exploração

Como referido nos capítulos anteriores, de forma a dar resposta aos desafios foram pensadas múltiplas manobras. Estas manobras foram pensadas para solucionar os diferentes tipos de cenários de uma mina, como túneis, poços e galerias.

Na figura 6.2 apresentam-se destacadas as manobras a serem desenvolvidas e implementadas no capítulo seguinte desta dissertação como resposta aos diferentes cenários.

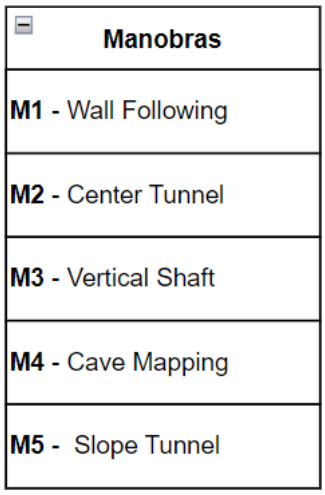


Figura 6.2: Manobras a serem desenvolvidas

A primeira manobra *Wall Following* foi pensada para o objetivo principal do UX-1, que é, identificar possíveis minérios nas paredes da mina. Como tal será necessário desenvolver, uma manobra capaz de colocar o robô a um distância constante da parede de forma aos utilizadores poderem observar através das câmeras a parede em questão.

Como esperado esta manobra será desenvolvida com duas variantes, uma para examinar o lado esquerdo, e outra para analizar o lado direito.

A segunda manobra *Center Tunnel*, como idealizada para o veículo se movimentar ao longo dos túneis, pois muitas das vezes as minas são compostas por todo um conjunto de túneis. A necessidade de chegar a um ponto da mina leva a que seja necessário uma manobra que não dê tanta importância ás paredes, como a primeira manobra, mas que apenas chegue ao destino evitando sempre os obstáculos.

A *Vertical Shaft*, foi uma manobra planeada para dar resposta aos poços existentes na mina. Visto ser um corte vertical na mina, estes poços permitem a visualização das várias camadas do solo. Como tal torna-se importante mapear e identificar toda a sua extensão pois encontrando uma determinada camada pose significar uma maior probabilidade de se encontrar uma certa matéria-prima. Além disso, os poços normalmente permitem o acesso a diferentes níveis de profundidade da mina, como tal também é necessário identificar todas as ramificações de túneis.

A quarta funcionalidade, *Cave Mapping* destina-se a ser utilizada quando o AUV encontra uma galeria. Uma galeria é uma cavidade com um enorme volume de espaço em comparação aos túneis e poços, como tal é difícil de mapear esta secção apenas com uma passagem pois os sensores do robô tem limite de alcance. Será então necessário estudar algoritmos usados para mapear grandes áreas.

Quinta e última manobra, será a *Slope Tunnel*. Inicialmente estavam apenas projetadas quatro manobras, no entanto decidiu-se adicionar uma quinta que visa resolver o problema de túneis inclinados. As manobras M1 e M2 dão resposta a este cenário, mas de forma pouco eficiente pois da forma que o UX-1 foi construído, necessitam que o robô esteja constantemente a se movimentar para linearmente e verticalmente de forma a executar um movimento com um grau de inclinação. Uma das funcionalidades do UX-1 é movimentar o seu pêndulo interno de forma a mexer no seu centro de gravidade. Isto é uma ferramenta muito útil para se utilizar nestes cenários pois alterando o centro de massa, o UX-1, passa a executar o seu movimento linear num ângulo semelhante ao pêndulo, permitindo acompanhar o declive do túnel de forma mais eficiente.

6.3 Missões

A informação das missões define essencialmente os *behaviors* necessários. Sendo as BT um objeto de estudo desta dissertação, os objetivos e dimensões das missões irão evoluir consoante as necessidades dos utilizadores. Na lista abaixo, encontram-se descrito as diferentes fases, aumentando em complexidade:

- Nível 1 - Autonomamente ser capaz de explorar e entender a mudança de cenário, como por exemplo percorrer um túnel, e em seguida descer um poço.

- Nível 2 - Ser capaz de explorar uma mina até encontrar um beco sem saída.

- Nível 3 - Capacidade de explorar uma mina por completo, sendo capaz de identificar e explorar todas as ramicações.

- Nível 4 - Ser possível definir objetivos de exploração, como tempo de missão ou profundidade.

A figura 6.3, exemplifica uma BT capaz de executar uma missão de nível 1. Assumindo que o veículo começa num túnel, a primeira condição vai ser verdadeira como tal irá executar a manobra M2. Após algum tempo, a condição de testar se é um poço irá retorna sucesso como tal passará a executar a manobra M3. Isto só funciona pois ao testar de novo a condição do túnel esta irá retornar falha e que por sua vez por ser um nó sequencial irá retornar falha de novo, obrigando o nó de *fallback* a executar o ramo correspondente ao poço.

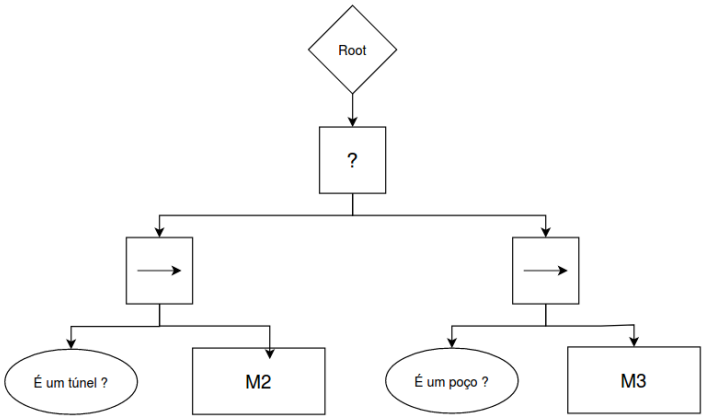


Figura 6.3: BT capaz de executar uma missão de nível 1.

A evolução do nível 1 para uma missão de nível 2 passa por apenas incorporar todos os casos possíveis de cenários. Apenas para completar os três tipos de cenários basta adicionar uma ramificação capaz de movimentar o robô numa galeria. Na figura 6.4, podemos observar essa transição.

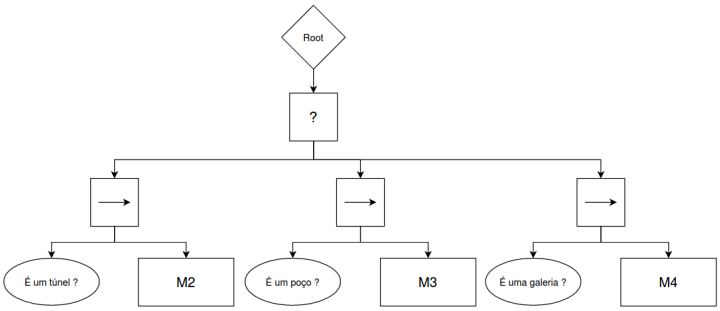


Figura 6.4: BT capaz de executar uma missão de nível 2.

Uma missão de nível 3, consiste em explorar uma mina por completo. Como tal não será suficiente apenas testar o cenário de atuação e ter uma manobra de exploração como nos níveis inferiores de missão. A este nível será necessário implementar condições que verifiquem o nível da bateria do robô, identificar interseções e guardar a sua posição, saber se ainda ou não existem secções por explorar,etc... A figura 6.5 apresenta a BT principal, para uma missão de nível 3.

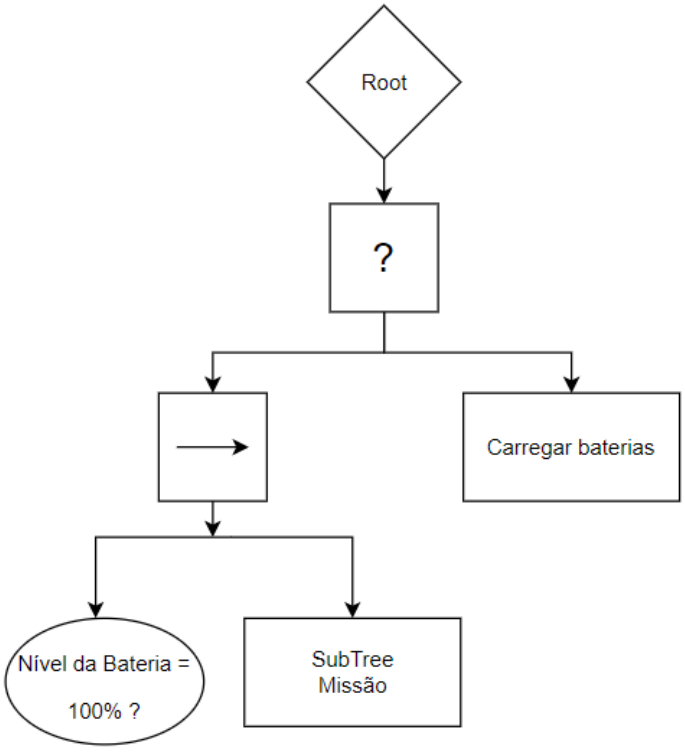


Figura 6.5: BT capaz de executar uma missão de nível 3.

Como observado, na figura acima, existem condições e procedimentos a ser efetuados antes do começo de uma missão. Um deles é por exemplo verificar a carga das baterias, e só dar inicio a missão após estarem totalmente carregadas. Outras adições poderiam ser fazer a verificação dos sensores e atuadores abordo.

Após o robô estar apto para dar inicio a missão, na figura 6.6 podemos observar o comportamento a ter na exploração, na *SubTree* "Missão".

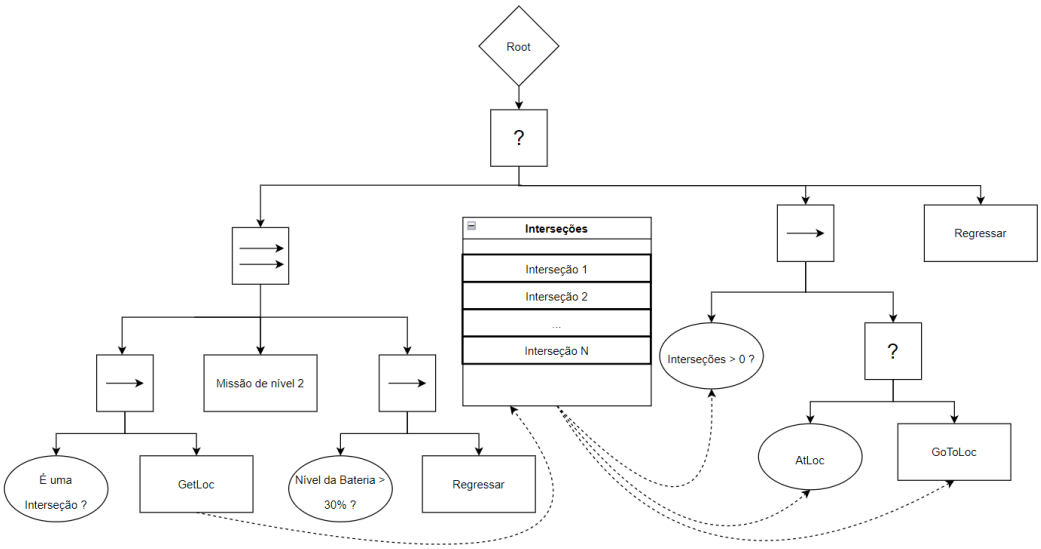


Figura 6.6: *SubTree* responsável pela exploração.

A nossa árvore secundária responsável pela exploração pode ser divida em 3 blocos. Em primeiro lugar temos o bloco que dá inicio a exploração. Aqui utilizando um nó paralelo, em simultâneo iremos guardar as interseções encontradas numa *Blackboard*, executar a exploração das cavidades incorporando a *SubTree* "Missão de nível 2"e claro verificar ao longo do tempo a carga das baterias.

O Segundo bloco existe para dar solução ao problema de quando o robô ainda tem bateria e se encontra num beco sem saída. Como tal é necessário verificar na *Blackboard* se existem interseções por explorar. A *Blackboard* permitirá ao robô se movimentar até a localização da interseção anteriormente guardada.

Caso não existam mais interseções a explorar, significa que explorarmos a mina na totalidade sendo necessário ativar o terceiro bloco, composto apenas pela manobra "Regressar". Esta manobra poderá ser um algoritmo que tenha a localização da entrada para guiar o robô ou então ser outra *Blackboard* onde está guardada a ordem de manobras utilizadas entre a entrada e a posição atual do robô. Por exemplo "M2, M3, M4, M2,

M2, M3". Sendo apenas necessário efetuar na ordem inversa para regressa a entrada da mina.

Capítulo 7

Implementação

Este capítulo descreve os desenvolvimentos realizados na resposta aos desafios e objetivos definidos.

O ambiente de simulação criado em Gazebo, pelo laboratório é detalhado em termos de funcionalidades, características e propriedades definidas. O ambiente de simulação serve de ferramenta para todos os desenvolvimentos subsequentes detalhados neste capítulo.

Como descrito no capítulo anterior, foram planeadas cinco manobras independentes como resposta a todos os cenários que se acha ser possível encontrar. Neste capítulo será desenvolvimento os algoritmos responsáveis por desempenhar as devidas tarefas, bem como toda a teoria que suporta a sua construção.

Por último, também se tentará implementar, todas as *Behaviour trees*, projetadas, começando pelo nível mais simples e partir para os mais avançados.

7.1 Cenário de Simulação

Os dois ambientes de simulação disponíveis, a o tanque do LSA e a mina Kaatalia permitiram testar e perceber o comportamento do UX-1 em diferentes contextos. O aspeto visual tem um papel fundamental na simulação, particularmente na robótica com recurso a sensores de visão, onde o mundo adquirido pelas câmaras pode pode ser suficientemente realista para testar algoritmos e aplicá-los com sucesso na realidade. Para gerar um cenário mais realista, o mesmo inclui texturas, efeitos de luz (externa e de subaquática), sombras, e outros objetos em redor do veículo.

Em primeiro lugar, o tanque do LSA, este foi o ambiente onde foram realizados os primeiros testes experimentais do UX-1. Utilizando o controlador manual do robô foi

possível entender os limites de estabilidade, velocidade, e as suas funcionalidades. Além disso, a medida que o AUV era testado era possível ver o funcionamento de todos os sensores abordo, tendo assim uma ideia de como os dados a serem enviados por cada um podiam ser utilizados para as diferentes manobras planeadas. Por ser um paralelepípedo, com um grande volume de água permitiu testar todos os algoritmos iniciais das manobras e perceber se os resultados obtidos eram os esperados e fazer as devidas alterações.

Em segundo lugar, temos a mina Kaatalia, que como referido foi feita através de uma *meshe* de pontos adquiridos pelo robô, quando foram feitos testes na mina verdadeira. Aqui, em relação ao tanque a dificuldade aumentou exponencialmente. As paredes deixaram de ser lisas, podendo levar a erros por parte dos SLS, o espaço de manobra ficou mais limitado, pois passamos de um tanque de água para um labirinto de túneis e poços, a mina tem um volume muito superior levando a que os testes se tornem mais longos e propícios a erros.

7.2 Manobras de exploração

As manobras de exploração desenvolvidas para a exploração do ambiente em questão. As manobras de exploração são as seguintes:

7.2.1 M1 - Wall Following

Através desta manobra pretende-se utilizar uma parede como ponto de referência para a navegação do UX-1. Semelhante ao DEPTHX, o UX-1 vai-se movimentar ao longo de uma parede mantendo uma distância fixa da mesma. Esta manobra é importante porque um dos objetivos do veículo é analisar matérias-primas e mapear no meio envolvente. Por isso, interessante existir uma manobra capaz de manter o veículo perto, não só como ponto de referência pois a água em algumas minas tem um elevado nível de opacidade, como também ser possível mapear através dos sensores ou mesmo observar através das câmeras existentes.

A M1 utiliza apenas os 4 lasers presentes na frente do UX-1 como forma de navegação. O algoritmo tem duas variações, dependendo do lado a que se encontra a parede, esquerda ou direita, sendo que apenas é necessário modificar o laser de referência. Através dos tópicos $/UX_1/laser/scan_x$ (onde x = 1,2,3,4) pode-se obter as distâncias em metros dos 160 feixes laser existentes em cada SLS. Devido a orientação dos SLS em relação ao UX-1 podemos assumir que os feixes que obtiverem menores valores são os que se encontram mais próximos ao veículo. A partir daí utilizamos uma função que obtém o valor mínimo

do vetor de 160 feixes. Os SLS 2 e 3 atuam á direita e esquerda do UX-1, enquanto os SLS 1 e 4 atuam em cima e em baixo respetivamente. Sabendo a distância a que o veículo se encontra da parede, podemos através do controlador PID, ajustar esse valor para distância pretendida, modificando a velocidade angular em *yaw* para aproximar ou se afastar.

Um controlador Proporcional-Integral-Derivativo (PID) é um mecanismo de *feedback* em *loop* para controlar um sistema. Como nome sugere, o algoritmo PID consiste em três coeficientes: proporcional , integral e derivativo que juntos permitem ao sistema definir a lei de controlo, figura 7.1.

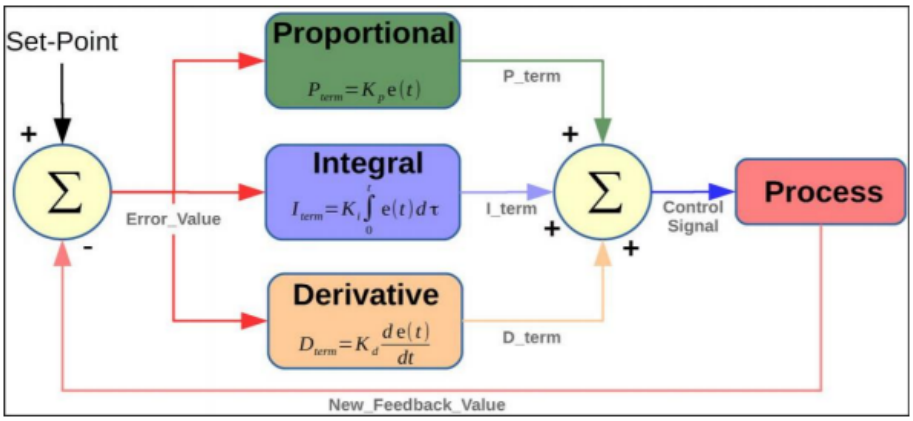


Figura 7.1: Controlador PID

A ideia deste algoritmo gira em torno da manipulação do erro. Como evidente o erro do sistema é obtido através da diferença entre o resultado obtido e o valor definido. O algoritmo utiliza a parte proporcional para corrigir o erro, a parte integrativa para corrigir a acumulação de erro e a parte derivativa para corrigir o erro presente em relação ao erro obtido na interação anterior. A componente derivativa é utilizada para diminuir o *overshoot* causado pelo P e I. Quando o erro é grande, o P e o I aumentam a resposta do sistema. Esta resposta do controlador faz com que o erro mude rapidamente, por sua vez, o D fica mais agressivo de forma a corrigir o efeito extra causado pelas restantes componentes.

As componentes P, I e D podem ser configuradas manualmente no caso do sistema ser previsível ou simuladas caso seja necessário prever a reação do sistema antes da implementação

O controlador PID foi construído com as seguintes constantes, equação 7.1. Estes parâmetros foram calculados através do método *Ziegler–Nichols* [31], e depois afinados

através de testes.

$$Kp = 2, kd = 0.12, Ki = 0.1875, dt = 0.01. \tag{7.1}$$

Ao longo da manobra a distância medida pelos SLS 2 ou 3 dependendo se a parede está à esquerda ou à direita, vai sendo utilizada como fator de referência para o controlador PID, como observado na figura 7.2, sendo que os restantes SLS's são utilizados para desvio de obstáculos. A velocidade linear do veículo é calculada através da equação 7.2 onde x varia consoante o SLS de referência, permitindo ao UX-1 uma velocidade constante quando se encontra acima da distância de referncia, mas ao mesmo tempo reduz-se velocidade exponencialmente consoante a proximidade a parede.

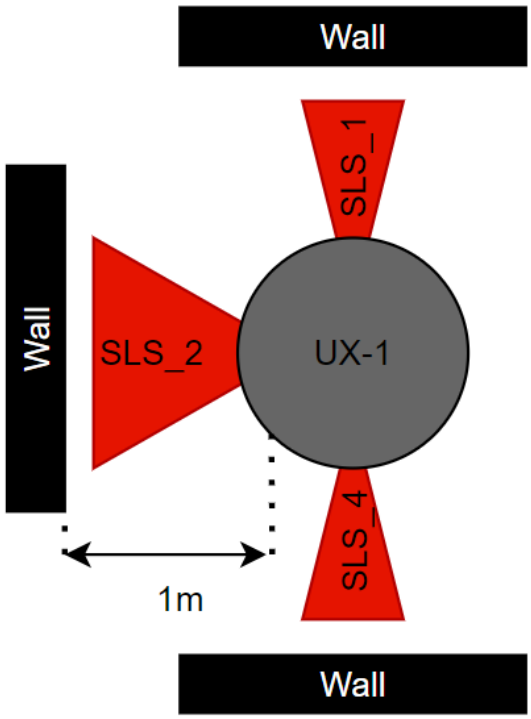


Figura 7.2: Manobra capaz de seguir uma parede constantemente.

$$Vlinear_x = \log_{10}(minimo_x - 0.1) + 1.2, \tag{7.2}$$

7.2.2 M2 - Center Tunnel

A necessidade do UX-1 se deslocar ao longo dos túneis da mina, levou a que fosse necessário desenvolver o manobra capaz de navegar o veículo centrado ao longo dos mesmos. A M2 foi pensada para ser utilizada para mapear túneis, mas também como forma de descolamento onde apenas interessa que o veículo se movimente entre secções da mina o mais rápido possível.

Tendo em conta esta situação, para realizar esta manobra de exploração foi preciso utilizar de novo os 4 SLS em conjunto com M3-*Multibeam* que existe na parte frontal do veículo. Para resolver o problema da centralização, apenas foi necessário que o UX-1 obtivesse a distância mínima de cada SLS para depois calcular a posição central do túnel através da diferença entre os SLS's opostos ($X_2 e X_3$ para posicionamento lateral, $X_1 e X_4$ para um posicionamento vertical). A partir daí, o UX-1 pode se movimentar centrado ao longo do percurso, figura 7.3.

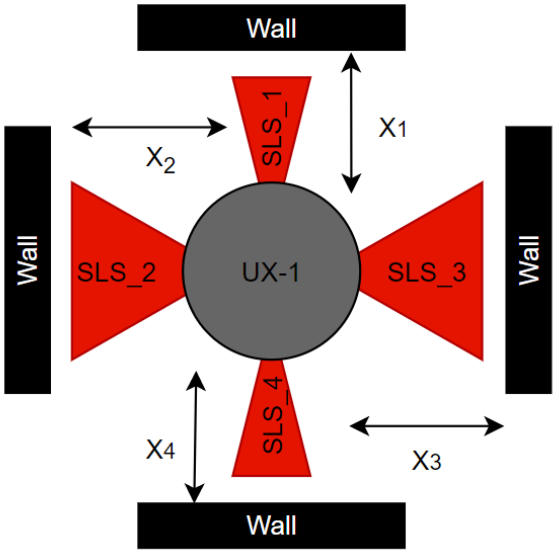


Figura 7.3: Centralização do UX-1, utilizando todos os SLS.

O M3-*Multibeam* é utilizado para detecção de obstáculos. Este sensor produz 240 feixes verticais, com uma abertura até 180º, capazes detetar obstáculos na frente do robô, figura 7.4.

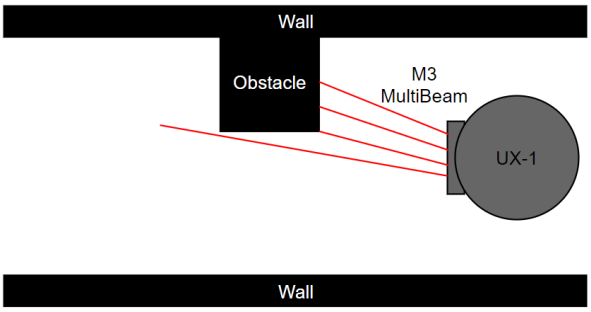


Figura 7.4: Deteção de obstáculos utilizando o M3 *Multibeam*.

Os dados do sensor são recebidos através do tópico $/m3Profile$, no formato $sensor\_msgs$ :: $PointCloud2$. Para se poder utilizar os dados sensoriais é necessário converter a mensagem recebida para um formato de *Pointcloud*. Devido à biblioteca Point Cloud Library (PCL) [32] é possível através das funções da Figura 7.5, converter a mensagem recebida numa *Pointcloud*.

```
//This is the datatype that is coming in to our subscriber
pcl::PCLPointCloud2 m3_pcl;
//These two PCL functions convert the pcl_pc input into usable PointCloud
pcl_conversions::toPCL(msg,m3_pcl);
pcl::fromPCLPointCloud2(m3_pcl, m3_cloudxyzi);
```

Figura 7.5: Funções que convertem mensagens do tipo sensor_msgs::PointCloud2 em Pointcloud.

Através da *Pointcloud* são seleccionados os feixes centrais pois os SLS's 1 e 4 já cobrem a parte inferior e superior. No final utilizando as coordenadas x, y e z de cada ponto é calculado a distância a que se encontra o obstáculo através da equação 7.3, que calcula a distância cartesiana de um vetor num espaço 3D.

$$d_{AB} = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2 + (z_B - z_A)^2} \tag{7.3}$$

Sabendo a posição do obstáculo, basta ajustar a posição do UX-1 utilizando os SLS, figura 7.6.

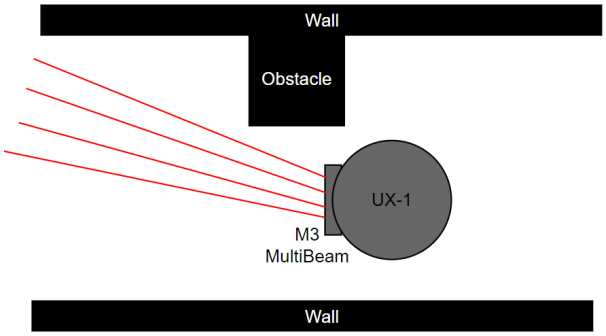


Figura 7.6: Ajuste da posição em relação ao obstáculo.

Esta manobra tem como condição, funcionar enquanto os SLS's mantêm uma distância mínima superior a 30cm das paredes ou quando o M3-*Multibeam* identifica obstáculos a menos de 1m e não seja possível contornar.

7.2.3 M3 - Vertical Shaft

Com esta manobra pretendemos que o UX-1 seja capaz de descer ou subir um poço da mina autonomamente, mapeando o poço ao longo da manobra com o objetivo de descobrir novos túneis ou galerias.

Esta manobra faz uso dos SLS verticais (SLS1 caso a manobra seja ascendente e SLS4 caso seja descendente). De forma a ser possível mapear e executar a manobra no centro do poço será utilizado o *Mechanical Scanning Imaging Sonar* (MSIS) - Micron DST de forma a obter 360º *scan* completo, existente no topo do UX-1. O algoritmo inicia com um *scan* completo para mapear o perímetro em torno do UX-1. O*scan* demora em simulação aproximadamente 20 segundos, Figura 7.7, com um alcance de 15m em cada feixe.

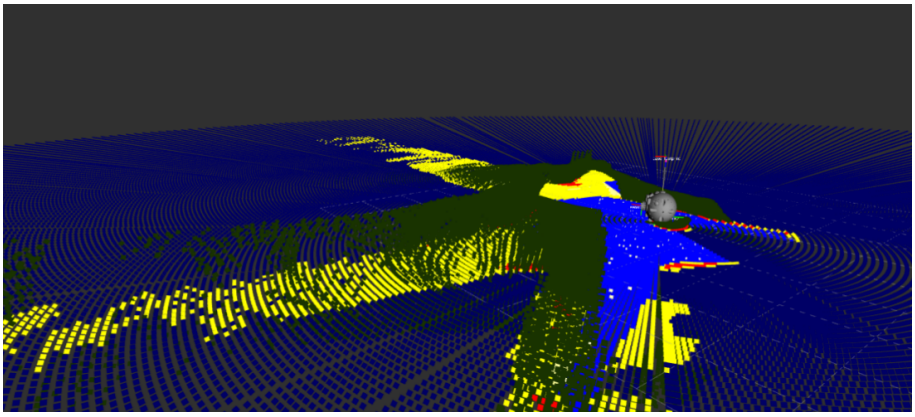


Figura 7.7: Scan efetuado pelo MSIS do UX-1, em simulação.

Os dados fornecidos pelo MSIS, são obtidos pelo tópico $/tritech\_micron/scan$, que tal como o M3-*Multibeam* é preciso converter para Pointcloud pois são do tipo $sensor\_msgs :: PointCloud2$.

Para eliminar possíveis falsas medidas apenas é definido um obstáculo numa determinada direção caso sejam detetados 2 pontos, consecutivos com intensidades iguais a 255, caso não exista nenhum obstáculo detetado num dos feixes é lhe atribuído a distância máxima 15m de forma a ser possível calcular um perímetro em torno do veículo.

O perímetro em torno do veículo é calculado ao fim do *scan* completo (rotação de 360 graus, composta por 240 feixes), onde através da adição vectorial das componentes X e Y de cada feixe a dividir pelo número total de feixes, 240. As coordenadas do centro $Center(X_{Center}, Y_{Center})$ para onde o veículo se deve dirigir de forma a estar a mesma distância de todos os feixes são calculadas através das seguintes equações, 7.4 e 7.5.

$$X_{Center} = \frac{X_1 + X_2 + X_3 + X_4 + ... + X_n + ... + X_{240}}{240} \tag{7.4}$$

$$Y_{Center} = \frac{Y_1 + Y_2 + Y_3 + Y_4 + ... + Y_n + ... + Y_{240}}{240} \tag{7.5}$$

Em seguida o veículo determina orientação em relação ao centro do perímetro criado e executa uma rotação em *yaw* até se encontrar de frente para o centro, seguido de um movimento linear até atingir o centro do perímetro. Como última etapa, após a deslocação do UX-1 da sua posição até ao centro do perímetro criado, desce ou sobe 4m (dependendo se a manobra é ascendente ou descendente), figura 7.8.

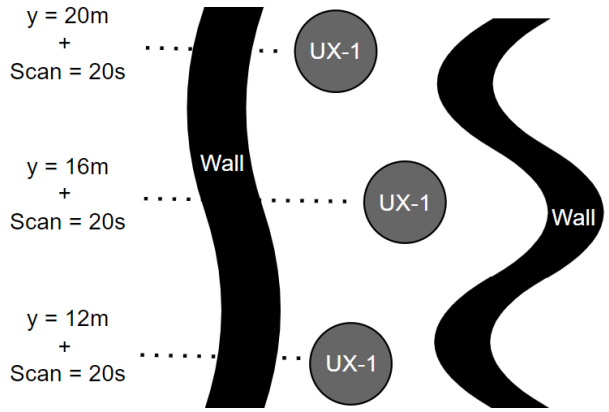


Figura 7.8: Mapeamento do poço durante 20s, seguido de uma descida de 4metros.

O UX-1 irá repetir todo o processo acima descrito (*scan*, movimento horizontal, movimento vertical) até atingir o fundo ou o topo do poço onde se encontra.

7.2.4 M4 - Cave Mapping

A existência de galerias e secções de grandes volumes nas minas impede que a passagem do veículo seja insuficiente para mapear todo o espaço. Com esta manobra, o veículo através de um movimento em Zig-Zag consegue mapear de forma eficiente estas secções volumosas. Para o veículo detetar obstáculos, ele faz uso dos 4 SLS's em conjunto com o M3-*Multibeam*.

A manobra inicia o seu processo com a aproximação do UX-1 a uma parede até se encontrar a 1m de distância dela, semelhante ao que é feito pela manobra M1. Em seguida, segue a parede até o *Multibeam* detetar à sua frente um obstáculo a menos de 1m. Para não desperdiçar tempo, o UX-1 através do SLS aposto à parede de referência determina se existe algum obstáculo a menos de 1,5m, caso seja verdade, não justifica continuar a manobra, caso contrário indica que existe ainda espaço por mapear na galeria e prossegue com a manobra. A etapa seguinte é a rotação em *yaw* do veículo até a sua frente ter 2m livres, com isto, é possível mapear a maior área possível, independentemente da orientação das paredes. No passo seguinte, o robô percorre 1m na direção livre, efetuando uma nova rotação em *yaw* até se encontrar virado na direção oposta a que percorreu a seguir a parede. A orientação da parede de referência é calculada através da sua posição inicial e final em relação a parede de referência. Em seguida, inicia de novo um movimento linear com a direção oposta à parede de referência até detetar de novo

um obstáculo. O veículo irá executar todo este processo de Zig-Zag até o SLS oposto á parede obter leituras inferiores a 1,5m. Tudo o processo pode ser observado na figura 7.9.

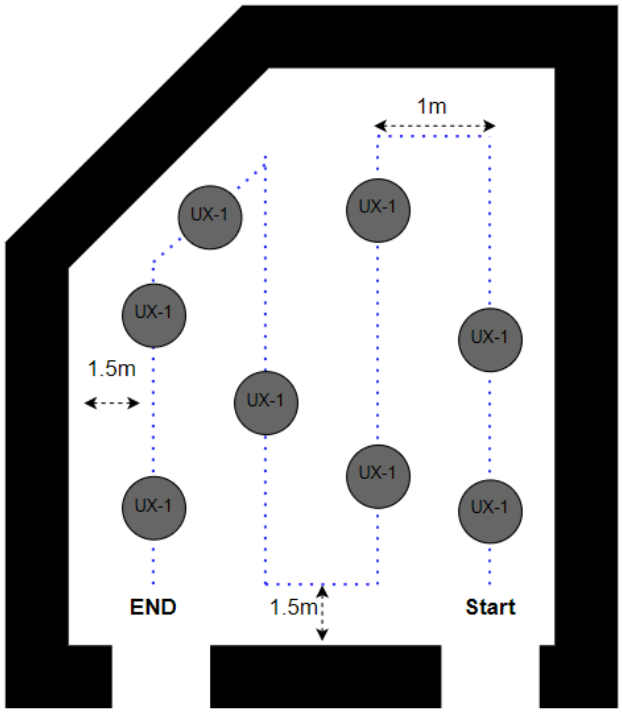


Figura 7.9: Mapeamento completo de uma galeria.

Com esta manobra torna-se possível mapear uma galeria, cobrindo a maior área possível, sem se importar com a sua dimensão ou a orientação das paredes.

7.2.5 M5 - Slope Tunnel

Esta manobra foi a última a ser desenvolvida, pois como descrito anteriormente só estavam previstas as 4 manobras acima descritas e respetivas variantes. Como explicado decidiu-se implementar esta manobra para dar resposta aos túneis com uma certa inclinação. Esta a manobra pode ser comparada a M2 apenas como a particularidade de o veículo se movimentar como uma inclinação. Tendo essa informação esta manobra será muito semelhante á M2, sendo apenas necessário publicar para nó responsável pelo pêndulo do UX-1 o valor correto da inclinação.

Para calcular a inclinação podemos recorrer a seguinte formula:

$$slope = \arctan \frac{Y_2 - Y_1}{X_2 - X_1}, \tag{7.6}$$

Com a equação 7.6, podemos calcular então a inclinação do túnel. Para tal apenas precisamos de dois pontos de referência para o efeito. A altura do túnel pode ser calculada através dos SLS1 e SLS3 e sabendo a distância percorrida entre as duas posições, podemos obter o ângulo, figura 7.10.

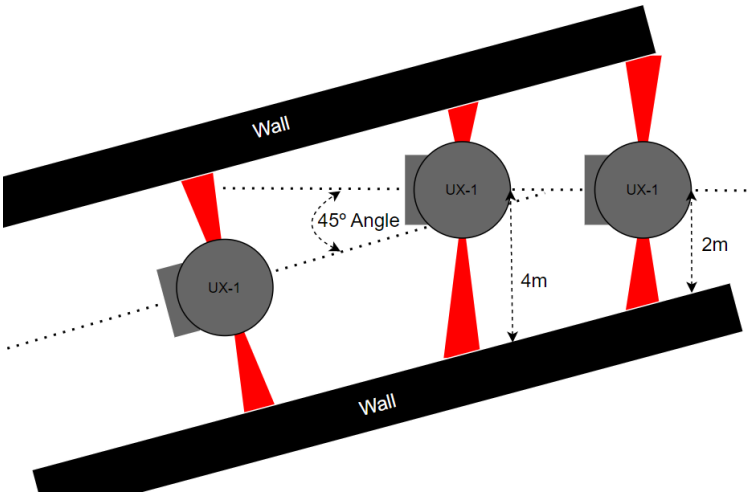


Figura 7.10: Ajuste da inclinação do UX-1 em relação ao declive do túnel.

A figura acima exemplifica um caso prático onde o UX-1 inclina devido á distância entre o fundo e o veículo ter aumentado 2m. Como tal utilizando a equação 7.6 podemos obter o valor de 45 graus de inclinação.

Após o cálculo basta enviar o valor utilizando para o tópico $'/UX_1/pendulum'$ que é responsável por alterar o ângulo do pêndulo interno do UX-1.

Não sendo necessário alterar constantemente alterar a inclinação do UX-1, o algoritmo implementado apenas irá modificar a inclinação caso o declive do túnel tenha uma diferença de 5°.

7.3 Behviour tree

A implementação de uma BT como controlador, só foi possível utilizando a a biblioteca **BehaviorTree.CPP**. Todo este processo demorou vários meses de trabalho pois no final de 2021 quando este projeto começou a ser construído, não existia quase nenhum

suporte por parte da comunidade, bem como do criador Davide Faconti. Apesar de já apresentar soluções para funcionar em conjunto com o ROS, a biblioteca era muito limitada em termos de funcionalidades, falta de exemplos de como desenvolver, construir e compilar os projetos desenvolvidos, mudança de nomenclatura entre versões. A falta de exemplos práticos de como implementar uma simples BT funcional em ROS, levou a muitos problemas, pois muitas vezes os erros eram provocados, por falta de dependências entre os algoritmos.

A existência de múltiplos nós e pacotes devido ao simulador do UX-1, e das manobras construídas levou ao problema de como interligar tudo. A solução para o problema passou por utilizar uma funcionalidade do ROS denominada *actionlib*. Esta funcionalidade permite dois ou mais nós comunicarem entre eles tendo uma relação de servidor-cliente.

Na figura 7.11, podemos observar como funciona este processo. Existe um cliente que precisa de um determinado serviço, este serviço é requisitado a um servidor. A medida que o tempo passa o servidor vai transmitindo o *feedback* ao cliente. No final o servidor envia o resultado ao cliente, na forma de sucesso ou falha.

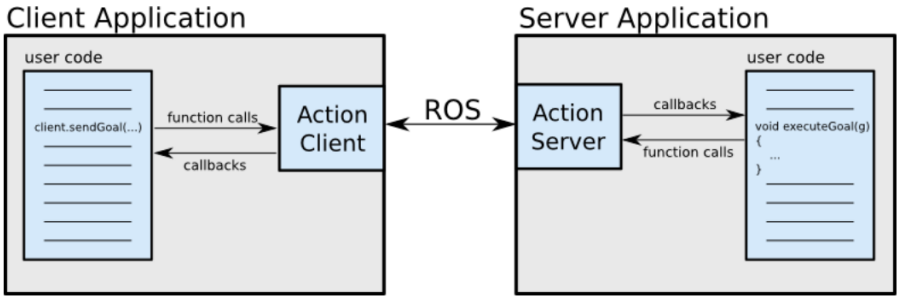


Figura 7.11: Interação de cliente-servidor no ROS.

Este sistema proporcionado pela **ActionLib**, permite por exemplo termos o nosso controlador de missão a requisitar explorar a mina durante uma hora, em seguida a BT aceita o pedido e executa os nós necessários. A BT estará constantemente a informar, do tempo de funcionamento da missão, as manobras que se encontram a funcionar. Caso aconteça algum imprevisto ou o tempo de missão acabe, será retornado o resultado para o controlador.

Para começar implementar, foi primeiro necessário adicionar as tais dependências da biblioteca para que se pudesse utilizar as BT. Para tal é necessário introduzir as dependências demonstradas nas figura 7.12, seguindo da dependência para utilizar a ActionLib, figura 7.13.

```
#include <behaviortree_cpp_v3/action_node.h>
#include <behaviortree_cpp_v3/bt_factory.h>
```

Figura 7.12: Dependências da biblioteca BehaviorTree.CPP.

```
13  #include <ros/ros.h>
14  #include <actionlib/server/simple_action_server.h>
```

Figura 7.13: Dependências do protocolo de comunicação ActionLib.

Após a configuração dos nós a utilizar, podemos então idealizar como construir a nossa árvore. Tendo por base os níveis definidos para missão de exploração, iremos começar por construir uma árvore que permita executar a BT exemplificada na figura 6.3.

Para tal precisamos então de implementar uma árvore com um *fallback node*, dois *sequence node*, e claro as nossas manobras e condições.

Em primeiro lugar iremos definir as condições. A forma abordada de definir um túnel é utilizar os 4 SLS. Sabendo a distância do robô as quatro paredes de um túnel podemos dizer que estamos num túnel quando todos os SLS medem valores de 3metros ou menos. Na figura 7.14, podemos observar como implementar essa condição.

```
83    public:
84        explicit BTAction(std::string name) :
85            as_(nh_, name, boost::bind(&BTAction::execute_callback, this, _1), false),
86            action_name_(name)
87        {
88            // start the action server (action in sense of Actionlib not BT action)
89            as_.start();
90            ROS_INFO("Condition Server Started");
91        }
92
93        ~BTAction(void)
94        { }
95        void execute_callback(const behavior_tree_core::BTGoalConstPtr &goal)
96        {
97
98            if ((SLS_1 <= 3) && (SLS_2 <= 3) && ((SLS_3 <= 3) && (SLS_4 <= 3))){
99                ROS_INFO("It's a Tunnel");
00                set_status(SUCCESS);
01            }
02            else {
03                ROS_INFO("It's something else.");
04                set_status(FAILURE);
05            }
06        }
```

Figura 7.14: Condição que testa se estamos num túnel.

Função que retorna o estado(Sucesso ou falha) da condição para a árvore, figura 7.15:

```
108     void set_status(int status)
109     {
110         // Set The feedback and result of BT.action
111         feedback_.status = status;
112         result_.status = feedback_.status;
113         // publish the feedback
114         as_.publishFeedback(feedback_);
115         // setSucceeded means that it has finished the action (it has returned SUCCESS or FAILURE).
116         as_.setSucceeded(result_);
117 
118         switch (status)  // Print for convenience
119         {
120         case SUCCESS:
121             ROS_INFO("Condition %s Succeeded", ros::this_node::getName().c_str() );
122             break;
123         case FAILURE:
124             ROS_INFO("Condition %s Failed", ros::this_node::getName().c_str() );
125             break;
126         default:
127             break;
128         }
129     }
130 };
```

Figura 7.15: Retorno de Sucesso ou falha da condição para a árvore.

Para detetarmos se estamos na presença de um poço é o utilizar o SLS 3, que é responsável por medir a distância do robô ao chão. Existe no entanto outra possibilidade que passa por modificar o pêndulo no ângulo de 90°, virando assim o Sonar *Multibeam* em direção ao chão. Ao utilizarmos o SLS podemos definir que encontramos um poço se ele obter valores superiores a 4 metros. Caso optamos pelo segundo método podemos definir um poço quando o Sonar obtém valores superiores a 5metros.

Agora que já temos as nossas condições implementadas basta então criar a nossa árvore no ficheiro **tree.cpp**. Para tal iremos a definir como exemplificado na figura 7.16.

```
BT::ROSCondition* ItsTunnel = new BT::ROSCondition("ItsTunnel");
BT::ROSCondition* ItsShaft = new BT::ROSCondition("ItsShaft");
BT::ROSAction* M2 = new BT::ROSAction("M2");
BT::ROSAction* M3 = new BT::ROSAction("M3");

BT::FallbackNode* Explore = new BT::FallbackNode("Explore");
BT::SequenceNode* Tunnel = new BT::SequenceNode("Tunnel");
BT::SequenceNode* Shaft = new BT::SequenceNode("Shaft");

Explore->AddChild(Tunnel);
Explore->AddChild(Shaft);
Tunnel->AddChild(ItsTunnel);
Tunnel->AddChild(M2);
Shaft->AddChild(ItsShaft);
Shaft->AddChild(M3);
```

Figura 7.16: Definição de uma BT para executar uma missão de nível 1.

Como podemos observar para implementar a BT, basta primeiro definir as condições *ItsTunnel* e *ItsShaft*, onde determinados se estamos num túnel ou num poço. Seguidas das ações que no nosso caso são as manobras M2 e M3. O próximo passado é definir a quantidade e o tipo de nós. Terminando com a relação entre os diferentes nós, começando pelos mais importantes e finalizando com as folhas mais simples.

Em resumo para implementar uma missão de nível 1 é necessário os seguintes passos:

- **Arquitetura:** Construção da arquitetura gráfica da árvore.
- **Condições:**Construção de ficheiros independentes para cada condição existente, exemplo ItsTunnel.cpp e ItsShaft.cpp.
- **Acções:** Semelhante as condições, é necessário implementar nas manobras um serviço de comunicação baseado em ActionLib, de forma a ser possível retorna o estado da manobra a árvore, exemplo M2.cpp e M3.cpp.
- **Árvore**: Construção do ficheiro tree.cpp, que contém a árvore da missão, definindo primeiro as ações e condições, em seguida o tipo de nós(Sequence, Fallback, Parallel ou Decorator), tendo por último definido a hierarquia entre os diferentes nós.

Para implementar a missão de nível 2, que adiciona a exploração de galerias, bastou adicionar mais um nó sequencial responsável pelas galerias. Para tal foi construída a condição *ItsGallery*, que identifica uma galeria quando o Sonar *Multibeam* obtém valores superiores a 10metros e um dos SLS laterais obtém distâncias superiores a 4 metros, figura 7.17.

```
BT::ROSCondition* ItsTunnel = new BT::ROSCondition("ItsTunnel");
BT::ROSCondition* ItsShaft = new BT::ROSCondition("ItsShaft");
BT::ROSCondition* ItsShaft = new BT::ROSCondition("ItsGallery");
BT::ROSAction* M2 = new BT::ROSAction("M2");
BT::ROSAction* M3 = new BT::ROSAction("M3");
BT::ROSAction* M4 = new BT::ROSAction("M4");

BT::FallbackNode* Explore = new BT::FallbackNode("Explore");
BT::SequenceNode* Tunnel = new BT::SequenceNode("Tunnel");
BT::SequenceNode* Shaft = new BT::SequenceNode("Shaft");
BT::SequenceNode* Gallery = new BT::SequenceNode("Gallery");

Explore->AddChild(Tunnel);
Explore->AddChild(Shaft);
Explore->AddChild(Gallery);
Tunnel->AddChild(ItsTunnel);
Tunnel->AddChild(M2);
Shaft->AddChild(ItsShaft);
Shaft->AddChild(M3);
Gallery->AddChild(ItsGallery);
Gallery->AddChild(M4);
```

Figura 7.17: Definição de uma BT para executar uma missão de nível 2.

Capítulo 8

Resultados

Este capítulo apresenta os resultados alcançados nas diferentes frentes abordadas desta dissertação. Detalha a solução de simulação desenvolvida, focando aspetos mais relevantes de conceção e funcionalidade. Foram efetuados testes às cinco manobras desenvolvidas nos dois ambientes de simulação disponíveis, o tanque do laboratório e a mina Kaatiala.

O processo desenvolvimento por detrás de cada manobra de exploração, envolveu a programação de algoritmos teoricamente idealizados para o funcionamento individual de cada uma das manobras. Em seguida, procedeu-se aos teste dos algoritmos programados de forma a ser possível identificar possíveis erros e adicionar melhorias. Após esse processo foram realizados alguns testes de longa duração, com o propósito de identificar casos únicos e específicos onde as manobras não funcionassem como esperado.

O capítulo é finalizado com os resultados das *beehaviour trees*, como controlo de missão, apresentando que objetivos foram alcançados bem como os que não foram.

8.1 Manobras de movimento

8.1.1 M1

Sendo o objetivo o movimento linear tendo por referência uma parede, o grande objetivo foi tentar afinar o controlador PID de forma à distância de referência ser o mais próximo da realidade. A existência de túneis e cavidades com dimensões reduzidas, levou a ser definido no algoritmo a distância de referência igual a 0,8m. Na Figura 8.1, é possível observar um gráfico que mostra o UX-1 a seguir uma parede no interior da mina.

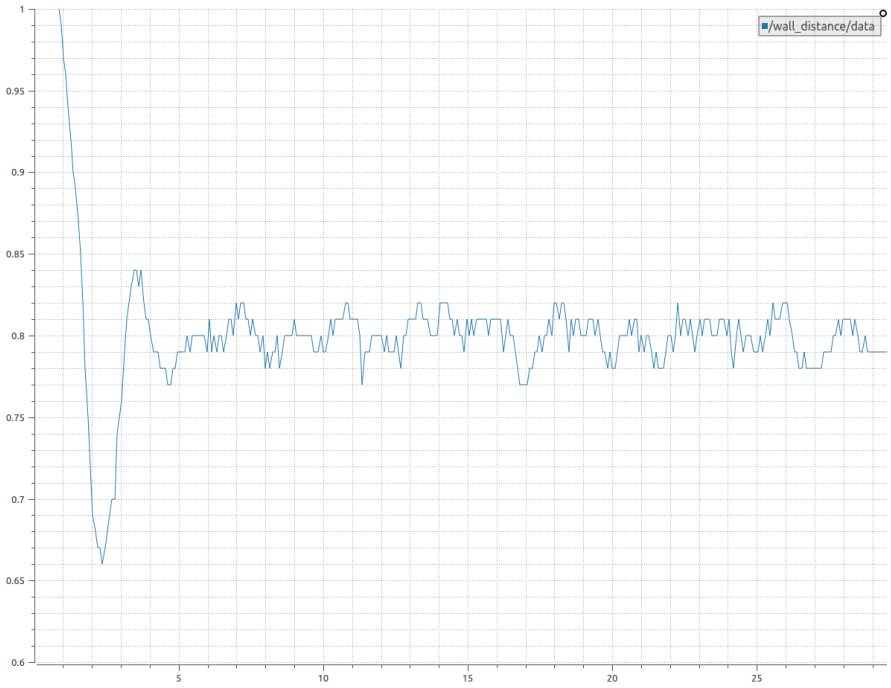


Figura 8.1: Distância entre o UX-1 e a parede da mina, ao longo da manobra.

Através deste gráfico pode-se concluir que o controlador PID demora 4,5s a estabilizar o UX-1 nos 0,8m, mas que depois mantém essa distância com um erro de ±5 cm. Em termos de simulação, demonstrou valores consistentes, mas devido ao grande período de estabilização levou a que em certos casos, por exemplo cantos e incidências, onde as medidas do SLS oscilam, a perda do controlo do veículo. No futuro é importante afinar melhor o controlador PID, de forma a baixar os tempos de estabilização, permitindo um maior controlo da manobra.

8.1.2 M2

A segunda manobra desenvolvida neste trabalho teve como objetivo centralizar o UX-1 num túnel à medida que este se movimenta. Para efeito foi necessário fazer uso dos quatro SLS's presentes em conjunto com o M3-*Multibeam*. Analisando a Figura 8.2, é possível observar o comportamento do robô ao longo da manobra. Inicialmente o veículo foi colocado no túnel numa posição aleatória. Em seguida, após o inicio da simulação o UX-1 À medida que se deslocava em frente, foi ajustando a sua posição horizontal (SLS 2 e 3) e a sua posição vertical (SLS 1 e 4). Ao longo do percurso o veículo foi mantendo

a mesma distância entre os pares de SLS's independentemente das dimensões do túnel.

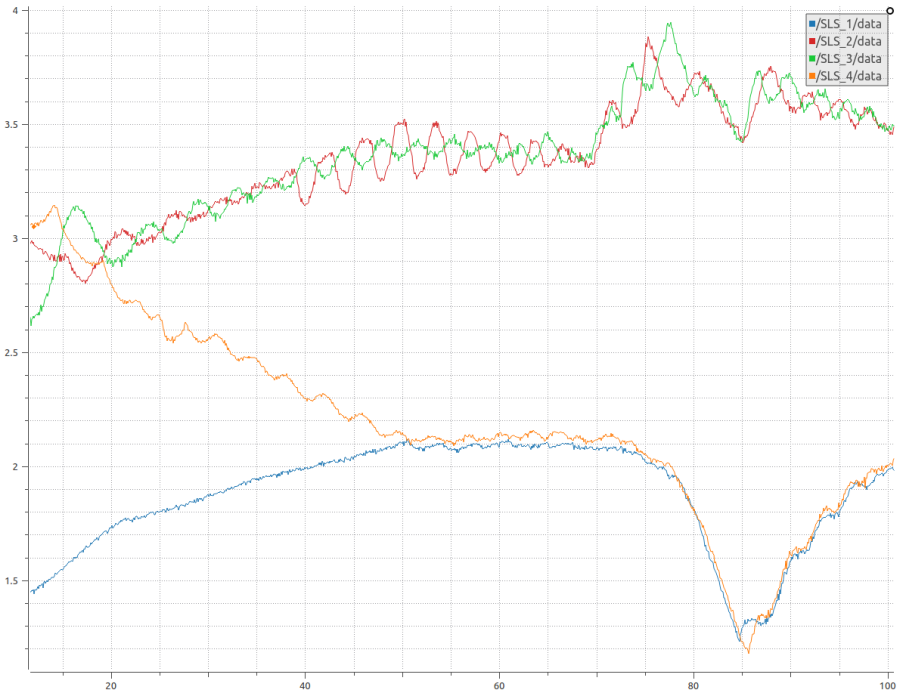


Figura 8.2: Posicionamento do UX-1 no centro, com recurso aos 4 SLS's.

Os resultados obtidos foram satisfatórios, no entanto, apesar da configuração dos propulsores limitar o movimento em algumas direções o robô demorou 50 segundos no inicio da simulação para corrigir a sua posição inicial. No futuro é importante afinar este tipo de controlo de forma a tornar o UX-1 mais reativo.

8.1.3 M3

Esta manobra foi desenvolvida para a exploração vertical de poços. Ao longo da manobra o robô tem como objectivo percorrer o poço na vertical à medida que ia fazendo um *scan* em busca de novas cavidades, túneis ou galerias. Semelhante à manobra M2, o veículo foi colocado numa posição aleatória, Figura 8.3, em seguida efectuou um *scan* de 360^{o} de forma a poder saber o centro do poço. Através da Figura 8.4, no Rviz é possível observar a *PointCloud* obtida pelo (MSIS) - Micron DST.

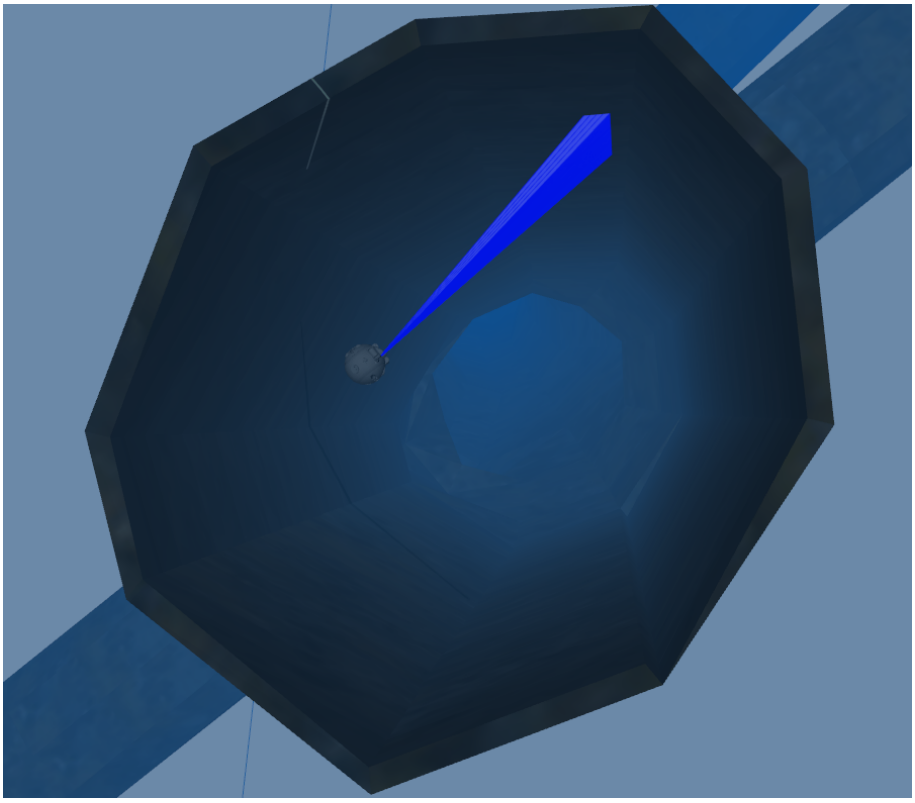


Figura 8.3: Posicionamento inicial do UX-1 num túnel da mina kaatiala.

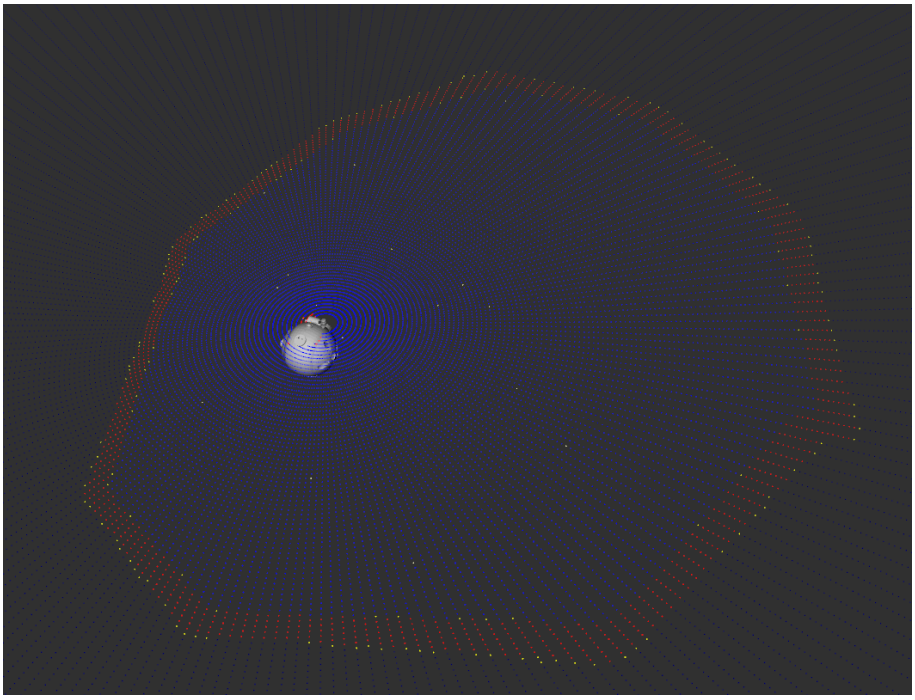


Figura 8.4: Pointcloud obtida pelo Scan do MSIS em simulação, no RVIZ.

Os gráficos das figuras 8.5 e 8.6, demonstra em simulação, a posição do robô ao longo de uma manobra descendente num poço da mina. Através do gráfico podemos confirmar que ele inicialmente mantém a sua posição durante 20 segundos (tempo necessário para completar um *scan* completo), em seguida apenas varia a sua posição em X e Y para puder-se movimentar para o centro calculado. Como última etapa, inicia um movimento descende, ao longo de Z, até descer 4 metros. A manobra após ter completado todas as etapas volta a executar todo o processo desde o inicio, até atingir o fundo do poço.

Figura 8.5: Gráfico 2D que demonstra ao longo do tempo, a posição do robô nas três componentes X, Y, Z.

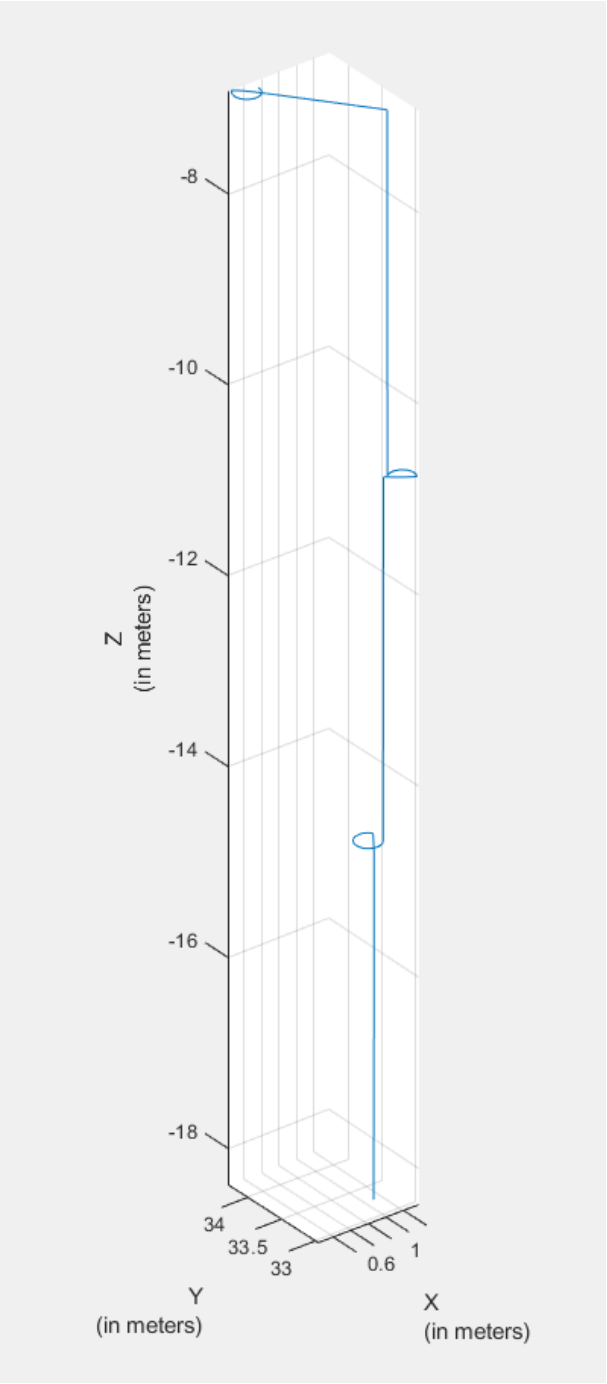


Figura 8.6: Gráfico 3D que demonstra ao longo do tempo, a posição do robô.

Esta manobra foi uma das mais difíceis de completar pois foi preciso muito trabalho para analisar a *Pointcloud*, obtida pelo sonar. Como explicado no capítulo anterior foi

necessário no algoritmo introduzir algumas seguranças como leitura de duas intensidades vermelhas, para evitar os falsos obstáculos. Em simulação um *scan* completo demora 20 segundos, mas na realidade esse tempo baixa para um terço. Quando for possível ter esse tempo de *scan* em simulação, a manobra será mais efectiva e diminuirá o tempo em que o UX-1 se encontra estático.

8.1.4 M4

A última manobra desenvolvida permite ao UX-1 ser capaz de mapear as grandes galerias e cavidades das minas onde com uma passagem não é possível ter as dimensões das mesmas. Nas Figuras 8.7 e 8.8, mostram o percurso efectuado pelo UX-1, em simulação no tanque e na mina Kaatiala respectivamente.

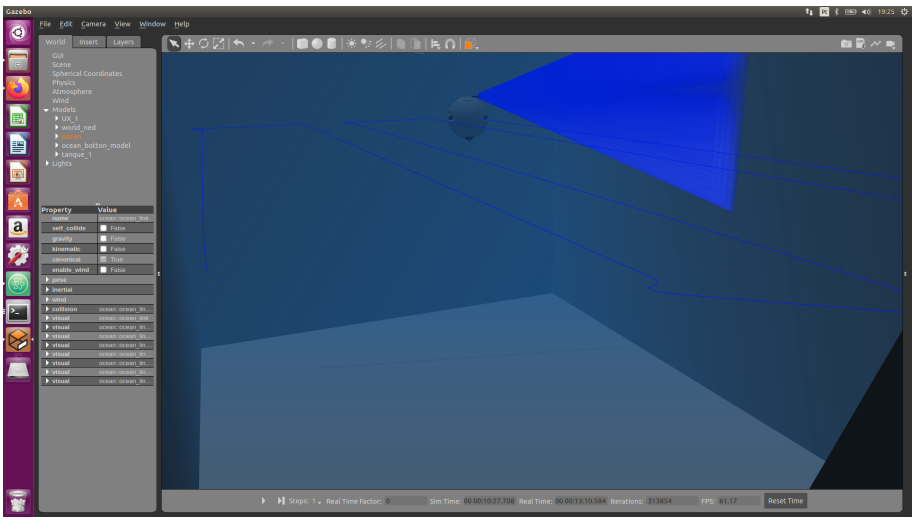


Figura 8.7: Trajetória efetuada para mapear o tanque do laboratório.

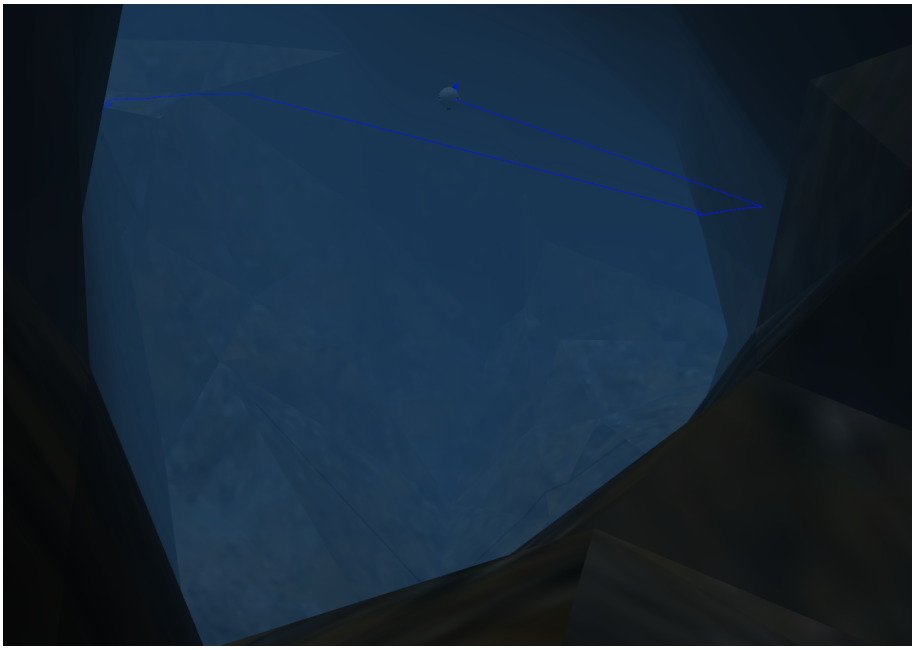


Figura 8.8: Trajectória efectuada para mapear a mina Kaatiala.

Como se consegue verificar nas Figuras acima, o veículo desempenhou uma trajectória em Zig-Zag, para cobrir a maior área possível. Os resultados foram óptimos quando se simulou o tanque do laboratório, mas pioraram quando utilizados na mina. Esta diminuição, deveu-se à existência de cavidades e interacções na galeria, sendo necessário melhorar a aproximação do veículo a este tipos de situações. Uma das ideias, é talvez adicionar o sonar rotativo para melhorar o padrão da trajectória, mas também dar possibilidade de explorar as galerias não só em área mas como em volume.

8.1.5 M5

Apesar desta manobra ter sido implementada e testada, os resultados não foram os melhores. Como indicado no capítulo anterior, esta manobra executa os mesmos procedimentos que a manobra M2, modificando apenas a inclinação do robô de forma acompanhar o declive do túnel. Por serem quase idênticas apenas foi necessário modificar algumas partes do algoritmo. Em termos de algoritmo a manobra executou exatamente para aquilo que foi programada, no entanto em termos de resultados não atingiu o objetivo pretendido. O que se verificou foi o movimento do robô corretamente pelos diferentes túneis, mas quando a inclinação passava os 30°, o veículo perdia o controlo sendo impossível continuar a manobra. Este problema deve-se ao centro de gravidade do robô, como o

pêndulo modifica a localização do mesmo, a medida que se aumenta o ângulo afastando-o da posição inicial, cada vez fica mais instável e difícil do controlar. Teoricamente esta manobra funciona sem nenhum problema, no entanto até se resolver o problema, não é prático utiliza-la.

8.2 Controlo de missão

A utilização das BT como controlador de missão provou por ser uma boa experiência. Pois ao contrário de outros algoritmos que definem os critérios de implementação, as BT permitem ao utilizador construir as diversas funcionalidades e tarefas necessários para o projeto, individualmente sendo que depois podem ser adicionadas ás árvores tendo por detrás um estrutura e respeitando os protocolos de comunicação.Os resultados obtidos podem ser analisados em duas fases.

Em primeiro lugar temos de analisar os resultados obtidos pelas missões de baixo nível(nível 1 e 2) e missões de alto nível(nível 3 e 4).

8.2.1 Low Level Mission

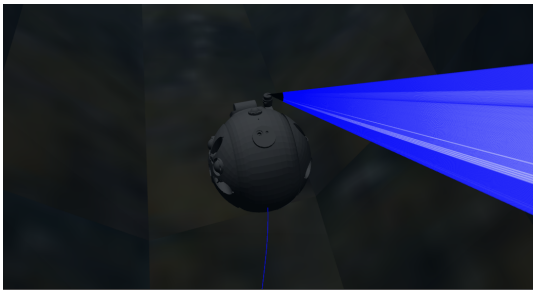
As missões de baixo nível obtiveram resultados positivos sendo que o objetivo de usar as BT para executar as missões foi alcançado. Para efeitos de simulação, o robô foi posicionado em cenários específicos da mina de forma a tirar o maior proveito.

8.2.1.1 Missão de Nível 1

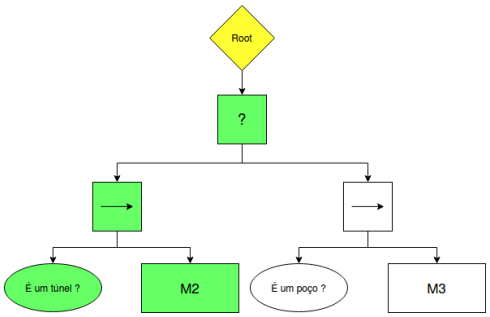
Para executar uma missão de nível 1, o UX-1 foi então colocado num túnel onde passado alguns metros encontra-se uma interseção com um poço.

As figuras 8.9(a), 8.12 e 8.11(a), demonstram o funcionamento da missão:

O primeiro passo foi verificar que se encontrava-num túnel, através da condição *Its-Tunnel*, tendo essa condição retornado sucesso. Ele decidiu executar a manobra M2.



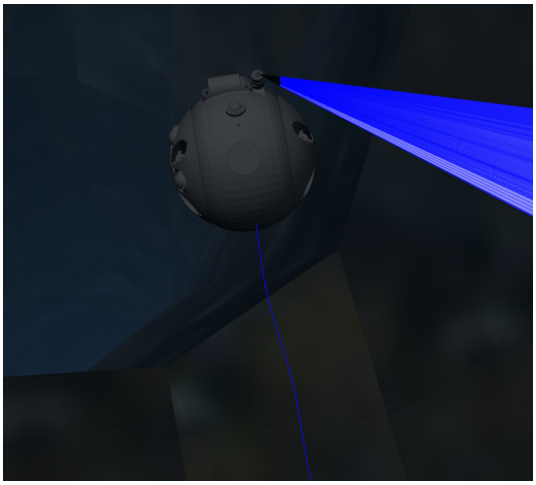
(a) O UX-1 reconhece que está num túnel, executando a M2



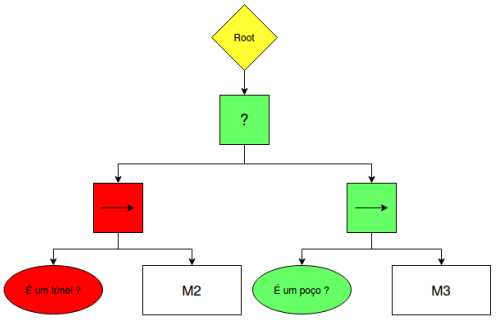
(b) A condição *ItsTunnel* retorna Sucesso, iniciando assim a manobra M2.

Figura 8.9: Inicio da missão.

Na aproximação a interseção, o robô deteta o poço, com isso a condição *ItsShaft* retorna sucesso, e a condição *ItsTunnel*, retorna falha.



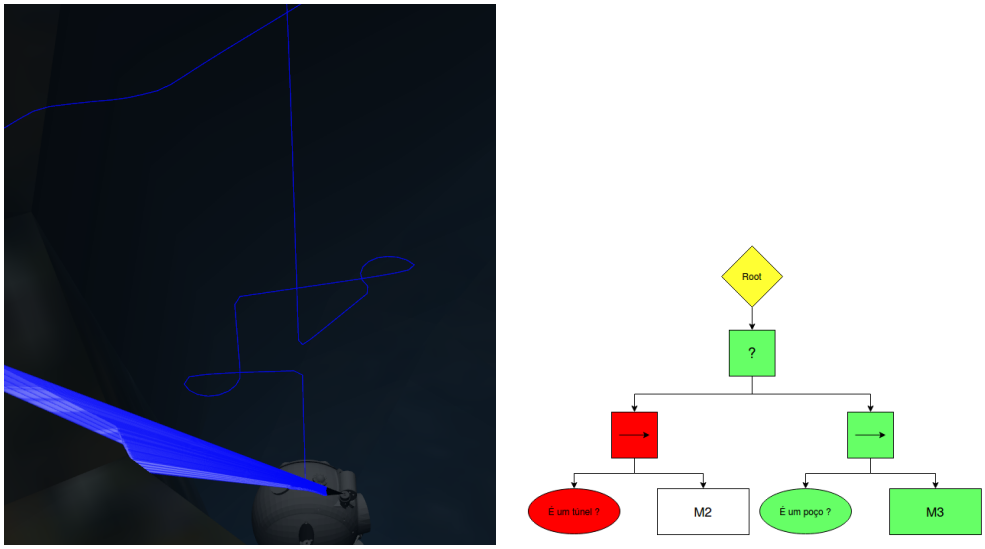
(a) O UX-1 encontra um poço.



(b) Ao detetar um poço a condição *ItsShaft* retorna Sucesso e a condição *ItsTunnel* retorna falha.

Figura 8.10: Deteção de um novo cenário.

Como a condição de poço retornou sucesso, o veículo executou a manobra M3 para descer o poço.



(a) O UX-1 desce ao longo do poço, utilizando a M3.

(b) Com a condição *ItsShaft* a retornar Sucesso o veículo Executa da manobra M3.

Figura 8.11: Execução da manobra M3.

As figura 8.12 e 8.13, demonstração a posição do robô ao longo do tempo e percurso efetuado:

Figura 8.12: Posição ao longo do tempo do robô.

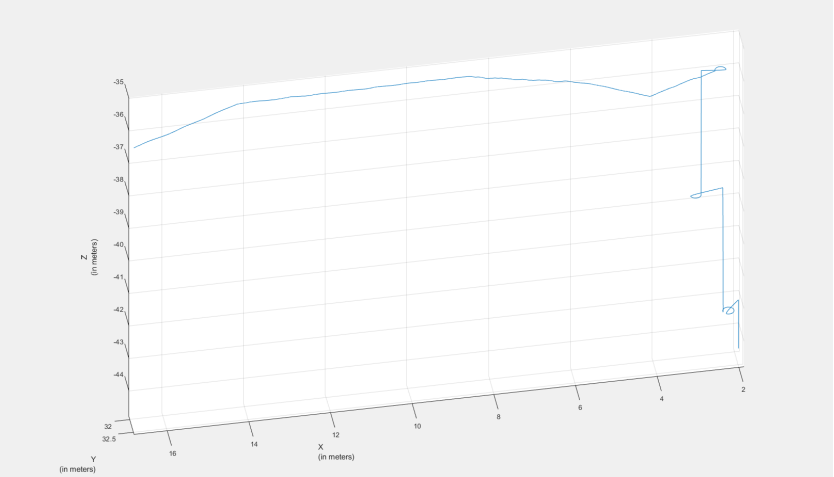


Figura 8.13: Gráfico 3D da trajetória do robô ao longo da missão.

Para demonstração, foi realizado outro teste com uma missão de nível 1. O objetivo desta nova missão foi descer um poço, percorrer um túnel terminando com a descida de um novo poço. Nas figuras 8.14 e 8.15, podemos observar a trajetória efetuada pelo veículo e as transições da BT da missão respetivamente.

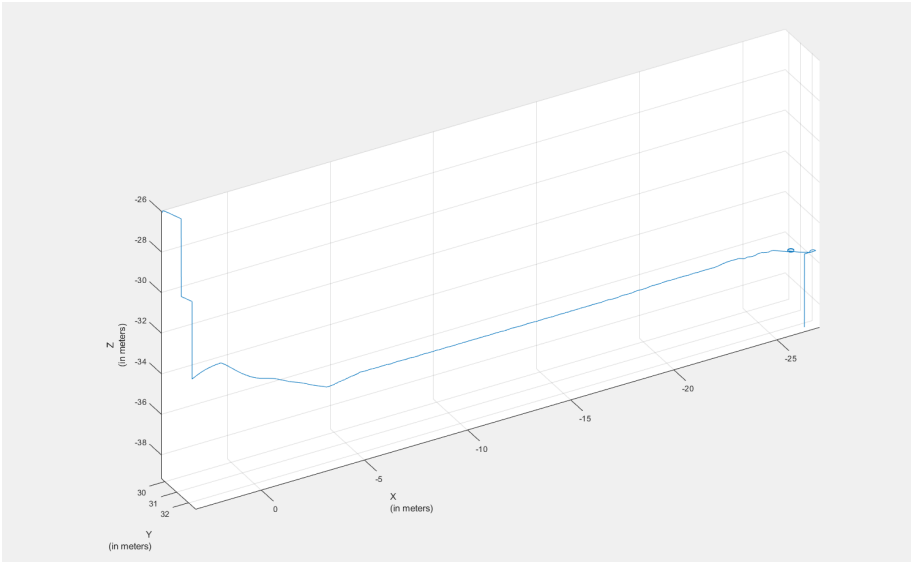
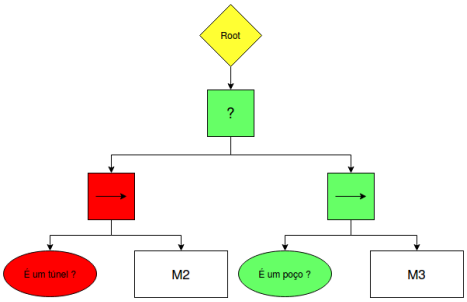
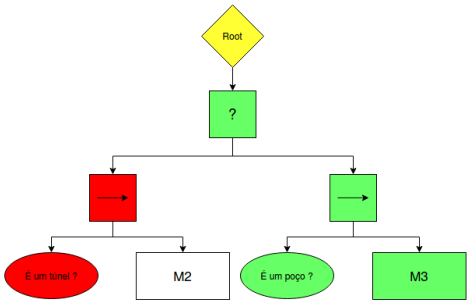


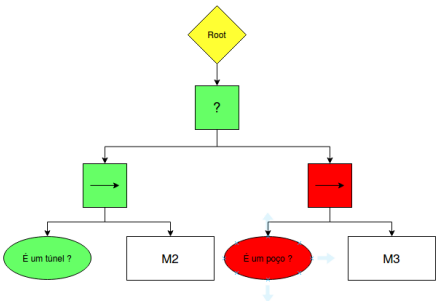
Figura 8.14: Trajetória 3D do robô ao longo da segunda missão.



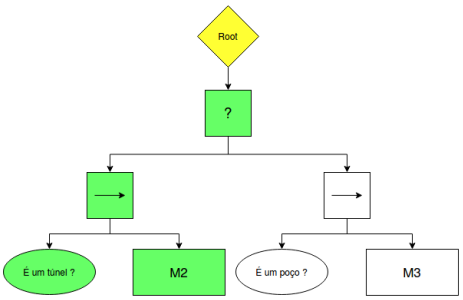
(a) A condição *ItsShaft* retornou Sucesso.



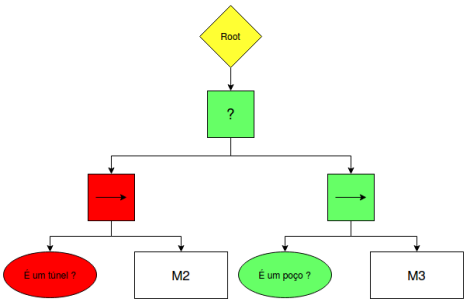
(b) O UX-1 executou a manobra M3.



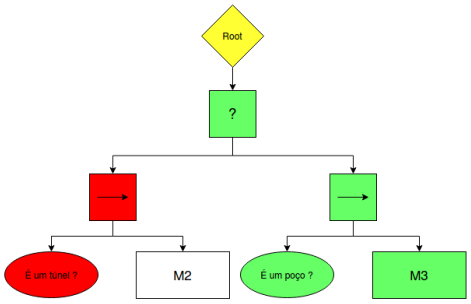
(c) A condição *ItsTunnel* retornou Sucesso, fazendo com que a *ItsShaft* retorne Falha.



(d) O UX-1 executou a manobra M2.



(e) Ao encontrar um novo poço a condição *ItsShaft* retorna Sucesso.



(f) O UX-1 executa de novo a manobra M3.

Figura 8.15: Comportamento da BT ao longo da missão de exploração.

As transições observadas na figura podem ser complementadas analisando, a posição ao longo do templo, figura 8.16.

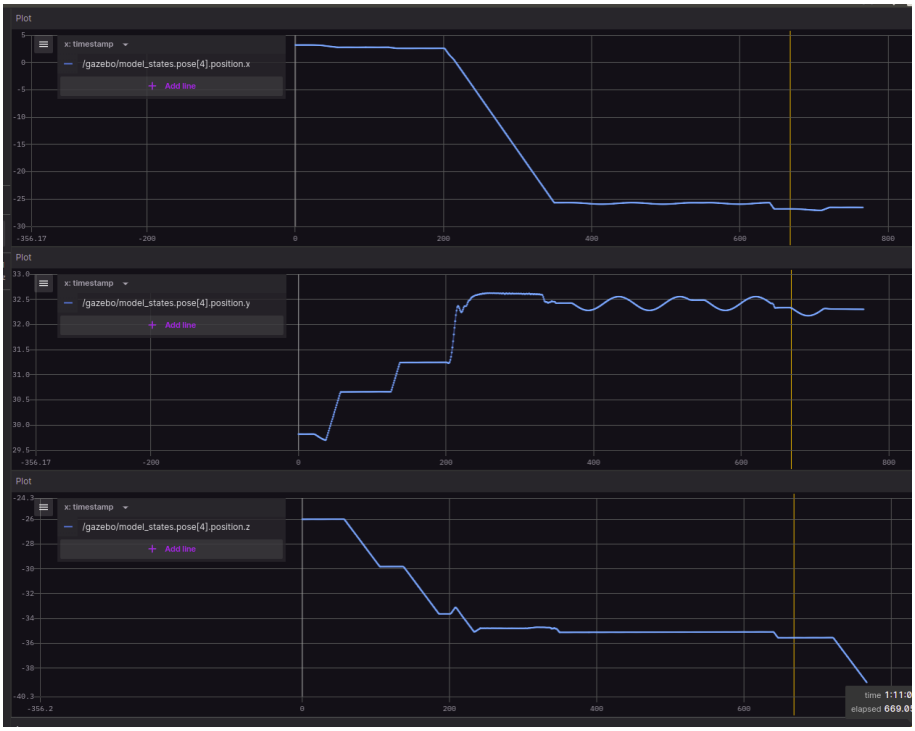


Figura 8.16: Posição ao longo do tempo do UX-1.

8.2.1.2 Missão de Nível 2

Os resultados para a missão de nível 2 foram semelhantes, pois as missões são quase idênticas, tendo apenas sido acrescentado o ramo para explorar galerias. A missão planeada para o UX-1, foi descer um túnel, em seguida percorrer um túnel, terminando com o mapeamento de uma galeria.

A figura 8.17, demonstra o percurso efetuado pelo robô ao longo da manobra de exploração.

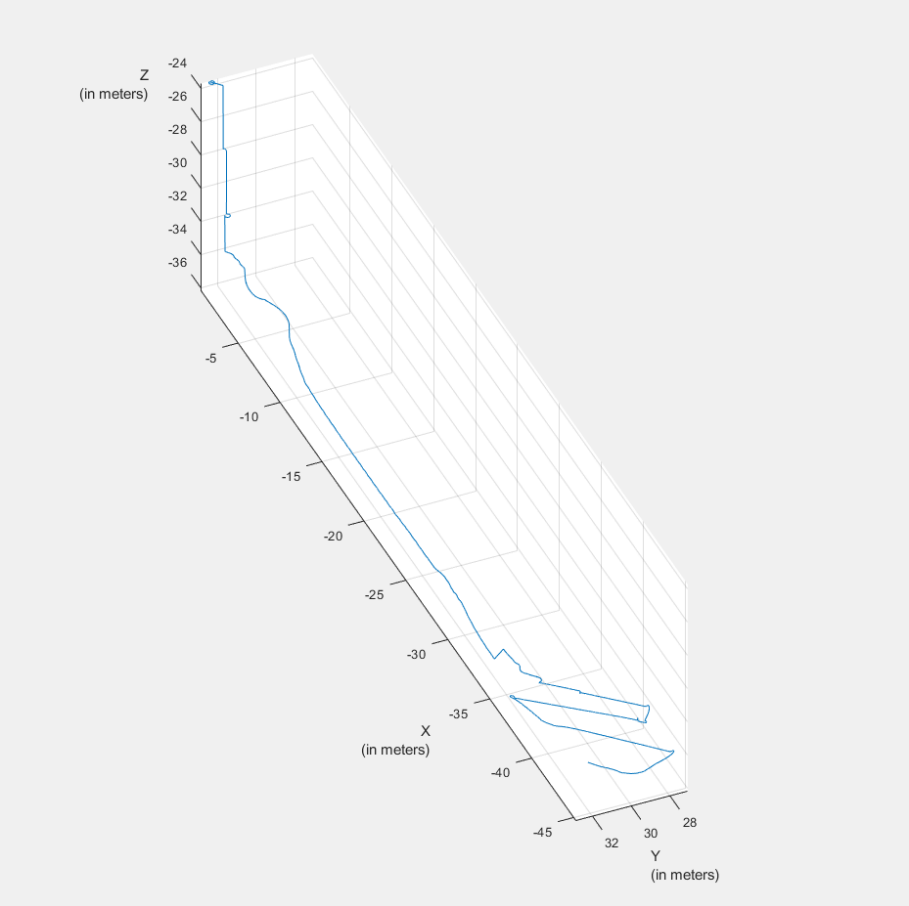


Figura 8.17: Trajetória 3D do UX-1 ao longo da missão.

Observando a trajetória, podemos concluir que o UX-1 iniciou a missão com a manobra M3 para descer um poço com 10 metros. Em seguida percorreu um túnel recorrendo a manobra M2 com um cumprimento de 30 metros. Por último podemos observar a aproximação do robô a galeria, com aproximadamente $50m^2$ de área, figura 8.18. Onde a condição *ItsGallery* retornou sucesso, mudando assim da M2 para a M4 para começar a explorar e mapear a galeria.

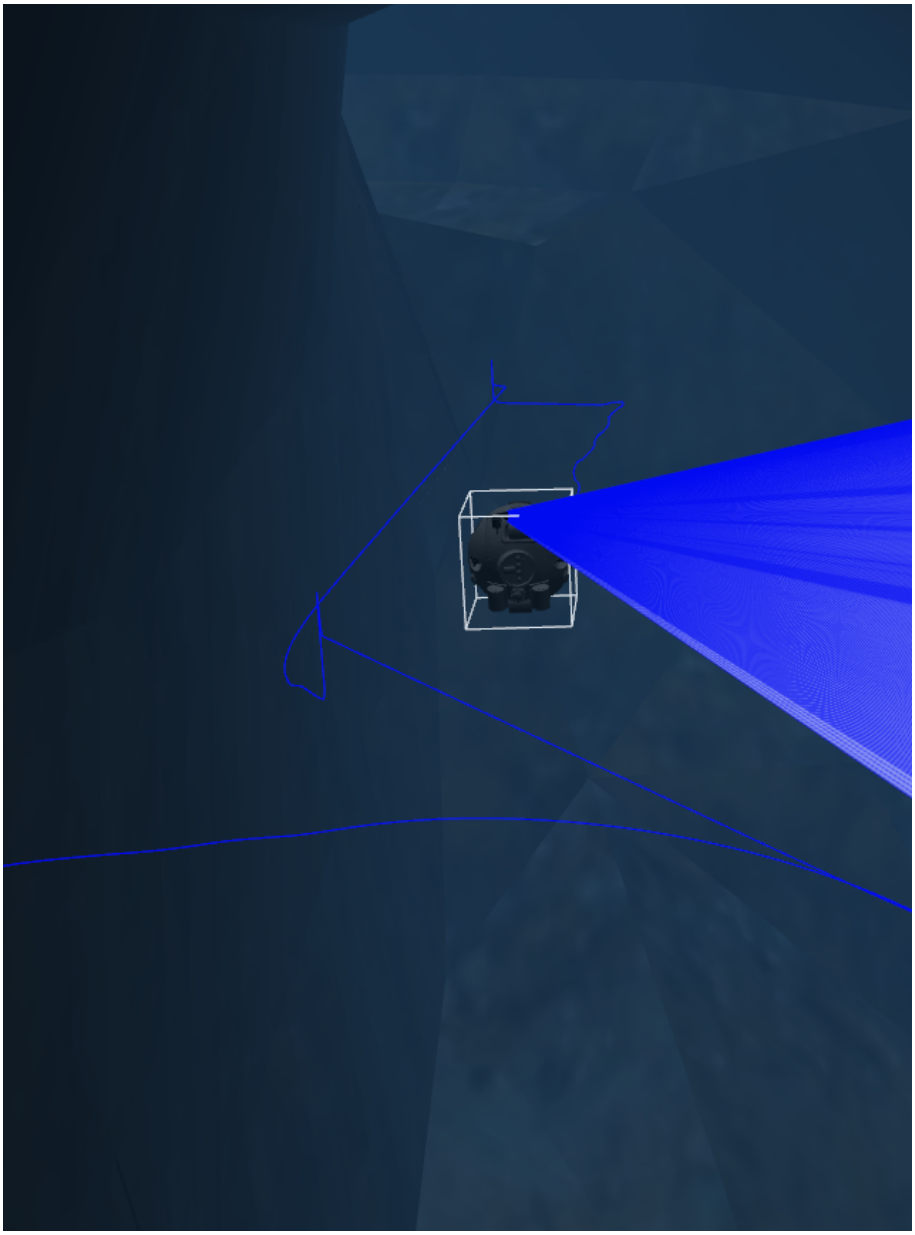


Figura 8.18: Transição com sucesso de um túnel para uma galeria.

Na figura 8.19 abaixo, podemos ver a BT utilizada. A BT de nível 1, foi adicionado o ramo responsável pela exploração de galerias, como forma de demonstrando que também é possível implementar o conceito de *Sub-Trees*.

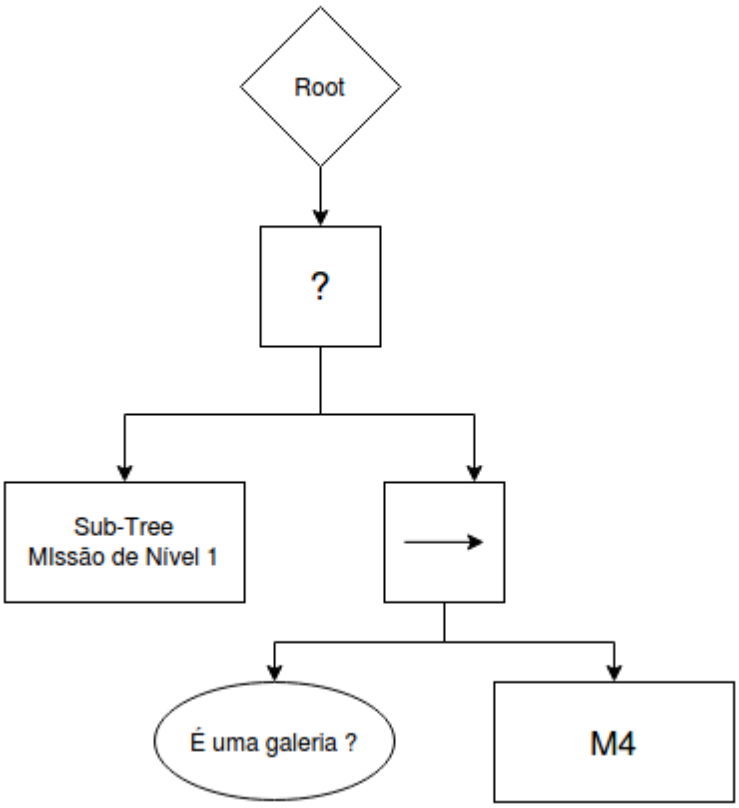
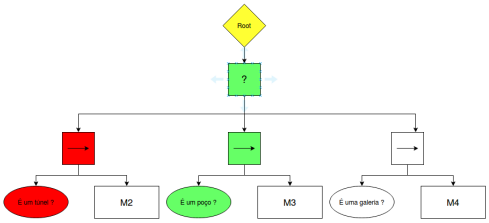
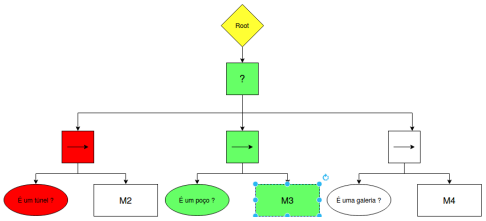


Figura 8.19: Construção de uma BT de nível 2 utilizando uma *Sub-Tree* com a missão de nível 1 e uma ramificação de exploração de galerias.

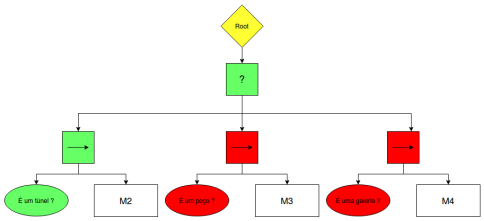
As transições da BT da missão, podem ser observadas na figura 8.20 abaixo.



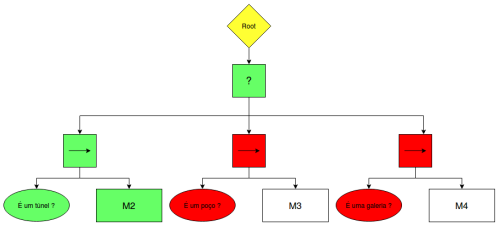
(a) Falha da condição *ItsTunnel*, passando a testar a condição *ItsShaft*, que retornou Sucesso, por ser um *Fallback node*.



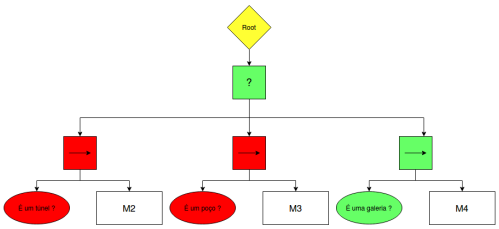
(b) O UX-1 executou a manobra M3, para descer o poço de 10m.



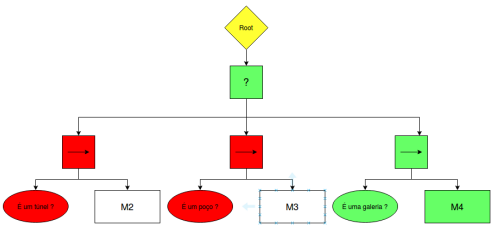
(c) Ao encontrar um túnel, as condições *ItsShaft* e *ItsGallery* retornam Falha. Obrigando a BT a recomeçar e a testar a condição *ItsTunnel*, que retorna Sucesso.



(d) O UX-1 executou a manobra M2 para percorrer o túnel com 30m.



(e) Deteção de uma galeria, provocou a Falha das condições *ItsTunnel* e *ItsShaft*, e o Sucesso da condição *ItsGallery*.



(f) O UX-1 executa a manobra M4, para explorar e mapear a galeria com $50m^2$.

Figura 8.20: Transições da BT ao longo da missão de nível 2.

Existiram dois problemas com esta missão, a serem abordados no capítulo do trabalho futuro. Um deles é o facto da manobra M4 de mapeamento ainda ter alguns problemas ao começar a manobra, nomeadamente, como se aproximar de uma parede. Um segundo problema acontece no final de uma galeria, pois a não ser que o robô detete alguma interseção, ele termina a missão.

8.2.2 High Level Mission

Nesta parte do projeto não houve qualquer resultados positivos, pois não foi possível implementar uma missão de nível 3 ou superior. Isto deveu-se nas dificuldades, encontradas a desenvolver os dois primeiros níveis, em conjunto com a falta de recursos e ferramentas tanto da biblioteca, como do GUI Groot. Para as missões de alto nível é suposto que nas BT, seja possível implementar blocos de memória, de forma a guardar as interseções encontradas. No entando, com a versão utilizada ainda não existem esse nós e condições a funcionar, partindo da parte do utilizador o desenvolvimento de novo *software*, envolvendo mexer em toda a *framework* da biblioteca. A forma de resolver estes problemas, e futuras soluções serão abordadas no capítulo seguinte.

Capítulo 9

Conclusões

O trabalho realizado nesta dissertação providencia à comunidade um conjunto de ferramentas e soluções enquadradas nos desafios propostos pela exploração sub-aquática autónoma. Além disso os resultados obtidos da utilização de *Behaviour trees* como *path planning*, permite em futuros projetos ter mais um exemplo de aplicação e tomar a decisão de as incorporar no mesmo. O trabalho proposto, foi desenvolvido num ambiente de simulação com elevado detalhe e funcionalidades.

Em termos de resultados esta dissertação mostrou tanto pontos positivos como negativos, sendo que grande parte dos objetivos definidos foram concretizados. Este projeto permitiu o desenvolvimento de cinco manobras independentes de exploração em ROS. Esta manobras são nós independentes e como tal no futuro podem ser aplicadas noutros projetos e mesmo no UX-1 real. As cinco manobras desenvolvidas foram a M1, M2, M3, M4 e M5(*Wall-Following*, *Center Tunnel*, *Vertical Shaft*, *Cave Mapping* e *Slope Tunnel* respetivamente). Os resultados obtidos em simulação foram positivos. A incorporação das manobras nas BTs, foi um desafio, mas que permitiu a sua afinação, tornando assim em algoritmos mais robustos. A implementação das *Behaviour Trees*, como controlador de missão foi o desafio mais difícil e mais demorado do projeto. As BTs provaram ser uma metodologia simples e com boa compatibilidade com os algoritmos para exploração autónoma. Foram produzidos com sucesso dois níveis de missões. Os resultados observados no capítulo anterior provaram a sua viabilidade de implementação. No entanto devido a todos os problemas referidos, não foi possível implementar missões de alto nível. Em resumo as BTs podem ser utilizadas na exploração autónoma, pois provaram ser adequadas para resolver problemas encontrados, utilizando apenas o recursos disponíveis sem qualquer informação externa.

9.1 Dificuldades

Ao longo do projeto, foram impostas múltiplas barreiras que se provaram ser difícil de avançar. A principal, foi a falta de informação e exemplos práticos e funcionais, para implementar as *Behaviours Trees*. A quando a implementação do projeto, a biblioteca utilizada apesar de ser uma das poucas compatível com ROS, ainda existia pouco desenvolvimento por parte do autor, pois na altura era mais um projeto secundário dependente de doações para o seu desenvolvimento. Um dos fatores, para que a implementação das BT's tenha sido tão difícil foi também, pela falta do GUI Groot. Com esse ambiente de desenvolvimento é possível criar as BT's sem termos de nos preocupar com dependências, fazer *debbug* em tempo real e muito mais. Outras dificuldades apresentadas, foram relacionadas com o utilização do simulador e falta de conhecimentos de ROS e de programação em C++.

9.2 Trabalho futuro

No futuro será importante melhorar as manobras desenvolvidas, e possivelmente também adicionar novas para complementar o leque de opções. Uma manobra a ser implementada que não foi possível, é uma manobra onde robô seja capaz de descer ou subir com um ângulo de 90 graus um poço, dando a possibilidade do M3-*Multibeam* estar na frente do movimento e detetar possíveis obstáculos. O processo de aprimorar as manobras também passa pela incorporação de de novos sensores existentes abordo, fazendo com que em cada manobra exista o máximo de informação sobre o meio ambiente. Em relação as BT's, será importante atualizar a biblioteca utilizada, pois no momento a que esta dissertação se encontra a ser escrita, já se encontram dísponiveis novas versões, com mais funcionalidades da biblioteca **BehaviorTree.CPP**, e do Groot. Com essas atualizações será mais fácil implementar as arquiteturas das missões de alto nível. Tendo como objetivo a implementação e validação no veículo autónomo real. Como descrito no capítulo 6, o UX-1 para realizar missões reais de exploração autónoma, tem de ser capaz de ter por exemplo uma *interface* de controlo, onde o utilizador define parâmetros de tempo, prioridades, objetivos,etc...

Bibliografia

[1] Alfredo Martins, José Almeida, Carlos Almeida, André Dias, Nuno Dias, Jussi Aaltonen, Arttu Heininen, Kari T. Koskinen, Claudio Rossi, Sergio Dominguez, Csaba Vörös, Stephen Henley, Mike McLoughlin, Hilco van Moerkerk, James Tweedie, Balazs Bodo, Norbert Zajzon, and Eduardo Silva. Ux 1 system design - a robotic system for underwater mining exploration. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1494–1500, 2018.

[2] Clment Petres, Yan Pailhas, Pedro Patron, Yvan Petillot, Jonathan Evans, and David Lane. Path planning for autonomous underwater vehicles. *IEEE Transactions on Robotics*, 23(2):331–341, 2007.

[3] Christopher Whitt, Jay Pearlman, Brian Polagye, Frank Caimi, Frank Muller-Karger, Andrea Copping, Heather Spence, Shyam Madhusudhana, William Kirkwood, Ludovic Grosjean, Bilal Muhammad Fiaz, Satinder Singh, Sikandra Singh, Dana Manalang, Ananya Sen Gupta, Alain Maguer, Justin J. H. Buck, Andreas Marouchos, Malayath Aravindakshan Atmanand, Ramasamy Venkatesan, Vedachalam Narayanaswamy, Pierre Testor, Elizabeth Douglas, Sebastien de Halleux, and Siri Jodha Khalsa. Future vision for autonomous ocean observations. *Frontiers in Marine Science*, 7, 2020.

[4] Nathaniel Fairfield, George Kantor, Dom Jonak, and David Wettergreen. Depthx autonomy software: Design and field results. 05 2012.

[5] Angelos Mallios, Pere Ridao, David Ribas, Marc Carreras, and Richard Camilli. Toward autonomous exploration in confined underwater environments. *Journal of Field Robotics*, 33:n/a–n/a, 11 2015.

[6] Nathaniel Fairfield, George A. Kantor, Dominic Jonak, and David S. Wettergreen. Depthx autonomy software: Design and field results. 2008.

[7] Angelos Mallios, Pere Ridao, David Ribas, Marc Carreras, and Richard Camilli. Toward autonomous exploration in confined underwater environments. *Journal of Field Robotics*, 33:n/a–n/a, 11 2015.

[8] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[9] Kristof Richmond, Chris Flesher, Laura Lindzey, Neal Tanner, and William C. Stone. Sunfish®: A human-portable exploration auv for complex 3d environments. In *OCEANS 2018 MTS/IEEE Charleston*, pages 1–9, 2018.

[10] Sunfish® auv – pushing the boundaries of exploration amp; technology.

[11] Mohammad Alenezi and Abdullah Almeshal. Optimal path planning for a remote sensing unmanned ground vehicle in a hazardous indoor environment. *Intelligent Control and Automation*, 09:147–157, 01 2018.

[12] Genya Ishigami, Keiji Nagatani, and Kazuya Yoshida. Path planning for planetary exploration rovers and its evaluation based on wheel slip dynamics. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 2361–2366, 2007.

[13] Anantha Sai Hari Haran V Injarapu and Suresh Kumar Gawre. A survey of autonomous mobile robot path planning approaches. In *2017 International Conference on Recent Innovations in Signal processing and Embedded Systems (RISE)*, pages 624–628, 2017.

[14] Hendrik Hoogeboom and Joost Engelfriet. *Pushdown Automata*, volume 148. 01 2004.

[15] Richard Balogh and David Obdržálek. *Using Finite State Machines in Introductory Robotics: Methods and Applications for Teaching and Learning*, pages 85–91. 01 2019.

[16] Matteo Iovino, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith. A survey of behavior trees in robotics and ai. *Robotics and Autonomous Systems*, 154:104096, 2022.

[17] Michele Colledanchise and Petter Ögren. *Behavior Trees in Robotics and AI*. CRC Press, jul 2018.

[18] Michele Colledanchise and Petter Ogren. *Behavior Trees in Robotics and AI: An Introduction*. 07 2018.

[19] ROS.org | Powering the world's robots. `https://www.ros.org/`. Accessed: 2022-06-15.

[20] Ayssam Elkady and Tarek Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012, 05 2012.

[21] Osrf. Ros overview.

[22] M3 multibeam sonar.

[23] Doppler velocity log (dvl) for underwater navigation i dvl1000 - 300 m.

[24] Sebastian Thrun. Simultaneous localization and mapping. In *Robotics and cognitive approaches to spatial mapping*, pages 13–41. Springer, 2007.

[25] Posted bySebastian Castro. Introduction to behavior trees - robotic sea bass, Jun 2022.

[26] Renato de Pontes Pereira and Paulo Martins Engel. A framework for constrained and adaptive behavior-based agents. *CoRR*, abs/1506.02312, 2015.

[27] Michele Colledanchise and Petter Ögren. How behavior trees modularize robustness and safety in hybrid systems. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1482–1488, 2014.

[28] Soheil Zavari, Arttu Heininen, Jussi Aaltonen, and Kari T. Koskinen. Early stage design of a spherical underwater robotic vehicle. In *2016 20th International Conference on System Theory, Control and Computing (ICSTCC)*, pages 240–244, 2016.

[29] David Kortenkamp, Reid Simmons, and Davide Brugali. Robotic systems architectures and programming. In *Springer Handbook of Robotics*, pages 283–306. Springer, 2016.

[30] Ronald C Arkin, Ronald C Arkin, et al. *Behavior-based robotics*. MIT press, 1998.

[31] J. G. Ziegler and N. B. Nichols. Optimum settings for automatic controllers. *Journal of Fluids Engineering*, 64(8):759–765, 1942.

[32] PointCloudLibrary. pointcloudlibrary/pcl.