

Projeto Final

Veículos Não Tripulados

2024 / 2025

24705 Rúben Filipe Alcobia Dias

24993 Diogo Filipe Videira dos Santos Larangeira

Licenciatura em Engenharia Informática

Veículos Não Tripulados

24705 Rúben Filipe Alcobia Dias

24993 Diogo Filipe dos Santos Videira Larangeira

Janeiro, 2025

Orientadores:

Ana Lopes

Luís Oliveira

Pedro Correia

Agradecimentos

Agradecemos a ajuda dos nossos orientadores, que nos acompanharam constantemente e ajudaram-nos com as devidas soluções face às dificuldades encontradas.

Resumo

Recorrendo ao simulador Gazebo na versão Harmonic, pretende-se criar cenários realistas, de modo a testar as capacidades de um veículo não tripulado terrestre daqui em diante designado por UV(Unmanned Vehicle), cujo controlo será executado através do sistema de navegação do ROS2, denominado Nav2.

Assim, utilizando os mesmos parâmetros que seriam usados para comandar um UV terrestre, possibilitou-se através do ROS2 Humble a comunicação entre os diferentes componentes do sistema.

No intuito de aprender os conceitos necessários à execução deste projecto, seguiram-se os tutoriais existentes na documentação do ROS2[[1](#)].

Palavras Chave: ROS2 Humble, Gazebo Harmonic, Nav2, Veículo Não Tripulado Terrestre, UV(Unmanned Vehicle).

Conteúdo

1	Introdução	1
1.1	Plano de Trabalho	3
1.1.1	XACRO e LAUNCH	3
1.1.2	LIDAR no RVIZ2	4
1.1.3	SITL (ArduPilot).....	4
1.1.4	Parâmetros ArduPilot.....	4
1.1.5	Implementação do Algoritmo Dijkstra	4
1.1.6	Utilizar LIDAR data para realizar Decisões	4
1.1.7	Integração de Sensores Térmicos.....	4
1.1.8	Integração de Algoritmos de Navegação	4
1.1.9	Testes e Documentação.....	5
2	Estado da arte	6
2.1	Estado da Arte - Veículos Terrestres	6
2.2	Estado da Arte - Veículos Aéreos.....	8
	Exemplos de uso	8
2.3	Tecnologias Utilizadas.....	9
3	Trabalho de Projecto	10
3.1	Objetivos do Trabalho desenvolvido	10
3.2	Simulação com Gazebo Fuel	12
3.3	Estrutura do modelo de um veículo	18
3.4	Criação do modelo de um veículo e de um mundo.....	23
3.4.1	Construção de um Cenário com um mundo e um modelo personalizado	25
3.5	Navegação do Veículo	28
3.5.1	Componentes do NAV2	37

3.5.2	Algoritmos de planeamento de caminhos	38
	planner_server.....	38
3.6	Navegação Autónoma com o Turtlebot3	40
3.6.1	Cenário Turtlebot3	42
3.6.2	Cenário Mundo Irregular	46
	47
3.7	Navegação no mapa através de um Servidor Flask	53
4	Conclusões	56
	Bibliografia	57
	Anexo 1 – Enunciado do projeto	59
	Anexo 2 - Troubleshooting na instalação do software (ROS2, Gazebo)	60
1.1.2	Instalação do ROS2:	62
1.1.3	Verificar a instalação:	62
1.2.1	Criação de uma nova diretoria	63
1.2.2	Clonar um repositório	63
1.2.3	Resolver dependências	63
1.2.4	Construir o workspace	63
1.2.5	Criação do overlay	64
1.2.6	Verificação da configuração correcta do <i>underlay/overlay</i>	64
1.2.7	Criação de uma Package	67
1.2.8	Verificação do pacote “simulacao_package”.....	67
1.3	Simulação de um veículo com 2 rodas com RVIZ	68
1.3.1	Criação de uma Package	68
1.3.2	Ficheiro XACRO	68
1.3.3	Ficheiro launch.....	71
1.3.4	Inicialização do projeto	72

1.3.5	Gazebo launch file	75
1.3.6	Spawn veículo na simulação Gazebo.....	77
1.4	Gazebo	79
1.4.1	Instalação	79
1.4.2	Verificação da instalação	79
1.4.3	Comunicar com a simulação através do ROS2.....	81
1.5	Visualizar data LIDAR	84
1.5.1	Criação de uma nova bridge	84
Anexo 3 - Instalação do software necessário (ROS2, Gazebo, ArduPilot)		86
Passo 1: Instalação do ROS2 Humble		86
Definição de um locale que suporte UTF-8		86
Configuração das Fontes (Sources)		86
Instalação dos pacotes do ROS2		87
Preparação do ambiente		87
Passo 2: Instalação do ArduPilot com ROS2.....		88
Passo 3: Instalação do Gazebo Harmonic		89
Correr a Simulação		91
Descolagem e Aterragem.....		92
Anexo 4 - Versões Gazebo		93

Figura 1 - Gráfico de Gant	3
Figura 2 - ANYmal, UV Terrestre	6
Figura 3 - UV Terrestre utilizado pelas autoridades na China.....	7
Figura 4 - Locais onde a empresa Zipline opera.....	8
Figura 5 - Mundo simples gerado pelo Gazebo, Mundo de Cubos.	12
Figura 6 - Página com o nosso Perfil na Coleção “Projeto Final”.....	13
Figura 7 - Modelo X1 Config5.	13
Figura 8 - Página no Software Gazebo que permite escolher “Resource Spawner”....	14
Figura 9 - Escolha do Modelo X1 Config5 e a sua inserção no mundo.	15
Figura 10 - Imagem do mundo escolhido.	15
Figura 11 - Página no Gazebo Fuel do mundo escolhido	16
Figura 12 - Comando no terminal que permite iniciar o Gazebo no novo mundo.....	16
Figura 13 - Visualização do modelo no mundo.	17
Figura 14 - Estrutura de um ficheiro XACRO.....	21
Figura 15 - Definição do ficheiro XACRO.....	21
Figura 16 - Modelo Veículo.....	23
Figura 17 - Modelo Mundo Irregular.....	24
Figura 18 - Modelo Mundo Realista	24
Figura 19 - Modelo do veículo em ambiente de simulação	25
Figura 20 - Modelo do veículo em ambiente de simulação	26
Figura 21 - Simulação utilizando os modelos criados (veículo e mundo).....	27
Figura 22 - Observação do veículo no RViz pelo algoritmo AMCL.....	29
Figura 23 - Solução para um grafo utilizando o algoritmo Dijkstra	29
Figura 24 - Solução para um grafo utilizando o algoritmo A*	30
Figura 25 - Diferença entre algoritmo A* e Hybid A*	30
Figura 26 - Algoritmo Theta*	31

Figura 27 - Algoritmo State Lattice	32
Figura 28 - Algoritmo Dynamic Window Approach	32
Figura 29 - Algoritmo Time Elastic Band	33
Figura 30 - Trajetória seguindo o algoritmo RPP	34
Figura 31 - Escolha de um caminho usando o algoritmo MPPI	34
Figura 32 - Exemplo Mapa de custos	36
Figura 33 - Exemplo Mapa de obstáculos.....	36
Figura 34 - Descrição do “planner_server” para Dijkstra.....	38
Figura 35 - Descrição do “planner_server” para A*.....	39
Figura 36 - Descrição do “planner_server” para Theta*.	39
Figura 37 - Mundo Turtlebot3	40
Figura 38 - Escolher um objetivo para o veículo.....	41
Figura 39 - Posição final.....	42
Figura 40 - Posição inicial.	42
Figura 41- Mapeamento manual do mundo criado.	47
Figura 42 - Imagem .png convertida do ficheiro gerado .pgm.	48
Figura 43 - Posição final do cenário Irregular	49
Figura 44 - Posição inicial do cenário Irregular.....	49
Figura 45 - Página definida no ficheiro HTML	53
Figura 46 - Resposta da página à interação no mapa.....	54

1 Introdução

Os veículos não tripulados devem a sua autonomia ao facto de conseguirem obter informação através dos dados provindos dos seus sensores, e, com ela, tomarem decisões em tempo real. São essas mesmas decisões, executadas pelos seus atuadores, que lhes levam a obter o sucesso na concretização dos seus objetivos. Propõe-se, portanto, neste trabalho, explorar e aplicar estas tecnologias em simulações de situações de emergência.

Este projeto insere-se num panorama mais amplo de iniciativas tecnológicas que visam não apenas aumentar a eficiência das operações, mas também mitigar riscos para a segurança.

O trabalho realizado aborda um tema central na evolução tecnológica, nomeadamente a integração de veículos não tripulados no dia a dia, focando-se em situações mais críticas. Face ao estado da arte, este estudo destaca-se por identificar lacunas em soluções existentes e propor melhorias que podem aumentar a eficiência e a adaptabilidade destes veículos. Entre os aspetos inovadores estão a combinação de algoritmos de navegação avançados que permitem ao UV tomar decisões em tempo real autonomamente.

Os benefícios desta investigação para organizações incluem a possibilidade de reduzir custos operacionais, melhorar a segurança de trabalhadores e ampliar a capacidade de atuação em ambientes adversos. A incorporação de elementos inovadores também pode servir como base para novas aplicações industriais ou ambientais.

O projeto consiste na simulação de cenários em que veículos não tripulados terrestres são utilizados para atingir diversos objetivos. A necessidade do uso destes veículos varia de cenário para cenário. O objetivo principal será conseguir realizar diga-se missões, ou seja, um conjunto de atividades que incluem, nos veículos terrestres, trajetos autónomos, para atingir um fim desejado, entenda-se missões de socorro.

Um exemplo prático de aplicação seria a sua utilização em situações de emergência, como incêndios florestais, onde os veículos podem aceder a zonas de difícil alcance

para recolher informações cruciais que auxiliem na tomada de decisões e definição de estratégias de resposta ao incidente.

É utilizado o simulador Gazebo, na sua versão Harmonic (compatível com o ROS 2 Humble), para a construção dos cenários num ambiente tridimensional realista, que inclui terrenos variados e objetos dinâmicos.

O controlo do veículo não tripulado é realizado diretamente através dos módulos do ROS2, recorrendo ao Nav2 para a navegação autónoma, o que permite planear trajetórias, evitar obstáculos e alcançar objetivos definidos no mapa.

Para a comunicação entre os diferentes módulos do veículo como sensores, algoritmos de navegação, planeadores globais e locais, é utilizado o middleware ROS 2, que assegura uma integração modular e eficiente entre os diversos componentes.

O uso de software externo como o ArduPilot foi uma solução inicialmente estudada, mas acabou por ser descartada devido à falta de sucesso na sua integração com o restante sistema, sendo substituído pelo Nav2

O presente trabalho teve como principais objetivos:

- Compreender o funcionamento das principais ferramentas, nomeadamente o Gazebo Harmonic, o ROS 2 (Humble) e o sistema de navegação autónoma Nav2;
- Estudar em detalhe os planeadores e algoritmos integrados no Nav2;
- Desenvolver modelos personalizados, incluindo a criação de mundos, obstáculos e veículos simulados, para testar cenários realistas;
- Avaliar o desempenho dos algoritmos de planeamento, através da recolha de métricas;
- Estabelecer comunicação com o sistema de navegação através de uma interface externa, concretamente por meio de um servidor web desenvolvido em Flask, que permite enviar pontos de destino ao robô de forma remota.

Os objetivos gerais descritos nesta secção encontram-se mais detalhados e especificados no Capítulo 4, onde são apresentadas as metas técnicas específicas, bem

como a forma como cada uma foi abordada e implementada ao longo do desenvolvimento do trabalho.

No capítulo seguinte, será apresentada uma revisão aprofundada do estado da arte (Capítulo 2).

1.1 Plano de Trabalho

Para um desenvolvimento organizado e, por consequência, eficiente, do trabalho a realizar, foi criado um gráfico de Gantt:

Simulação UV de Socorro

→ Gráfico de Gant (Data limite 10 de Julho 2025)

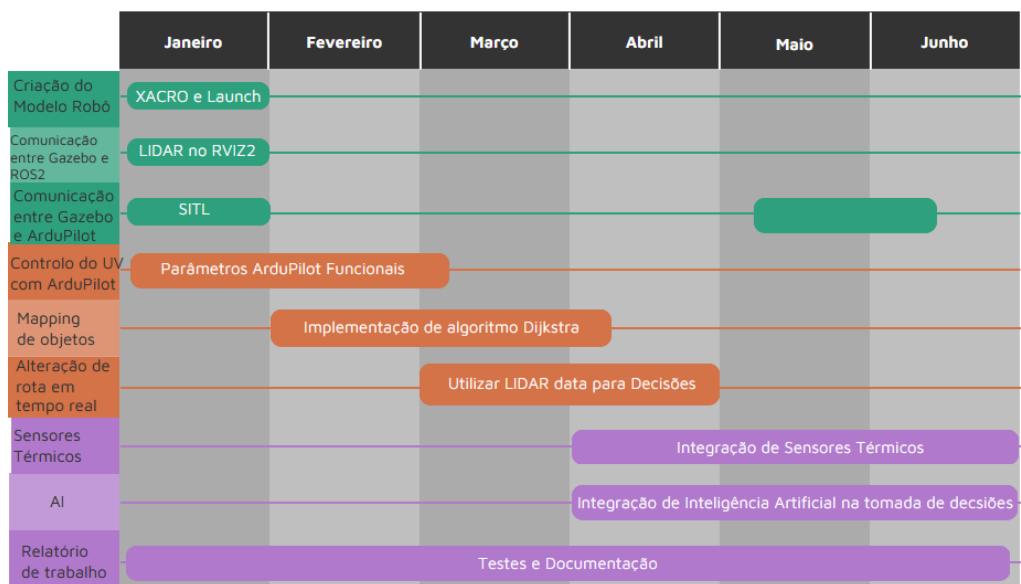


Figura 1 - Gráfico de Gant

1.1.1 XACRO e LAUNCH

Para ser possível visualizar o modelo no Gazebo, é necessário a criação de um ficheiro XACRO. Este modelo será depois publicado através de um node no ficheiro LAUNCH assim como outros componentes que forem necessários.

1.1.2 LIDAR no RVIZ2

Para se visualizar a data do sensor LIDAR que é enviada pelo simulador Gazebo, recorrer-se-á ao software RVIZ2. Para tal, é necessário que este receba corretamente a data e a represente dentro de um fixed frame.

1.1.3 SITL (ArduPilot)

Para conseguir-se integrar o ArduPilot neste projeto utiliza-se um software de simulação chamado SITL. Este software permite correr o ArduPilot autopilot sem que seja necessário qualquer hardware. Possibilita a navegação autónoma.

1.1.4 Parâmetros ArduPilot

Com a instalação correcta do SITL, é necessária a configuração correcta dos ficheiros do veículo para que integre no seu código a sintaxe correta dos parâmetros do ArduPilot.

1.1.5 Implementação do Algoritmo Dijkstra

Para realizar o mapeamento dos seus arredores, o modelo podemos recorrer ao algoritmo Dijkstra, tendo este de ser implementado de acordo com a informação recebida do sensor LIDAR.

1.1.6 Utilizar LIDAR data para realizar Decisões

Assim, com a implementação de um algoritmo, será possível escolher qual é o melhor caminho a seguir para atingir o objetivo pretendido.

1.1.7 Integração de Sensores Térmicos

Para o UV conseguir detectar a presença de pessoas, recorre-se a um sensor térmico. Outra utilidade para o mesmo seria a deteção de incêndios.

1.1.8 Integração de Algoritmos de Navegação

Espera-se conseguir utilizar bases de dados já livremente existentes para que através do recurso à *knowledge base* o UV tome decisões óptimas.

1.1.9 Testes e Documentação

Os testes realizados durante todo o desenvolvimento do trabalho, sendo também feita a documentação dos resultados provindos dos mesmos.

Houve uma necessidade de abandonar o plano de trabalho anteriormente apresentado, devido a incompatibilização entre versões de ferramentas. Contudo procurou-se sempre preservar os objetivos centrais do trabalho, adaptando-os às novas ferramentas disponíveis. As alterações realizadas permitiram superar os entraves técnicos encontrados, como problemas de compatibilidade entre versões do ROS2, Gazebo e o ArduPilot.

A nova configuração, descrita no Anexo 4, garantiu uma maior estabilidade e integração entre os módulos, viabilizando o desenvolvimento de uma solução funcional e eficiente. Esta mudança revelou-se uma oportunidade para explorar outras abordagens mais modernas, como o uso do sistema de navegação Nav2 com o Gazebo Harmonic, proporcionando um ambiente de simulação mais estável e alinhado com as práticas atuais na robótica autónoma.

2 Estado da arte

2.1 Estado da Arte - Veículos Terrestres

No domínio da robótica terrestre, destaca-se o ANYmal^[2], desenvolvido pela Anybots, como um dos robôs mais avançados atualmente. Este robô combina locomoção adaptativa com sensores de alta precisão para operar autonomamente em terrenos desafiantes. Equipado com capacidades como mapeamento tridimensional, inspeção em tempo real e análise de dados, o ANYmal tem sido amplamente utilizado em setores como energia, construção e exploração ambiental.



Figura 2 - ANYmal, UV Terrestre

Ainda no domínio dos veículos terrestres, na China, um novo veículo não tripulado terrestre acompanha agentes da polícia chinesa [3], ajudando as autoridades no combate ao crime. A estrutura do veículo assemelha-se muito a uma esfera, o que lhe permite uma mobilidade eficiente em diversos terrenos, incluindo o meio aquático, garantindo maior alcance em zonas urbanas e suburbanas. O veículo combina sensores sofisticados, sistemas de inteligência artificial e conectividade avançada para monitorizar áreas públicas e identificar situações de risco. O mesmo ainda está equipado com armas não letais, como bombas de fumo, gás pimenta e dispersores de multidões acústicos.



Figura 3 - UV Terrestre utilizado pelas autoridades na China

2.2 Estado da Arte - Veículos Aéreos

No domínio dos veículos aéreos, o uso de drones para fins de alívio de desastres destaca-se como uma tecnologia essencial para situações de emergência. Drones equipados com câmaras de alta resolução e sensores térmicos são utilizados para mapear as áreas afetadas rapidamente.

No caso da ocorrência de desastres naturais — como terremotos ou furacões — são utilizados também na busca e no resgate da população, na entrega de medicamentos, alimentos, água e outros suplementos críticos a áreas isoladas como no caso das inundações, onde o uso dos drones torna desnecessária a incorrência em perigos de transporte terrestre para fornecer ajuda diretamente às comunidades isoladas.

Exemplos de uso

Existem já veículos aéreos não tripulados que realizam entregas ao domicílio como o caso do drone da Zipline[4].

Até à data, vários países já contam com as entregas destes drones que distribuem desde medicamentos a alimentos dentro de um curto espaço de tempo.

A livre circulação destes drones por zonas urbanas abre portas para que num futuro próximo possam também vir a ser utilizados em caso de emergências.

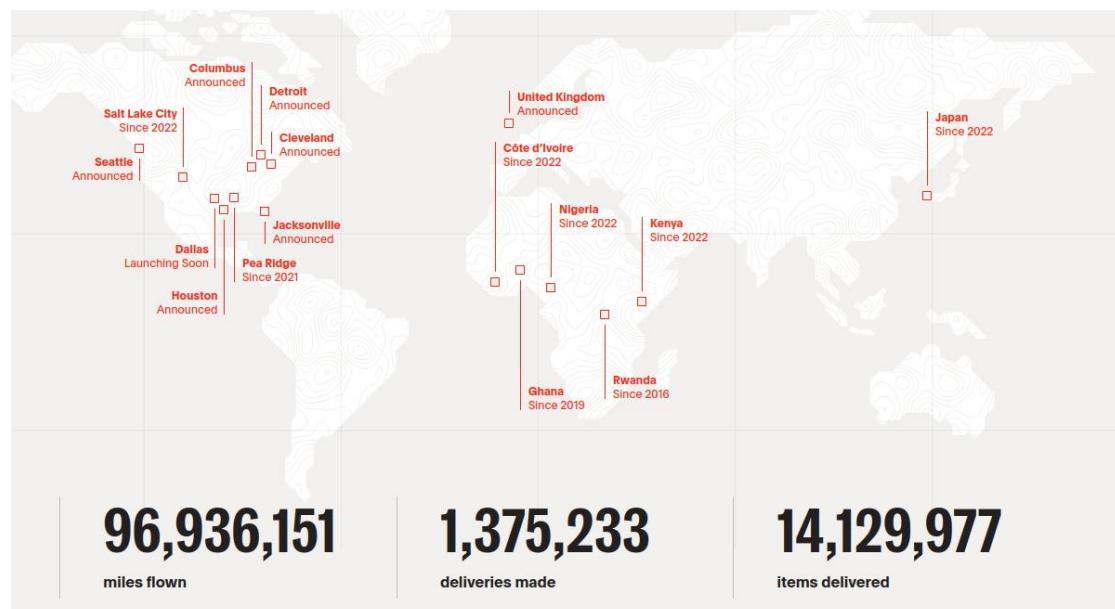


Figura 4 - Locais onde a empresa Zipline opera.

2.3 Tecnologias Utilizadas

A escolha do ROS2 e do Gazebo para este projeto justifica-se pela sua ampla aceitação e suporte na comunidade de robótica. O ROS2 (*Robot Operating System*) oferece uma infraestrutura modular e escalável que facilita a integração de algoritmos e sensores, enquanto o Gazebo permite uma simulação realista de robôs, sensores e ambientes. O ROS2 permite utilizar uma ferramenta gráfica denominada RViz (*ROS Virtualization*).

O RViz é uma ferramenta essencial no ROS2 que permite visualizar de forma simples e intuitiva os dados trocados entre os vários nós do sistema. Para além de mostrar informação como a posição do robô, sensores e mapas, o RViz também pode ser usado para interagir com o sistema, por exemplo, ao definir um objetivo de navegação. Por isso, é amplamente utilizado no desenvolvimento, teste e depuração de aplicações de robótica autónoma.

Comparativamente a outras opções, como o CoppeliaSim ou o Webots, o ROS2 e o Gazebo destacam-se pelo suporte extensivo a bibliotecas de código aberto e pela capacidade de integração direta com robôs físicos. Além disso, a utilização de um sistema operativo Linux garante um desempenho robusto e garante a compatibilidade com as ferramentas escolhidas, tornando-o ideal para projetos como este.

Inicialmente, considerou-se a utilização do ArduPilot para controlar o veículo terrestre no contexto simulado. Esta escolha deveu-se à existência de uma *bridge* já incluída no seu repositório oficial, pronta para ser integrada com o Gazebo, o que não se verifica com a principal alternativa, o PX4, cuja configuração é mais complexa.

No entanto, esta abordagem acabou por ser descartada devido a dificuldades na integração com o restante sistema. Optou-se assim por uma solução mais direta, baseada em ferramentas nativas do ROS2, o sistema de navegação Nav2, cuja implementação será abordada mais adiante neste documento.

Para partilha e melhor organização dos ficheiros do projeto, utilizou-se a ferramenta Github, repositório do projeto [\[31\]](#).

3 Trabalho de Projecto

Para a implementação e descrição dos capítulos seguintes, a simulação foi configurada e preparada de acordo com os passos descritos no anexo 3, sem a instalação do Ardupilot.

3.1 Objetivos do Trabalho desenvolvido

Para atingir os objetivos do trabalho já mencionados no início do documento, definiram-se objetivos mais simples e específicos, que visam abranger todos os tópicos:

Compreender o funcionamento das principais ferramentas utilizadas:

- Explorar em profundidade a estrutura e o funcionamento do middleware ROS 2 (*Robot Operating System*), nomeadamente a sua arquitetura baseada em tópicos, serviços, ações e transformações;
- Analisar o simulador Gazebo Harmonic, que permite a criação de mundos tridimensionais realistas com suporte a física, sensores e robôs complexos;
- Estudar a integração entre ROS2 e Gazebo, nomeadamente os mecanismos de comunicação entre plugins, sensores virtuais e controladores;
- Investigar o Sistema de navegação modular do ROS2, Nav2 (Navigation 2), responsável por permitir que o robô se localize, planeie e execute trajetórias de forma autónoma.

Simular o movimento autónomo de um veículo terrestre, utilizando o Nav2 em conjunto com o Gazebo e o ROS2. Para isso:

- Testar os planeadores globais e locais disponíveis no Nav2 e os seus respectivos algoritmos, como A*, Dijkstra, Theta*, DWB.
- Avaliar o desempenho destes planeadores com base em métricas como tempo de navegação, distância percorrida, número de manobras de recuperação, distância mínima aos obstáculos, entre outros;
- Comparar o comportamento do veículo em diferentes configurações de cenário, algoritmos e modelos.

Entender o processo de construção e descrição de modelos:

- Estudar como criar e configurar mundos no Gazebo com terrenos, obstáculos e elementos dinâmicos;
- Aprender a modelar veículos, definindo as suas estruturas físicas (elos e juntas) e sensores (ex.: LIDAR, câmaras), recorrendo a ficheiros URDF, XACRO ou SDF;
- Configurar corretamente os parâmetros de simulação, colisão, visualização e integração com o ROS2.

Recolher e analisar dados dos testes de navegação, de forma a:

- Validar o funcionamento do sistema de forma objetiva;
- Comparar diferentes algoritmos de planeamento;
- Tirar conclusões fundamentadas sobre o desempenho e comportamento do robô em múltiplos cenários.

Estabelecer uma interface externa para envio de objetivos ao robô, através da:

- Criação de um servidor web em Flask que permite ao utilizador selecionar um ponto no mapa (via interface gráfica simples);
- Envio desse ponto como goal para o sistema de navegação do Nav2, simulando uma integração remota entre operador e veículo.

3.2 Simulação com Gazebo Fuel

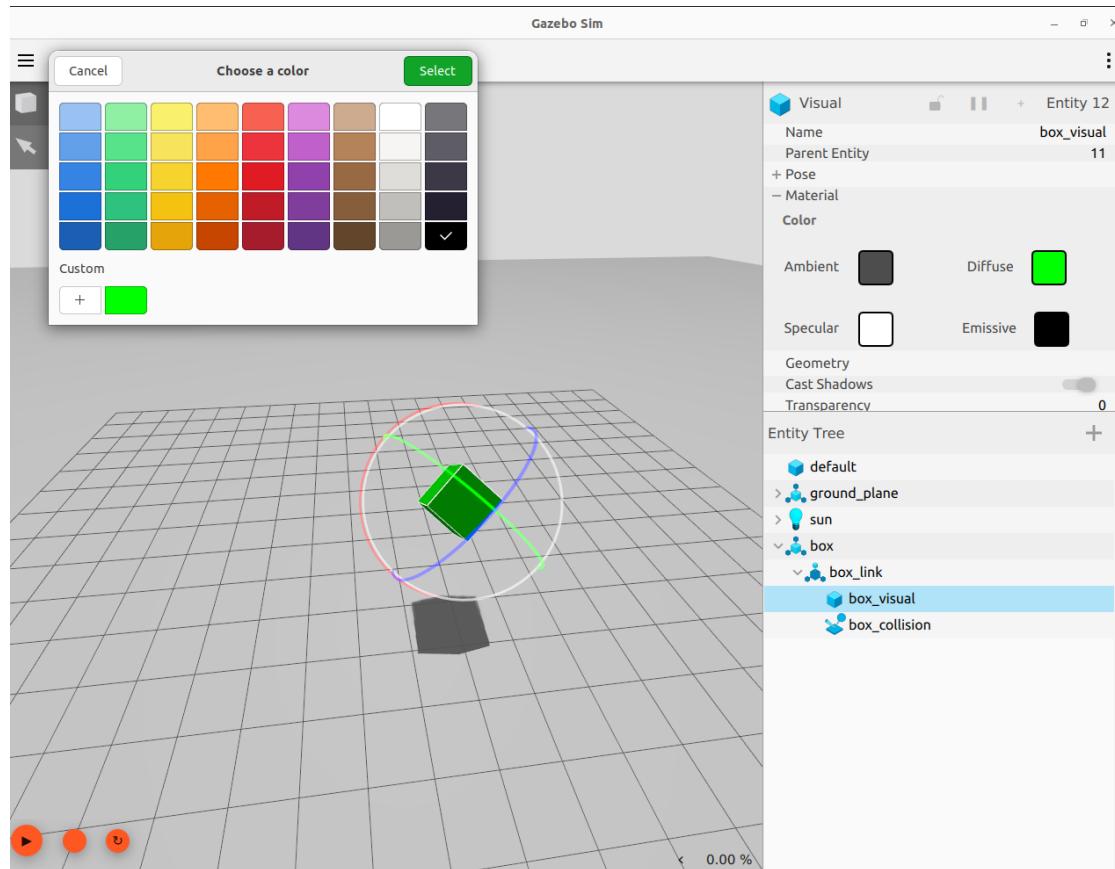


Figura 5 - Mundo simples gerado pelo Gazebo, Mundo de Cubos.

Embora a criação de um mundo de cubos, esferas e pirâmides seja interessante para um iniciante que esteja a começar a utilizar o Gazebo, neste projeto pretendemos aproximar-nos da realidade, e para tal, será necessário recorrer a modelos [15].

A Figura 5 mostra o mudo de cubos e formas, gerado pelo Gazebo. Como é pretendido mundos mais complexos e modelos, será necessário, inicialmente, recorrer ao uso de modelos e mundos pré-criados pela comunidade.

Esta plataforma (Gazebo Fuel) permite a consulta e utilização de modelos tanto de veículos ou robôs, como de mundos para utilizar os modelos. O Gazebo Fuel permite também partilhar os nossos modelos com outros utilizadores. Todos os modelos podem ser utilizados em simulação no Gazebo Harmonic, versão utilizada no projeto. Podemos comparar a plataforma a uma biblioteca online de modelos 3D.

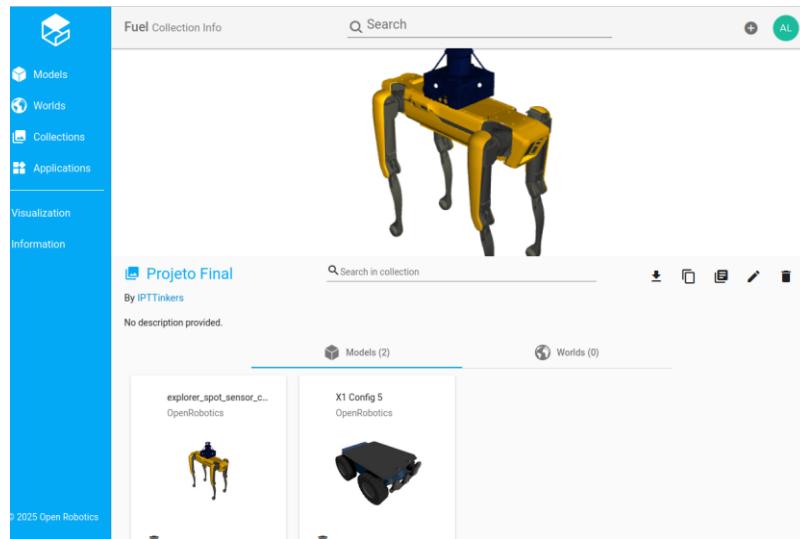


Figura 6 - Página com o nosso Perfil na Coleção “Projeto Final”.

Para a utilização dos modelos ser possível é necessária a criação de um perfil, onde os modelos podem ficar guardados. Neste projeto, os modelos e mundos utilizados poder-se-ão encontrar no perfil IPTTinkers[16], dentro da coleção “Projeto Final”

Através da Figura 6, é possível verificar que já foram subscritos dois veículos, mas para efeitos de testes apenas iremos utilizar um modelo. Para esta demonstração inicial, é então, utilizado o modelo “X1 Config5”.

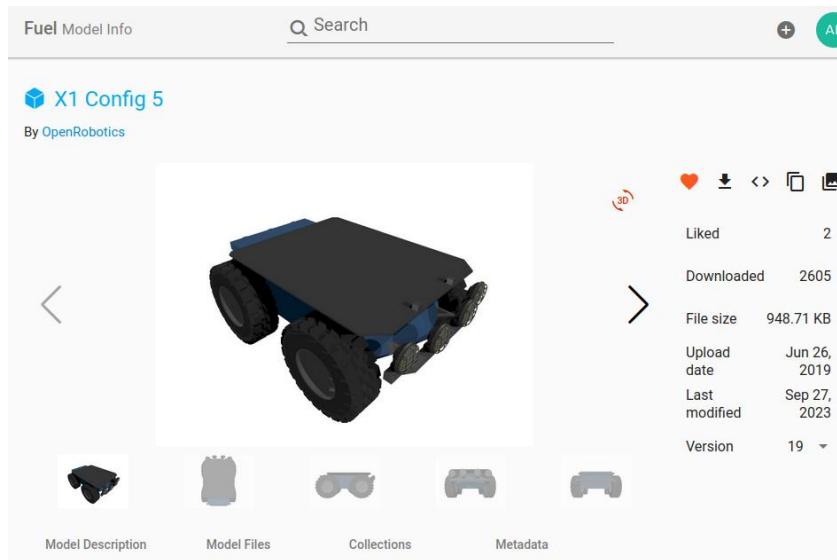


Figura 7 - Modelo X1 Config5.

A utilização do modelo escolhido (“X1 Config5”), Figura 7, ou de qualquer outro, em simulação no Gazebo pode ser feita de diversas formas. A solução adotada recorre ao uso do *resource spawner*, integrado no software Gazebo. Esta abordagem permite adicionar um modelo de forma mais simples ao mundo, sem que seja necessário especificá-lo diretamente num ficheiro. Além disso, o *resource spawner* permite importar modelos diretamente da coleção do utilizador no Gazebo Fuel, sem necessidade de os descarregar manualmente.

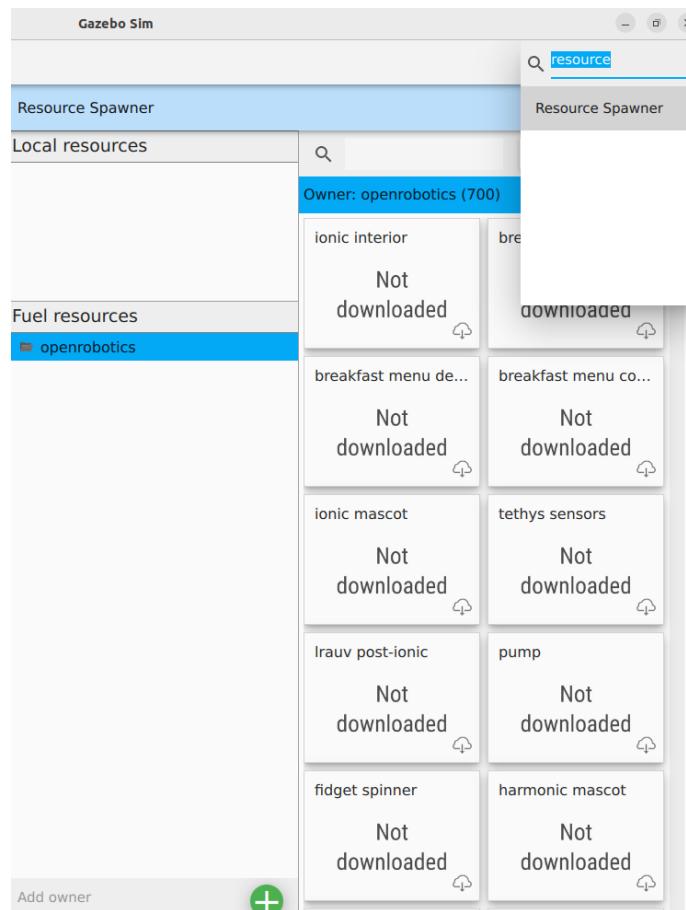


Figura 8 - Página no Software Gazebo que permite escolher “Resource Spawner”.

Após ser selecionada a opção *resource spawner*, Figura 8, é possível escolher um modelo a partir da coleção do utilizador. Como se pretende utilizar o modelo denominado “X1 Config5”, procede-se à sua pesquisa pelo nome e, em seguida, à sua seleção para inserção no mundo simulado.

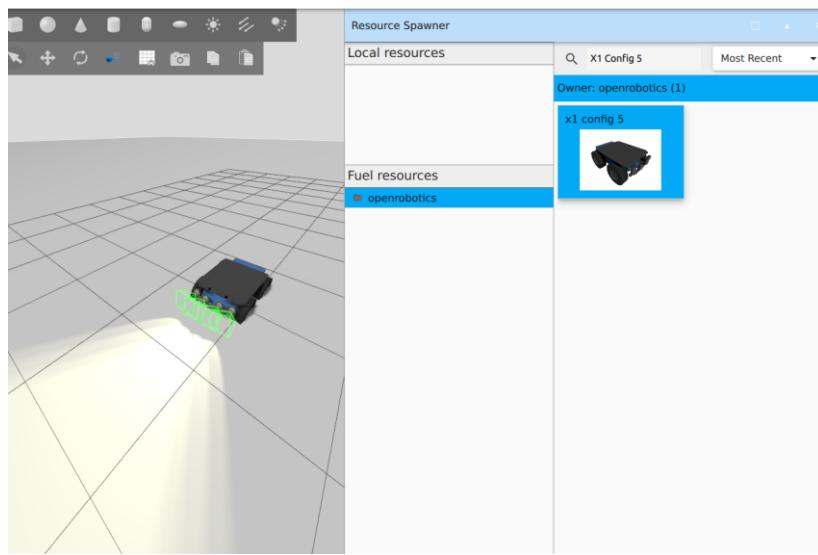


Figura 9 - Escolha do Modelo X1 Config5 e a sua inserção no mundo.

A Figura 9, mostra já o nosso modelo no mundo por defeito do Gazebo.



Figura 10 - Imagem do mundo escolhido.

A fim de visualizarmos o modelo escolhido num mapa sem ser o mapa aberto, por defeito, do Gazebo, procedemos à procura de um mapa na plataforma Gazebo Fuel. O mundo escolhido foi um mundo que retrata uma gruta.

A incorporação de um novo mundo no software é feita através de um método ligeiramente diferente daquele utilizado para os modelos. No caso dos modelos, é possível adicioná-los diretamente a uma coleção pessoal para facilitar a sua inserção no ambiente de simulação, no caso dos mundos é mais comum descarregar o ficheiro com

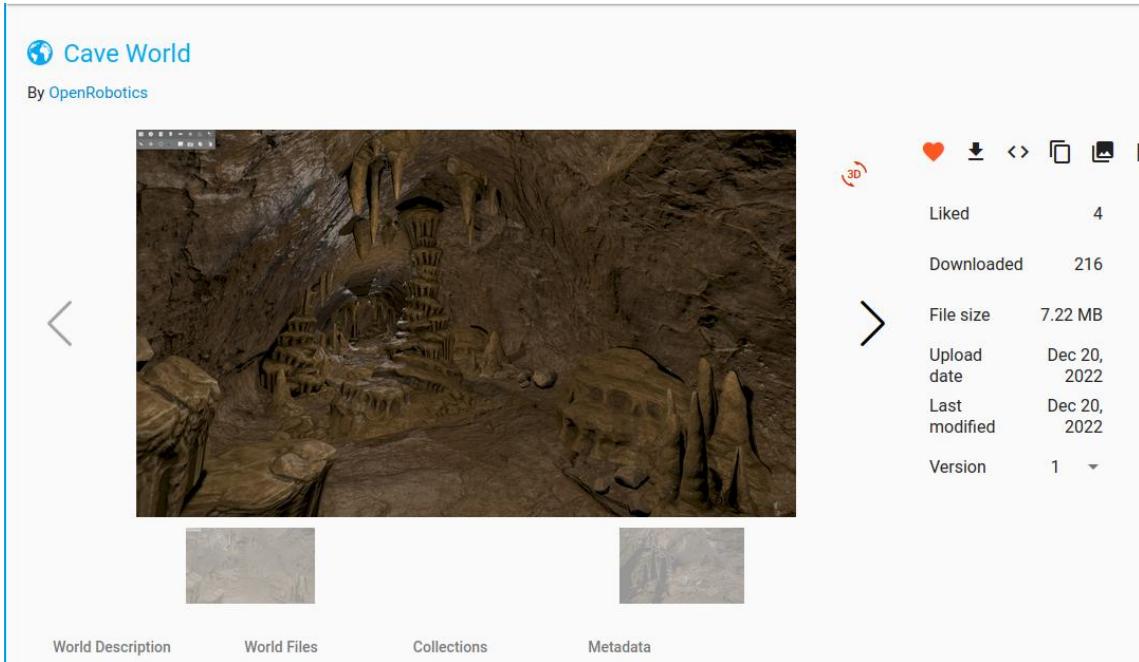


Figura 11 - Página no Gazebo Fuel do mundo escolhido

a extensão “.sdf” (*Simulation Description Format*) a partir da plataforma. Um ficheiro com esta extensão permite descrever modelos, mundos e sensores, sendo facilmente interpretado pelo software de simulação. O mundo escolhido está representado na Figura 10.

Por fim, tendo o ficheiro descarregado, pela página do Gazebo Fuel, Figura 11, a inserção desse mundo no Gazebo torna-se bastante simples. Apenas é necessário indicar ao Gazebo o local onde deve correr o ficheiro para carregar o novo mapa. Neste caso, como o ficheiro foi guardado na pasta “Downloads”, o comando a utilizar no terminal para iniciar o Gazebo com esse mundo é o seguinte:

❖ `gz sim --verbose 3 ~/Downloads/"Cave World"/cave.sdf`

```
ruben@ruben-HP-Laptop:~$ gz sim --verbose 3 ~/Downloads/"Cave World"/cave.sdf
[Msg] Gazebo Sim GUI v8.9.0
[Msg] Detected Wayland. Setting Qt to use the xcb plugin: 'QT_QPA_PLATFORM=xcb'.
[Msg] Received world [/home/ruben/Downloads/Cave World/cave.sdf] from the GUI.
[Msg] Gazebo Sim Server v8.9.0
Warning: Ignoring XDG_SESSION_TYPE=wayland on Gnome. Use QT_QPA_PLATFORM=wayland to run on Wayland anyway.
[Msg] Loading SDF world file[/home/ruben/Downloads/Cave World/cave.sdf].
[Msg] Downloading model [fuel.gazebosim.org/openrobotics/models/cave]
```

Figura 12 - Comando no terminal que permite iniciar o Gazebo no novo mundo.

Como resultado do comando anterior, e possível visualização na Figura 12, o Gazebo é iniciado e carrega automaticamente o mundo definido no ficheiro .sdf, apresentando no ecrã o mapa e o veículo, caso exista, como neste caso, pronto para ser utilizado na simulação.

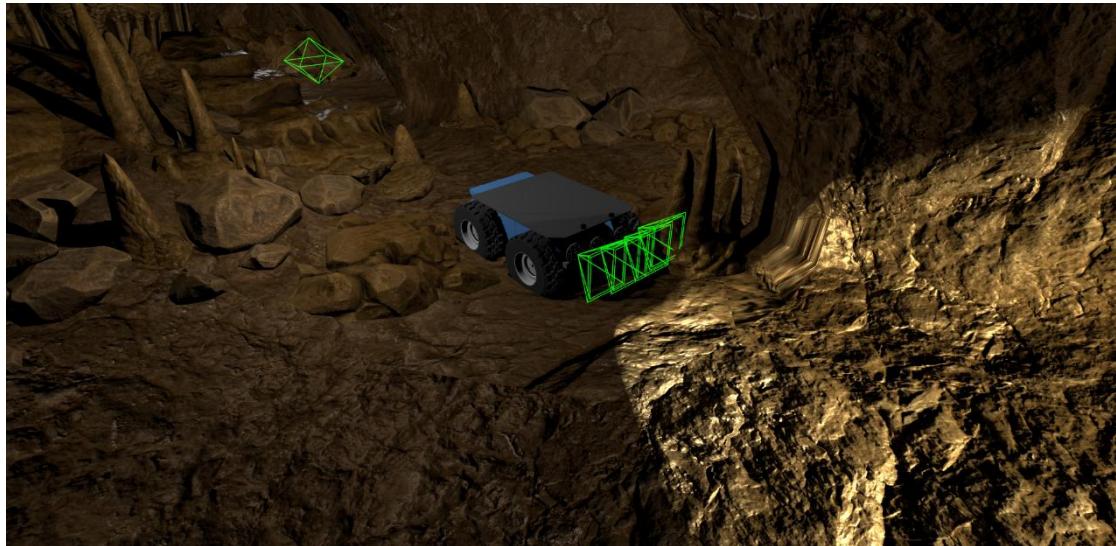


Figura 13 - Visualização do modelo no mundo.

A Figura 13, apresenta o nosso modelo no mundo escolhido. O modelo permite ao utilizador deslocar o veículo e movimentar-se pelo terreno do mundo.

3.3 Estrutura do modelo de um veículo

Desconstruir a estrutura física em componentes separados chamados links.

Definição de elos (*link*):

- Componente que se consegue mover separadamente dos outros componentes
- Componente em que simplesmente faz sentido ser separado dos outros (ex: Sensores).

Definição de juntas (*joints*):

- Efetua a conexão entre os elos.
 - Cada elo, sem contar com o primeiro, terá uma junta correspondente que diz a qual outro elo está conectado.
- Cada elo apenas terá um pai mas poderá ter vários filhos.

Tipos de juntas comuns:

- Rotacionais (*Revolute*), onde existe uma rotação com um ponto fixo de início e de fim;
- Contínuos (*Continuous*), onde existe rotação sem limitações.
- Prismáticos (*Prismatic*), movimento de translação linear.
- Fixos (*Fixed*), onde o elo filho não se mexe relativamente ao pai.

Syntaxe URDF (*Unified Robot Description Format*):

Um ficheiro URDF é escrito em XML, cuja estrutura requer que todas as tags estejam contidas numa única *root tag* (etiqueta raiz). No contexto do nosso projeto, esta etiqueta raiz é designada por `<robot>`, à qual é atribuído o atributo `name`, que identifica o nome do robô.

Dentro da tag `<robot>`, são definidas as várias componentes do robô, nomeadamente as tags elo (`<link>`) e as juntas (`<joint>`), que representam respetivamente os elementos físicos (partes rígidas) e as ligações articuladas entre esses elementos.

Numa tag de elo, é possível definir-se três características: *visual*, *collision*, *inertial*.

Visual: Isto é o que é observável no RVIZ e no Gazebo. Nela, existem três aspetos: Geometria (`<geometry>`) – onde podemos definir a forma do link (quadrada, esfera, cilíndrica...) ou pode-se definir um path para uma 3D mesh.

Origem (*<origin>*) – localização da geometria, de modo a que não fique centrada na origem do link por defeito.

Material (*<material>*) – resume-se à cor. Define apenas no RViz e não no Gazebo.

```
<visual>  
<geometry>  
<origin>  
<material>  
</visual>
```

Collision: é utilizado para as calculações das colisões físicas.

Novamente, pode-se definir a geometria, e a origem – como na tag visual.

```
<collision>  
<geometry>  
<origin>  
</collision>
```

Inertial: esta secção define as propriedades físicas que determinam como o elo responde a força.. Inclui:

Massa (*<Mass>*): Especifica a massa do elo.

Origem (*<origin>*): Posição do centro de massa em relação ao link.

Inércia (*<inertia>*): Matriz que descreve a distribuição da massa e como esta influencia a rotação do link.

```
<inertial>  
<mass>  
<origin>  
<inertia>
```

```
</intertial>
```

A *tag* juntas `<joint>` define a ligação entre dois elos (*links*) no modelo do robô, especificando como um elo está posicionado e como pode mover-se em relação ao outro.

De seguida, é apresentado um exemplo de uma *tag* de uma junta.

```
<joint name="arm_joint" type="revolute"> Define uma junta rotacional, denominada  
"arm_joint"  
  
<parent link="slider_link"/> - Elo pai da junta  
  
<child link="arm_link"/> - Elo filho que se move em relação ao pai  
  
<origin xyz="0.25 0 0.15" rpy="0 0 0"/> - Posição e orientação da junta no elo pai  
  
<axis xyz="0 -1 0"/> - Eixo de rotação da junta  
  
<limit lower="0" upper="1.5708" velocity="100" effort="100"/> - Limites físicos e  
dinâmicos  
  
</joint>
```

XACRO — XML Macros para URDF

XACRO (XML Macros) é uma ferramenta providenciada pelo ROS que facilita a criação de ficheiros URDF. Permite o uso de variáveis, macros, expressões e inclusão de ficheiros, tornando a descrição de robôs mais modular, reutilizável e fácil de manter, especialmente em projetos grandes ou complexos.

Para permitir o uso de XACRO num ficheiro URDF, é necessário adicionar um atributo especial à *tag* `<robot>`, indicando que o ficheiro será processado como XACRO e não como XML simples.

```
<robot xmlns:xacro="http://www.ros.org/wiki/xacro"> - o uso do atributo  
"xmlns:xacro" define o espaço XACRO, e permite utilizar comandos desta ferramenta.
```

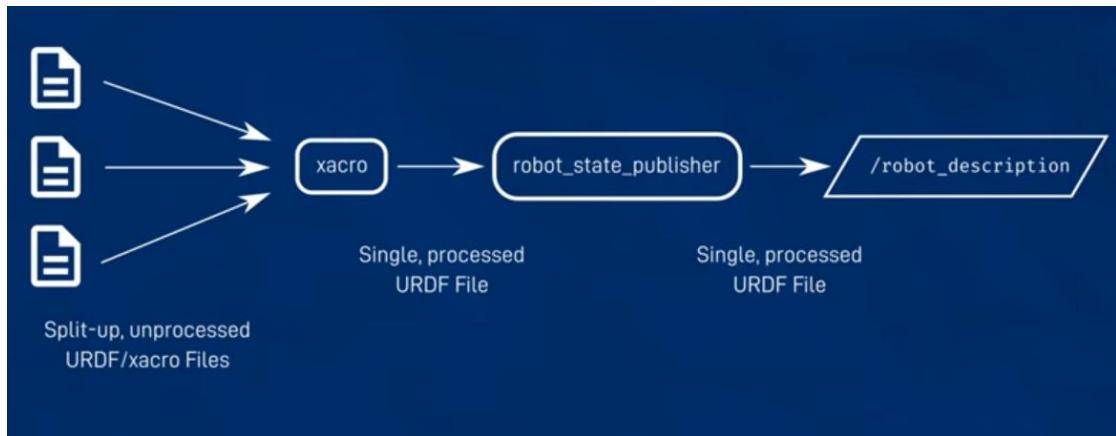


Figura 14 - Estrutura de um ficheiro XACRO.

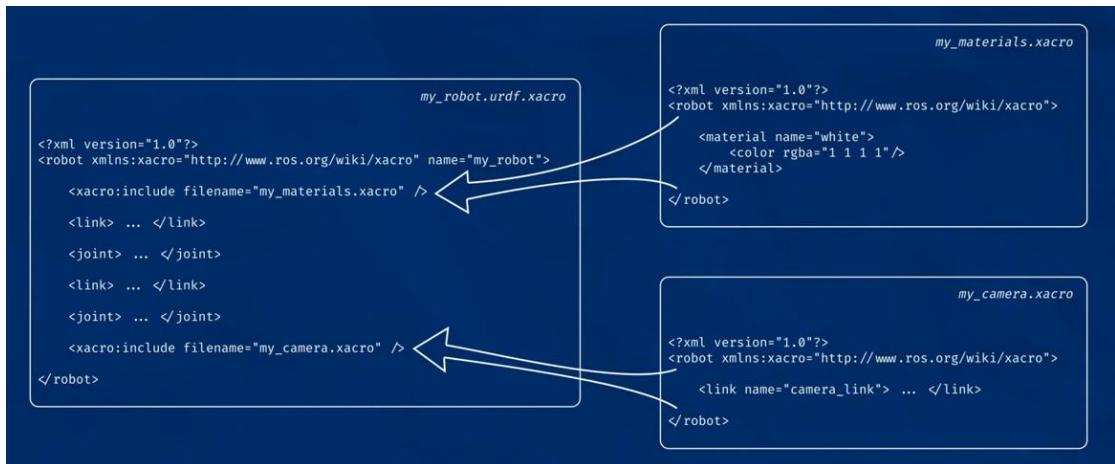


Figura 15 - Definição do ficheiro XACRO.

A construção da descrição do veículo pode ser organizada em várias camadas, utilizando ficheiros separados para cada componente e juntando-os progressivamente num ficheiro principal.

A Figura 14 apresenta uma visão esquemática da junção de vários ficheiros URDF, cada um representando uma parte específica do robô, que são depois reunidos num único ficheiro XACRO principal. Este processo modular facilita a manutenção e reutilização de componentes.

Já a Figura 15 mostra a estrutura final do ficheiro urdf.xacro, onde são incluídos os diferentes subcomponentes (também em XACRO) e definidas propriedades globais como dimensões ou materiais. Este ficheiro atua como o ponto central da descrição do robô, sendo aquele que será passado ao robot_state_publisher para publicação no tópico /robot_description.

Ambas as figuras anteriores, Figura 14 e Figura 15, os diagramas que apresentam podem ser consultados no decorrer de um vídeo disponível no Youtube[[14](#)].

3.4 Criação do modelo de um veículo e de um mundo

Para o desenvolvimento de ambos modelos de veículo e de um mundo, foi utilizada uma ferramenta gratuita de modulação 3D, Blender. O Blender para além de modelação 3D, permite criar e adicionar efeitos visuais a modelos, animações entre muitas outras funcionalidades. A utilização desta tecnologia é comum para a descrição de modelos para simulação, sendo também um pouco complexa e requer alguma prática e pesquisa.

- **Criação de um veículo**

Foi decidido e pensado que devido à dificuldade de manuseamento da aplicação, o veículo seria algo muito simples, sendo composto apenas por um “base link” que é a estrutura do veículo, um cubo, quatro rodas, corretamente identificadas “wheel (lado (esquerda ou direita) e profundidade (frente ou atrás))”, uma câmara “camera link”, formada por um cubo e um pequeno cilindro a fazer de lente e por fim, um cilindro no topo de um paralelipípedo que funciona como o sensor LIDAR “lidar link”. À estrutura do veículo foi adicionado um material que se assemelha a relva, a adição deste material deve-se à melhor visualização do veículo em ambiente de simulação, a troca para outro material é simples, devendo apenas ser necessário um ficheiro do formato png, com a “textura” do material pretendido. Cada um destes componentes foi exportado individualmente para um ficheiro com a extensão “dae”.

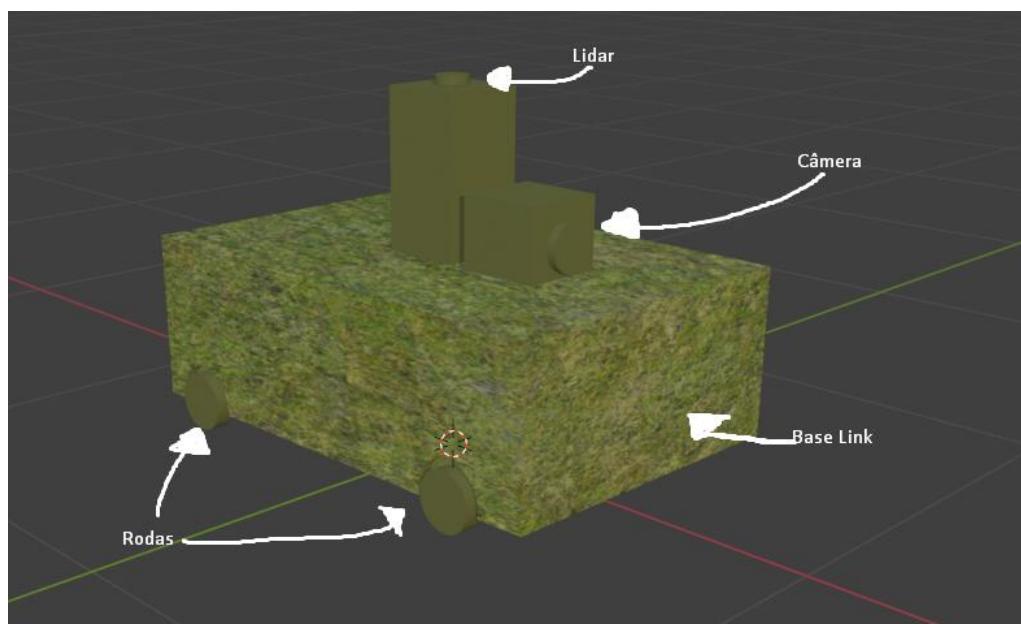


Figura 16 - Modelo Veículo

A Figura 16, mostra o resultado final da criação do veículo, identificando cada parte do mesmo. É possível ainda verificar o material colocado em todo o veículo, como mencionado anteriormente.

- **Criação de um mundo**

Na criação do mundo, houve duas ideias principais para mundos, um que teste as capacidades do veículo em termos de calcular o melhor caminho num mapa cheio de obstáculos, por outro lado houve a ideia de criar um mapa que representasse melhor as necessidades reais de um possível veículo autónomo. Assim, foi escolhido um cenário mais aproximado do mundo real.

No primeiro mundo, foi decidido então criar um mundo cheio de irregularidades como podemos verificar pela imagem seguinte.



Figura 17 - Modelo Mundo Irregular

Como é possível observar na Figura 17 o mundo está rodeado de obstáculos, uns maiores e outros menores, permitindo assim verificar o desempenho do veículo.

No segundo mundo, foi decidido a aplicação de um cenário mais realista, um corredor com várias aberturas de ambos os lados.

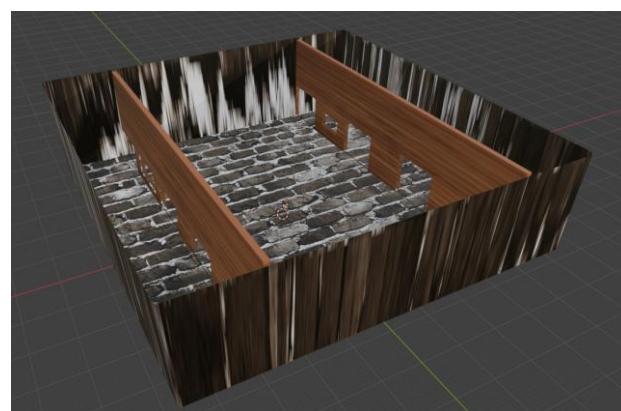


Figura 18 - Modelo Mundo Realista

A Figura 18, demonstra o mapa mais realista, onde é fácil visualizar o tipo de obstáculos. Como mencionado, o mundo tenta representar de forma mais realista um corredor, por exemplo de um armazém, e onde o veículo poderia de ter de se movimentar autonomamente entre cada lado do armazém.

3.4.1 Construção de um Cenário com um mundo e um modelo personalizado

3.4.1.1 Veículo

O modelo completo apresentado na Figura 16 corresponde à representação do veículo no software Blender. O próximo passo consiste em incorporar este modelo no ambiente de simulação Gazebo. Para isso, é necessário agrupar todas as partes exportadas individualmente a partir do Blender — como as rodas, a câmara, os sensores e a estrutura principal do veículo. Estas entidades são então reunidas e organizadas num único ficheiro .sdf (Simulation Description Format), que descreve a composição e o comportamento físico do modelo dentro do Gazebo.

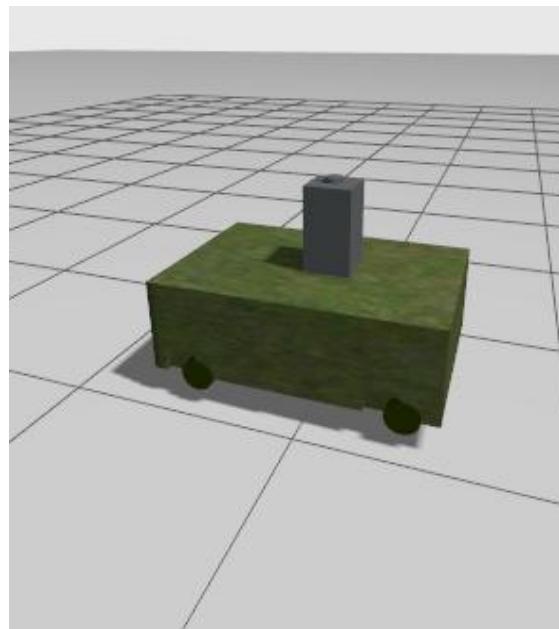


Figura 19 - Modelo do veículo em ambiente de simulação

A Figura 19 apresenta o veículo, modelado no Blender, já integrado no ambiente de simulação Gazebo. Nesta fase, o modelo foi importado com sucesso, incluindo os seus

componentes visuais, estruturais e a textura adicionada, permitindo a sua visualização e interação no mundo simulado.

3.4.1.2 Mundo

A incorporação do mundo criado, representado na Figura 18, no ambiente de simulação requer novamente a utilização de um ficheiro .dae exportado a partir do Blender. No entanto, neste caso, é utilizado um único ficheiro .dae que representa todo o cenário. Este ficheiro é referenciado dentro de um ficheiro .sdf, o qual o Gazebo consegue interpretar para visualizar e carregar corretamente o mundo na simulação.

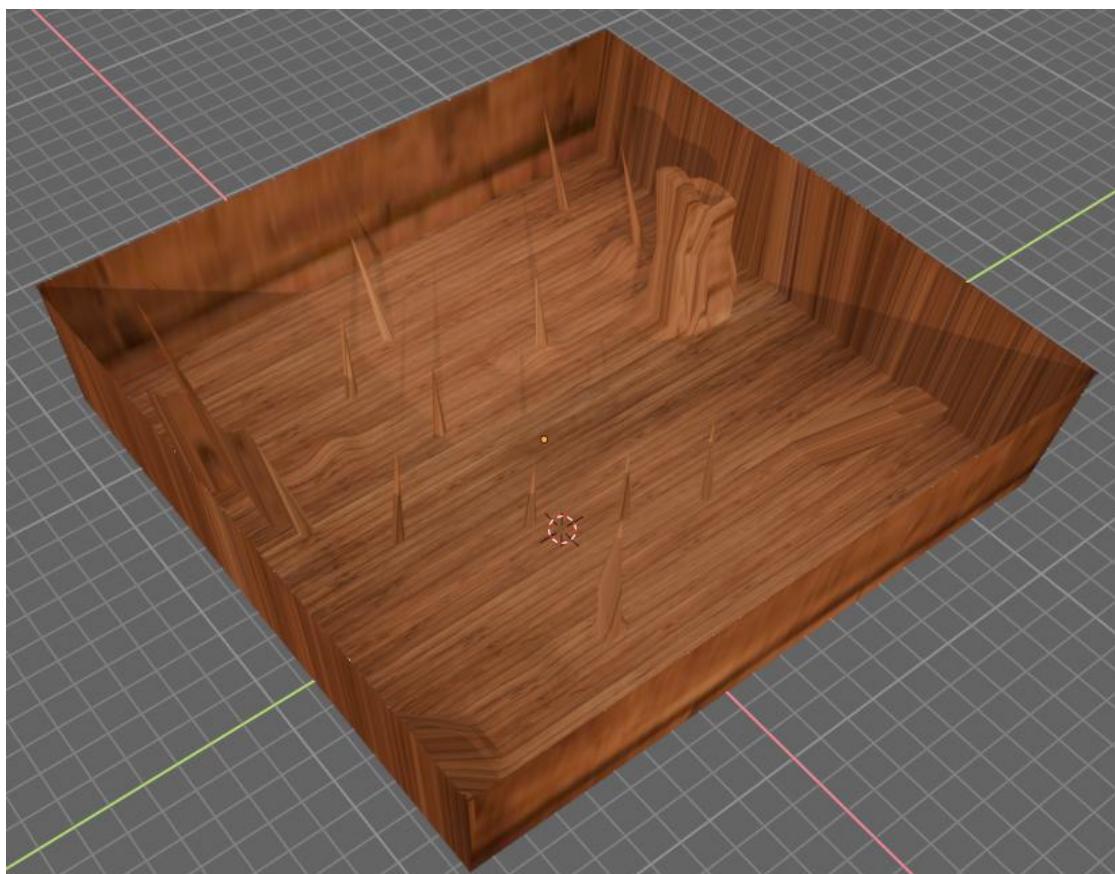


Figura 20 - Modelo do veículo em ambiente de simulação

A Figura 20 apresenta o mundo criado, já anteriormente introduzido na Figura 17, agora incorporado no ambiente de simulação Gazebo. Nesta fase, o cenário foi convertido com sucesso para o formato suportado pelo simulador, permitindo a sua visualização e interação no contexto da navegação autónoma.

3.4.1.3 Simulação juntando o modelo e o mundo dos capítulos anteriores

Para iniciar a simulação de forma prática e automatizada, foi criado um ficheiro de lançamento. Neste ficheiro, reunimos dois elementos fundamentais: o *spawn* do mundo e o *spawn* do veículo, previamente modelado e configurado, nos subcapítulos anteriores.

Desta forma, ao executar apenas este ficheiro, o mundo é automaticamente carregado no Gazebo e, de seguida, o veículo é posicionado corretamente dentro do cenário. Isto simplifica todo o processo de simulação, evitando a necessidade de executar múltiplos comandos em separado.



Figura 21 - Simulação utilizando os modelos criados (veículo e mundo)

Por fim, é possível verificar na Figura 21 o mundo e o veículo em simulação simultânea no ambiente Gazebo. Esta imagem demonstra que tanto o cenário como o modelo do robô foram corretamente integrados e estão a ser executados em conjunto, permitindo a realização de testes de navegação autónoma em condições controladas.

3.5 Navegação do Veículo

Neste capítulo é importante falar do sistema de navegação modular do ROS2, o Nav2 pois é este sistema que vai permitir o movimento autónomo do veículo.

O Nav2[17] é basicamente a evolução do sistema de navegação usada no ROS1, chamada *Navigation Stack*. Com o lançamento do ROS2, surgiu a necessidade de adaptar o conjunto de pacotes que constituía a *Navigation Stack*, para a nova arquitetura. Assim nasceu o Nav2, que veio substituir a stack antiga e que hoje é a principal solução para navegação autónoma de veículos no ROS2.

Este sistema permite que um veículo consiga deslocar-se autonomamente de um ponto A para um ponto B, mesmo em ambientes com obstáculos inesperados. O Nav2 não só calcula o caminho entre dois pontos, como também é capaz de reagir em tempo real a mudanças no ambiente. Para este cálculo do caminho a seguir ou para um novo cálculo caso um objeto seja detetado, são utilizados planeadores, globais e locais.

Através de sensores LIDAR, colocados no veículo, por norma numa parte superior, é transmitido ao veículo o “mapa” do espaço e através do algoritmo AMCL, o veículo consegue ter a percepção de onde se encontra, ou seja da sua posição atual. É necessário ao veículo conseguir entender a sua posição para conseguir calcular através dos planeadores um melhor caminho. Este caminho descrito pelos planeadores é com base nos custos vindos do mapa de custos, assim como o mapa de obstáculos, para que os planeadores possam utilizar os seus algoritmos sobre esses custos:

Planeadores:

- Globais: permitem calcular o caminho do ponto inicial ao ponto final (objetivo), nas primeiras versões apenas utilizava algoritmos de Dijkstra e A*, nas novas versões pode também utilizar Hybrid-A*, Theta* e State Lattice.
- Locais: permitem “calcular” o novo caminho, caso existam obstáculos no caminho calculado pelos planeadores globais, utilizam por norma algoritmos DWB (*Dynamic Window Approach*). Podem também utilizar novos algoritmos como, TEB (*Time Elastic Band*), RPP (*Regulated Pure Pursuit*), MPPI (*Model Predictive Path Integral*), e VP (*Velocity Planner*).

Algoritmos:

- AMCL (Adaptive Monte Carlo Localization) [20]: algoritmo de localização probabilística que permite ao veículo saber onde está num mapa conhecido com base nas informações dos sensores LIDAR. O AMCL é baseado num filtro de partículas (Monte Carlo), daí o nome do algoritmo que utiliza este filtro.

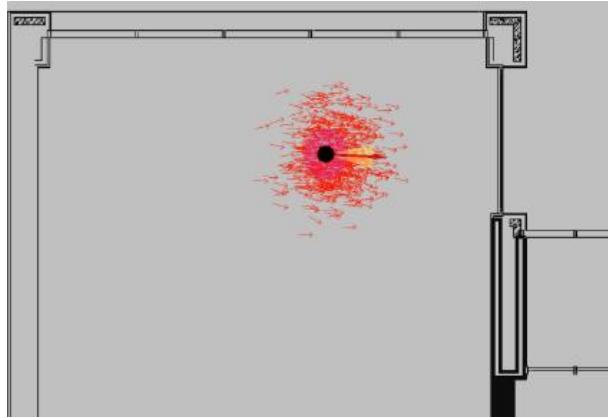


Figura 22 - Observação do veículo no RViz pelo algoritmo AMCL

A Figura 22, permite visualizar no RViz a posição do veículo, só é possível pois, o RViz subscreve o tópico “/amcl_pose” para representar o veículo. É neste tópico que o algoritmo AMCL coloca a sua posição.

- Dijkstra: algoritmo que encontra o caminho de custo mínimo. Garante a solução mais curta, mas como não utiliza heurística, pode ser mais lento, especialmente em mapas grandes. É ideal para aplicações em que a precisão é mais importante que a velocidade, como veículos que operam em ambientes altamente congestionados ou com zonas de custo muito variável.

Dijkstra's Algorithm

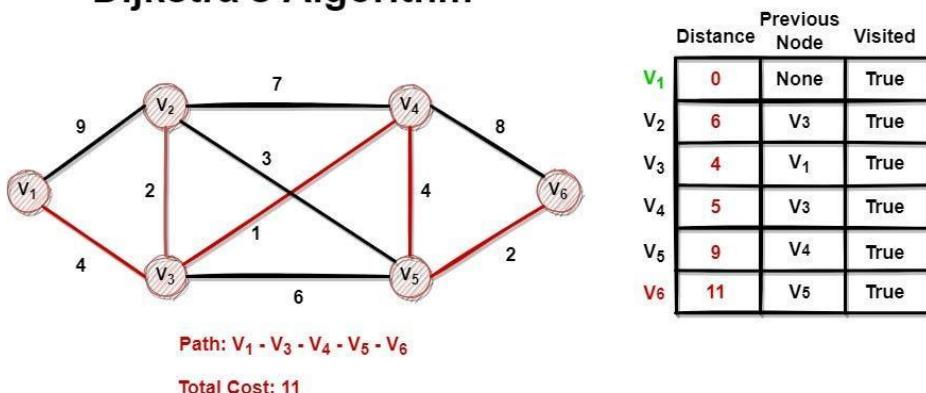
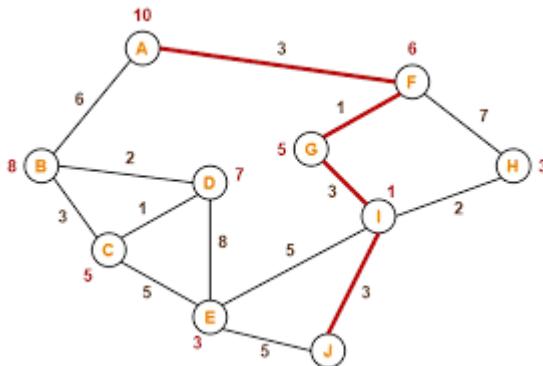


Figura 23 - Solução para um grafo utilizando o algoritmo Dijkstra

Na Figura 23, podemos visualizar uma solução para o garfo descrito utilizando o algoritmo de Dijkstra. Pela figura, é verificado o funcionamento deste algoritmo que procura o caminho de custo mínimo

- A*: algoritmo melhorado do algoritmo de Dijkstra, que acha um caminho ótimo, é mais rápido e utiliza heurística (estimativa do custo que falta até ao objetivo, ou seja, ajuda na decisão de caminhos a explorar). É um dos algoritmos mais usados em navegação autónoma por equilibrar bem entre tempo de execução e qualidade do caminho. É adequado para veículos móveis com movimentação simples e quando se pretende um planeamento eficiente em tempo real.



*Figura 24 - Solução para um grafo utilizando o algoritmo A**

Semelhante à Figura 23, também na Figura 24, é apresentada uma solução para de um grafo, mas desta vez, é utilizado o algoritmo A*.

- Hybrid A*: algoritmo que utiliza a eficiência do algoritmo base A*, mas adaptado a robôs ou no caso a veículos cujos movimentos são mais limitados, pela sua estrutura (rodar sobre o próprio eixo por exemplo), sendo ideal para planeamento de movimento em espaços apertados ou estradas

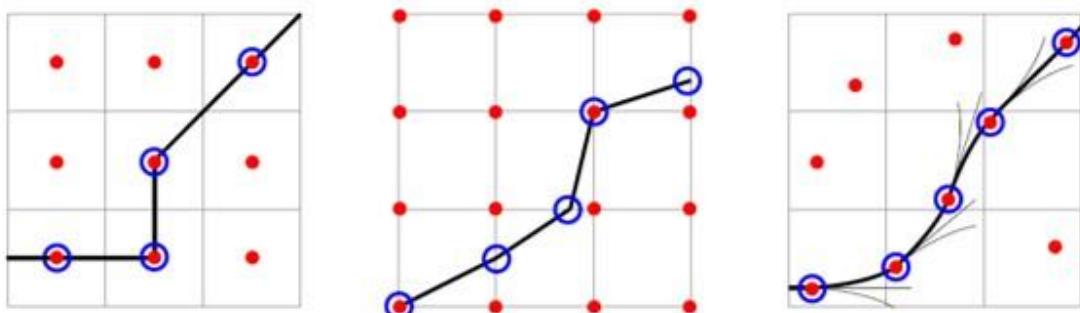


Figura 25 - Diferença entre algoritmo A e Hybrid A**

A partir da figura 25, é possível verificar as diferenças entre os algoritmos A* e Hybrid A*, enquanto que o A* (grafo mais à esquerda), apresenta trajetórias mais diretas, o grafo resultante do algoritmo Hybrid A*, revela trajetórias mais suaves, descrevendo curvas contínuas. Estas curvas são resultantes das restrições e limitações referidas em cima.

- Theta*: algoritmo variante do algoritmo A*, permite calcular caminhos mais diretos para o entre pontos, desde que exista uma visibilidade entre pontos direta. Este algoritmo caso a condição anterior descrita seja verificada, pode ser mais rápido que o algoritmo A*. Este algoritmo é usado com alguma moderação pois não respeita as condições do veículo (limitações e restrições de movimento) apesar de poder ser melhor que o A*. Muito útil quando existe uma grande área livre para o veículo se movimentar, ambientes abertos.

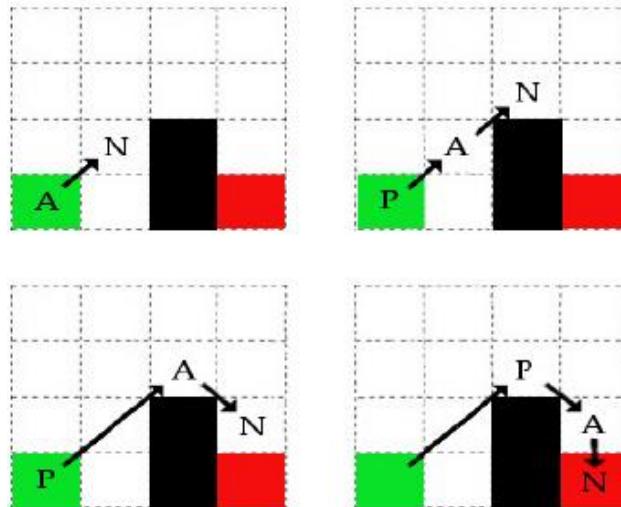


Figura 26 - Algoritmo Theta*

Na representação gráfica da Figura 26, encontramos 4 fases do movimento de um objeto desde a sua origem (célula a verde) até ao seu destino (célula a vermelho), utilizando o algoritmo Theta*. Na segunda fase, o objeto “salta” a célula representada com a letra “A”, pois tem visibilidade direta entre os pontos “P” e “N”, demonstrando assim, a característica deste algoritmo.

- *State Lattice* : algoritmo que permite numa primeira fase criar uma “teia” com todos os movimentos possíveis do veículo, ou seja, uma “teia” com todas as células que o veículo consegue se deslocar e tendo estas células, utiliza o algoritmo A* para escolher o melhor caminho. Como é um algoritmo que respeita as condições dos veículos, é um algoritmo muito mais usado em

simulações reais com veículos físicos e não em ambiente de simulação, por norma é também um algoritmo mais pesado computacionalmente.

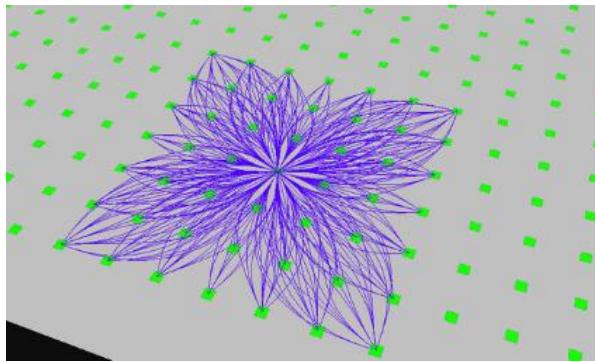


Figura 27 - Algoritmo State Lattice

A Figura 27 demonstra com alguma clareza, a “teia” de trajetórias possíveis criadas pelo algoritmo *State Lattice*, ainda antes de este tomar qualquer decisão quanto ao caminho a seguir.

- DWB (*Dynamic Window Approach*): algoritmo de planeamento local que, em tempo real, escolhe a melhor velocidade (linear e angular) para o veículo seguir com segurança e eficiência em direção ao objetivo. A cada ciclo de controlo, o algoritmo calcula várias combinações possíveis de velocidade, para cada uma das combinações, é simulado por quanto tempo o veículo segue esse comando, e estima a trajetória que ele faria. Cada uma dessas trajetórias é então avaliada por funções de custo.

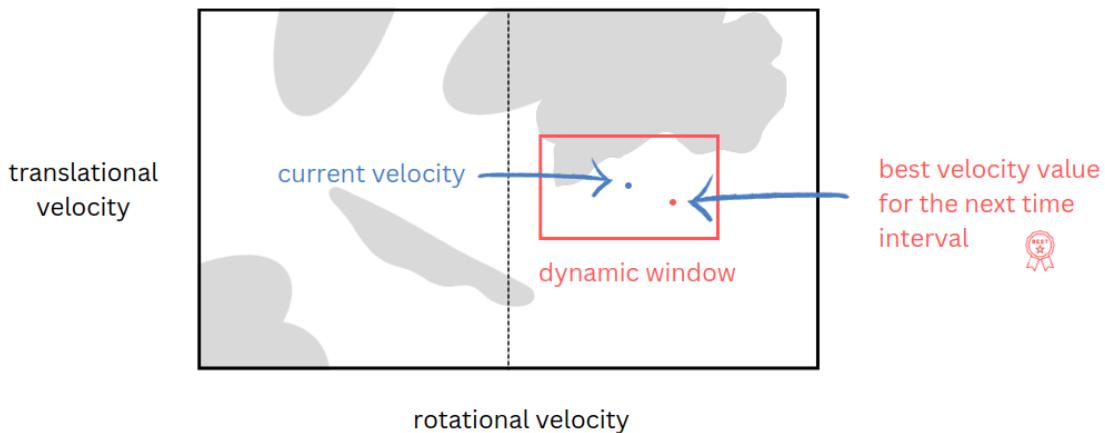


Figura 28 - Algoritmo Dynamic Window Approach

A Figura 28, mostra o espaço de velocidades lineares e angulares, onde o algoritmo DWB escolhe o próximo comando de velocidade dentro de uma janela dinâmica. A

melhor opção (em vermelho) é aquela que respeita as restrições dinâmicas e evita colisões.

- TEB (*Time Elastic Band*) : algoritmo que trata o caminho como se fosse uma mola elástica que pode ser distendida ou comprimida para evitar obstáculos. O algoritmo permite ajustar o caminho e o tempo que o veículo leva para percorrê-lo, garantindo que o movimento seja natural e seguro. É ideal para cenários onde o ambiente pode mudar rapidamente.

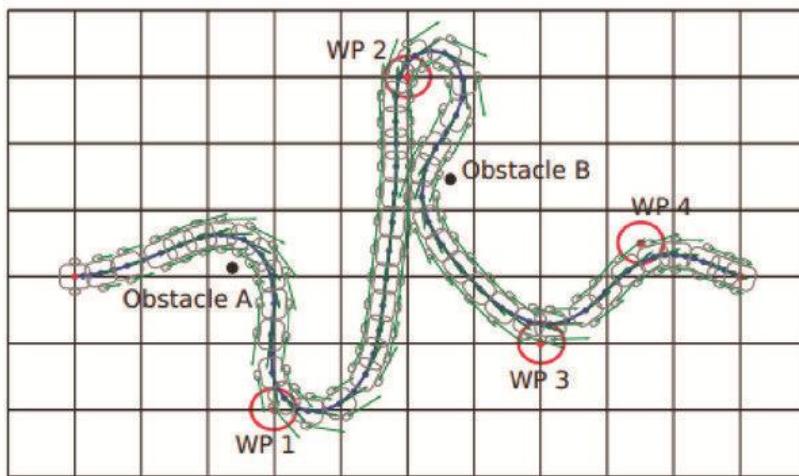


Figura 29 - Algoritmo Time Elastic Band

Na Figura 29, é visível uma trajetória característica do algoritmo TEB. É possível identificar facilmente este algoritmo pela presença de múltiplas trajetórias possíveis entre os *waypoints* (WP1–WP4), contornando obstáculos com recurso a bandas elásticas ajustáveis. Esta representação demonstra a capacidade do TEB em adaptar dinamicamente o caminho, ajustando curvas e evitando colisões de forma eficiente.

- RPP (*Regulated Pure Pursuit*) : algoritmo que escolhe um ponto no caminho e calcula como o veículo deve avançar para alcançá-lo. Além disso, regula a velocidade para o veículo andar mais devagar em curvas apertadas e acelerar em retas, garantindo um movimento mais seguro e confortável. É simples, eficiente e ótimo. Muito usado em veículos simples ou de baixa potência, devido à sua leveza computacional.

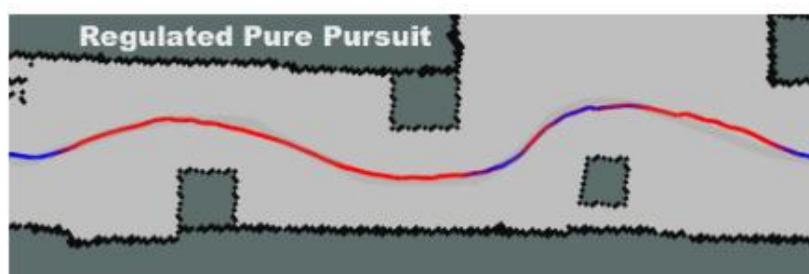


Figura 30 - Trajetória seguindo o algoritmo RPP

Através da Figura 30, é possível observar o comportamento do algoritmo *Regulated Pure Pursuit*, que segue suavemente uma trajetória entre obstáculos.

A trajetória (vermelha e azul) mostra como o veículo ajusta a sua velocidade e direção ao seguir um ponto-alvo à frente, mantendo uma navegação segura e fluida mesmo em ambientes com obstáculos próximos.

- MPPI (*Model Predictive Path Integral*) : algoritmo que permite visualizar caminhos possíveis. Para cada caminho calcula se é seguro e rápido, consoante a avaliação de cada caminho escolhe o melhor caminho. Ótimo para ambientes complexos ou ambientes com vários obstáculos em movimento.

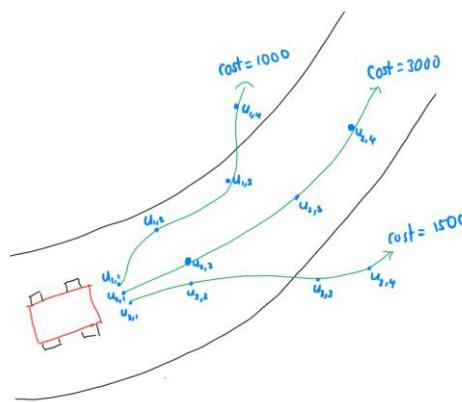


Figura 31 - Escolha de um caminho usando o algoritmo MPPI

Observando a Figura 31, vê-se o veículo a simular várias trajetórias possíveis. Cada uma destas trajetórias representa uma opção diferente de movimento que o veículo pode seguir. Como descrito anteriormente, o algoritmo procede agora a uma avaliação de cada trajetória, e escolhe a trajetória melhor.

-
- VP (*Velocity Planner*): não funciona diretamente como um algoritmo, mas sim como um plugin complementar, é utilizado em conjunto com os algoritmos anteriores, permite suavizar a velocidade do veículo para que este tenha um movimento mais controlado e estável. Muito útil em qualquer cenário onde se deseja estabilidade no movimento do veículo (travagens e acelerações bruscas).

Plugins [19]:

A solução para o uso destes algoritmos no Nav2 é feita através do uso de plugins. Existem diversos plugins para ambos os tipos de planeadores, cada plugin tem associado assim um ou mais algoritmos como poderemos verificar na lista seguinte:

- nav2_navfn_planner (NavFnPlanner) [21]: algoritmos como Dijkstra e A*. Utilizamos este plugin para projetos mais simples e quando não precisamos de simulações exaustivas.
- nav2_smac_planner (SmacPlannerHybrid) [23]: algoritmo Hybrid A*, utilizado para veículos com restrições e limitações físicas.
- nav2_smac_planner (SmacPlanner2D) [22]: algoritmo A*, utilizado para mapas dinâmicos, mais complexos, neste algoritmo utiliza heurística mais sofisticada. Tornando-se “mais” inteligente que o uso do A* no plugin anterior (NavFnPlanner).
- nav2_smac_planner (SmacLatticePlanner) [24]: algoritmo *State Lattice*, utilizado em ambientes mais realistas e em veículos com restrições e limitações físicas.
- nav2_theta_star_planner [25]: algoritmo Theta* utilizado para ter caminhos mais diretos que o A*.

Para Planeadores locais existem também alguns plugins:

- dwb_controller [26]: algoritmo DWB
- TEB Controller : algoritmo TEB
- regulated_pure_pursuit_controller [27]: algoritmo RPP
- mppt_controller [28]: algoritmo MPPI
- vp_controller: algoritmo VP

Mapas:

- Mapas de custo: mapa 2D em grelha (grid map) onde cada célula contém um valor de custo, este custo pode ser calculado devido a diversos fatores, dificuldade na passagem do veículo ou perigo, ficar preso, por exemplo. O mapa de custos é composto pelo mapa estático do espaço e do mapa de objectos (Figura 32).

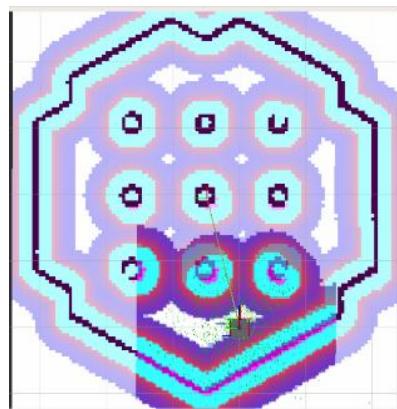


Figura 32 - Exemplo Mapa de custos

- Mapas de objetos: este mapa representa a camada dos obstáculos detetados em tempo real, com base em sensores como o LIDAR ou câmaras de profundidade. Os obstáculos são traduzidos com um custo máximo (254) no mapa de custos, esta informação mostra que é impossível um veículo passar por aquela célula. Aos objetos são ainda colocados uma camada “insuflada”, para que os veículos não passem demasiado perto e que exista a probabilidade de o veículo ficar danificado ou preso (Figura 33).

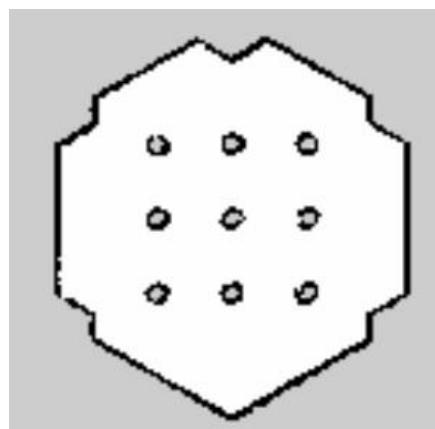


Figura 33 - Exemplo Mapa de obstáculos

3.5.1 Componentes do NAV2

Para implementar a navegação autónoma no veículo criado, é necessário configurar um ficheiro.yaml que satisfaça todas as necessidades da stack do NAV2 . Um stack de NAV2 contém os seguintes componentes:

map_server: Publica o mapa para que o veículo conheça o ambiente global.

amcl: Localiza o veículo no mapa através do LIDAR e através de odometria.

planner_server: Planeia um caminho desde o ponto inicial até ao destino pretendido.

controller_server: Segue o caminho planeado e evita obstáculos.

local_costmap: Representa o mapa de custos em redor do veículo.

global_costmap: Representa o mapa de custos global.

behavior_server: Gere comportamentos de recuperação, como por exemplo: rodar sobre si próprio, recuar... Para ajudar o veículo a sair de situações de bloqueio.

bt_navigator: Orquestra a navegação do veículo recorrendo a uma árvore de comportamentos para coordenar o planeamento, controlo e recuperação.

smoother_server: Suaviza o caminho planeado de modo a melhorar as curvas. Tem como intuito evitar que o veículo faça movimentos bruscos.

waypoint_follower: Permite seguir uma sequência de waypoints.

velocity_smoother: Suaviza os comandos de velocidade enviados ao veículo para deste modo evitar acelerações e travagens bruscas.

rviz: Permite ao utilizador visualizar o estado do veículo, dos mapas, trajetos, objetivos, etc.

robot_state_publisher: Publica a árvore transformações do veículo para que todos os frames (referenciais) sejam conhecidos.

3.5.2 Algoritmos de planeamento de caminhos

planner_server

O planner_server é o componente da stack do NAV2 responsável por definir o algoritmo do caminho a seguir até ao objetivo. O mesmo encontra-se definido no ficheiro “robo_camuflado.yaml” com os seguintes parâmetros:

```

planner_server:
  ros_parameters:
    expected_planner_frequency: 20.0
    use_sim_time: True
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_navfn_planner/NavfnPlanner"
      tolerance: 0.5
      use_astar: false
      allow_unknown: true
  
```

Figura 34 - Descrição do “planner_server” para Dijkstra.

planner_plugins: Define a lista de planeadores que queremos carregar no Nav2; neste caso, está apenas um, denominado “GridBased”.

plugin: Indica qual plugin vai realmente implementar o algoritmo de planeamento de caminhos; aqui, está definido como “nav2_navfn_planner/NavfnPlanner”, que corresponde ao planeador por defeito do NAV2. Este planner baseia-se no algoritmo de Dijkstra e A*.

use_astar: Caso a este parâmetro tivesse sido atribuído o valor “true”, utilizar-se-ia então o algoritmo A* e não o de Dijkstra.

Assim, a configuração da Figura 34, indica que estamos a usar o algoritmo do planeador global tradicional do Nav2, que por padrão utiliza Dijkstra.

```
planner_server:  
  ros_parameters:  
    expected_planner_frequency: 20.0  
    use_sim_time: True  
    planner_plugins: ["GridBased"]  
    GridBased:  
      plugin: "nav2_navfn_planner/NavfnPlanner"  
      tolerance: 0.5  
      use_astar: true  
      allow_unknown: true
```

Figura 35 - Descrição do “planner_server” para A.*

A Figura 35 apresenta a opção `use_astar`, um parâmetro já analisado anteriormente.

A presença deste parâmetro, em combinação com o plugin utilizado, permite confirmar que o algoritmo selecionado para o planeador global é o A*. Esta opção ativa o uso do algoritmo A* no plugin `nav2_navfn_planner`, em vez do padrão Dijkstra, ajustando assim o comportamento do planeamento global.

```
planner_server:  
  ros_parameters:  
    expected_planner_frequency: 20.0  
    use_sim_time: True  
    planner_plugins: ["GridBased"]  
    GridBased:  
      plugin: "nav2_theta_star_planner/ThetaStarPlanner"  
      tolerance: 0.5  
      allow_unknown: true  
      how_many_corners: 8  
      w_euc_cost: 1.0
```

Figura 36 - Descrição do “planner_server” para Theta.*

Por outro lado, a Figura 36 apresenta a configuração necessária para a utilização do algoritmo Theta*, proveniente do plugin já abordado no subcapítulo anterior, o `nav2_theta_star_planner`.

Este plugin implementa o algoritmo Theta*, permitindo gerar trajetórias mais diretas ao considerar linhas de visão livres entre os nós do mapa, otimizando o percurso em relação ao A* tradicional.

3.6 Navegação Autónoma com o Turtlebot3

Entre os veículos disponíveis para download na página do gazebo fuel, o grupo não se deparou com nenhum que utilizasse sensores. Assim, optou-se por utilizar o turtlebot3[18], que se trata de um modelo concretizado por uma empresa colaboradora da Open Robotics (a organização que desenvolveu o ROS), que incorpora um sensor LIDAR e uma câmara.

Para tal efetuou-se o seguinte comando:

- sudo apt install -y ros-humble-turtlebot3*

No entanto, após testar uma simulação com o turtlebot3, verificou-se que os ficheiros não são compatíveis com a versão Harmonic do Gazebo. Assim, após alguma pesquisa, encontrou-se um repositório uma adaptação do turtlebot3 já compatível com a versão do Gazebo utilizada.

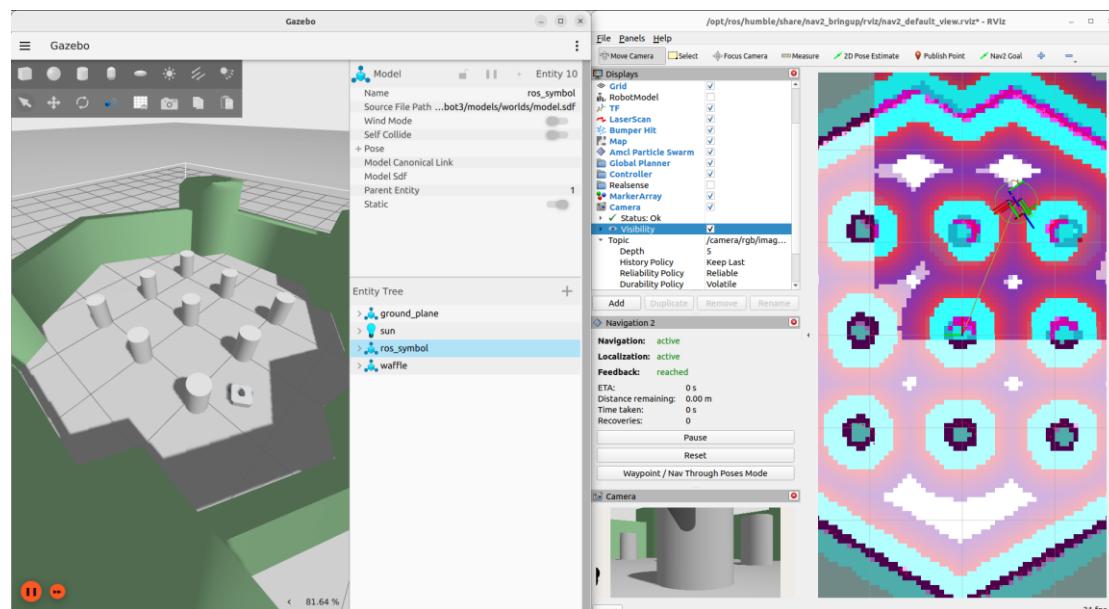


Figura 37 - Mundo Turtlebot3

Na Figura 37 é apresentado o mundo e o modelo utilizados no cenário do Turtlebot3, sendo este o ambiente base para os testes iniciais. Este cenário inclui o modelo oficial do Turtlebot3 e um mapa simples, frequentemente usado para validar navegação básica no ROS2 com o Nav2.

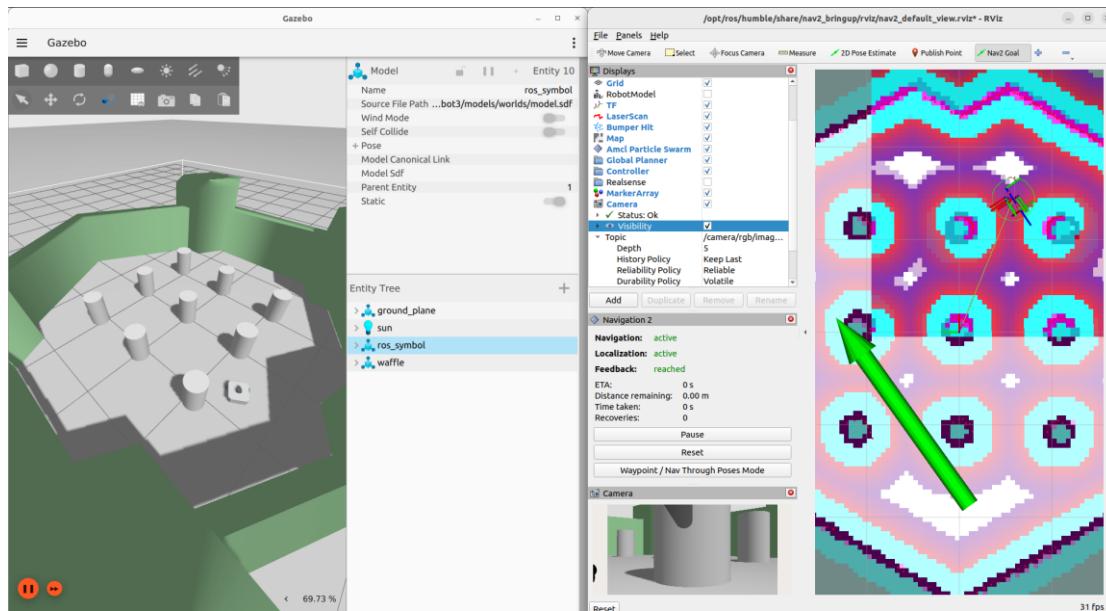


Figura 38 - Escolher um objetivo para o veículo.

Na Figura 38, é apresentado o ambiente de simulação com o RViz, onde é possível definir manualmente o objetivo do veículo utilizando o rato. Para isso, o utilizador seleciona a ferramenta “2D Nav Goal”, clica no mapa na posição desejada e arrasta na direção pretendida. Ao soltar o botão do rato, é colocada uma seta verde, que representa a posição e orientação final do veículo. Esta interação permite definir com precisão para onde o robô deve ir e em que direção deve ficar orientado ao chegar ao destino.

3.6.1 Cenário Turtlebot3

Para o veículo e para o mundo mostrados anteriormente foram colocados à prova 3 algoritmos dos planeadores globais. Foram escolhidos os 3 algoritmos mais comuns do Nav2, são eles, Dijkstra, A* com heurística simples e Theta*. Ou seja, foi utilizado apenas 2 plugins, NavFnPlanner e nav2_theta_star_planner. Foram realizados 10 testes para cada algoritmo do planeador global, sem alterar o algoritmo do planeador local que se encontra como DWB. A fim de se obter confiança nos testes, a posição inicial (Figura 38) e final (Figura 39) do veículo é sempre a mesma.



Figura 40 - Posição inicial.



Figura 39 - Posição final.

Dijkstra* (NavFnPlanner)						
Teste	Distância (m)	Tempo (s)	Colisão	Dist. min. Obstáculo (m)	Dist Max Obstáculo (m)	Observações (Manobras recuperação)
1	3,939	20,38	Não	0,317	4,723	
2	4,23	23,4	Não	0,321	4,82	
3	3,953	20,84	Não	0,315	4,757	
4	4,187	21,22	Não	0,334	4,828	
5	3,91	20,44	Não	0,278	4,778	
6	4,168	34,48	Não	0,242	5,071	1 manobra de recuperação
7	4,071	25,58	Não	0,299	4,906	
8	4,237	39,9	Não	0,226	5,076	3 manobras de recuperação
9	3,877	20,44	Não	0,277	4,752	
10	3,899	20,44	Não	0,299	4,722	
A* (NavFnPlanner)						
Teste	Distância (m)	Tempo (s)	Colisão	Dist. min. Obstáculo (m)	Dist Max Obstáculo (m)	Observações (Manobras recuperação)
1	3,969	24,46	Não	0,278	4,953	
2	4,133	24,04	Não	0,358	4,764	
3	3,91	22,58	Não	0,257	4,687	1 manobra de recuperação
4	4,001	20,5	Não	0,343	4,782	
5	3,898	22,16	Não	0,263	4,673	
6	4,152	23,82	Não	0,348	4,75	
7	4,103	35,32	Não	0,196	5,037	3 manobras de recuperação
8	4,15	21,41	Não	0,322	4,84	
9	3,913	20,82	Não	0,285	4,771	
10	4,217	22,32	Não	0,331	4,839	

Theta* (nav2_theta_star_planner)						
Teste	Distância (m)	Tempo (s)	Colisão	Dist. min. Obstáculo (m)	Dist Max Obstáculo (m)	Observações (Manobras recuperação)
1	3,932	18,16	Não	0,251	4,759	
2	3,995	18,5	Não	0,249	4,753	
3	4,059	18,14	Não	0,248	4,731	
4	3,949	19,16	Não	0,288	4,746	5 manobras de recuperação
5	3,949	18,26	Não	0,27	4,752	
6	3,942	18,24	Não	0,26	4,755	
7	3,975	18,46	Não	0,267	4,808	
8	4,019	18,6	Não	0,252	4,763	
9	3,952	18,3	Não	0,273	4,794	
10	3,918	17,86	Não	0,257	4,731	

Tabela 1- Resultados do Cenário do Turtlebot3.

Através dos dados da Tabela 1, é possível ser retiradas algumas conclusões relativamente aos algoritmos.

Algoritmo	Média de Tempo (s)
Dijkstra	24,71
A*	23,74
Theta*	18,37

Tabela 2 - Médias de tempo para cada algoritmo.

Observando os valores da Tabela 2, conclui-se que o algoritmo Theta* é o mais rápido entre os três utilizados, apresentando o menor tempo médio de navegação (18,37 segundos). Como referido na definição dos algoritmos, este resultado era expectável, o Theta* é uma variação do A* que tem a capacidade de gerar caminhos mais diretos. A utilização destes caminhos diretos resulta, numa execução do caminho mais rápida e eficiente.

A partir da Tabela 1, é possível observar que, nos três algoritmos, a distância percorrida pelo veículo se mantém praticamente constante, com variações mínimas. Isto indica que, independentemente do algoritmo utilizado, o caminho gerado em termos de comprimento é semelhante.

Quanto às distâncias aos obstáculos, verifica-se que o algoritmo Theta* tende a gerar caminhos que passam ligeiramente mais próximos dos obstáculos, apresentando os menores valores de distância mínima entre os três algoritmos. Uma vez mais, tendo em conta a definição do algoritmo, o resultado é esperado, o algoritmo permite trajetórias mais diretas e menos restritas, o algoritmo Theta* aproveita melhor os espaços livres, mesmo que isso implique navegar mais perto de zonas potencialmente mais perigosas. Apesar das aproximações aos obstáculos, não foram registadas colisões em nenhum dos testes.

Apesar de não terem sido registadas colisões, verificaram-se manobras de recuperação durante os testes. Estas manobras indicam que o veículo, em alguns momentos, não escolheu inicialmente o melhor caminho e teve de corrigir a sua trajetória.. Todos os três algoritmos apresentaram manobras de recuperação, contudo num dos testes do algoritmo Theta*, foi registado o maior número de manobras, cinco (5) manobras. Nos outros algoritmos o número de manobras de recuperação por teste é significativamente menor. Esta diferença sugere que, apesar de o Theta* ser mais rápido, pode ser menos estável ou exigir mais ajustes durante a navegação. Este comportamento está alinhado com as características do Theta*, que prioriza caminhos mais diretos e potencialmente mais agressivos, em contraste com os algoritmos Dijkstra e A*, que tendem a gerar trajetórias mais conservadoras e estáveis.

Por fim, ao analisar a segurança apresentada por cada algoritmo, verifica-se que, em nenhum dos testes, foram registadas colisões, o que demonstra que tanto o planeador global quanto o controlador local estão a funcionar corretamente para evitar obstáculos. No entanto, analisando os algoritmos individualmente, observa-se que o Theta* assume maior risco ao navegar mais próximo dos obstáculos, na sua tentativa de otimizar o tempo de percurso.

3.6.2 Cenário Mundo Irregular

Este cenário, é um cenário criado por nós e que incorpora o mundo já visto na Figura 16.

Este mundo só poderá ser utilizado para testes semelhantes aos testes realizados no outro cenário, quando conseguirmos ter o mapa de custos deste novo mundo. Para o Nav2 o mapa de custos e próprio mapa são descritos em dois ficheiros, um com a extensão .pgm que se refere à imagem do mapa em 2D, apresenta todos os obstáculos e por onde o veículo pode passar, e um ficheiro com a extensão .yaml, que fornece informações complementares, como a escala, origem, e os valores de custo atribuídos a cada célula do mapa.

Existe uma ferramenta denominada SLAM Toolbox, utilizada no ROS 2 para realizar SLAM (Simultaneous Localization and Mapping), ou seja, a construção de um mapa de um ambiente desconhecido ao mesmo tempo que o robô se localiza dentro dele. Esta ferramenta é especialmente útil quando se pretende gerar automaticamente o ficheiro com a extensão .pgm, e o respetivo ficheiro com a extensão .yaml, ambos mencionados anteriormente.

A utilização da SLAM Toolbox permite resolver o problema da criação dos ficheiros necessários para proceder à simulação neste cenário. Como esta ferramenta não faz parte do Nav2 por defeito, é necessário instalá-la previamente. Para isso, pode ser utilizado o seguinte comando:

- sudo apt install ros-humble-slam-toolbox

Esta ferramenta gera os ficheiros após o mundo ser manualmente mapeado, para isso, temos de correr o nosso cenário no Gazebo e no RViz. Tendo a simulação a correr, é necessário apenas executar o ficheiro do SLAM da ferramenta, rapidamente no RViz conseguimos observar o mapa a ser construído.

O veículo utilizado para o mapeamento, é um veículo muito semelhante ao mostrado no cenário anterior, onde tem também uma câmara e sensores LIDAR (sensores que permitem realizar o mapeamento).

Os valores lidos pelos sensores do veículo são publicados em três tópicos principais.

O tópico /scan contém os dados provenientes do sensor LIDAR, representando as leituras de distância aos obstáculos no ambiente.

Já o tópico /odom fornece a odometria do robô, ou seja, a estimativa da sua posição e orientação com base nos seus próprios movimentos.

Por fim, o tópico /tf publica as transformações entre diferentes frames de coordenadas, permitindo que o sistema mantenha uma noção espacial coerente e atualizada da posição do veículo.

Estes três tópicos são fundamentais tanto para o funcionamento correto da SLAM Toolbox, durante a construção do mapa, como para a operação do Nav2, que utiliza estas informações para localizar o robô e planear a sua navegação de forma autónoma.

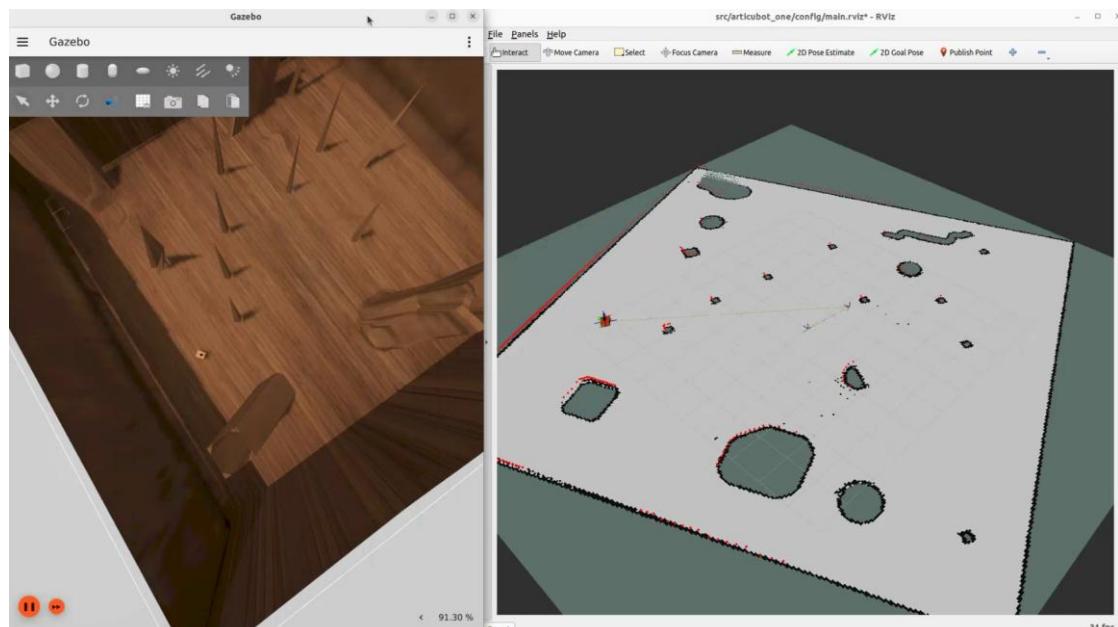


Figura 41- Mapeamento manual do mundo criado.

Realizado o mapeamento, Figura 41, torna-se necessário guardar os ficheiros gerados automaticamente durante o processo. São então guardados os dois ficheiros essenciais já mencionados anteriormente: o .pgm, que representa graficamente o mapa, e o .yaml, que contém a configuração associada.

A SLAM Toolbox, uma vez instalada, disponibiliza um painel no RViz que facilita esta operação. Através desse painel, é possível guardar os ficheiros de forma rápida e intuitiva, sem necessidade de recorrer à linha de comandos.

Com os ficheiros do mapa devidamente guardados, torna-se possível utilizar o novo cenário para a execução de testes. Isto significa que o mundo criado passa a estar totalmente compatível com o Nav2, permitindo ao veículo realizar navegação autónoma, bastando para isso indicar-lhe um ponto de destino.

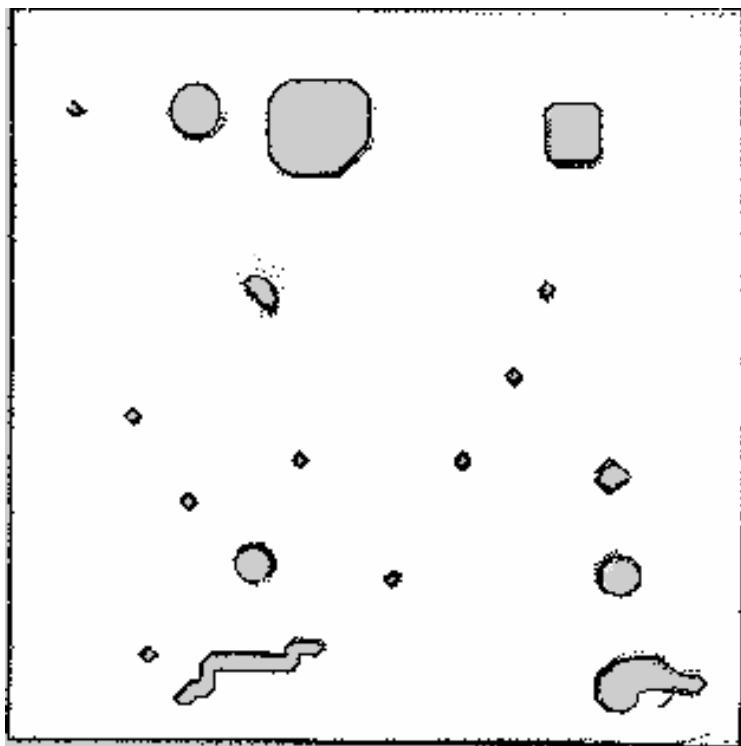


Figura 42 - Imagem .png convertida do ficheiro gerado .pgm.

A Figura 42 apresenta a imagem do mapa gerado pela SLAM Toolbox, convertida para o formato .png para facilitar a visualização e integração no documento. Esta imagem representa o ambiente em 2D, mostrando os obstáculos e as áreas navegáveis pelo veículo.

Juntamente com o ficheiro complementar em formato YAML, que contém informações sobre escala, origem e custos atribuídos a cada célula, esta imagem é utilizada para gerar o mapa de custos que o Nav2 utiliza para planejar trajetórias e garantir a navegação autónoma. O mapa de custos pode ser visualizado no RViz, permitindo confirmar a correta interpretação do ambiente pelo sistema.

À semelhança do cenário anterior, também neste foram realizados testes com o objetivo de verificar o desempenho do veículo no novo mundo. Neste cenário, foram testados dois algoritmos: o algoritmo A*, disponibilizado pelo plugin “NavFnPlanner”, e o algoritmo Theta*, pertencente ao plugin “nav2_theta_star_planner”.

As posições inicial (Figura 43) e final (Figura 44) foram previamente definidas e mantidas constantes ao longo de todos os testes, garantindo assim a fiabilidade dos resultados obtidos.

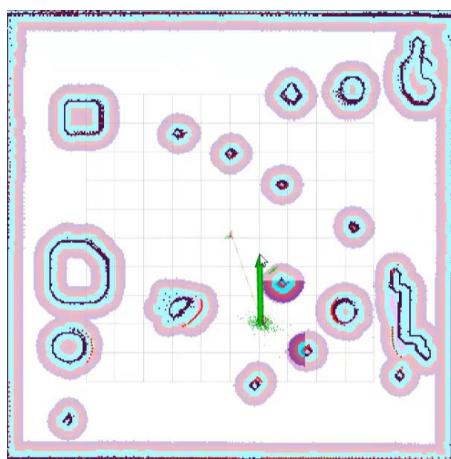


Figura 44 - Posição inicial do cenário Irregular

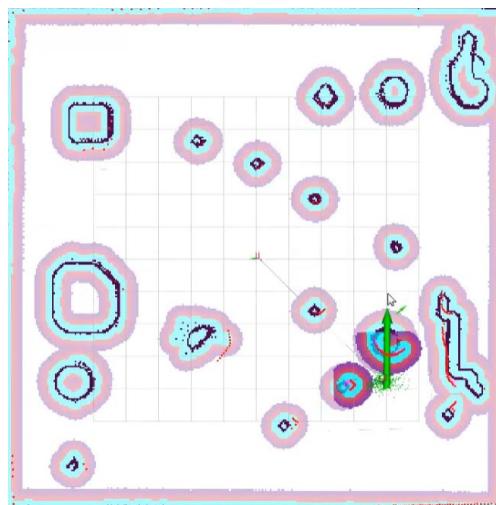


Figura 43 - Posição final do cenário Irregular

A* (NavFnPlanner)						
Teste	Distância (m)	Tempo (s)	Colisão	Dist. min. Obstáculo (m)	Dist Max Obstáculo (m)	Observações (Manobras recuperação)
1	3,09	17,3	Não	0,588	16,261	
2	3,149	17,7	Não	0,567	16,307	
3	3,066	17,04	Não	0,572	16,187	
4	3,074	17,48	Não	0,519	16,795	
5	3,415	32,62	Não	0,576	16,513	1 manobra de recuperação
6	3,17	17,5	Não	0,584	16,267	
7	3,086	17	Não	0,6	16,253	1 manobra de recuperação
8	3,509	27,04	Não	0,574	16,731	
9	3,085	17,54	Não	0,558	16,274	
10	3,011	16,86	Não	0,588	16,255	
Theta* (nav2_theta_star_planner)						
Teste	Distância (m)	Tempo (s)	Colisão	Dist. min. Obstáculo (m)	Dist Max Obstáculo (m)	Observações (Manobras recuperação)
1	2,95	16,88	Não	0,531	16,254	
2	3,072	17,42	Não	0,542	16,324	
3	3,116	17,26	Não	0,531	16,298	
4	3,025	17,18	Não	0,529	16,239	
5	3,019	16,92	Não	0,536	16,208	
6	3,072	17,44	Não	0,555	16,23	
7	3,08	17,24	Não	0,53	16,249	
8	2,997	17,04	Não	0,541	16,179	
9	3,172	18,1	Não	0,538	16,296	
10	3,148	17,56	Não	0,533	16,278	

Tabela 3 - Resultados do cenário mundo Irregular

Uma vez mais através dos dados da Tabela 3 é possível tirar algumas conclusões dos algoritmos utilizados em cada plugin dos planeadores globais.

Algoritmo	Média de Tempo (s)
A*	19,21
Theta*	17,31

Tabela 4- Tabela com as médias de tempo de cada algoritmo.

Os resultados da Tabela 4 confirmam a principal característica do algoritmo Theta*, que consiste em gerar trajetórias mais diretas, permitindo uma ligeira redução no tempo necessário para completar o percurso. Embora, na maioria dos testes, os tempos de navegação sejam bastante semelhantes entre os dois algoritmos, nota-se que o algoritmo A* apresentou valores mais elevados sempre que foram necessárias manobras de recuperação, o que influenciou negativamente o seu desempenho temporal nesses casos específicos.

É possível concluir que, consoante o cenário em estudo, as diferenças de desempenho entre os algoritmos A* e Theta* podem variar significativamente. Embora o algoritmo Theta* tenha apresentado um tempo médio de navegação inferior nos testes realizados, essa vantagem não foi tão evidente quanto a observada no cenário anterior. Tal diferença pode estar relacionada com a configuração do mundo, uma vez que o primeiro cenário continha um maior número de pontos com “visibilidade direta” entre si, favorecendo o comportamento do algoritmo Theta*. Este tipo de cenário permite ao Theta* explorar trajetórias mais diretas, reduzindo o tempo de execução, enquanto em ambientes mais complexos e com menos visibilidade direta, essa vantagem tende a diminuir.

As distâncias percorridas pelos dois algoritmos são muito semelhantes, o que não justifica, nem permite, uma comparação significativa entre eles com base apenas neste parâmetro. No entanto, no que diz respeito à distância mínima aos obstáculos, esse fator já merece uma análise mais cuidadosa. Apesar das diferenças observadas serem também reduzidas, neste caso, qualquer pequena variação pode ser relevante, dado que afeta diretamente a segurança da trajetória dos veículos.

No caso verificado, o algoritmo A* manteve-se como o mais seguro, ao apresentar uma maior distância mínima média aos obstáculos, o que está em concordância com os resultados obtidos no cenário anterior.

Por outro lado, o algoritmo A* também apresentou valores ligeiramente superiores na média da distância máxima aos obstáculos. Este resultado vai ao encontro das definições teóricas de cada algoritmo, uma vez que o A* tende a gerar trajetórias mais conservadoras e afastadas dos obstáculos, em comparação com o Theta*, que privilegia caminhos mais diretos, mesmo que passem mais próximo de zonas potencialmente arriscadas.

Nenhum dos testes realizados registou colisões, mantendo-se consistente com os resultados do cenário anterior. Contudo, manobras de recuperação foram observadas apenas no algoritmo A*, sugerindo que, mesmo adotando uma abordagem mais direta, o Theta* consegue manter a estabilidade do percurso. Estes resultados indicam que o Theta* não sacrifica a segurança em favor da rapidez, conseguindo ser ao mesmo tempo eficiente e estável. Por outro lado, o algoritmo A* registou duas manobras de recuperação, e nesses testes verificou-se um aumento significativo do tempo necessário para concluir o percurso.

Fazendo um balanço geral entre os dois algoritmos no que diz respeito à segurança, podemos afirmar que o Theta*, ao apresentar uma média de distância mínima aos obstáculos ligeiramente inferior, tende a navegar mais próximo das zonas de risco. Esta característica confere-lhe menos margem de erro e, teoricamente, maior probabilidade de incidentes. No entanto, as distâncias registadas continuam suficientemente seguras para não comprometer a integridade da navegação. Já o algoritmo A*, com uma distância mínima média mais elevada, revela-se mais conservador, priorizando trajetórias mais afastadas de obstáculos, o que traduz uma navegação potencialmente mais segura. Quanto ao parâmetro das colisões, ambos os algoritmos demonstraram um desempenho exemplar, sem qualquer incidente registado ao longo dos testes.

3.7 Navegação no mapa através de um Servidor Flask

Para além da criação dos cenários, veículos e realização de simulações, foi também explorada uma funcionalidade adicional que tornasse o sistema mais interativo com o utilizador.

Este capítulo explora a ideia de existir um controlo remoto do veículo através de uma interface web. Considerou-se interessante esta ideia de permitir que um utilizador, a partir de qualquer dispositivo com acesso à internet, pudesse indicar um objetivo diretamente num mapa e que o veículo se deslocasse até esse ponto, essa trajetória pode ser vista em tempo real, tanto no simulador Gazebo como no RViz.

Para isso, foi desenvolvido um pequeno servidor web local em Python utilizando a biblioteca Flask [29]. Esta tecnologia permite criar aplicações web locais de forma simples e eficiente.

O funcionamento deste servidor reflete, a apresentação de um ficheiro HTML (*HyperText Markup Language*), neste ficheiro com o nome “index.html”, é apresentada uma figura, que é interativa com o utilizador, ao clicar numa posição do mapa, existem duas caixas de texto que são preenchidas com as coordenadas da posição escolhida, é também enviado um *popup*, a registar a ação do utilizador.

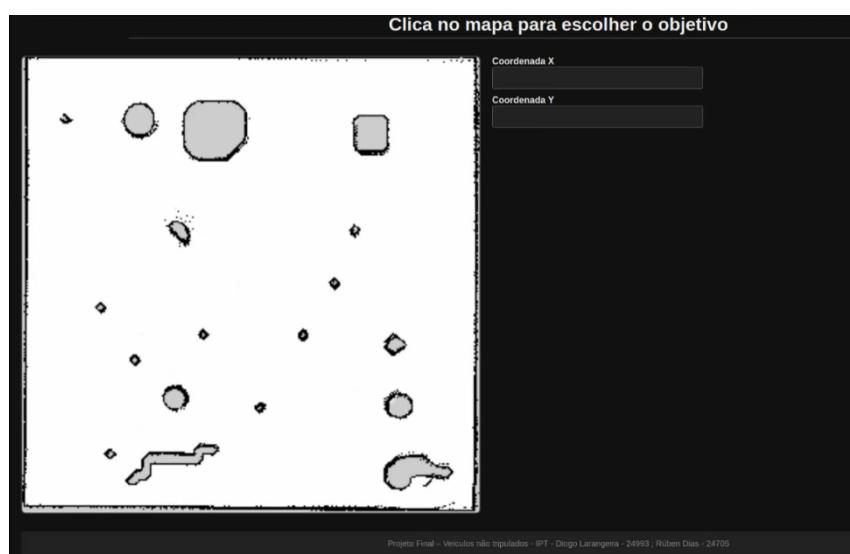


Figura 45 - Página definida no ficheiro HTML

A Figura 45, apresenta a página definida no ficheiro HTML, previamente mencionado.

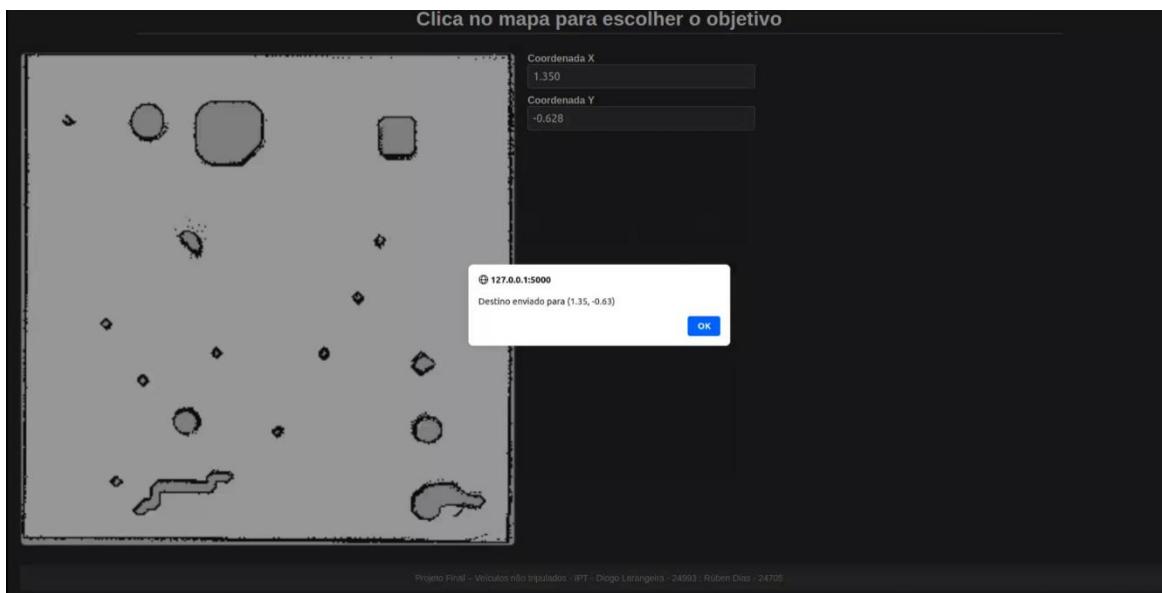


Figura 46 - Resposta da página à interação no mapa

A Figura 46, apresenta a resposta do *website* face à interação com o utilizador, nomeadamente após a escolha de uma posição no mapa. A resposta consiste no preenchimento das caixas de texto presentes na interface e no envio de um *popup* que confirma a posição enviada.

No ficheiro base do servidor denominado “app.py”, é então definido que vai apresentar a página apresentada nas Figura 45 e Figura 46, e a funcionalidade de enviar a posição para o veículo, para o ambiente de simulação. Foram criados mais dois *scripts* em python, um que permite a comunicação entre o ambiente de simulação e os dados recebidos, “send_goal.py”, este ficheiro é chamado pelo servidor para executar esta comunicação. E o ficheiro “send_goal_listener.py”, foi definido uma funcionalidade de apenas permitir enviar dados para o ambiente de simulação caso o veículo esteja sem objetivo, ou seja, esteja “parado”.

É fundamental compreender o funcionamento do sistema de navegação do ROS 2 e do Nav2 para que as mensagens sejam enviadas corretamente para os tópicos e *actions* adequados. Numa primeira abordagem foi identificado um tópico denominado “goal_pose”, no qual a aplicação foi testada, contudo, conclui-se que este tópico apenas recebe as coordenadas do objetivo, mas não provoca movimento por parte do veículo. Este tópico apenas atualiza apenas atualiza as coordenadas do objetivo no RViz.

Após investigação adicional, identificou-se que o que realmente inicia a navegação autónoma é a *action* “*navigation_to_pose*”. Esta *action* recebe as coordenadas de um objetivo e enviar para o planeador global em ação, o planeado dá então início à trajetória do veículo até ao objetivo. O *script* mencionado anteriormente, “*send_goal.py*”, envia as coordenadas para esta *action*, permitindo assim a visualização do deslocamento do veículo em tempo real nas ferramentas de simulação.

Por fim, considerou-se a possibilidade de disponibilizar este website a utilizadores externos, utilizando o Ngrok [30], uma ferramenta que cria túneis seguros entre uma aplicação local e a internet. O Ngrok gera um link acessível de qualquer lugar, o que tornaria possível controlar o veículo à distância a partir de qualquer *browser*.

No entanto, esta solução levanta algumas limitações importantes. Esta abordagem, indicava que seria necessário manter o dispositivo que executa o servidor Flask permanentemente ligado, o que não é ideal para sistemas de produção, e levanta também questões de segurança e fiabilidade, uma vez que se está a expor a máquina local à internet.

4 Conclusões

O trabalho realizado permitiu a concretização de um sistema de navegação autónoma para um veículo terrestre simulado, integrando as tecnologias ROS2 Humble, Gazebo Harmonic e Nav2. Consegiu-se com sucesso modelar e simular um veículo num ambiente 3D realista utilizando o software Blender, equipá-lo com sensores, e garantir que estes publicassem corretamente dados nos respetivos tópicos do ROS2. Esses dados foram visualizados em tempo real através da ferramenta RViz2, possibilitando a construção automática de mapas e a localização contínua do veículo no ambiente.

Além disso, foi desenvolvida uma interface web, com recurso a um servidor Flask, que permite o envio de objetivos remotamente para o veículo. Após o envio, é possível observar a trajetória do veículo em tempo real nas ferramentas de simulação já mencionadas.

Foram realizados testes com diferentes algoritmos de planeamento global da stack Nav2 — nomeadamente Dijkstra, A* e Theta* — tendo sido recolhidas métricas como tempo de execução, distância percorrida, proximidade a obstáculos, número de manobras de recuperação e número de colisões.

Com base nos resultados obtidos, conclui-se que, embora o algoritmo Theta* apresente vantagens em termos de rapidez ao gerar trajetórias mais diretas, o algoritmo A* oferece um melhor equilíbrio entre desempenho e estabilidade, com menos manobras de recuperação. Já o Dijkstra, apesar de mais seguro, demonstrou ser menos eficiente em termos de tempo. Assim, o algoritmo A* revelou-se a opção mais adequada no contexto testado. Contudo é necessário avaliar sempre o cenário apresentado, pois a escolha do algoritmo para cada cenário pode ser diferente devido ao número e tipo de requisitos bem como a topologia e arquitetura do mundo presente no cenário.

Este projeto demonstrou não só o funcionamento integrado das ferramentas ROS2, Gazebo e Nav2, como também o seu potencial para aplicações de navegação autónoma em veículos. A metodologia seguida permite ser replicada ou expandida para cenários mais complexos ou veículos reais no futuro.

Bibliografia

Todos os recursos enunciados nesta secção devem ser referidos no texto do relatório dentro de parêntesis retos.

1. <https://docs.ros.org/en/iron/index.html> (15 de Novembro 2024)
2. <https://www.anybotics.com/robotics/anymal/> (20 de Dezembro 2024)
3. <https://newatlas.com/robotics/chinese-police-amphibious-robot-ball/> (23 de Dezembro 2024)
4. <https://www.flyzipline.com/> (23 de Dezembro 2024)
5. <https://www.mdpi.com/2072-4292/15/13/3266> (24 de Janeiro 2025)
6. <https://docs.ros.org/en/iron/Tutorials.html> (28 de Janeiro 2025)
7. <https://www.theconstruct.ai/ros-projects-exploring-ros-using-2-wheeled-robot-part-1/> (31 de Janeiro 2025)
8. <https://ardupilot.org/dev/docs/ros2-gazebo.html> (31 de Janeiro de 2025)
9. <https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html> (30 de Março de 2025)
10. <https://ardupilot.org/dev/docs/ros2.html> (30 de Março de 2025)
11. <https://gazebosim.org/docs/harmonic/install> (30 de Março de 2025)
12. <https://ardupilot.org/dev/docs/ros2-gazebo.html> (30 de Março de 2025)
13. <https://gazebosim.org/about> (12 de abril de 2025)
14. <https://www.youtube.com/watch?app=desktop&v=K4rHgJW7Hg&t=0s> (1 de Junho de 2025)
15. <https://app.gazebosim.org/dashboard> (1 de Junho de 2025)
16. <https://app.gazebosim.org/IPTTinkers/fuel/collections/Projeto%20Final> (1 de Junho de 2025)
17. <https://docs.nav2.org/> (20 de Junho de 2025)

-
- 18. https://github.com/Onicc/navigation2_ignition_gazebo_turtlebot3 (20 de Junho de 2025)
 - 19. https://docs.nav2.org/setup_guides/algorithm/select_algorithm.html (20 de Junho de 2025)
 - 20. <https://docs.nav2.org/configuration/packages/configuring-amcl.html> (5 Julho de 2025)
 - 21. <https://docs.nav2.org/configuration/packages/configuring-navfn.html> (5 Julho de 2025)
 - 22. <https://docs.nav2.org/configuration/packages/smac/configuring-smac-2d.html> (5 Julho 2025)
 - 23. <https://docs.nav2.org/configuration/packages/smac/configuring-smac-hybrid.html> (5 Julho 2025)
 - 24. <https://docs.nav2.org/configuration/packages/smac/configuring-smac-lattice.html> (5 Julho 2025)
 - 25. <https://docs.nav2.org/configuration/packages/configuring-thetastar.html> (5 Julho 2025)
 - 26. <https://docs.nav2.org/configuration/packages/dwb-params/controller.html> (5 Julho 2025)
 - 27. <https://docs.nav2.org/configuration/packages/configuring-regulated-pp.html> (6 Julho 2025)
 - 28. <https://docs.nav2.org/configuration/packages/configuring-mppic.html> (6 Julho 2025)
 - 29. <https://flask.palletsprojects.com/en/stable/> (7 de julho de 2025)
 - 30. <https://ngrok.com/> (7 de Julho de 2025)
 - 31. <https://github.com/aspaceusername/ProjetoFinal> (Repositório Github do projeto)

Anexo 1 – Enunciado do projeto

Este projeto tem como objetivo desenvolver uma simulação realista de um sistema de controlo e planeamento de trajetórias para UAVs (Veículos Aéreos não Tripulados), integrando a plataforma de simulação Gazebo com o ArduPilot e o middleware ROS2. O foco será na criação de um ambiente simulado onde um ou mais UAVs podem realizar missões de voo autónomas, incluindo descolagem, navegação e aterragem, de acordo com trajetórias previamente planeadas ou geradas dinamicamente[5]. A simulação será responsável por testar algoritmos de planeamento de trajetórias, usando controladores baseados no ArduPilot, que será integrado ao ROS2 para facilitar a comunicação entre os diferentes módulos do sistema, tais como, sensores virtuais e algoritmos de navegação.

O uso do ArduPilot oferece uma plataforma robusta e amplamente utilizada em controlo de veículos aéreos não tripulados, proporcionando suporte a diversas funcionalidades, tais como controlo de atitude e velocidade.

O projeto também abordará a geração de trajetórias em ambientes com obstáculos, a fim de desenvolver estratégias eficientes para evitar colisões e otimizar percursos, além de explorar a capacidade do UAV em responder a mudanças no ambiente simulado. Pretende-se que o sistema desenvolvido possa ser uma base para futuras implementações em UAVs físicos, aproveitando a portabilidade das soluções baseadas em ArduPilot e ROS2.

Anexo 2 - Troubleshooting na instalação do software (ROS2, Gazebo)

Numa fase inicial, utilizaram-se as seguintes versões do software:

- Ubuntu Jammy
- Gazebo Fortress
- ROS2 Iron Irwini

No entanto, chegou-se à conclusão que estas duas versões são incompatíveis. Isto porque para tentar estabelecer uma comunicação entre o ROS2 Iron Irwini e o Gazebo Fortress, foi necessário criar uma bridge, que, embora permitisse controlar o veículo através do ROS2, não se verificou a correcta comunicação na direção oposta, do Gazebo para o ROS2, pois os dados adquiridos pelos sensores do robô não eram possíveis de visualizar no RViz.

Neste anexo, portanto, pretende-se documentar o trabalho realizado até este ponto onde o grupo posteriormente decidiu utilizar versões diferentes do ROS e do Gazebo como documentado no Anexo 3.

Para iniciar este projecto, foi necessário primeiro adquirir os conhecimentos necessários para realizar uma simulação, então para este efeito, seguiram-se os tutoriais disponibilizados na documentação oficial do ROS2[6].

Todo o trabalho foi desenvolvido num sistema ubuntu na versão 22.04.5 LTS

```
ruben@ruben-HP-Laptop:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.5 LTS
Release:        22.04
Codename:       jammy
```

Figura Anexo 1 - Versão Ubuntu utilizada (22.04.05 LTS)

1.1 Instalação do ROS2 Iron Irwini

1.1.1 Habilitação dos repositórios requeridos

Precisar-se-á de adicionar o repositório apt do ROS2 ao sistema[9]. O primeiro passo será verificar se o repositório Ubuntu Universe está ativado.

- sudo apt install software-properties-common
- sudo add-apt-repository universe

Agora, adiciona-se a chave GPG do ROS2 com o apt.

- sudo apt update && sudo apt install curl -y
- sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg

Em seguida, adiciona-se o repositório à lista de fontes (sources list).

- echo "deb [arch=\$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu \$(. /etc/os-release && echo \$UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null

Instala-se ferramentas de desenvolvimento:

- sudo apt update && sudo apt install ros-dev-tools

1.1.2 Instalação do ROS2:

Antes de iniciar a instalação, é importante correr os seguintes comandos no terminal:

- sudo apt update
- sudo apt upgrade

E por fim, corre-se o comando para instalar a versão iron do ROS2:

- sudo apt install ros-iron-desktop

1.1.3 Verificar a instalação:

Num terminal, faz-se *source* do ficheiro setup e inicializa-se um talker C++:

- source /opt/ros/iron/setup.bash
- ros2 run demo_nodes_cpp talker

Noutro terminal, faz-se *source* do ficheiro setup e inicializa-se um listener Python:

- source /opt/ros/iron/setup.bash
- ros2 run demo_nodes_py listener

Resultado esperado:

<pre>ruben@ruben-HP-Laptop:~\$ source /opt/ros/iron/setup.bash ros2 run demo_nodes_cpp talker [INFO] [1738179225.187426010] [talker]: Publishing: 'Hello World: 1' [INFO] [1738179226.187348239] [talker]: Publishing: 'Hello World: 2' [INFO] [1738179227.187462170] [talker]: Publishing: 'Hello World: 3' [INFO] [1738179228.187343591] [talker]: Publishing: 'Hello World: 4' [INFO] [1738179229.187605814] [talker]: Publishing: 'Hello World: 5' [INFO] [1738179230.187367563] [talker]: Publishing: 'Hello World: 6' [INFO] [1738179231.187392023] [talker]: Publishing: 'Hello World: 7' [INFO] [1738179232.187408484] [talker]: Publishing: 'Hello World: 8' [INFO] [1738179233.187355909] [talker]: Publishing: 'Hello World: 9' [INFO] [1738179234.187367014] [talker]: Publishing: 'Hello World: 10' [INFO] [1738179235.187374581] [talker]: Publishing: 'Hello World: 11' [INFO] [1738179236.187357147] [talker]: Publishing: 'Hello World: 12' [INFO] [1738179237.187423584] [talker]: Publishing: 'Hello World: 13' [INFO] [1738179238.187415455] [talker]: Publishing: 'Hello World: 14' [INFO] [1738179239.187423065] [talker]: Publishing: 'Hello World: 15' [INFO] [1738179240.187389212] [talker]: Publishing: 'Hello World: 16'</pre>	<pre>ruben@ruben-HP-Laptop:~\$ source /opt/ros/iron/setup.bash ros2 run demo_nodes_py listener [INFO] [1738179236.206547214] [listener]: I heard: [Hello World: 12] [INFO] [1738179237.189043105] [listener]: I heard: [Hello World: 13] [INFO] [1738179238.188483476] [listener]: I heard: [Hello World: 14] [INFO] [1738179239.189066983] [listener]: I heard: [Hello World: 15] [INFO] [1738179240.188437945] [listener]: I heard: [Hello World: 16] [INFO] [1738179241.188332001] [listener]: I heard: [Hello World: 17] [INFO] [1738179242.188942090] [listener]: I heard: [Hello World: 18] [INFO] [1738179243.188610063] [listener]: I heard: [Hello World: 19] [INFO] [1738179244.188894140] [listener]: I heard: [Hello World: 20] [INFO] [1738179245.188216215] [listener]: I heard: [Hello World: 21] [INFO] [1738179246.188895146] [listener]: I heard: [Hello World: 22] [INFO] [1738179247.188925569] [listener]: I heard: [Hello World: 23] [INFO] [1738179248.188501132] [listener]: I heard: [Hello World: 24] [INFO] [1738179249.188483272] [listener]: I heard: [Hello World: 25] [INFO] [1738179250.188634804] [listener]: I heard: [Hello World: 26] [INFO] [1738179251.189137424] [listener]: I heard: [Hello World: 27]</pre>
--	--

Figura Anexo 2 – Talker e Listener ROS, a estabelecerem comunicação

Source – comando utilizado para permitir ao utilizador utilizar as packages do ROS2 e utilizar os comandos do ROS2.

1.2 Configuração do ambiente de trabalho ROS2

Assumindo que o ROS2 foi devidamente instalado no sistema, será necessário criar-se um ambiente de trabalho para iniciar o desenvolvimento de uma simulação.

O ambiente de trabalho núcleo chama-se de *underlay*, subsequentes ambientes de trabalho chamam-se de *overlays*.

Ambiente de trabalho – diretoria no sistema onde se está a desenvolver o trabalho.

1.2.1 Criação de uma nova diretoria

Será necessário fazer *source* dos ficheiros setup ROS e de seguida criar o ambiente de trabalho.

- source /opt/ros/iron/setup.bash
- mkdir -p ~/ros2_ws/src
- cd ~/ros2_ws/src

1.2.2 Clonar um repositório

Estando dentro da diretoria ros2_ws/src, pretende-se agora clonar um repositório que contém o pacote (*package*) turtlesim que irá ser utilizado para verificar o correto funcionamento dos vários componentes do ROS2.

- git clone https://github.com/ros/ros_tutorials.git -b iron

1.2.3 Resolver dependências

Antes de construir o workspace, é necessário resolver dependências de packages, caso existam, assim, na root do workspace (ros2_ws):

- rosdep install -i --from-path src --rosdistro iron -y

1.2.4 Construir o workspace

Para construir o workspace, e estando o nosso terminal encontrado na root do workspace (ros2_ws), utilizar-se-á o seguinte comando:

- colcon build

1.2.5 Criação do overlay

Para se realizar o *source* do *overlay*, abrir-se-á um novo terminal, separado do que foi utilizado para construir o workspace. No novo terminal, executar-se-ão os seguintes comandos:

- source /opt/ros/iron/setup.bash
- source ~/ros2_ws/install/local_setup.bash

Assim, no terminal original, encontra-se o *source* do ambiente ROS2 que corresponde ao *underlay*, e, num novo terminal, na *root* do projecto, realiza-se *source* do *overlay*, permitindo deste modo que se construa por cima do *underlay*.

Sempre que se pretender usar o ambiente de trabalho, será necessário repetir estes passos no terminal. No entanto, pode-se encurtar um passo adicionando o *source* do *underlay* diretamente na shell script startup:

- echo "source /opt/ros/iron/setup.bash" >> ~/.bashrc

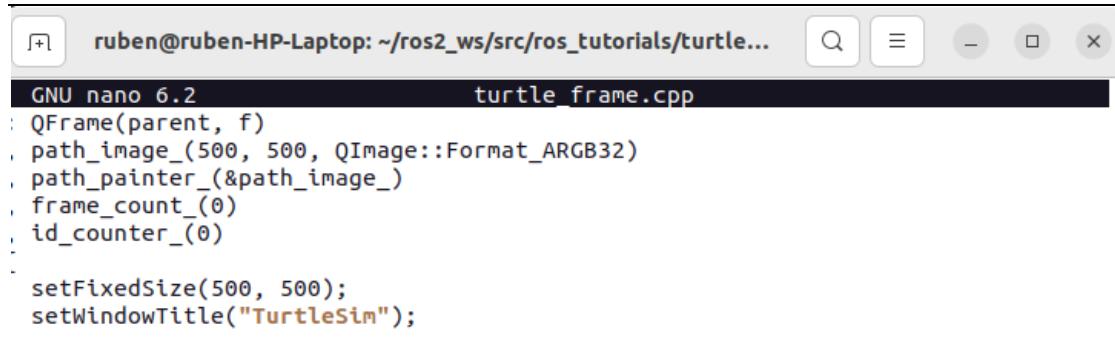
Deste modo, apenas ter-se-á futuramente que executar *source* do ficheiro *install* do ambiente de trabalho (source ~/ros2_ws/install/local_setup.bash) para iniciar um *overlay* num novo terminal.

1.2.6 Verificação da configuração correcta do *underlay/overlay*

Para verificar se o *overlay* foi corretamente configurado, podemos recorrer ao turtlesim, localizado na diretoria:

- cd ~/ros2_ws/src/rosl_tutorials/turtlesim/src
- nano turtle_frame.cpp

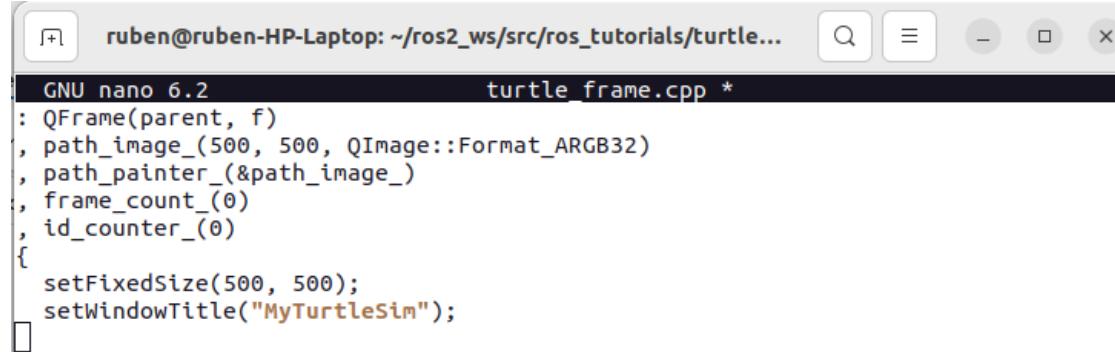
Dentro deste ficheiro, temos de alterar o valor de “setWindowTitle("TurtleSim")” para “setWindowTitle("MyTurtleSim")”.



```
GNU nano 6.2              turtle_frame.cpp
: QFrame(parent, f)
, path_image_(500, 500, QImage::Format_ARGB32)
, path_painter_(&path_image_)
, frame_count_(0)
, id_counter_(0)

setFixedSize(500, 500);
setWindowTitle("TurtleSim");
```

Figura Anexo 3 - Nome da janela que será aberta pelo terminal sourced no underlay



```
GNU nano 6.2              turtle_frame.cpp *
: QFrame(parent, f)
, path_image_(500, 500, QImage::Format_ARGB32)
, path_painter_(&path_image_)
, frame_count_(0)
, id_counter_(0)
{
    setFixedSize(500, 500);
    setWindowTitle("MyTurtleSim");
}
```

Figura Anexo 4 - Nome da janela que será aberta pelo terminal sourced no overlay

Agora, ao corrermos a simulação no *underlay*, reparamos que a janela terá como título "TurtleSim" enquanto que no *overlay* será "MyTurtleSim"

- ros2 run turtlesim turtlesim_node

Underlay:

Overlay:

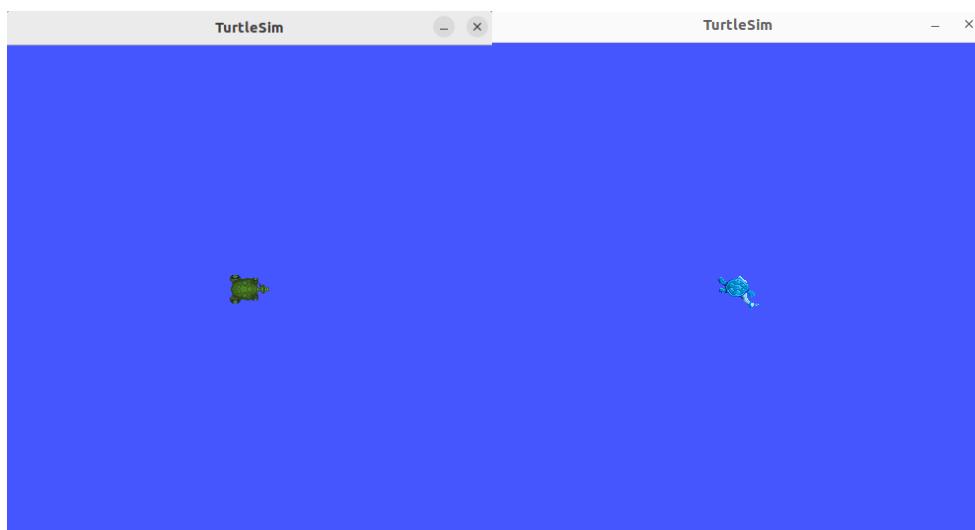


Figura Anexo 5 – Ambas as janelas ficaram com o mesmo título

Não se verificou o pretendido. Isto aconteceu porque não foi executado o “*colcon build*” na *root* do ambiente de trabalho depois de se ter alterado o ficheiro.

Assim, executaram-se os seguintes comandos antes de se tentar novamente:

- cd ~/ros2_ws
- colcon build
- ros2 run turtlesim turtlesim_node

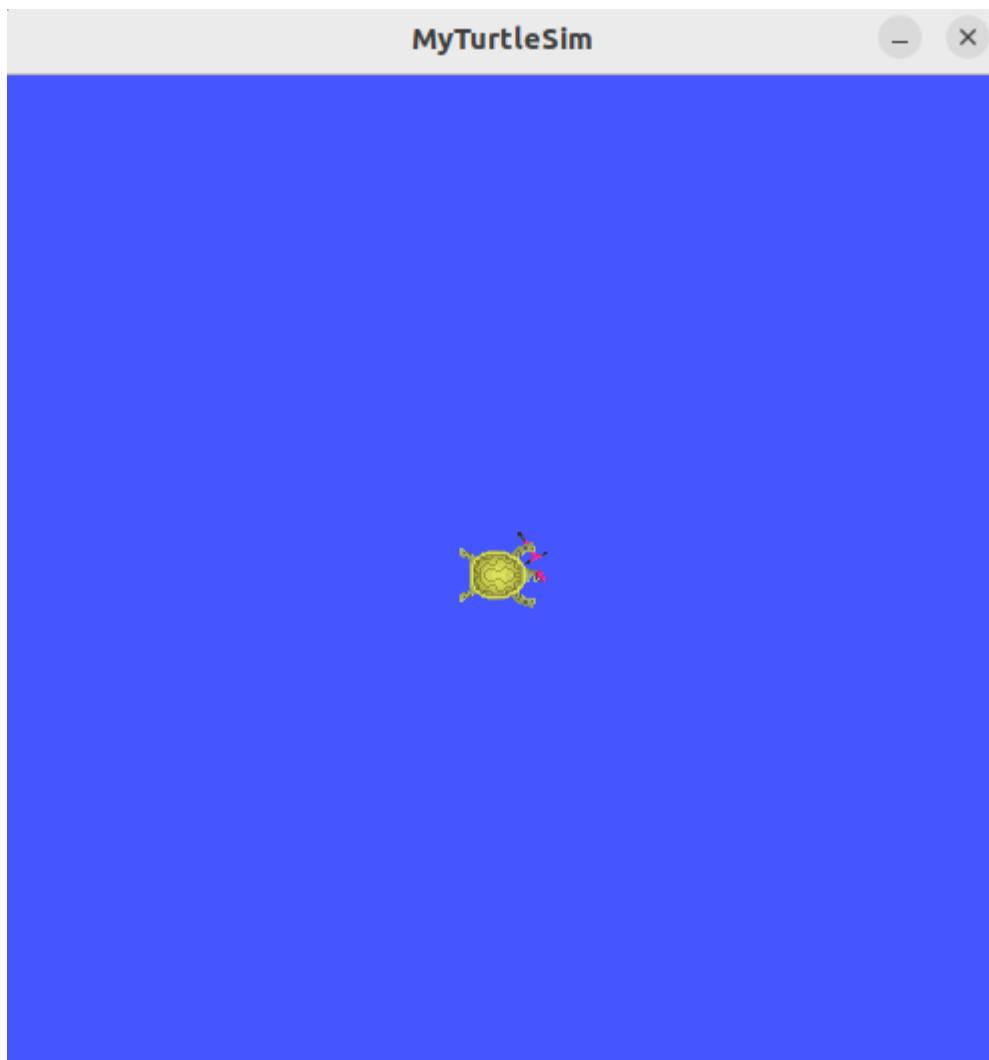


Figura Anexo 6 - Simulação do Overlay iniciou com um nome diferente do Underlay

Sucesso! Confirma-se deste modo que se alteraram ficheiros do *overlay*, no ambiente de trabalho sem alterar os ficheiros do *underlay*, no *setup*.

1.2.7 Criação de uma Package

Um pacote (*package*) é uma unidade organizacional para o nosso código ROS2. Se se pretende partilhar o código desenvolvido com outras pessoas, será então necessário organizá-lo dentro de um pacote.

O pacote será criada dentro do ambiente de trabalho que acabou de ser configurado, assim, para esse efeito, executa-se os seguintes comandos:

- `cd ~/ros2_ws/src`
 - `ros2 pkg create --build-type ament_cmake --license Apache-2.0 --node-name my_node simulacao_package`
 - `cd ~/ros2_ws`
 - `colcon build`
- .

1.2.8 Verificação do pacote “simulacao_package”

Para verificar a correta criação do pacote “simulacao_package” abre-se um novo terminal, e como foi referido anteriormente, sempre que se abre um novo terminal é necessário correr os seguintes comandos:

- `source /opt/ros/iron/setup.bash`
- `cd ~/ros2_ws`
- `source install/local_setup.bash`

De seguida, podemos correr um executável que foi criado automaticamente durante a criação do pacote:

- `ros2 run simulacao_package my_node`

E o resultado esperado deste comando será “hello world simulacao_package package”

1.3 Simulação de um veículo com 2 rodas com RVIZ

1.3.1 Criação de uma Package

Para poder-se visualizar os dados transmitidos pelos sensores do UV, ir-se-á utilizar o RViz2. Assim, no intuito de aprender como utilizar esta ferramenta, seguiu-se um tutorial que pretende criar um UV simulado com sensores funcionais[[7](#)].

1.3.2 Ficheiro XACRO

Dentro do pacote criado anteriormente na diretoria “`~/ros2_ws/src/simulacao_package`”, criou-se uma diretoria “`urdf`” e nela o ficheiro “`m2wr.xacro`”:

- `cd ~/ros2_ws/src/simulacao_package`
- `mkdir urdf`
- `cd urdf`
- `nano m2wr.xacro`

De seguida, popolou-se o ficheiro com o seguinte conteúdo:

```
❖ <?xml version="1.0"?>

<robot name="m2wr" xmlns:xacro="http://www.ros.org/wiki/xacro">

<!-- Material definition -->

<xacro:macro name="blue_material" >

  <material name="blue">

    <color rgba="0.203125 0.23828125 0.28515625 1.0"/>

  </material>

</xacro:macro>

<!-- Base Link (Root link) -->

<link name="base_link">
```

```
<visual>

<geometry>
  <box size="0.5 0.3 0.07"/>
</geometry>
<xacro:blue_material/>

</visual>
</link>

<!-- Right Wheel (attached to base link) -->
<link name="link_right_wheel">
  <visual>
    <geometry>
      <cylinder length="0.04" radius="0.1"/>
    </geometry>
    <xacro:blue_material/>
  </visual>
</link>

<!-- Left Wheel (attached to base link) -->
<link name="link_left_wheel">
  <visual>
    <geometry>
      <cylinder length="0.04" radius="0.1"/>
    </geometry>
  </visual>
</link>

<!-- Define joints to connect to wheels to the base link -->
<joint name="right_wheel_joint" type="revolute">
```

```
<parent link="base_link"/>  
    <child link="link_right_wheel"/>  
        <origin xyz="0.25 0.15 0"/>  
        <axis xyz="0 0 1"/>  
        <limit lower="-3.14" upper="3.14" effort="100" velocity="1.0"/>  
    </joint>  
  
<joint name="left_wheel_joint" type="revolute">  
    <parent link="base_link"/>  
        <child link="link_left_wheel"/>  
        <origin xyz="0.25 -0.15 0"/>  
        <axis xyz="0 0 1"/>  
        <limit lower="-3.14" upper="3.14" efforts="100" velocity="1.0"/>  
    </joint>  
  
</robot>
```

1.3.3 Ficheiro launch

Para inicializar o RViz2, iremos chamar um ficheiro launch que será escrito em python.

Para tal, iremos voltar à *root* do pacote e criar a diretoria “launch” e lá dentro o ficheiro “view_robot.launch.py”.

- cd ~/ros2_ws/src/simulacao_package
- mkdir launch
- cd launch
- nano view_robot.launch.py

De seguida iremos popular este ficheiro com o seguinte conteúdo:

```
❖ from launch import LaunchDescription
  from launch_ros.actions import Node
  from launch.substitutions import Command

  def generate_launch_description():
    return LaunchDescription([
      Node(
        package="robot_state_publisher",
        executable="robot_state_publisher",
        name="robot_state_publisher",
        output="screen",
        parameters=[{
          "robot_description": Command([
            "ros2 run xacro xacro ",
            "/home/ruben/ros2_ws/src/ simulacao_package
urdf/m2wr.xacro"
          ])
        }],
      ),
      Node(
```

```
    package="joint_state_publisher_gui",
    executable="joint_state_publisher_gui",
    name="joint_state_publisher_gui",
    output="screen"

),
Node(
    package="rviz2",
    executable="rviz2",
    name="rviz2",
    output="screen"
),
])

])
```

1.3.4 Inicialização do projeto

Criados ambos os ficheiros, volta-se agora à raíz do pacote e corre-se o seguinte comando:

- cd ~/ros2_ws
- colcon build

De seguida, inicializa-se o projeto.

- source ~/ros2_ws/install/local_setup.bash
- ros2 launch simulacao_package view_robot.launch.py

Para se conseguir visualizar o modelo será necessário editar os seguintes parâmetros no RViz2:

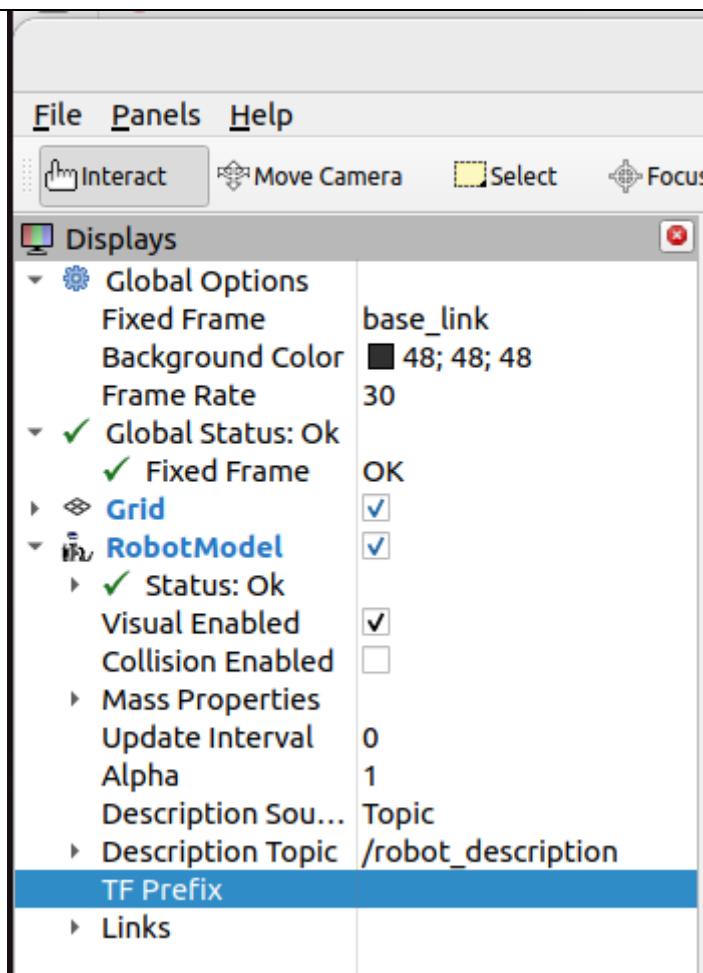


Figura Anexo 7 – Parâmetros que têm de ser selecionados no rviz2.

E, como consequência, é mostrado o modelo. Por fim, através do Joint State Publisher é possível rodar as rodas do robô simulado:

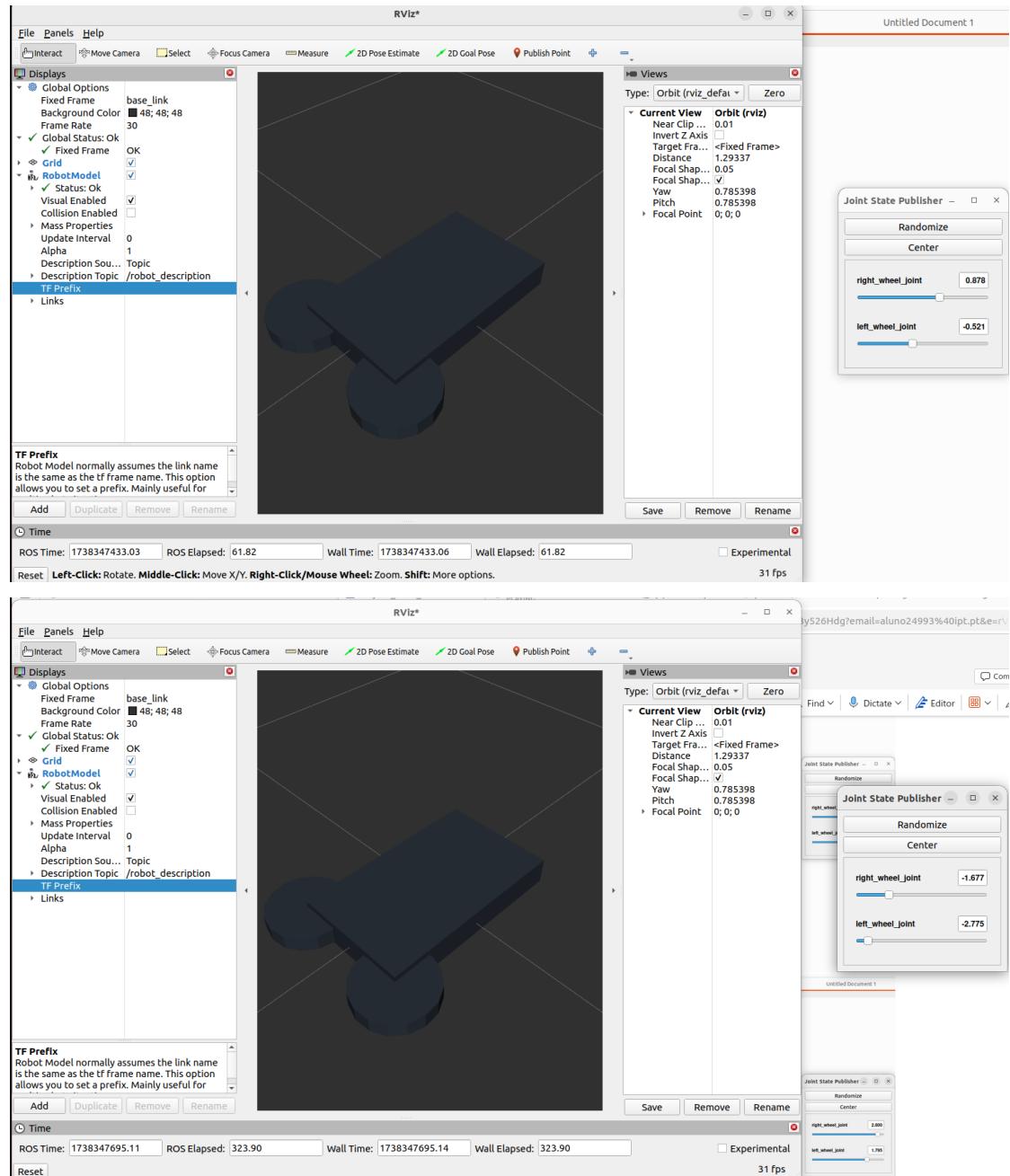


Figura Anexo 8 – Antes e após o nodo Joint State Publisher transmitir novas informações relativas ao estado das joints.

1.3.5 Gazebo launch file

Para se poder ver o modelo numa simulação Gazebo, será necessário criar uma nova launch file:

- cd ~/ros2_ws/src/simulacao_package/launch
- nano spawn.launch.py

E de seguida, preenche-la com o seguinte conteúdo:

```
❖ from launch import LaunchDescription
  from launch_ros.actions import Node
  from launch.substitutions import LaunchConfiguration, Command
  from launch.actions import DeclareLaunchArgument

  def generate_launch_description(): return LaunchDescription([
    DeclareLaunchArgument("x", default_value="0", description="X position"),
    DeclareLaunchArgument("y", default_value="0", description="Y position"),
    DeclareLaunchArgument("z", default_value="0.5", description="Z position"),

    Node(
      package="robot_state_publisher",
      executable="robot_state_publisher",
      name="robot_state_publisher",
      output="screen",
      parameters=[{
        "robot_description": Command([
          "ros2 run xacro xacro ",
          "/home/ruben/ros2_ws/src/m2wr_description/urdf/m2wr.xacro"
        ])
      }],
    ),
  ])
```

```
Node(  
    package="ros_gz_sim",  
    executable="create",  
    name="mybot_spawn",  
    output="screen",  
    arguments=[  
        "-name", "m2wr",  
        "-topic", "robot_description",  
        "-x", LaunchConfiguration("x"),  
        "-y", LaunchConfiguration("y"),  
        "-z", LaunchConfiguration("z")  
    ]  
)  
])
```

1.3.6 Spawn veículo na simulação Gazebo

Primeiro, inicializa-se a simulação “empty world” do gazebo

- ign gazebo

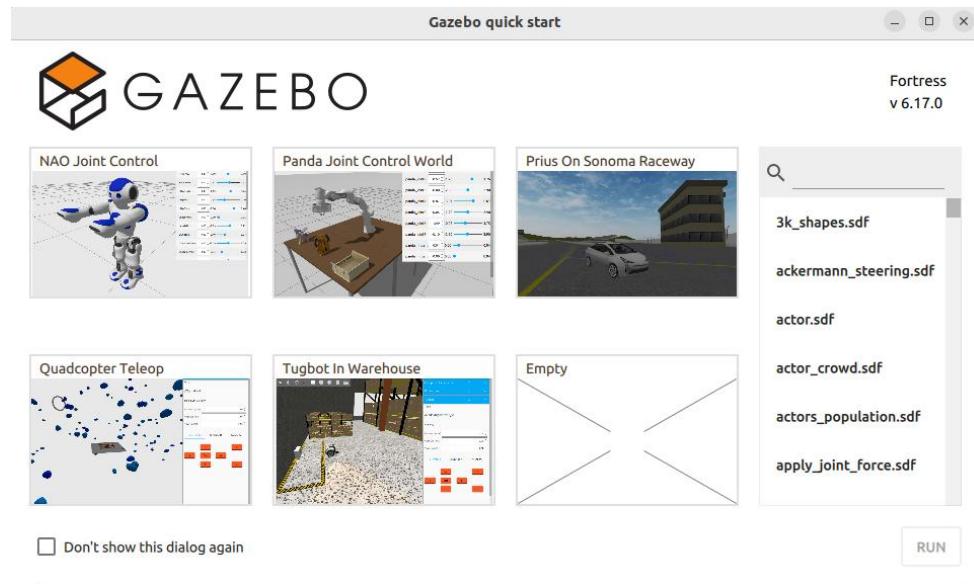


Figura Anexo 9 – Simulação de mundo vazio no Gazebo (Empty)

E, de seguida, num novo terminal, executa-se a launch file criada anteriormente

- source ~/ros2_ws/install/local_setup.bash
- ros2 launch m2wr_description view_robot.launch.py

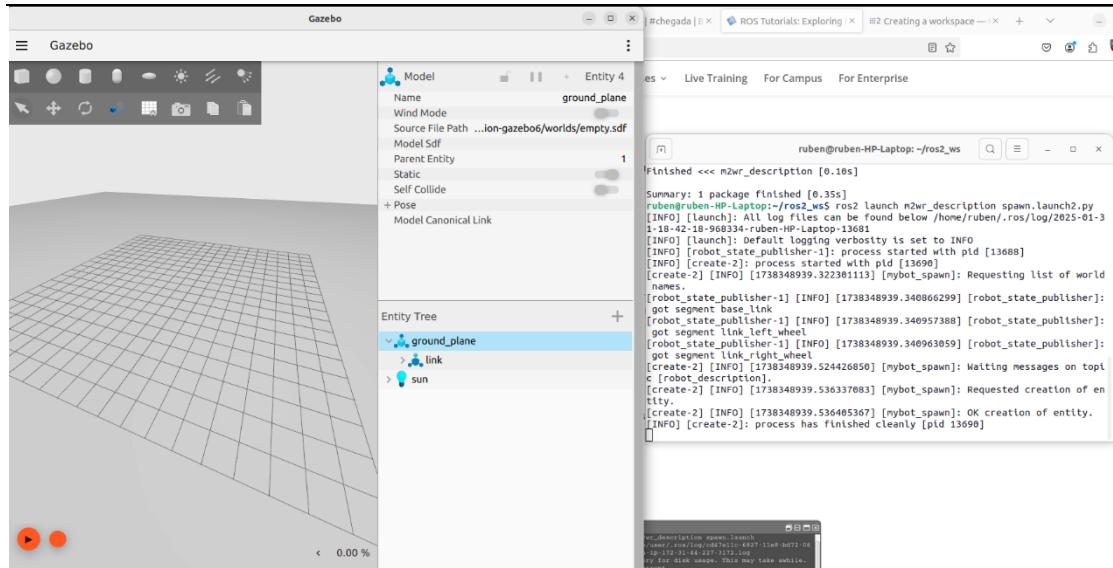


Figura Anexo 10 - Simulação Gazebo, o ficheiro launch indica que foi feito o spawn do modelo, no entanto não se consegue visualizar nada no mundo vazio.

Embora apareçam mensagens no terminal que indicam que foi realizado o spawn do veículo, este não aparece na simulação gazebo.

O conteúdo do ficheiro xacro está a ser corretamente transmitido, no entanto, dentro do gazebo aparenta não haver qualquer entidade correspondente ao modelo.

1.4 Gazebo

1.4.1 Instalação

Será instalada a versão Fortress do Gazebo, pois para além de ser a recomendada para a versão do Ubuntu que está a ser utilizada neste projeto (Ubuntu Jammy (22.04)), é também compatível tanto com a versão Iron Irwini do ROS2 como com o ArduPilot que irá ser integrado posteriormente.

Instalação de ferramentas necessárias:

- sudo apt-get update
- sudo apt-get install curl lsb-release gnupg

Instalação do Gazebo Fortress:

- sudo curl https://packages.osrfoundation.org/gazebo.gpg --output /usr/share/keyrings/pkgs-osrf-archive-keyring.gpg
echo "deb [arch=\$(dpkg --print-architecture) signed-by=/usr/share/keyrings/pkgs-osrf-archive-keyring.gpg] http://packages.osrfoundation.org/gazebo/ubuntu-stable \$(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/gazebo-stable.list > /dev/null
- sudo apt-get update
- sudo apt-get install ignition-fortress

1.4.2 Verificação da instalação

Abriu-se um novo terminal e realizou-se *source* do *overlay*,

- source ~/ros2_ws/install/local_setup.bash

De seguida executou-se o seguinte comando:

- ign gazebo -v 4 -r visualize_lidar.sdf

E como consequência, inicializou-se uma simulação de um robô com 2 rodas:

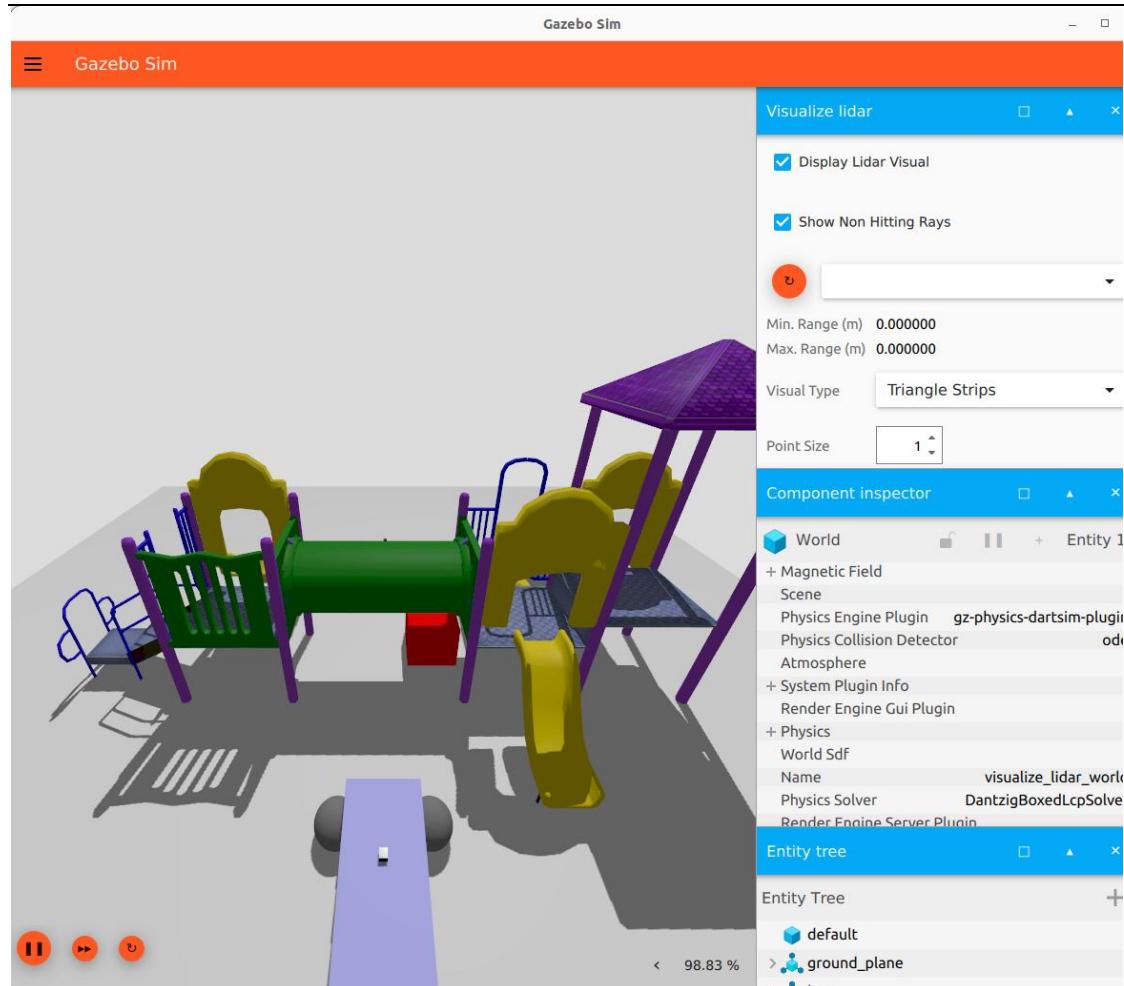


Figura Anexo 11 – Simulação pré-existente dentro dos ficheiros do Gazebo Fortress

Se corrermos o seguinte comando:

- `ign topic -l`

Verificamos todos os tópicos provenientes do Gazebo.

```
ruben@ruben-HP-Laptop:~$ ign topic -l
/clock
/gazebo/resource_paths
/gui/camera/pose
/lidar
/lidar/points
/lidar2
/lidar2/points
/model/vehicle_blue/odometry
/model/vehicle_blue/tf
/sensors/marker
/stats
/world/visualize_lidar_world/clock
/world/visualize_lidar_world/dynamic_pose/info
/world/visualize_lidar_world/pose/info
/world/visualize_lidar_world/scene/deletion
/world/visualize_lidar_world/scene/info
/world/visualize_lidar_world/state
/world/visualize_lidar_world/stats
```

Figura Anexo 12 - Tópicos que estão a ser publicados pelo Gazebo Fortress

1.4.3 Comunicar com a simulação através do ROS2

Para conseguir comunicar com a nossa simulação utilizando o ROS 2, é necessário usar um pacote chamado ros_gz_bridge. Este pacote fornece uma network bridge que permite a troca de mensagens entre o ROS 2 e o Gazebo Transport*.

- sudo apt-get install ros-iron-ros-ign-bridge

Após a instalação, vamos correr a bridge num novo terminal:

- ros2 run ros_gz_bridge parameter_bridge /model/vehicle_blue/cmd_vel@geometry_msgs/msg/Twist[ignition.msgs.Twist

Enviamos agora uma mensagem para o topic para que o robô se movimente num outro novo terminal.

- ros2 topic pub /model/vehicle_blue/cmd_vel geometry_msgs/Twist "linear: { x: 0.1 }"

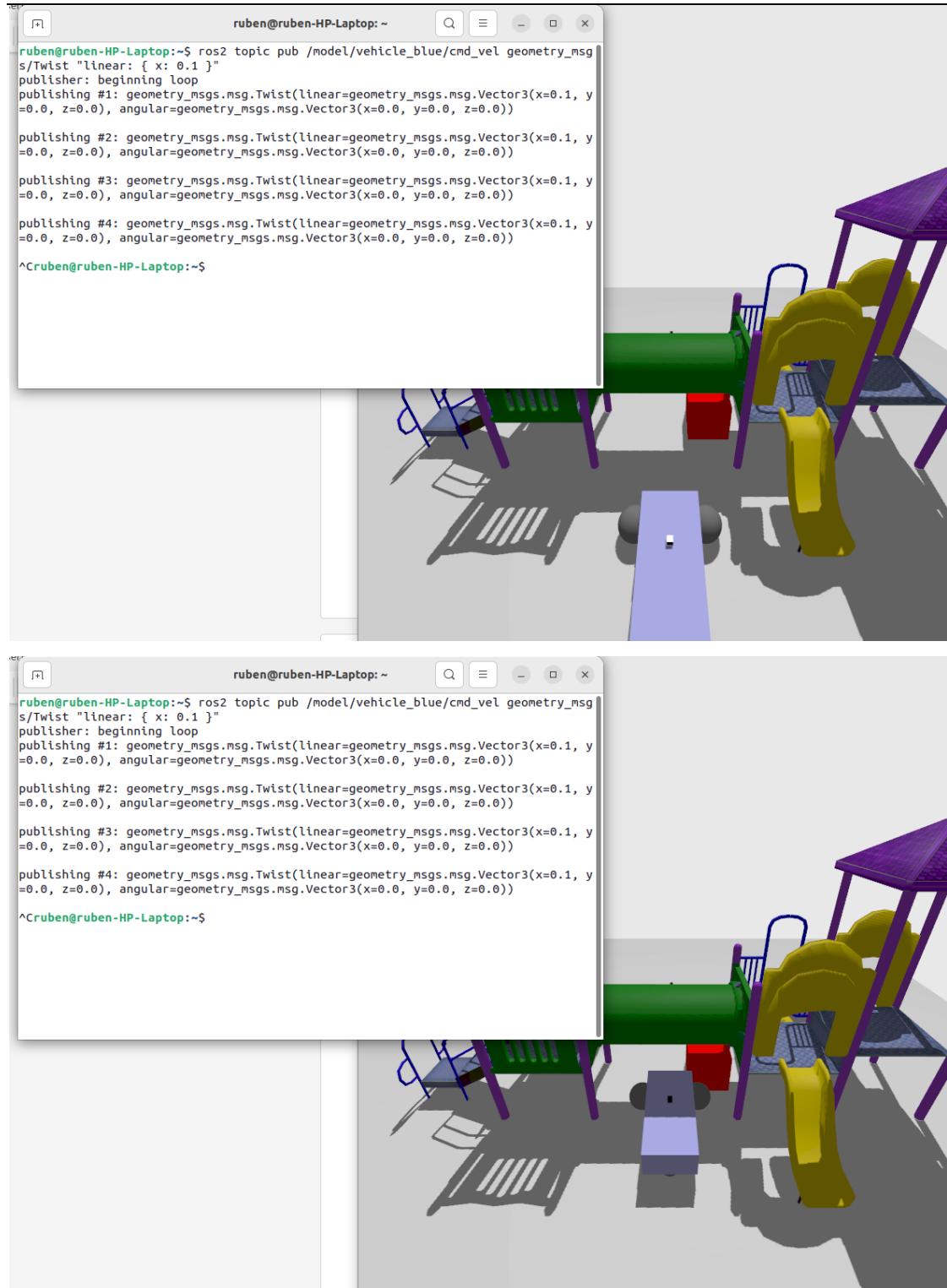


Figura Anexo 13 – Imagens elucidativas do movimento que o robô executa

Sucesso! Foi possível estabelecer uma comunicação entre o ROS2 e o Gazebo Fortress através de uma bridge.

Network Bridge - software que conecta duas ou mais redes diferentes, permitindo a comunicação entre elas. permite que o ROS2 e o Gazebo Transport troquem informações entre si, apesar de serem sistemas independentes

Gazebo Transport - Gazebo Transport é um sistema de comunicação utilizado pelo simulador Gazebo para permitir a troca de dados entre diferentes componentes dentro da simulação. Ele é responsável por gerir a comunicação entre o Gazebo e os outros sistemas, como sensores, controladores, e outros módulos de simulação.

1.5 Visualizar data LIDAR

1.5.1 Criação de uma nova bridge

Para visualizar a data LIDAR provinda dos tópicos mencionados anteriormente aos quais a simulação do gazebo está subscrita:

```
ruben@ruben-HP-Laptop:~$ ign topic -l
/clock
/gazebo/resource_paths
/gui/camera/pose
/lidar
/lidar/points
/lidar2
/lidar2/points
```

Figura Anexo 14 - Tópicos relevantes à data que está a ser transmitida pelo sensor LIDAR

Recorrer-se-á a uma nova bridge num novo terminal que irá estabelecer o contacto entre o Gazebo e o ROS2.

- source /opt/ros/iron/setup.bash
- ros2 run ros_gz_bridge parameter_bridge /lidar2@sensor_msgs/msg/LaserScan[ignition.msgs.LaserScan --ros-args -r /lidar2:=/laser_scan

No entanto, no RViz2 não é possível escolher um fixed frame para visualizar a data LIDAR. Assim, será necessário realizar o *troubleshooting*.

Primeiro, verificou-se se existem tópicos TF a publicarem tf data:

```
ruben@ruben-HP-Laptop:~$ ros2 topic list | grep tf
/tf
/tf_static
ruben@ruben-HP-Laptop:~$ ros2 topic echo /tf
```

Figura Anexo 15 - Verifica-se que o ros2 está subscrito ao tópico tf, no entanto não se verifica nenhuma entrada de dados

Verificou-se que sim, no entanto não existe nenhuma data a chegar ao destino pretendido, o que significa que a bridge não está a conseguir realizar o forwarding de tf data.

Assim, embora a data esteja a ser publicada pelo Gazebo:

```
ruben@ruben-HP-Laptop:~$ rostopic echo /model/vehicle_blue/tf
pose {
  header {
    stamp {
      sec: 1467
    }
    data {
      key: "frame_id"
      value: "vehicle_blue/odom"
    }
    data {
      key: "child_frame_id"
      value: "vehicle_blue/chassis"
    }
}
```

Figura Anexo 16 – Verifica-se que o Gazebo está a transmitir a tf data relevante ao seu modelo robô

Não está a ser corretamente recebida pelo RViz2. Após diversas tentativas de *troubleshooting*, o problema foi considerado uma falha de compatibilidade entre as versões do Gazebo e do Ros. Assim, estando ainda numa fase inicial, decidiu-se recomeçar o projeto com versões compatíveis.

Anexo 3 - Instalação do software necessário (ROS2, Gazebo, ArduPilot)

Tendo em conta os problemas encontrados na tentativa descrita anteriormente [Anexo 2], o grupo optou por seguir a documentação do ArduPilot, onde é explicado passo a passo como instalar tudo o que é necessário para se conseguir simular o veículo no Gazebo.

Neste caso, instalaram-se as seguintes versões dos programas:

- Ubuntu **Jammy (22.04)**
- ROS2 **Humble**
- Gazebo **Harmonic**
- ArduPilot

Passo 1: Instalação do ROS2 Humble

Seguiram-se os passos definidos na documentação do ROS :

Definição de um locale que suporte UTF-8

- sudo apt update && sudo apt install locales
- sudo locale-gen en_US en_US.UTF-8
- sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
- export LANG=en_US.UTF-8

Configuração das Fontes (Sources)

Será necessário adicionar o repositório apt ROS2 ao sistema. Assim, começa-se por verificar que o repositório Ubuntu Universe [\[9\]](#) está disponível.

- sudo apt install software-properties-common
- sudo add-apt-repository universe

Now add the ROS 2 GPG key with apt.

- sudo apt update && sudo apt install curl -y
- sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg

Adição do repositório ao *source*:

- echo "deb [arch=\$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu \$(. /etc/os-release && echo \$UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null

Instalação dos pacotes do ROS2

Atualizar a cache dos repositórios depois de configurar os repositórios.

- sudo apt update

Os pacotes do ROS 2 são compilados em sistemas Ubuntu que são atualizados com frequência. É bastante recomendado manter o sistema sempre atualizado.

- sudo apt upgrade

Instalação do ROS2 para *desktop*, a versão mais recomendada

- sudo apt install ros-humble-desktop

Preparação do ambiente

Realizar o *source* no projeto.

- source /opt/ros/humble/setup.bash

Para não estar sempre a escrever este código em cada terminal novo, realizamos o comando seguinte que permite sempre que um terminal é aberto o comando seja de imediato executado:

- echo "source /opt/ros/humble/setup.bash"
>> ~/.bashrc

Passo 2: Instalação do ArduPilot com ROS2

Seguiram-se os passos definidos na documentação do ArduPilot[[10](#)]:

Clonaram-se os repositórios requeridos através do vcs.

- mkdir -p ~/ardu_ws/src
- cd ~/ardu_ws
- vcs import --recursive --input https://raw.githubusercontent.com/ArduPilot/ardupilot/master/Tools/ros2/ros2_repos.src

Depois, atualizaram-se as dependências.

- cd ~/ardu_ws
- sudo apt update
- rosdep update
- source /opt/ros/humble/setup.bash
- rosdep install --from-paths src --ignore-src -r -y

Instalação do MicroXRCEDDSGen.

- sudo apt install default-jre
- cd ~/ardu_ws
- git clone --recurse-submodules <https://github.com/ardupilot/Micro-XRCE-DDS-Gen.git>
- cd [Micro-XRCE-DDS-Gen](#)
./gradlew assemble
- echo "export PATH=\\$PATH:\$PWD/scripts" >> ~/.bashrc

Por fim, o *build* do *workspace*.

- cd ~/ardu_ws
- colcon build --packages-up-to ardupilot_dds_tests

Passo 3: Instalação do Gazebo Harmonic

Seguiram-se os passos definidos na documentação do Gazebo [\[11\]](#):

Primeiro, instalaram-se as ferramentas necessárias.

- sudo apt-get update
- sudo apt-get install curl lsb-release gnupg

E, de seguida, o Gazebo Harmonic.

- sudo curl https://packages.osrfoundation.org/gazebo.gpg --output /usr/share/keyrings/pkgs-osrf-archive-keyring.gpg
- echo "deb [arch=\$(dpkg --print-architecture) signed-by=/usr/share/keyrings/pkgs-osrf-archive-keyring.gpg] http://packages.osrfoundation.org/gazebo/ubuntu-stable \$(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/gazebo-stable.list > /dev/null
- sudo apt-get update
- sudo apt-get install gz-harmonic

De seguida, clonaram-se todos os repositórios de código fonte necessários para o nosso ambiente de trabalho ROS[\[12\]](#):

- cd ~/ardu_ws
- vcs import --input https://raw.githubusercontent.com/ArduPilot/ardupilot_gz/main/ros2_gz.repos --recursive src

Set the Gazebo version to harmonic. It's recommended to set this in your `~/.bashrc` file.

Definiu-se também a variável correspondente à versão do gazebo para a versão que iremos utilizar, “harmonic”:

- echo "export GZ_VERSION=harmonic" >> `~/.bashrc`

Adionaram-se, depois, Gazebo APT sources.

- sudo apt install wget

-
- wget https://packages.osrfoundation.org/gazebo.gpg -O /usr/share/keyrings/pkgs-osrf-archive-keyring.gpg
 - echo "deb [arch=\$(dpkg --print-architecture) signed-by=/usr/share/keyrings/pkgs-osrf-archive-keyring.gpg] http://packages.osrfoundation.org/gazebo/ubuntu-stable \$(lsb_release -cs) main" | tee /etc/apt/sources.list.d/gazebo-stable.list > /dev/null
 - sudo apt update

Devido a se estar a recorrer a uma combinação não padrão (ROS 2 Humble com Gazebo Harmonic), as dependências do Gazebo podem não estar disponíveis nos repositórios padrão do rosdep. Assim, efetuou-se o seguinte passo:

- wget https://raw.githubusercontent.com/osrf/osrf-rosdep/master/gz/00-gazebo.list -O /etc/ros/rosdep/sources.list.d/00-gazebo.list
- rosdep update

Atualização das dependências do ROS e do Gazebo:

- cd ~/ardu_ws
- source /opt/ros/humble/setup.bash
- sudo apt update
- rosdep updaterosdep install --from-paths src --ignore-src -y

Nota: No próximo passo, devido a estar-se a fazer uma compilação de ficheiros C++ grandes, pode ocorrer um crash por falta de RAM. Este obstáculo foi ultrapassado pelo grupo aumentando o tamanho da swap file:

- cd ~/ardu_ws
- colcon build --packages-up-to ardupilot_gz Bringup

Para testar a instalação, correram-se os seguintes comandos:

- cd ~/ardu_ws
- source install/setup.bash
- colcon test --packages-select ardupilot_sitl ardupilot_dds_tests ardupilot_gazebo ardupilot_gz_applications ardupilot_gz_description ardupilot_gz_gazebo ardupilot_gz_Bringup
- colcon test-result --all --verbose

Poderá haver alguma necessidade de corrigir erros nesta fase, para tal o grupo guiou-se pelo output retornado pela flag “--verbose”.

Correr a Simulação

Por fim, após fazer-se source, resta-nos lançar uma das simulações exemplo do Gazebo:

- source install/setup.bash
- ros2 launch ardupilot_gz_bringup iris_runway.launch.py

Descolagem e Aterragem

Com a simulação a funcionar, temos de instalar o MAVROS para comunicar entre o ROS2 e o ArduPilot de modo a meter o veículo, neste caso, “iris_runway” a voar.

- sudo apt install ros-humble-mavros ros-humble-mavros-extras

De seguida, será necessário instalar o GeographicLib:

- Sudo wget https://raw.githubusercontent.com/mavlink/mavros/master/mavros/scripts/install_geographiclib_datasets.sh
- sudo chmod +x install_geographiclib_datasets.sh
- sudo ./install_geographiclib_datasets.sh

Correr o nodo do MAVROS num novo terminal:

- ros2 run mavros mavros_node --ros-args -p fcu_url:=udp://:14550@127.0.0.1:14550

Descolar:

- ros2 service call /mavros/set_mode mavros_msgs/srv/SetMode "{custom_mode: 'GUIDED'}"
- ros2 service call /mavros/cmd/arming mavros_msgs/srv/CommandBool "{value: true}"
- ros2 service call /mavros/cmd/takeoff mavros_msgs/srv/CommandTOL "{min_pitch: 0.0, yaw: 0.0, latitude: 0.0, longitude: 0.0, altitude: 3.0}"

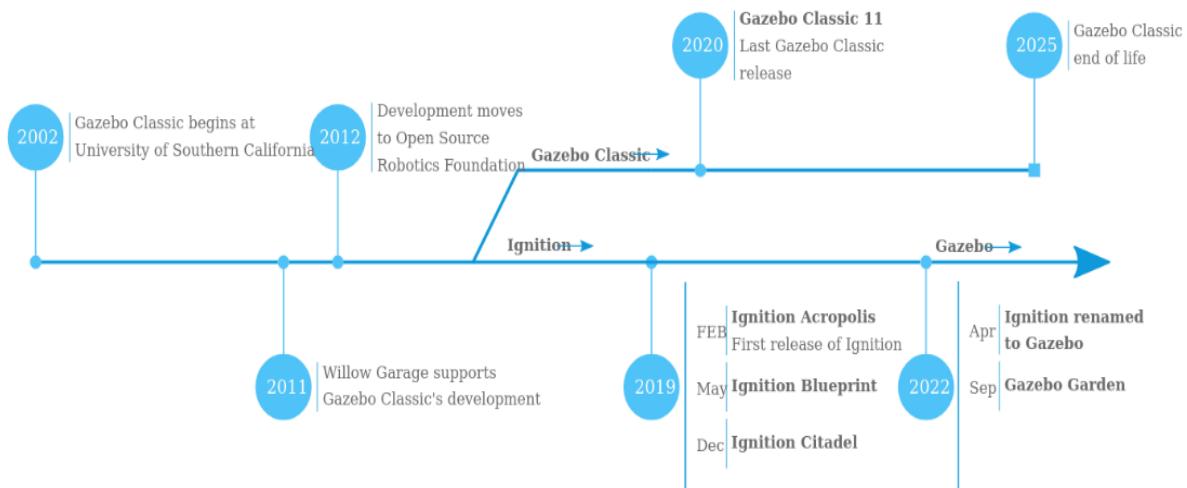
Aterratar:

- ros2 service call /mavros/set_mode mavros_msgs/srv/SetMode "{custom_mode: 'LAND'}"

Deste modo, conclui-se que foi possível implementar o ArduPilot com o ROS2 e o Gazebo para um veículo aéreo.

No entanto, esta porção do trabalho foi desenvolvida com um intuito didático, seguindo os tutoriais disponibilizados pela documentação do ROS, não correspondendo concretamente aos objetivos do grupo e foi portanto, após esta fase, dado como concluído.

Anexo 4 - Versões Gazebo



Terminology

Figura Anexo 18 - Versões do software Gazebo[[13](#)]