FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Desenvolvimento de sistema de visualização 3D para simulador robótico

Miguel Ferreira de Andrade

Mestrado em Engenharia Informática e Computação

Supervisor: Dr. Nuno Miguel Neves de Abreu

27 de julho de 2023

Desenvolvimento de sistema de visualização 3D para simulador robótico

Miguel Ferreira de Andrade

Mestrado em Engenharia Informática e Computação

27 de julho de 2023

Resumo

A computação gráfica tem vindo a ser essencial na interação humano-computador, permitindo uma rápida introdução de dados e a respectiva visualização dos resultados. Esta interação torna-se especialmente útil quando aplicada sistemas robóticos. Aliada à computação gráfica, a simulação também tem um papel bastante importante para a robótica. Neste projecto o foco são os sistemas robóticos dedicados à exploração. Este tipo de actividade robótica contém vários desafios que necessitam de ser ultrapassados, sendo um deles o mapeamento do espaço envolvente. A forma como o robô interpreta o meio que o rodeia é essencial para o funcionamento do próprio e também para os humanos que estão monitorizar ou mesmo a controlar o veículo. Assim existe uma necessidade de desenvolver uma interface que permita visualizar e interagir com esta informação. Pretende-se portanto manter e actualizar um *Digital Twin* do espaço envolvente, considerando os dados recolhidos em tempo real pelo veículo. *Digital Twins* são representações digitais de um determinado produto real que se encontra num espaço real. Existe assim uma ponte entre o modelo digital e o mundo real onde a informação flui bidireccionalmente, actualizando o modelo digital a partir dos dados adquiridos no mundo real e optimizando decisões relativas à operação do robô no mundo real.

Este projecto apresenta uma interface gráfica na web, desenvolvida em *python* e *javascript*, que recebe uma nuvem de pontos proveniente de um sensor *LiDAR* simulado e permite visualizar, interagir e modificar o modelo 3D construído através dos dados recebidos, em tempo real. Este projecto utilizou várias ferramentas das quais destacam-se a biblioteca *Open3D* usada para gerar a geometria 3D proveniente da nuvem de pontos, o simulador *Gazebo* usado para simular o comportamento do sensor *LiDAR* e a biblioteca *Three.js* responsável pela visualização do modelo 3D. Esta interface é independente do simulador, o que torna possível receber dados de um veículo real com sensores verdadeiros. Foram usados quatro cenários de testes, um modelo de um café, bomba de gasolina, escolas e uma garagem com várias divisões. Os resultados obtidos demonstram que o modelo 3D gerado representa de forma precisa a escala e a posição correcta dos cenários de testes. Verificou-se também que a interface se comporta da maneira pretendida, conseguindo visualizar e modificar os parâmetros de geração do modelo enquanto está a consumir a nuvem de pontos proveniente do simulador. No entanto nem todos os algoritmos usados para geração da geometria 3D foram satisfatórios, sendo que o algoritmo *poisson surface reconstruction* foi o que apresentou pior desempenho. A geometria gerada também não mantém o detalhe que os cenários de teste possuem.

Keywords: Computação Gráfica, Digital Twin, Simulação Robótica

Abstract

Computer graphics has become essential in human-computer interaction, allowing a quick input of data and the respective visualization of the results, becoming especially useful when applied to robotic systems. Allied to computer graphics, simulation also plays a very important role for robotics. In this project, the focus is on robotic systems dedicated to exploration, which contains several challenges that need to be overcome, one of them is the mapping of the surrounding space. The way in which the robot perceives the environment that surrounds it is essential for the functioning of itself and also for the humans who are monitoring or even controlling the vehicle, creating a need for an interface that allows viewing and interacting with this information. It is therefore intended to maintain and update a *Digital Twin* of the surrounding space, considering the data collected in real time by the vehicle. *Digital Twins* are digital representations of a certain real product that is in a real space. There is thus a bridge between the digital model and the real world where information flows bidirectionally, updating the digital model from data acquired in the real world and optimizing decisions regarding the robot's operation.

This project presents a web graphical interface, developed in *python* and *javascript*, which receives a point cloud from a simulated *LiDAR* sensor and allows viewing, interacting and modifying the 3D model built using the received data, in real time. This project used several tools, including the *Open3D* library used to generate the 3D geometry from the point cloud, the *Gazebo* simulator used to simulate the behavior of the *LiDAR* sensor and the *Three.js* library responsible for visualizing the 3D model. This interface is independent of the simulator, which makes it possible to receive data from a real vehicle with real sensors. Four test scenarios were used, a model of a cafe, gas station, schools and a multi-room garage. The results demonstrate that the generated 3D model accurately represents the scale and the correct position of the test scenarios. It was also verified that the interface behaves as intended, being able to visualize and modify the model generation parameters while consuming the point cloud from the simulator. However, not all algorithms used to generate the 3D geometry were satisfactory, and the *poisson surface reconstruction* algorithm presented the worst performance. The generated geometry also doesn't maintain the detail that the test scenarios have.

Keywords: Computer Graphics, Digital Twin, Robot Simulation

Agradecimentos

Ao Professor Doutor Nuno Miguel Neves de Abreu pela paciência e apoio demonstrado durante todo este projecto.

Aos meus pais pelo constante apoio e encaminhamento durante toda a minha vida, permitindo-me alcançar o patamar onde estou hoje e crescer enquanto ser humano.

Ao meu irmão pela disponibilidade constante e orientação, assim como todas as trocas de ideias malucas que aguçaram a minha criatividade.

Aos meu amigos por todos os momentos memoráveis que me deixaram experienciar.

Miguel Ferreira de Andrade

Conteúdo

1 Introdução **1**
- 1.1 Contexto . . . 1
- 1.2 Motivação . . . 1
- 1.3 Objetivos . . . 2
- 1.4 Estrutura do Documento . . . 2

2 Background **3**
- 2.1 Digital Twins . . . 3
  - 2.1.1 O que são Digital Twins . . . 3
  - 2.1.2 Ponte entre o mundo físico e o mundo digital . . . 4
  - 2.1.3 A dimensão HMI (Human-Machine Interaction) . . . 4
  - 2.1.4 A dimensão de entrada de dados . . . 5
- 2.2 Robótica . . . 7
  - 2.2.1 Veículos marinhos autónomos . . . 7
  - 2.2.2 Arquitectura de software de um sistema robótico . . . 8

3 Estado da Arte **13**
- 3.1 Aplicação dos digital twins na robótica . . . 13
  - 3.1.1 Indústria . . . 13
  - 3.1.2 Medicina . . . 13
  - 3.1.3 Aviação . . . 14
  - 3.1.4 Robótica . . . 15
- 3.2 Ferramentas de simulação existentes em robótica . . . 15
  - 3.2.1 UWSim . . . 15
  - 3.2.2 Gazebo . . . 16
  - 3.2.3 MORSE . . . 18
- 3.3 Revisão Científica . . . 19

4 Desenvolvimento **27**
- 4.1 Abordagem ao Problema . . . 27
  - 4.1.1 Arquitectura . . . 28
- 4.2 Tecnologias utilizadas . . . 28
  - 4.2.1 Gazebo . . . 28
  - 4.2.2 Open3D . . . 29
  - 4.2.3 WebGL e Three.js . . . 29
  - 4.2.4 C++, Python e Javascript . . . 29
- 4.3 Implementação . . . 29
  - 4.3.1 Ambiente de Simulação . . . 30

4.3.2 Interface Gráfica 34
4.3.3 Interconexão entre Simulador e Interface gráfica 44

5 Resultados **47**
5.1 Cenários de teste 47
5.2 Resultados dos testes 49
5.3 Discussão dos resultados 58

6 Conclusões e Trabalho Futuro **61**
6.1 Sumário 61
6.2 Trabalho futuro 62

Referências **63**

Lista de Figuras

2.1 *Digital Twin* . . . 3
2.2 Relação entre *Digital Twin* e o robô . . . 6
2.3 Conceitos Básicos *ROS* [25] . . . 9

4.1 Fluxograma da Aplicação . . . 28
4.2 Diagrama básico de um sensor *(*LiDAR) [10] . . . 30
4.3 Veículo com o sensor . . . 31
4.4 Diagrama da simulação do sensor . . . 32
4.5 Estrutura de dados que o sensor exporta . . . 33
4.6 Diagrama do Sensor Simulado . . . 34
4.7 Interface Gráfica . . . 35
4.8 Diagrama da Função *pointCloudMesh()* . . . 39
4.9 Tratamento da nuvem de pontos . . . 40
4.10 Algoritmo *Alpha shapes* . . . 41
4.11 Algoritmo *Ball pivoting* . . . 43
4.12 Algoritmo *Poisson surface reconstruction* . . . 44

5.1 Café . . . 47
5.2 Bomba de combustível . . . 48
5.3 Escola . . . 48
5.4 Garagem . . . 49
5.5 Nuvem de pontos do cenário café . . . 50
5.6 *Ball pivoting* para o cenário café . . . 50
5.7 *Alpha shapes* para o cenário café . . . 51
5.8 *Poisson surface reconstruction* para o cenário café . . . 51
5.9 Nuvem de pontos do cenário bomba de combustível . . . 52
5.10 *Ball pivoting* para o cenário bomba de combustível . . . 53
5.11 *Alpha shapes* para o cenário bomba de combustível . . . 53
5.12 *Poisson surface reconstruction* para o cenário bomba de combustível . . . 54
5.13 Nuvem de pontos do cenário escola . . . 54
5.14 *Ball pivoting* para o cenário escola . . . 55
5.15 *Alpha shapes* para o cenário escola . . . 55
5.16 *Poisson surface reconstruction* para o cenário escola . . . 56
5.17 Nuvem de pontos do cenário garagem . . . 56
5.18 Antes da movimentação do sensor . . . 57
5.19 Depois da movimentação do sensor . . . 57
5.20 Posição final do sensor . . . 58

Lista de Tabelas

3.1 Tabela análise crítica dos artigos. 23

4.1 Tabela descritiva dos parâmetros. 35

Lista de excertos de código

4.1 Formato dos pontos em JSON 34
4.2 Formato dos parâmetros em JSON 37

Abreviações

MBSE	*Model-Based System Engineering*
IoT	*Internet of Things*
ROV	*Remotely Operated Vehicle*
AUV	*Autonomous Underwater Vehicle*
ASV	*Autonomous Surface Vehicle*
URDF	*Unified Robot Description Format*
ODE	*Open Dynamic Engine*
GLUT	*OpenGL Utility Toolkit*
LiDAR	*Light Detection and Ranging*
IMU	*Internal Measurement Unit*
SLAM	*Simultaneous Localization and Mapping*

Capítulo 1

Introdução

1.1 Contexto

Nos dias de hoje a computação gráfica está presente em todo o lado. Esta permite-nos interagir com dispositivos complexos de uma forma mais fácil. Não só é uma mais valia para o cidadão comum como também é um instrumento valioso para os investigadores que assim conseguem analisar resultados de uma forma mais acessível. A simulação ganha uma outra dimensão quando aliada à computação gráfica permitindo que seja possível projetar um produto de início ao fim e com a facilidade de testar novas variantes do produto sem que seja necessário começar tudo de novo. Os sistemas robóticos beneficiam deste método, visto que os testes em ambiente real nem sempre são simples, podendo mesmo acarretar riscos, dificuldades logísticas e/ou custos adicionais. Desta forma é possível testar o produto em diversas situações e analisar o seu comportamento sem que haja necessidade de colocar o modelo real perante esses mesmo ambientes. No entanto a utilidade da computação gráfica pode ainda servir para representar de forma visual os dados obtidos pelo próprio veículo robótico, como por exemplo, a representação tridimensional do ambiente que envolve o robô.

1.2 Motivação

Na exploração existem vários desafios que necessitam de ser ultrapassados. Em muitos casos é necessário recorrer a veículos autónomos para explorar determinada região pois estas são demasiado perigosas ou completamente inacessíveis aos seres humanos. É necessário conseguir desenvolver mecanismos que permitam ao veículo saber onde se encontra em relação ao espaço que o rodeia. Essa mesma percepção do ambiente que envolve o robô, não só introduz os meios que permitem a orientação e a navegação do veículo, como também a capacidade de validar dados dos sensores do mesmo, exercendo comparações entre os resultados obtidos e o ambiente real. Existe assim uma necessidade de criar mecanismos que permitam fazer um mapeamento do ambiente envolvente.

1.3 Objetivos

Neste trabalho os objetivos são, desenvolver um sistema que permita a visualização do espaço envolvente através dos dados fornecidos por um sensor e fazer uma simulação responsável por imitar o comportamento de um sensor *LiDAR (Light Detection and Ranging)*. É necessário fazer uma mapeamento 3D do ambiente permitindo, não só, a sua visualização como também interagir com o ambiente gerado. É também importante tornar a interface independente do simulador de forma a que seja possível enviar dados reais e não exclusivamente simulados.

1.4 Estrutura do Documento

Este documento contém seis partes.

O Capítulo 1 contém a introdução com o contexto de problema, a motivação, objetivos e a estrutura do relatório.

O Capítulo 2 apresenta o conhecimento base necessário para este projeto, abrangendo o conceito de *digital twins* e a sua utilização assim como a arquitectura de um sistema robótico.

O Capítulo 3 contém o estado da arte com os trabalhos relacionados a este projecto e as tecnologias existentes.

O Capítulo 4 aborda o desenvolvimento e a implementação do projecto.

O Capítulo 5 inclui os resultados obtidos do desenvolvimento do projecto.

O Capítulo 6 apresenta o fecho do trabalho, fazendo uma reflexão do que foi feito e do trabalho futuro.

Capítulo 2

Background

2.1 Digital Twins

2.1.1 O que são Digital Twins

O conceito de *twin* teve origem no programa *Apollo* da *NASA* [53] que consistia em duas naves espaciais, em que uma delas ficava na Terra e era então chamada de *twin*. Esta nave era usada de maneira a replicar as condições da nave espacial durante as missões e assim permitir dar apoio em situações críticas aos astronautas que se encontravam em órbita. Tendo um protótipo semelhante à nave final, permitia então testar continuamente várias situações diferentes de forma a obter dados relevantes para as missões em concreto. Com o avanço da capacidade computacional este protótipo passou a transitar para o mundo digital.

Figura 2.1: *Digital Twin*

A primeira aparição do conceito *digital twins* foi apresentada por *Michael Grieves* [54]. Aqui é explorado o conceito aplicado a uma linha de montagem de uma fábrica. Tendo uma instância digital do modelo físico e tendo estas duas entidades a comunicar entre si, é possível avaliar de uma forma mais completa o sistema assim como permitir correcções. *Michael Grieves* dá-nos uma definição do que é um *Digital Twin*. *Digital Twins* são representações digitais de um determinado produto real que se encontra num espaço real. Segundo *Michael Grieves*, um *Digital Twin* contém

3 partes [54], um produto real, um produto digital e uma ligação para troca de informação entre as duas partes.

2.1.2 Ponte entre o mundo físico e o mundo digital

Os *Digital Twins* são instâncias digitais de um modelo físico que comunicam entre si de uma forma bilateral. Esta ponte é o que permite não só actualizar o modelo digital com os dados do modelo físico como é o canal que indica ao modelo físico melhoramentos que pode fazer mediante os dados criados pelas simulações do modelo digital [76].

Esta interação é especialmente poderosa quando aliada à metodologia *MBSE* (*Model-Based Systems Engineering*) [72]. O *MBSE* é uma metodologia que é usada para suportar os requisitos, design, verificação e validação de sistemas complexos. Os *Digital Twins* ganham uma outra dimensão quando aplicados juntamente com esta metodologia. Ao criar uma instância digital de um produto podemos desde logo trabalhar em cima desse modelo de forma a projetar qual o melhor design e verificar quais os requisitos mínimos necessários para o fim pretendido. Esses dados podem depois servir de base para o modelo físico. Ao longo do desenvolvimento do produto esta constante troca de dados permite testar novas ideias sem perigo de estragar o modelo físico ou mesmo sem que este esteja completamente desenvolvido. Aqui a simulação tem um papel central.

É possível dividir o desenvolvimento de sistemas complexos em quatro fases, design, engenharia, operação e serviço [46]. Nestas fases é possível usar os *digital twins* de forma a auxiliar o processo e demonstrando que o conceito pode ser usado em todas as fases de desenvolvimento. Na fase de design, o *digital twin* é usado de forma a criar um produto abstrato que cumpra os requisitos. Um protótipo virtual pode acompanhar todo o processo de conceção do produto sendo este modificado quando os requisitos mudam também. Na próxima fase as simulações começam a ter um papel mais ativo. Aqui os requisitos mínimos já estão estabelecidos. É necessário saber como é que vai funcionar a nível técnico o produto. Usando simulações é possível determinar as melhores soluções para os diversos componentes. Já na terceira fase os dados do funcionamento do modelo físico podem ser usados para melhoramento do próprio sistema ou ainda atualizar o comportamento do produto caso o seu uso pretendido também tenha sido alterado. A fase de serviço serve de suporte ao produto, recolhendo dados da operação do modelo. Isto permite que estes dados sejam utilizados para preparar uma nova geração do produto e ao mesmo tempo servir de melhoramento aos já existentes.

Os dados obtidos do modelo físico têm grande valor para o seu homólogo digital. Este valor pode ser aumentado quando o modelo digital liga-se à *IoT* (*Internet of Things*) [79].

2.1.3 A dimensão HMI (Human-Machine Interaction)

A cooperação entre humanos e máquinas tem sido essencial para o progresso da humanidade. Normalmente as máquinas operam num ambiente controlado e isolado. A monitorização da máquina é feita pelos humanos mas sempre num local diferente da máquina. Quando estas operam

num ambiente colaborativo com humanos as suas capacidades estão de certa forma limitadas devido ao risco de acidente. De forma a evitar estes mesmos acidentes, regras tiveram que ser criadas tornando assim a colaboração física entre humanos e robôs muito difícil de implementar. Para esta colaboração existir pode-se seguir por duas rotas. A primeira será criar uma barreira física entre a máquina e o humano de forma a evitar o contacto entre as duas entidades [41]. De forma a melhorar a acessibilidade do humano ao robô é possível substituir a barreira física por sensores que inibam a máquina quando um humano se encontra perto. A segunda rota será criar um *digital twin*. Com um *digital twin* é possível recriar no mundo digital o ambiente de funcionamento do robô sendo que este está ligado ao modelo físico e então tem acesso aos dados dos seus sensores. A interação entre o humano e máquina é então feita por intermédio do *digital twin* que permite uma monitorização detalhada do robô sem que estas duas entidades estejam no mesmo espaço físico. Quando um *digital twin* é criado, existem alguns requisitos que têm de ser cumpridos [41]:

- Permitir a comunicação de forma a que existam mecanismos de monitorização do robô;
- Permitir recolher dados dos sensores e ajustar o comportamento do robô com os dados recolhidos;
- A comunicação entre o modelo físico e o modelo digital tem que ter comandos e protocolos bem definidos;
- O armazenamento dos dados deve ser processada de acordo com o *big data*.

No entanto, é possível criar máquinas que não necessitem de monitorização humana. Estas máquinas têm que ser autónomas, ou seja, as suas acções não têm intervenção humana. Neste caso, o *digital twin* poderá ser a plataforma que permitirá o planeamento das acções que o robô irá executar. De forma a que o modelo digital ajude a decidir o que fazer para desempenhar determinada tarefa, é necessário adquirir o máximo de informação possível para permitir recriar o ambiente envolvente e perceber como é que o ambiente pode influenciar o estado actual do veiculo. Com acesso a estes dados, podemos recriar o ambiente e o próprio veiculo (ou seja construir o *digital twin*), correr simulações e analisar os resultados de forma a que as melhores acções sejam escolhidas considerando o estado actual do modelo físico. Poderá assim ser criado um mapa de acções que permite ao robô executar as suas tarefas.

2.1.4 A dimensão de entrada de dados

Como referido anteriormente, um *digital twin* é composto por duas entidades (modelo físico e modelo virtual) ligadas por uma "ponte"que serve para partilhar dados de forma bidirecional [54]. Este sistema troca dados entre as duas entidades de forma a melhorar o seu desempenho através do processamento e análise desses mesmo dados em tempo real. Esta interação não tem que ser a única forma de obter dados, dado que podem ser obtidos a partir de outras fontes. Em vez de estar isolado, o *Digital Twin* pode estar inserido numa rede e/ou num outro sistema e consumir dados dessa mesma rede e/ou sistema [84, 85]. Desta forma o volume de dados aumenta, o que pode

contribuir para a precisão da simulação do *Digital Twin* que por sua vez pode afinar o comportamento do modelo físico, melhorando o seu desempenho. Este aumento de dados consumidos cria também uma necessidade de usar outras tecnologias que permitam usar essa informação de forma eficiente, como é o caso da *Big Data* e da Inteligência Artificial [74].

Estes dados fornecidos pela rede onde o *Digital Twin* está inserido, podem conter o histórico de missões passadas, parâmetros de otimização do modelo físico e/ou resultados de simulações já executadas. Desta forma pode-se consumir dados não só através da geração de informação dos modelos físico e virtuais, como também de material obtido em simulações/missões já efetuadas [83]. A figura 2.1 ilustra esta relação.

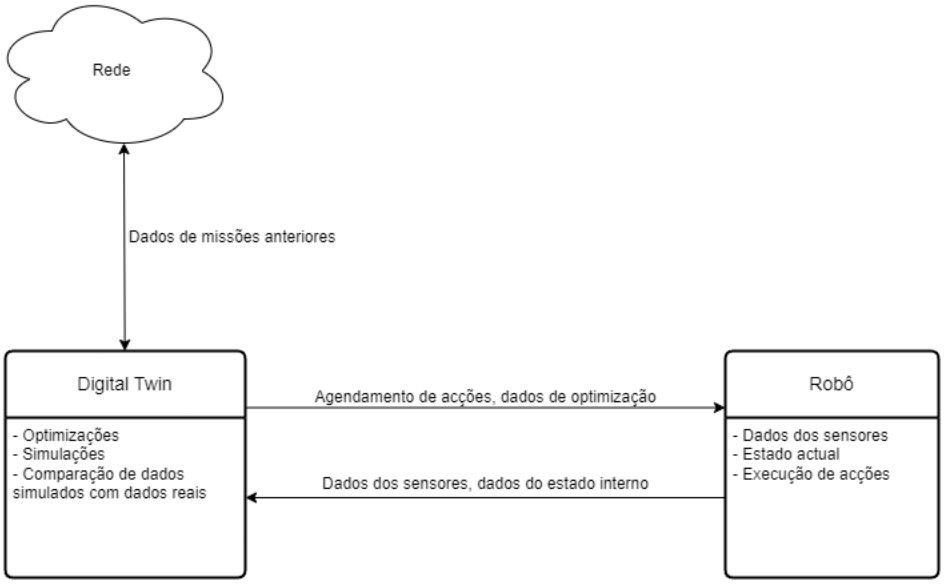

Figura 2.2: Relação entre *Digital Twin* e o robô

2.2 Robótica

2.2.1 Veículos marinhos autónomos

A curiosidade humana é o que leva a civilização a alcançar patamares inimagináveis. O oceano sempre fascinou a Humanidade levando esta a desenvolver meios que a permitem explorar esse meio. Numa primeira fase a superfície dos oceanos foi a primeira a ser navegada fazendo a ponte entre os diversos continentes. No entanto, faltava explorar o fundo marinho que ainda era envolto em mistério. Com a aparição do submarino este novo ambiente tornou-se acessível à exploração humana. Nos dias de hoje existem vários aparelhos que nos permitem explorar em detalhe os oceanos do planeta. Existem dois tipos de veículos, os de superfície e os submarinos. Neste projeto o foco será nos veículos marinhos não tripulados. Estes podem ser veículos controlados remotamente ROV (*Remotely Operated Vehicles*) ou veículos que operam sem intervenção humana, ASV (*Autonomous Surface Vehicles*) e AUV (*Autonomous Underwater Vehicles*) [89]. Os ROVs são veículos que estão disponíveis em maior quantidade no mercado. Estes veículos permitem mergulhar a grandes profundidades sem que seja necessário que uma pessoa esteja fisicamente dentro dele. Como o próprio nome indica os ROVs são controlados remotamente sendo que o operador está numa sala de operações a comandar o veículo. A operação do veículo é conseguida através de um cabo umbilical que liga directamente a sala de operações ao ROV. Este tipo de veículos têm um grande custo operacional o que leva a procurar outras alternativas. Os AUV e ASV vêm colmatar algumas desvantagens dos ROVs. Não sendo necessário uma pessoa dedicada a comandar o veículo, permite reduzir o custo de operação. Também não usam cabos, pois a comunicação entre o veículo e os controladores é feita através de sinais de rádio. Apesar de bastante diferentes, estes veículos têm características centrais semelhantes, apresentando:

- um sistema de controlo que permite ao veículo mover-se pelo meio [88].
- um sistema de sensores que permite a recolha de dados, sendo que dentro deste existem ainda três subsistemas: sensores de navegação, que permitem perceber o movimento do veículo, sensores de missão para a recolha de dados do ambiente de operação e por fim, sensores de sistema que servem para diagnosticar o próprio veículo.
- um sistema de comunicação para que haja a partilha de dados da missão, permitindo a sua monitorização e supervisão a partir de um centro de controlo.
- um sistema que fornece a energia necessária para todos os seus componentes.

Estes veículos podem desempenhar várias funções, desde do uso militar até à indústria do petróleo. É possível resumir os potenciais usos dos robôs marinhos da seguinte forma [88]:

- **Ciência** — mapeamento do fundo do mar; resposta rápida a eventos oceanográficos e geotermais; amostragem geológica.

- **Ambiente** — monitorização de longo período (poluição, fuga de radiação, derramamento de hidrocarbonetos); remediação ambiental; inspeção de estruturas subaquáticas, incluindo barragens, oleodutos, etc.
- **Uso militar** — pesquisa e eliminação de minas em águas pouco profundas; proteccção de infraestruturas; guerra anti-submarinos.
- **Mineração oceânica e indústria do petróleo** — Pesquisa oceânica e avaliação de recursos, manutenção e construção de estruturas submarinas.
- **Outras aplicações** — inspecção de navios; inspecção de centrais nucleares; comunicação subaquática; instalação de cabos de energia; entretenimento (excursões subaquáticas); indústria piscatória.

2.2.2 Arquitectura de software de um sistema robótico

Um robô é uma máquina capaz de alterar o seu meio envolvente através de ações que são executadas dentro de determinadas regras comportamentais intrínsecas e também da aquisição de dados referentes ao seu estado e ao seu meio [38]. De forma a ter sucesso a executar essas mesmas tarefas o robô tem que possuir uma série de componentes, sendo possível dividir em três grupos, controlador, sensores e atuadores [38].

De forma a poder executar estas tarefas o robô necessita de um sistema que faça o processamento de toda a informação que adquire. Existem *frameworks* de *software* dedicadas para os sistemas robóticos.

2.2.2.1 ROS

ROS é um *middleware* para robôs *open-source* que age como um sistema operativo e uma camada de abstração ao *hardware*. Isto faz com que a programação de robôs seja mais fácil, pois os programadores não se têm que preocupar com *drivers*. ROS também disponibiliza bibliotecas e *drivers* para vários robôs e sensores. Estes dados são designados por *topic* e fornecem uma camada de abstração que podem ser facilmente manipulados. O ROS segue o desenvolvimento de *software component-based* o que torna o sistema modular, extensível e flexível [67].

O *ROS* [24] tem 3 níveis, *Filesystem*, *Computation Graph* e *Community*. Adicionalmente a estes 3 níveis, possui também 2 tipos de nomes, *Package Resource Names*, que definem a estrutura hierárquica que é usada para todos os recursos no nível *Computation Graph*, e *Graph Resource Names*, que é usado no nível *Filesystem* para referenciar de forma mais simples ficheiros e dados guardados no disco rígido.

Filesystem aborda os recursos ROS que se podem encontrar em disco, como *Packages*, responsáveis por organizar *software* no *ROS*. *Metapackages* são *packages* especializados usados para representar um grupo de outros *packages*. *Package Manifests* providenciam informação sobre os *packages*, como o nome, versão, descrição, etc. *Repositories* são uma coleção de *packages* que partilham o mesmo sistema de versionamento. *Message types* definem a estrutura de dados para

as mensagens usados no *ROS*. Por fim, *Service types* definem a estrutura de dados dos pedidos e respostas nos serviços presentes no *ROS*.

Computation Graph é uma rede *peer-to-peer* de processos *ROS* que processam dados em conjunto. Os conceitos básicos deste nível são os seguintes: *Nodes*, processos que fazem a computação. *Master* que providencia o registo do nome e a pesquisa para o grafo. *Parameter Server* permite que dados sejam guardados através de uma chave numa localização central. *Messages* estruturas de dados que os *nodes* usam para comunicar. *Topics* são estruturas que têm um determinado nome que reflete o conteúdo de uma mensagem, este tópico pode ser subscrito por um node para obter uma mensagem desse tópico específico. *Services* são definidos por um par de estruturas de mensagens, a mensagem pedido e a mensagem de resposta. Um *node* pode definir um serviço em que um cliente envie um pedido a esse serviço e fique à espera da resposta. *Bags* são um formato que permite guardar e reproduzir de novo dados de mensagens *ROS*.

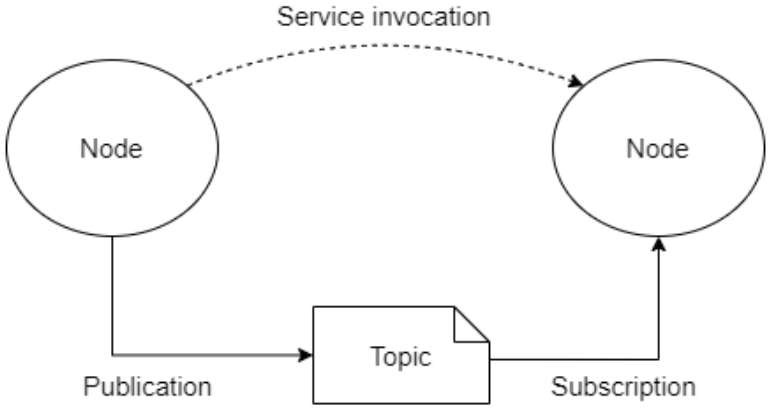

Figura 2.3: Conceitos Básicos *ROS* [25]

Community contém vários recursos que permitem a várias comunidades separadas trocar *software* e conhecimento. *Distributions*, à semelhança das distribuições *Linux*, são *software* com versões que podem ser instalados. *Repositories* são repositórios onde as instituições podem desenvolver e lançar o seu *software*. *The ROS Wiki* é fórum principal onde se encontra a documentação do *ROS*. *Bug Ticket System* sistema de *tickets* para reportar *bugs*. *Mailing Lists* é um sistema de *mailing* que é usado para manter os utilizadores actualizados sobre o *software ROS*. *ROS Answers* é uma página web que contém perguntas e respostas acerca do *ROS*. *Blog* é uma página web que possui informação actualizada, incluindo imagens e vídeos, do *ROS*.

2.2.2.2 Orca

Orca é uma *framework* que implementa os principios da CBSE (*Component-Based Software Engineering*) à robótica. Estes princípios trazem os seguintes benefícios [62]:

- **Modularidade** — Um sistema modular controlado e com dependências explícitas.

- **Substituibilidade** — Desenvolvimento de sistemas flexíveis em que cada componente pode ser desenvolvido independentemente e substituído.
- **Reutilização** — Possibilidade de usar diversos componentes que foram testados e usados em diferentes projetos.

Orca consiste fundamentalmente em duas partes, infraestrutura, onde se define as interfaces e o suporte de comunicação entre componentes, e o repositório de componentes que armazena os componentes reutilizáveis [62].

- **Infrastrutura**
  - **Middleware**
    Chama-se *middleware* o *software* que atua numa camada intermédia nas aplicações dos componentes. Orca não tem um *middleware* específico. Possui no entanto padrões de comunicação que permitem o envio de dados e objectos como mensagens.
  - **Utilities**
    Orca possui uma série de *utilities* que permite facilitar a configuração, desenvolvimento e monitorização dos componentes.
- **Repositório de componentes**
  Orca disponibiliza um repositório online onde se pode encontrar componentes reutilizáveis e bem documentados.

2.2.2.3 OPRoS

OPRoS é também uma *component-based framework* que consiste em vários componentes e actua como um processo no sistema operativo. A *framework* funciona como um recipiente que providencia a execução, a gestão do ciclo de vida, configuração e comunicação com os vários componentes [39].

A *framework* apresenta as seguintes características [56]:

- código *open-source* e licenças grátis para funcionalidades básicas.
- disponibiliza componentes standard.
- oferece um *IDE* e um ambiente de simulação.
- carrega e executa conteúdo e componentes, agenda a execução dos componentes e realiza eventos e faz o tratamento de erros.
- importa a tecnologia do servidor para melhorar a performance do *software* do robô e ultrapassar limitações de *hardware*.
- oferece uma ferramenta para avaliação e de testes do *software*.
- disponibiliza um método fácil para ligar comunicações *middleware* como *UPnp* e *CORBA*.

2.2.2.4 SHAGE

SHAGE contém dois elementos centrais, módulos que são coleções de componentes que suportam várias funcionalidades e os repositórios que servem para a aquisição de dados e a sua utilização [39].

Os módulos estão instalados diretamente nos robôs e contém sete partes, *Monitor, Architecture Broker, Component Broker, Decision Maker, Learner, Reconfigurator* e os repositórios internos [65].

O monitor é responsável por observar a a situação actual do ambiente e avaliar o resultado da adaptação que a *framework* utiliza. O *architecture broker* procura arquitecturas baseadas nas estratégias de reconfiguração e apresenta composições de componentes para essas mesmas arquitecturas, componentes esses que são recolhidos pelo *component broker*. O *component broker* vai procurar componentes concretos nos repositórios e reordená-los numa determina arquitectura. O *decision maker* escolhe a melhor arquitectura entre a lista encontrada pelo *architecture broker* e escolhe também a melhor composição de componentes que o *architecture broker* compôs. o *learner* vai acumular recompensas do *evaluator*, dentro do módulo monitor. São com estas recompensas acumuladas que o *decision maker* baseia as suas escolhas. *learning data* é um repositório de conhecimento para o *learner*. O *reconfigurator* faz a gestão e a reconfiguração da arquitectura de *software* do robô baseando-se na arquitectura e na composição de componentes escolhidas pelo *decision maker*. Os repositórios internos consistem em duas partes, repositório ontológico e o repositório de componentes. O repositório ontológico contém estratégias das reconfigurações das arquitecturas que descrevem as funcionalidades do robô e contém também componentes ontológicos que descrevem as características de um componente. O repositório de componentes armazena os componentes que estão prontos a ser usados pela arquitectura de *software* do robô.

A parte exterior da *framework* são uma coleção de servidores que contêm repositórios externos. Cada servidor tem um repositório ontológico e um repositório de componentes. Repositórios internos do robô fazem pedidos a estes repositórios externos, requisitando novas ontologias e componentes quando o robô não consegue adaptar-se a uma nova situação. Os repositórios externos também atualizam o repositórios internos do robô globalmente. O gestor de repositórios é instalado em cada servidor e possui ferramentas que permitem adicionar e eliminar ontologias e componentes [65].

2.2.2.5 PRISM

PRISM é uma *framework* que providencia uma *middleware* de suporte para a uma implementação eficiente, desenvolvimento e execução de elementos arquitecturais [39].

Prism-MW é uma plataforma *middleware* implementada em Java e C++. Cada objecto da arquitectura desempenha o papel de contentor para a topologia de objectos *Component* e *Connector*. Os componentes implementam serviços de aplicação enquanto os conectores implementam serviços de interação. Estes dois objectos comunicam entre si através de troca de eventos [52].

A arquitectura *Prism-MW* apresenta quatro camadas. A primeira disponibiliza a abstração de baixo nível necessária para comunicação com o *hardware*, a segunda consegue incorporar diversas bibliotecas robóticas, a terceira implementa os sistemas de *software* e a quarta permite o uso de serviços avançados, como a tolerância de erros e monitorização [52].

Comparando todas as *frameworks* referidas, o *ROS* consegue oferecer tudo o que é necessário para desenvolver um sistema robótico. Embora todas as opções apresentem pontos em comum, como o desenvolvimento baseado em componentes, possibilidade de integração de *plugins* e repositórios para os componentes desenvolvidos, o *ROS* contém uma comunidade bastante grande que contém reporte de *bugs*, fóruns de discussão e uma documentação detalhada. Aliado a isto é utilizado na industria de forma extensiva, desde projectos pessoais a projectos de larga escala na industria robótica, permitindo o desenvolvimento multi-plataforma e multi-domínio, tornando-se numa ferramenta verdadeiramente versátil.

Capítulo 3

Estado da Arte

3.1 Aplicação dos digital twins na robótica

Os *Digital Twins* podem ser versáteis o suficiente para serem usados em variadíssimas aplicações. A partir do conceito base (um produto real, um produto virtual e uma ponte de ligação que permita a troca de dados entre as duas entidades [54]) é possível aplicar um *Digital Twin*, praticamente a qualquer área de atividade.

3.1.1 Indústria

Na indústria já é possível ver a digitalização a ganhar terreno, através de sistemas complexos que controlam as grandes linhas de produção incluindo cada um dos passos intermédios que um produto tem que passar até ficar concluído. Os *Digital Twins* conseguem trazer uma mais valia neste contexto industrial, sendo que podem representar um papel importante em todas as fases do ciclo de vida do produto [55]:

1. Os designers de produto podem fazer uso dos *Digital Twins* para simular e analisar a fiabilidade do produto, usando depois estes dados para fazer melhorias e atualizações ao produto.
2. Para os fabricantes, avaliações de fiabilidade podem ser desenhadas através de *Digital Twins*.
3. Na manutenção é possível usar os *Digital Twins* para fazer previsões das manutenções futuras. Quando o produto estiver em fim de vida pode ser feita uma avaliação ao processo de fabrico do produto, ajudando na produção de uma nova geração.
4. Para os utilizadores os *Digital Twins* podem ser usados para fazer previsões da disponibilidade do produto assim como avaliar formas mais convenientes de uso.

3.1.2 Medicina

O uso de *Digital Twins* na saúde pode parecer um pouco incompatível, no entanto é possível fazer uso desta tecnologia de forma a ser muito útil nesta área. Podemos ver os vários usos dos *Digital Twins* em diversas áreas da medicina [55, 69].

3.1.2.1 Tratamento de doenças

Uma das funções mais importantes da medicina é precisamente arranjar soluções para os problemas de saúde da sociedade. Embora a tecnologia ainda esteja longe de substituir os profissionais de saúde, esta pode auxiliar estes trabalhadores a fazer o seu trabalho com melhor qualidade e de forma mais eficiente. Os *Digital Twins* podem ser usados como companheiros auxiliares. À semelhança de outras áreas, é possível usar esta capacidade de simulação para prever comportamentos de doenças e prevenir certas complicações derivadas a essas doenças [45]. Os *Digital Twins* também permitem trazer uma atendimento mais personalizado ao utente podendo ser usado o histórico clínico do mesmo como dados de entrada que o *Digtal Twin* pode consumir [42]. Com isto, torna-se plausível criar um ambiente altamente personalizado ao utente, desde o agendamento de consultas até mesmo à previsão da reação do corpo do utente aos tratamentos ou medicamentos [55].

3.1.2.2 Gestão da saúde

A gestão da saúde pode beneficiar desta tecnologia para criar processos médicos e organização estrutural mais eficientes. Comparando um hospital a uma fábrica, torna-se possível usar *Digital Twins* para melhorar todo o conjunto. É possível assim tornar mais eficiente os serviços médicos tornando-os mais acessíveis data a automação do sistema e também melhorar a gestão de pessoal médico de forma a que nunca falhe profissionais de saúde mantendo uma carga de trabalho adequada a cada um deles [55].

3.1.2.3 Planeamento desportivo

Para além de os *Digital Twins* poderem ser usados para prever o comportamento de doenças nos utentes, estes também podem ser usados para criar planos de treino personalizados a atletas e prever o seu potencial físico para cada plano de treino. Através de dispositivos *wearable*, consegue-se recolher dados do atleta praticamente em tempo real e fornecer estes dados ao *Digital Twin* podendo depois correr simulações, prevendo a performance e, caso seja necessário, alterar comportamentos de forma a que o atleta obtenha os melhores resultados possíveis [55].

3.1.3 Aviação

Na aviação os *Digital Twins* podem ter um papel preventivo. Estes servem para verificar alterações nas estruturas da aeronave que ponham em risco o próprio funcionamento do aparelho. Para além de detetar estas mudanças estruturais, ativam também procedimentos para reparar os mesmos, apoio à decisão, otimização e diagnóstico [86]. Através da recolha destes dados é possível chegar a um nível de prevenção ainda mais alto, conseguindo prever quando estas alterações podem ocorrer tendo em conta mudanças anteriores e que condições a estrutura suportou. Assim é possível fazer uma intervenção mais cirúrgica, reduzindo o custo e o tempo de desenvolvimento e manutenção [42]. Sendo a aviação uma área em que o produto final tem que ter um certo nível de

redundância de forma a que seja considerado seguro para a sua operação, é importante que existam mecanismos poderosos que previnam o mau funcionamento do aparelho, algo que os *Digital Twins* podem ajudar a alcançar.

3.1.4 Robótica

A área da robótica é onde se consegue visualizar melhor os benefícios da utilização dos *Digital Twins*. É possível desenvolver um robô e, com recurso ao um *Digital Twin*, melhorar a performance do mesmo ou desenvolver novas funcionalidades que ampliem o leque de ferramentas que o robô tem ao seu dispor.

Como um robô pode ter vários fins e servir várias áreas, é possível criar um robô que ajude ou que possa mesmo substituir um médico num contexto de operação, criar um robô mais competente numa linha de montagem ou criar um modelo de aeronave não tripulado que ajude a desenvolver um veículo mais eficiente. Num contexto mais específico, já é viável enveredar por alguns destes desafios, como por exemplo, o uso de um robô para cirurgia à distância com recurso a um *Digital Twin* [68]. Embora seja somente uma prova de conceito, já se consegue perceber o potencial do uso deste tipo de tecnologia. Aqui os autores do artigo pretenderam criar um protótipo de um robô que consegue ser controlado por um médico à distância recorrendo à realidade virtual e à rede móvel. Esta abordagem é especialmente útil quando não existe tempo suficiente ou não é viável juntar o utente e o pessoal médico no mesmo local físico.

Esta ideia de obter dados, processar esses mesmos dados no *Digital Twin* e analisar os resultados obtidos de forma a melhorar o modelo real, pode ser aplicado praticamente a qualquer área da nossa sociedade, desde a engenharia ao planeamento urbano [43, 49], até mesmo ao mundo dos negócios e financeiro [55, 60].

3.2 Ferramentas de simulação existentes em robótica

O mercado apresenta várias soluções para a simulação robótica, dos quais se destacam três: *UWSim*, *Gazebo* e *MORSE*.

3.2.1 UWSim

UWSim é um *software* que permite a simulação e visualização de missões robóticas marinhas. Usa as bibliotecas OSG (*OpenSceneGraph* e *osgOcean*) para a renderização. Sendo modular, permite que seja facilmente integrado com outras aplicações. Suporta vários veículos que são descritos por um ficheiro XML segundo a norma URDF (*Unified Robot Description Format*). Os sensores que o simulador disponibiliza são os seguinte [75, 80]:

- **Câmara** — Disponibiliza imagens virtuais que podem ser usadas para desenvolver algoritmos;
- **Alcance de câmara** — Profundidade da imagem;

- **Sensor de distância** — Mede a distância ao objeto mais próximo, numa determinada direção;
- **Apanhador** — Simula o apanhar de um objeto quando este está dentro de uma distância pré-definida;
- **Pressão** — Mede a pressão;
- **DVL** — Estima a velocidade linear do veículo;
- **IMU (*Internal Measurement Unit*)** — Estima a orientação do veículo em respeito ao mundo;
- **GPS** — Disponibiliza a posição do veículo no mundo, mas só funciona quando está perto da superfície;
- **Multibeam** — Simula vários sensores de distância que fornecem as distâncias dos objetos mais próximos num determinado plano e com incrementos constantes do ângulo;
- **Força** — Estima a força e a torção aplicada a um componente do veículo;
- **Projetor de luz** — Projeta um laser ou luz no mundo simulado;
- **Draga** — Limpa e retira a lama de objetos enterrados.

Utiliza o motor de físicas *Bullet* que permite simular contactos e forças. Torna-se assim possível simular as dinâmicas dos veículos submarinos e as interações com o mundo simulado podem ser criadas com recurso a sensores de força que agem como forças externas ao veículo. O simulador também faz uso do *ROS* (*Robot Operating System*), permitindo aceder e atualizar a posição do veículo ou velocidade, mover braços ou adquirir os dados dos sensores simulados, como as imagens geradas pelas câmaras virtuais.

3.2.2 Gazebo

O *Gazebo* [66] é um *software open-source*, disponível de forma livre e de código aberto. Como resultado, dispõe de uma comunidade que está constantemente a fazer evoluir o *software* de forma a satisfazer as necessidades dos utilizadores. *Gazebo* oferece um ambiente rico que permite desenvolver e testar rapidamente sistemas multi-robô, sendo eficaz, escalável e é uma ferramenta simples.

Direcionado para a simulação 3D de alta fidelidade das dinâmicas de ambientes exteriores, *Gazebo* complementa duas ferramentas o *Player* (um servidor de dispositivos em rede) e o *Stage* (simulador de grandes populações de robôs móveis em ambientes 2D complexos) [66]. Apesar de ser totalmente compatível com o *Player*, *Gazebo* não substitui o simulador *Stage*, visto que a simulação é tridimensional, requer muitos recursos computacionais o que o torna mais apto para simulações com poucos dispositivos, sendo o *Stage* a escolha mais acertada para simulações com um maior número de robôs.

3.2.2.1 Motor de Físicas

O motor de físicas usado pelo *Gazebo* é o ODE (*Open Dynamic Engine*). Este motor permite simular as dinâmicas e as cinemáticas associadas ao corpo rígido articulado. O *Gazebo* usa uma camada de abstração entre o ODE e os seus modelos de forma a facilitar a criação de objetos. Com isto é também possível alterar o motor de físicas caso surja uma alternativa melhor [66].

3.2.2.2 Visualização

Para a visualização, o *Gazebo* usa *OpenGL* e GLUT (*OpenGL Utility Toolkit*). *OpenGL* é uma biblioteca que é usada para a criação de aplicações interativas 2D e 3D. Muitas das características desta biblioteca tiram partido da capacidade de processamento da placa gráfica deixando o processador livre para outras tarefas como o processamento dos cálculos do motor de físicas. O *GLUT* é um sistema de janelas para o aplicações *OpenGL*. Este sistema permite visualizar as imagens renderizadas e receber input do utilizador através de periféricos básicos, como um teclado e rato [66].

3.2.2.3 Sensores

O *Gazebo* possui várias ferramentas que permitem ao robô interagir com o mundo simulado. A seguir são enumeradas as estruturas base que compõem um robô[66]:

- **Modelo** — É um objeto que mantém a representação física. Pode ser desde uma geometria simples até um robô complexo. Estes modelos são compostos pelo menos por um corpo rígido, zero ou mais articulações e sensores e, por fim, interfaces que facilitam o fluxo de dados;
- **Corpos** — Representam a base de um modelo. A sua representação pode ter qualquer forma geométrica, desde cubos, esferas, cilindros, planos e linhas. Cada corpo tem associada uma massa, fricção, *bounce factor* e propriedades de renderização como a cor, textura, transparência, etc.
- **Juntas** — Permitem ligar corpos entre si de forma a criar relações dinâmicas. Existem vários tipos de juntas que incluem dobradiças (permite a rotação de um ou dois eixos), *slider* (para a translação num único eixo), *ball* e *socket* (rotação nos três eixos) e, por fim, juntas universais (para a rotação de duas juntas perpendiculares). Para além de conectar dois corpos, as juntas podem-se comportar como motores. Quando uma força é aplicada, a fricção entre o corpo que está conectado à junta e os outros corpos faz com que haja movimento. No entanto é necessário cautela, pois quando se conecta demasiadas juntas a um único modelo, este e a simulação podem perder estabilidade caso os parâmetros escolhidos sejam errados.
- **Interfaces** — É o que permite o acesso aos controlos dos modelos. Comandos enviados via a interface podem ordenar o modelo a mover juntas, mudar as suas configurações ou pedir

os dados dos seus sensores. As interfaces não impõem restrições no modelo, permitindo assim ao modelo interpretar os comandos da forma que ache mais correta.

Os sensores permitem recolher dados sobre o ambiente que envolve o robô. Os sensores no *Gazebo* são dispositivos abstractos que não possuem uma representação física até ser incorporados num modelo. Isto permite reutilizar sensores de outros modelos reduzindo o código. Existem três sensores implementados, câmara, sensor de proximidade e sensor que contabiliza a distância percorrida[66].

O *Gazebo* ainda permite usar interfaces de terceiros através das suas bibliotecas. Normalmente é usado em conjunto com o *Player* o que faz com que, do ponto de vista do utilizador, o modelo simulado do *Gazebo* seja igual ao robôs reais. Isto é possível pois o *Player* trata o *Gazebo* como um dispositivo normal capaz de receber e enviar dados. Embora seja comum o uso conjunto com o *Player*, o *Gazebo* não está de todo limitado a este. Como pode correr de forma independente, nem é necessário uma conexão direta às bibliotecas[66].

3.2.3 MORSE

O MORSE é um simulador construído por cima do *software Blender*, que oferece um sistema modular e genérico, que em vez de se focar numa característica específica da simulação robótica, representa o robô como um todo. Pretende assim ser versátil, flexível e reutilizável [50].

Blender possui um motor de jogo que permite criar simulações dinâmicas. O MORSE não só utiliza este motor, como também utiliza a composição de ficheiros do *Blender* para criar as simulações. Um projeto no *Blender* pode referenciar um objeto *Blender* que está guardado num outro ficheiro. Sempre que este objeto é atualizado, todas as cenas que dependem dele também são atualizadas com as alterações. O MORSE usa este conceito modular para disponibilizar uma biblioteca de componentes simples que podem ser combinados entre si. Cada componente consiste num ficheiro *Python* e um ficheiro *Blender*. O ficheiro *Python* define uma classe objeto para o tipo de componente, as suas variáveis de estado, dados e métodos. O ficheiro *Blender* especifica as propriedades visuais e físicas do objeto dentro da simulação, como a sua cor, textura ou mesmo propriedades dinâmicas como massa ou fricção [50].

Existem três tipos de componentes robóticos no MORSE [50]:

- **Sensores** — Recolhe dados do mundo simulado, copiando a funcionalidade dos sensores reais usando a lógica do motor de jogo do *Blender*;
- **Atuadores** — Produz uma ação através dos robôs ou componentes que lhes estão associados. Podem mover o robô dependendo de um determinado parâmetro (coordenadas ou velocidade angular/linear). Outros atuadores pode afetar outro componentes, como a posição de braços.
- **Robôs** — Plataformas onde os sensores e atuadores estão montados. Também definem as propriedades físicas (tamanho, peso, fricção, mobilidade, limite de colisão) do robô virtual.

Os braços robóticos também se inserem nesta categoria. Estes são compostos por vários segmentos que podem ser articulados e possuem atuadores especiais.

Para além destes componentes, existem também três outras classes disponíveis [50]:

- **Cenários** — Ambientes onde o robô vai interagir durante a simulação. O MORSE disponibiliza alguns exemplos como um cenário exterior com edifícios e árvores assim como uma sala interior. As cenas são *Blender scenes* simples e qualquer ambiente modelado anteriormente pode ser usado;
- **Middlewares** — Os canais de comunicação entre o simulador e o *software* avaliado são configurados através de objetos *Blender* especiais que tratam da ligação com os sensores simulados e os atuadores;
- **Modificadores** — Os sensores simulados produzem dados perfeitos obtidos do mundo virtual. Como no mundo real isto dificilmente acontece é possível adicionar modificadores aos dados através de funções que acrescentem ruído de forma a tornar os dados da simulação mais realistas. Estes modificadores expõe métodos que são inseridos no fluxo de dados entre o simulador e o *software* avaliado.

O MORSE não está "preso"a nenhum *middleware*, podendo mesmo usar vários em paralelo. Para conseguir atingir esta modularidade, o MORSE é inspirado numa ferramenta chamada *GenoM 3* [73, 50], que permite gerar módulos de *software* que podem ser compilados de forma independente do *middleware*, permitindo assim manter o código do componente completamente separado deste. Quando um módulo é criado, este pode ser conectado ao *software* do robô e trocar dados através de estruturas de dados e pedidos.

3.3 Revisão Científica

Existem várias formas de gerar geometria tridimensional. As diferenças começam logo pelo tipo de dados que cada método usa, podendo ser nuvem de pontos provenientes de um sensor *LiDAR*, através de um conjunto fotos, ou mesmo a junção dos dois. Fazemos uma breve abordagem deste métodos de reconstrução de geometria [64].

Métodos de reconstrução de superfície explícita triângulada são um dos primeiros métodos a surgir para gerar uma geometria 3D. Têm como base as triângulações de *Delaunay* e os diagrama de *Voronoi*. Um dos algoritmos que usa estes cálculos é o *Ball pivoting* que será descrito em maior detalhe no capítulo 4.

Métodos de reconstrução de superfície explícita paramétrica têm como base a combinação e a deformação de estruturas primitivas, como um cubo ou cilindro, de forma a que encaixem num determinado *dataset*.

Métodos implícitos de reconstrução de superfície fazem uso de funções para gerar a geometria, funções essas que são chamadas depois do cálculo de triangulação. Este método em específico

foca-se na aproximação da geometria através de uma função ou combinação de várias funções. Neste caso existe um conjunto de pontos que não representam necessariamente a superfície do objecto, daí que seja necessário calcular uma aproximação dessa superfície. Um dos algoritmos que se encaixa neste método é o *Poisson surface reconstruction*, que está descrito com maior detalhe no capítulo 4.

Estimativa de profundidade de perspectiva única [61] calcula a informação de profundidade de um ambiente através dos pixeis de uma imagem desse mesmo ambiente. É possível criar um modelo com a informação correcta de cor, mas só a partir de um determinado ângulo.

Estimativa de profundidade multi-perspectiva [61] reconstrói um determinado ambiente através de uma sequência de imagens. Por norma estas imagens são obtidas por uma câmara que orbita à volta do objecto de interesse. Este tipo de reconstrução gera um modelo detalhado com texturas de definidas.

Simultaneous Localization and Mapping (SLAM) [61] permite calcular um determinado ambiente desconhecido e suas localização no mesmo. À medida que o sistema 'anda' pelo ambiente vai calculando a sua posição no mesmo e reconstruindo-o através de imagens que vai obtendo.

De forma a poder ter uma ideia mais completa das ferramentas que já existem no mercado, foi efetuada uma pesquisa na literatura por trabalhos que abordassem a geração de uma geometria 3D através de sensores e visualização da geometria 3D. Segue-se um breve resumo dos 13 artigos seleccionados.

Sánchez *et al.*[81] implementa uma aplicação que usa uma nuvem de pontos proveniente de um sensor no *Gazebo* que recria um sensor real existente num robô. Estes dados são enviados para o *MatLab* onde são processados e visualizados, não existindo um modelo gerado, mas sim uma nuvem de pontos com informação de cor. Esta interação entre *Gazebo* e o *MatLab* tem como interface o *ROS*.

Serdel *et al.*[77] propõe uma implementação baseada em *ROS* e *ROS2* em que processa a nuvem de pontos proveniente de um sensor *LiDAR (Light Detection and Ranging)*, criando uma geometria 3D com as texturas corretas do ambiente A reconstrução faz uso do pacote *open-source* presente no *ROS*, *Voxblox ROS*. Texturas essas que são obtidas através de fotografias de uma câmara presente no robô. O resultado é uma geometria com bastante detalhe e muito semelhante ao ambiente envolvente.

Liu *et al.*[71] apresenta uma proposta que passa pela recolha dos dados do ambiente através de uma câmara *RGB-D*, reconstruindo essa nuvem de pontos recolhida pela câmara diretamente no computador que o robô possui (*Jetson TX2*). Neste computador de bordo é feita uma reconstrução primitiva da geometria que depois passa para um computador fixo para refinar essa mesma geometria. Para gerar a geometria primitiva a biblioteca *PCL* é usada. Esta abordagem, embora criando uma representação precisa da geometria assim como uma textura correta da superfície, como passa por dois processos - um de reconstrução e outro de otimização da geometria - faz com que comprometa o tempo de processamento em tempo real, ou seja, o processamento é efectuado no imediato.

Hu *et al.*[58] propõe um protótipo com três componentes *off-the-shelf* (sensor *Azure Kinect*, *Jetson TX2* e um *smartphone Pixel3XL*) em que são estudados os *bottlenecks* na captura e *streaming* de nuvens de pontos densas em dispositivos móveis em tempo real, através de uma conexão TCP entre o computador e o cliente receptor. Neste caso é visualizada no dispositivo móvel a nuvem de pontos com dados de cor e não uma geometria, conseguindo no entanto fazer essa visualização em tempo real, como referido anteriormente.

You *et al.*[87] desenvolve uma aplicação que permite visualizar um ambiente virtual em realidade aumentada provenientes de um sensor *LiDAR* colocado num robô e sensores presentes no *headset* de realidade virtual. A ideia é obter uma nuvem de pontos de baixa densidade e enviar esta dados para um *headset* de realidade virtual, juntando as duas fontes de informação (*LiDAR* e *headset*) que vai depois apresentar de forma alinhada com o ambiente real. O que é visualizado no final é uma nuvem de pontos e não uma geometria. Esta transferência de dados é feita através de uma ligação por cabo *ethernet* entre o computador presente no robô e um computador fixo que faz o processamento da nuvem de pontos. Estes dados conseguem ser visualizados em tempo real devido à sua reduzida densidade e complexidade.

Lee *et al.*[70] apresenta uma forma de trabalhar colaborativamente através Realidade Aumentada. É feita uma distinção entre o ambiente em que o utilizador se encontra e o objeto que utilizador interage. O processo de criação do ambiente colaborativo passa pelo mapeamento do ambiente envolvente através de um sensor *LiDAR* e a sua consequente reconstrução em três dimensões, em tempo real. O ambiente colaborativo não é actualizado, pois neste caso o ambiente é gerado para dar uma percepção do meio envolvente ao utilizador que usa o equipamento de realidade virtual. O objeto que vai ser interagido pelos os utilizadores é pré-processado e colocado no ambiente gerado. Estes dados são depois enviados para o dispositivo de Realidade Aumentada. Esta solução faz com que um utilizador consiga dar instruções a outro utilizador que esteja no local físico e orientá-lo numa eventual reparação de um objeto.

Heo *et al.*[57] desenvolveu um método de compressão para uma nuvem de pontos *LiDAR* chamado *FLiCR*. Este método permite enviar nuvens de pontos de forma rápida e leve via online o que faz que seja mais adequado para aplicações em tempo real. O foco são dispositivos móveis beneficiando diretamente desta ferramenta que faz com que seja necessária menos capacidade de processamento e consequentemente menos desgaste da bateria.

Hughes *et al.*[59] propõe um sistema em tempo real de perceção espacial chamado *Hydra*. *Hydra* cria um cenário 3D em grafo através de dados de um sensor presente num robô. Este cenário em grafo permite ajudar a perceção espacial do robô contendo vários níveis de abstração e de detalhe.

Soman *et al.*[78] apresenta uma *cloud framework* que visa a permitir monitorização de progresso numa construção entre o sítio dessa mesma construção e os escritórios. Nos escritórios fazem um modelo *BIM* do projeto que depois é enviado para o servidor. Os trabalhadores no local de construção terão acesso a este modelo num equipamento móvel podendo descarregá-lo e usar Realidade Virtual para visualiza-lo. É possível também utilizar os equipamentos móveis para mapear o local e assim ver o modelo através de Realidade Aumentada. Propõe-se que esta

reconstrução seja feita directamente no equipamento móvel.

Ali *et al.*[40] propõe um sistema monitorização de um local de construção, o *iVR*. Recorre ao sensor *Kinect V2* para fazer o mapeamento do local que depois se traduz numa reconstrução da geometria em 3D. Para a reconstrução da geometria são usados dois algoritmos, *alpha shapes* e *ball pivoting*. Essa geometria é de seguida partilhada para um servidor onde os trabalhadores nos escritórios podem fazer a análise da obra adicionando dados relevantes ao modelo para os trabalhadores no local. Este podem visualizar esse *feedback* através de Realidade Aumentada.

Tagarakis *et al.*[82] implementa reconstrução de ambientes agrícolas de modo a que seja possível simular operações robóticas autónomas. O foco foi a aquisição de densas nuvens de pontos de várias nogueiras com recurso a um veiculo terrestre não tripulado, equipado com uma camera *RGB-D*. Estas nuvens de pontos foram depois importadas para um ambiente *Gazebo/ROS* onde serviram como teste para simular a movimentação autónoma de um robô.

Cheng *et al.*[48] desenvolve uma aplicação que permite recriar geometria 3D de ambientes onde ocorreram desastres naturais através de imagens obtidas por *drones*. À medida que o *drone* vai captando as imagens, o modelo 3D é reprocessado e acrescenta à geometria nova os dados obtidos. Este processo é executado quase em tempo-real levando cerca de 90 segundos desde as aquisição de imagens até à reconstrução da geometria. A reconstrução do modelo é conseguido através de algoritmos de geolocalização juntamente com o processamento de imagens.

Cai *et al.*[47] apresenta uma ferramenta que permite completar dados em falta das nuvens de pontos com recurso a *Deep Learning*. A nuvem é dividida em várias áreas pequenas onde para cada uma delas são calculados os pontos em falta tendo em conta os dados locais (referentes a cada área) e os dados globais (referentes à área em relação ao resto da nuvem de pontos). Depois de calculados os dados, as áreas são fundidas gerando uma nova de nuvem de pontos mais completa que a original. O mesmo acontece com os dados de cor que são também calculados para estes novos pontos criados.

Na seguinte tabela 3.1 apresenta-se uma análise crítica dos artigos mencionados com os seus pontos positivos e negativos.

Tabela 3.1: Tabela análise crítica dos artigos.

Artigos	**Pontos Positivos**	**Pontos Negativos**
Sánchez *et al.*[81]	- Nuvem de pontos precisa e com dados de cor	- Aquisição e visualização da dados não ocorre em tempo real - É visualizado uma nuvem de pontos e não uma geometria - Não é permitida a interação com a dados apresentada
Serdel *et al.*[77]	- Aquisição e visualização dos dados ocorre em tempo quase real - Gera uma geometria detalhada muito semelhante ao ambiente recolhido através da biblioteca *Voxblox ROS* - Apresenta texturas realistas	- Só permite visualizar a geometria criada e não a sua interação
Liu *et al.*[71]	- Gera uma geometria básica do ambiente envolvente em tempo real - O resultado final apresenta texturas realistas e uma geometria mais detalhada com recurso à biblioteca *PCL*	- Passa por dois processos para gerar a geometria final - A geometria detalhada não é gerada em tempo real
Hu *et al.*[58]	- Aquisição e apresentação dos dados é feita em tempo real - Tem a capacidade de visualizar a nuvem de pontos num dispositivo móvel	- Não gera uma geometria - Só é possível visualizar a geometria e não interagir com ela
You *et al.*[87]	- Visualização em realidade virtual da geometria gerada - Aquisição e visualização dos dados ocorre em tempo quase real	- Só permite visualizar a geometria e não interagir com ela

Lee *et al.*[70]	- Permite a visualização da geometria em ambiente colaborativo - Capacidade de visualizar a geometria em Realidade Aumentada e Virtual	- A interação com a geometria não acontece com o ambiente gerado, mas sim com o objeto partilhado - O objeto partilhado não é gerado a partir da aquisição de nuvem de pontos a partir de um sensor
Heo *et al.*[57]	- Permite o envio de dados em tempo real	- Não obtem nuvem de pontos nem gera uma geometria a partir dos mesmo
Hughes *et al.*[59]	- Permite criar um cenário a três dimensões em tempo real - Diferentes níveis de detalhe	- Só permite visualizar o grafo e não interagir com ele - O cenário é uma representação interna para um robô poder ter uma perceção espacial e não uma interface
Soman *et al.*[78]	- Permite visualizar geometria em Realidade Aumentada - Possibilidade de visualizar os dados num dispositivo móvel	- Aquisição e visualização dos dados não ocorre em tempo real
Ali *et al.*[40]	- Permite a visualização e interação com a geometria gerada - A geometria gerada é efectuad com recurso a *alpha shapes* e *ball pivoting*	- Aquisição e visualização dos dados não ocorre em tempo real
Tagarakis *et al.*[82]	- Nuvem de pontos bastante densa e precisa dos objetos alvo	- Aquisição e visualização dos dados não ocorre em tempo real
Cheng *et al.*[48]	- Aquisição e visualização dos dados ocorre em tempo quase real (90 segundos) - Capacidade de adicionar nova informação à geometria existente - Usa o método *SLAM* para gerar a geometria	- Só permite visualizar a geometria e não interagir com ela

Cai *et al.*[47]	- Capacidade de recriar nova informação que não existia na geometria existente - Dados detalhados com informação de cor do ambiente envolvente	- Aquisição, processamento e visualização dos dados não ocorre em tempo real - Só permite visualizar a geometria e não interagir com ela

Os trabalhos referidos apresentam boas implementações na reconstrução da geometria, mas pecam na interface gráfica que não permite a visualização e a interação de forma concorrente ou que não conseguem adquirir, gerar e visualizar a geometria em tempo real. Assim propomos uma aplicação que permita visualizar, interagir e modificar os parâmetros de criação de uma geometria gerada através de uma nuvem de pontos proveniente de um simulador em tempo real, em que a receção dos pontos será imediata, mas o processamento desses mesmo pontos acontecerá periodicamente. Essa interação passa por modificar os parâmetros dos algoritmos que permitem gerar a geometria a partir da nuvem de pontos.

Capítulo 4

Desenvolvimento

4.1 Abordagem ao Problema

O objetivo deste trabalho é conseguir recriar o ambiente em que o robô se encontra através dos dados extraídos do sensor e visualizar esse mesmo ambiente em tempo real através do desenvolvimento de uma interface gráfica. Esta interface permitirá interagir com o modelo que estará a ser gerado, controlando alguns parâmetros que afetarão a forma como o ambiente é construído. Para isto propomos uma série de pontos que deverão ser implementados para que a aplicação cumpra os requisitos pretendidos:

1. **Ler nuvens de pontos** — A aplicação terá que ser capaz de ler e interpretar um formato universal de nuvem de pontos, armazenando esta informação numa estrutura de dados interna. O formato dos dados será 'xyz'.

2. **Recriar e visualizar o ambiente obtido através da nuvem de pontos** — Com a informação da nuvem de pontos armazenada, o próximo passo será recriar o ambiente e mostrá-lo ao utilizador através de uma interface.

3. **Interagir com o ambiente recriado** — A interface deverá permitir a interação do utilizador com o ambiente, modificando os parâmetros de geração do próprio ambiente assim como diferentes ângulos para a sua visualização.

4. **Permitir a troca de informação com o simulador/robô** — Os dados da nuvem de pontos deverão ser enviados em tempo real para a aplicação, permitindo a sua visualização aquando da sua recepção.

Estes requisitos representam as funcionalidades que a aplicação terá de cumprir. Em baixo explicamos a forma como desenvolvemos a aplicação com a arquitectura, as tecnologias utilizadas e a descrição da implementação do projecto.

4.1.1 Arquitectura

O projeto está organizado em três partes, *Gazebo*, *backend* e *frontend*. O *Gazebo* é o simulador onde é criado o mundo simulado que contém um ambiente e um veículo (neste caso o veículo é representado pelo sensor) que 'analisa' o mundo envolvente e exporta uma nuvem de pontos que depois vai ser consumida pela interface. O *backend* contém uma API que permite que simulador comunique com este. É no *backend* que o processamento da nuvem de pontos é efetuado. O *frontend* possui a interface com que o utilizar vai interagir. O *frontend* também comunica com o *backend* onde exibe o modelo 3D resultante do processamento efectuado. Na figura 4.1 é possível ver todos os componentes que compõem a aplicação e as suas interações.

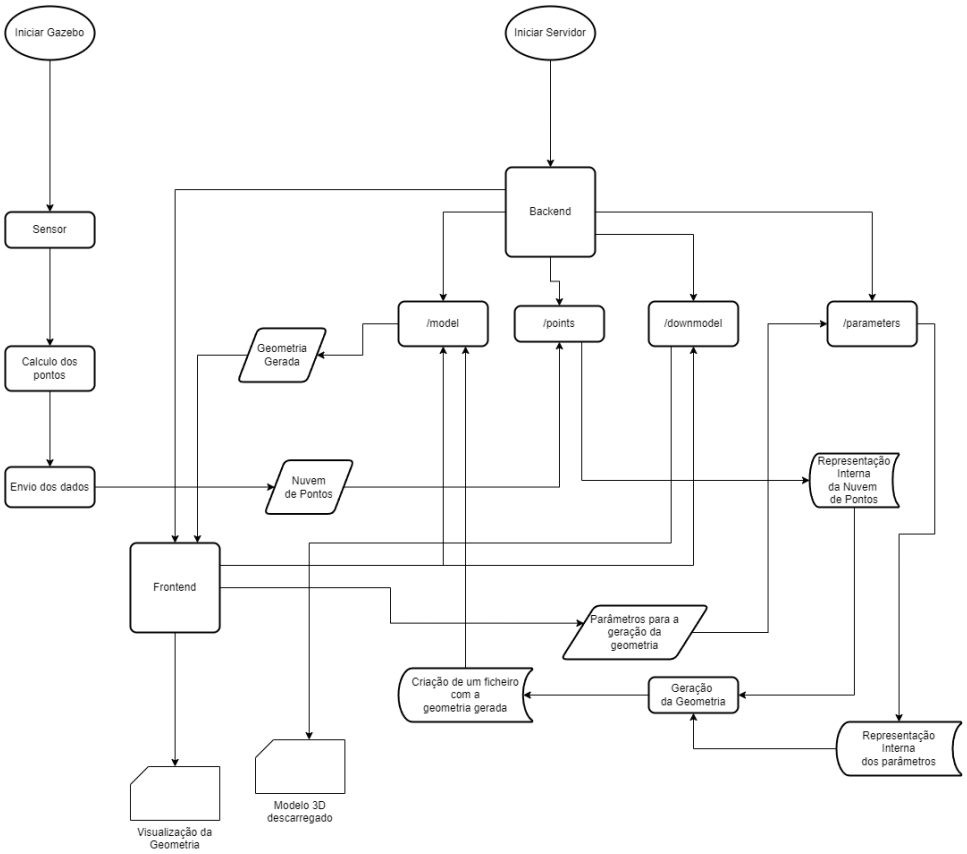

Figura 4.1: Fluxograma da Aplicação

4.2 Tecnologias utilizadas

4.2.1 Gazebo

Como referido anteriormente, o *Gazebo* [66] é um simulador robótico que tem a capacidade de criar objetos num espaço tridimensional e simular as interações entre eles. Neste projeto o *Gazebo* será usado para simular um robô com um sensor *(*LiDAR) num ambiente arbitrário.

4.2.2 Open3D

O *Open3D* é uma biblioteca que permite lidar com dados 3D [90]. Esta biblioteca possui um leque alargado de algoritmos e estruturas de dados tanto em *C++* como em *Python*. O código está altamente otimizado e possui um numero reduzido de dependências. Para além de permitir trabalhar sobre os dados tridimensionais, também é possível visualizar esses mesmos dados através de um visualizador integrado. É um projeto open-source que tem sido desenvolvido desde 2015. Devido à facilidade de instalação e de uso, a escolha recaiu nesta ferramenta que detém um papel central no projeto. Será usada para recriar geometria 3D a partir da nuvem de pontos extraída do simulador.

4.2.3 WebGL e Three.js

O WebGL [35] é uma API *Javascript* que permite renderizar conteúdo 2D e 3D dentro de um navegador compatível sem a necessidade de *plugins*. Ao usar uma API que está dentro da conformidade *OpenGL ES 2.0* e que pode ser usada dentro de o elemento *HTML <canvas>*, é possível usar o hardware gráfico da máquina do utilizador. Esta API é um sistema baixo nível e para evitar desenvolver todas as ferramentas que permitem a visualização 3D, torna-se necessário uma biblioteca que nos auxilie nesta tarefa.

A biblioteca *Three.js* [31] permite criar uma camada de abstração entre o programador e a API *WebGL*, assim torna-se mais acessível criar conteúdo 3D na Web. *Three.js* vai ficar encarregue de mostrar a geometria 3D gerada num navegador.

4.2.4 C++, Python e Javascript

Para o desenvolvimento do sensor e a extracção dos pontos do simulador é usado o *C++* [4], pois a API do *Gazebo* está implementada em *C++*. De forma a facilitar o trabalho foi decidido manter esta linguagem de programação.

O *website* é implementado com *Python* e *Javascript*. A escolha recaiu nestas duas tecnologias pois, *Javascript* é mandatório quando se fala de desenvolvimento web e possui uma panóplia de bibliotecas que ajudam a esse mesmo desenvolvimento, neste caso está encarregue do *frontend*. O *Python* está a ser usado como *backend*. O *Python* é bastante versátil no que toca às suas possibilidades de desenvolvimento, possuindo bastantes bibliotecas que facilitam a implementação. Para este projecto foi usada a biblioteca *Flask* [12]. Como o processamento da nuvem de pontos ocorre no *backend* do *website* as escolhas eram limitadas, sendo que o *Open3D* só permite desenvolvimento em *C++* ou *Python*. Devido à sua facilidade de uso e versatilidade, o *Python* foi escolhido.

4.3 Implementação

O desenvolvimento deste projeto possui três grandes fases: simulação no *Gazebo*, interface gráfica e a interconexão entre o simulador e a interface. Segue-se uma explicação detalhada de

cada uma das fases.

4.3.1 Ambiente de Simulação

Como já foi referido anteriormente, a plataforma de simulação escolhida é o *Gazebo* versão 11. Numa primeira fase foi necessário simular o sensor que permite obter a nuvem de pontos. A escolha recaiu para o modelo *LiDAR Velodyne HDL-32E* [33].

Um sensor *LiDAR (Light Detection and Ranging)* [30] mede distâncias através de emissão de luz. Um raio de luz é emitido pelo sensor e é reflectido num objecto, posteriormente retornando de novo ao sensor. O cálculo da distância é feito medindo o tempo que o raio de luz demora desde o momento em que foi emitido até ao momento que volta ao sensor. Para obter as coordenadas dos pontos adquiridos o sensor possui um GPS para a identificação das coordenadas cartesianas, um *IMU (Internal Measurement Unit)* para a orientação do sensor e a localização da energia do raio de luz.

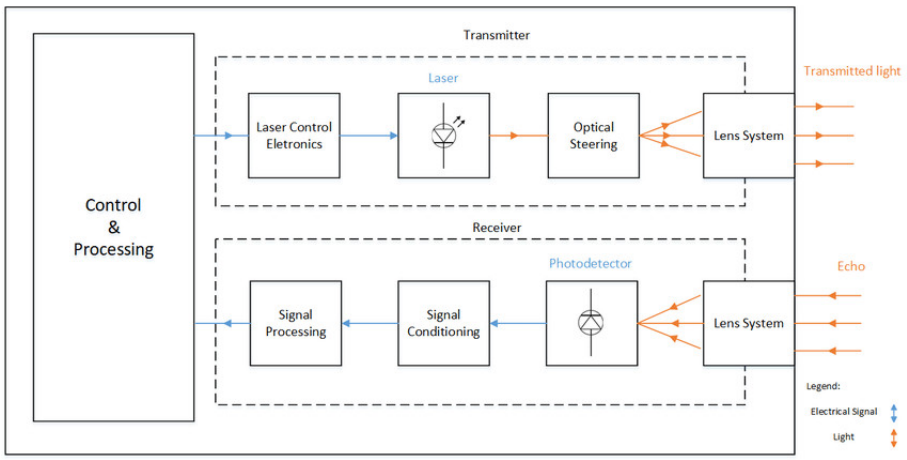

Figura 4.2: Diagrama básico de um sensor (LiDAR) [10]

O processo de simulação do sensor *LiDAR* pode ser dividido em duas partes, criação e simulação do comportamento do sensor e recolha e interpretação dos dados exportados pelo sensor.

A primeira parte foi baseada no guia de criação de um sensor *LiDAR* na página web oficial do *Gazebo* [15]. O sensor é criado a partir de um ficheiro *SQL Server Compact Database File (sdf)* [27] onde contém toda a informação referente ao modelo do sensor, contentores de colisão, informação de inércia (para o cálculo do comportamento do objecto quando forças são aplicadas), geometria do sensor, o sensor em si (emissor e receptor dos raios de luz) e uma junção que conecta as duas geometrias do sensor. O sensor é constituído por dois cilindros (um em cima do outro) em que existe uma junção a ligar o cilindro de cima ao do baixo. Esta junção, que só permite efetuar rotações no eixo Z, existe para permitir a rotação do cilindro de cima de forma a simular o comportamento real do sensor.

De forma a colocar o sensor a uma altura que permita ao sensor possuir um maior campo de visão, foi criado um 'veículo' muito simples que é constituído por uma caixa e o sensor. A figura 4.3 ilustra o modelo criado.

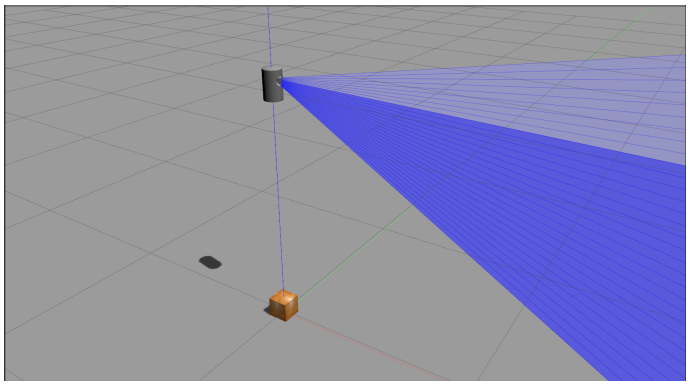

Figura 4.3: Veículo com o sensor

O sensor propriamente dito (parte que simula o comportamento da emissão e recepção dos raios de luz) possui vários parâmetros. Como no nosso caso estamos a simular o sensor *Velodyne HDL-32E* [33], estas são as opções que usamos para aproximar o comportamento do sensor simulado ao real.

Posição e orientação, o emissor é colocado na posição que mais se aproxima da posição real e é orientado a 90 graus sobre o eixo X para que os raios horizontais se tornem verticais (por defeito os raios são horizontais).

Taxa de actualização é o número de vezes que atualiza por segundo, no nosso caso são 10*Hz*.

Número de raios de luz refere-se ao número de raios que o sensor possui, sendo que o nosso possui 32.

Ângulo mínimo e ângulo máximo, refere-se ao ângulo de visão do sensor que está compreendido entre um ângulo máximo (+10.67 graus) e um ângulo mínimo (-30.67 graus).

Alcance mínimo e o alcance máximo, indicam o intervalo de distância que os raios conseguem obter informação, neste caso o sensor possui um alcance mínimo de 5cm e um alcance máximo de 100m.

Ruído é usado para simular as imperfeições do sensor real. Um sensor real não consegue obter os dados todos de uma forma perfeita, vai haver sempre interferências que podem gerar dados com erros. No nosso caso usamos o ruído *gaussiano* com um desvio padrão de 0.02.

Para a simulação da rotação do sensor precisamos de criar um *plugin* que vai controlar a velocidade de rotação da *joint* que une as duas partes do modelo do sensor. Este *plugin* é escrito em *C++* e faz uso das bibliotecas do *Gazebo*.

Para modificar a velocidade é necessário criar um controlador e associá-lo à *joint*. A atribuição da velocidade é feita usando a propriedade *SetVelocityTarget* presente no controlador da *joint*. Definimos também um outro ficheiro *sdf* para criar um mundo que contém o nosso sensor e um ambiente onde pode interagir. Neste ficheiro associamos ao modelo do sensor o nosso *plugin* e

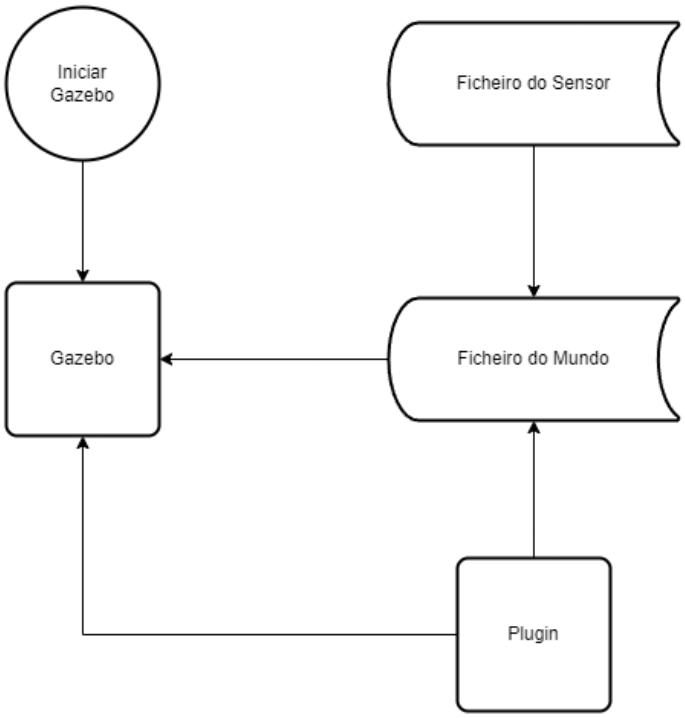

Figura 4.4: Diagrama da simulação do sensor

criamos também um campo <*velocity*> onde podemos colocar a velocidade de rotação do sensor. Assim podemos ler o valor deste campo e atribuí-lo ao controlador da *joint* no *plugin*. Desta forma podemos iniciar o *Gazebo* com um mundo, contendo o nosso sensor e um ambiente que permite com que o sensor interaja com este. A compilação deste *plugin* necessita de ser uma compilação onde é criado um ficheiro de biblioteca partilhada para que o *Gazebo* use as ferramentas desenvolvidas.

Com a simulação do comportamento do sensor finalizada é necessário processar os dados que são obtidos através do sensor. A imagem 4.5 descreve a estrutura de dados provenientes da exportação do sensor. A lista só contém os dados usados no projecto.

Os dados exportados estão contidos na interface ***LaserScan*** [19]. Aqui dentro encontra-se o ***angle_min***, que define o ângulo mínimo dos raios laser, ***angle_step***, representa a diferença de ângulo entre cada um dos raios laser, ***ranges***, contém a distância que cada raio registou e ***world_pose***, que possui a posição e a orientação do sensor.

O sensor contém neste momento 32 raios alinhados num plano vertical (plano XZ), compreendidos entre um ângulo máximo e mínimo, em que estão espaçados de igual forma por um valor fixo (*angle_step* figura 4.6). Este valor é calculado pela diferença entre os ângulos máximo e mínimo a dividir pelo número total de raios, vindo já devidamente calculado nos dados exportados pelo sensor.

O processo de cálculo da nuvem de pontos passa por extrair o ângulo de rotação do sensor no eixo Z. Este processo é feito convertendo a informação de rotação do sensor, que é exportada,

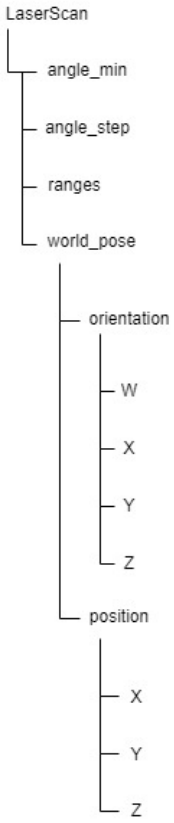

Figura 4.5: Estrutura de dados que o sensor exporta

para a classe *Quaternion* [21] que permite efectuar operações com quarterniões. De seguida isola-se somente o eixo de rotação que pretendemos com o método *Yaw*, neste caso o eixo Z. Depois volta-se a converter esta rotação para um quarternião.

Para calcular cada ponto onde os raios colidem com o ambiente, são usadas as fórmulas de conversão de coordenadas esféricas para coordenadas cartesianas [28]. Para cada um dos raios é calculado as coordenadas x, y e z do ponto de contacto com recurso ao *angle_step* acumulado (representado por α) e ao *range* (representado por r) de cada raio. O eixo y é inicializado a zero para todos os raios. Estes valores são guardados na classe *Vector3* [32].

$$x = r * cos(\alpha) \tag{4.1}$$

$$z = r * sin(\alpha) \tag{4.2}$$

Com este cálculo efectuado, é necessário converter cada um dos pontos para a rotação do sensor. Para isso usa-se o método *RotateVector* que recebe como argumento um vector, que neste caso será o ponto recentemente calculado. Por fim soma-se a posição do sensor ao ponto calculado depois da rotação e obtém-se o valor final do ponto.

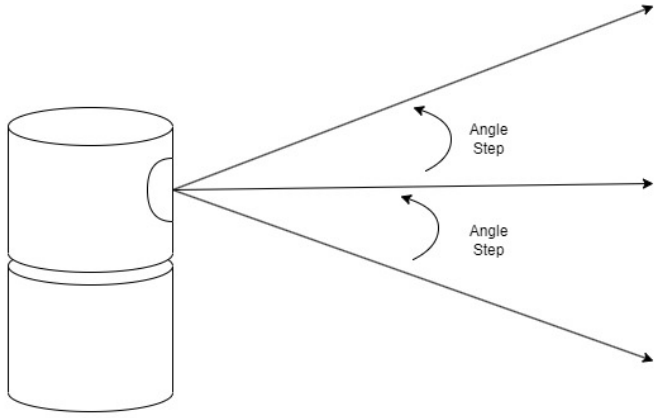

Figura 4.6: Diagrama do Sensor Simulado

Este processo é executado pela função *calculate_points* que tem como único argumento uma variável do tipo *gazebo::msgs::LaserScan*, que contém os dados todos exportados pelo sensor. A função é executada com a mesma frequência do sensor, ou seja, 10 vezes por segundo.

Este pontos são enviados em formato *JSON* [16] via *POST* para o *endpoint /points* da interface gráfica através da função *sendPoints*. Esta função tem como único argumento *std::string data* que são os pontos em formato *JSON*. Para poupar recursos e tornar a aplicação mais eficiente, esta função é executada cerca de 2 vezes por segundo.

```
[
    {
        "x": x_value,
        "y": y_value,
        "z": z_value
    },
    {
        "x": x_value2,
        "y": y_value2,
        "z": z_value2
    },
    ...
]
```

Código 4.1: Formato dos pontos em JSON

4.3.2 Interface Gráfica

A interface possui uma arquitetura clássica de um *website*, com um *backend* e um *frontend*. Existe também uma API que permite receber a nuvem de pontos do simulador (figura 4.1).

Como referido anteriormente, foi escolhida a biblioteca *Open3D* [90] para a reconstrução 3D da nuvem de pontos. O *Open3D* permite efetuar operações na nuvem de forma a que seja gerada uma mesh que depois pode ser exportada para um formato de ficheiro de fácil visualização. Para essa visualização foi usado *WebGl* através da framework *javascript*, *Three.js* [31].

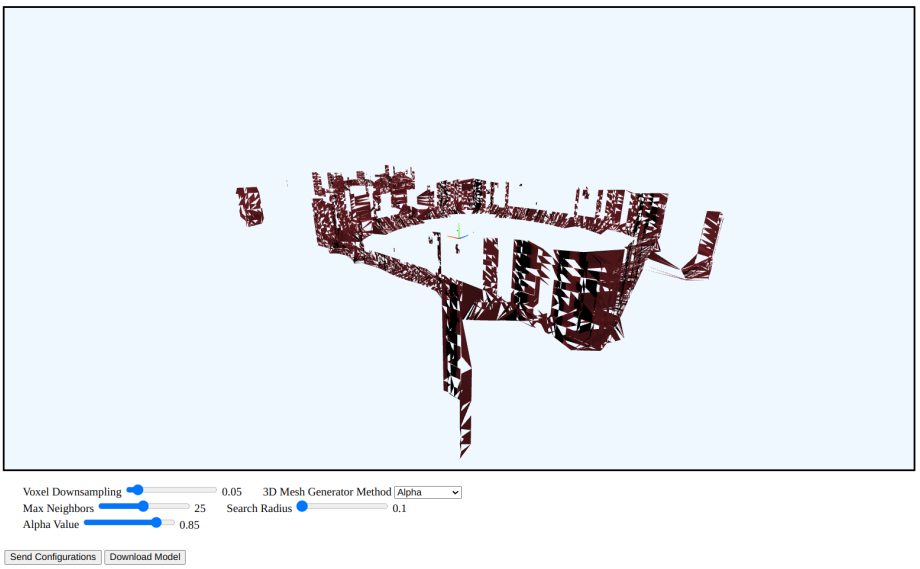

Figura 4.7: Interface Gráfica

Os componentes da interface estão divididos em duas partes simples: a tela de visualização, onde se pode observar e rodar a geometria 3D e o painel de controle dos parâmetros, que permite modificar os parâmetros da geração da geometria. Sempre que o utilizador faça uma alteração aos parâmetros, para que estes tenham efeito, tem que carregar no botão *Send Configurations*. O utilizador tem ainda a possibilidade de descarregar o modelo 3D, no formato *obj*, carregando no botão *Download Model*. O nome do ficheiro que contém o modelo 3D varia consoante o método utilizado para a criação da geometria. Segue-se uma breve explicação do efeito de cada parâmetro.

Tabela 4.1: Tabela descritiva dos parâmetros.

Voxel downsampling	Valor que responsável por indicar a redução do número de pontos. Quanto menor o valor maior a redução. Este parâmetro está compreendido entre *0.01* e *0.5*.
3D mesh generator method	Possui uma lista dos 3 métodos para gerar uma superfície 3D, *ball pivoting*, *alpha shapes* e *poisson surface reconstruction*. Cada método possui parâmetros que são específicos a cada um, sendo que estes tornam-se disponíveis ao utilizador assim que este seleciona o método pretendido.

Max neighbors e ***Search radius***	Definem os parâmetros necessários para a árvore de pesquisa que é usada para o cálculo dos vectores normais nos pontos. Os valores estão compreendido entre *1* e *50* para *Max neighbors* entre *0.1* e *1*.
Min radius	Define o raio mínimo da esfera para o método *ball pivoting*. Os valores estão compreendidos entre *0.01* e *0.5*.
Max radius	Define o raio máximo da esfera para o método *ball pivoting*. Os valores estão compreendidos entre *0.5* e *1*.
Alpha value	Define o valor *alpha* para o método *alpha shapes*. Este está compreendido entre *0.01* e *1*.
Poisson depth	Define o valor do detalhe da superfície para o método *poisson surface reconstruction*. Está compreendido entre *2* e *10*.
Poisson scale	Define a escala da geometria para o método *poisson surface reconstruction*. Está compreendido entre *0.1* e *1.1*.

O *frontend* faz uso da biblioteca *javascript*,*Three.js*. Esta biblioteca tem como estrutura base uma *scene* [8]. Nesta *scene* são adicionados objectos que podem ser, entre outros, modelos 3D, câmaras, luzes, controles, etc. Para poder-se ver o resultado desta *scene* existe um outro objecto, o *renderer*, que faz a renderização de todos os objectos associados à *scene* apresentando o resultado no *browser*.

Neste caso, temos uma *scene* base que possui uma câmara, os controlos que a permitem controlar, uma luz direccional e uma ajuda visual que mostra os 3 eixos do espaço 3D. A interface em si possui 3 partes que implementam todas as suas funcionalidades. Carregamento e visualização da geometria, actualização da geometria, envio dos parâmetros e actualização dos mesmos quando o utilizador os insere.

O carregamento da geometria está implementado pela função *objLoader()* [13] que faz uso do objecto *GLTFLoader()* presente na biblioteca *Three.js*. Este objecto possui um método que permite fazer o carregamento de um ficheiro do tipo *gltf*. Desta forma pode-se pedir ao *backend* o ficheiro que gerou e adicioná-lo como um modelo 3D à *scene*.

De forma a poder-se actualizar a visualização, definiu-se uma função que será chamada continuamente. Esta função chama-se *animate()* [2] e dentro dela definiram-se as funções que queremos 'animar', permitindo que toda a informação seja actualizada. Para que esta função seja chamada continuamente precisa-se de chamar o método *requestAnimationFrame()* e passar-lhe como campo a função *animate()*.

Cada um dos campos dos parâmetros tem associado o evento *input* [14] que é chamado sempre que existe uma alteração aos valores desses campos. O evento tem uma função associado que é actualiza os valores internos de cada parâmetro. Esta representação interna dos parâmetros é depois enviada para o *backend* como um pedido *POST* pela função *sendParametersData()*. Estes dados são enviados em formato *JSON*.

```
{
    "method": methodType,
    "voxelDownsampling": voxelDownsampling,
    "pointNeighbors": pointNeighbors,
    "pointRadius": pointRadius,
    "poissonDepth": poissonDepth,
    "poissonScale": poissonScale,
    "alpha": alpha,
    "minRadius": minRadius,
    "maxRadius": maxRadius
}
```

Código 4.2: Formato dos parâmetros em JSON

O *backend* foi desenvolvido em *Python* e contém todo o processamento da nuvem de pontos com recurso à biblioteca *Open3D*. O *frontend* possui a interface propriamente dita, onde é possível visualizar e interagir com o modelo 3D. Esta interação é conseguida através da framework *Three.js* que permite fazer a visualização de objetos 3D no browser, com recurso a *WebGL*.

Como referido anteriormente, o *backend* foi desenvolvido em *Python* com recurso à biblioteca *Flask* [12]. Esta biblioteca permite definir vários *routes* [26] que permite retornar ficheiros quando um *browser* acede a uma das *routes*. Também se pode adaptar essas mesmas *routes* como *endpoints* de uma API, podendo definir os métodos HTTP.

O *backend* encontra-se à escuta na porta 8080 e possui cinco *routes*, *root*, *model*, *points*, *parameters* e *downmodel*. Foi implementada também a função *pointCloudMesh()* que processa a nuvem de pontos e gera a geometria 3D. Segue-se uma breve explicação de cada componente do *backend*.

A ***root* ('/')** é a *home page* onde mora a interface gráfica. Retorna o ficheiro *html* da interface.

O ***model* ('/model')** é usado para retornar um ficheiro *gltf* que contém o modelo 3D gerado a partir da nuvem de pontos. Este ficheiro é enviado ao *frontend* para ser visualizado na interface.

O ***points* ('/points')** é responsável por receber e processar os dados dos pontos provenientes do simulador. Recebe este dados em formato *JSON*, como indicado em 4.1. Os dados são extraídos do *JSON* e são guardados em variáveis globais, de modo a que estejam acessíveis à função responsável pela geração da geometria, *pointCloudMesh()*. Esta função é executada todas as vezes que recebe um pedido.

O ***parameters* ('/parameters')** recebe a informação que permite modificar os parâmetros da geração da geometria 3D. Este também recebe os dados em formato *JSON*, como indicado em

4.2. À semelhança do que acontece nos pontos, aqui os parâmetros também são guardados em variáveis globais, mas com a diferença de que a função *pointCloudMesh()* é executada sempre que se recebe novos parâmetros. Neste caso é importante que sempre que os parâmetros mudem, a geometria reflita essa mesma mudança.

O ***downmodel*** **('/downmodel')** retorna um ficheiro *obj* que contém o modelo 3D gerado a partir da nuvem de pontos. Este ficheiro é uma cópia do ficheiro *gltf* e o nome varia consoante o método de geração de geometria escolhido. O nome ***geometry-alpha.obj*** é usado para quando o método *alpha shapes* é o escolhido, ***geometry-ball.obj*** para o método *ball pivoting* e ***geometry-poisson.obj*** para o método *poisson surface reconstruction*.

A função ***pointCloudMesh()***, como referido anteriormente, é responsável pela criação da geometria a partir da nuvem de pontos criada a partir dos pontos enviados pelo simulador. Para este processo foi usado a biblioteca *Open3D* que possui vários algoritmos de geração de geometria a partir de uma nuvem de pontos. A figura 4.8 apresenta o fluxograma dos passos que a função possui.

Numa primeira fase a nuvem de pontos passa por um processo de 'limpeza' e preparação constituído por 4 processos, retirar pontos iguais, eliminar pontos extremos, reduzir a densidade da nuvem de pontos e cálculo dos vectores normais. O método *Open3D remove_duplicated_points()* [22] está responsável por retirar os pontos iguais. O método *remove_statistical_outlier()* [23] elimina os pontos que em média estão mais longe que os seus vizinhos. Esta função recebe dois argumentos, *nb_neighbors* e *std_ratio*, que são os número de vizinhos e o desvio padrão, respectivamente. Estes argumentos estão inicializados com os valores 30 para *nb_neighbors* e 1 para *std_ratio*. A redução da densidade da nuvem de pontos é obtida através da função *voxel_down_sample(voxel_size)* [34] que recebe como argumento o valor *Voxel downsampling* proveniente do utilizador.

A fase seguinte é o cálculo dos vectores normais dos pontos. Este cálculo é efectuado pela função *estimate_normals(search_param)* [11] que tem como argumento um parâmetro de árvore de pesquisa. Esta árvore é obtida pela classe *Open3D KDTreeSearchParamHybrid(radius, max_nn)* [18]. Esta classe contém como argumentos de criação o raio de pesquisa e o número de vizinhos. A figura 4.9 ilustra os processos que a nuvem de pontos passa.

Para a reconstrução da geometria, *Open3D* implementa 3 algoritmos [29], *alpha shapes*, *ball pivoting* e *Poisson surface reconstruction*.

Alpha shapes [51] é uma generalização de um conjunto de pontos de uma *convex hull*. Seja S um conjunto finito de $\Re^3$ e α um número real entre $0 \leq \alpha \leq \infty$. A *u-shape* de S é um polítopo que não é necessariamente convexo nem está necessariamente conectado. Para $\alpha = \infty$, a *alpha shape* aproxima-se do *convex hull* de S. Contudo, no momento em que α diminui, a *o-shape* encolhe criando cavidades. Estas cavidades podem-se conectar para formar túneis e até buracos podem aparecer. Quando α se torna pequeno o suficiente para que uma esfera de raio α possa ocupar um espaço sem conter nenhum dos pontos de S, um pedaço do polítopo desaparece. Isto pode ser interpretado da seguinte forma [17]: imagine-se uma nuvem de pontos representada por pepitas de chocolate numa bola de gelado e na esfera com raio α como uma colher de gelados,

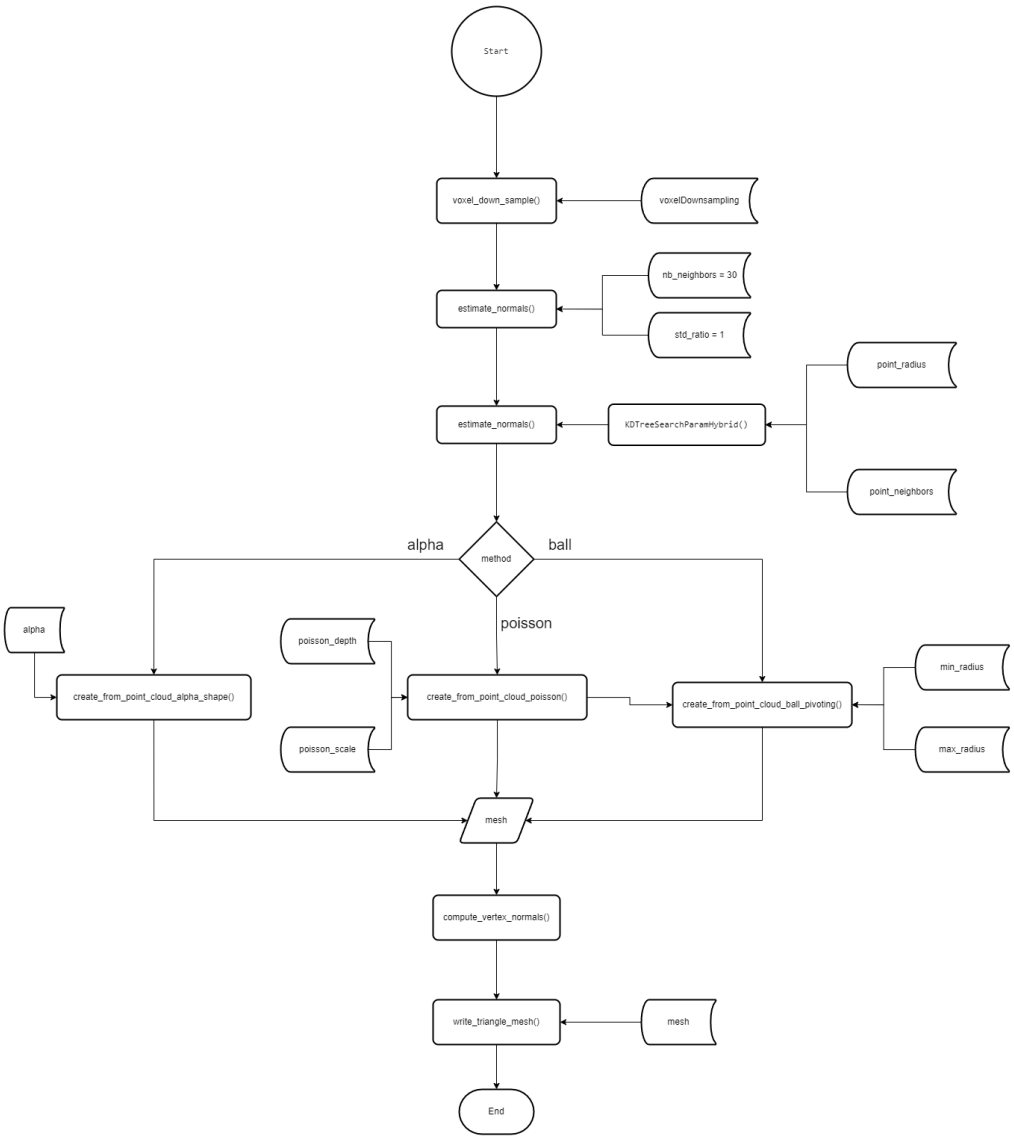

Figura 4.8: Diagrama da Função *pointCloudMesh()*

consegue-se imaginar esta a retirar material entre os pontos, deixando uma série de curvas que ligam esses mesmos pontos. Convertendo essas curvas em rectas, pode-se criar uma *mesh* e gerar uma geometria.

Existem dois conceitos que são importantes para este algoritmo, pois são passos que fazem parte integrante dele, triangulação *Delaunay* e diagrama de *Voronoi*. O diagrama de *Voronoi* [37] tem como definição o seguinte, dado um conjunto S com n pontos distintos em R^d, o diagrama de *Voronoi* é uma partição de R^d em n regiões de poliedros definidas como $vo(p)$. Cada região $vo(p)$ (célula *Voronoi* de p) são definidos como um conjunto de pontos R^d que estão mais perto de p do que quaisquer outros pontos de S. A triangulação *Delaunay* [36] tem a seguinte definição: dado um conjunto S com n pontos em R^d, a *convex hull* $conv(nb(S,v))$ do conjunto vizinho mais

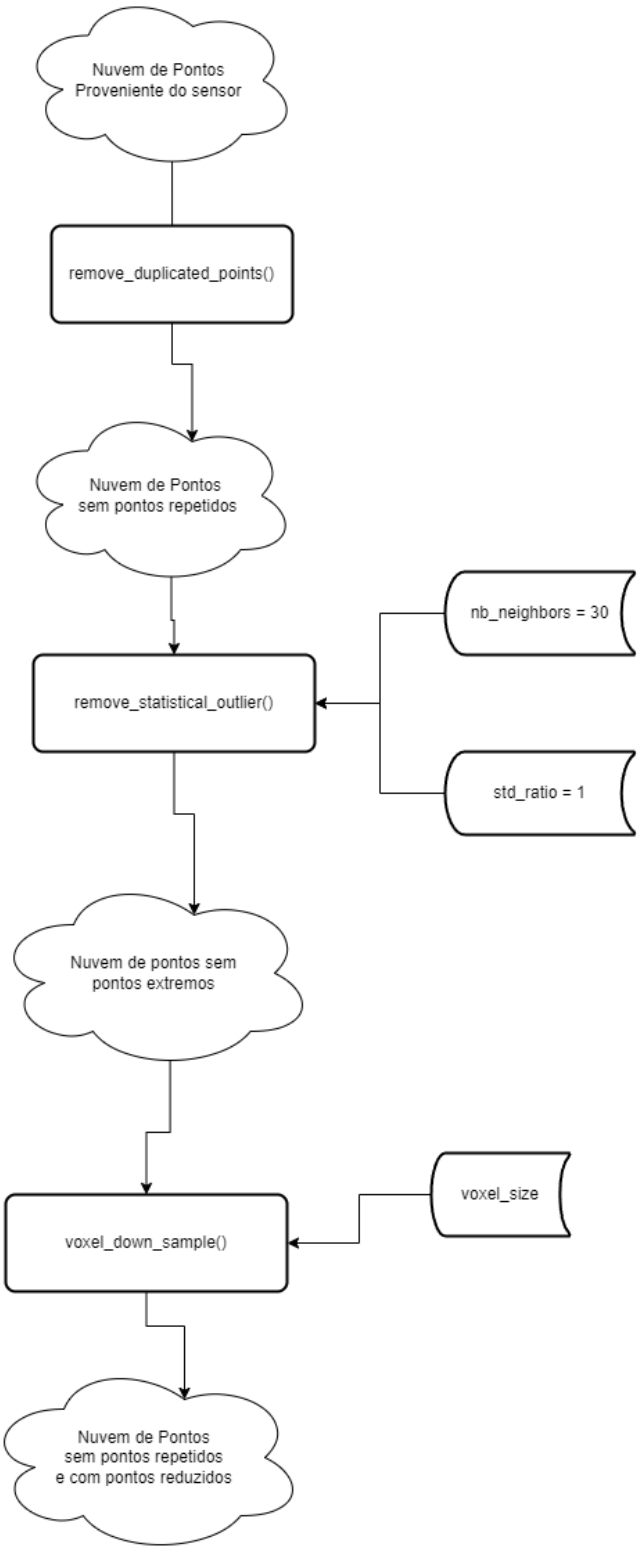

Figura 4.9: Tratamento da nuvem de pontos

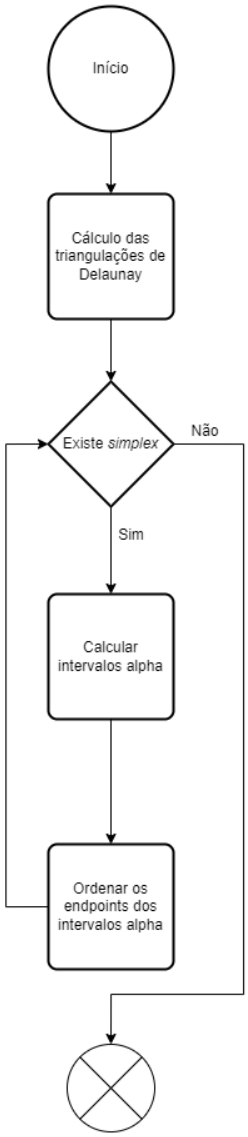

Figura 4.10: Algoritmo *Alpha shapes*

próximo de um vértice *Voronoi v* chama-se uma célula *Delaunay* de *v*. A triangulação *Delaunay* de *S* é a partição do *convex hull* para as células *Delaunay* dos vértices de *Voronoi* juntamente com as suas faces.

A execução da triangulação *Delaunay* resulta num *simplex*, que são basicamente faces. Uma destas faces pode pertencer à *alpha shape* caso esteja dentro de um único intervalo se e só se o valor α esteja também dentro deste intervalo. Estes intervalos são calculados com base na classificação dos *simplex* obtidos, *interior, singular e regular*. Esta verificação das faces é uma ordenação desses mesmos intervalos calculados. A figura 4.10 ilustra de forma abstracta o algoritmo.

Este algoritmo está disponível através da função *create_from_point_cloud_alpha_shape()* [5] que recebe como argumentos a nuvem de pontos e o valor α.

Ball pivoting [44] está relacionado com o algoritmo *alpha shapes*. Seja M uma superfície de um objecto tridimensional e S um conjunto de pontos de M. Assume-se que os pontos têm densidade suficiente para que uma bola com raio p não possa atravessar S sem colidir com nenhum ponto. Numa fase inicial a bola é colocada em contacto com três pontos. Mantendo o contacto com dois dos pontos iniciais, a bola roda sobre um eixo até tocar noutro ponto. A bola vai percorrendo as arestas da superfície e a cada três pontos em que a bola entre em contacto, um novo triângulo é formado. O conjunto de triângulos que se vão formando à medida que a bola percorre a superfície, cria a geometria.

Cada triângulo T criado por uma bola de raio p possui uma bola com raio menor que p. O algoritmo *Ball pivoting* processa então um subconjunto *2-faces* do conjunto de pontos S. Estas faces são, também elas, *2-skeleton* da triangulação *Delaunay* tridimensional do conjunto de pontos, daí a relação com *alpha shapes*.

Este algoritmo usa *advancing-front paradigm* [1] (estes tipo de métodos são usados para criar triangulações num determinado domínio em duas dimensões ou três dimensões) para construir triangulações interpoladas de forma incremental. Recebe como *input* uma lista de pontos da superfície, cada uma com informação de vectores normais, e um raio p. O algoritmo encontra uma semente (três pontos) em que a bola com raio p não entra em contacto com mais nenhum ponto e adiciona um triângulo de cada vez executando a operação de pivotagem (rodando a bola sobre um eixo). Chama-se $front\ F$ a uma colecção de listas ligadas de arestas, sendo a primeira definida por um único ciclo contendo as três arestas do primeiro triângulo criado pela primeira semente. Cada aresta é representada pelos seus dois pontos, um vértice oposto, o centro da bola com raio p que toca nos três pontos e apontadores para a aresta anterior e a aresta seguinte. Uma aresta pode estar num deste três estados, *active, boundary ou frozen*. *Active* é a aresta que é usada para executar a operação de pivotagem. Não é possível executar esta operação se as arestas estiverem no estado *boundary*. *Frozen* designa uma 'zona' onde a a operação de pivotagem ocorre, sendo que as arestas fora desta zona ficam no estado *frozen*. À medida que a bola 'percorre' o conjunto de pontos, existem duas operações que adicionam triângulos, modificando as arestas: *Join* adiciona as arestas e *glue* retira as arestas que coincidem. A figura 4.11 ilustra de forma abstracta o algoritmo.

A função *create_from_point_cloud_ball_pivoting()* [6] está responsável por executar o algoritmo de *ball pivoting* e tem como argumentos a nuvem de pontos e um vector com os raios das esferas.

Poisson surface reconstruction [63] permite criar uma geometria mais suave. Seja S um conjunto de amostras $s \in S$, em que $s.p$ são os pontos e $s.\vec{N}$ os vectores normais apontando para dentro, assumindo que estes pontos encontram-se na superfície ∂M de um modelo M desconhecido. De forma a construir a geometria é necessário aproximar uma função indicador em relação à superfície do modelo e extrair a *isosurface* (curva de nível para o espaço tridimensional). Para isto é derivado um relacionamento entre o gradiente da função indicador e o integral do campo normal de superfície. Existe uma aproximação do integral de superfície através de uma soma sobre os respectivos pontos orientados. Por fim existe uma reconstrução da função indicador a partir do campo de gradiente como um problema de *poisson*. A figura 4.12 ilustra de forma abstracta o

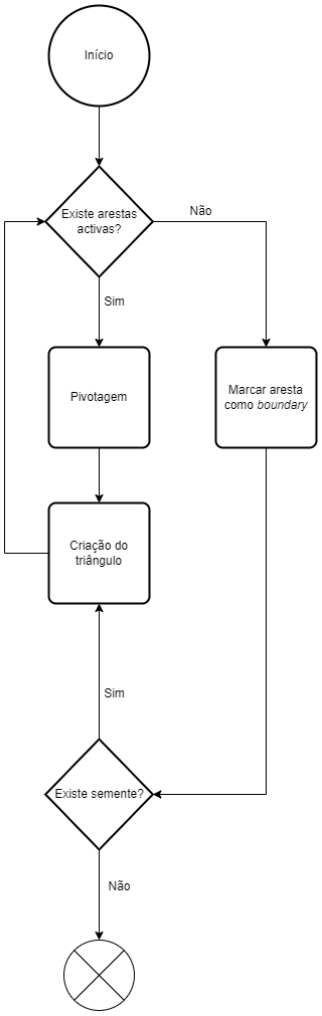

Figura 4.11: Algoritmo *Ball pivoting*

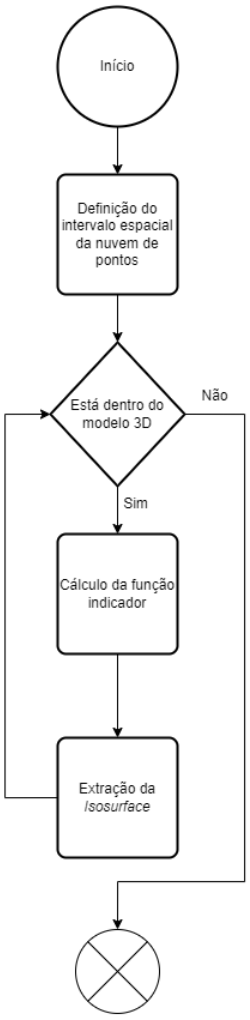

Figura 4.12: Algoritmo *Poisson surface reconstruction*

algoritmo.

A execução deste algoritmo está ao encargo da função *create_from_point_cloud_poisson()* [7] e recebe como argumentos a nuvem de pontos e valor *depth* que indica o nível de detalhe da geometria, quanto maior o valor maior o detalhe.

Para concluir este processo de geração da geometria, são calculados, através da função *compute_vertex_normals()* [3], os vectores normais das faces para as tornar vísiveis no visualizador.

4.3.3 Interconexão entre Simulador e Interface gráfica

O envio dos pontos do simulador para a interface está encarregue da função *sendPoints()*, que tem como argumento uma *string* dos pontos. Estes pontos estão em formato *JSON*, como indicado em 4.1. Para o envio da informação é usado a ferramenta *curl* [9], que já vem nativa com o *C++*. Os pontos são enviados com uma taxa de actualização diferente do sensor, cerca de $2Hz$. É feito

um pedido *POST* ao *endpoint* ***/points*** do *backend* da nossa interface. Cada envio contém 160 pontos.

Do lado da interface é criada uma variável do tipo *PointCloud* através do método *PointCloud()* [20], presente na biblioteca *Open3D*. Esta variável é iniciada sem pontos e sempre que são recebidos novos pontos, estes são adicionados a esta variável. Sempre que um pedidos é recebido, os algoritmos de geração de geometria são executados.

Capítulo 5

Resultados

5.1 Cenários de teste

Os resultados foram obtidos através de 4 cenários de teste criados no *Gazebo*. Estes cenários possuem modelos 3D diferentes que permitem verificar o nível de fidelidade com que são depois reproduzidos a partir da interface, modificando os parâmetros presentes na interface de forma a obter uma geometria que seja semelhante aos modelos dos cenários de teste. Os 4 cenários são: (1) um café (figura 5.1), (2) uma bomba de combustível (figura 5.2), (3) uma escola (figura 5.3) e (4) uma garagem com várias salas (figura 5.4). Aliados aos resultados da interface, adicionou-se também uma imagem da nuvem de pontos correspondente a cada cenário, enviada pelo simulador, de forma a perceber melhor os resultados obtidos. A visualização desta nuvem de pontos foi conseguida através do visualizador integrado do *Open3D*.

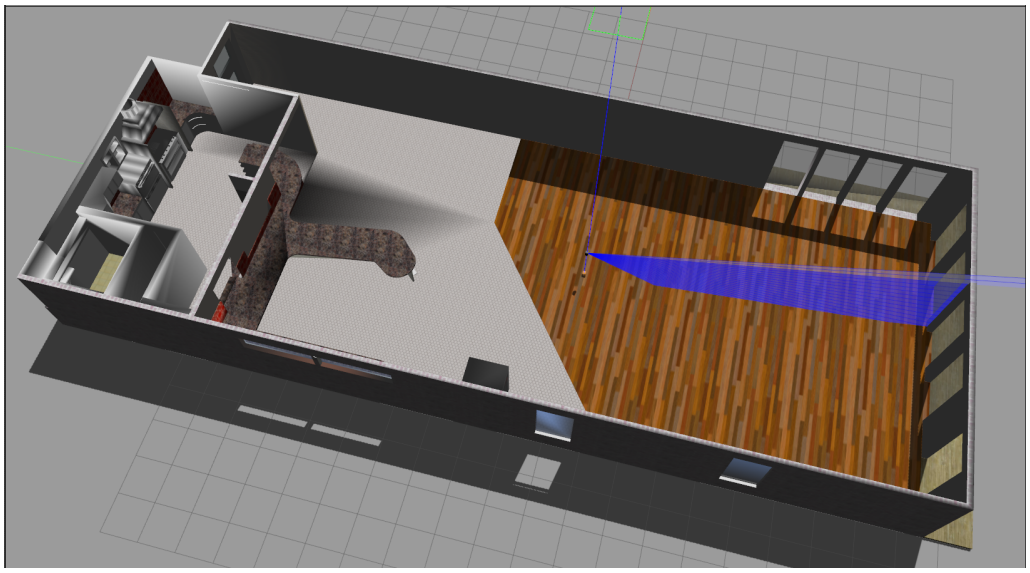

Figura 5.1: Café

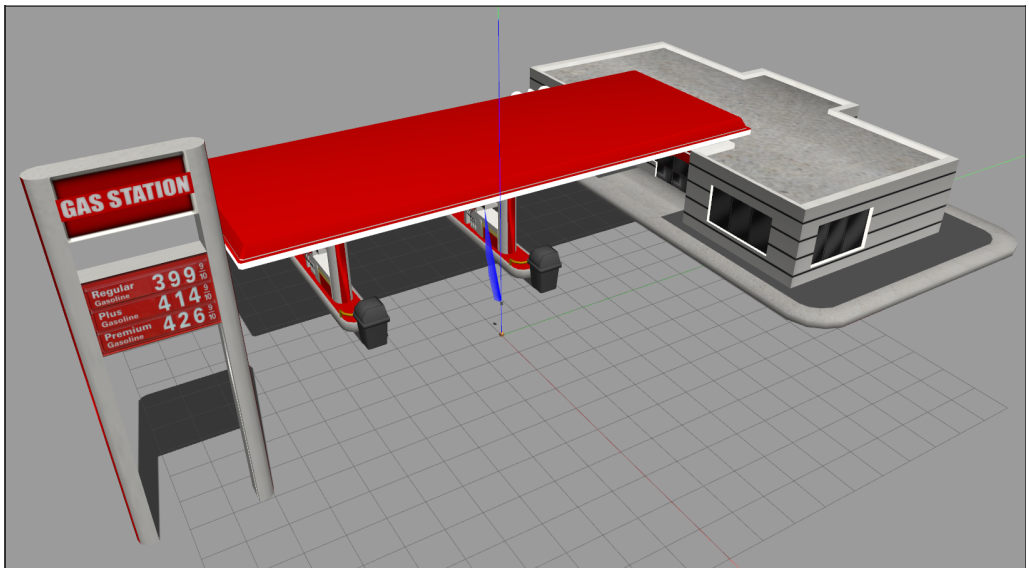

Figura 5.2: Bomba de combustível

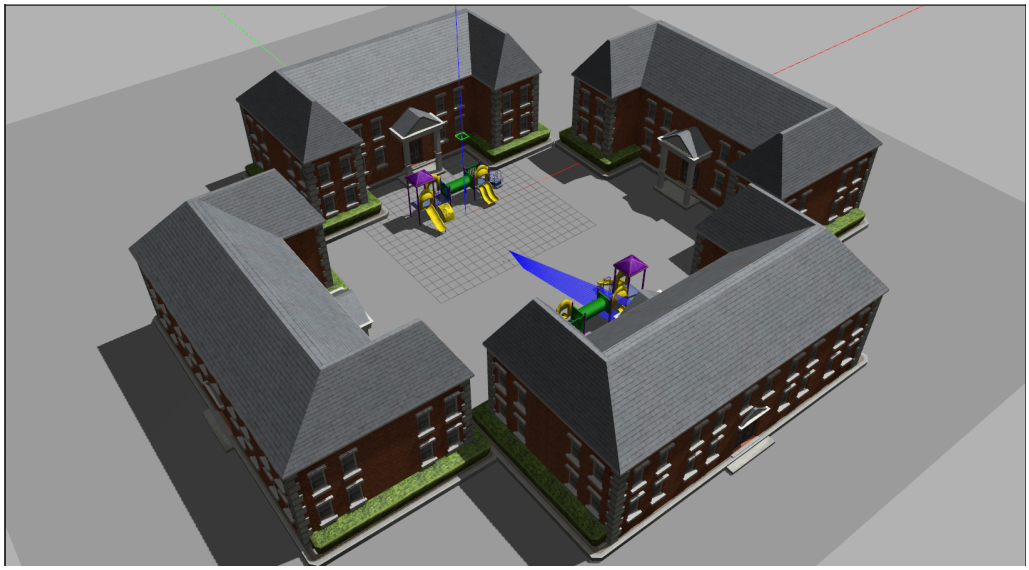

Figura 5.3: Escola

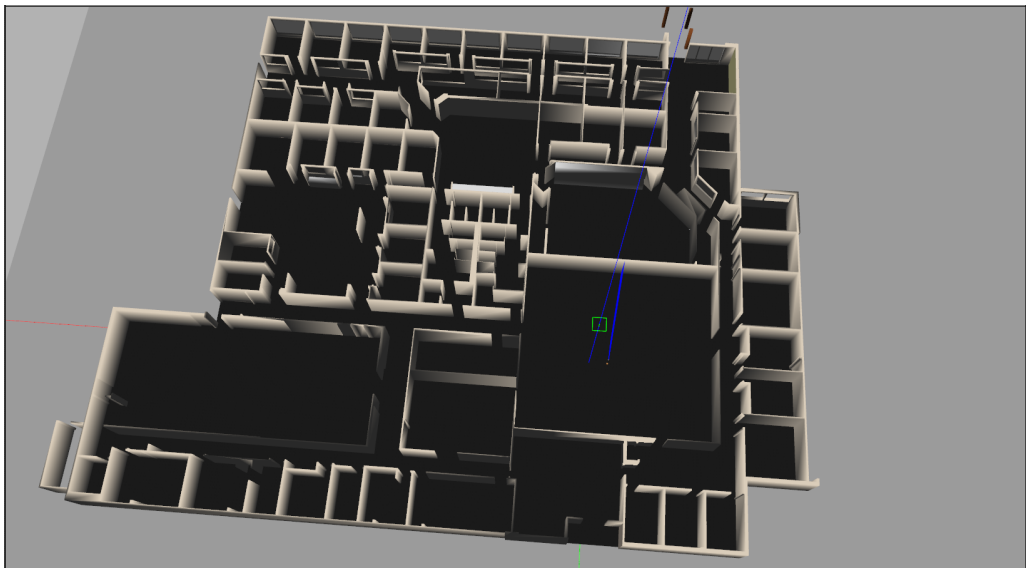

Figura 5.4: Garagem

Os resultados foram obtidos num computador portátil com um processador *Intel Core i7-6500U 2.50GHz*, 8GB de *RAM* e uma placa gráfica *NVIDIA GeForce 920M*, com o sistema operativo *Linux Mint 19.3 Cinnamon*.

5.2 Resultados dos testes

Como referido no capítulo 4, a interface possui a zona de visualização do modelo gerado e a zona de alteração dos parâmetros dos diferentes algoritmos de geração de geometria. O cenário com o café mostra como é gerado uma geometria simples que rodeia o sensor na totalidade. A figura 5.6 mostra os resultados obtidos para este cenário, gerado com o algoritmo *ball pivoting*. A figura 5.7 mostra o resultado para o algoritmo *alpha shapes* e a figura 5.8 mostra para o algoritmo *poisson surface reconstruction*. A figura 5.5 mostra a nuvem de pontos obtida pelo simulador para o cenário do café.

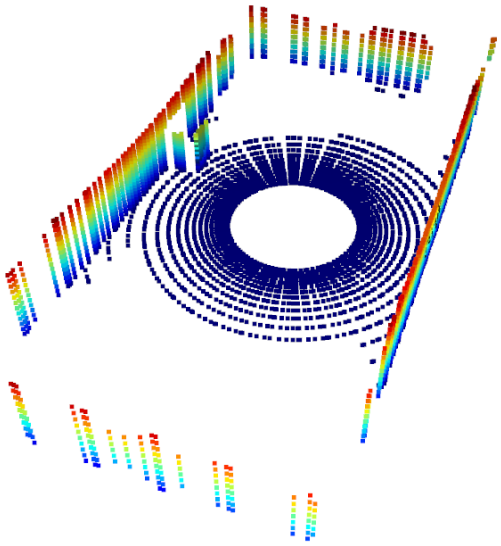

Figura 5.5: Nuvem de pontos do cenário café

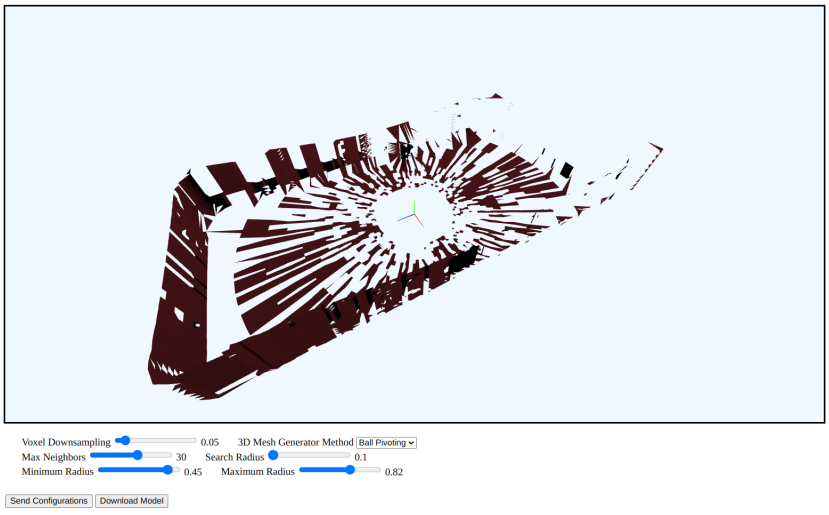

Figura 5.6: *Ball pivoting* para o cenário café

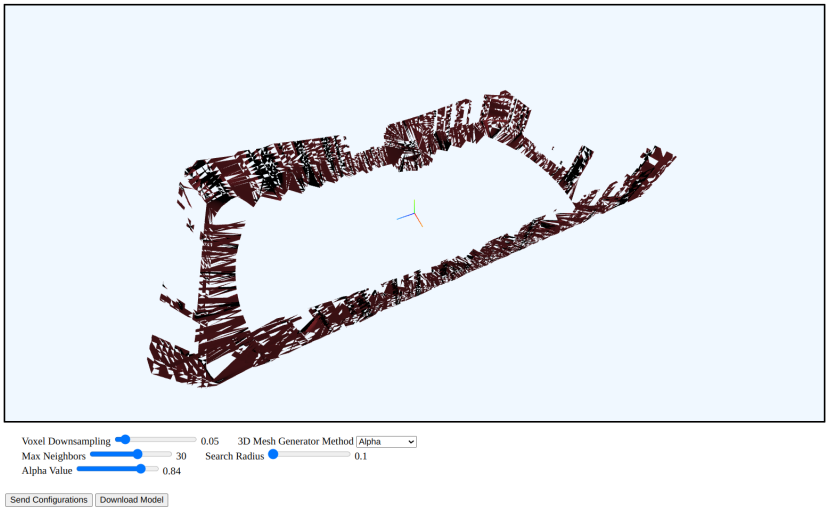

Figura 5.7: *Alpha shapes* para o cenário café

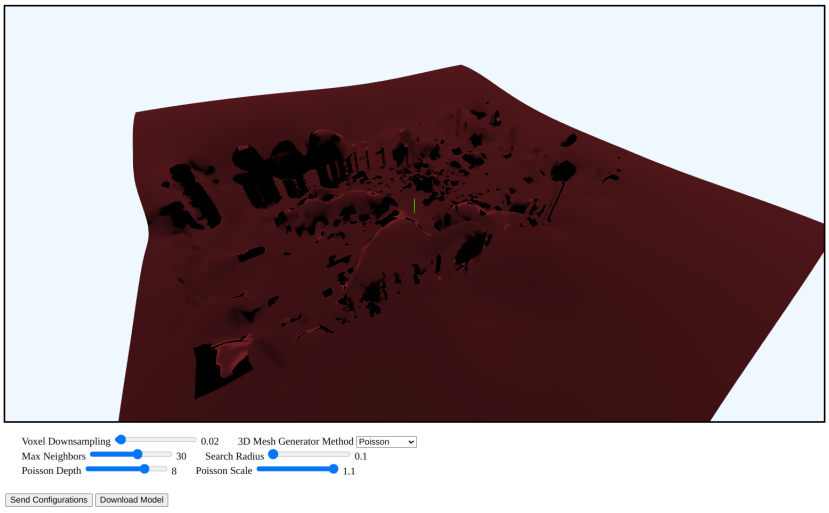

Figura 5.8: *Poisson surface reconstruction* para o cenário café

No cenário que contém um modelo de uma bomba de combustível, conseguimos verificar como é gerado uma geometria que não envolve totalmente o sensor. As figuras 5.10, 5.11 e 5.12 mostram respectivamente os algoritmos *ball pivoting*, *alpha shapes* e *poisson surface reconstruction*. A figura 5.9 mostra a nuvem de pontos obtida pelo simulador para o cenário da bomba de combustível.

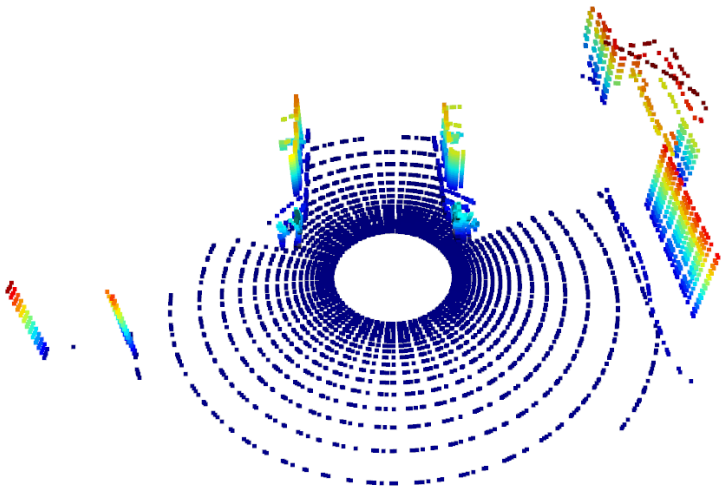

Figura 5.9: Nuvem de pontos do cenário bomba de combustível

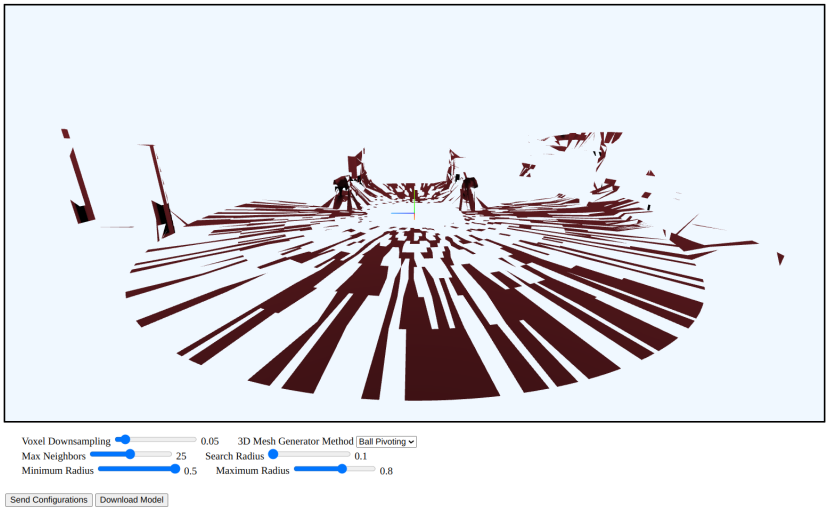

Figura 5.10: *Ball pivoting* para o cenário bomba de combustível

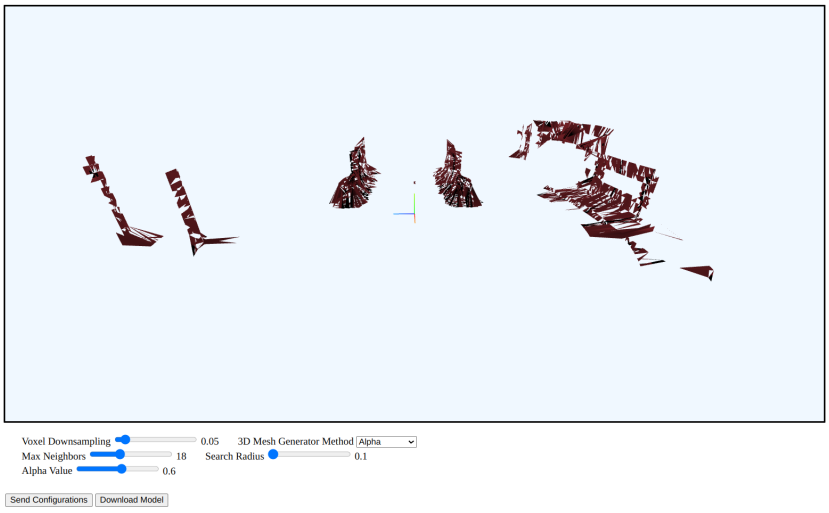

Figura 5.11: *Alpha shapes* para o cenário bomba de combustível

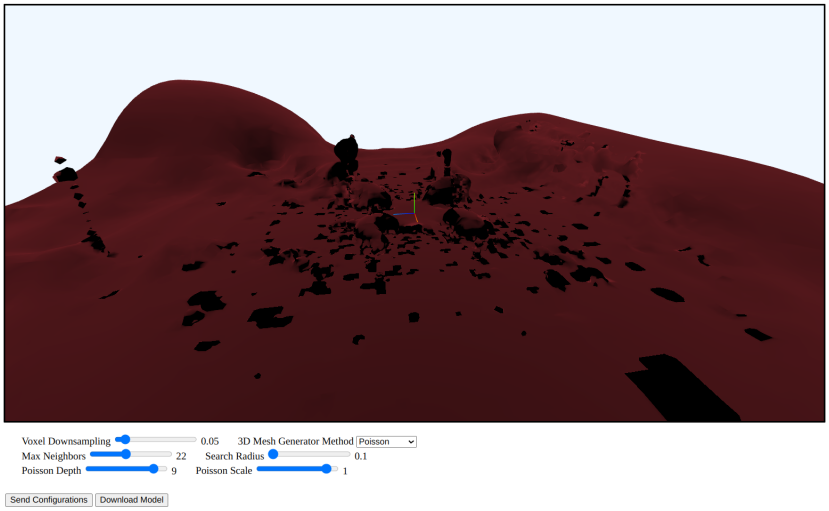

Figura 5.12: *Poisson surface reconstruction* para o cenário bomba de combustível

De forma a poder verificar o nível de detalhe que é possível gerar através da simulação, escolheu-se este cenário em específico. Contém 4 edifícios que possuem um grande nível de detalhe e aliado a isto foram colocados também 2 modelos de parques infantis pequenos opostos um do outro. As figuras 5.14, 5.15 e 5.16 mostram respectivamente os algoritmos *ball pivoting*, *alpha shapes* e *poisson surface reconstruction*. A figura 5.13 mostra a nuvem de pontos obtida pelo simulador para o cenário das escolas.

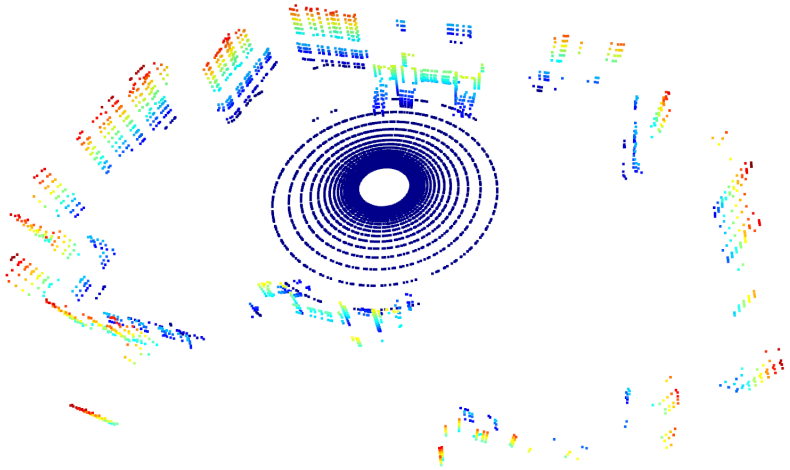

Figura 5.13: Nuvem de pontos do cenário escola

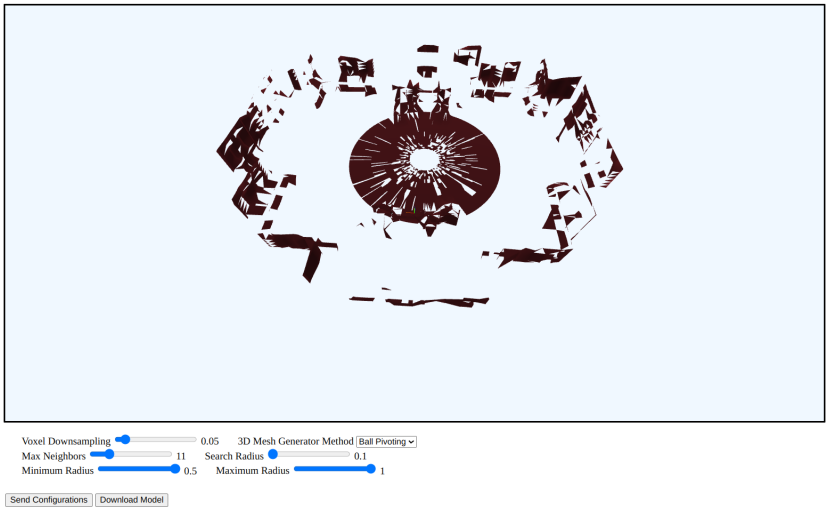

Figura 5.14: *Ball pivoting* para o cenário escola

Figura 5.15: *Alpha shapes* para o cenário escola

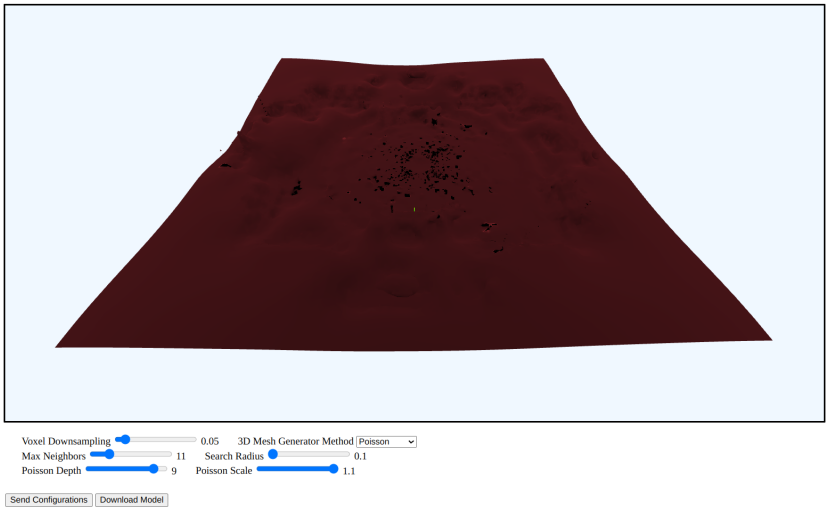

Figura 5.16: *Poisson surface reconstruction* para o cenário escola

Por fim este último cenário, que contém uma garagem com várias divisões, foi criado para testar a geração da geometria quando se altera a posição do sensor. Começamos com a posição inicial do sensor, gerando a divisão inicial, movendo depois o sensor para outra divisão. A figuras 5.18 e 5.19 mostram respectivamente o antes e o depois da movimentação do sensor. As figuras 5.4 e 5.20 mostram a posição inicial e a posição final do sensor no *Gazebo*. A figura 5.17 mostra a nuvem de pontos obtida pelo simulador para o cenário da garagem.

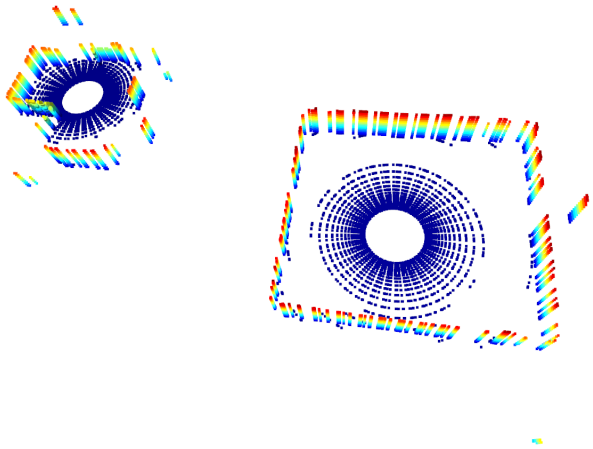

Figura 5.17: Nuvem de pontos do cenário garagem

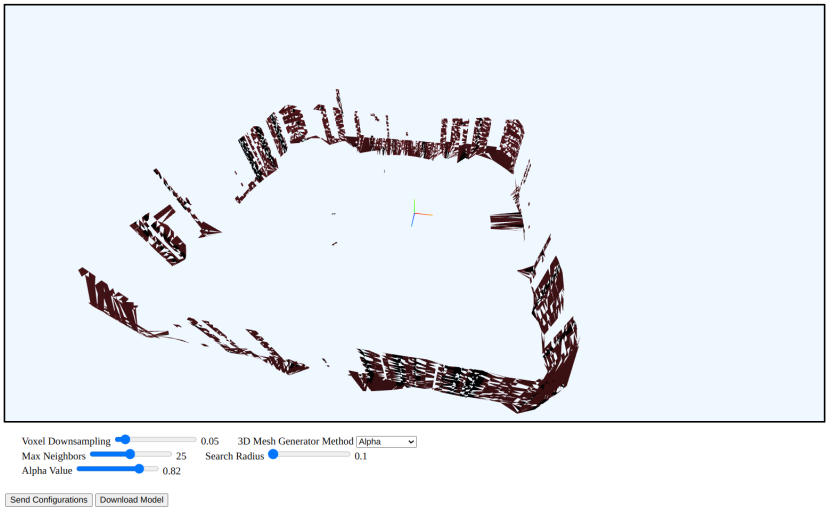

Figura 5.18: Antes da movimentação do sensor

Figura 5.19: Depois da movimentação do sensor

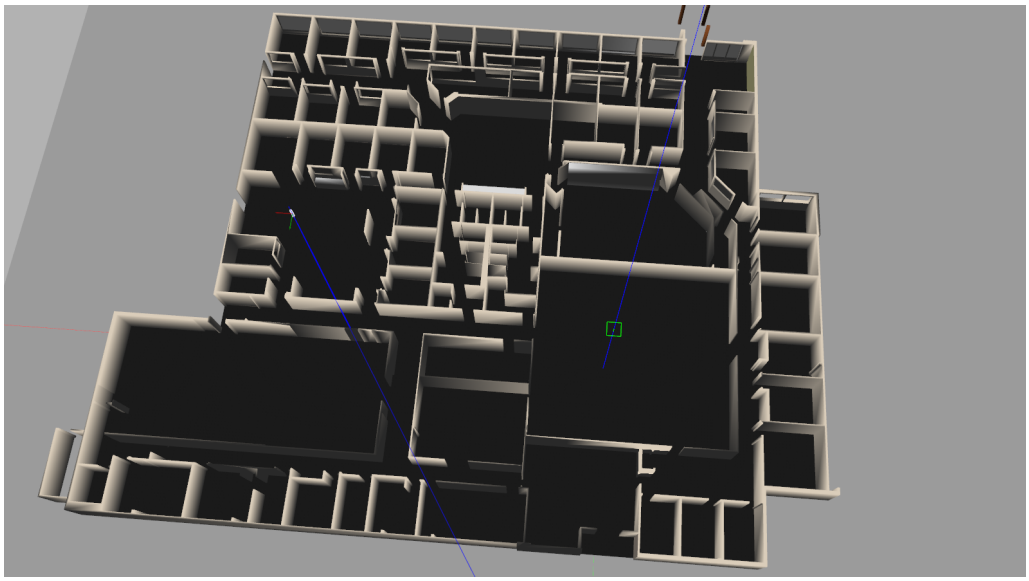

Figura 5.20: Posição final do sensor

5.3 Discussão dos resultados

Os resultados demonstram que a representação em termos da escala e do espaço está bem conseguida permitindo alterar os parâmetros de cada algoritmo para obter uma melhor geração. Verifica-se também que a interface é responsiva, espelhando o que está a acontecer no simulador. De notar ainda que em alguns casos o chão não é gerado na totalidade como por exemplo, no cenário da escola com o algoritmo *ball pivoting* (figura 5.14) ou que não é gerado de todo como no algoritmo *alpha shapes* (figura 5.15). Isto acontece devido à pouca densidade de pontos nessa zona tornando-se difícil manter o detalhe e gerar todas as faces do modelo com os parâmetros usados. Uma das formas de corrigir seria simular o comportamento de um robô a mover-se pelo espaço de forma a gerar uma maior densidade na nuvem de pontos. Isto introduziria uma maior complexidade na simulação em que o próprio comportamento do robô teria que ser diferente para cada um dos cenários, tornando-se especialmente complexo num cenário como o da garagem (figura 5.4).

Notou-se ainda que dos três algoritmos o que possui o desempenho mais fraco é o *poisson surface reconstruction*. Uma das causas deste fenómeno pode ser a própria natureza do algoritmo que considera os dados todos de uma vez. O algoritmo também está implementado de forma a focar-se na geração de geometria *smooth* que por si só depende de uma nuvem de pontos com bastante densidade. Por estas razões verificou-se que para este tipo de fim o algoritmo *poisson surface reconstruction* não é o mais adequado.

Estes resultados foram obtido num computador portátil que não possui uma performance gráfica suficientemente boa para conseguir executar este tipo de testes em tempo real, verificando-se um *delay* de poucos segundos entre o que estava acontecer no simulador e a sua representação na interface, tornando-se este *delay* mais notório à medida que a interface ia processando mais pon-

tos. Este défice de capacidade de computação tornou-se evidente quando se alterou a posição do sensor, demorando alguns minutos (cerca de 3 minutos) para mover o sensor para a nova posição e verificar essa alteração na interface.

Capítulo 6

Conclusões e Trabalho Futuro

6.1 Sumário

A computação gráfica é uma área que possibilita a criação de ferramentas que nos permite interagir com sistemas complexos. Um desses sistemas são os sistemas robóticos. É essencial conseguir inserir e extrair dados nestes sistemas de forma simplificada e eficaz. De forma a tornar o desenvolvimento destes sistemas mais eficaz são necessárias ferramentas de simulação que permitam testar antes de passar à fase de fabricação. Aqui estão incluídas interfaces gráficas que consigam expor os dados obtidos deste sistemas de forma clara e que permita a interação do utilizador com esses mesmos dados.

Neste projecto, o foco foi criar uma ferramenta que permita a visualização e interação de um ambiente gerado através de um sensor *LiDAR* simulado no *Gazebo*. Analisou-se o estado da arte das ferramentas existentes, chegando à conclusão que não existe propriamente uma interface que permita a visualização, interação e modificação da geometria gerada, como uma ferramenta web. Desta forma foi então desenvolvida uma interface gráfica web que permite essa visualização interação da geometria, permitindo a modificação dos parâmetros responsáveis pela sua geração, possuindo ainda a capacidade de receber e processar a nuvem de pontos de um sensor *LiDAR* perto de tempo real, de forma periódica.

Para conferir a eficácia da interface, criámos quatro cenários de testes em que cada um procura testar um ponto crítico do projecto. Verificou-se que a geração da geometria é fidedigna em termos de escala e mapeamento espacial, como se pode ver em todos os cenários de teste, mas peca pela preservação do detalhe. Verificou-se também que é responsivo, em que as modificações que se fizeram no simulador *Gazebo* tinham reflexo na interface gráfica. No entanto como estes testes foram computacionalmente intensivos a máquina usada não tinha a capacidade ideal para verificar a fundo o quão rápido ou não é o reflexo dessas modificações na interface.

Em conclusão foi conseguido desenvolver a interface gráfica adicionando características que permitem uma interação mais pormenorizada com a geometria gerada, num ambiente web.

6.2 Trabalho futuro

Um dos pontos que necessitam mais atenção, são os testes com dados de sensores reais. Apesar de o projecto possuir uma simulação de um sensor, o foco não foi a simulação mas sim o desenvolvimento da interface, aliado à falta de acesso a dados reais, não foi possível efetuar este tipo de testes.

Devido à complexidade da implementação, não foi possível simular um veículo robótico. Seria uma mais valia este tipo de simulação, pois tornava os testes mais próximos do que seria consumir dados reais e devido ao aumento da densidade da nuvem de pontos os resultados seriam melhores.

Seria uma mais valia implementar a bidireccionalidade de comunicação entre a interface e o simulador de forma a que se aplique na prática o conceito de *digital twin*.

Como foi possível verificar nos testes existem algoritmos cujo os resultados aparentam ser mais adequados que outros para a geração de geometria. Seria interessante adicionar algoritmos com outras bibliotecas de forma a tornar as possibilidades de geração mais amplas e com melhores resultados.

A optimização da aplicação também seria um aspecto a rever no futuro, tentando melhorar a reposta da interface ao dados recebidos pelo simulador ou sistema robótico.

Aproveitando o facto de se tratar de uma interface web, independente do simulador, seria possível efectuar testar em cenário real, utilizar a interface desenvolvida com um sistema robótico presente no terreno a recolher os dados do ambiente e enviá-los para uma equipa no laboratório que poderia estar numa outra parte do mundo.

Referências

[1] Advancing Front Methods. https://link.springer.com/referenceworkentry/10.1007/978-3-540-70529-1_313. Accessed: 2023-06.

[2] Animation system. https://threejs.org/docs/index.html?q=ani#manual/en/introduction/Animation-system. Accessed: 2023-06.

[3] Compute Vertex Normals. http://www.open3d.org/docs/release/python_api/open3d.geometry.TriangleMesh.html#open3d.geometry.TriangleMesh.compute_vertex_normals. Accessed: 2023-06.

[4] cplusplus. https://cplusplus.com/. Accessed: 2023-06.

[5] Create from point cloud alpha shape. http://www.open3d.org/docs/release/python_api/open3d.geometry.TriangleMesh.html#open3d.geometry.TriangleMesh.create_from_point_cloud_alpha_shape. Accessed: 2023-06.

[6] Create from point cloud ball pivoting. http://www.open3d.org/docs/release/python_api/open3d.geometry.TriangleMesh.html#open3d.geometry.TriangleMesh.create_from_point_cloud_ball_pivoting. Accessed: 2023-06.

[7] Create from point cloud poisson surface reconstruction. http://www.open3d.org/docs/release/python_api/open3d.geometry.TriangleMesh.html#open3d.geometry.TriangleMesh.create_from_point_cloud_poisson. Accessed: 2023-06.

[8] Creating a scene. https://threejs.org/docs/index.html#manual/en/introduction/Creating-a-scene. Accessed: 2023-06.

[9] Curl. https://curl.se/. Accessed: 2023-06.

[10] Design of an Intrinsically Safe Series-Series Compensation WPT System for Automotive LiDAR - Scientific Figure on ResearchGate. https://www.researchgate.net/figure/Basic-diagram-about-how-light-detection-and-ranging-LiDAR-technology-works_fig1_338366874. Accessed: 2023-06.

[11] Estimate Normals. http://www.open3d.org/docs/release/python_api/open3d.geometry.PointCloud.html#open3d.geometry.PointCloud.estimate_normals. Accessed: 2023-06.

[12] Flask. https://flask.palletsprojects.com/en/2.3.x/. Accessed: 2023-06.

[13] GLTFLoader. https://threejs.org/docs/index.html#examples/en/loaders/GLTFLoader. Accessed: 2023-06.

[14] HTMLElement: input event. https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/input_event. Accessed: 2023-06.

[15] Intermediate: Velodyne. https://classic.gazebosim.org/tutorials?cat=guided_i&tut=guided_i1. Accessed: 2023-06.

[16] Introducing JSON. https://www.json.org/json-en.html. Accessed: 2023-06.

[17] Introduction to Alpha Shapes. https://graphics.stanford.edu/courses/cs268-11-spring/handouts/AlphaShapes/as_fisher.pdf. Accessed: 2023-06.

[18] KDTree Search Param Hybrid. http://www.open3d.org/docs/release/python_api/open3d.geometry.KDTreeSearchParamHybrid.html. Accessed: 2023-06.

[19] LaserScan File Reference. https://osrf-distributions.s3.amazonaws.com/gazebo/msg-api/9.0.0/laserscan_8proto.html. Accessed: 2023-06.

[20] Point Cloud. http://www.open3d.org/docs/release/python_api/open3d.geometry.PointCloud.html. Accessed: 2023-06.

[21] Quaternion Class Reference. https://osrf-distributions.s3.amazonaws.com/gazebo/api/7.1.0/classgazebo_1_1math_1_1Quaternion.html. Accessed: 2023-06.

[22] Remove duplicated points. http://www.open3d.org/docs/release/python_api/open3d.geometry.PointCloud.html#open3d.geometry.PointCloud.remove_duplicated_points. Accessed: 2023-06.

[23] Remove Statistical Outlier. http://www.open3d.org/docs/release/python_api/open3d.geometry.PointCloud.html#open3d.geometry.PointCloud.remove_statistical_outlier. Accessed: 2023-06.

[24] ROS Concepts. http://wiki.ros.org/ROS/Concepts. Accessed: 2023-06.

[25] ROS_basic_concepts. http://wiki.ros.org/ROS/Concepts?action=AttachFile&do=view&target=ROS_basic_concepts.dia. Accessed: 2023-06.

[26] Routing. https://flask.palletsprojects.com/en/2.3.x/quickstart/#routing. Accessed: 2023-06.

[27] .SDF File Extension. https://fileinfo.com/extension/sdf. Accessed: 2023-06.

[28] Spherical coordinates. https://encyclopediaofmath.org/index.php?title=Spherical_coordinates. Accessed: 2023-06.

[29] Surface reconstruction. http://www.open3d.org/docs/release/tutorial/geometry/surface_reconstruction.html. Accessed: 2023-06.

[30] The Basics of LiDAR - Light Detection and Ranging - Remote Sensing. https://www.neonscience.org/resources/learning-hub/tutorials/lidar-basics. Accessed: 2023-06.

[31] Threejs fundamentals. https://threejs.org/manual/#en/fundamentals. Accessed: 2023-04.

[32] Vector3 Class Reference. https://osrf-distributions.s3.amazonaws.com/gazebo/api/7.1.0/classgazebo_1_1math_1_1Vector3.html. Accessed: 2023-06.

[33] Velodyne Acoustics HDL-32E user manual. https://manualsdump.com/en/manuals/velodyne_acoustics-hdl-32e/275446. Accessed: 2023-06.

[34] Voxel Down Sampling. http://www.open3d.org/docs/release/python_api/open3d.geometry.PointCloud.html#open3d.geometry.PointCloud.voxel_down_sample. Accessed: 2023-06.

[35] WebGL webgl: 2d and 3d graphics for the web. https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API. Accessed: 2023-04.

[36] What is the Delaunay triangulation in R^d? https://www.cs.mcgill.ca/~fukuda/soft/polyfaq/node30.html. Accessed: 2023-06.

[37] What is Voronoi diagram in R^d? https://www.cs.mcgill.ca/~fukuda/soft/polyfaq/node29.html. Accessed: 2023-06.

[38] *Introduction*, páginas 1–37. Springer London, London, 2009.

[39] Aakash Ahmad e Muhammad Ali Babar. Software architectures for robotic systems: A systematic mapping study. *Journal of Systems and Software*, 122:16–39, Dec 2016.

[40] Ahmed Khairadeen Ali, One Jae Lee, e Chansik Park. Near real-time monitoring of construction progress: integration of extended reality and kinect v2. Em *ISARC. Proceedings of the International Symposium on Automation and Robotics in Construction*, volume 37, páginas 24–31. IAARC Publications, 2020.

[41] Halldor Arnarson. Adigital twin simulation with visual components. Tese de mestrado, Faculty of Engineering Science and Technology, 2018.

[42] Barbara Rita Barricelli, Elena Casiraghi, e Daniela Fogli. A survey on digital twin: Definitions, characteristics, applications, and design implications. *IEEE Access*, 7:167653–167671, 2019.

[43] Michael Batty. Digital twins, 2018.

[44] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva, e G. Taubin. The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):349–359, 1999.

[45] Bergthor Björnsson, Carl Borrebaeck, Nils Elander, Thomas Gasslander, Danuta R Gawel, Mika Gustafsson, Rebecka Jörnsten, Eun Jung Lee, Xinxiu Li, Sandra Lilja, et al. Digital twins to personalize medicine. *Genome medicine*, 12:1–4, 2020.

[46] Stefan Boschert e Roland Rosen. *Digital Twin—The Simulation Aspect*, páginas 59–74. Springer International Publishing, Cham, 2016.

[47] Pingping Cai e Sanjib Sur. Deeppcd: Enabling autocompletion of indoor point clouds with deep learning. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol*, 6(2), 2022.

[48] Min-Lung Cheng, Masashi Matsuoka, Wen Liu, e Fumio Yamazaki. Near-real-time gradually expanding 3d land surface reconstruction in disaster areas by sequential drone imagery. *Automation in Construction*, 135:104105, 2022.

[49] Li Deren, Yu Wenbo, e Shao Zhenfeng. Smart city based on digital twins. *Computational Urban Science*, 1:1–11, 2021.

[50] Gilberto Echeverria, Nicolas Lassabe, Arnaud Degroote, e Séverin Lemaignan. Modular open robots simulation engine: Morse. Em *2011 IEEE International Conference on Robotics and Automation*, páginas 46–51, 2011.

[51] Herbert Edelsbrunner e Ernst P. Mücke. Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13(1):43–72, jan 1994.

[52] George Edwards, Joshua Garcia, Hossein Tajalli, Daniel Popescu, Nenad Medvidovic, Gaurav Sukhatme, e Brad Petrus. Architecture-driven self-adaptation and self-management in robotics systems. Em *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, páginas 142–151, 2009.

[53] Edward Glaessgen e David Stargel. *The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles*.

[54] Michael Grieves. Digital twin: manufacturing excellence through virtual factory replication. *White paper*, 1:1–7, 2014.

[55] Jinkang Guo e Zhihan Lv. Application of digital twins in multiple fields. *Multimedia Tools and Applications*, Feb 2022.

[56] Soohee Han, Mi-sook Kim, e Hong Seong Park. Open software platform for robotic services. *IEEE Transactions on Automation Science and Engineering*, 9(3):467–481, 2012.

[57] Jin Heo, Christopher Phillips, e Ada Gavrilovska. Flicr: A fast and lightweight lidar point cloud compression based on lossy ri. Em *2022 IEEE/ACM 7th Symposium on Edge Computing (SEC)*, páginas 54–67, 2022.

[58] Jinhan Hu, Aashiq Shaikh, Alireza Bahremand, e Robert LiKamWa. Characterizing real-time dense point cloud capture and streaming on mobile devices. Em *Proceedings of the 3rd ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges*, HotEdgeVideo '21, página 1–6, New York, NY, USA, Oct 2021. Association for Computing Machinery.

[59] Nathan Hughes, Yun Chang, e Luca Carlone. Hydra: A real-time spatial perception engine for 3d scene graph construction and optimization. *arXiv preprint arXiv:2201.13360*, 2022.

[60] Yuchen Jiang, Shen Yin, Kuan Li, Hao Luo, e Okyay Kaynak. Industrial applications of digital twins. *Philosophical Transactions of the Royal Society A*, 379(2207):20200360, 2021.

[61] Zhizhong Kang, Juntao Yang, Zhou Yang, e Sai Cheng. A review of techniques for 3d reconstruction of indoor environments. *ISPRS International Journal of Geo-Information*, 9(5), 2020.

[62] Tobias Kaupp, Alex Brooks, Ben Upcroft, e Alexei Makarenko. Building a software architecture for a human-robot team using the orca framework. Em *Proceedings 2007 IEEE International Conference on Robotics and Automation*, páginas 3736–3741, 2007.

[63] Michael Kazhdan, Matthew Bolitho, e Hugues Hoppe. Poisson surface reconstruction. Em *Proceedings of the fourth Eurographics symposium on Geometry processing*, volume 7, página 0, 2006.

[64] Alireza Khatamian e Hamid R Arabnia. Survey on 3d surface reconstruction. *Journal of Information Processing Systems*, 12(3), 2016.

[65] Dongsun Kim e Sooyong Park. Designing dynamic software architecture for home service robot software. Em Edwin Sha, Sung-Kook Han, Cheng-Zhong Xu, Moon-Hae Kim, Laurence T. Yang, e Bin Xiao, organizadores, *Embedded and Ubiquitous Computing*, páginas 437–448, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[66] N. Koenig e A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. Em *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, páginas 2149–2154 vol.3, 2004.

[67] Anis Koubaa. A service-oriented architecture for virtualizing robots in robot-as-a-service clouds. Em Erik Maehle, Kay Römer, Wolfgang Karl, e Eduardo Tovar, organizadores, *Architecture of Computing Systems – ARCS 2014*, páginas 196–208, Cham, 2014. Springer International Publishing.

[68] Heikki Laaki, Yoan Miche, e Kari Tammi. Prototyping a digital twin for real time remote control over mobile networks: Application of remote surgery. *IEEE Access*, 7:20325–20336, 2019.

[69] Reinhard Laubenbacher, James P Sluka, e James A Glazier. Using digital twins in viral infection. *Science*, 371(6534):1105–1106, 2021.

[70] Yongjae Lee, Byounghyun Yoo, e Soo-Hong Lee. Sharing ambient objects using real-time point cloud streaming in web-based xr remote collaboration. Em *The 26th International Conference on 3D Web Technology*, página 1–9, Pisa Italy, Nov 2021. ACM.

[71] Ruixu Liu, Tao Peng, Vijayan K.Asari, e John S. Loomis. Real-time 3d scene reconstruction and localization with surface optimization. Em *NAECON 2018 - IEEE National Aerospace and Electronics Conference*, páginas 280–285, 2018.

[72] Azad M. Madni, Carla C. Madni, e Scott D. Lucero. Leveraging digital twin technology in model-based systems engineering. *Systems*, 7(1), 2019.

[73] Anthony Mallet, Cédric Pasteur, Matthieu Herrb, Séverin Lemaignan, e Félix Ingrand. Genom3: Building middleware-independent robotic components. Em *2010 IEEE International Conference on Robotics and Automation*, páginas 4627–4632, 2010.

[74] Flávia Pires, Ana Cachada, José Barbosa, António Paulo Moreira, e Paulo Leitão. Digital twin in industry 4.0: Technologies, applications and challenges. Em *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, volume 1, páginas 721–726, 2019.

[75] Mario Prats, Javier Pérez, J. Javier Fernández, e Pedro J. Sanz. An open source tool for simulation and supervision of underwater intervention missions. Em *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, páginas 2577–2582, 2012.

[76] Concetta Semeraro, Mario Lezoche, Hervé Panetto, e Michele Dassisti. Digital twin paradigm: A systematic literature review. *Computers in Industry*, 130:103469, 2021.

[77] Quentin Serdel, Christophe Grand, Julien Marzat, e Julien Moras. Online localisation and colored mesh reconstruction architecture for 3d visual feedback in robotic exploration missions. Em *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, páginas 8690–8697, 2022.

[78] Ranjith K Soman e Jennifer K Whyte. A framework for cloud-based virtual and augmented reality using real-time information for construction progress monitoring. Em *Proceedings of the Joint Conference on Computing in Construction (JC3), Heraklion, Greece*, páginas 4–7, 2017.

[79] Panagiotis Stavropoulos e Dimitris Mourtzis. Chapter 10 - digital twins in industry 4.0. Em Dimitris Mourtzis, organizador, *Design and Operation of Production Networks for Mass Personalization in the Era of Cloud Technology*, páginas 277–316. Elsevier, 2022.

[80] Denys Sytnyk. Ambiente de simulação para o sistema de exploração robótica subaquática unexmin. Tese de mestrado, Instituto Superior de Engenharia do Porto, 2018.

[81] Manuel Sánchez, Jorge L. Martínez, Jesús Morales, Alfredo Robles, e Mariano Morán. Automatic generation of labeled 3d point clouds of natural environments with gazebo. Em *2019 IEEE International Conference on Mechatronics (ICM)*, volume 1, páginas 161–166, 2019.

[82] Aristotelis Christos Tagarakis, Damianos Kalaitzidis, Evangelia Filippou, Lefteris Benos, e Dionysis Bochtis. *3D Scenery Construction of Agricultural Environments for Robotics Awareness*, páginas 125–142. Springer International Publishing, Cham, 2022.

[83] Fei Tao, Jiangfeng Cheng, Qinglin Qi, Meng Zhang, He Zhang, e Fangyuan Sui. Digital twin-driven product design, manufacturing and service with big data, Mar 2017.

[84] Hendrik van der Valk, Hendrik Haße, Frederik Möller, Michael Arbter, Jan-Luca Henning, e Boris Otto. A taxonomy of digital twins. 08 2020.

[85] Hendrik van der Valk, Joachim Hunker, Markus Rabe, e Boris Otto. Digital twins in simulative applications: A taxonomy. Em *2020 Winter Simulation Conference (WSC)*, páginas 2695–2706, 2020.

[86] Minglan Xiong e Huawei Wang. Digital twin applications in aviation industry: A review. *The International Journal of Advanced Manufacturing Technology*, 121(9-10):5677–5692, 2022.

[87] Hengxu You, Fang Xu, e Eric Du. Robot-based real-time point cloud digital twin modeling in augmented reality. *Transforming Construction with Reality Capture Technologies*, 2022.

[88] J. Yuh. Design and control of autonomous underwater robots: A survey.

[89] J. Yuh, Giacomo Marani, e D. Richard Blidberg. Applications of marine robotic vehicles, Jul 2011.

[90] Qian-Yi Zhou, Jaesik Park, e Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.