universidad de león

MOVIMENTOS REATIVOS NO ROBÔ TURTLEBOT UTILIZANDO O KINECT

Tânia Ferreira Carrera

Dissertação de Mestrado em Engenharia Industrial

Projeto realizado sob a orientação de:
Professor Doutor José Lima e Professor Doutor José Gonçalves
do Departamento de Eletrotecnia do Instituto Politécnico de Bragança
Professor Doutor Vicente Matellán e Professor Doutor Francisco Lera
do Departamento de Informática da Universidade de León

Bragança, 17 de Julho de 2013

Agradecimentos

Quero agradecer, em primeiro lugar, aos meus pais. Sem o seu constante apoio, amor e sacrifício, tudo isto não seria possível.

Aos meus orientadores da Universidade de León de Espanha, o Professor Doutor Vicente Matellán e o Professor Doutor Francisco Lera e, aos meus orientadores do Instituto Politécnico de Bragança, o Professor Doutor José Lima e o Professor Doutor José Gonçalves por me receberem tão bem e estarem sempre dispostos a ajudar.

Aos meus amigos e colegas de faculdade por me darem apoio e estarem sempre dispostos a ajudar nos momentos mais complicados deste percurso.

Um obrigado a todos!

Resumo

Com a chegada da visão computacional, cada vez mais os investigadores estudam novos métodos para aquisição de imagens 3D de objetos para poderem ser úteis no nosso dia-a-dia na recolha de informações tanto nas áreas de engenharia, medicina, arquitetura, educação, entre outras.

O lançamento do sensor Microsoft Kinect para a Xbox, projetado como acessório para detetar o movimento dos jogadores, veio chamar a atenção, não só pelo seu preço acessível, mas também por conseguir disponibilizar informações sem criptografia, permitindo que fosse usado para outros fins além dos videojogos.

Este trabalho teve como base a análise e investigação da capacidade de obter informações acerca das distâncias em profundidade, deste sensor e, aplicá-la aos movimentos reativos efetuados pelo robô TurtleBot permitindo desta forma a interação com o meio e tornar a sua movimentação autónoma.

O algoritmo utilizado em linguagem C++, na estrutura de software ROS (Robot Operating System), é uma síntese do método de mudança de direção baseado no centro (*centroid* em inglês), dos pontos detetados nos objetos, que permitirá o desvio do robô perante um obstáculo e, um método com *clustering* de PCL (Point Cloud Library), que permitirá o reconhecimento de humanos.

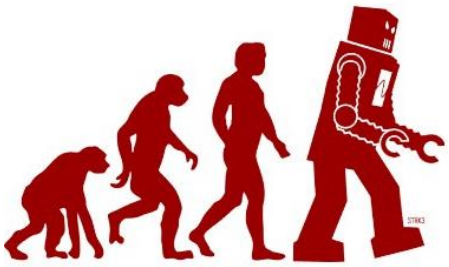
Abstract

With the arrival of computer vision, researchers have been doing research in methods to capture 3D images of objects, in order to use the information collected in different areas such as medicine, architecture, engineering, education, and among others.

The launch of the Microsoft Kinect sensor for Xbox, designed as an accessory to detect the movement of the players came calling the attention of researchers, not only by your affordable price but also for getting available information unencrypted, allowing it to be used for other purposes than videogames.

In this work, have been explored the ability to get information about the distances in depth, of this sensor, and apply it to the movements made by the TurtleBot robot, thus enabling the interaction with the environment and make their movement autonomously.

The algorithm used in C language + +, on software structure of ROS (Robot Operating System), is a summary of the method of changing direction based on the centroid from detected points on objects, which will allow the change of direction of the robot when it finds an obstacle, and a method, with clustering of PCL (Point Cloud Library), which will allow the recognition of humans.



"O passado serve para evidenciar as nossas falhas e dar-nos indicações para o progresso do futuro."

Henry Ford

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABELAS

Acrónimos

API – Application Programming Interface

BSD – Berkeley Software Distribution

CCD – Charge-Coupled Device

CMOS – Complementary Metal-Oxide-Semiconductor

IP – Internet Protocol

NTP – Network Time Protocol

OpenCV – Open source Computer Vision

OpenNI – Open Natural Interaction

PC – Personal Computer

PCD – Point Cloud Data

PCL – Point Cloud Library

RGB – Red, Green, Blue

ROS – Robot Operating System

SDK – Software Development Kit

SI – Sistema Internacional

SSH – Secure Shell

URI – Uniform Resource Identifier

USB – Universal Serial Bus

VGA – Video Graphics Array

1. Introdução

A constante evolução tecnológica permite que os robôs estejam cada vez mais preparados para interagir de acordo com o mundo que os rodeia.

Prevê-se que a utilização de robôs móveis venha a aumentar e que entre no nosso dia-a-dia, como já existe atualmente para o auxílio na realização de tarefas doméstica, o aspirador Roomba [3] da iRobot.

Ao serem dispositivos móveis há uma maior importância que estes possam reconhecer em todos os momentos a situação onde se encontram na execução de uma tarefa e, detetar rapidamente as alterações e ser capaz de adaptar-se, como na interação com humanos ou animais. Possíveis aplicações poderão ser os trabalhos de vigilância e a teleassistência a idosos.

1.1 Motivação

O objetivo principal deste trabalho foca-se em conseguir que o robô TurtleBot, um robô móvel com o auxílio de rodas, navegue de forma autónoma pelo meio envolvente e que seja capaz de interagir com o mesmo, desviando-se dos objetos que se tornem obstáculos na sua movimentação e que interrompa o seu movimento perante a presença de um humano, emitindo um sinal.

O seu movimento no meio apenas será bem-sucedido se usufruir de sensores capazes de recolher informações e transforma-las em sinais para o robô. Um destes sensores essenciais para esta ocorrência, neste projeto, é o sensor Microsoft Kinect.

Foi utilizado o ROS que é um ambiente que veio facilitar e agilizar a tarefa de construção de programas de controlo para robôs, uma vez que proporciona uma grande quantidade de pacotes que implementam cada tarefa, como captação de dados de sensores e transformação, comunicação, entre outros. Isto permite que o utilizador

apenas se preocupe em programar o comportamento do robô, tendo em atenção aos sensores e atuadores com que este está equipado.

1.2 Organização do documento

Este documento está organizado em seis capítulos, iniciado por esta breve introdução onde consta a apresentação e contextualização do problema, e o objetivo deste projeto.

O segundo capítulo deste documento descreve o software ROS, utilizado neste projeto, realça e descreve os seus principais componentes, comandos e ferramentas mais utilizados. Apresenta também exemplos de robôs que já suportam ROS.

No terceiro capítulo aborda-se o tema de sensores, onde são descritos os tipos existentes e o funcionamento de sensores semelhantes ao utilizado neste trabalho, comparando o Microsoft Kinect (sensor usado) com o que tem as características mais similares.

O quarto capítulo deste relatório caracteriza e descreve o sensor Microsoft Kinect – um dos materiais essências para a realização deste projeto. Apresenta a sua constituição e o método de cálculo de profundidade, principal característica deste sensor para captação de imagens 3D.

No capítulo cinco é apresentado o Point Cloud Library (PCL). É exposta a sua composição, descrevendo os seus módulos constituintes. É apresentado o resultado dos algoritmos passados de PCL a pcl_ros, durante o decorrer deste trabalho, descrevendo as suas funções.

O capítulo seis trata do desenvolvimento do projeto. Descreve o material utilizado, o processo de inicialização de ROS e do robô. Apresenta a descrição do programa criado e os resultados obtidos.

Finalmente, são expostas as conclusões do trabalho realizado, assim como apresentadas sugestões de trabalho futuro.

Em anexo, encontra-se os comandos mais utilizados em ROS e exposto o código em linguagem C++ desenvolvido durante este projeto.

2. Estado da Arte

Neste capítulo são discutidos os aspetos fundamentais de ROS tais como a sua caracterização, os seus componentes principais na realização de comunicações, comandos e ferramentas.

2.1 ROS

As dificuldades encontradas na criação de softwares robóticos devido ao protocolo de comunicação que cada robô possui (sendo diferente de robô para robô), aos diferentes formatos de imagem que cada câmara captura e à capacidade de poder adaptar vários sensores em computadores diferentes, colocou a necessidade de criar um sistema que permitisse gerir a comunicação entre eles, ROS.

Robot Operating System (ROS) é uma estrutura de software destinada a facilitar o desenvolvimento de software em robôs. Para este desenvolvimento, ROS fornece bibliotecas e ferramentas que auxiliam a criação de aplicações, fornece abstração de hardware, controlo de dispositivos de baixo nível, drivers de dispositivos, visualizadores, transmissão de mensagens e na gestão de pacotes.

ROS possui uma linguagem independente. Existem três bibliotecas principais definidas para ROS o que torna possível programar em Python, Lisp ou C++, contudo, existem ainda duas bibliotecas experimentais que possibilitam a programação de ROS em Java ou LUA.

É um software totalmente livre, sob os termos de licença BSD (Berkeley Software Distribution) que permite que qualquer pessoa possa utilizar para investigação e comercialização, permitindo que se desenvolvam aplicações para robôs mais eficientes de forma mais rápida numa plataforma comum. ROS também utiliza códigos (Drivers e algoritmos) de outros projetos igualmente livres como, por exemplo:

- Simuladores de projetos,
- Processamento de imagens e bibliotecas de visão artificial de OpenCV (Open source Computer Vision),
- Algoritmos de planeamento de OpenRave.

A biblioteca está direcionada para um sistema UNIX, Ubuntu (Linux) que é um dos sistemas operativos que esta biblioteca suporta, contudo também pode ser utilizada em Microsoft Windows, Fedora, Mac OS X, Gentoo, Slackware, Debian, OpenSUSE e Arch, sendo estes ainda considerados como "experimentais".

ROS tem tido as seguintes distribuições até a atualidade:

- ROS 1.0, publicado a 22 de Janeiro de 2010
- ROS Box Turtle, publicado a 2 de Março de 2010
- ROS C Turtle, publicado a 2 de Agosto de 2010
- ROS Diamondback, publicado a 2 de Março de 2011
- ROS Electric, publicado a 30 de Agosto de 2011
- ROS Fuerte, publicado a 23 de Abril de 2012
- ROS Groovy Galapagos, publicado a 31 de Dezembro de 2012
- ROS Hidro Medusa, publicado a 22 de Maio de 2013 [1]

2.2 ROS e componentes

Um dos componentes de ROS é o ROS Core que contém: o ROS Master que se caracteriza como um servidor XML-RPC centralizado que negoceia as conexões das comunicações, regista e examina os recursos gráficos de ROS; os parâmetros de

servidor que armazena os parâmetros de configuração e outros dados arbitrários; rosout que basicamente é um stdout "*network-based*" para mensagens legíveis pelo utilizador.

O sistema de ROS está organizado por *packages* e *stacks*. Por um lado, um *package* é a unidade fundamental dentro da organização do software ROS, a entidade de mais baixo nível na organização, é um diretório que contém os nós, bibliotecas externas, dados, arquivos de configuração e um arquivo de configuração XML chamado *manifest*.xml.

Por outro, uma *stack* é uma coleção de *packages* que oferece um conjunto de funcionalidades como a navegação, posicionamento, entre outros. Uma *stack* é um diretório que contém os diretórios do *package*, além de um arquivo de configuração chamado *stack*.xml.

O princípio básico de um "robot operating system" é executar um grande número de arquivos em paralelo que necessitam de trocar dados de forma síncrona ou assíncrona.

Os conceitos fundamentais da implementação ROS que permite que se cumpram esses objetivos denominam-se por nós, mensagens, tópicos e serviços.

2.2.1 **Nós**

O nó é um arquivo executável que pode corresponder a um sensor, ao processamento ou monitorização de um algoritmo, etc. Cada vez que um nó é executado, é declarado ao Mestre que por sua vez, é um nó e um serviço de registo que possibilita que outros nós se encontrem e troquem dados entre eles como banco de dados centralizado no qual os nós podem armazenar os dados.

Os nós comunicam-se com outros nós através do envio de mensagens.

2.2.2 **Mensagens**

Uma mensagem é formada por uma estrutura de dados que compreende uma combinação de tipos primitivos (inteiros, strings, booleanos, etc.). A mensagem pode ser composta por outra mensagem e por matrizes de outras mensagens.

Existem vários tipos de mensagens de ROS como:

- GridCells,
- Image,
- PointCloud,
- Polygon,
- LaserScan,
- Odometry,
- PoseArray.

A descrição da mensagem é armazenada num arquivo que descreve a estrutura da mensagem: *package_name/msg/myMessageType.msg*.

Um nó envia uma mensagem, dependendo da sua publicação, num determinado tópico.

2.2.3 **Tópicos**

Um tópico é um sistema de transporte de dados com base num sistema de subscrever / publicar, é, de certa forma, um barramento de mensagens assíncrono. O nó envia e recebe mensagens em tópicos e, quando está interessado num determinado tipo de dados irá subscrever para o tópico adequado.

Um ou mais nós são capazes de publicar dados de um tópico, e um ou mais nós podem ler dados sobre o assunto, os dados são trocados de forma assíncrona, por meio de um tópico e em sincronia por meio de um serviço.

2.2.4 **Serviços**

Um serviço é descrito por um nome de *string* e um par de mensagens de um determinado tipo: um para a solicitação e outro para a resposta, situação análoga a serviços web em que são definidos por URIs (Uniform Resource Identifier) e têm os tipos definidos.

Contrariamente aos tópicos, apenas um nó pode anunciar um serviço sem um nome em particular, como por exemplo, haver só um serviço chamado como ”classificar imagem”, assim como só pode haver um serviço web em um determinado URI.

A descrição do serviço é armazenada num arquivo que descreve as estruturas de dados de solicitações e respostas: *package_name / srv / myServiceType.srv* [2].

Na imagem seguinte é ilustrado as comunicações possíveis entre nós, serviços e tópicos.

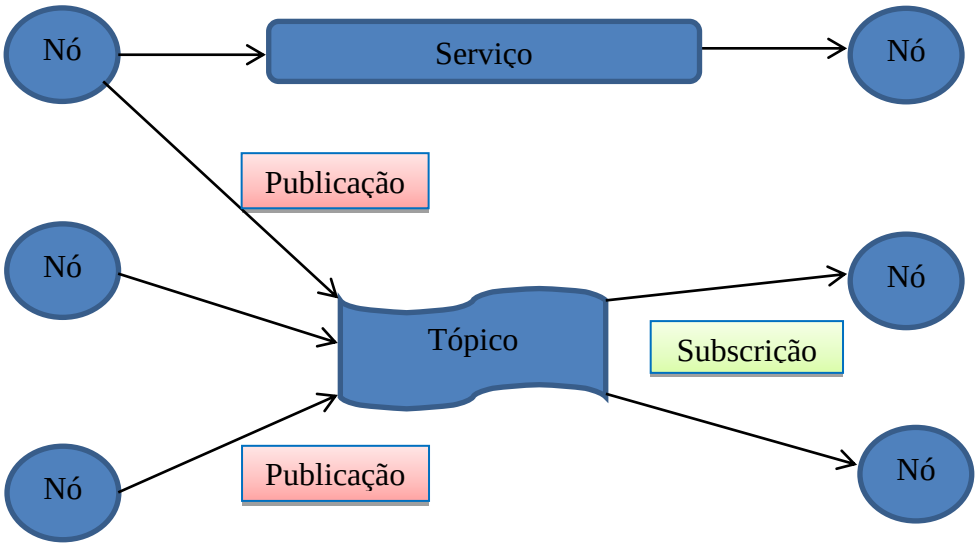


Figura 1: Comunicações possíveis entre nós, serviços e tópicos.

De seguida serão descritos alguns dos principais comandos utilizados para efetuar uma interface usando ROS.

2.3 Comandos de ROS

Aqui serão abordados alguns comandos essenciais na realização de interface com ROS, descrevendo alguns argumentos mais comuns.

- roscore: chama um mestre ROS, que é útil para a realização de experiências com ROS numa estação de trabalho quando o TurtleBot, por exemplo, está offline.

- roscd: este pacote é um script de conveniente que permite mover imediatamente para o diretório do pacote especificado.
- roscreate-pkg: é o pacote que contem dependências e inicializa um diretório de pacotes que contem o código-fonte para um ou mais módulos. A estrutura da diretoria do pacote será criada num novo pacote chamado subdiretório, dentro da pasta atual, que deve aparecer em $ ROS PACOTE PATH. Tipicamente, as dependências devem incluir o pacote roscpp (que contém as ligações ROS C++), assim como outros pacotes que serão utilizados tal como pcl_ros. As dependências podem ser modificadas posteriormente, editando o arquivo manifest.xml no diretório raiz do pacote, a fim de poder adicionar mais tags dependentes. Os nós de ROS podem ser adicionados a um projeto adicionando pela adição do diretório rosbuild_add_executable no ficheiro CMakeLists.txt, também localizado na raiz do pacote.
- rosmsg: é um "verbo" contém argumentos e mostra informações sobre os atuais tipos de mensagens definidos que podem ser transmitidos sobre tópicos. Quando o "verbo" é show e os argumentos são package / messagetype, por exemplo, uma "Referência API" (Application Programming Interface) de tipos e nomes das variáveis, na estrutura hierárquica da mensagem correspondente, é exibida.
- rosparam: suporta "verbos" tais como: *list*, *set*, *get* e *delete*. No caso do primeiro, o *parameterpath* pode ser omisso se uma lista completa é desejada. A invocação *set* espera um argumento adicional que contém o novo valor a ser anexado.
- rosrun: é um pacote simplesmente usado para executar um nó, uma vez que o pacote foi compilado com o make.
- rosservice: permite a interface com os serviços disponíveis no momento, sobre os tipos de serviços podem ser enviados. Quando o "verbo" é *list*, o ServicePath pode, opcionalmente ser omisso, no caso em que todos os serviços poderão ser mostrados. Com *call*, o usuário pode chamar serviço

passando argumentos e recebendo uma resposta suportada pelo tipo de serviço. O tipo de "verbo" é importante, pois permite devolver o pacote e o tipo de serviço correspondente ao serviço no caminho especificado.

- rossvc: permite a consulta de tipos de serviços definidos no momento para passar sobre serviços. Quando o "verbo" é *show* e os argumentos são da forma *package* / *servicetype*, o comando gera uma " Referência API" – uma lista modelo dos tipos e nomes das variáveis no tipo struct que representa o tipo de serviço.
- rostopic: fornece uma ponte para tópicos anunciados no momento, durante o qual as mensagens podem ser passadas. Quando o "verbo" é *list*, *topic-path* pode, opcionalmente, ser omisso para listar todos os tópicos disponíveis. Com *echo*, o usuário pode subscrever um tópico e ver os dados que estão a ser subscritos a ele. Por outro lado, ao chamar *pub* vai permitir publicar o tópico, que vai influenciar os nós que estão subscritos naquele momento a ele. O "verbo" *type* é particularmente útil: ele imprime o tipo de pacote e mensagem de um determinado tópico registado.

2.4 Ferramentas utilizadas em ROS

ROS, como já foi referido, é uma coleção de ferramentas e algoritmos. Alguns são muito usados durante a programação, simulação ou execução de tarefas do robô. Algumas das ferramentas e algoritmos que o programador ROS mais utiliza são:

- Stage: simulador 2D,
- Gazebo: simulador 3D,
- Rviz: sistema de visualização 3D,
- Pacotes Tf: utilizado para a manipulação de coordenadas e transformação,

- OpenCV : processamento de imagem,
- PointCloudLibrary: reconstrução do ambiente 3D a partir de medições a laser.

Cada vez mais a utilização de ROS torna-se importante para o desenvolvimento e evolução do mundo robótico, existindo atualmente 175 organizações ou indivíduos que divulgam publicamente software ROS nos arquivos indexados de *ROS.org* onde, até Outubro de 2009 existiam apenas 50. Esta notória evolução pode ser observada no gráfico da Figura 2.

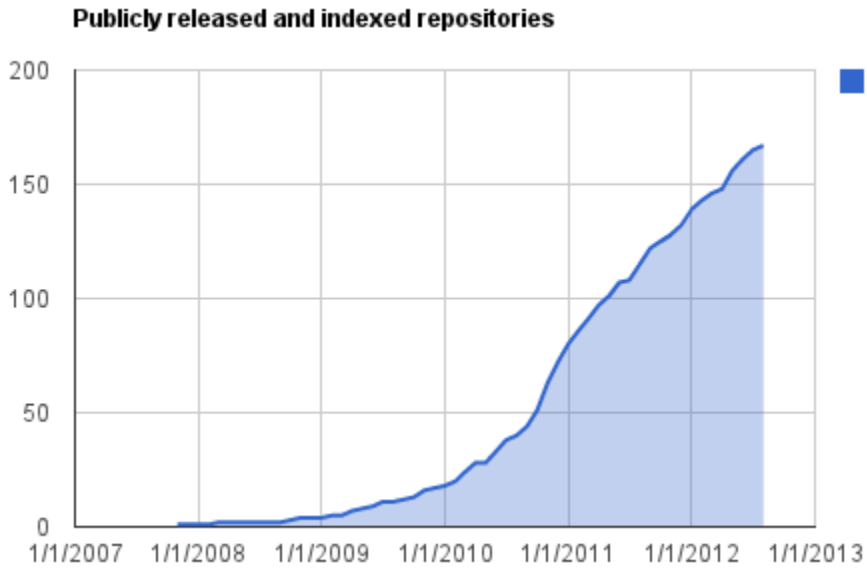


Figura 2: Arquivos indexados até meados de 2012 [4].

Como se pode analisar através da Figura 2, houve um aumento significativo nos últimos dois anos, de organizações ou indivíduos que divulgaram publicamente software ROS nos arquivos indexados e, esse crescimento tende a manter-se.

Como consequência, a lista de robôs compatíveis com ROS está em crescimento constante.

Sem contar com os cerca de 40 robôs PR2s em todo o mundo, existem já centenas de robôs que utilizam ROS [4]. Existem atualmente 28 robôs com instruções de instalação suportados como podemos analisar no gráfico da Figura 3:

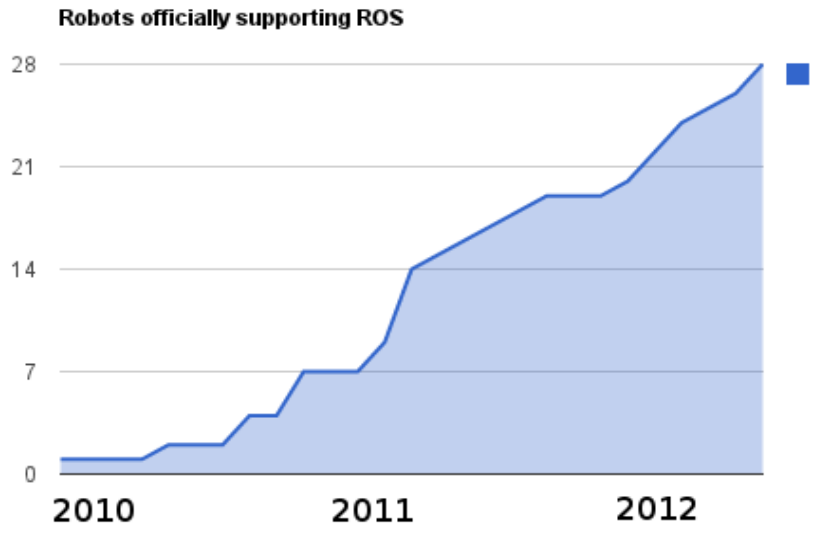


Figura 3: Gráfico de número de robôs que suportam ROS [4].

Como se pode observar, pela análise da Figura 3, existe um aumento constante da criação de robôs que sejam compatíveis com ROS até finais de 2012.

Seguidamente serão enunciados alguns dos robôs mais conhecidos que já usufruem de ROS:

a) Aldebaran Nao

b) Willow Garage PR2

c) Lego Mindstorms NXT

d) iRobot Roomba

e) TurtleBot

f) Kobuki

Figura 4: Exemplos de Robôs que usufruem de ROS [3].

O capítulo 3 será dirigido aos sensores existentes, classificando-os e descrevendo o seu funcionamento perante o meio envolvente. Será também apresentada uma comparação entre os dois sensores mais parecidos no que diz respeito às suas arquiteturas.

3. Sensores

Uma das mais importantes tarefas de um sistema autónomo é a aquisição de informações do meio envolvente, contudo, isto apenas se torna possível se forem utilizados os diversos sensores que possui um robô.

Depois de capturada essa informação é necessário tratá-la de modo a que seja percetível para o robô entender o meio que o rodeia.

Existem vários tipos de sensores que podem equipar um robô, podendo estes serem classificados de diferentes formas [5]. No que diz respeito à sua função podem ser classificados como:

- Propriocetivos – medem valores internos ao robô como a velocidade, os ângulos das juntas, a bateria, entre outros;
- Exterocetivos – adquirem informação do meio envolvente do robô e os dados que têm de ser interpretados de modo a poderem ser percebidos pelo robô como por exemplo, distâncias e intensidade luminosa.

Uma vez que para a concretização deste projeto foi utilizado um sensor exterocetivo, será aprofundado mais o tema sobre este tipo de sensores.

Na classificação de sensores existe outra componente importante, o modo de interação com o meio. Desta forma, a classificação anteriormente mencionada pode ainda ser subdividida em:

- Passivos – medem diretamente a energia do meio que é adquirida pelo sensor como por exemplo os microfones e os termómetros;
- Ativos – emitem energia no meio e medem a sua correspondente reação como por exemplo os ultrassónicos, que medem a reflexão das ondas sonoras e, *laser rangefinders* que medem a reflexão dos feixes luminosos.

No projeto em concreto foi utilizado um sensor exterocetivo e ativo que utiliza o mapeamento ativo baseado na triangulação. Estes tipos de sensores serão aprofundados durante este capítulo.

3.1 Mapeamento ativo baseado na triangulação

Este tipo de sensores usa propriedades geométricas nas suas medições a fim de poderem deduzir sobre a distância a que se encontram os objetos.

Os sensores que são baseados em triangulação são ativos, pois emitem um feixe de luz (textura, ponto, linha…) no ambiente que está a visualizar. A reflexão desse feixe é capturada por um recetor que, dependendo das variáveis do ambiente conhecidas, irá tirar conclusões acerca da distância do objeto [5].

3.1.1 **Luz Estruturada**

O sistema de visão por luz estruturada é bastante utilizado na indústria, não só pela sua simplicidade, como também pela robustez na deteção de defeitos.

Na Figura 5 pode ser observado um exemplo deste tipo de tecnologia, constituída por uma câmara CCD (Charge-Coupled Device) ou CMOS (Complementary Metal-Oxide-Semiconductor) e um emissor, que projeta no ambiente um feixe de luz de padrão e textura conhecidos – luz estruturada [5].

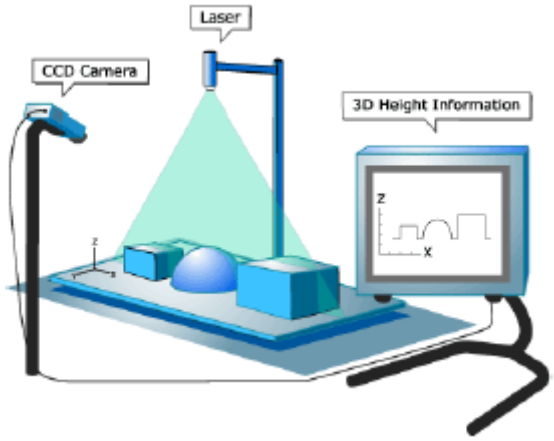


Figura 5: Exemplo de um sistema de luz estruturada.

No caso da Figura 5, o recetor mede a posição da reflexão ao longo dos dois eixos ortogonais, podendo desta forma, ser classificado como sensor 2D.

3.1.2 **Tecnologias da PrimeSense**

A PrimeSense [6] é uma empresa israelita, fundada em 2005, que ficou conhecida pela implementação da tecnologia que equipa o Microsoft Kinect.

Atualmente esta empresa é a responsável pela criação de diversos equipamentos que funcionam com base num princípio igual ao demonstrado na Figura 6.

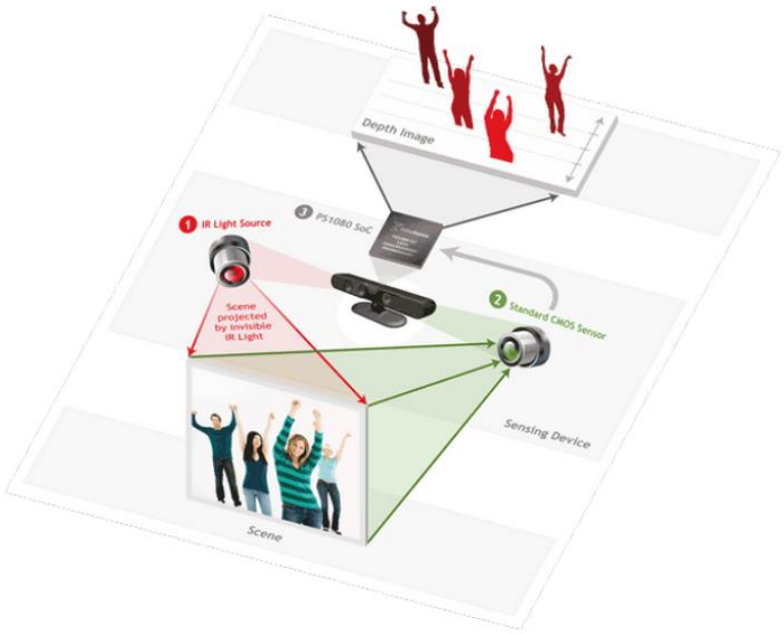


Figura 6: Princípio de funcionamento do Kinect

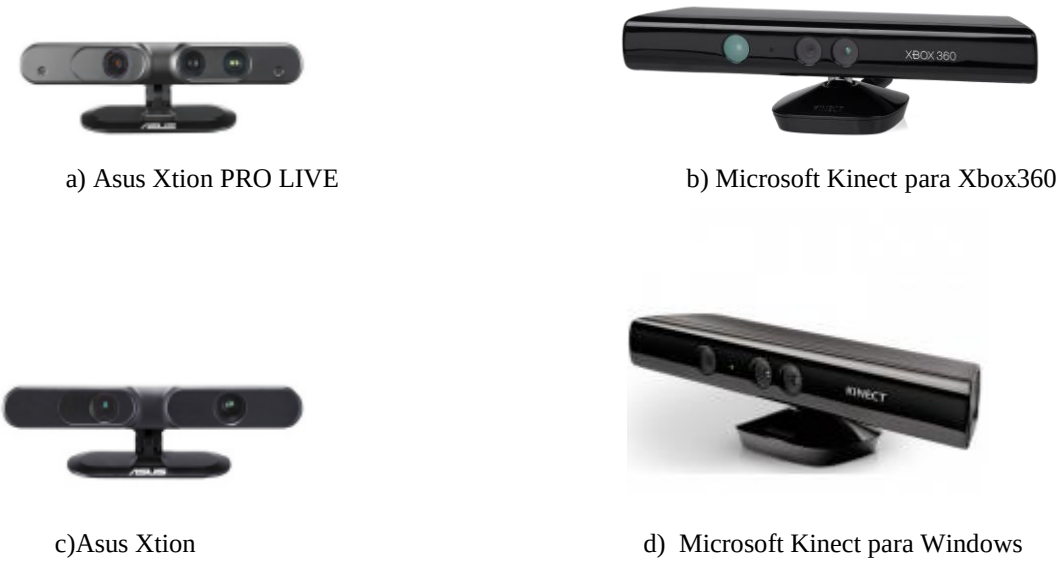
Este tipo de sensores têm vindo a tornar-se cada vez mais importante na comunidade científica, nomeadamente no domínio da imagem tridimensional. As suas características rapidamente atraíram a atenção dos investigadores de sistemas de mapeamento e modelação 3D.

Estes sensores são uma alternativa atrativa aos *lasers rangefinders* (dispositivos que mapeiam o ambiente baseando-se no tempo de voo do feixe luminoso; adquirem dados num plano e estão normalmente associados a um mecanismo mecânico com um espelho rotativo que projeta vários feixes em todo o meio envolvente) para aplicações na área de *mapping, indoor robot localization*, vigilância, entre outros.

O primeiro dispositivo criado pela PrimeSense foi o PSDK reference apresentado na Figura 7, contudo, já foi retirado do mercado.

Figura 7: PSDK reference

Existem no mercado, atualmente, quatro produtos deste tipo ilustrados na Figura 8.



a) Asus Xtion PRO LIVE

b) Microsoft Kinect para Xbox360

c)Asus Xtion

d) Microsoft Kinect para Windows

Figura 8: Dispositivos de mapeamento 3D atualmente no mercado da PrimeSense [7].

Na Figura 8, em a) a Asus Xtion PRO LIVE, apenas usada em computadores mas muito semelhante ao Kinect b) está uma imagem do Microsoft Kinect que surgiu como dispositivo de entrada para consolas e jogos de Xbox 360, em c) observamos a Asus Xtion que é igual à anterior, contudo, não usufrui da camara RGB (Red, Green, Blue) e, por último em d) a imagem do Microsoft Kinect para Windows que, em relação ao Kinect para Xbox 360, possui algumas vantagens tais como:

- É licenciado oficialmente (pela Microsoft) para uso comercial (enquanto que o Kinect para a Xbox apenas é licenciado para jogos pessoais);
- A qualidade dos drivers e suportes são melhores;
- Possui de configurações manuais para a câmara RGB;
- Capacidade de adquirir imagens a menor distância (40 cm).

O sensor Kinect é um dispositivo rápido e preciso dentro dos seus limites, pois possui da capacidade de capturar simultaneamente uma imagem em profundidade e outra colorida a uma taxa de 30 fps. Esta integração permite o resultado de uma nuvem de pontos que contém cerca de 300 mil pontos em cada *frame* [7].

A tabela que se segue, apresenta uma comparação entre os dois dispositivos mais acessíveis no mercado e também mais idênticos – o Microsoft Kinect e o Asus Xtion PRO LIVE – pois, apesar de terem o mesmo princípio de funcionamento, existem características que os distinguem.

Tabela 1: Tabela comparativa entre o Microsoft Kinect e o Asus Xtion PRO LIVE.

Dispositivo	Vantagens	Desvantagens
Microsoft Kinect	• Boa qualidade dos *drivers* de fábrica, • Estável com vários modelos de hardware, • Motor de inclinação controlado remotamente • Mais acessível no mercado – mais popular.	• Maior e menos compacto, • Mais pesado, • Atinge no máximo 30fps, • Necessita de alimentação externa.
Asus Xtion PRO LIVE	• Mais pequeno e compacto, • Mais leve, • Consegue 60fps a uma resolução de 320x240, • Alimentação por USB.	• Sem motor de inclinação, • Não funciona com alguns controladores USB (como o USB 3.0), • Menor qualidade dos *drivers*, de fábrica, • Menos acessível no mercado – menos popular.

3.2 OpenNI SDK

Atualmente, OpenNI (Open Natural Interaction) é a maior estrutura de desenvolvimento de deteção de 3D, sendo o SDK reconhecido para o processo evolutivo na área da visão computacional e em soluções 3D, utilizada por exemplo pelo sensor Kinect. Fundado em novembro de 2010, foi formado com o intuito de promover e padronizar a interoperabilidade e a compatibilidade – Natural Interaction (NI) – dos dispositivos, aplicações e *middleware* [8].

A comunidade OpenNI fornece aos seus utilizadores uma gama completa de ferramentas software, recursos, tutoriais, bibliotecas de *middleware*, aplicações e plataformas de correspondência ativas. Com a utilização do OpenNI SDK2 os investigadores têm a capacidade de desenvolver aplicações para diversos mercados como a robótica, jogos e TV, computadores, serviços de saúde, dispositivos móveis, exposições interativas, indústria entre outros. A Figura 9 apresenta um esquema ilustrativo da arquitetura de OpenNi SDK2.

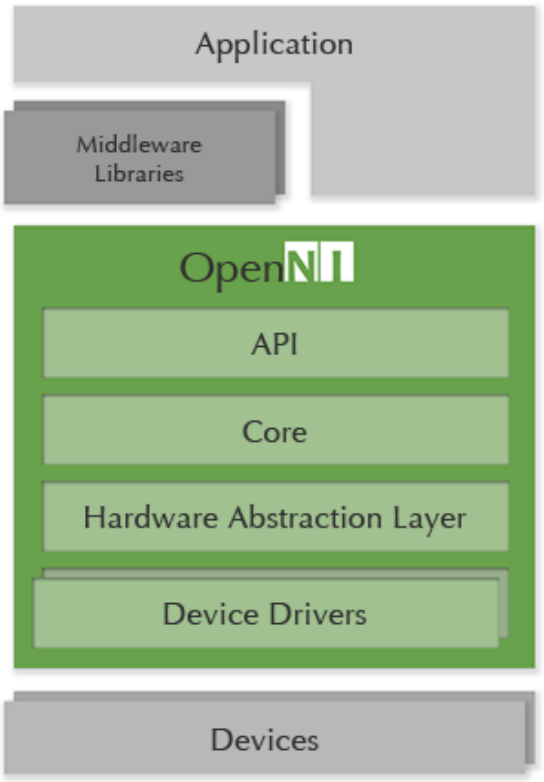


Figura 9: Arquitetura do OpenNI SDK2 [8].

OpenNI permite obter um conjunto de APIs para escrever as aplicações de deteção de 3D, que permite facilitar a comunicação com dispositivos de baixo nível como os sensores de áudio.

A utilização de OpenNI SDK2 acarreta inúmeras vantagens entre as quais são destacadas as seguintes:

- Maior compatibilidade;
- Grande oferta de bibliotecas *middleware*: reconstrução 3D, reconhecimento de objetos, entre outros;
- Permite a programação de eventos orientados;
- Suporta as mais recentes gerações de sensores 3D;
- Mais flexível: controlo de unidades de profundidade, controlo de orientação, etc;

- Simplicidade na distribuição: cópia privada de OpenNI e Nite para cada aplicação.
- Suporte melhorado e simplificado para multisensores [8].

No capítulo 4 será apresentado mais detalhadamente o sensor utilizado neste projeto – o Microsoft Kinect, descrevendo a sua arquitetura e referindo os vários processos pelos quais se obtem uma imagem de profundidade.

4. Microsoft Kinect

O sensor Kinect da Microsoft Corp foi introduzido no mercado em novembro de 2010 como um dispositivo de entrada para consolas e jogos de Xbox 360, sendo um produto muito bem sucedido, com mais de 10 milhões de aparelhos vendidos em março de 2011.

A sociedade Computer Vision, descobriu rapidamente que a tecnologia dos sensores de profundidade no Kinect, poderia ser utilizada para outros fins para a além dos jogos de computadores e, a um custo mais baixo que as tradicionais câmaras-3D, (tais como câmaras baseadas em tempos de voo).

Em junho de 2011, a Microsoft lançou um kit de desenvolvimento de software (SDK) para o Kinect, permitindo que seja usado como uma ferramenta para produtos não comerciais, promovendo a programação do Kinect no ambiente Microsoft Windows e, estimular ainda mais o interesse pelo produto.

A tecnologia utilizada no sensor Kinect foi originalmente desenvolvida pela empresa PrimeSense, como referido anteriormente, lançando a sua versão de um SDK para ser utilizado com o sensor Kinect como parte da organização OpenNI. O SDK deve ser independente dos outros dispositivos e, a empresa ASUS, agora, também apostou na produção de um sensor com muitas das mesmas capacidades utilizadas pelo sensor Kinect (incluindo a deteção de profundidade), como já mencionado, que trabalha com o OpenNI SDK.

4.1 Constituição do Microsoft Kinect

Este dispositivo de mapeamento 3D é constituído por uma câmara de infravermelhos, uma câmara RGB, dois microfones, um projetor de infravermelhos e um motor que lhe possibilita variar a inclinação sobre o eixo horizontal.

A câmara de infravermelhos, juntamente com o projetor, faz uma triangulação dos pontos no espaço, originando uma nuvem de pontos num espaço 3D. A câmara RGB pode ser usada para obter informações acerca da cor.

Estes componentes quando estão calibrados permitem obter uma nuvem de pontos colorida.

Como sensor de medida, o Kinect, pode fornecer a medida sobre a imagem de infravermelhos, a imagem RGB e a imagem de profundidade.

Na Figura 10 apresenta-se um diagrama que exemplefica a arquitetura do Kinect.

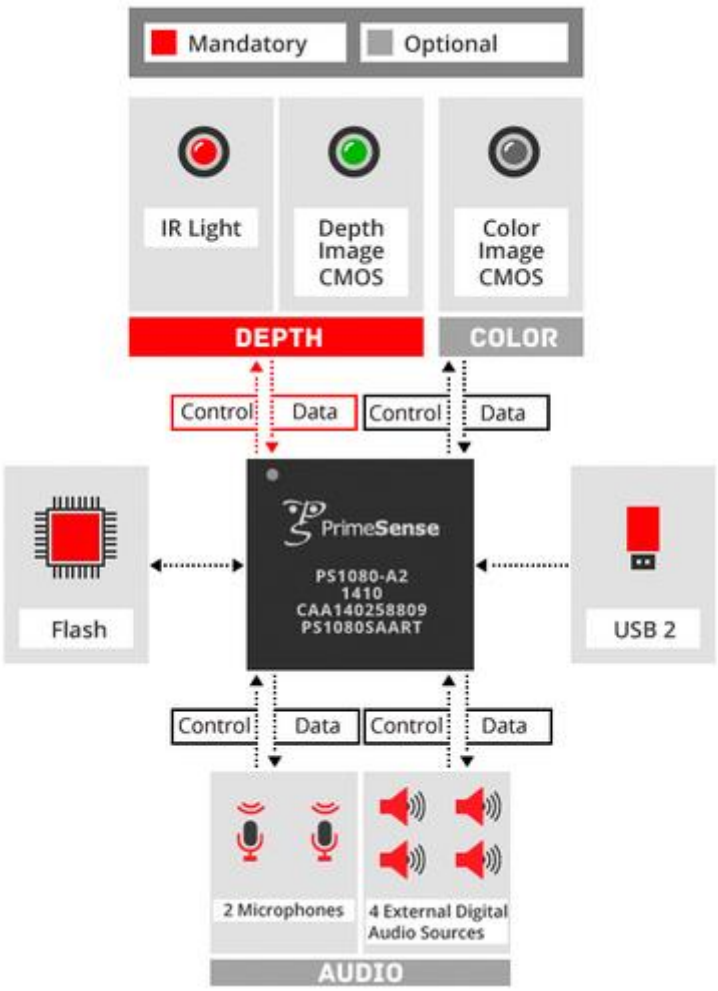


Figura 10: Diagrama da arquitetura do sensor Kinect [9].

A câmara de infravermelhos tem como função analisar e descodificar a projeção do padrão de infravermelhos utilizado para triangular o meio 3D, tem uma resolução de 640x480 pixéis com 11 bits de sensibilidade e um ângulo de visão até 57º na horizontal e 45º na vertical.

A câmara de RGB, por sua vez, tem uma resolução também de 640x480 pixéis, com 8 bits de resolução VGA (Video Graphics Array) e um ângulo de visão até 63º na horizontal e 50º na vertical, esta câmara necessita de calibração que, no domínio da visão computacional, entende-se como um processo para a determinação da relação matemática que, para o respetivo sistema, transforma os pontos 3D existente no mundo, em pontos 2D na câmara e, vice-versa.

A imagem de profundidade é obtida através de triangulação entre a imagem adquirida pela câmara de infravermelhos e, o projetor. Quanto menor for a distância à câmara, maior será a precisão, ou seja, menores serão os valores de quantização.

4.2 Cálculo da profundidade

As explicações relativamente ao funcionamento do Kinect, encontradas em diversos artigos, baseiam-se principalmente no seu desmantelamento, de forma a compreender a construção e constituição deste dispositivo e em testes para compreender o seu funcionamento no cálculo da profundidade.

4.2.1 Princípio de funcionamento

A projeção de pontos na gama do infravermelho tem um padrão conhecido, demonstrado na Figura 11 e, recorre a um elemento difusor e difrativo [10].

No momento da construção do Kinect foram adquiridas uma série de imagens de referência, a diferentes distâncias e guardadas no próprio Kinect, assim para as primeiras aquisições, a câmara de infravermelhos analisa o meio envolvente e calcula, para cada ponto projetado, a triangulação entre a imagem virtual (o padrão guardado), e o padrão observado. Nas seguintes aquisições de imagem, irá calcular a triangulação entre a *frame* atual e a anterior.

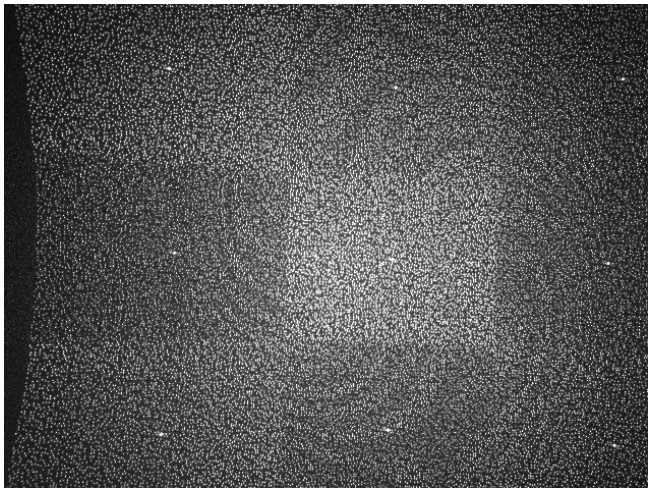


Figura 11: Imagem padrão de infravermelhos. [10]

O dispositivo Kinect calcula, portanto, a profundidade com base na deformação, do seu padrão conhecido, induzida pelo meio, em que cada ponto da imagem de infravermelhos tem um ponto do mundo associado [5].

O tamanho dos pontos projetados pelo dispositivo, varia dependendo da distância e da orientação do sensor ao objeto como demonstra a Figura 12.

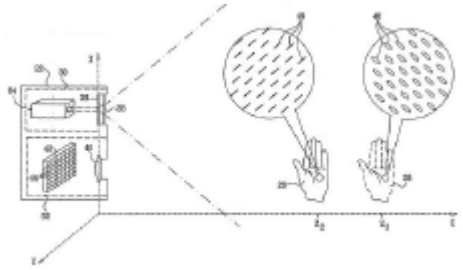


Figura 12: Variação do tamanho dos pontos

Quanto maior for a distância do ponto projetado ao emissor, maior será o tamanho desse ponto, devido ao feixe emitido tomar a forma cónica com o vértice coincidente no sensor. A forma deste feixe também pode alterar com a orientação do objeto, a uma dada distância, tendendo para uma forma mais ou menos oval dependendo da inclinação do objeto.

Este aumento do tamanho dos pontos tem como consequência a perda de precisão pois, a triangulação é feita sobre os pontos maiores, logo podemos afirmar que o cálculo de profundidade será mais incerto em distância maiores.

No Kinect, o sensor de infravermelhos, possui diferentes focagens para orientações distintas o que permite que a PrimeSense possa dividir o cálculo de profundidade em três regiões diferentes, observáveis na Figura 13 [5]:

- Região de alta precisão: objetos situados entre 0,8m e 1,2m;
- Região de média precisão: objetos situados entre 1,2m e 2,0m;
- Região de baixa precisão: objetos situados entre 2,0m e 3,5m.

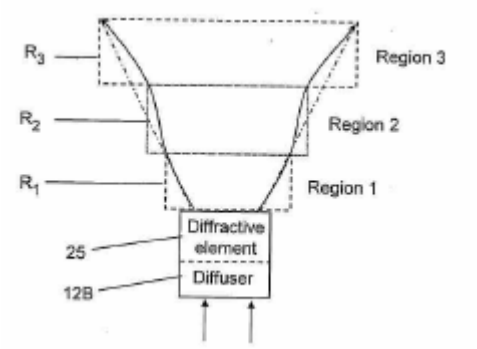


Figura 13: Esquema representativo das regiões de funcionamento, original da patente PrimeSense

Este sensor também deteta objetos acima dos 3,5m, contudo pode apresentar erros de medições que podem ultrapassar os 10cm.

Na Figura 14 pode examinar-se a imagem real em RGB à esquerda e a sua correspondente imagem de profundidade:

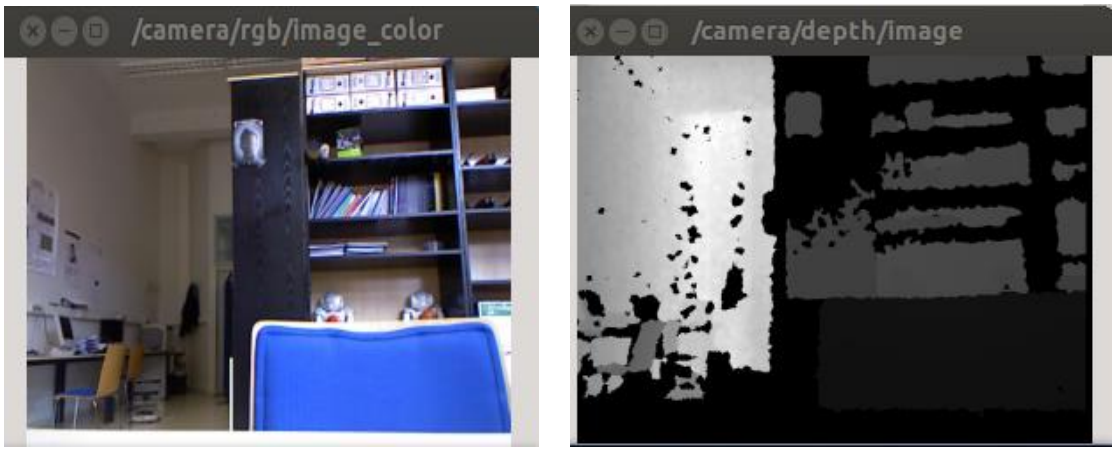


Figura 14: Imagens captadas com o Kinect: a da direita em RGB e a da esquerda em profundidade

A captação destas imagens foi feita através dos comandos de ROS: *rosrun image_view image_view image:=/camera/rgb/image_color* e *rosrun image_view image_view image:=/camera/depth/image*, respetivamente, na linha de comandos do terminal do Ubuntu 12.10.

4.2.2 **Cálculo por aproximação matemática**

O cálculo de profundidade também pode ser efetuado através de uma função matemática. Esta função comporta-se como um conversor para metros (SI) dos dados devolvidos diretamente pelo Kinect.

Um dos modelos atuais que consegue aproximar esta conversão mais próxima da realidade é o modelo proposto para a calibração apresentado por Stephane Magnenat e que se traduz na seguinte equação [11]:

$$p_m = 0{,}1236 \tan((p_k / 2842{,}5) + 1{,}1863)$$

No que diz respeito à equação anterior, p_m corresponde ao valor em metros e, p_k ao valor que é devolvido pelo dispositivo Kinect. Esta conversão não utiliza parâmetros intrínsecos do Kinect, permitindo obter informações de profundidade em unidades SI sem ser necessário ter os dados calculados pela calibração.

4.2.3 **Cálculo por disparidade**

Um sistema estereoscópico comum é calibrado de forma às imagens serem paralelas e terem linhas horizontais correspondentes. Este tipo de sistema é traduzido por:

$$z = b\, f / d$$

em que *z* diz respeito à profundidade em metros, *b* à distância entre as câmaras, *f* à distância focal em pixéis e *d* à disparidade também em pixéis [10].

No caso de a disparidade ser zero, a profundidade é infinita logo corresponde a uma situação em que os raios de cada câmara nunca se encontram por serem paralelos, o que pode conduzir a um caso em que as imagens não estão sobrepostas e como consequência impede que haja informações sobre profundidade.

Contudo, neste dispositivo, a disparidade nula não corresponde a uma profundidade infinita pois é calculada através da seguinte equação:

$$d=(1/8)(d_{off} - k_d)$$

em que *d* diz respeito à disparidade, k_d à disparidade devolvida pelo Kinect e o d_{off} corresponde a um valor próprio de cada dispositivo. O coeficiente 1/8 resulta da conversão de unidades de [pixel] para [pixel / 8]. Dito isto, o cálculo da profundidade pode ser efetuado através da seguinte expressão:

$$z=(d f) ((1/8)(d_{off} - k_d))$$

em que aqui, o *b* toma um valor aproximado de 0,075m (distância entre a câmara de infravermelhos e o projetor), e o d_{off} toma valores, tipicamente, por volta dos 1090 [11].

Na Figura 15 é possível observar a imagem real em RGB à esquerda e à direita a imagem correspondente à disparidade.

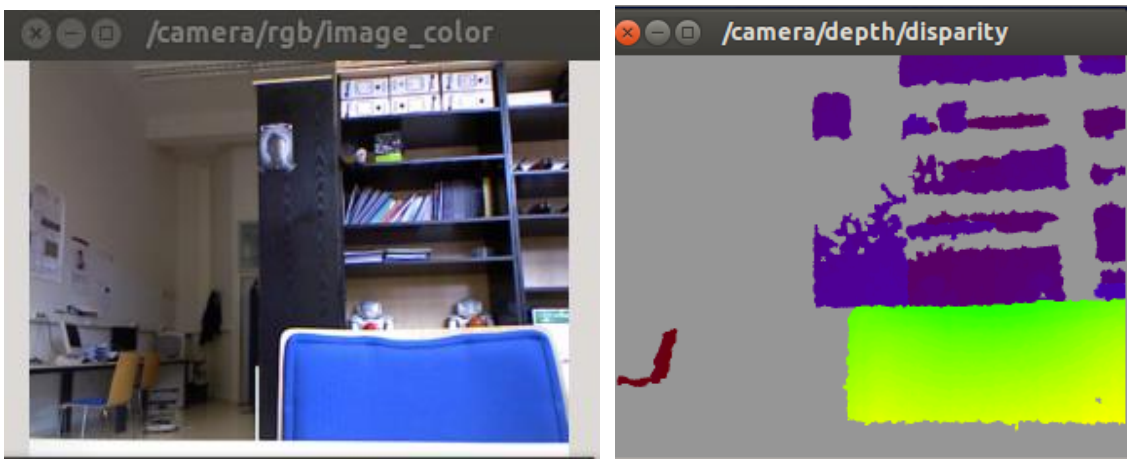


Figura 15: Imagem em RGB à direita, à esquerda correspondente à disparidade.

A captação destas imagens foi feita através dos comandos de ROS: *rosrun image_view image_view image:=/camera/rgb/image_color* e *rosrun image_view disparity_view image:=/camera/depth/disparity*, respetivamente, na linha de comandos do terminal do Ubuntu 12.10, como também mencionado anteriormente.

No capítulo 5 será abordado o tema sobre PCL, descrevendo o seu desenvolvimento e em que consiste. Será realizada uma breve descrição sobre os módulos que constitui abordando suas características e funcionalidades. Serão também referidos e descritos os algoritmos analisados (passados a ROS), durante este projeto.

5. Point Cloud Library (PCL)

PCL (Point Cloud Library) é, como o próprio nome indica uma biblioteca independente, que contém inúmeros algoritmos para projetos de imagens 2D e 3D e, processamento de nuvens de pontos para aplicações robóticas.

Encontra-se em linguagem C++ e está disponível para uso comercial e de pesquisa sob a licença BSD e é financiado por várias empresas tais como Willow Garage, NVIDIA, Google, Toyota, Open Perception, entre outras [12].

A biblioteca contém algoritmos de filtragem (*filtering*), de reconstrução da superfície (*surface reconstruction*), de estimativa de recurso (*feature estimation*), de registo (*registration*), de segmentação (*segmentation*), de ajuste do modelo (*model fitting*) entre outros [13].

Alguns dos primeiros algoritmos desenvolvidos que sustentam a base de hoje, tiveram início durante o trabalho do Dr. Radu Bogdan Rasu, na Technische Universitaet Muenchen, em Munique, Alemanha. Posteriormente, em 2009, foi aperfeiçoado na Willow Garage. O objetivo do Dr. Rusu era criar uma infraestrutura comum para aplicações e pesquisas de processamento de nuvens de pontos em 3D.

O seu desenvolvimento teve início em Março de 2010, em Willow Garage, com o propósito de ter um conjunto de bibliotecas e, uma série de ferramentas foram escritas como parte de ROS com o objetivo de ajudar a manipular e navegar em ambientes complexos 3D, com o robô PR2.

Antes de novembro de 2010, foi tomada a decisão de começar a criação de um novo projeto independente denominado de PCL, a fim de poder beneficiar a grande comunidade 3D em geral.

No final de Março de 2011, PCL deu os seus primeiros passos como um projeto lançado em separado, tendo o seu próprio domínio web em www.pointclouds.org e, tendo uma rápida e grande expansão devido ao apoio financeiro, como referenciado anteriormente.

Figura 16: Um dos logos de PCL

Em Junho de 2011, três meses depois do lançamento da primeira versão (versão 1.0), PCL contou com o apoio de mais de 120 colaboradores em todo o Mundo, com cerca de 30 universidades diferentes, institutos de pesquisa e empresas comerciais. Atualmente, este projeto continua em constante crescimento, estando atualmente na versão 1.7 [14].

5.1 Módulos de PCL

Como foi mencionado anteriormente, a biblioteca contém algoritmos que simplificam a tarefa de processamento de nuvens de pontos em imagens 2D e 3D. Esses algoritmos estão organizados por módulos:

- Filtros;
- Features;
- Registros;
- *Kdtree*;
- Segmentação;
- *Sample_consensus*;
- *Range_image*;
- *Io*;
- Pontos-chave;
- *Octree*;

- Superfícies;
- Visualizações.

5.1.1 **Filtros**

O módulo de filtros apresenta algoritmos de filtros para remoção de ruído, por exemplo. Devido aos erros de medição, alguns conjuntos de dados apresentam um grande número de pontos sombra o que irá complicar uma estimativa das características de nuvem de pontos local em 3D.

Alguns desses pontos indesejados podem ser filtrados através de uma análise estatística sobre a vizinhança de cada ponto, e eliminar aqueles que não seguem determinados critérios.

A implementação mais dispersa em PCL de remoção dos pontos indesejados, baseia-se no cálculo da distribuição das distâncias entre cada ponto e os seus pontos vizinhos, no conjunto de dados de entrada. Assumindo uma distribuição Gaussiana, com uma média e um desvio padrão, todos os pontos cujas distâncias estejam fora do intervalo definido, serão eliminados do conjunto de dados.

5.1.2 ***Features***

Este módulo contém as estruturas de dados e mecanismos para a estimativa de *features* 3D, a partir dos dados da nuvem de pontos. *Features* 3D são representações, em determinados pontos 3D, ou posições no espaço, que descrevem padrões geométricos com base na informação disponível à volta do ponto.

A região de dados selecionada à volta do ponto em estudo é designada como vizinhança. Duas das características geométricas mais utilizadas são a curvatura estimada da superfície subjacente e a normal do ponto em estudo *p*. Ambas são consideradas características locais, assim como caracterizar um ponto usando as informações fornecidas pelos seus pontos vizinhos mais próximos.

Para determinar esses vizinhos de forma eficiente, o conjunto de dados de entrada é geralmente dividido em pedaços menores, usando técnicas de decomposição espaciais, tal como *octrees* ou *kd-trees* e, de seguida as pesquisas do ponto mais próximo são efetuadas nesse espaço.

Dependendo da aplicação, pode optar-se pela determinação de um número fixo de pontos na vizinhança de *p*, ou de todos os pontos que se encontram dentro de uma esfera de raio *r* centrado em *p*.

5.1.3 **Pontos-chave**

A biblioteca do módulo de pontos-chave contém implementações de algoritmos de duas nuvens de pontos de deteção de pontos-chave. Estes pontos-chave, também definidos como pontos de interesse, são os pontos numa imagem ou de uma nuvem de pontos que estão estáveis e podem ser identificados utilizando um critério de deteção bem definido.

Normalmente, o número de pontos de interesse numa nuvem de pontos será muito menor que o número total de pontos existentes na nuvem.

5.1.4 **Registros**

A técnica de registro é normalmente utilizada na combinação de vários conjuntos de dados num modelo consistente global. A ideia é identificar pontos correspondentes entre os conjuntos de dados e encontrar uma transformação que minimize a distância (erro de alinhamento) entre outros pontos correspondentes.

Este processo é repetido uma vez que a procura de correspondência é influenciada pela posição relativa e a orientação dos conjuntos de dados. Uma vez que os erros de alinhamento forem menores que um determinado valor, o registro é considerado completo.

A biblioteca de registro contém uma variedade de algoritmos de registro de nuvens de pontos de ambos os conjuntos de dados, organizados e desorganizados.

5.1.5 ***KD-tree***

O módulo de *kd-tree* de PCL fornece estrutura de dados *kd-tree*, utilizando *FLANN*, que permite a rápida pesquisa do vizinho mais próximo do ponto *p*.

Kd-tree é uma estrutura de dados de partição de espaço que armazena um conjunto de pontos k-dimensionais numa estrutura em *tree* que permite pesquisas de gamas eficientes e pesquisas do vizinho mais próximo.

5.1.6 ***Octree***

A biblioteca de *octree* fornece métodos eficazes para a criação de uma estrutura de dados hierárquica a partir de uma nuvem de pontos. Permite partições espaciais, diminuição da resolução e operações de pesquisa para um conjunto de dados de pontos.

Cada nó *octree* ou não tem "filhos" ou tem 8 "filhos". O nó raiz descreve uma caixa cúbica delimitadora que engloba todos os pontos, em todos os níveis da árvore este espaço fica subdividido por um fator de 2 o que resulta num aumento da resolução do voxel.

A implementação *octree* fornece eficientes rotinas de pesquiza para o vizinho mais próximo como "vizinhos dentro da pesquiza voxel", "pesquisa do vizinho mais próximo" e "vizinhos num raio de pesquisa". Ajusta-se automaticamente a sua dimensão para o conjunto de dados de pontos.

Um conjunto de folhas de classes de nós fornece funcionalidades adicionais. Assim como "ocupação" espacial e verificações de "densidade de pontos por voxel". Funções de *serialization* e *deserialization* permite codificar em formato binário a estrutura *octree.*

5.1.7 **Segmentação**

O módulo de segmentação apresenta algoritmos para segmentar uma nuvem de pontos em grupos distintos. Estes algoritmos são os mais apropriados para o

processamento de uma nuvem de pontos, que por sua vez, é composta por um certo número de regiões isoladas espacialmente.

Nestes casos o agrupamento – *clustering* – é usado frequentemente a fim de poder fragmentar a nuvem nas zonas onde o processamento independente não pode ser efetuado.

5.1.8 ***Sample_consensus***

A biblioteca de *sample_consensus* em PCL possui métodos como o RANSAC e modelos como planos e cilindros. Estes podem ser combinados a fim de poder detetar modelos específicos e os seus parâmetros numa nuvem de pontos.

Alguns dos modelos implementados na biblioteca incluem: esferas e cilindros, planos e linhas. Os planos são frequentemente aplicados às tarefas de deteção de superfícies interiores comuns, assim como, paredes, tampos de mesa, entre outros. Outros modelos podem ser utilizados na deteção de segmentos de objeto com estruturas geométricas comuns assim como, a utilização do modelo de cilindro para uma caneca.

5.1.9 **Superfícies**

O módulo de superfícies lida com a reconstrução de superfícies originais de digitalizações 3D. A suavização e a reamostragem podem ser importantes no caso de a nuvem apresentar ruído ou se ela é composta por várias digitalizações que não estão alinhadas perfeitamente.

A complexidade da estimativa da superfície pode ser ajustada e podem ser também estimadas as normais caso necessário. *Meshing* é uma maneira para criar uma superfície de pontos e, atualmente existem dois algoritmos que são fornecidos: uma rápida triangulação dos pontos originais e um lento *meshing* que faz o alisamento e o preenchimento dos “buracos”.

A criação de *convex* ou *concave hull* é útil, por exemplo, quando existe a necessidade de uma representação simplificada da superfície ou, quando os limites têm de ser extraídos.

5.1.10 ***Range_image***

A biblioteca de *range_image* tem à disposição duas classes para representar e trabalhar com imagens de profundidade. Uma imagem de profundidade – *range_image* – é uma imagem de pixéis, cujos valores representam uma distância ou profundidade a partir da origem do sensor.

As imagens de profundidade têm geralmente uma representação 3D e, muitas vezes são geradas por câmaras stereo ou por câmaras time-of-flight. Através do conhecimento dos parâmetros da calibração da câmara, uma imagem de profundidade pode ser convertida numa nuvem de pontos.

5.1.11 ***IO***

O módulo *IO* possui classes e funções para a leitura e escrita de ficheiros de dados de nuvens de pontos (PCD), assim como a captura de nuvens de pontos a partir de uma variedade de dispositivos de deteção.

5.1.12 **Visualização**

O módulo de visualização de PCL foi construído a fim de permitir a visualização de algoritmos que atuam nos dados das nuvens de pontos 3D.

De forma semelhante ao OpenCV, na exibição de imagens 2D e no desenho de formas básicas em 2D no ecrã, a biblioteca fornece métodos de processamento e definição de propriedades visuais (cores, tamanhos de pontos, opacidade…) para qualquer conjunto de dados de nuvem de pontos em *pcl::PointCloud<T> format.*

Fornece métodos básicos para o desenho de formas 3D no ecrã (como esferas, cilindros, polígonos…) provenientes de conjuntos de pontos ou de equações

paramétricas; faculta um módulo de visualização de histograma (PCL Histogram Visualizer) para *plots* 2D.

Esta biblioteca dispõe também de um módulo de visualização de imagens de profundidade e, de um grande número de manipuladores de cores e geometria para *pcl::PointCloud<T> datasets* [13].

5.2 Algoritmos analisados

Uma vez que o código deste projeto é realizado utilizando ROS, foram passados alguns algoritmos de PCL a PCL_ROS, a fim de poder exercitar e facilitar o manuseamento e a compreensão posterior na conceção de um novo código que será aplicado ao robô TurtleBot.

Os algoritmos abordados tiveram também uma especial atenção, quanto às suas funções desempenhadas numa nuvem de pontos.

Foram analisados, do módulo de filtros, os algoritmos *Pass Through*, *Voxel Grid* e *Stastistical Outlier Removal*, do módulo de segmentação foi analisado o algoritmo *Plane Model Segmentation.*

5.2.1 ***Pass Through***

O algoritmo deste filtro permite uma filtragem da nuvem de pontos ao longo de uma dimensão especificada, ou seja, dos valores que estão fora ou dentro do intervalo determinado pelo usuário [15].

Pass Through percorre então toda a entrada da nuvem de pontos de uma vez, com base nas restrições, filtrando automaticamente os pontos não-finitos e os pontos fora do intervalo especificado pelo *setFilterLimits (),* que se aplica apenas ao campo especificado pelo *setFilterFieldName ()* [16].

A figura 17 apresenta em Rviz, a entrada da nuvem de pontos efetuada pelo sensor Kinect à esquerda e, o resultado após a utilização deste filtro, testado em ROS, à direita.

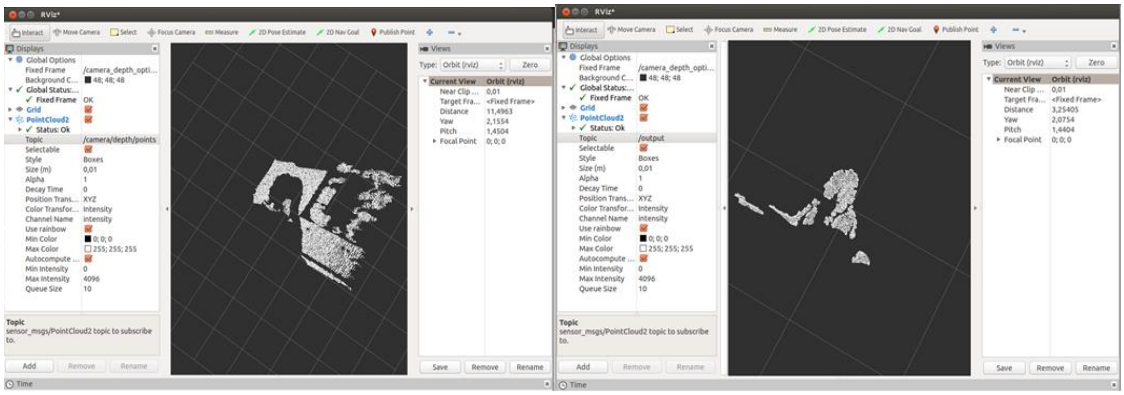


Figura 17: Input e output da nuvem de pontos após a filtragem com *Pass Through*

Através da Figura 17, é claramente possível observar a filtragem efetuada, onde no resultado observa-se a eliminação dos pontos não pretendidos, deixando na nuvem apenas os pontos que, neste caso, correspondem a uma figura humana.

5.2.2 ***Voxel Grid***

O algoritmo do filtro Voxel Grid de PCL permite diminuir a resolução, ou seja, reduz o número de pontos nos dados de pontos utilizando uma "rede".

Este filtro cria uma "rede" 3D local sobre o ponto na nuvem de pontos de entrada (podemos imaginar essa "rede" como um conjunto de pequenas caixas 3D no espaço) e, em cada voxel, todos os pontos que la estejam presentes, serão aproximados ao seu *centroid*, concretizando a redução da resolução [17].

A Figura 18 apresenta em Rviz a nuvem de pontos de entrada à esquerda e a de saída à direita.

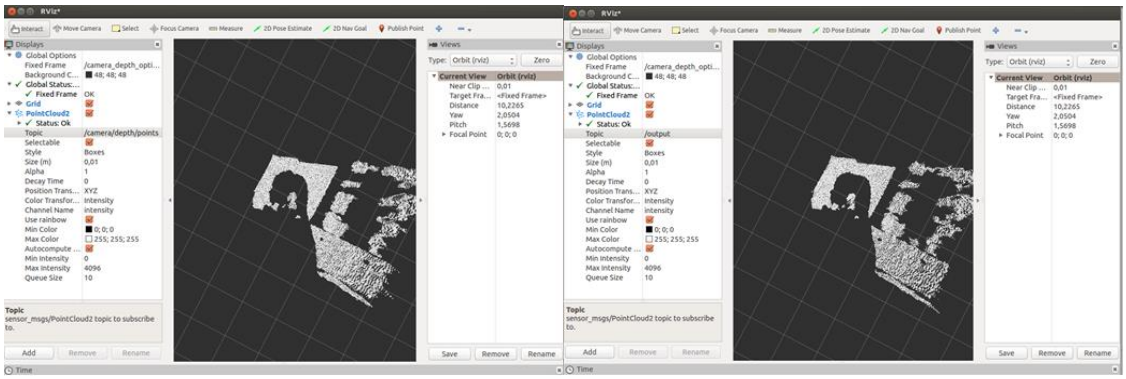


Figura 18: Input e output da nuvem de pontos após a filtragem com Voxel Grid.

Na figura 18 não é possível esclarecer através da visualização, a diminuição da resolução, contudo aquando da observação real dos resultados, há uma perceção notória no que diz respeito ao tempo de resposta.

Tendo uma resolução menor, implica que exista uma menor quantidade de informação a ser processada, consequentemente, o tempo de resposta será menor o que será favorável, por exemplo no tempo de reação de um robô perante um obstáculo.

*5.2.3 **Stastistical Outlier Removal***

O algoritmo de *Stastistical Outlier Removal* tem a capacidade de remoção de, por exemplo, valores discrepantes dos dados de uma nuvem de pontos.

A digitalização da nuvem, normalmente gera um conjunto de dados de nuvem de pontos com uma variedade de densidades de pontos.

Esses erros de medição conduzem a pontos indesejados dispersos que como consequência corrompem os resultados, complicando a estimativa das características da nuvem de pontos local, tal como as normais da superfície ou a mudança de curvatura, conduzindo a valores errados que, por sua vez, podem causar falhas no registo da nuvem.

Estas irregularidades podem ser resolvidas através de uma análise estatística sobre a vizinhança de cada ponto e, eliminando o que não corresponde aos critérios.

Esta eliminação tem como base o cálculo da distribuição das distâncias dos pontos aos seus vizinhos, no conjunto de dados de entrada.

Para cada ponto é calculada a distância média entre o ponto e os seus vizinhos e, assumindo um resultado de uma distribuição de Gauss, com uma média e um desvio padrão, todos os pontos em que a média das distâncias está fora dos intervalos definidos, serão eliminados [18].

A Figura 19 apresenta em Rviz a nuvem de pontos de entrada à esquerda e a de saída, com o filtro *Stastistical Outlier Removal* de PCL, à direita.

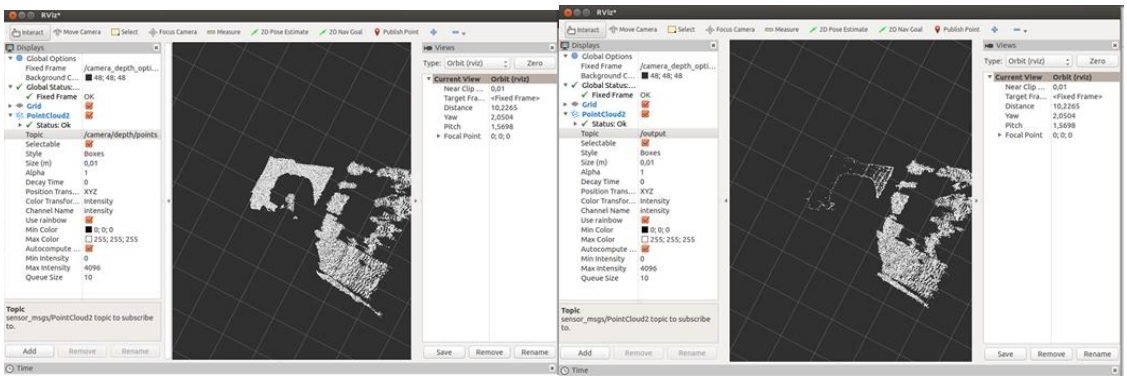


Figura 19: Input e output da nuvem de pontos após a filtragem com *Stastistical Outlier Removal*.

Na figura 19 é possível a visualização da aplicação desse filtro, onde são removidos os pontos não desejáveis, simplificando a nuvem de pontos substancialmente.

5.2.4 ***Plane Model Segmentation***

Este algoritmo permite a segmentação plana de um conjunto de pontos, ou seja, vai permitir encontrar todos os pontos dentro da nuvem de pontos que estejam numa posição plana.

Como do resultado desta segmentação resultam apenas índices, não é possível observar uma nuvem de pontos em Rviz, apenas se pode analisar estes índices através do terminal.

Este método foi utilizado no código deste projeto para obter os índices das superfícies planas e para depois elimina-los, ou seja para permitir fazer o inverso e detetar apenas todas as superfícies não planas.

Na realização do código deste projeto foi ainda utilizado, da biblioteca de segmentação, o algoritmo *Euclidean Cluster Extraction.*

5.2.5 ***Euclidean Cluster Extraction***

O método *Euclidean Cluster Extraction é* utilizado como a classe *pcl::EuclideanClusterExtraction.*

O método de *clustering* permite obter através de grandes conjuntos de dados, grupos de conjuntos menores de dados semelhantes, ou seja, divide uma nuvem de pontos desorganizados em partes menores para que o tempo de processamento geral da nuvem seja reduzido. A figura ilustra esse agrupamento de dados semelhantes.

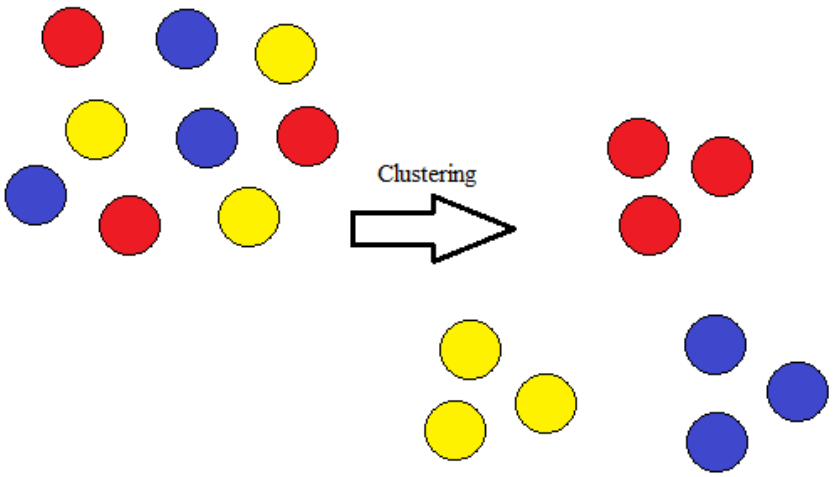


Figura 20: Ilustração do processo de *clustering.*

Um processo de *clustering*, pode ser efetuado através da utilização de uma grelha 3D de subdivisão do espaço, utilizando caixas de largura fixa ou, uma estrutura de dados *octree*, geralmente.

Esta representação particular torna-se rápida de construir e é útil para situações onde é necessário uma representação volumétrica do espaço ou, os dados resultantes de cada caixa 3D, (ou folha *octree*), possam ser aproximados a uma estrutura diferente.

No geral, pode utilizar-se a informação dos vizinhos mais próximos e implementar a técnica de *clustering* que é semelhante a um algoritmo de preenchimento [19].

No capítulo 6 será apresentado o desenvolvimento deste trabalho.

6. Desenvolvimento do Projeto

Neste capítulo é descrito todo o processo efetuado para a concretização deste projeto, referindo, com uma breve caracterização, o material utilizado, o processo para a inicialização de ROS e do robô Turtlebot. É descrito o programa e apresentado os resultados finais.

Como mencionado anteriormente, neste projeto foi utilizado um robô TurtleBot, similar ao aspirador iRobot Roomba, que tem na sua constituição um robô iRobot Create na sua base móvel, um sensor de captação de imagens, o Microsoft Kinect, um laptop ASUS 1215N e uma estrutura de suporte fixada na base. Durante o desenvolvimento do projeto o uso desse laptop foi alternado com um SONY VAIO SVE151E11M, contudo o objetivo final concentra-se com a utilização deste último.

A Figura 21 exemplifica o setup mencionado anteriormente.

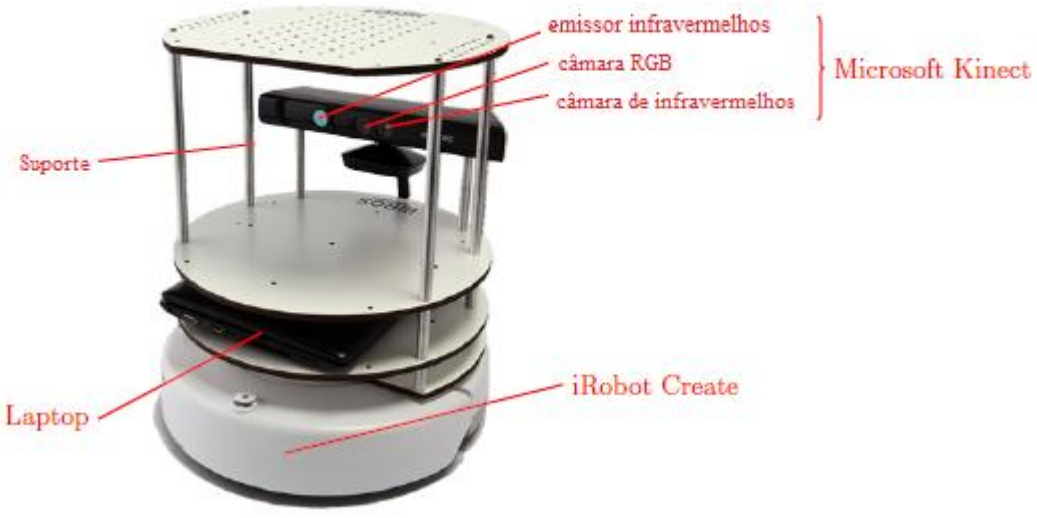


Figura 21: Constituição do robô TurtleBot

O iRobot Create está equipado com uma série de sensores integrados, odómetro, detetores de queda, de paredes e de choques. Possuí uma placa de alimentação ligada ao porto DB25 do iRobot Create (que contém um giroscópico de um eixo), para alimentar o Kinect. Com exceção do laptop, todo o conjunto é alimentado com uma bateria de níquel-hidreto metálico (Ni-MH) de 3000mAh. A comunicação do iRobot Create com o laptop é efetuada através de um adaptador serie – USB [20].

O laptop é visto como o cérebro do robô uma vez que se encarrega das comunicações, do processamento de sinais dos sensores e da execução de programas.

O software usado no laptop ASUS 1215N está equipado com o sistema operativo Ubuntu versão 12.04 e inclui ROS na versão Fuerte; o software usado no laptop SONY VAIO SVE151E11M está equipado com o sistema operativo Ubuntu 12.10 e inclui ROS na versão Groovy.

Durante o decorrer deste trabalho foi simultaneamente desenvolvido um calendário semanal, sobre o trabalho que estava a ser realizado, na página wiki do grupo de robótica da Universidade de León, Espanha [21].

6.1 Inicialização de ROS

Para iniciar ROS é necessário um computador com o sistema operativo Ubuntu, uma vez que os outros sistemas operativos ainda estão em fase experimental, como referido anteriormente.

Procedeu-se então à instalação do Ubuntu 12.10 (Quantal), acedendo ao terminal do computador.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu quantal main" > /etc/apt/sources.list.d/ros-latest.list'
```

Para verificar se o software descarregado está autorizado, é necessário adicionar as chaves do repositório executando o comando que se segue:

```
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

A fim de concluir a instalação de ROS, resta executar a sequência de comandos seguinte:

```
sudo apt-get update
sudo apt-get install ros-groovy-desktop-full
```

Antes da utilização de ROS, é necessário iniciar *rosdep*. Este irá facilitar a instalação das dependências para o sistema que se pretenda compilar e, é necessário para executar alguns componentes principais em ROS. Executam-se então os seguintes comandos no terminal:

```
sudo rosdep init
rosdep update
```

Para ser possível executar os comandos de ROS no terminal é necessário adicionar no arquivo de configuração do terminal (~/.bashrc), uma linha de código encarregue pelo arquivo das variáveis de ROS. Para isso executa-se o seguinte comando:

```
echo "source /opt/ros/groovy/setup.bash" >> ~/.bashrc
```

Depois disto, os comandos que se encontram no capítulo 9 (anexos), já podem ser executados.

Um dos comandos executados frequentemente em ROS é o *rosinstall* que é distribuído separadamente e que permite descarregar com mais facilidade pacotes de ROS.

O seguinte comando possibilita essa instalação:

```
sudo apt-get install python-rosinstall
```

É muito importante que a estação de trabalho esteja sincronizada com o computador do robô para evitar possíveis perdas da rede sem fios. Para a instalação do cliente NTP (Network Time Protocol) Chrony executa-se o seguinte comando:

```
sudo apt-get install chrony
```

Para permitir a comunicação entre o computador do robô e o computador da estação de trabalho, usualmente usa-se uma rede Wi-Fi. Em ambos é adicionado, no arquivo de configuração (~/.bashrc), o ROS_MASTER_URI que vai apontar para a direção de IP (Internet Protocol) do PC (Personal Computer) do robô, que por sua vez é o executa o serviço de master. Para isto, no terminal coloca-se o seguinte comando, substituindo "IP_OF_TURTLEBOT" pela direção IP do PC do robô:

```
echo "export ROS_MASTER_URI=http://IP_OF_TURTLEBOT:11311" >> ~/.bashrc
```

Também é necessário adicionar nos dois PCs um nome de equipamento, e para isso, seguindo a mesma ideia descrita acima, no terminal utiliza-se o seguinte comando:

```
echo "export ROS_HOSTNAME=IP_OF_TURTLEBOT" >> ~/.bashrc
```

No terminal do computador da estação de trabalho deve executar-se o comando seguinte, substituindo "IP_OF_WORKSTATION" pela IP correspondente a este PC:

```
echo "export ROS_HOSTNAME=IP_OF_WORKSTATION" >> ~/.bashrc
```

6.1.1 **Criação de um espaço de trabalho**

Para que o utilizador trabalhe mais comodamente, cria-se um diretório próprio para stacks e packages.

ROS tem já por defeito, como diretório /opt/ros/groovy/stacks onde se podem criar novos stacks e packages contudo, torna-se mais cómodo criar um diretório particular mais acessível e organizado. Para isso podemos atribuir-lhe um nome à escolha no local que se pretenda [22].

Inicia-se criando o diretório, por exemplo, na pasta pessoal:

```
mkdir ~/groovy_workspace
```

Neste diretório, cria-se um *package*, no qual irão depender outros *packages* de ROS. As dependências são criadas, dependendo do tipo de linguagem usado no desenvolvimento dos programas e, do tipo de mensagem que se envie ou receba.

No caso deste trabalho foram adicionadas as dependências, seguidamente à criação do *package*, que se encontram na seguinte sequência de comandos:

```
cd ~/groovy_workspace
roscreate-pkg NOME_PACKAGE pcl pcl_ros roscpp sensor_msgs
```

No caso em que se pretenda adicionar dependências a um *package* já criado deve editar-se o arquivo "manifest.xml" que se encontra dentro da pasta do *package* criado.

Por fim deve-se compilar para que as dependências, ou qualquer modificação, sejam validadas, para isto utiliza-se o seguinte comando:

```
rosmake NOME_PACKAGE
```

6.2 Ativação do Robô TurtleBot

Para iniciar a utilização do robô é necessário conectar os cabos de USB para a comunicação do robô ao PC robô, ligar o iRobot Create e o PC robô.

Depois de já configurada a rede, pode-se efetuar as operações de trabalho no PC robô, contudo, para uma maior comodidade do utilizador, o PC robô é conectado ao PC de estação de trabalho através de SSH (Secure Shell).

Efetua-se o seguinte comando e, substitui-se "IP_OF_TURTLEBOT" pela direção IP do PC robô.

```
ssh turtlebot@IP_OF_TURTLEBOT
```

Após a execução do comando anterior no terminal do PC de estação de trabalho é pedida a password do PC robô e, concluído isto, é efetuada a conexão.

Para verificar o estado de serviço do robô escreve-se o seguinte comando:

```
sudo service turtlebot status
```

Por defeito, o robô já estará ligado uma vez que se liga o PC robô, contudo é possível desligar e ligar ou reiniciar através dos seguintes comandos, respetivamente:

```
sudo service turtlebot stop
sudo service turtlebot start
sudo service turtlebot restart
```

Num novo terminal do PC de estação de trabalho liga-se a câmara Kinect através do seguinte comando:

```
Roslaunch openni_launch openni.launch
```

No PC robô, neste caso, inicia-se então, o dashboard, através do comando a seguir, que se define por uma janela com botões que permitem controlar o estado do robô.

```
rosrun turtlebot_dashboard
```

A Figura 22 representa a janela dashboard.

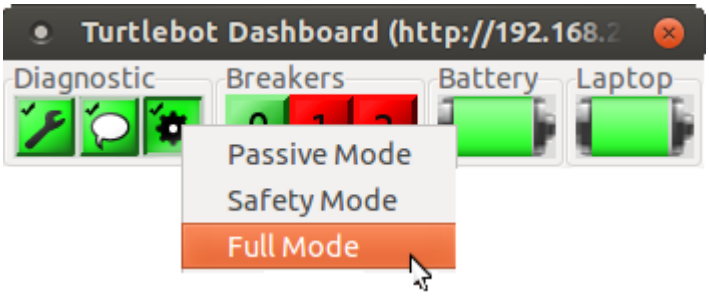


Figura 22: Janela de dashboard

Como é visível na Figura 22, turtlebot dashboard está dividido em 3 secções:

- Diagnostic, onde se obtém informações acerca do diagnóstico, mensagens e o modo de operação do sistema;
- Breakers, onde se verifica os interruptores dos circuitos das três saídas digitais do iRobot Create;
- Battery e Laptop, onde fornece informações acerca dos níveis de bateria do iRobot Create e do PC robô.

Quando está em correto funcionamento apresenta-se com cor verde, caso contrario apareceria com cor vermelho. Para que o robô possa operar, deve colocar-se em Full Mode, nas outras duas opções o robô não opera pois Passive Mode corresponde ao momento em que a bateria está a carregar e, Safety Mode corresponde ao modo de espera.

Por defeito, o interruptor da entrada 0 é acionado, contudo, em caso contrario, deve ser ativado com um clique no botão que lhe corresponde.

6.3 Programa desenvolvido

O programa desenvolvido para a concretização deste projeto foi criado em linguagem C++ e os valores inseridos encontram-se em unidades SI.

Foram utilizados algoritmos em pcl_ros, pois, utilizando os algoritmos de PCL juntamente com as bibliotecas de ROS tornou a mais simples a realização deste código.

O objetivo do projeto debruça-se sobre os movimentos reativos do robô TurtleBot, utilizando a câmara Kinect. Esses movimentos reativos caracterizam-se com o facto de o robô ter capacidade de ser desviar sempre que se depare com um objeto e, cada vez que aviste um humano, se desloque até uma distância definida, parando durante um determinado tempo e emitindo um sinal.

O fluxograma da Figura 23 representa esquematicamente este processo.

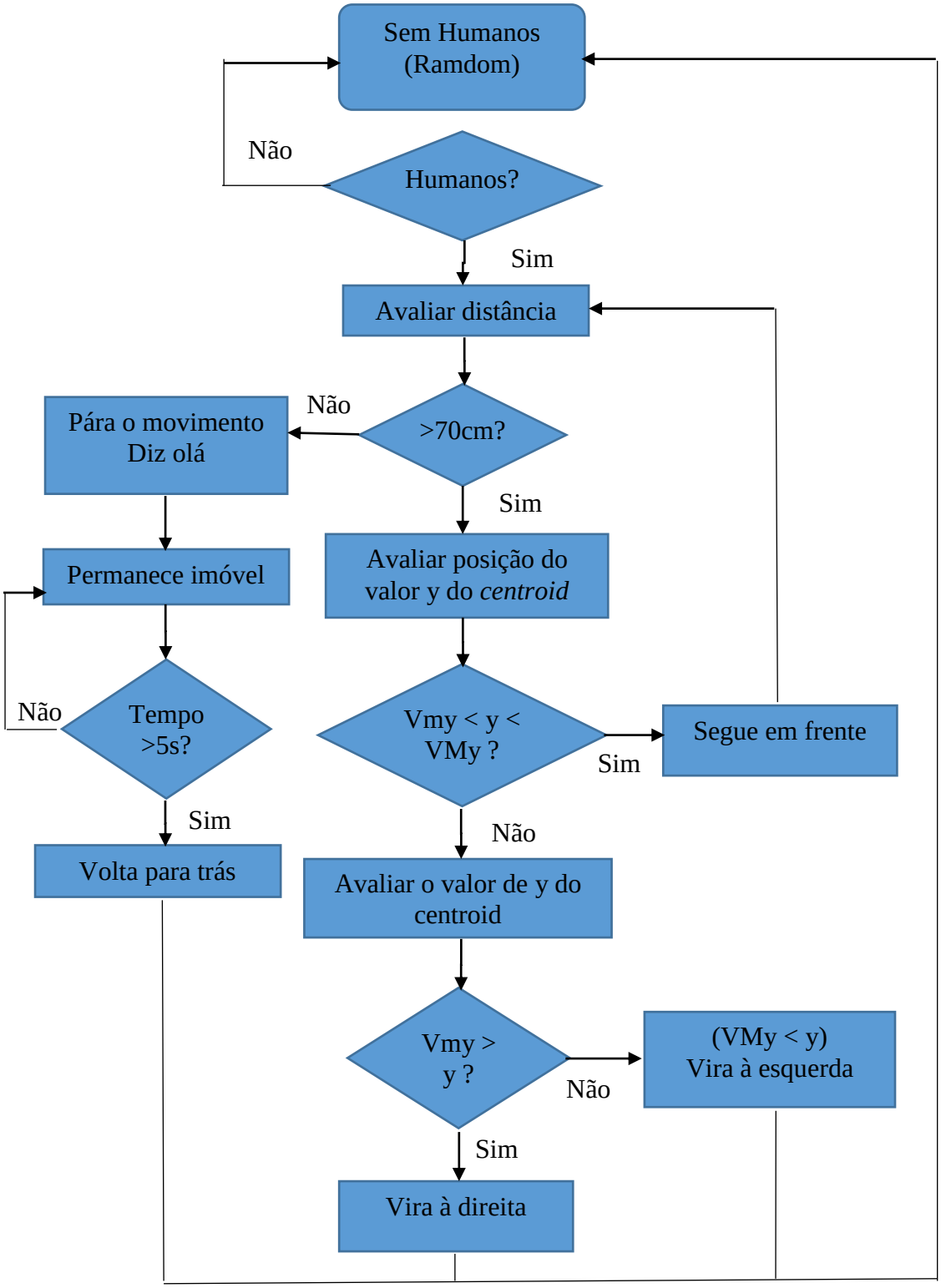


Figura 23: Fluxograma do processo de captação de humanos

Relativamente ao fluxograma anterior é de referir que VMy corresponde ao valor máximo de y e Vmy corresponde ao valor mínimo de y.

6.3.1 **Tortue**

Tortue é a parte do programa responsável pelo movimento aleatório do robô e, que permite a deteção dos objetos a uma distância máxima de 70cm,

Desvia-se para a esquerda ou para a direita se o objeto se encontrar mais à direita ou mais à esquerda respetivamente e, seguindo em frente sempre que não encontrar objetos. A Figura 24 apresenta num fluxograma o funcionamento do processo de *random*, descrito.

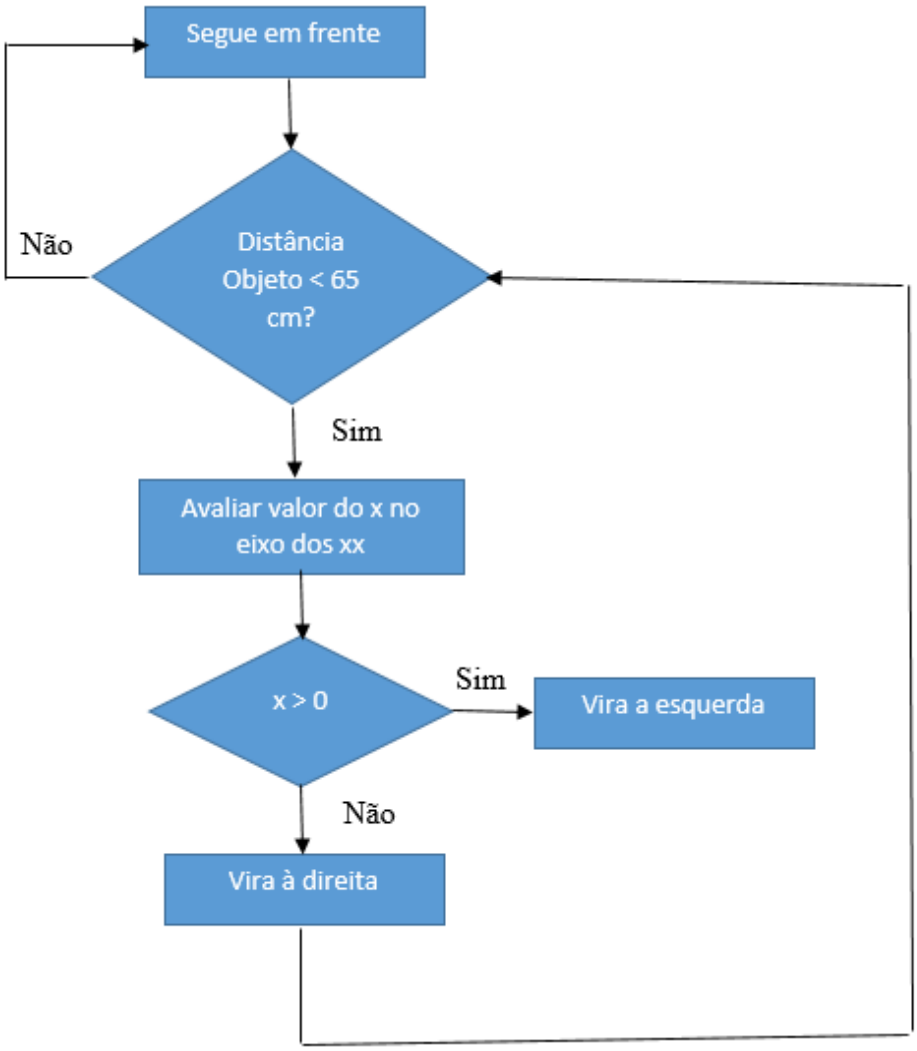


Figura 24: Fluxograma do processo tortue.

Uma das particularidades deste algoritmo de navegação está nos seus eixos de orientação, que se dispõe conforme ilustrado na Figura 25.

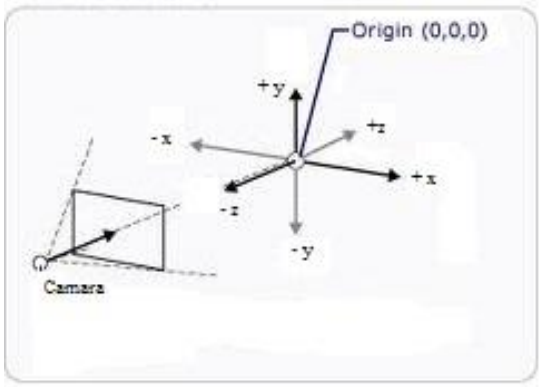


Figura 25: Eixos ortogonais correspondentes ao movimento do robô.

O eixo dos xx corresponde à direção esquerda/direita, o eixo dos yy, que neste trabalho não é utilizado, corresponde à altura, por sua vez, o eixo dos zz corresponde à distância entre a câmara Kinect e o ponto *centroid* da nuvem de pontos correspondente neste caso ao objeto.

O referencial tem como origem o centroid do objeto e, desta forma, a distância z nunca toma valores negativos.

6.3.2 **Kinect_tracker**

O Kinect_tracker é a outra parte do código e corresponde à deteção de humanos, bem como ao deslocamento do robô até aos mesmos.

Para a deteção utiliza-se o sistema de referências mais habitual, onde o z corresponde à altura e o x e o y ao comprimento e largura, como está ilustrado na figura 26.

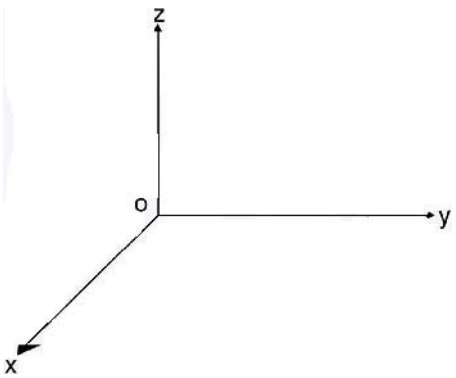


Figura 26: Eixos ortogonais correspondentes à deteção de humanos

Esta parte do código utiliza o filtro *voxel grid*, o algoritmo *planel model segmentation*, o método *euclidean clustering* e ainda um *marker* que permitirá reconhecer o humano dentro das restrições definidas, numa nuvem de pontos ou numa imagem da câmara.

Com o algoritmo *planel model segmentation* pretende-se obter todos os índices dos pontos dentro, da nuvem de pontos, que estejam numa posição plana para posteriormente utiliza-los numa função, a fim de os extrair e deixar apenas visíveis regiões não planas.

Com o método *Euclidean Cluster Extraction* é pretendido que a segmentação da nuvem de pontos faça grupos de conjuntos de dados semelhantes como referido no capítulo 5.

Depois de processar a nuvem de pontos de entrada de forma a eliminar a maior parte da informação não desejada, foram definidas restrições de altura com mínimo de 1,40m e máximo de 1,90m, e de largura com o mínimo de 0,20m e 0,85m e com as linhas de código apresentadas na Figura 27, implementou-se a restrição.

```
if(((max[2]-min[2]>MIN_PERSON_HEIGHT)&&(max[2]-min[2]<MAX_PERSON_HEIGHT)) ||
  ((max[1]-min[1]<MAX_PERSON_WIDTH)||(max[0]-min[0]<MAX_PERSON_WIDTH)) ||
  ((max[1]-min[1]>MIN_PERSON_WIDTH)||(max[0]-min[0]>MIN_PERSON_WIDTH)))
```

Figura 27: Extrato de código correspondente às restrições para captura do humano.

Depois da deteção é devolvido o valor do *centroid* em coordenadas XYZ.

Sendo o objetivo, a deslocação do robô até ao humano de forma a permanecer o mais centrado possível perante este, foi utilizado o valor de y, que corresponde à largura e, para a câmara direita/esquerda, a fim de centrar o robô aquando a sua deslocação.

É então adicionado movimento para que, enquanto o humano estiver a mais de 70cm o robô se desloque, verificando constantemente se o *centroid* do humano se encontra dentro dos limites da região definida pelo usuário.

A Figura 28 representa um extrato de código em que se a distância for superior a 70cm e o humano estiver fora do limite, mas ainda detetado pela câmara e, mais próximo do limite superior, vira-se para a esquerda 15º.

```
else if (((min_dist[0] >= 0) && (min_dist[2]>=0.70)) && (centroid[1] > DIST_MAX_Y))
{
        printf("Turn Left tom\n");
        cmdvel.angular.z=0.2617;//15º
        cmdvel.linear.x=0;
}
```

Figura 28: Extrato de código representativo de uma restrição, seguida de um comando para virar à esquerda.

Quando o humano se encontrar a uma distância menor ou igual a 70cm o robô pára, emite um sinal, neste caso “olá” e entra em modo *sleep* durante 5s. Após os 5s o robô volta para trás dando uma volta de 180º e entra em modo *random* até encontrar novamente um humano. O extrato de código da Figura 29 apresenta o que foi acima descrito.

```
    if (((min_dist[0] >= 0) && (min_dist[2]<0.70))&& ((DIST_MIN_Y < centroid[1]) && (centroid[1] < DIST_MAX_Y)))
{
        cmdvel.angular.z=0;
        cmdvel.linear.x=0;
    //vel_pub.publish(geometry_msgs::Twist());
    system("echo hola  | festival --tts");
    ros::Duration(5.0).sleep();

        printf("Turn back\n");
        cmdvel.angular.z=3.14;// 180º
        cmdvel.linear.x=0;
}
```

Figura 29: Extrato de código representativo de uma restrição, seguida de uma série de comandos

6.3.3 **Subscrições/publicações**

São efetuadas duas subscrições, uma correspondente à deteção de pessoas e às reações perante essa deteção (Cloud_cb), e outra correspondente ao modo *random*, ou seja ao tortue (callback).

São feitas seis publicações assim como se pode observar na Figura 30, que podem ser selecionadas em Rviz de modo a observar as suas saídas.

```
ros::Subscriber subA = nh.subscribe ("/camera/depth_registered/points", 1, cloud_cb);
ros::Subscriber subB = nh.subscribe<Pointi>("/camera/depth/points", 1, callback);

 // Create a ROS publisher for the output point cloud
 pub = nh.advertise<sensor_msgs::PointCloud2> ("base_link/filtered_cloud", 1);
 // Create a ROS publisher for the detections
detections_pub = nh.advertise<geometry_msgs::PointStamped>("kinect_tracker/detections",1);
 // Create a ROS publisher for the output segmented plane
pub_plane = nh.advertise<pcl::ModelCoefficients> ("base_link/plane", 1);
 // Create a ROS publisher for the clusters
   pub_cluster = nh.advertise<sensor_msgs::PointCloud2> ("base_link/cluster_cloud", 1);
 // Create a ROS publisher for the cluster markers
 cluster_marker = nh.advertise<visualization_msgs::Marker> ("base_link/cluster_marker", 1);

 vel_pub = nh1.advertise<geometry_msgs::Twist>("cmd_vel",1);
```

Figura 30: Extrato de código correspondente às publicações/subscrições

Para observar as saídas das publicações no terminal no PC de estação de trabalho escreve-se o seguinte comando, substituindo "Publisher" pelo nome atribuído à publicação.

```
rostopic echo Publisher
```

6.4 Resultados

Para a obtenção dos resultados acerca a captação de humanos recorreu-se à ferramenta de visualização Rviz, acionada a partir do terminal do PC de estação de trabalho, com o seguinte comando:

```
rosrun rviz rviz
```

Depois de selecionados devidamente os tópicos, foi capturada uma imagem onde se pode verificar a captação de um humano como se verifica na figura31:

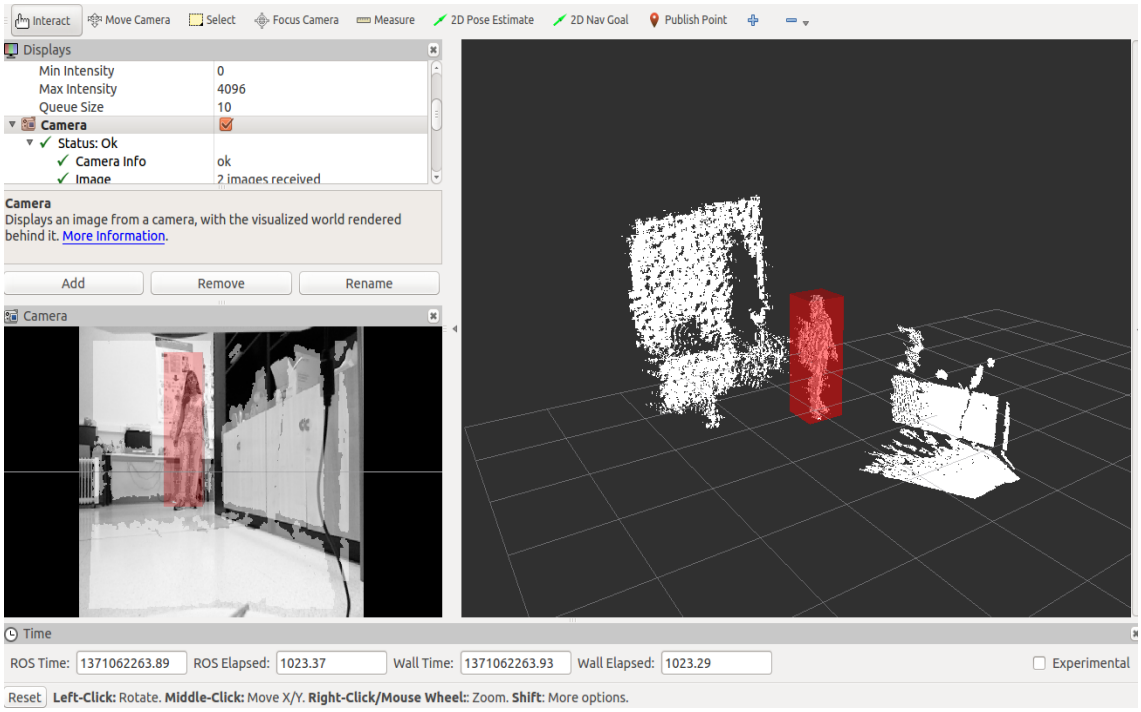


Figura 31: imagem com a captura do humano e o meio envolvente

A imagem observada na Figura 31 apresenta, para além do que se pretende demonstrar, o meio envolvente de forma a poder situar melhor o humano. É uma

informação adicional que se adiciona através da implementação de mais um nó de PointCloud2, onde se escolhe o tópico de /camera/depth/points.

Contudo, na Figura 32 pode observar-se apenas o resultado obtido quando se executa o programa, apenas a imagem do humano aparece.

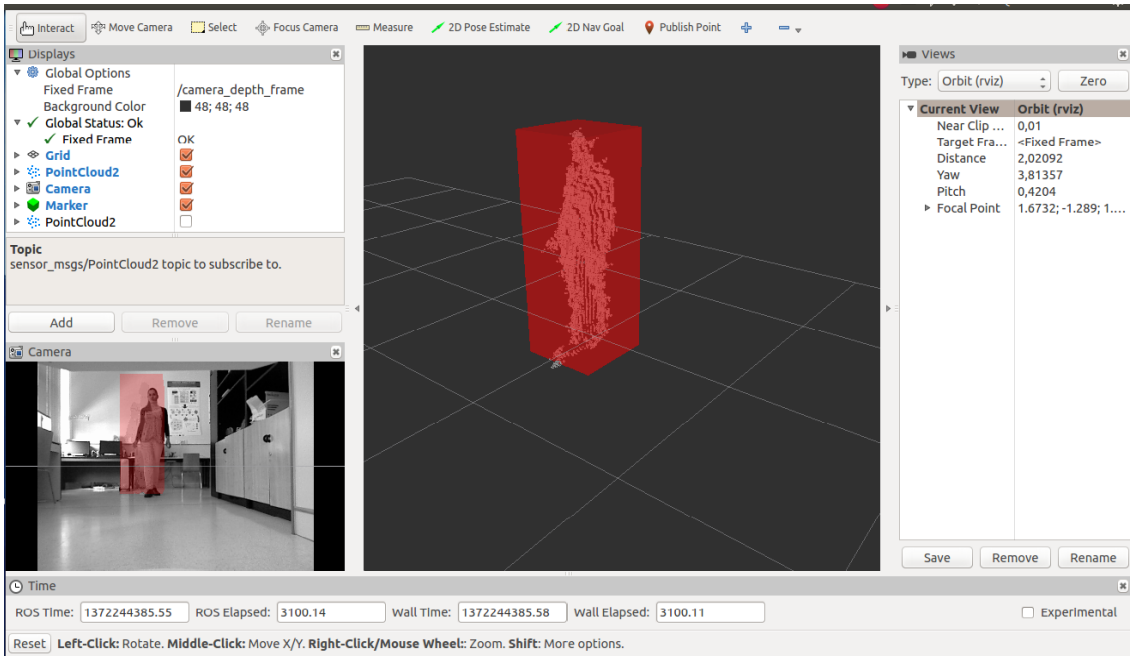


Figura 32: Captação do humano

6.4.1 **Falsos-positivos**

Existem falsos-positivos que se devem a vários fatores tais como a luminosidade e o excesso de informação a ser processada.

Devido ao processamento de excesso de informação, o reconhecimento do humano é muito lento, variando entre 10s e 1min aproximadamente, o que faz com que muitas vezes fique com a informação de que esse humano está numa determinada posição enquanto ele já se deslocou a outro lugar.

A figura 33 apresenta um exemplo da situação descrita, apresentando na imagem da câmara apenas a sombra do humano depois de ele ter abandonado aquela posição e apresentando na nuvem de pontos como se ele ali estivesse.

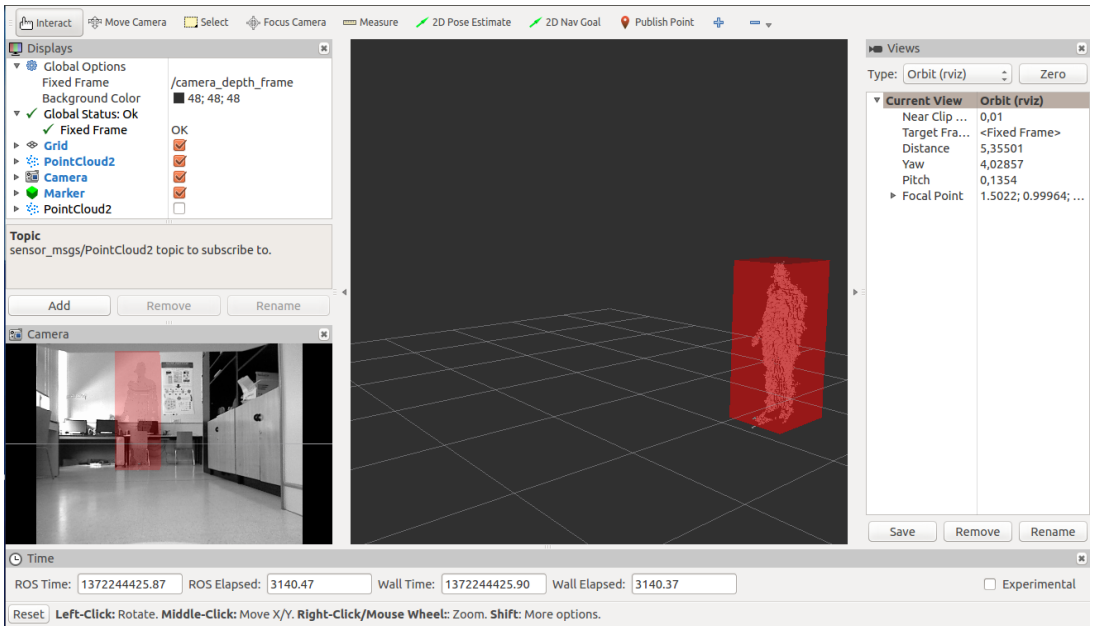


Figura 33: Falso-positivo devido ao atraso do processamento

As condições de luminosidade também têm grande importância no que diz respeito à captação do humano. O humano para poder ser capturado não deve vestir roupa de cor preta, uma vez que a câmara é de infravermelhos.

Um dos contras deste método de captação de humanos é que é capaz de capturar qualquer objeto que se encontre dentro dos limites impostos. Um exemplo disto está representado na Figura 34.

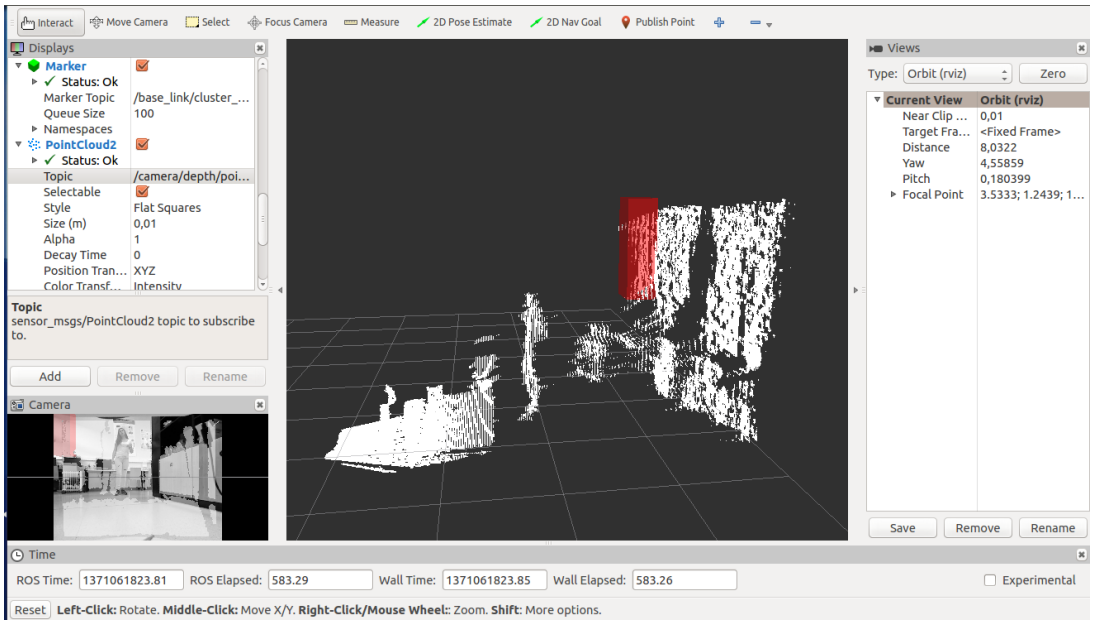


Figura 34: Falso-positivo devido a um objeto com medidas similares às de um humano.

6.4.2 **Resultados no terminal do PC de estação de trabalho**

Ao colocar o robô em marcha com o programa, acompanha-se através do terminal no PC de estacão de trabalho, as instruções seguidas por ele. A Figura 35 apresenta uma captura dessas instruções num determinado instante.

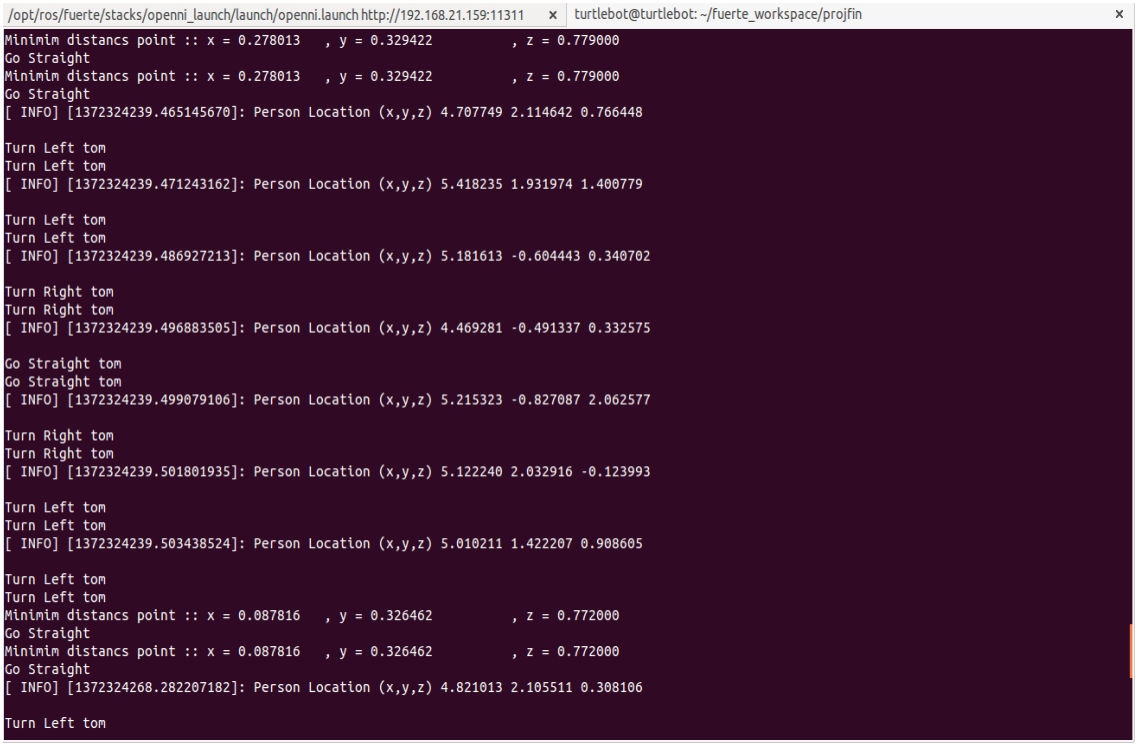


Figura 35: Captação das instruções seguidas pelo robô

Como é possível observar na Figura 35 o robô segue instruções do modo *random* enquanto não tem captação do humano e, sempre que adquire a informação de captação do humano, segue as instruções devidas, tendo em conta a posição que toma.

Foi implementado no código um ciclo *while* na parte das restrições após a deteção do humano, com o objetivo de tornar o movimento do robô mais rápido, pois quando processa as informações de captura, torna-se extremamente lento impedindo a sua deslocação de forma "natural". Contudo ainda não foi suficiente! Toma um movimento com "soluços", impedindo verificar com clareza fisicamente algumas das instruções tomadas.

No capítulo 7 serão descritas as conclusões retiradas com a execução deste projeto e serão apresentadas algumas propostas futuras.

7. Conclusões e Trabalho Futuro

O trabalho desenvolvido neste projeto teve como objetivo a criação de um programa em linguagem C++, utilizando ROS, capaz de movimentar um robô até um humano, desviando-se sempre que algum objeto perturbe o seu trajeto até ao humano.

Para este trabalho foi necessário trabalhar com o sistema operativo Linux, que durante um mês e pouco gerou conflitos com a instalação do Ubuntu paralelamente com o Windows 8.

Depois da instalação de todo o software necessário, foi inicialmente passado algoritmos de PLC a plc_ros com o objetivo de facilitar o posterior manuseamento das bibliotecas utilizadas de ROS e PLC.

Inicializando o programa deste projeto foi adicionado movimento a um programa de reconhecimento de humanos. Um dos problemas detetados foi o sistema de eixos de coordenadas pois, como já referido os eixos de coordenadas para o movimento do robô é diferente dos eixos de coordenadas do programa de deteção de humanos.

Este fato interfere no momento em que pretende determinar a distância a que se encontra o humano.

Foi também adicionado o programa tortue, já mencionado, que funciona perfeitamente.

A tabela 2 apresenta as restantes conclusões positivas e negativas deste projeto.

Tabela 2: Tabela conclusiva com aspetos positivos e negativos.

Conclusões	
Positivas	**Negativas**
É capaz de detetar uma pessoa.	A luminosidade influencia os resultados.
Vai ao encontro da pessoa, mantendo-se frente-a-frente e emitindo um sinal.	Deteta objetos com tamanhos semelhantes a uma pessoa, e transmite a

	informação como sendo uma pessoa – falsos-positivos.
Ao detetar um objeto é capaz de apartar-se.	O tempo de reação é muito elevado devido ao excesso de informação processada que se reflete no movimento aos "soluços" do robô.

Como propostas futuras existem inúmeras contudo aqui são apresentadas as mais relevantes.

A realização de uma expressão analítica relacionando os eixos de coordenadas do sistema de deteção de humanos com os eixos de coordenadas do sistema de movimento com o objetivo de ultrapassar o problema em relação à determinação da distância do humano.

Implementação de uma função que permita reduzir o número de *frame*s a serem processadas, a fim de reduzir o tempo de reação, apesar de perder informação e não se tornar tão fiável, acelerará o processamento e como consequência melhorará o comportamento do robô.

Após o correto funcionamento podem adicionar-se restrições no que diz respeito à altura, ou seja, que impeça o robô de se deslocar por baixo de uma mesa.

8. Referências

[1] Robot Operating System, disponível em URL: http://en.wikipedia.org/wiki/Robot_Operating_System , com acesso a 19/04/2013.

[2] M.Quigley, B.Gerkey, K.Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, A. Ng. ROS: an open-source Robot Operating System. Computer Science Department, Stanford University, Stanford, CA. Willow Garage, Menlo Park, CA. Computer Science Department, University of Southern California.

[3] Robots Using ROS. Disponível em url: http://www.ros.org/wiki/Robots, com acesso a 25/04/2013.

[4] ROS Groovy Galapagos Released. Disponível em url: http://www.ros.org/news/2012/12/ , com acesso a 2/05/2013.

[5] P.J. Moreira da Costa. Operação de Pick-and-Place Adaptativa em Ambientes Pouco Estruturados. Tese de Mestrado Integrado em Engenharia Electrotécnica e de Computadores. 31 de Julho de 2012.

[6] PrimeSense. Disponível em url: http://www.primesense.com/, com acesso a 5/05/2013.

[7] Ipisoft. Disponível em url: http://ipisoft.com/, com acesso a 5/05/2013.

[8] OpenNi. Disponível em url: http://www.openni.org/, com acesso a 10/05/2013.

[9] OUR FULL 3D SENSING SOLUTION. Disponível em url: http://www.primesense.com/solutions/technology/, com acesso a 12/05/2013.

[10] Kinect operation. Disponível em url: http://www.ros.org/wiki/kinect_calibration/technical, com acesso a 16/05/2013.

[11] Stephane Magnenat. Raw depth to meters.

[12] Point Cloud Library. Disponível em url: http://pointclouds.org/, com acesso a 18/05/2013

[13] PCL (Point Cloud Library). Disponível em url: http://en.wikipedia.org/wiki/PCL_(Point_Cloud_Library), com acesso a 18/05/2013

[14] Point Cloud Library. Disponível em url: http://www.willowgarage.com/pages/software/pcl, com acesso a 18/05/2013.

[15] Point Cloud Library, Passtrough. Disponível em url: http://docs.pointclouds.org/trunk/classpcl_1_1_pass_through.html, com acesso a 20/05/2013.

[16] Filtering a PointCloud using a PassThrough filter. Disponível em url: http://pointclouds.org/documentation/tutorials/passthrough.php, com acesso a 20/05/2013.

[17] Downsampling a PointCloud using a VoxelGrid filter. Disponível em url: http://pointclouds.org/documentation/tutorials/voxel_grid.php, com acesso a 21/05/2013.

[18] Removing outliers using a StatisticalOutlierRemoval filter. Disponível em url: http://pointclouds.org/documentation/tutorials/statistical_outlier.php, com acesso a 31/05/2013.

[19] Euclidean Cluster Extraction. Disponível em url: http://www.pointclouds.org/documentation/tutorials/cluster_extraction.php, com acesso a 01/06/2013.

[20] Fernando Casado. Desarrollo de programas de control para el robot turtlebot sobre ROS (Robot Operating System). Escuela de Ingenierías Industrial, Informática y Aeronautica de la Universidad de León. Julio de 2012

[21] Tânia Carrera. Kinect with PCL and ROS. Disponível en url: http://robotica.unileon.es/mediawiki/index.php/Tania-TFM-ROS04, 2013.

[22] Installing and Configuring Your ROS Environment. Disponível em url: http://www.ros.org/wiki/ROS/Tutorials/InstallingandConfiguringROSEnvironment, com acesso a 21/06/2013.

9. Anexos

Neste capítulo são expostos os comandos mais utilizados em ROS e o código em linguagem C++ do programa.

9.1 Comandos mais utilizados em ROS

Comando	Descrição	Modo de utilização
Comandos do sistema de arquivos		
rospack/rosstack	Ferramenta de inspeção de *packages/stacks*	rospack find [package]
roscd	Muda diretórios a um *package* ou *stack*	roscd [package[/subdir]]
rosls	Apresenta o conteúdo de um *package* ou *stack*	rosls [package[/subdir]]
roscreate-pkg	Cria um novo *package* ROS	roscrate-pkg [nome package]
roscreate-stack	Cria um novo *stack* ROS	roscrate-stack [nome stack]
rosdep	Instala as dependências dos *packages* ROS	rosdep install [package]
rosmake	Compila um *package* ROS	rosmake [package]
roswtf	Mostra os erros e avisos de um sistema ROS em funcionamento ou de um arquivo .launch	roswtf ou roswtf [arquivo]
rxdeps	Mostra a estrutura de um *package* e dependências	rxdeps [opções]
Comandos habituais		
roscore	Executa os *nodes master*, *parameter server* e *rosout*, necessários para comunicação entre nós.	roscore
rosmsg/rossrv		
rosmsg show	Mostra os campos na mensagem	rosmsg show [tipo mensagem]
rosmsg users	Busca os arquivos que usam a mensagem	rosmsg users [tipo mensagem]
rosmsg md5	Mostra o md5sum da mensagem	rosmsg md5 [tipo mensagem]

rosmsg package	Apresenta todas as mensagens em um *package*	rosmsg package [package]
rosmsg packages	Apresenta todos os *packages* com mensagens	rosmsg packages
rosrun	Inicia um executável de um *package*	rosrun [package] [executável]
rosnode		
rosnode ping	Comprova a conexão a um nó	rosnode ping [node]
rosnode list	Apresenta os nós ativos	rosnode list
rosnode info	Mostra a informação sobre um nó	rosnode info [node]
rosnode machine	Apresenta os nós ativos numa máquina	rosnode machine [máquina]
rosnode kill	Detém um nó ativo	rosnode kill [node]
roslaunch	Executa um arquivo .launch	roslaunch [package] [arquivo]
rostopic		
rostopic bw	Mostra a largura de banda usada pelo *topic*	rostopic bw [topic]
rostopic echo	Mostra as mensagens do *topic* no ecrã	rostopic echo [topic]
rostopic hz	Mostra a velocidade da publicação do *topic*	rostopic hz [topic]
rostopic list	Apresenta os *topics* ativos	rostopic list
rostopic pub	Publica dados num *topic*	rostopic pub [velocidade] [nome topic] [tipo mensaje] [dados]
rostopic type	Mostra o tipo de *topic*	rostopic type [topic]
rostopic find	Procura *topics* por tipo	rostopic find [tipo mensagem]
rosparam		
rosparam set	Modificar o valor de um parâmentro	rosparam set [parâmetro] [valor]
rosparam get	Obter o valor de um parâmetro	rosparam get [parâmetro]
rosparam load	Cargar parâmetros de um arquivo YAML	rosparam load [arquivo]
rosparam dump	Guardar parâmetros num arquivo YAML	rosparam dump [arquivo]
rosparam delete	Eliminar um parâmetro	rosparam delete [parâmetro]
rosparam list	Apresentar todos os parâmetros	rosparam list
rosservice		
rosservice list	Apresenta os serviços ativos	rosservice list
rosservice node	Mostra o nó que dá um serviço	rosservice node [serviço]
rosservice call	Chama o serviço com os	rosservice cal [serviço]

	argumentos dados	[argumentos]
rosservice args	Apresenta os argumentos do serviço	rosservice args [serviço]
rosservice type	Mostra o tipo de serviço	rosservice type [serviço]
rosservice uri	Mostra o serviço ROSRPC uri	rosservice uri [serviço]
rosservice find	Procura serviços por tipo	rosservice find [tipo serviço]
Comandos de registro		
rosbag		
rosbag record	Guarda o conteúdo de um ou de todos os *topics*	rosbag record [topic] o -a
rosbag play	Reproduz o conteúdo de um arquivo .bag	rosbag play [arquivo]
Comandos gráficos		
rxgraph	Mostra um gráfico com os nós ativos e os *topics* que os conectam	rxgraph
rxplot	Mostra os dados de um ou mais campos dos *topics* graficamente	rxplot [topic/campo1:campo2]
rxbag	Ferramenta de visualização, inspeção e reprodução de arquivos .bag	rxbag [arquivo]
rxconsole	Ferramenta de visualização e filtro de mensagens publicados em rosout	rxconsole
Comandos transformações frames		
tf_echo	Mostra a informação da transformação entre um frame de origem e um frame de destino	rosrun tf tf_echo [frame origem] [frame destino]
view_frames	Ferramenta de geração da árvore completa de transformações coordenadas, num arquivo .pdf	rosrun tf view_frames

9.2 Código em linguagem C++

```
#include <ros/ros.h>
#include <sensor_msgs/PointCloud2.h>
#include <visualization_msgs/Marker.h>
#include <tf/transform_listener.h>
#include <geometry_msgs/PointStamped.h>
// PCL specific includes
#include <pcl/ros/conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/common/common.h>
#include <pcl_ros/transforms.h>
// PCL Filtering
#include <pcl/filters/voxel_grid.h>
// PCL Plane Segmentation
#include <pcl/ModelCoefficients.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/segmentation/extract_clusters.h>
#include <pcl/filters/extract_indices.h>
// PCL Clustering
#include <pcl/features/normal_3d.h>
#include <pcl/kdtree/kdtree.h>
#include <pcl/kdtree/kdtree_flann.h>
// PCL Visualizer
#include <pcl/visualization/pcl_visualizer.h>
#include <boost/thread/thread.hpp>
//Mov
#include <boost/foreach.hpp>
#include <geometry_msgs/Twist.h>
#include <pcl_ros/point_cloud.h>

// Parameters (TODO: Create a yaml file and move params to the param server)
#define MAX_PERSON_HEIGHT 1.90
#define MIN_PERSON_HEIGHT 1.40
#define MAX_PERSON_WIDTH 0.85 //1
#define MIN_PERSON_WIDTH 0.2//0.2
#define MAX_PERSON_CLUSTER_SIZE 20000
#define MIN_PERSON_CLUSTER_SIZE 100
#define PERSON_CLUSTER_TOLERANCE 0.2 //0.1
#define LEAF_SIZE 0.01 //0.01 - Downsampled grid size
#define DIST_MIN_Y -0.5
#define DIST_MAX_Y 0.5
```

```
ros::Publisher pub;
ros::Publisher pub_plane;
ros::Publisher pub_cluster;
ros::Publisher cluster_marker;
ros::Publisher detections_pub;
ros::Publisher vel_pub;

tf::TransformListener *tf_listener;
pcl::ModelCoefficients::Ptr plane_coefficients (new pcl::ModelCoefficients);
geometry_msgs::Twist cmdvel;

float min_dist[3];

// Visualization Marker for Point Cloud Cluster

void publishClusterMarker(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster,
std::string ns ,int id, float r, float g, float b)
{
  // Tranform cloud to base_link
   pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud (new
pcl::PointCloud<pcl::PointXYZ>);
   (*cloud_cluster).header.frame_id = "camera_depth_optical_frame";
 //  tf_listener-
>waitForTransform("openni_depth_optical_frame","base_link",ros::Time::now(),ros::D
uration(3.0));
   pcl_ros::transformPointCloud("base_link",*cloud_cluster,*transformed_cloud,
*tf_listener);

 // transformed_cloud = cloud_cluster;

  Eigen::Vector4f centroid;
  Eigen::Vector4f min;
  Eigen::Vector4f max;

  pcl::compute3DCentroid (*transformed_cloud, centroid);
  pcl::getMinMax3D (*transformed_cloud, min, max);

  uint32_t shape = visualization_msgs::Marker::CUBE;
  visualization_msgs::Marker marker;
  marker.header.frame_id = "base_link";
  marker.header.stamp = ros::Time::now();

  marker.ns = ns;
  marker.id = id;
```

```
  marker.type = shape;
  marker.action = visualization_msgs::Marker::ADD;

  marker.pose.position.x = centroid[0];
  marker.pose.position.y = centroid[1];
  marker.pose.position.z = centroid[2];
  marker.pose.orientation.x = 0.0;
  marker.pose.orientation.y = 0.0;
  marker.pose.orientation.z = 0.0;
  marker.pose.orientation.w = 1.0;

  marker.scale.x = (max[0]-min[0]);
  marker.scale.y = (max[1]-min[1]);
  marker.scale.z = (max[2]-min[2]);

  if (marker.scale.x ==0)
    marker.scale.x=0.1;

  if (marker.scale.y ==0)
    marker.scale.y=0.1;

  if (marker.scale.z ==0)
    marker.scale.z=0.1;

  marker.color.r = r;
  marker.color.g = g;
  marker.color.b = b;
  marker.color.a = 0.5;

  marker.lifetime = ros::Duration();
  //   marker.lifetime = ros::Duration(0.5);
  cluster_marker.publish(marker);
  return;
}


//-------------------------------------------------------------------------------------------------
//MOV
typedef pcl::PointCloud<pcl::PointXYZ> Pointi;
void callback(const Pointi::ConstPtr& msg)

{

int j;
```

```
  j=0;

  //printf ("Cloud: width = %d, height = %d\n", Point_cloud->width, Point_cloud-
>height);
  BOOST_FOREACH (const pcl::PointXYZ& pt, msg->points)  //it works like a loop
which assignes the values 1 by 1 to a variable from an array

{

    if( !pcl_isfinite( pt.x ) || !pcl_isfinite( pt.y ) || !pcl_isfinite( pt.z ) )  // this is done so
that the nan values from the pointcloud are not processed
        continue;

  if (j==0)
  {
          min_dist[0]=pt.x;
          min_dist[1]=pt.y;
          min_dist[2]=pt.z;
          //printf("First run");
  }

  else if (pt.z<min_dist[2])// && pt.z>0)
  {
       min_dist[0]=pt.x;
          min_dist[1]=pt.y;
          min_dist[2]=pt.z;
          //printf("Koushik\n");
  }

  j++;
//printf ("\t(%f, %f, %f)\n", pt.x, pt.y, pt.z);

}

printf("Minimim distancs point :: x = %f \t , y = %f \t , z = %f \n",
min_dist[0],min_dist[1],min_dist[2]);

if(min_dist[0] > 0 && min_dist[2]<=0.65)
{
          printf("Turn Left\n");
          cmdvel.angular.z=0.30;
          cmdvel.linear.x=0;
}
else if(min_dist[0] < 0 && min_dist[2]<=0.65)
{
          printf("Turn Right\n");
```

```
        cmdvel.angular.z=-0.30;
        cmdvel.linear.x=0;
}
else
{
        printf("Go Straight\n");
        cmdvel.linear.x=0.20;
        cmdvel.angular.z=0;
}
        vel_pub.publish(cmdvel);
}


//-------------------------------------------------------------------------------------------------
bool  evaluateCluster(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster)
{

  // Tranform cloud to base_link
  pcl::PointCloud<pcl::PointXYZ>::Ptr transformed_cloud (new
pcl::PointCloud<pcl::PointXYZ>);
  (*cloud_cluster).header.frame_id = "camera_depth_optical_frame";
  //tf_listener-
>waitForTransform("openni_depth_optical_frame","base_link",ros::Time::now(),ros::D
uration(3.0));
  pcl_ros::transformPointCloud("base_link",*cloud_cluster,*transformed_cloud,
*tf_listener);
 int tom=0;

 // transformed_cloud = cloud_cluster;
  Eigen::Vector4f centroid;
  Eigen::Vector4f min;
  Eigen::Vector4f max;

  pcl::compute3DCentroid (*transformed_cloud, centroid);
  pcl::getMinMax3D (*transformed_cloud, min, max);


   // All human beings must be shorter than 1,90m and taller than 1,40m. Human beings
cannot be wider than 1m
  if(((max[2]-min[2]>MIN_PERSON_HEIGHT)&&(max[2]-
min[2]<MAX_PERSON_HEIGHT)) ||
        ((max[1]-min[1]<MAX_PERSON_WIDTH)||(max[0]-
min[0]<MAX_PERSON_WIDTH)) ||
        ((max[1]-min[1]>MIN_PERSON_WIDTH)||(max[0]-
min[0]>MIN_PERSON_WIDTH)))
        {
```

```
                        ROS_INFO("Person Location (x,y,z) %f %f
%f\n",centroid[0],centroid[1],centroid[2]);

                        geometry_msgs::PointStamped cluster_centroid;
                        cluster_centroid.header.frame_id = "base_link";
                        cluster_centroid.header.stamp = ros::Time::now();
                        cluster_centroid.point.x = centroid[0];
                        cluster_centroid.point.y = centroid[1];
                        cluster_centroid.point.z = centroid[2];
                        detections_pub.publish(cluster_centroid);
while (tom<5)


//ros::Rate loop_rate(25); // 5Hz

 // while (ros::ok())
  {

//{
                        if ((min_dist[2]>=0.70) && ((DIST_MIN_Y < centroid[1]) &&
(centroid[1] < DIST_MAX_Y)))
                {
                                printf("Go Straight tom\n");
                                cmdvel.linear.x=0.40;
                                cmdvel.angular.z=0;
                }

                else if ((min_dist[2]>=0.70) && (DIST_MIN_Y > centroid[1]))
                {
                                printf("Turn Right tom\n");
                                cmdvel.angular.z=-0.2617; //15º
                                cmdvel.linear.x=0;
                }
                else if ((min_dist[2]>=0.70) && (centroid[1] > DIST_MAX_Y))
                {
                                printf("Turn Left tom\n");
                                cmdvel.angular.z=0.2617;//15º
                                cmdvel.linear.x=0;

                }

                        if ((min_dist[2]<0.70)&& ((DIST_MIN_Y < centroid[1]) &&
(centroid[1] < DIST_MAX_Y)))
                {
                                cmdvel.angular.z=0;
                                cmdvel.linear.x=0;
```

```
        //vel_pub.publish(geometry_msgs::Twist());
                        system("echo hola  | festival --tts");
                        ros::Duration(5.0).sleep();

                                printf("Turn back\n");
                                cmdvel.angular.z=3.14;// 180º
                                cmdvel.linear.x=0;
                }

 tom ++;
vel_pub.publish(cmdvel);
}
//loop_rate.sleep();

}

//printf("Mostrando TOM:%d\n",tom);

//vel_pub.publish(cmdvel);
  return true;
}

//---------------------------------------------------------------------------------------------------

// Get Ground Plane Coefficients from Point Cloud using Sample Consensus
pcl::ModelCoefficients::Ptr getPlaneCoefficients(pcl::PointCloud<pcl::PointXYZ>::Ptr
cloud)
{

  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_temp (new
pcl::PointCloud<pcl::PointXYZ>);
  cloud_temp = cloud;

  pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
  pcl::PointIndices::Ptr inliers (new pcl::PointIndices);;
  // Create the segmentation object
  pcl::SACSegmentation<pcl::PointXYZ> seg;
  // Optional
  seg.setOptimizeCoefficients (true);
  // Mandatory
  seg.setModelType (pcl::SACMODEL_PLANE);
  seg.setMethodType (pcl::SAC_RANSAC);
  seg.setDistanceThreshold (0.3); //0.01
  seg.setInputCloud (cloud_temp->makeShared ());
  seg.segment (*inliers, *coefficients);
```

```
  // Publish the model
coefficpwd/opt/ros/groovy/stacks/turtlebot_apps/turtlebot_teleop/srcients
  pub_plane.publish (*coefficients);
  //std::cerr << "Model coefficients: " << coefficients->values[0] << " "
   // << coefficients->values[1] << " "
  //  << coefficients->values[2] << " "
   // << coefficients->values[3] << std::endl;
  return coefficients;

}
//-----------------------------------------------------------------------------------------------------
// Remove Ground Plane from Point Cloud
pcl::PointCloud<pcl::PointXYZ>::Ptr
removeGroundPlane(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud,
pcl::ModelCoefficients::Ptr coefficients)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr ground_removed_pcd (new
pcl::PointCloud<pcl::PointXYZ>);

  pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
  // Create the segmentation object
  pcl::SACSegmentation<pcl::PointXYZ> seg;
  // Optional
  seg.setOptimizeCoefficients (false);
  // Mandatory
  seg.setModelType (pcl::SACMODEL_PLANE);
  // seg.setModelType (pcl::SACMODEL_PARALLEL_PLANE);
  //seg.setAxis (Eigen::Vector3f (0.0, 0.0, 1.0));
  //seg.setEpsAngle (15*3.14/180);
  seg.setMethodType (pcl::SAC_RANSAC);
  seg.setDistanceThreshold (0.1); //0.01
  seg.setInputCloud (cloud->makeShared ());
  seg.segment (*inliers, *coefficients);
 // std::cerr << "Model coefficients: " << coefficients->values[0] << " "
   // << coefficients->values[1] << " "
   // << coefficients->values[2] << " "
   // << coefficients->values[3] << std::endl;

  // Remove the planar inliers from the input cloud
  pcl::ExtractIndices<pcl::PointXYZ> extract;
  extract.setInputCloud (cloud);
  extract.setIndices (inliers);
  extract.setNegative (true);
  extract.filter(*ground_removed_pcd);
  return ground_removed_pcd;
}
```

```
//-------------------------------------------------------------------------------------------------------
// Euclidean Clustering
pcl::PointCloud<pcl::PointXYZ>::Ptr
doEuclideanClustering(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud)
{
  pcl::PointCloud<pcl::PointXYZ>::Ptr cluster_pcd (new
pcl::PointCloud<pcl::PointXYZ>);

  // Creating the KdTree object for the search method of the extraction
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new
pcl::search::KdTree<pcl::PointXYZ>());
  tree->setInputCloud (cloud);

  std::vector<pcl::PointIndices> cluster_indices;
  pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
  ec.setClusterTolerance (PERSON_CLUSTER_TOLERANCE); // 2cm
  ec.setMinClusterSize (MIN_PERSON_CLUSTER_SIZE); //100
  ec.setMaxClusterSize (MAX_PERSON_CLUSTER_SIZE);
  ec.setSearchMethod (tree);
  ec.setInputCloud (cloud);
  ec.extract (cluster_indices);

  int j = 0;
  bool human_flag = true;
  for (std::vector<pcl::PointIndices>::const_iterator it = cluster_indices.begin (); it !=
cluster_indices.end (); ++it)
  {
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster (new
pcl::PointCloud<pcl::PointXYZ>);
    for (std::vector<int>::const_iterator pit = it->indices.begin (); pit != it->indices.end ();
pit++)
      cloud_cluster->points.push_back (cloud->points[*pit]); //*

    cloud_cluster->width = cloud_cluster->points.size ();
    cloud_cluster->height = 1;
    cloud_cluster->is_dense = true;
    cloud_cluster->sensor_origin_ = cloud->sensor_origin_;
    cloud_cluster->sensor_orientation_ = cloud->sensor_orientation_;
    //printf("Size: %d\n",(int)cloud_cluster->points.size ());


    // Publish the cluster marker
      human_flag = evaluateCluster(cloud_cluster);
    if(human_flag == true){

      float r = 1, g = 0, b = 0;
```

```
    std::string ns = "base_link";
    publishClusterMarker(cloud_cluster,ns,1,r,g,b);

    // Publish the data
      sensor_msgs::PointCloud2 output_cloud;
      pcl::toROSMsg(*cloud_cluster,output_cloud);
      output_cloud.header.frame_id = "camera_depth_optical_frame";
      pub_cluster.publish (output_cloud);
      // return cloud_cluster;
    }
    j++;

  }
  return cluster_pcd;

}

//-------------------------------------------------------------------------------------------------
// Callback for ROS Subscriber
  void cloud_cb (const sensor_msgs::PointCloud2ConstPtr& cloud)
{

  // Input PCD Point Cloud
  pcl::PointCloud<pcl::PointXYZ>::Ptr input_cloud(new
pcl::PointCloud<pcl::PointXYZ>);
  pcl::fromROSMsg (*cloud, *input_cloud);

  // Output Sensor MSG point cloud
  sensor_msgs::PointCloud2 output_cloud;

  // Processing
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new
pcl::PointCloud<pcl::PointXYZ>);

  // One time processing
  static int i = 1;
  if(i==1)
  {
    plane_coefficients = getPlaneCoefficients(input_cloud);
    i = 0;
  }



  // Perform the actual filtering
  pcl::VoxelGrid<pcl::PointXYZ> sor;
```

```
  sor.setInputCloud (input_cloud);
  sor.setLeafSize (LEAF_SIZE,LEAF_SIZE,LEAF_SIZE);
  sor.filter (*cloud_filtered);

  // Plane Segmentation
  cloud_filtered = removeGroundPlane(cloud_filtered,plane_coefficients);

 //Euclidean Clustering
  pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster (new
pcl::PointCloud<pcl::PointXYZ>);
  cloud_cluster = doEuclideanClustering(cloud_filtered);

  // Publish the data
  pcl::toROSMsg(*cloud_filtered,output_cloud);
  pub.publish (output_cloud);
}
//------------------------------------------------------------------------------------------------
  int
main (int argc, char** argv)
{
  // Initialize ROS
  ros::init (argc, argv, "pcdtest");
  ros::NodeHandle nh,nh1;

  tf_listener = new tf::TransformListener();


  cmdvel.linear.x=0;
  cmdvel.linear.y=0;
  cmdvel.linear.z=0;
  cmdvel.angular.x=0;
  cmdvel.angular.y=0;
  cmdvel.angular.z=0;


  // Create a ROS subscriber for the input point cloud


ros::Subscriber subA = nh.subscribe ("/camera/depth_registered/points", 1, cloud_cb);
ros::Subscriber subB = nh.subscribe<Pointi>("/camera/depth/points", 1, callback);

  // Create a ROS publisher for the output point cloud
  pub = nh.advertise<sensor_msgs::PointCloud2> ("base_link/filtered_cloud", 1);
  // Create a ROS publisher for the detections
 detections_pub =
nh.advertise<geometry_msgs::PointStamped>("kinect_tracker/detections",1);
```

```
  // Create a ROS publisher for the output segmented plane
 pub_plane = nh.advertise<pcl::ModelCoefficients> ("base_link/plane", 1);
  // Create a ROS publisher for the clusters
          pub_cluster = nh.advertise<sensor_msgs::PointCloud2>
("base_link/cluster_cloud", 1);
  // Create a ROS publisher for the cluster markers
  cluster_marker = nh.advertise<visualization_msgs::Marker>
("base_link/cluster_marker", 1);

  vel_pub = nh1.advertise<geometry_msgs::Twist>("cmd_vel",1);
 //vel_pub = nh.advertise<geometry_msgs::Twist>("cmd_vel_mux/input/teleop",1);



  // Spin
 ros::spin ();

//Modo 2

/*ros::Rate loop_rate(50); // 5Hz

  while (ros::ok())
  {


loop_rate.sleep();
 ros::spinOnce ();
          //printf(">>==============================================
=====<<\n");
  }
*/

}
```